Journal of Logical and Algebraic Methods in Programming 125 (2022) 100733

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

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

Shukun Tokas [*], Olaf Owe, Toktam Ramezanifarkhani

Department of Informatics, University of Oslo, Norway

A R T I C L E   I N F O

Article history:
Received 30 April 2019
Received in revised form 24 May 2021
Accepted 27 October 2021
Available online 4 November 2021

Keywords:
Privacy
GDPR
Privacy compliance
Data protection by design
Static analysis
Language-based privacy

A B S T R A C T

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, 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 privacy compliance, as well as a high-level modeling 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.

Contents

[*] Corresponding author.
E-mail address: shukun.tokas@sintef.no (S. Tokas).

https://doi.org/10.1016/j.jlamp.2021.100733
2352-2208/© 2021 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

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 [17]. 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) [19]. The GDPR is said to be *"The single most important change in data privacy regulation in 20 years"* [39].

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* [14] 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 [41] for analyzing a program's compliance with respect to the stated privacy policies. We will focus on privacy aspects and policies that can be checked statically. 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. Static techniques range from manual or semi-automatic deductive methods to automatic checking. We will only consider analysis methods that are fully automatic because they have a greater potential for practical usage.

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 modeling language centered around asynchronous and synchronous method interaction, extended with policy specifications. The static checking of privacy policies is modular and is performed on classes that are type-correct with respect to ordinary typing. In particular, the checking can be done class-wise in the sense that one may check each class independently (assuming access to inherited code), and a class that has passed the check need not be checked again, when other classes and subclasses are added. 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. In GDPR, *sensitive data* is a special category of *personal data* that needs more protection, and in our language setting both sensitive and personal data use the same policy specifications (with stricter policies on sensitive data). For our purposes 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.

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 privacy 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 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 paradigm [12,31,33,42], based on the actor model [28]. This paradigm is considered to be one of the most promising candidates to model asynchronously parallel and distributed computations in a safe manner [12]. 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 and non-modular.

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 a modeling language for loosely coupled distributed systems.
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.

The two first contributions are independent of the choice of language, while the last three are not. 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 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 4 introduces this language. Section 5 presents the static compliance checking by means of a type and effect system, and demonstrates the analysis on the case study. Section 6 briefly discusses an extension to deal with consent and self access to personal data about a data subject. Section 7 presents an operational semantics and proves soundness and progress. Section 8 discusses related work, and Section 9 concludes the paper. An algorithmic version of the static compliance checking is shown in Appendix A and notational conventions used in the paper are listed in Appendix B.

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. Fig. 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.

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 3.

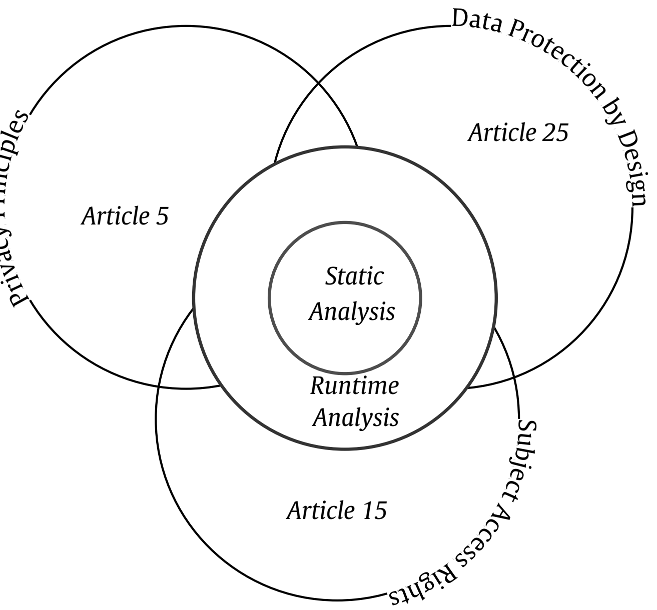

Fig. 1. Research Focus.

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 [19].

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 studied in [51]).

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.

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 behavior of objects, declared through interfaces. We extend this specification of observable behavior 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

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

Fig. 2. BNF syntax definition of the policy language. I ranges over interface names and R over purpose names.

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 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.

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 $(\mathit{Doctor}, \mathit{treatm}, \mathit{rincr})$ 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 real word individual or entity such as a person, authority, company, or organization, or a role representing a set of such entities. At the programming level, an individual corresponds to an object representing that individual or an interface representing all objects supporting that interface. An interface I is used to restrict the access to information, by requiring that the accessing object supports I. (We say that 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* or by object expressions. The interface *Subject* corresponds to a "data subject", extending *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, $\mathit{Specialist} < \mathit{Doctor} < \mathit{HealthWorker}$. 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* [26]. 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 (Fig. 4). 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, $\mathit{treatm}, \mathit{diagnosis}, \mathit{research} < \mathit{health}$, and $\mathit{monitoring} < \mathit{treatm}$. 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

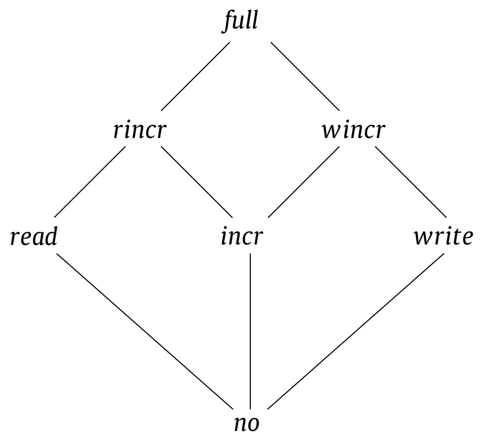

Fig. 3. The lattice for general access rights (without *self*). Note that *rincr* is the same as $read \sqcup incr$, *wincr* is the same as $write \sqcup incr$, and *full* is the same as $read \sqcup write$.

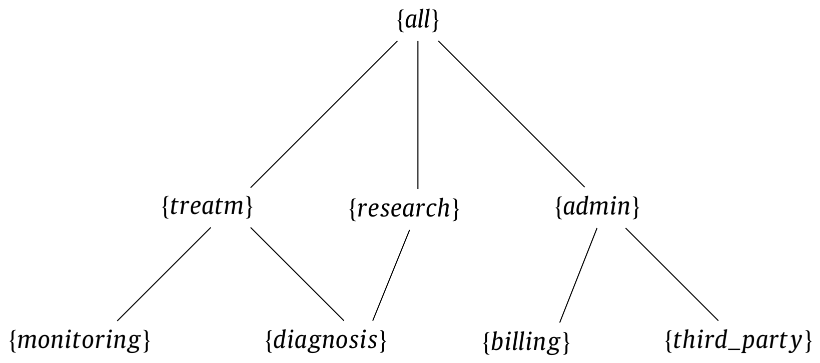

Fig. 4. Sample purpose hierarchy.

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 Fig. 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 Fig. 3. In subsection 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 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 [19]). 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.

3.2. Access rights for data subjects

Under Article 15 [19], 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 subject. One may then express general access rights in combination with access rights on self data. The policy triple

$$(Subject, all, self \sqcap read)$$

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 this principle. With the addition of *self*, we need to refine and revise our definition of access rights and their placement in a lattice. Mathematically, our lattice can be defined by a pair lattice as follows:

Definition 1 *(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 Fig. 3). We define

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

It follows that the redefined $\sqsubseteq_A$ is a partial order, and the carrier set has 22 elements. The element (a, b) is written $a \sqcup (self \sqcap b)$, and the access right $a \sqcup (self \sqcap a)$ is abbreviated a. The access right $a \sqcup (self \sqcap b)$ 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, $no \sqcup (self \sqcap full)$ is the same as *self*, meaning full access to self data, but no access to data about others. Furthermore, $self \sqcap rincr$ gives a principal the right to read self data and add new information about herself. In contrast, $read \sqcup self$ means full access to self data and read access to other data, and $read \sqcup (self \sqcap rincr)$ 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 6.

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 Fig. 2 and some sample policies are found in Fig. 5. Optional parts are written in brackets (as in [...]), while superscripts * and $^+$ denote repetition and non-empty repetition, respectively.

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

policy $\mathcal{P}_{Doc} = (Doctor, treatm, full)$
policy $\mathcal{P}_{DocTask} = (Any, treatm, full)$
policy $\mathcal{P}_{AddPresc} = (Doctor, treatm, rincr)$
policy $\mathcal{P}_{GetPresc} = (Nurse, treatm, read)$
policy $\mathcal{P}_{GetSelfPresc} = (Nurse, health, read)$
policy $\mathcal{P}_{PatientPresc} = (Patient, treatm, read)$
policy $\mathcal{P}_{Start} = (Any, treatm, no)$

policy $\mathcal{P}_{Presc} = \{\mathcal{P}_{GetPresc}, \mathcal{P}_{AddPresc}, \mathcal{P}_{Doc}\}$

type Presc = Patient * String :: $\underline{\mathcal{P}_{Presc}}$

interface Patient extends Subject {Void getSelfData() :: $\underline{\mathcal{P}_{Start}}$}
interface AddPresc {Void makePresc(Presc newp):: $\underline{\mathcal{P}_{AddPresc}}$}
interface GetPresc {Presc getPresc(Patient p) :: $\underline{\mathcal{P}_{GetPresc}}$}
interface PatientData extends AddPresc, GetPresc {}
interface Nurse extends Principal {
    Presc nurseTask() :: $\underline{\mathcal{P}_{GetPresc}}$
  with Patient
    Presc getMyPresc() :: $\underline{\mathcal{P}_{PatientPresc}}$ }
interface Doctor extends Nurse{
  Void doctorTask(Patient p) :: $\underline{\mathcal{P}_{DocTask}}$ }
```

Fig. 5. Interface, type and policy definitions for the Prescription Example. Grey policy specifications are implicit while underlined ones need to be explicitly stated.

Definition 2 *(Policy compliance).* The sub-policy relation *less*, expressing *policy compliance*, is defined by

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

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*, 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 3 *(Compliance of policy sets).*

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

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 4 *(Join and meet over policy sets).*

$$S \sqcup S' \triangleq closure(S \cup S')$$
$$S \sqcap S' \triangleq closure(\{P \mid P \sqsubseteq S \wedge P \sqsubseteq S'\})$$

where *the closure operation* is defined by

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

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 $\{(Doctor, treatm, full)\}$.

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 5 *(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 $\{\bullet\}$.

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.)

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 4 by means of an imperative-style language for active objects. An example with policies and interfaces is given in Fig. 5.

Definition 6 *(Interface syntax).* An *interface* is declared with the BNF syntax

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

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 [44]. 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, unless a new policy is specified (by renewing its signature).

Definition 7 *(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 8 *(Type declaration syntax).*

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

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

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

A list type $List[T]$ is sensitive if T is sensitive, and has the policy of T, i.e., $\mathcal{P}_{List[T]} \triangleq \mathcal{P}_T$. 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

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

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 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 Fig. 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.

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, while fields in objects are mutable and can be updated by the methods defined in the class of the object. 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 and fields. 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. Fig. 7 defines classes corresponding to the interfaces in Fig. 5, using the imperative language.

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 mechanisms [12]. Although intended for system modeling, the language is executable and has an interpreter written in Maude [15], and an extended language based on the same concurrency model has a compiler to Erlang [1,53]. The active object paradigm is based on concurrent autonomous 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 [31,45], 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 [32]. We formalize the analysis outlined in Section 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 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 Fig. 6. 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

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

Fig. 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.

```
class PATIENTDATA() implements PatientData {
  type PData = List[Presc] :: $\mathcal{P}_{Presc}$
  PData pd = empty();

  Presc getPresc(Patient p){return last(pd/p)} :: $\mathcal{P}_{GetPresc}$
  Void makePresc(Presc newp) {
     if newp ≠emptyString() then pd:+ newp fi } :: $\mathcal{P}_{AddPresc}$
  }

class NURSE(PatientData pdb) implements Nurse{
  Presc nurseTask(Patient p){ return pdb.getPresc(p)} :: $\mathcal{P}_{GetPresc}$
  with Patient
    Presc getMyPresc() {return pdb.getPresc(caller)} :: $\mathcal{P}_{PatientPresc}$
  }

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)}:: $\mathcal{P}_{DocTask}$
  }

class PATIENT(String id, Doctor d, Nurse n) implements Patient{
  Void getSelfData(){ n!getMyPresc() } :: $\mathcal{P}_{Start}$
  }

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() } :: $\underline{\mathcal{P}_{Start}}$ // class constructor
  }
```

Fig. 7. Doctor and Nurse classes accessing patient prescriptions.

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 Fig. 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 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*!$m(\ldots)$ 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 [31]. This would be orthogonal to the treatment of privacy policies.

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 $List[T] = empty() + append(List[T] * T)$, meaning that lists have the form $empty()$ or $append(l, x)$, where l is a list and x a value of type T. A pair product type can be defined by $PatientInfo = (Patient * Nat)$ 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 $List[PatientInfo]$. We may define a projection operator $(proj : List[PatientInfo] * Patient \rightarrow List[PatientInfo])$ by $proj(empty(), p) = empty()$ and $proj(append(l, (p, d)), p') = \texttt{if}\ p = p'\ \texttt{then}\ append(proj(l, p'), (p, d))\ \texttt{else}\ proj(l, p')$. Using the infix notion / for $proj$ and $+$ for list append, this gives the definition $empty()/v = empty()$ and $(l + (v1, v2))/v = \texttt{if}\ v = v1\ \texttt{then}\ (l/v) + (v1, v2)\ \texttt{else}\ l/v$. 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 accommodate encryption functions.) An application of a constructor of a sensitive type and with an argument which is either sensitive or of interface Subject, may create new sensitive information. This requires write access (as checked by the type rules for expressions).

4.2. An example

An example program is given in Fig. 7, showing class implementations of the interfaces given in Fig. 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 are shown in Section 5.1.

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 [41]. 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 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 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 $\Gamma[v]$. Our 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 [46]. 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 $\texttt{if}\ h\ \texttt{then}\ l := true\ \texttt{else}\ l := false\ \texttt{fi}$, 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 (list) 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'. We use a syntax similar to Hoare triples, letting Γ and Γ' express knowledge of the pre- and post-state, but in contrast to Hoare logic the rules can be applied mechanically. 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

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

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

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

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

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

Fig. 8. Policy Rules for Classes and Methods. (Note: read-only access for $\overline{z}$ and $\overline{y}$.)

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

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

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

$$\vdash Cl\ \mathbf{ok}$$

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

$$\vdash In\ \mathbf{ok}$$

expresses that any re-defined policy of the method in In must comply with that of the superinterface. The interface and class judgments are analyzed in the context of earlier type and interface definitions (when needed). In the analysis of a class, inherited code must be included, but no other class definitions need to be known. Thus class definitions are analyzed independently. An algorithmic version of the analysis is provided in appendix A, showing that the analysis is terminating and with a unique result.

The typing rules for interfaces, classes, and methods are given in Fig. 8, Fig. 9 defines the typing rules for expressions and right-hand sides, and Fig. 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. Here, $\overline{M}$ ranges over methods declared or inherited (possibly with renewed signatures). 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 environment 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

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

$$(\text{P-CONSTANT})\ \dfrac{}{C,m \vdash [\Gamma]\ \ \mathit{const}() :: \Gamma[pc]}$$

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

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

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

$$(\text{P-NEW})\ \dfrac{C,m \vdash [\Gamma]\ \ e_i :: \mathcal{P}_i \qquad \mathcal{P}_i \Longrightarrow \Gamma_{C'}[z_i]}{C,m \vdash [\Gamma]\ \ \textbf{new}\ C'(\overline{e}) :: \Gamma[pc]}$$

Fig. 9. Policy Rules for Expressions and Right-Hand Sides.

$$(\text{P-SKIP})\ \dfrac{}{C,m \vdash [\Gamma]\ \mathit{skip}\ [\Gamma]}$$

$$(\text{P-COMPOSITION})\ \dfrac{C,m \vdash [\Gamma]\ s_1\ [\Gamma_1] \quad C,m \vdash [\Gamma_1]\ s_2\ [\Gamma_2]}{C,m \vdash [\Gamma]\ s_1; s_2\ [\Gamma_2]}$$

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

$$(\text{P-LOCAL-WRITE})\ \dfrac{C,m \vdash [\Gamma]\ \ \mathit{rhs} :: \mathcal{P}}{C,m \vdash [\Gamma]\ x := \mathit{rhs}\ [\Gamma[x \mapsto \mathcal{P}]]}$$

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

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

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

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

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

Fig. 10. Policy Rules for Statements. In the last rule n is the least i such that $\Gamma_{i+1} = \Gamma_i$.

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 the *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 9 *(Method body policy set).* The policy set of the *body of a method* m in class C is defined by

$$\begin{array}{lcll} (I,R,A)@(C,m) & \triangleq & \{(I,R,A)\} & \text{if } I \leq \mathit{Principal} \\ & & \cup_i \{(I_i,R,A)\} & \text{otherwise} \end{array}$$

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 Fig. 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 $f_T(\overline{e})$ 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., $(I',R',A') \sqsubseteq_{Co,R} (I,R,A) \triangleq I \leq I' \wedge R' \leq R$. 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 Fig. 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 straightforward, 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.

5.1. Static compliance checking of the example

In Fig. 7, which is a continuation of the example in Fig. 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 Fig. 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 $[\Gamma]\quad e :: \mathcal{P}'$
$[\Gamma]\quad d :: \{\bullet\}\quad //object\ references\ are\ not\ sensitive$

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

$\mathcal{P}_i \Longrightarrow \mathcal{P}_{par(I,n)_i}$
$\{\bullet\} \Longrightarrow \mathcal{P}_{par(Doctor,doctorTask)}$
$\{\bullet\} \Longrightarrow \mathcal{P}_p$
$\{\bullet\} \Longrightarrow \{\bullet\}$ // 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 $[\Gamma]\ \ rhs :: \mathcal{P}$ // 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 $[\Gamma]\ \ e_i :: \mathcal{P}_i$
$[\Gamma]\ \ p :: \{\bullet\}$

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

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

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 $[\Gamma]\ \ p :: \Gamma[p] \sqcap \Gamma[pc] \Leftrightarrow$
$[\Gamma]\ \ p :: \{\bullet\}$, since pc is non-sensitive here.

These premises ensure that the variables in the constructor function have 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 } :: $\mathcal{P}_{AddPresc}$
  }
```

if e **then** s_1 **else** s_2 **fi**

4.1 $[\Gamma]$ $e :: \mathcal{P}$ // P-Var
$[\Gamma]$ $newp \neq emptyString() :: \mathcal{P}$

$read \sqsubseteq \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C, m))$ // 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.
$read \sqsubseteq_A \{(Nurse, treatm, read), (Doctor, treatm, full)\} \sqcap (Doctor, treatm, rincr)$
$read \sqsubseteq_A (Doctor, treatm, rincr)$

$[\Gamma]$ $newp :: \Gamma[newp] \sqcap \Gamma[pc]$
$[\Gamma]$ $newp :: \mathcal{P}_{Presc} \sqcap \mathcal{P}_{Presc}$
$[\Gamma]$ $newp :: \mathcal{P}_{Presc}$

$[\Gamma]$ $emptyString() :: \mathcal{P}$ // P-Constant
$[\Gamma]$ $emptyString() :: \Gamma[pc]$
$[\Gamma]$ $emptyString() :: \{\bullet\}$, since pc is non-sensitive here.

$[\Gamma]$ $newp \neq emptyString() :: \mathcal{P}_{Presc}$

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 $[\Gamma]$ $rhs :: \mathcal{P}$
$[\Gamma]$ $newp :: \mathcal{P}_{Presc}$ // 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 6.

```
policy $\mathcal{P}_{Doc} = (Doctor, treatm(Patient), full)$
policy $\mathcal{P}_{GetPresc} = (Nurse, treatm(Patient), read)$
policy $\mathcal{P}_{Presc} = \{\mathcal{P}_{GetPresc}, \mathcal{P}_{Doc}\}$

class PATIENTDATA() implements PatientData {
  type PData = List[Presc] :: $\mathcal{P}_{Presc}$
  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
  ... }
```

Fig. 11. Example with subject awareness. As before, gray parts are implicit.

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 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 Fig. 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 Fig. 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

$$Sensitive!upd\_consent(np)$$

which will go to all *Sensitive* objects and lead to adjustments of all sensitive data in the system where subject is *caller*.

$$\begin{array}{lll} v & ::= \ldots \mid pcs \mid nextFut & \text{added variables} \\ s & ::= \ldots \mid v := \mathtt{get}\ u & \text{added statement} \end{array}$$

Fig. 12. BNF syntax of additional constructs used in the operation semantics.

$$\begin{array}{lll}
\text{assign :} & & o : \mathbf{ob}(\delta, v := e; \overline{s}) \\
& \rightarrow & o : \mathbf{ob}(\delta[v := e], \overline{s}) \\[1ex]
\text{if-true :} & & o : \mathbf{ob}(\delta, \mathtt{if}\ b\ \mathtt{then}\ \overline{s1}\ \mathtt{else}\ \overline{s2}\ \mathtt{fi}; \overline{s}) \\
& \rightarrow & o : \mathbf{ob}(\delta[pcs := push(pcs, l)], \overline{s1}; pcs := pop(pcs); \overline{s}) \\
& & \mathtt{if}\ \delta[b] = true_l \\[1ex]
\text{if-false :} & & o : \mathbf{ob}(\delta, \mathtt{if}\ b\ \mathtt{then}\ \overline{s1}\ \mathtt{else}\ \overline{s2}\ \mathtt{fi}; \overline{s}) \\
& \rightarrow & o : \mathbf{ob}(\delta[pcs := push(pcs, l)], \overline{s2}; pcs := pop(pcs); \overline{s}) \\
& & \mathtt{if}\ \delta[b] = false_l \\[1ex]
\text{while :} & & o : \mathbf{ob}(\delta, \mathtt{while}\ b\ \mathtt{do}\ \overline{s1}\ \mathtt{od}; \overline{s}) \\
& \rightarrow & o : \mathbf{ob}(\delta, \mathtt{if}\ b\ \mathtt{then}\ \overline{s1};\ \mathtt{while}\ b\ \mathtt{do}\ \overline{s1}\ \mathtt{od}\ \mathtt{fi}; \overline{s}) \\[1ex]
\text{new :} & & o : \mathbf{ob}(\delta, v := \mathtt{new}\ C(\overline{e}); \overline{s}) \\
& \rightarrow & o : \mathbf{ob}(\delta[v := o'], \overline{s}) \\
& & o' : \mathbf{ob}(\delta_C[this \mapsto o', \overline{z} \mapsto \delta[\overline{e}]], init_C) \\
& & \mathtt{where}\ \ o' = (fresh, C),\ \text{with } fresh \text{ a fresh reference relative to } C \\[1ex]
\text{async. call :} & & o : \mathbf{ob}(\delta, a!m(\overline{e}); \overline{s}) \\
& \rightarrow & o : \mathbf{ob}(\delta[nextFut := next(nextFut)], \overline{s}) \\
& & \mathtt{msg}\ o \rightarrow \delta[a].m(\delta[nextFut, \overline{e}]) \\[1ex]
\text{sync. call :} & & o : \mathbf{ob}(\delta, v := a.m(\overline{e}); \overline{s}) \\
& \rightarrow & o : \mathbf{ob}(\delta, a!m(\overline{e});\ v := \mathtt{get}\ \delta[nextFut]; \overline{s}) \\[1ex]
\text{start :} & & \mathtt{msg}\ o' \rightarrow o.m(u, \overline{c}) \\
& & o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \\
& \rightarrow & o : \mathbf{ob}((\alpha|(\beta[caller \mapsto o', myfuture \mapsto u, \overline{y} \mapsto \overline{c}, pcs \mapsto empty()])), \overline{s}) \\
& & \mathtt{where}\ (m, \overline{y}, \beta, \overline{s}) \text{ is the body of } m \text{ in the class of } this \\[1ex]
\text{return :} & & o : \mathbf{ob}(\delta, \mathtt{return}\ e) \\
& \rightarrow & o : \mathbf{ob}(\delta, \mathbf{idle}) \\
& & \mathtt{msg}\ \delta[caller] \leftarrow \delta[this].(\delta[myfuture], \delta[e]) \\[1ex]
\text{query :} & & \mathtt{msg}\ o \leftarrow o'.(u, c) \\
& & o : \mathbf{ob}(\delta, v := \mathtt{get}\ u; \overline{s}) \\
& \rightarrow & o : \mathbf{ob}(\delta, v := c; \overline{s}) \\[1ex]
\text{no-query :} & & \mathtt{msg}\ o \leftarrow o'.(u, c) \\
& & o : \mathbf{ob}(\delta, \overline{s}) \\
& \rightarrow & o : \mathbf{ob}(\delta, \overline{s}) \\
& & \mathtt{if}\ \mathtt{get}\ u \notin \overline{s}
\end{array}$$

Fig. 13. Operational rules defining small-step semantics with policies.

7. Operational semantics

The operational semantics of the considered language is given in Fig. 13. 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 objects 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 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 straightforward 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 Fig. 12). By lifting these call labels and **get** statements to the language syntax, we would obtain support for object-local futures, as described in [33].

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

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

representing a call to m with o as caller, o' callee, and $\overline{e}$ actual parameters, or

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

representing a completion event where d is the returned value and u the identity of the call. In addition, $\mathbf{msg}\; o \rightarrow I.m(\overline{e})$ 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

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

saying that the execution of *skip* has no effect on the state δ of the object.

The semantics in Fig. 13 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 *nextFut* for generating unique identities for calls. These appear as fields in the operational semantics (*nextFut* initialized with some value and with a *next* function to generate new unique values). Furthermore, *this* is handled as an implicit class parameter, while *myfuture* 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 Fig. 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 *nextFut*) and class parameters $\overline{z}$ (including *this*), and β is the state of the local variables $\overline{x}$ and formal parameters $\overline{y}$ (including *myfuture* 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 *myfuture*. With Rule return, a return value is generated upon method termination and passed in a completion message together with the call identity stored in *myfuture*. 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 $\mathbf{msg}\; o \leftarrow o'.(u, d)$ 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 *nextFut* 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 are 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 [30] and it may be extended with cooperative scheduling and synchronization control as in [31].

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 1. 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 1.

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.

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 guaranteesthe 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 1 *(Soundness). Consider a given state $o : \mathbf{ob}(\delta, \overline{s})$ 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*

$$(C, m \vdash [\Gamma]\ \ v :: \mathcal{P}) \Rightarrow (\delta[v].tag \Longrightarrow \mathcal{P}) \tag{1}$$

$$(C, m \vdash [\Gamma]\ \ e :: \mathcal{P}) \Rightarrow (\delta[e].tag \Longrightarrow \mathcal{P}) \tag{2}$$

$$level(\delta[pcs]) \Longrightarrow \Gamma[pc] \tag{3}$$

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

$$level(\delta[pcs]) \Longrightarrow level(\Gamma[pcs]) \tag{4}$$

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 straightforward 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 2 *(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 $o : \mathbf{ob}(\delta, \overline{s}) \in \mathcal{C}'$ is* **idle** *($\overline{s}$ is* **idle***) and there is no invocation message* **`msg`** *$o' \rightarrow o.m(\ldots) \in \mathcal{C}'$, 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 [32].

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. The present work investigates static aspects of privacy policies and compliance; while the dynamic aspects including consent management are considered in [51].

Several attempts have been made to express privacy polices, through a language with formal syntax and semantics such as XACML [50], EPAL [6], APPEL [37], and XPref [3]. An analysis of these policy languages can also be found in [8,34]. Privacy restrictions are also expressed formally using ontologies [11] or dedicated logics such as [7,10,13]. However, a direct comparison of these policy languages and logics with our policy language is not straightforward, 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) [35] and role based access control (RBAC) [47], have been historically utilized in order to support security requirements [18]. In RBAC, permissions (to perform operations) are associated with a role or set of roles [47]. 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 [20], 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 [23], 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 [38,40,54].

Privacy by Design (PbD) has been discussed and promoted from several viewpoints such as formal approaches [36,48,52], privacy engineering [17,24,43], privacy-enhancing technologies (PET) [22,25], and privacy design patterns [16,29]. Tschantz and Wing, and Daniel Métayer, discuss the significance of formal methods for foundational formalizations of privacy related aspects [36,52]. In [52], 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 [48], 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 [29] 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 [26] 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. [9] 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 [9] propose to audit logs and detect if a system supports a purpose. In a continuation of this work, in [5] 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. [4] 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 [49], Sen et al. develop and demonstrate techniques for policy compliance checking in big data systems. Privacy policies are specified using a policy language *LEGALEASE*, restricting the information-flow based on (data) store, purpose, role, and other considerations. The data inventory tool *GROK* maps code-level schema elements to data types in *LEGALEASE*. Compliance checking then reduces to information flow analysis.

In [2], 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 [21], 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 [27] also applies to privacy, as pointed out by Schneider in [48]. 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.)

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 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 modeling language supporting active objects, augmented 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 healthcare case study, 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, so that a class may be checked independently without access to the code of other classes, apart from code inherited from superclasses. 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 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.

Future work. In the future we would like to extend the specification language 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 the operational semantics 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 the regulations and laws are extensive and may have conflicting expectations. Furthermore, identification of creator and information owner, 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, public interest, etc.). This can be further extended to accommodate other legal bases (including 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. In addition, we would like to work out a larger case study, using an implementation of the framework.

Declaration of competing interest

All authors have participated in (a) conception and design, or analysis and interpretation of the data; (b) drafting the article or revising it critically for important intellectual content; and (c) approval of the final version.

This manuscript has not been submitted to, nor is under review at, another journal or other publishing venue.

The authors have no affiliation with any organization with a direct or indirect financial interest in the subject matter discussed in the manuscript.

Acknowledgements

We thank the anonymous reviewers for extensive comments and substantial feedback, which have helped us to improve the manuscript significantly.

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 the project SCOTT (www.scott-project.eu) funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No. 737422.

Appendix A. Algorithmic version of the static analysis

In order to show how the type and effect system of Section 5 gives a terminating and fully automatic algorithm, resulting in accept or fail, we here provide a functional definition of the analysis, obtained from the rules. The functional definition has the advantage that it shows explicitly all cases of failure. The functional definition is formulated by a boolean function *ok* for checking classes, interfaces, and method declarations, as well as the two functions $E_{C,m}(\Gamma, s)$ defining the environment resulting from statement (list) s starting in environment Γ, and $P_{C,m}(\Gamma, s)$ defining the policy set of an expression/right-hand-side e evaluated in environment Γ.

We first define $E_{C,m}$ below, following closely the rules of Fig. 10:

$$
\begin{array}{lcl}
E_{C,m}(\Gamma, \mathit{skip}) & = & \Gamma \\
E_{C,m}(\Gamma, s_1; s_2) & = & E_{C,m}(E_{C,m}(\Gamma, s_1), s_2) \\
E_{C,m}(\Gamma, x := \mathit{rhs}) & = & \Gamma[x \mapsto P_{C,m}(\Gamma, \mathit{rhs})] \\
E_{C,m}(\Gamma, w := \mathit{rhs}) & = & \Gamma[w \mapsto P_{C,m}(\Gamma, \mathit{rhs})], \\
 & & \qquad \mathbf{if}\ \mathit{write} \sqsubseteq_A \Gamma_C[w] \sqcap (\mathcal{P}_{C,m}@(C,m)) \\
 & & \mathit{fail}, \qquad\qquad\qquad\qquad \mathbf{otherwise} \\
E_{C,m}(\Gamma, w :+ \mathit{rhs}) & = & \Gamma[w \mapsto P_{C,m}(\Gamma, w) \sqcup P_{C,m}(\Gamma, \mathit{rhs})], \\
 & & \qquad \mathbf{if}\ \mathit{incr} \sqsubseteq_A \Gamma_C[w] \sqcap (\mathcal{P}_{C,m}@(C,m)) \\
 & & \mathit{fail}, \qquad\qquad\qquad\qquad \mathbf{otherwise} \\
E_{C,m}(\Gamma, e!n_I(\overline{e})) & = & \Gamma, \qquad\qquad \mathbf{if}\ \mathit{isok}(P_{C,m}(e.n_I(\overline{e}))) \\
 & & \mathit{fail}, \qquad\qquad\qquad\qquad \mathbf{otherwise} \\
E_{C,m}(\Gamma, I!n(\overline{e})) & = & \Gamma, \quad \mathbf{if}\ P_{C,m}(\Gamma, e_i) \Longrightarrow \mathcal{P}_{par(I,n)_i}, \text{for each } i \\
 & & \qquad \mathbf{and}\ \mathcal{P}_{I,n} \sqsubseteq_{Co,R}\ \mathcal{P}_{C,m}@(C,m) \\
 & & \mathit{fail}, \qquad\qquad\qquad\qquad \mathbf{otherwise} \\
E_{C,m}(\Gamma, \mathbf{if}\ b\ \mathbf{then}\ s_1 & & \\
\qquad\qquad \mathbf{else}\ s_2\ \mathbf{fi}) & = & (\Gamma_1 \sqcup \Gamma_2)[pc \mapsto \Gamma[pc]],\ \mathbf{if}\ \mathit{isok}(P_{C,m}(b)) \\
 & & \mathit{fail}, \qquad\qquad\qquad\qquad \mathbf{otherwise} \\
 & & \mathbf{where}\ \Gamma_i = E_{C,m}(\Gamma', s_i) \\
 & & \qquad\qquad \Gamma' = \Gamma[pc \mapsto (\Gamma[pc] \sqcap P_{C,m}(\Gamma, b))] \\
E_{C,m}(\Gamma, \mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{od}) & = & \Gamma_n[pc \mapsto \Gamma_1[pc]] \\
 & & \mathbf{where}\ \Gamma_{i+1} = \Gamma_i \sqcap E_{C,m}(\Gamma'_i, s) \\
 & & \qquad\qquad \Gamma'_i = \Gamma_i[pc \mapsto \Gamma[pc] \sqcap P_{C,m}(\Gamma, b)]
\end{array}
$$

where n in the last equation is the smallest number such that $\Gamma_n + 1 = \Gamma_n$, and the boolean function *isok* takes a policy set or an environment and returns *false* if it is *fail*, and *true* otherwise. In order to explicitly capture failures, we add a special value $fail$. The equations are written in the style of Maude, using conditions and **otherwise** to cover all remaining cases [15]. The equations given here can be executed in Maude.

Similarly, the definition of the $P_{C,m}$ function follows directly from the rules in Fig. 9 (except that the two cases for *call* are incorporated in one equation):

$$\begin{array}{lcll}
P_{C,m}(\Gamma, v) & = & \Gamma[v] \sqcap \Gamma[pc], & \textbf{if } read \sqsubseteq_A \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C,m)) \\
 & & fail, & \textbf{otherwise} \\
P_{C,m}(\Gamma, const) & = & \Gamma[pc] & \\
P_{C,m}(\Gamma, f_T(\overline{e})) & = & \mathcal{P}_T \sqcap \Gamma[pc], \textbf{ if } isok(P_{C,m}(e_i)), \text{ for } e_i \text{ of sensitive type} & \\
 & & \textbf{and } write \sqsubseteq_A \mathcal{P}_T \sqcap (\mathcal{P}_{C,m}@(C,m)), \text{ if } f_T \text{ is sensitive} & \\
 & & fail, & \textbf{otherwise} \\
P_{C,m}(\Gamma, e.n_I(\overline{e})) & = & \mathcal{P}_{out(I,n)} \sqcap \Gamma[pc], \ \textbf{if } isok(P_{C,m}(\Gamma, e)) & \\
 & & \quad \textbf{and } P_{C,m}(\Gamma, e_i) \Longrightarrow \mathcal{P}_{par(I,n)_i}, \text{ for each } i & \\
 & & \quad \textbf{and } C \not\leq I \textbf{ implies } \mathcal{P}_{I,n} \sqsubseteq_{Co,R} \mathcal{P}_{C,m}@(C,m) & \\
 & & \quad \textbf{and } C \leq I \textbf{ implies } \mathcal{P}_{I,n} \sqsubseteq \mathcal{P}_{C,m}@(C,m) & \\
 & & fail, & \textbf{otherwise} \\
P_{C,m}(\Gamma, \textbf{new}\, C'(\overline{e})) & = & \Gamma[pc], & \textbf{if } P_{C,m}(\Gamma, e_i) \Longrightarrow \Gamma_{C'}[z_i] \\
 & & fail, & \textbf{otherwise}
\end{array}$$

In a conditional equation we use **if** to define the condition and **otherwise** to cover all other cases. We use **and** and **implies** to reflect the premises of the effect rules, letting here **implies** bind tighter than **and**. Thus, **if** C **and** C' **implies** C'' means **if** $(C$ **and** $(C'$ **implies** $C''))$.

With the use of *fail* and **otherwise**, it is clear that the functions are total and terminating since the boolean conditions used are all executable and terminating. It is also clear that the left-hand-sides are disjoint and cover all cases of the effect rules. Therefore the definitions of $E_{C,m}$ and $P_{C,m}$ are terminating with a unique result for each input.

Finally to check interface, class, and method definitions, we define the corresponding *ok* functions over these:

$$\begin{array}{rcl}
ok(\textbf{interface}\ I\ \textbf{extends}\ \overline{J} & & \\
\{\overline{D}\} :: \mathcal{P}) & = & \mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}\ , \text{ for each } J \in \overline{J}, m \in J \\[2ex]
ok(\textbf{class}\ \ C(\overline{Z}\ \overline{z}) & & \\
\textbf{implements}\ \overline{I}\ \{\overline{W}\ \overline{w};\ \overline{M}\}) & = & \mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{I,m}\, , \text{for each } I \in \overline{I}, m \in I \\
 & & \textbf{and } ok_C(M), \text{for each } M \in \overline{M} \\[2ex]
ok_C(T\ m(\overline{Y}\ \overline{y}) & & \\
\{\overline{X}\ \overline{x}; s; \textbf{return}\ \ rhs\} :: \mathcal{P}) & = & isok(\Gamma)\ \textbf{and}\ isok(\mathcal{P}')\ \textbf{and}\ \mathcal{P}' \Longrightarrow \mathcal{P}_T \\
 & & \textbf{and}\ \Gamma[w] \Longrightarrow \Gamma_C[w], \text{ for each field } w \\
 & & \textbf{where}\ \Gamma = E_{C,m}(\Gamma_m, s) \\
 & & \qquad \mathcal{P}' = P_{C,m}(\Gamma, rhs)
\end{array}$$

In the last equation Γ_m denotes the starting environment of the method as defined in the first premise of Rule P-METHOD. The *ok* checks for classes, interfaces, and methods follow directly from the rules in Fig. 8. It is clear that these functions are total and terminating. Since $E_{C,m}$ and $P_{C,m}$ are total and terminating, the compliance check of a class or interface gives a boolean result, *true* if it passes the check and *false* otherwise. The efficiency of the analysis is comparable to ordinary static type checking, but as we have seen, while-statements require a (terminating) fix-point calculation and conditions involving $\Longrightarrow$ require nested iteration over policy sets.

Appendix B. Notational conventions

For convenience, we here list the abbreviations used in the paper:

A	-	access
a	-	object expression
B	-	block
b	-	boolean expression
C	-	class
c	-	tagged data value
D	-	method declaration

d	-	data value
e	-	expression
f	-	function
I	-	interface
J	-	interface
l	-	policy level / tag
m	-	method
n	-	method
N	-	type name
o	-	object
P	-	policy
pr	-	program
p	-	subject
R	-	purpose
s	-	statement
S	-	policy set
T	-	type
u	-	method call identity
v	-	variable
w	-	field
x	-	local variable
y	-	method parameter
z	-	class parameter

References

[1] The ABS language https://abs-models.org/. (Accessed 14 May 2021).
[2] Robin Adams, Sibylle Schupp, 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.
[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, Yirong Xu, An XPath-based preference language for P3P, in: Proceedings of the 12th International Conference on World Wide Web, ACM, 2003, pp. 629–639.
[4] Pauline Anthonysamy, Matthew Edwards, Chris Weichel, Awais Rashid, Inferring semantic mapping between policies and code: the clue is in the language, in: International Symposium on Engineering Secure Software and Systems, Springer, 2016, pp. 233–250.
[5] Emma Arfelt, David Basin, Søren Debois, Monitoring the GDPR, in: European Symposium on Research in Computer Security, Springer, 2019, pp. 681–699.
[6] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, Matthias Schunter, Enterprise privacy authorization language (EPAL), IBM Res. 30 (2003) 31.
[7] A. Barth, A. Datta, J.C. Mitchell, H. Nissenbaum, Privacy and contextual integrity: framework and applications, in: 2006 IEEE Symposium on Security and Privacy (S&P'06), May 2006, pp. 15–198.
[8] Adam Barth, Design and analysis of privacy policies, PhD thesis, Stanford University, 2008.
[9] David Basin, Søren Debois, Thomas Hildebrandt, On purpose and by necessity: compliance under the GDPR, in: Proceedings of Financial Cryptography and Data Security, vol. 18, 2018, pp. 20–37.
[10] David Basin, Felix Klaedtke, Samuel Müller, Birgit Pfitzmann, 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.
[11] Saliha Irem Besik, Johann-Christoph Freytag, Ontology-based privacy compliance checking for clinical workflows, in: LWDA, 2019, pp. 218–229.
[12] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, Albert Mingkun Yang, A survey of active object languages, ACM Comput. Surv. 50 (5) (October 2017) 76.
[13] Travis D. Breaux, Hanan Hibshi, Ashwini Rao, Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements, Requir. Eng. 19 (3) (2014) 281–307.
[14] Ann Cavoukian, 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.
[15] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, José F. Quesada, Maude: specification and programming in rewriting logic, Theor. Comput. Sci. 285 (2) (2002) 187–243.
[16] Michael Colesky, Jaap-Henk Hoepman, Christiaan Hillen, A critical analysis of privacy design strategies, in: 2016 IEEE Security and Privacy Workshops (SPW), IEEE, 2016, pp. 33–40.
[17] George Danezis, Josep Domingo-Ferrer, Marit Hansen, Jaap-Henk Hoepman, Daniel Le Métayer, Rodica Tirtea, Stefan Schiffner, Privacy and data protection by design-from policy to engineering, arXiv preprint arXiv:1501.03726, 2015.
[18] Steven A. Demurjian, Eugene Sanzi, Thomas P. Agresta, William A. Yasnoff, Multi-level security in healthcare using a lattice-based access control model, Int. J. Privacy Health Inf. Manag. (IJPHIM) 7 (1) (2019) 80–102.
[19] 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 24 November 2019).
[20] José Luis Fernández-Alemán, Inmaculada Carrión Señor, Pedro Ángel Oliver Lozoya, Ambrosio Toval, Security and privacy in electronic health records: a systematic literature review, J. Biomed. Inform. 46 (3) (2013) 541–562.
[21] Pietro Ferrara, Fausto Spoto, Static analysis for GDPR compliance, in: Proceedings of the Second Italian Conference on Cyber Security, Milan, Italy, in: CEUR Workshop Proceedings, vol. 2058, 2018, Aachen, Proceedings available online at http://ceur-ws.org/Vol-2058/paper-10.pdf.
[22] Simone Fischer-Hbner, Stefan Berthold, Privacy-enhancing technologies, in: Computer and Information Security Handbook, Elsevier, 2017, pp. 759–778.
[23] Simone Fischer-Hübner, IT-Security and Privacy: Design and Use of Privacy-Enhancing Security Mechanisms, Springer-Verlag, 2001.
[24] Seda Gürses, Carmela Troncoso, Claudia Diaz, Engineering privacy by design reloaded, in: Amsterdam Privacy Conference, 2015, pp. 1–21.
[25] Marita Hansen, Jaap-Henk Hoepman, Meiko Jensen, Stefan Schiffner, Readiness analysis for the adoption and evolution of privacy enhancing technologies: methodology, pilot assessment, and continuity plan, Technical report, ENISA, 2015.

[26] Katia Hayati, Martín Abadi, Language-based enforcement of privacy policies, in: International Workshop on Privacy Enhancing Technologies, Springer, 2004, pp. 302–313.
[27] Daniel Hedin, Luciano Bello, Andrei Sabelfeld, Information-flow security for JavaScript and its APIs, J. Comput. Secur. 24 (2) (2016) 181–234.
[28] Carl Hewitt, Peter Bishop, Richard Steiger, A universal modular ACTOR formalism for artificial intelligence, in: Proceedings of the Third International Joint Conference on Artificial Intelligence, IJCAI'73, Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.
[29] Jaap-Henk Hoepman, Privacy design strategies, in: IFIP International Information Security Conference, Springer, 2014, pp. 446–459.
[30] Christian Johansen, Olaf Owe, Dynamic structural operational semantics, J. Log. Algebraic Methods Program. 107 (2019) 79–107.
[31] Einar Broch Johnsen, Olaf Owe, An asynchronous communication model for distributed concurrent objects, Softw. Syst. Model. 6 (1) (Mar 2007) 39–58.
[32] Einar Broch Johnsen, Olaf Owe, Ingrid Chieh Yu. Creol, A type-safe object-oriented model for distributed concurrent systems, Theor. Comput. Sci. 365 (1–2) (2006) 23–66.
[33] Farzane Karami, Olaf Owe, Toktam Ramezanifarkhani, An evaluation of interaction paradigms for active objects, J. Log. Algebraic Methods Program. 103 (2019) 154–183.
[34] Ponnurangam Kumaraguru, Lorrie Faith Cranor, Jorge Lobo, Seraphin B. Calo, 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.
[35] Butler W. Lampson, Protection, Oper. Syst. Rev. 8 (1) (1974) 18–24.
[36] Daniel Le Métayer, 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.
[37] Massimo Marchiori, Lorrie Cranor, Marc Langheinrich, Martin Presler-Marshall, Joseph Reagle, The platform for privacy preferences 1.0 (P3P1.0) specification, in: World Wide Web Consortium Recommendation REC-P3P-20020416, 2002.
[38] Amirreza Masoumzadeh, James B.D. Joshi Purbac, Purpose-aware role-based access control, in: OTM Confederated International Conferences "On the Move to Meaningful Internet Systems", Springer, 2008, pp. 1104–1121.
[39] 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-regulation-in-20-years-gdpr-b9026b9acfa9. (Accessed 20 December 2019).
[40] Qun Ni, Elisa Bertino, Jorge Lobo, Carolyn Brodie, Clare-Marie Karat, John Karat, Alberto Trombeta, Privacy-aware role-based access control, ACM Trans. Inf. Syst. Secur. 13 (3) (2010) 24.
[41] Flemming Nielson, Hanne R. Nielson, Chris Hankin, Principles of Program Analysis, Springer Publishing Company, Incorporated, 2010.
[42] Nierstrasz Oscar, A tour of hybrid – a language for programming with active objects, in: Advances in Object-Oriented Software Engin., Prentice-Hall, 1992, pp. 67–182.
[43] Nicolás Notario, Alberto Crespo, Yod-Samuel Martín, Jose M. Del Alamo, Daniel Le Métayer, Thibaud Antignac, Antonio Kung, Inga Kroener, David Wright, PRIPARE: integrating privacy best practices into a privacy engineering methodology, in: 2015 IEEE Security and Privacy Workshops, IEEE, 2015, pp. 151–158.
[44] Olaf Owe, Verifiable programming of object-oriented and distributed systems, in: Luigia Petre, Emil Sekerinski (Eds.), From Action Systems to Distributed Systems - the Refinement Approach, Chapman and Hall/CRC, 2016, pp. 61–79.
[45] Toktam Ramezanifarkhani, Olaf Owe, Shukun Tokas, A secrecy-preserving language for distributed and object-oriented systems, J. Log. Algebraic Methods Program. 99 (2018) 1–25.
[46] Andrei Sabelfeld, Andrew C. Myers, Language-based information-flow security, IEEE J. Sel. Areas Commun. 21 (1) (2003) 5–19.
[47] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, Charles E. Youman, Role-based access control models, Computer 29 (2) (1996) 38–47.
[48] Gerardo Schneider, Is privacy by construction possible?, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2018, pp. 471–485.
[49] Shayak Sen, Saikat Guha, Anupam Datta, Sriram Rajamani, Janice Tsai, Jeannette Marie Wing, Bootstrapping privacy compliance in big data systems, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 327–342.
[50] OASIS Standard, Extensible access control markup language (XACML) version 2.0, 2005.
[51] Shukun Tokas, Olaf Owe, A formal framework for consent management, in: Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, in: Lecture Notes in Computer Science, vol. 12136, Springer, 2020, pp. 169–186.
[52] Michael C. Tschantz, Jeannette M. Wing, Formal methods for privacy, in: International Symposium on Formal Methods, Springer, 2009, pp. 1–15.
[53] Peter Y.H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, Rudolf Schlatte, The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems, Int. J. Softw. Tools Technol. Transf. 14 (5) (2012) 567–588.
[54] Naikuo Yang, Howard Barringer, Ning Zhang, A purpose-based access control model, in: Third International Symposium on Information Assurance and Security, IEEE, 2007, pp. 143–148.