



Kent Academic Repository

Baramashetru, Chinmayi Prabhu, Paja, Elda and Kamburjan, Eduard (2026)
Formal Specification and Boundary Objects: A Two-Layered Approach to GDPR Specification. In: 13th International Symposium, International Symposium on Leveraging Applications of Formal Methods. Lecture Notes in Computer Science , 17143. pp. 34-52. Springer ISBN 978-3-032-40117-5. E-ISBN 978-3-032-40118-2.

Downloaded from

<https://kar.kent.ac.uk/117322/> The University of Kent's Academic Repository KAR

The version of record is available from

https://doi.org/10.1007/978-3-032-40118-2_3

This document version

Publisher pdf

DOI for this version

Licence for this version

CC BY (Attribution)

Additional information

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in **Title of Journal** , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our [Take Down policy](https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies) (available from <https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies>).



Formal Specification and Boundary Objects: A Two-Layered Approach to GDPR Specification

Chinmayi Prabhu Baramashetru¹(✉), Elda Paja²,
and Eduard Kamburjan^{2,3}

¹ University of Kent, Canterbury, UK
c.baramashetru@kent.ac.uk

² IT University of Copenhagen, Copenhagen, Denmark

³ University of Oslo, Oslo, Norway

Abstract. Specification remains a major bottleneck for formal verification, all the more when it comes to non-functional properties or if the specification has to be understood by non-technical stakeholders. In this paper, we investigate formal specification from the perspective of boundary objects. A boundary object is an artifact that enables to create a shared understanding between different groups, which is a core property for auditing of code, where developers must demonstrate that they implemented the intended behaviour. To this end, we consider the General Data Protection Regulation, which governs the processing of personal data within the European Union. On an example program, we illustrate three different ways to express the intent to be compliant with the GDPR, and discuss how simple, informal specification can be used in auditing, while complex, formal specification lack the flexibility to act as boundary objects. We conclude with a discussion how to relate these two layers of specification formally.

Keywords: Privacy · Formal Specification · JML

1 Introduction

The General Data Protection Regulation (GDPR) [16] is a regulation of the European Union that governs the processing of personal data. It applies to organisations that handle the personal data of EU residents, regardless of where the organisation itself is located. Despite its practical importance, a comprehensive formal specification of the GDPR remains an open challenge. In this paper, we use the GDPR as a motivating case to highlight broader challenges in formal specification that become particularly visible in regulatory settings.

For software-intensive systems, demonstrating GDPR compliance is challenging because the regulation states obligations in high-level abstract legal terms, such as “data collection”, “purpose limitation”, “consent” and “lawful processing”,

while software systems operate through concrete implementations, APIs, data structures, and control flow. In practice, organisations typically demonstrate compliance through accountability mechanisms such as policies, records of processing activities, risk assessments, internal controls and audits. During such activities, stakeholders (typically a Data protection officer, DPO) must interpret how legal concepts correspond to technical behaviour and assess whether the implemented system aligns with the declared purposes and legal bases. However, these artefacts are often documented in natural language, maintained separately from source code and may become outdated as the software system evolves. As a result, there is often no systematic way to demonstrate that the implementation actually realizes the intended privacy guarantees.

Formal specification of GDPR compliance faces a challenge that is common to formal specifications in general: they serve different communication purposes for different audiences. On the one hand, developers use specifications to talk with non-technical stakeholders, such as DPOs, auditors, project managers or domain experts, explaining how requirements have been supported by the system design. On the other hand, specifications are also used to communicate with verification tools and other developers, where precision, complements and formal semantics are essential. These two purposes create different design pressures. Collaboration with (non-technical) stakeholders requires that the specification must be understandable enough to connect it to the requirements' domain, while formal reasoning and proving requires detailed, mathematically precise and often lengthy, specifications (a common problem in deductive verification) [11, 12, 19, 31].

The informality, however, is not only a limitation, it is also critical to function as a specification in the organization between different stakeholders to support collaboration across organizational roles [28, 29]: Such specifications act as *boundary objects* [33] – objects that (1) carry information, (2) are flexible enough to adapt to different local needs, and (3) support shared understanding between communities with different expertise. Critically, boundary objects must not only carry multiple meanings, but also actively support cooperation between different groups through shared but adaptable structures [32]. We argue that *formal* specifications do not act as boundary objects in general, simply because they are too rigid or specialized for negotiation. However, formal specification can strengthen the use of informal specification by connecting the specified, intended behaviour with the actual behaviour.

To use formal specifications in GDPR compliance, one must combine their formal semantics with the flexibility required to function as a boundary object. One way is to structure the specification into several layers, e.g., through refinement, and to trace realized non-technical specification throughout, akin to requirement traceability. The top-most layer acts as a boundary object, while the bottom-most layer acts a precise description of the behaviour that follows from the current understanding. We therefore propose a two-layered approach, where the first layer provides lightweight, stakeholder facing annotations that document the intended interpretation of GDPR-relevant behaviour, while the second

layer gives a precise formal embedding of a privacy-aware semantics as ghost specification code suitable for verification [9, 20, 23]. Proofs then establish that the implementation conforms to this generated semantics while preserving the observable behaviour of the original program. In this way, the first layer supports auditing and communication, while the second layer provides machine-checkable assurance for selected/relevant GDPR obligations.

2 Background: The General Data Protection Regulation

Since 2018, the GDPR has become a de facto standard for data protection in the European Union. Many jurisdictions have since adopted comparable data protection frameworks, including Brazil’s Lei Geral de Proteção de Dados (LGPD) [4], Japan’s Act on the Protection of Personal Information (APPI) [1], South Korea’s Personal Information Protection Act (PIPA) [2], and state-level legislation in the United States, most notably the California Consumer Privacy Act (CCPA) [3]. Although there might be differences in the legal details across countries, several core concerns are shared, including lawful processing, transparency, accountability, and the protection of individuals’ personal data. In this paper, we recall only those GDPR principles that our specification refers to directly and then discuss how compliance is validated in practice (Sect. 2.2) and which formal approaches have been proposed in the literature (Sect. 2.3).

2.1 Core Concepts

The GDPR is centred around the idea of protecting users’ *personal data* and giving them control over how their data is used. *Personal data* is any information relating to an identified or identifiable natural person, called the *data subject* (Art. 4(1)). An organisation responsible for lawful processing is a *controller* and one processing on a controller’s behalf is a *processor* (Art. 4(7–8)). *Processing* (Art. 4(2)) is an umbrella term covering any operation performed on personal data, such as collection, storage, retrieval, use, disclosure, transfer, or deletion. We follow the action vocabulary from the GDPR and its interpretation in recent privacy-aware language semantics [9, 20, 23] and distinguish five verbs: *use*, *collect*, *store* and *transfer*, and *delete*.

Two main GDPR principles that drive the specification in source code are *purpose limitation* (Art. 5(1)(b)) and *lawful processing* (Art. 6). Purpose limitation requires data to be collected for specified and legitimate purposes, any secondary use is not compliant. Lawful processing requires valid *consent* of the data subject (Art. 6(1)(a), Art. 7), as it provides a clear and illustrative basis for modelling purpose-dependent permissions. A valid consent under the GDPR is not a single Boolean but a per-purpose, per-controller decision by the subject that must be *informed*, *freely given*, and can be *revoked* at any time.

To reason about overall compliance, one needs to express *accountability* and how controllers and processors guarantee compliance. Some organizations must appoint a Data Protection Officer (DPO) (Art. 37) responsible for advising on, and monitoring compliance and reporting to the authorities.

2.2 Current Practice: Audits and DPOs

In practice, GDPR compliance is rarely determined through source-code analysis alone. Organisations typically rely on a combination of governance processes, documentation, technical controls, and internal or external *audits* together with the DPO and other stakeholders. The DPO reviews documentation, interviews engineers, and checks system configurations to check whether the system processes data in a GDPR-compliant way. Where required, a DPO advises on compliance, monitors organizational practices and supports communication with supervisory authorities. This whole process is largely informal and labour-intensive. Legal concepts such as collection, use, transfer, or purpose must be interpreted in technical terms. Moreover, evidence on compliance is often distributed across multiple documents, teams, and systems. Compliance documentation is typically natural language documents, disconnected from the actual implementation, and may become outdated as the software evolves. Moreover, GDPR mentions processing activities (Art. 30) and data protection impact assessments (Art. 35) that require a systematic way to track data flow and processing, which is not available in current practice and/or disconnected from actual code.

Empirical studies from industry show that one of the major challenges is inconsistent interpretation of compliance requirements between the organisations and regulatory bodies [36]. Developer-facing studies [5, 17, 25] show that developers struggle to interpret high-level GDPR terms ('data collection', 'data use', 'data transfer') and translate them to concrete software solutions. Generally speaking, legal and engineering practices often diverge in how requirements from the GDPR are interpreted and operationalized. A recent study [27] shows that the structure of the legal prose is deliberately vague so that it survives over time, whereas software requirements need precise, verifiable practices.

We address these challenges by proposing a two-layered specification approach. First, the artefacts the DPO uses during the audits are not formal specifications that a prover would consume. Rather, they are lightweight annotations that are structured enough to generate formal specification automatically, but also readable by a DPO without code knowledge. Second, the formal specification is used to connect the intended interpretation of the GDPR with the actual behaviour of the code, thus providing a supporting argument for the correctness of the interpretation.

2.3 Formal Approaches

Formal methods work on data-protection regulations even the ones prior to GDPR falls, broadly, into three lines of works. We summarise them here in relation of our approach.

Policy Languages for Privacy and Data Protection. A long line of work formalises privacy policies as machine-readable artifacts independent of any specific implementation, with the goal of supporting authoring, comparison, enforcement and audit at a level closer to legal text than to code. Early enterprise languages such as P3P [15] and EPAL [7] prior to the GDPR target authorisation rules

over data categories, purposes and obligations. Building on these, Barth *et al.* formalised Nissenbaum’s notion of *contextual integrity* in a first-order temporal logic and used it to capture portions of HIPAA, COPPA and GLBA [10]. Pardo and Schneider give a related framework for online social networks combining a knowledge-based logic with a privacy policy language [30].

Since the GDPR there are approaches targeting its specific structure. The Layered Privacy Language (LPL) [18] is designed explicitly to capture GDPR obligations such as consent, retention and provenance, and gives a layered formal definition intended to be both human- and machine-readable. Baramashetru *et al.* develop a policy language for expressing consent, business policies and regulatory obligations, with compliance checking against subject consent and against GDPR articles [9].

All of these policy languages are independent of implementation and compliance is a property of traces, logs or high-level service descriptions, and the connection to the implementation that produces those traces is established either informally or via a runtime-enforcement layer. A non-specialist stakeholder can read the policy, but the artifact they see is not necessarily tied to the running code.

Privacy-Aware Language Semantics. To operate GDPR specification as a functional requirement there are few works on programming language with an operational semantics that tracks consent, ownership and purposes, so that compliance becomes a property of the program’s reduction relation rather than of an external trace. P-AOL [8], on which we build, defines such a semantics for an active-object language. DPL [23] embeds GDPR-style purpose and consent constraints into a small-step operational semantics with runtime enforcement and the user-controlled-privacy runtime of Hublet et al. [20] combines taint-tracking and policy enforcement at the level of language primitives. These languages give a precise and mechanisable target for compliance, but they are designed for runtime enforcement or trace checking, and non-specialist stakeholders have no separate view on these as auditable artifact.

Model-Level Formalisations. Finally, several proposals model GDPR-relevant behaviour at the level of system or process models rather than code. Bigraphical reactive systems have recently been used to capture consent, purpose limitation and cross-border transfer with diagrammatic notation [6]. Such models are close to an auditor’s perspective and explicitly aim at usability, but what they verify is a property of the model, not of the running program and the connection to the implementation has to be re-established for each system.

Across these three lines of work, the formal artifact and the developer-facing artifact are either the same object (privacy-aware semantics, where the program *is* the specification) or wholly separate (regulatory logics and process models, where the formalism lives apart from the code). Neither configuration provides what GDPR auditing actually requires i.e., a specification artifact that a non-specialist stakeholder can read and contest, that is mechanically connected to the implementation, and that drives a verifiable obligation about the code rather than about a separate model. This is the gap the two-layered specification is designed to close.

3 Overview and Illustrative Example

As we have seen, reasoning about GDPR compliance on a source code level requires to connect deliberately informal terminology with the software. Furthermore, this reasoning requires collaboration accross organizational roles. We also point out, that this differs from classical, functional requirements, which can be *refined* into the implementation. Even in a privacy-by-design approach, refinement would require to argue how the GDPR terminology is mirrored in the resulting software, with auditing then considering both refinement process and final source code.

Formal specification thus plays a dual role. For one, it is used to document how the GDPR terminology is interpreted. This role is used for audits. For another, it is used to check that the program actually adheres to this interpretation. This second role is concerned with expressing forbidden and allowed traces of the program, and proving that no forbidden trace can be generated.

This is most prominent in GDPR compliance, but is a general phenomenon. Formal specification plays a dual role of (1) documenting intended interpretation of context or requirements, and (2) expressing intended behavior of the program interpreted *in itself*. Some formal specification, however, only has the second role: auxiliary specification, such as loop invariants, express no interpretation but only inteded behavior. For this reason, they can be automatically generated without additional validation towards requirements.

Java

```

1 public class Database {
2   private Map<User, String> db = new HashMap<>();
3
4   public void setUserData(User user, String d){ db.put(user, d); }
5   public String requestUserData(User user){ return db.get(user); }
6   public boolean hasUser(User u) { return db.containsKey(u); }
7 }

```

Java

```

1 public class Doctor {
2   public Hospital hospital;
3   public Doctor(Hospital hospital){ this.hospital = hospital; }
4
5   public void registerPatient(User u, String d){
6     hospital.registerPatient(u, d); }
7   public void startAppointment(User u){
8     String d = hospital.requestPatientInfo(u);
9     /* do things here */
10  }
11 }

```

Fig. 1. Hospital management example (1): Database and doctor.

```

Java
1 public class Hospital {
2   public Database db;
3   public Hospital(Database db){ this.db = db; }
4
5   void registerPatient(User u, String str){
6     if(!db.hasUser(u)){ db.setUserData(u, str); } }
7   public String requestPatientInfo(User u){
8     return db.requestUserData(u);}
9   public Boolean checkPatientState(String d1, String d2){
10    return mutualPartners(d1,d2); }
11 }

Java
1 public class Therapist {
2   public Hospital hospital;
3   public Therapist(Hospital hospital){ this.hospital = hospital; }
4
5   public void coupleTherapy(User u1, User u2){
6     String d1 = hospital.requestPatientInfo(u1);
7     String d2 = hospital.requestPatientInfo(u2);
8     if(hospital.checkPatientState(d1, d2)) { /* do things here */ }
9   }
10 }

```

Fig. 2. Hospital management example (2): Hospital and therapist.

We propose that formal specification should be structured in two layers. The first layer should be used for documentation of intended interpretation and used for auditing, the second layer should be used for verification. While ideally they should have the same formal semantics, we do not consider this crucial – layer 1 must be more abstract and less detailed than layer 2. The two layers are therefore connected by validation, or by a partial equivalence, as proposed by Kamburjan and Gurov in a different setting [22]. The difference between the layers, however, is not abstraction or level of detail. Instead, layer 1 is a *boundary object* targeting communication between different stakeholders and organizational roles, while the layer 2 is not. This decouples usability questions in language design – layer 1 is freed from being bound to a formal semantics, while layer 2 is freed from being understandable by non-experts. When verified, layer 2 provides supporting arguments for layer 1, but layer 1 can be audited on its own.

Example: Hospital. We use the following example of a hospital management system to illustrate our approach. All material, including full source code and specification, is available at <https://github.com/chinmayipr/PrivSpec>. The relevant source code is shown in Figs. 1 and 2. A **Hospital** is the data controller and mediates the access to a **Database** to store patient data. A **Doctor** registers patients and accesses their records for treatment purposes. A **Therapist** can

```

Java
1 public class HospitalAnnotated {
2     public DatabaseAnnotated db;
3     public HospitalAnnotated(DatabaseAnnotated db) {...}
4
5     @GDPRCollect(subject = "u",
6                 purpose = Purpose.TREATMENT)
7     @GDPRTransfer(subject = "u",
8                 purpose = Purpose.TREATMENT,
9                 toEntity = Entity.DB)
10    void registerPatient(User u, String str) {...}
11
12    @GDPRUse(subjects = {"u"},
13            purpose = Purpose.TREATMENT)
14    @GDPRTransfer(subject = "u",
15                purpose = Purpose.TREATMENT,
16                toEntity = Entity.DB)
17    public String requestPatientInfo(User u) {...}
18
19    @GDPRUse(subjects = {"d1", "d2"},
20            purpose = Purpose.TREATMENT)
21    public Boolean checkPatientState(String d1, String d2) {...}
22 }

```

Fig. 3. Layer 1 specification for the hospital

perform couple therapy, which needs consent from *both* the patients (this cross-subject case that makes GDPR compliance non-trivial). The system must be GDPR-compliant, i.e., the hospital must only process data that it has consent for, and only for the purpose of providing healthcare services. The challenge here is how do we know this code handles personal data correctly? The code itself gives no indication which operations constitute *collection* or *transfer* in the GDPR sense, which data is personal, or which purposes justify the processing. These interpretations exist only in the developer's mind and, at best, in informal documentation. This code compiles and runs as standard Java. We have no notion of consent, personal data and data owners. Below, we show how the first layers of specification is used.

The first layer adds GDPR-intent annotations to method signatures, expressing the developer's interpretation of the GDPR for this program. These annotations identify which methods constitute which GDPR-relevant actions, for which purposes, and under which consent conditions. Crucially, no method body is changed: the annotations are pure metadata, structured enough to drive the Layer 2 transformation automatically, yet readable by a DPO during an audit without Java knowledge. These annotations are the *boundary object* that are readable by a DPO during an audit, yet structured enough to generate the sec-

ond layer automatically. We use four kinds of annotations to document intended interpretation of GDPR terminology. They are used on method level.

- `@GDPRCollect(subject, purpose)` marks a method as constituting collection of the named subject’s personal data for the given purpose;
- `@GDPRUse(subjects[], purpose)` marks use;
- `@GDPRTransfer(subject, purpose, toEntity)` marks transfer to a named entity; and
- `@RequiresConsent(subjects[], purpose)` declares the consent precondition that must hold at call time. The parameter names in annotation elements refer to the corresponding method parameter names.

Figures 3 and 4 show the annotated classes for the hospital and therapist. The annotation `@RequiresConsent(subjects={"u1","u2"}, purpose=THERAPY)` on `coupleTherapy` is one central claim: a DPO reading this annotation understands immediately that couple therapy accesses data of two patients and requires independent consent from both for the purpose `THERAPY`. The DPO can challenge this interpretation – for instance, arguing whether joint consent suffices or whether the hospital entity should be listed as authorised processor – without needing to understand Java. This flexibility is the boundary object property: negotiation happens at the annotation level, not in the code.

4 Instrumented and Fully Specified Programs

The previous section shows annotations that can be used for auditing, and their design is based on the terminology of GDPR concepts. Per se, they have no semantics for the program execution. However, we can understand them in terms of the program itself as well. We can understand them as points where code has to be injected for GDPR-aware execution semantics, as proposed by Baramashetru

Java

```

1 public class TherapistAnnotated {
2   public HospitalAnnotated hospital;
3
4   public TherapistAnnotated(HospitalAnnotated hospital) {
5     this.hospital = hospital; }
6   @GDPRUse(subjects = {"u1", "u2"},
7             purpose = Purpose.THERAPY)
8   @RequiresConsent(subjects = {"u1", "u2"},
9                     purpose = Purpose.THERAPY)
10  public void coupleTherapy(User u1, User u2) {
11    String d1 = hospital.requestPatientInfo(u1);
12    String d2 = hospital.requestPatientInfo(u2);
13    if (hospital.checkPatientState(d1, d2)) {
14      /* do things here */ } } }

```

Fig. 4. Layer 1 specification for the therapist

et al. [9]. We can also understand them as a specification about GDPR-specific trace properties.

In the following, we examine both. First we show in Sect. 4.1 how the GDPR-aware semantics can be used as the basis for a program transformation that instruments the programs to ensure that the developer’s interpretation of the relevant GDPR concepts is enforced. Second we show in Sect. 4.2 how the annotations correspond to JML specifications, which we subsequently verify.

In either case, the transformed or specified code is significantly enlarging the program and makes it more complex. We argue that this second layer of specification is not suited for negotiation with the DPO or non-developers – the transformed object thus lacks the necessary properties to be a boundary object. For this reason, directly operating on this specification is not suitable for auditing.

4.1 Instrumented Semantics

After interpretation of the GDPR terminology, GDPR compliance is essentially a trace property: If an action requires a certain consent, then this consent has been given before and not revoked in between. This can be expressed either as a trace formula over program execution, using, e.g., history-based reasoning, special trace logics [14], or using privacy-aware semantics that enforce this trace property during execution. We use such a privacy-aware semantics as described by Baramashetru et al. [9]. This semantics works by interpreting statements in terms of the GDPR (e.g., assigning to a field is storage), so we can automatically instrument the program based on this semantics and the annotations. In the following we describe the instrumentation by example, and refrain from a full formal treatment.

The core of the instrumentation is a singleton class `PrivacyMonitor` that has an internal map from user to given consent, and offers methods that correspond to actions such as granting, enforcing and revoking consent. If consent is violated, an exception is thrown. The core of the class is as follows.

Java

```

1 public final class PrivacyMonitor {
2   private static PrivacyMonitor INSTANCE = new PrivacyMonitor();
3   private Map<User, Map<Action, Set<Pair<Entity, Purpose>>>> store
4     = new HashMap<>();
5
6   public void grantConsent(...) {...}
7   public void revokeConsent(...) {...}
8   public boolean checkAction(...) {...}
9   public void enforce(...) {...}
10  public Tag tagCombine(...) {...}
11  ...
12 }
```

The instrumentation operates on single data, i.e., different data can have different consent. Thus, it must also tag the data with its current consent. This

is a more complex program transformation than just adding statements and classes, as it modifies the data the program is operating on. We return to this detail further below.

Figure 5 shows a part of the `HospitalInstrumented` class, in particular the instrumented `registerPatient` methods. The method call in the branching condition is interpreted as collection for the purpose of treatment, which combines information from the annotation and the privacy-aware semantics (namely rule `Bind-Mtd` from [9]). The second call is interpreted as use and transfer for treatment, using rule `Async-Call`.¹

The above code is executable, and indeed prohibits privacy-violating execution. However, it is not usable for auditing. The code is longer (8 lines of instrumentation for 2 lines of code) and requires to argue about the program transformation, which as we have seen above is non-trivial. While the instrumented statements are quite close to the annotations, manipulating them now requires to manipulate *executable code*. As the code alters the behaviour, one must also argue why its results are transferable to the original code, while ideally one wants to verify that the instrumentation is never triggered, or at least that it throws no exception.

4.2 JML Specification

In the following, we specify a more abstract version of the privacy-aware semantics with smaller changes, and verify that it never triggers a violation. To this end, we have to make a minimally intrusive change and add a `ConsentStore` class, and pointers to an instance of this class, that keeps track of the sequence of relevant actions. Beyond this, all annotations are translated into JML specification. The `ConsentStore` class itself is a wrapper around three JML sequences, one for each action, and ghost code to modify them. Additionally, it has methods to check whether consent has been granted, and a field `consViol` that is set to true if such a check fails. Its fields are as follows. This code shows the three sequences for granted consent for collection, transfer. The size of the `ConsentStore` is the same as the one of the `PrivacyMonitor` (ca. 150 LoC).

Java

```

1 public class ConsentStore {
2   //@ public ghost boolean consViol = false;
3   //@ public ghost \seq collectLog = \seq_empty;
4   //@ public ghost \seq transferLog = \seq_empty;
5   //@ public ghost \seq useLog = \seq_empty;
6   ...
7 }

```

Figure 6 shows the method `recordCollect` that records the consent for collection (without contract) and the method `checkCollect` that checks whether consent has been granted before, and one of its contracts that expresses under

¹ This instrumentation was done manually, and the distinction is based on the operations inside the called methods. Both calls are synchronous.

```

Java
1 public class HospitalInstrumented {
2     private final Hospital delegate;
3     public final DatabaseInstrumented db;
4     ...
5     void registerPatient(User u, String str) {
6         // rule Bind-Mtd – method receives personal data as parameter
7         monitor.enforce(
8             new HashSet<User>(Set.of(u)),
9             new HashSet<Purpose>(Set.of(Purpose.TREATMENT)),
10            Entity.HOSPITAL, this, Action.COLLECT);
11        if (!delegate.db.hasUser(u)) {
12            // rule Async-Call – use and transfer before forwarding to Database
13            monitor.enforce(
14                new HashSet<User>(Set.of(u)),
15                new HashSet<Purpose>(Set.of(Purpose.TREATMENT)),
16                Entity.HOSPITAL, this, Action.USE, Action.TRANSFER);
17            db.setUserData(u, str);
18        }
19    }
20 }

```

Fig. 5. Instrumented class for the hospital.

which conditions (consent not violated so far and consent to be checked is given) correctness is preserved. For simplicity, we do not introduce tags. Consent is recorded and checked for the unique data of the user in the system. Whenever a check for consent is needed, a call to the corresponding `check*` method is injected, whenever consent is given `record*` is injected. The overall structure is more abstract than the runtime checks of the previous section, but is nonetheless rooted in the formal, privacy-aware semantics.

For a more detailed specification, let us consider the `DoctorAnnotated` class. In contrast to the `Doctor` class, it contains the injected `ConsentStore` and an invariant that expresses that the store of the hospital and its own store are the same.

```

Java
1 public class DoctorAnnotated {
2     public HospitalAnnotated hospital;
3     public ConsentStore store;
4     ...
5     /*@ public accessible \inv: this.hospital, this.store, this.hospital.store; @*/
6     /*@ public invariant hospital != null && hospital.store == store; @*/
7     ...
8 }

```

```

Java
1  /*@ public normal_behavior
2  @   requires subject != null && purpose != null;
3  @   assignable this.consViol;
4  @   requires !consViol && (!exists int i; 0 <= i && i < collectLog.length;
5  @       collectLog[i] == \dl_pair(subject, purpose));
6  @   ensures !consViol;
7  @
8  @ also
9  @ ...
10 @*/
11 public void checkCollect(User subject, Purpose purpose) {
12   //@ set consViol = consViol ||
13   //@   !(\exists int i; 0 <= i && i < collectLog.length;
14   //@       collectLog[i] == \dl_pair(subject, purpose));
15 }
16
17 /*@ public normal_behavior
18 @ ...
19 @*/
20 public void recordCollect(User subject, Purpose purpose) {
21   //@ set collectLog = \seq_concat(collectLog,
22   //@       \seq_singleton(\dl_pair(subject, purpose)));
23 }

```

Fig. 6. Excerpt of the consent store: Methods to record and check for the consent of a user to collect its data for a given purpose. The `\dl_pair` function in JML encodes pairs in the specification logic.

The specified `registerPatient` method is shown in Fig. 7. Its preconditions model the following.

- The parameters and the consent store are not null (l. 2, 8).
- The enumeration is not null (l. 3–4). This is due to limitations of the used verification tool.
- That the invariants of the nested objects hold (l. 5–7). This is due to the invariant model of the underlying logic.
- That the user has given consent to collect and use their data for treatment (l. 9–16).
- That the user has given consent to transfer their data for treatment (l. 17–20). This is needed by the `registerPatient` method.
- That the consent has not been violated so far (l. 21).

Its postcondition is that consent is not violated afterward.

This contract is far from obvious and not suited to communicate, let alone negotiate common understanding, with non-technical stakeholders such as DPOs. It requires understanding the embedding of the semantics, the invariant model of the program logic, first-order logic over ghost state and is, following

well-known results [11, 31], significantly longer than the implementation (22 lines of specification for 3 lines of implementation: 2 lines of injected checks for the consent store, 1 line of actual code). We conjecture, based on our experience, that the gap between the technical interpretation and the domain interpretation is too large and unnatural for non-developers [21]. The contract is also depending and interacting on other contracts and cannot be seen in isolation.

For these reasons, layer 2 specification is not suited for negotiation, is thus not a boundary object and cannot fulfil the role of creating common understanding between different groups: any proposed change to the specification from the, e.g., DPO, must also take into account all the formal background, which in turn prohibits focusing on the compliance.

Java

```

1  /*@ public normal_behavior
2  @   requires u != null && d != null;
3  @   requires Purpose.TREATMENT != null;
4  @   requires Entity.DB != null;
5  @   requires hospital.\inv;
6  @   requires store.\inv;
7  @   requires hospital.db.\inv;
8  @   requires store != null;
9  @   requires
10 @   (\exists int i;
11 @     0 <= i && i < store.collectLog.length;
12 @     store.collectLog[i] == \dl_pair(u, Purpose.TREATMENT));
13 @   requires
14 @   (\exists int i;
15 @     0 <= i && i < store.useLog.length;
16 @     store.useLog[i] == \dl_pair(u, Purpose.TREATMENT));
17 @   requires (\exists int i;
18 @     0 <= i && i < store.transferLog.length;
19 @     store.transferLog[i]
20 @     == \dl_pair(u, \dl_pair(Purpose.TREATMENT, Entity.DB)));
21 @   requires !store.consentViolated;
22 @   ensures !store.consentViolated;
23 @*/
24 public void registerPatient(User u, String d) throws Exception {
25   store.checkCollect(u, Purpose.TREATMENT);
26   store.checkUse(u, Purpose.TREATMENT);
27   hospital.registerPatient(u, d);
28 }
```

Fig. 7. Full specification of the `DoctorAnnotated.registerPatient` method.

The overall specification and verification effort was done in 2 working days by 1 person with limited JML and KeY experience, but a background in deductive verification. We used KeY 3.0, and verified all classes except the `Database`,

because it uses a `HashMap` for which no contracts are provided. Instead, we specified the methods for use in other methods.

5 Discussion

The method described above rests on connecting several specifications of the same program. A core feature is the encoding of GDPR-compliance using an alternative semantics, and then using this semantics to either instrument or specify the original program. We refrain from encoding the trace property explicitly in the program logic, but instead rely on a privacy-aware semantics to handle the complexity. In the following we sketch the underlying formal framework.

In the following we present a formal foundation to connect the two operational semantics: $\llbracket \cdot \rrbracket_{\mathcal{L}}$ and $\llbracket \cdot \rrbracket_{\Pi}$. The first, $\llbracket \cdot \rrbracket_{\mathcal{L}}$ is the standard semantics of the target programming language, while $\llbracket \cdot \rrbracket_{\Pi}$ is relative to Π , the set of traces that are GDPR compliant, and restricts the standard semantics to this set. We return to the description of Π further below, and first investigate these semantics.

Let Θ be the set of all traces and $\mathcal{L}$ be a programming language. The semantics of the language is a function $\llbracket \cdot \rrbracket_{\mathcal{L}} : \mathcal{L} \rightarrow \Theta$. A property Π is a set of allowed traces, i.e., $\Pi \subseteq \Theta$. The Π -semantics of $\mathcal{L}$ is a function $\llbracket \cdot \rrbracket_{\Pi} : \mathcal{L} \rightarrow \Theta$ such that

$$\forall P \in \mathcal{L}. \llbracket P \rrbracket_{\Pi} \subseteq \llbracket P \rrbracket_{\mathcal{L}} \wedge \llbracket P \rrbracket_{\Pi} \subseteq \Pi$$

That is, the semantics are indeed restricting ($\llbracket P \rrbracket_{\Pi} \subseteq \llbracket P \rrbracket_{\mathcal{L}}$) and the GDPR-semantics is indeed compliant for some Π ($\llbracket P \rrbracket_{\Pi} \subseteq \Pi$). To ensure that the program is compliant under its normal semantics ($\llbracket P \rrbracket_{\mathcal{L}} \subseteq \Pi$), we perform two steps and show that for a *given* program P the semantics are the same, i.e., $\llbracket P \rrbracket_{\mathcal{L}} \cong \llbracket P \rrbracket_{\Pi}$ for some fitting notion of equivalence $\cong$. The first step is that we require that the semantics are expressed in a certain format, for which we give a *generic* instrumentation in terms of the target language, i.e., some transformation $\tau(\cdot)$, such that the additional structure of $\llbracket \cdot \rrbracket_{\Pi}$ is encoded explicitly in the program and exactly mirrors the additional runtime checks. This introduces ghost code at every statement that is restricted by the relative semantics.

$$\forall P \in \mathcal{L}. \llbracket \tau(P) \rrbracket_{\mathcal{L}} \cong \llbracket P \rrbracket_{\Pi} \quad (1)$$

This transformation, which we show to be correct for *any* relative semantics and program, is then followed by a *program*-specific proof that the instrumentation of the program can be removed. We give a proof obligation $\phi(P)$, based on relational verification, that ensures this property for a fixed P .

$$\vdash \phi(P) \rightarrow \llbracket P \rrbracket_{\mathcal{L}} \cong \llbracket \tau(P) \rrbracket_{\mathcal{L}} \quad (2)$$

The complete argument is the given a correct transformation (Eq. 1) and a correct proof obligation scheme (Eq. 2) then for a fixed program P , the proof of $\vdash \phi(P)$ implies that every trace of P indeed falls into Π .

$$\vdash \phi(P) \rightarrow \llbracket P \rrbracket_{\mathcal{L}} \subseteq \Pi$$

There are two possibilities to construct $\phi(P)$: Either we rely on relational verification [13] to prove program equivalence, or we construct $\phi(P)$ such that it encodes Π directly. In Sect. 4.2 we used the later for pragmatic reasons: relational verification has less tool support to be used in a first study. However, we needed to introduce complex contracts that we must manually validate against the privacy-aware semantics, i.e., perform (Eq. 1) manually. Relational verification would rely only on generic properties, and we aim to develop the framework such that the system of Sect. 4.1 can be used directly for relational verification. The missing gaps are a formally correct program transformation and a proof obligation scheme, and we plan to investigate Abstract Execution [34] for the former.

To construct $\tau(P)$ different methods can be used. This work directly injected runtime enforcement statements into the code. Alternative approaches that use a mode structured mechanisms are `typstates` [35], which can be encoded in JML as well [26], or aspect-oriented programming [24], where the `PrivacyMonitor` class would be the core of an aspect for consent, and annotations would be joinpoints, which would allow for more fine-grained specifications than on method-level.

The annotations treat the two sides of consent asymmetrically, `@RequiresConsent` exposes the requirement of consent at the audit layer, while acquisition appears only at the runtime layer via the monitor’s grant and record operations. This is deliberate, and follows from the role of the annotations. Each `@GDPR*` annotation reinterprets an operation the program already performs in terms of the GDPR action vocabulary (use, collect, store, transfer, delete) whereas granting consent is a direct manipulation of the global consent state, and belongs to the layer that owns that state. A symmetric `@GrantsConsent` annotation could be possible by considering acquisition point as a boundary object too, but it can only confirm that recording occurs, not that the consent was validly obtained which requires a separate consent management framework.

6 Conclusion

Formal specification is commonly understood as a reflection of a requirement or regulation, or an alternative formulation of program behaviour. In this paper, we used the theory of boundary objects to investigate the tension between these two roles and illustrate that specification languages should have two different aspects. Only one of these aspects plays the role of a boundary object, which have been shown to play an important role in communication across organizational roles. We used GDPR, where the requirements are notoriously informal and even their interpretation must be audited, as an example and showed how one level of specification can be used to document intent, while another level can be based on a complex formalization of trace properties.

Formal specification remains a major challenge in verification. Its connection to requirements and regulations is an inherently cognitive aspect that cannot be fully automated. We hope that this work can contribute to the design of formal specification languages and tools for deductive verification that take a more human-centered view on formal methods and their role within an organization.

Acknowledgments. This work was partially supported by the DFF under the project *Graph-based Verification of Reflective Programs* (5254-00016B) and the European Union under the project *SM4RTENANCE* (101123490).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Act on the protection of personal information (APPI), act no. 57 of 2003 (2003). <https://www.japaneselawtranslation.go.jp/en/laws/view/4241/en>
2. Personal information protection act (PIPA), act no. 10465 (2011). https://elaw.klri.re.kr/eng_service/lawView.do?hseq=62389
3. California consumer privacy act (CCPA) (2018). <https://oag.ca.gov/privacy/ccpa>
4. Lei geral de proteção de dados pessoais (LGPD), law no. 13,709 (2018). <https://www.gov.br/mds/pt-br/acao-a-informacao/governanca/integridade/campanhas/lgpd>
5. Alhazmi, A., Arachchilage, N.A.G.: I’m all ears! listening to software developers on putting GDPR principles into software development practice. *Pers. Ubiquit. Comput.* **25**(5), 879–892 (2021). <https://doi.org/10.1007/S00779-021-01544-1>
6. Althubiti, E., Archibald, B., Sevegnani, M.: Formalising privacy regulations with bigraphs. *Softw. Syst. Model.* (2025). <https://doi.org/10.1007/s10270-025-01293-2>
7. Ashley, P., Hada, S., Karjoth, G., Powers, C., Schunter, M.: Enterprise privacy authorization language (EPAL 1.2). W3C Member Submission (2003). <https://www.w3.org/Submission/2003/SUBM-EPAL-20031110/>
8. Baramashetru, C.P., Tarifa, S.L.T., Owe, O.: Integrating data privacy compliance in active object languages. In: de Boer, F.S., Damiani, F., Hähnle, R., Johnsen, E.B., Kamburjan, E. (eds.) *Active Object Languages: Current Research Trends*. *Lecture Notes in Computer Science*, vol. 14360, pp. 263–288. Springer (2024). https://doi.org/10.1007/978-3-031-51060-1_10
9. Baramashetru, C.P., Tarifa, S.L.T., Owe, O., Gruschka, N.: A policy language to capture compliance of data protection requirements. In: IFM. *Lecture Notes in Computer Science*, vol. 13274, pp. 289–309. Springer (2022). https://doi.org/10.1007/978-3-031-07727-2_16
10. Barth, A., Datta, A., Mitchell, J.C., Nissenbaum, H.: Privacy and contextual integrity: framework and applications. In: 2006 IEEE Symposium on Security and Privacy (S&P 2006), pp. 184–198. IEEE Computer Society (2006). <https://doi.org/10.1109/SP.2006.32>
11. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification – specification is the new bottleneck. In: SSV, pp. 18–32. EPTCS (2012). <https://doi.org/10.4204/EPTCS.102.4>
12. Beckert, B., Hähnle, R.: Reasoning and verification: state of the art and current trends. *IEEE Intell. Syst.* **29**(1), 20–29 (2014). <https://doi.org/10.1109/MIS.2014.3>
13. Beckert, B., Ulbrich, M.: Trends in relational program verification. In: *Principled Software Development*, pp. 41–58. Springer (2018). https://doi.org/10.1007/978-3-319-98047-8_3

14. Bubel, R., Gurov, D., Hähnle, R., Scaletta, M.: Trace-based deductive verification. In: LPAR, pp. 73–95. EPiC Series in Computing. EasyChair (2023). <https://doi.org/10.29007/VDFD>
15. Cranor, L., Langheinrich, M., Marchiori, M., Presler-Marshall, M., Reagle, J.: The platform for privacy preferences 1.0 (P3P1.0) specification. W3C Recommendation (2002). <https://www.w3.org/TR/P3P/>
16. European Parliament and Council: Regulation (EU) 2016/679 of the European parliament and of the council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/EC (general data protection regulation) (text with EEA relevance). <http://data.europa.eu/eli/reg/2016/679/oj/eng>
17. Franke, L., Liang, H., Farzanehpour, S., Brantly, A., Davis, J.C., Brown, C.: An exploratory mixed-methods study on general data protection regulation (GDPR) compliance in open-source software. In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 325–336 (2024). <https://doi.org/10.1145/3674805.3686692>
18. Gerl, A., Bennani, N., Kosch, H., Brunie, L.: LPL, towards a GDPR-compliant privacy language: formal definition and usage. *Trans. Large Scale Data Knowl. Centered Syst.* **37**, 41–80 (2018). https://doi.org/10.1007/978-3-662-57932-9_2
19. Hähnle, R., Huisman, M.: Deductive software verification: from pen-and-paper proofs to industrial tools. In: Computing and Software Science. Lecture Notes in Computer Science, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
20. Hublet, F., Basin, D.A., Krstic, S.: User-controlled privacy: taint, track, and control. *Proc. Priv. Enhancing Technol.* **2024**(1), 597–616 (2024). <https://doi.org/10.56553/POPETS-2024-0034>
21. Kamburjan, E., Fiorini, S.R.: On the notion of naturalness in formal modeling. In: The Logic of Software. A Tasting Menu of Formal Methods, pp. 264–289. Lecture Notes in Computer Science. Springer (2022). https://doi.org/10.1007/978-3-031-08166-8_13
22. Kamburjan, E., Gurov, D.: Multi-perspective correctness of programs. In: ICTAC, pp. 69–86. Lecture Notes in Computer Science. Springer (2025). https://doi.org/10.1007/978-3-032-11176-0_6
23. Karami, F., Basin, D.A., Johnsen, E.B.: DPL: A language for GDPR enforcement. In: CSF, pp. 112–129. IEEE (2022). <https://doi.org/10.1109/CSF54842.2022.9919687>
24. Kiczales, G., et al.: Aspect-oriented programming. In: ECOOP. Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer (1997). <https://doi.org/10.1007/BFB0053381>
25. Klymenko, O., Kosenkov, O., Meisenbacher, S., Elahidoost, P., Mendez, D., Matthes, F.: Understanding the implementation of technical measures in the process of data privacy compliance: a qualitative study. In: Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 261–271 (2022). <https://doi.org/10.1145/3544902.3546234>
26. Mota, J., Giunti, M., Ravara, A.: On using verifast, vecors, plural, and key to check object usage (experience paper). In: ECOOP. LIPICs, vol. 263, pp. 40:1–40:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPICS.ECOOP.2023.40>
27. Negri-Ribalta, C., Lombard-Platet, M., Salinesi, C.: Understanding the GDPR from a requirements engineering perspective—a systematic mapping study on reg-

- ulatory data protection requirements. *Requirements Eng.* **29**(4), 523–549 (2024). <https://doi.org/10.1007/S00766-024-00423-4>
28. Ouriques, R., Fagerholm, F., Méndez, D., Bern, B.G.: An investigation of causes and effects of trust in boundary artefacts. *Inf. Softw. Technol.* **158**, 107170 (2023). <https://doi.org/10.1016/J.INFSOF.2023.107170>
 29. Ouriques, R., Fagerholm, F., Méndez, D., Gorschek, T., Bern, B.G., Vucic, V.: Crafting effective boundary artefacts in software engineering: a guideline-based approach. *Empir. Softw. Eng.* **31**(4), 100 (2026). <https://doi.org/10.1007/S10664-026-10847-X>
 30. Pardo, R., Schneider, G.: A formal privacy policy framework for social networks. In: Giannakopoulou, D., Salaün, G. (eds.) *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014. Lecture Notes in Computer Science*, vol. 8702, pp. 378–392. Springer (2014). https://doi.org/10.1007/978-3-319-10431-7_30
 31. Rozier, K.Y.: Specification: the biggest bottleneck in formal methods and autonomy. In: *VSTTE*, pp. 8–26. *Lecture Notes in Computer Science* (2016). https://doi.org/10.1007/978-3-319-48869-1_2
 32. Star, S.L.: This is not a boundary object: reflections on the origin of a concept. *Sci. Technol. Hum. Values* **35**(5), 601–617 (2010). <https://doi.org/10.1177/0162243910377624>
 33. Star, S.L., Griesemer, J.R.: Institutional ecology, ‘translations’ and boundary objects. *Soc. Stud. Sci.* **19**(3), 387–420 (1989). <https://doi.org/10.1177/030631289019003001>
 34. Steinhöfel, D., Hähnle, R.: Schematic program proofs with abstract execution. *J. Autom. Reason.* **68**(2), 7 (2024). <https://doi.org/10.1007/S10817-023-09692-0>
 35. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.* **12**(1), 157–171 (1986). <https://doi.org/10.1109/TSE.1986.6312929>
 36. Usman, M., Felderer, M., Unterkalmsteiner, M., Klotins, E., Mendez, D., Alégroth, E.: Compliance requirements in large-scale software development: an industrial case study. In: *International Conference on Product-Focused Software Process Improvement*, pp. 385–401. Springer (2020). https://doi.org/10.1007/978-3-030-64148-1_24

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

