Model-driven Privacy

Srđan Krstić
ETH Zürich
srdan.krstic@inf.ethz.ch

Hoang Nguyen
ETH Zürich
hoang.nguyen@inf.ethz.ch

David Basin
ETH Zürich
basin@inf.ethz.ch

ABSTRACT

Data protection regulations in many countries require IT systems to implement baseline privacy requirements like purpose limitation and consent as mandated by the GDPR. Such requirements are often specified in the system's privacy policy and are challenging to implement as system developers must address them consistently and in a cross-cutting manner. Moreover, without a formal connection between a system's privacy policy and its implementation, the system's correctness and evolution are extremely difficult to attain.

We propose a model-driven development methodology that incorporates privacy policies into the system design. Namely, we define a system's privacy model, which has precise semantics and is used to specify privacy policies. We provide semantic-preserving model transformations that generate system implementations that enforce the given privacy policies by design. We implement two such model transformations, targeting C# and Python system implementations. We evaluate our methodology on three substantial case studies and show the enforcement of privacy policies related to purpose limitation and consent. Our evaluation also demonstrates our approach's generality, effectiveness, and modest overhead.

KEYWORDS

Model-driven development, privacy, data protection, UML

1 INTRODUCTION

Motivation and problem statement. Modern IT systems implement a wide range of security and privacy requirements. The sources of privacy requirements are manifold and include privacy regulations, end-user concerns, self-imposed constraints, prioritized risk scenarios, best practices, and standards.

In this paper we focus on *data protection*, a particularly challenging class of privacy requirements. Data protection requires enforcing users' data ownership rights on data beyond the users' actual control or, dually, ensuring that any user's personal data is treated according to the user's data-usage policy [56]. As users rarely explicitly specify their data-usage policies, data protection regulations, like the EU's General Data Protection Regulation (GDPR) [32], mandate the enforcement of particular classes of baseline data-usage policies. For example, personal data may be collected only for specified, explicit, and legitimate *purposes* (Art. 5 §1 (b) GDPR) and processed for each purpose only when the owner has provided *consent* (Art. 7 §1 GDPR). In practice, a user's consent is often implicit. For example, by using a website, a user tacitly agrees with the (often hard to find) text of the website's *privacy policy* [63]. Ideally, a user may consent only to parts of the privacy policy and therefore create their *data-usage policy*, which should be enforced.

The question of how to support data protection during system development and evolution naturally arises. While notions like *privacy-by-design* have been advocated, we still lack effective methodologies and tools to formally specify privacy and data-usage policies, connect these policies with system implementations, and keep them in sync during the systems' evolution. In fact, the implementation of an enforcement mechanism and its synchronization with the data-usage policies still requires manual developer intervention in the system code, which is difficult and error-prone.

For security, model-driven development (MDD) [11, 20, 44] has proven effective in tightly coupling system implementations with security policies via design models that integrate security into the system design process. The model semantics are defined as the allowed system executions and MDD relies on semantic-preserving model transformations to generate an enforcement mechanism that prevents those system executions disallowed by the modeled policy. Model transformations are parametrized by a specific implementation technology and can target popular technology-specific infrastructure (e.g., authorization frameworks like JAAS [47]). MDD was shown to reduce system development time, and improve correctness [31] and security [17] with respect to a system specification.

Prior work. Existing MDD approaches for privacy (Section 2) focus on automatically transforming architectural system designs [8] (e.g., by adding access control mechanisms or integrity-protected logger components), or reasoning about privacy properties of different designs [7]. However, a large gap remains between designs and the concrete system implementations. In contrast, approaches for enforcing data-usage policies [19, 62] focus on the enforcement mechanism, making the policy statement implementation dependent and hence not a design artifact. Finally, there are approaches that do not propose any technical means for achieving privacy-by-design, but rather report on experiences [39], analyze regulations [6], or exclusively work with system design models [5].

Previously, Basin et al. [16] have specialized MDD to model-driven security, where they additionally model and enforce security requirements (Section 3). In particular, they focus on fine-grained access control policies that combine declarative and programmatic access control policies. In model-driven security, a software engineer first creates a design model in a design modeling language (e.g., a UML class diagram). Next, a security engineer creates a security model in a security modeling language (e.g., like SecureUML [51]). Using model transformations, a secure-by-design implementation is then obtained from the two models. The associated threat model is that system designers provide correct system specifications via design models, whereas developers may unintentionally make implementation errors. Relying on model transformations for security therefore reduces the risk of critical implementation errors.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–16

https://doi.org/XXXXXXX.XXXXXXX

Our approach. In this paper, we extend model-driven security with support for privacy by proposing a new *privacy model* (Section 4) that captures the GDPR's notions of purpose-limitation and consent. As an additional design step in our new methodology (Section 5), privacy engineers define the system's privacy model, which formalizes its privacy policy. We rely on UML as a modeling language for presenting models, but define our models more generally. We define the privacy model's set-theoretic semantics (Section 6), and present two semantic-preserving model transformations that generate ASP.NET C# and Flask Python web applications, respectively (Section 7). An application generated using our approach allows users to explicitly and selectively consent to (parts of) the application's privacy policy and the generated implementation automatically respects the user-provided consent (i.e., enforces the users' data-usage policies). In particular, whenever a user's personal data will be manipulated for a purpose not declared by the privacy policy or not consented to by the user, the attempted operation is prevented by the generated enforcement mechanism. By leveraging model transformations that systematically enforce policies in a cross-cutting manner, we can prevent policy violations resulting from developers' mistakes under the same threat model as in model-driven security [16]. Finally, we use our approach to implement three case study applications (Section 8). Our models simplify reasoning about the correctness of the generated applications, facilitate policy evolution, and generate enforcement mechanisms that operate efficiently, with low overhead.

In summary, we make the following contributions:

- We propose a privacy model that can express privacy policies incorporating purpose-limitation and consent.
- We extend model-driven development to support privacy.
- We propose two model transformations that generate complete, configured, enforcement infrastructures for privacy. One generates ASP.NET C# web applications, whereas the other generates Flask Python web applications.
- We evaluate our approach on three case studies demonstrating its generality, effectiveness, and modest overhead.

To our knowledge, this is the first approach proposing an MDD methodology for privacy with all technical means for obtaining configured enforcement infrastructure for typical privacy policies.

Scope and limitations. Privacy can be understood in a very broad and general sense. We focus on a particular privacy aspect: enforcing purpose-limitation and consent data protection requirements as typically specified in privacy policies. Therefore, for the purposes of this paper, we interpret privacy and the legal obligation of data protection as the enforcement of the appropriate data-usage policies.

While we focus on controlling the usage of collected personal data, more work remains to fully address data protection requirements. In particular, designing systems that *minimize* collected personal data (Art. 5 §1 (c) GDPR), and enforce a combination of access control and *information-flow* policies. The latter are needed to prevent subtle inference attacks. Finally, our privacy model reflects a simplified interpretation of GDPR; incorporating a more elaborate formalization of this law [59] is future work (Section 9).

2 RELATED WORK

We split related work into three parts: privacy specification, privacy enforcement, and model-driven development (MDD) for privacy.

Privacy specification. Privacy policies [63] are security policies [60] that encode the privacy rights of users of software systems. Such policies are now heavily anchored in national and international law. They can be domain-specific, like the Health Insurance Portability and Accountability Act (HIPAA) [41] and Gramm-Leach-Bliley Act (GLBA) [35]. Alternatively, they can be domain independent, like the General Data Protection Regulation (GDPR) [32], the California Consumer Privacy Act [24], and the Digital Charter Implementation Act [29]. Privacy policies must be both understandable to end users and precise. Precision is achieved by defining a formal semantics for the policy specification language, which is essential if policies are used to generate an enforcement mechanism.

Lam et al. [48] formalize certain HIPAA regulations that focus on access control rules for the use and disclosure of protected health information, including treatment information, healthcare operations, and payment. They implement their formalization using a fragment of the Datalog logic programming language and demonstrate its usability in a prototype hospital system.

Similarly, May et al. [52] extend the classical Harrison, Ruzzo, and Ullman access control model [40] to formally analyze and audit HIPAA regulations. They formalize the target regulations and automatically translate it into a Promela model for model checking.

DeYoung et al. [30] go beyond the aforementioned HIPAA regulations and formalize regulations on the protection of customer information under the GLBA, which they also automatically audit [25].

Various languages have been proposed [49] to specify privacy policies [63]. The, now obsolete, P3P [26, 27] and EPAL [10] languages have been proposed to declare the purposes for data processing in a machine readable form. With P3P, users can set their preferences, which the browser would than match against the websites' declared purposes, notifying the user about any mismatch. EPAL goes further by allowing enterprises to declare internal access control policies needed to enforce the declared privacy policies within the system. As they lack a formal semantics, P3P and EPAL policies only amount to machine readable documentation.

The Ponder language [28, 65] is an access control specification language. It can specify obligations, which is relevant for some common privacy requirements, like data minimization.

The PrimeLife Policy Language (PPL) [9] and its extensions [13] are based on the eXtensible Access Control Markup Language (XACML) with custom notions of purpose and obligation. It reuses existing XACML implementations, but it inherits its complexity.

Recently, Pilot [54] and the policy language by Baramashetru et al. [14] have been proposed to capture GDPR-specific privacy requirements. Besides purpose-limitation and consent, the former can also specify to whom private data may be transferred, while the latter can specify data retention and location-specific processing.

Our privacy model focuses on the most important aspect of GDPR: purpose limitation and consent. Unlike existing policy languages, the formal semantics of our privacy model (Section 6) can be used to generate the enforcement mechanisms.

Privacy enforcement. Guerriero et al. [38] propose a privacy policy language and an enforcement mechanism tailored for data-intensive applications. Application users can specify privacy policies in a fragment of past-time metric temporal logic. A policy is enforced by removing from the data streams all the user-owned

data items that violate the user's policy. While the approach can specify purpose limitation, consent is not accounted for and the approach is specialized for data-intensive applications.

Byun et al. [21, 22] model purpose hierarchies and distinguish between the *intended* purpose defined for each data item and the *access* purpose, which is the purpose for which the data item is accessed. Assuming that the latter is accurately stated by a user, their enforcement of purpose limitation amounts to checking whether this purpose conforms to the data item's intended purpose, where both may be complex objects. However, the above assumption is a strong limitation of this approach and, with the exception of our work and Karami et al. [43], of all other approaches in this section.

Karami et al. [43] propose Data Protection Language (DPL), a programing language with explicit constructs for purpose and consent. DPL's semantics guarantees the enforcement of purpose, consent, and storage limitation. While having these constructs available in the implementation language improves privacy, it imposes a significant technological restriction on the developer to use DPL when implementing both privacy policies and the business logic. In contrast, MDD targets multiple implementation technologies and hence developers can choose a model transformation for the technology best-suited for implementing the business logic.

MDD for privacy. Privacy by design [23] posits that privacy must be accounted for throughout the whole engineering process. In particular, software must be developed to ensure privacy policy enforcement. MDD supports this aspect of privacy by design by introducing privacy policies in the software design models.

Antignac et al. [8] propose an automatic transformation of a high-level architectural system design into a privacy preserving design. Their transformation takes a data-flow diagram (DFD) (i.e., the initial architectural design), and a privacy policy. The privacy policy classifies each DFD's data-flow based on the data subject whose data flows through it, the purpose of the flow, and the data's retention time. The model transformation enhances the input DFD with additional processes that check privacy policies before data is used (e.g., checking purpose and the existence of user's consent before data processing). These added processes are abstract and a large gap remains between designs and the concrete system implementation.

Antignac et al. [7] also support the thesis that privacy should be addressed at the architectural level and they specifically focus on data minimization policies. They propose two languages: a formal language for modeling architectures for data minimization and a logic for reasoning about the correctness of such architectures. The latter, called privacy logic, is a variant of epistemic logic with a proof system to reason about the knowledge and beliefs of different architectural components. Unfortunately, their privacy architectures are modeled at a very high level, and obtaining a concrete implementation requires a highly non-trivial manual refinement.

Privacy policies modeled in ModelSec [57] do not consider purpose or consent. According to the authors, privacy policies *ensure that information only can be read by those who are allowed*, which coincides with the standard notion of confidentiality.

3 PRELIMINARIES

We start by introducing an example application and its (functional, security, and privacy) requirements. We will use it as a running example to illustrate the concepts discussed throughout this paper. We rely on some software engineering and modeling concepts [36], such as UML notation, without introducing them.

Example 1. *Consider a simple conference management system used by researchers to publish and review papers. A researcher can be a conference chair, a committee member, or a normal user.*

Any researcher can publish papers as well as receive paper recommendations based on their personal data. To publish a paper, they must first create and initialize the object representing it. Then they must associate the object with themselves and their coauthors. Such papers are considered unpublished until the review process is completed. Researchers who are committee members can additionally review papers and possibly delegate reviews to other researchers. Finally, a researcher who is a conference chair can accept papers for publication.

Additionally, a researcher assigned to review a paper must not have any conflicts with the paper's authors, i.e., be in a direct advisor relationship or have co-authored a paper in the last two years.

We now introduce a model-driven security methodology [16], which we extend in this paper. To model the above application using the model-driven security methodology, we need to recap *data* and *security* models. The data model defines the structure of well-formed states of an application and it is designed by the software engineers. The security model represents application's fine-grained access control (FGAC) permissions, which security engineers define based on the actions allowed on data specified by the data model.

For ease of presentation, we use UML syntax for visualizing data models. Nevertheless, the ideas presented in this paper are independent from UML's particular syntax. Hence, we introduce a straightforward set-theoretic representation of our models.

Let ID be a countable set of names and $\mathsf{S} \subset \mathsf{ID}$ a set of sorts. Sorts can be *primitive sorts* from $\mathsf{PS} \subset \mathsf{S}$, consisting of natural numbers (N), integers (Z), floats (F), strings (String), and Booleans (B), or *custom* (i.e., defined by a data model).

We use $\times, +, \to$, and $\rightharpoonup$ to denote type constructors for product, sum, function, and partial function, respectively. We write A^* (A^+) for a finite (nonempty) sequence of A and () as the empty sequence. Starting from a set of sorts, we use these operations to build more complex *types*. The sets $\mathsf{dom}(f)$ and $\mathsf{rng}(f)$ are the domain and the range of the function f. We write $a \mapsto b$ for a pair (a, b) that belongs to a function.

A *data model* is a formalization of a UML class diagram.

Definition 1. *A* data model $\mathcal{D} = (\mathsf{CS}, \preceq_{\mathcal{D}}, \mathsf{attr}, \mathsf{meth}, \mathsf{asc})$ *has a finite set of sorts* CS*, a partial order* $\preceq_{\mathcal{D}}$ *on* CS*, and the functions:* $\mathsf{attr} : \mathsf{CS} \to \mathsf{ID} \rightharpoonup \mathsf{S}$, $\mathsf{meth} : \mathsf{CS} \to \mathsf{ID} \rightharpoonup \mathsf{CS} \times \mathsf{S}^* \to \mathsf{S}$, *and* $\mathsf{asc} : \mathsf{ID} \rightharpoonup (\mathsf{ID} \times \mathsf{CS}) \times (\mathsf{ID} \times \mathsf{CS})$.

For each sort $c \in \mathsf{CS}$, $\mathsf{attr}(c)$ *returns a function from attribute names to their types, and* $\mathsf{meth}(c)$ *returns a function from method names to their types. Every method name has a function type with at least one argument, which is its owning sort. A (binary) association name* $a \in \mathsf{dom}(\mathsf{asc})$*, relates an association end name and its sort to another association name and its sort, as returned by* $\mathsf{asc}(a)$.

Intuitively, sorts in CS correspond to UML classes and $\preceq_{\mathcal{D}}$ defines the classes' inheritance relationship. The remaining functions in $\mathcal{D}$ are a straightforward encoding of the classes' structure and their associations. For simplicity and without loss of generality, we

consider only binary associations. We say that a relation is $\mathsf{asc}(n)$-typed with $\mathsf{asc}(n) = ((\mathsf{a}, \mathsf{A}), (\mathsf{b}, \mathsf{B}))$ if it has type $\mathsf{A} \times \mathsf{B}$.

Example 2. *Consider the class diagram in Figure 1 (left) modeling the well-formed states of the system from Example 1. It corresponds to the data model* $(\{\mathsf{Paper}, \mathsf{Researcher}\}, \emptyset, \mathsf{attr}, \mathsf{meth}, \mathsf{asc})$ *where*

$$\begin{aligned}
\mathsf{attr}(\mathsf{Paper}) &= \{\mathsf{title} \mapsto \mathsf{String},\ \mathsf{year} \mapsto \mathbb{Z},\ \mathsf{published} \mapsto \mathsf{B}\}\\
\mathsf{attr}(\mathsf{Researcher}) &= \{\mathsf{name} \mapsto \mathsf{String},\ \mathsf{student} \mapsto \mathsf{B}\}\\
\mathsf{meth}(\mathsf{Paper}) &= \{\mathsf{publish} \mapsto \mathsf{Paper} \to (),\\
&\qquad \mathsf{assignReviewer} \mapsto \mathsf{Paper} \times \mathsf{Researcher} \to ()\}\\
\mathsf{meth}(\mathsf{Researcher}) &= \{\mathsf{recommendPapers} \mapsto \mathsf{Researcher} \to \mathsf{Papers}^*\};\\
\mathsf{asc}(\mathsf{authorship}) &= ((\mathsf{authors}, \mathsf{Researcher}), (\mathsf{papers}, \mathsf{Paper}))\\
\mathsf{asc}(\mathsf{reviewership}) &= ((\mathsf{reviewers}, \mathsf{Researcher}), (\mathsf{reviews}, \mathsf{Paper}))\\
\mathsf{asc}(\mathsf{advisorship}) &= ((\mathsf{students}, \mathsf{Researcher}), (\mathsf{advisers}, \mathsf{Researcher})).
\end{aligned}$$

An instance of a data model is an object model, which interprets sorts with finite sets of objects, and associations as relations on the objects. An object interprets all the attributes of the sort, whereas methods have a fixed interpretation for every object of a sort. Two objects are (structurally) equal if they interpret attributes equally. To distinguish structurally equal objects, each object has a unique name $o \in \mathsf{ID}$. Hence, an object of a sort c is a pair $(o \mapsto (c, ats))$, where ats is a set of attribute values. In other words, an object binds its name o to an interpretation ats of the attributes of the sort c. Since object names are unique, a finite collection of objects yields a (partial) function from object names to their interpretations.

Intuitively, an object model formalizes a UML object diagram used to model a state of a system.

Definition 2. Object model *of a data model* $\mathcal{D}$ *is a triple* $(\mathbb{O}, \mathbb{M}, \mathbb{R})$, *where* $\mathbb{O}$ *is a finite set of objects,* $\mathbb{M}$ *maps each method name* $n \in \mathsf{dom}(\mathsf{meth})$ *to a* $\mathsf{meth}(n)$*-typed function, and* $\mathbb{R}$ *maps each association name* $n \in \mathsf{dom}(\mathsf{asc})$ *to a* $\mathsf{asc}(n)$*-typed relation over objects* $\mathbb{O}$.

Example 3. *Consider the object diagram shown in Figure 1 (right) for the corresponding class diagram in the same figure. It shows three researchers, with names* $\mathsf{r}_1, \mathsf{r}_2$, *and* r_3 *representing two students and their adviser. They have authored a paper* p, *which is neither published nor reviewed yet. The corresponding object model* $(\mathbb{O}, \mathbb{M}, \mathbb{R})$ *is*

$$\begin{aligned}
\mathbb{O}(\mathsf{Researcher}) &= \{\mathsf{r}_1 \mapsto (\mathsf{Researcher}, (\text{"}\textit{Peter}\text{"}, \top)),\\
&\qquad \mathsf{r}_2 \mapsto (\mathsf{Researcher}, (\text{"}\textit{Mark}\text{"}, \top)),\\
&\qquad \mathsf{r}_3 \mapsto (\mathsf{Researcher}, (\text{"}\textit{David}\text{"}, \bot))\}\\
\mathbb{O}(\mathsf{Paper}) &= \{\mathsf{p} \mapsto (\mathsf{Paper}, (\text{"}\textit{UML, Formally}\text{"}, 2024, \bot))\};\\
\mathbb{R}(\mathsf{advisorship}) &= \{(\mathsf{r}_1, \mathsf{r}_3), (\mathsf{r}_2, \mathsf{r}_3)\}\\
\mathbb{R}(\mathsf{reviewership}) &= \{\}\\
\mathbb{R}(\mathsf{authorship}) &= \{(\mathsf{r}_1, \mathsf{p}), (\mathsf{r}_2, \mathsf{p}), (\mathsf{r}_3, \mathsf{p})\}\\
\mathbb{M}(\mathsf{publish}) &= \lambda p.\ f_1\\
\mathbb{M}(\mathsf{assignReviewer}) &= \lambda (p, r).\ f_2\\
\mathbb{M}(\mathsf{recommendPapers}) &= \lambda r.\ f_3,
\end{aligned}$$

where f_1, f_2, *and* f_3 *are method implementations returning* (), (), *and* Papers^*, *respectively. For conciseness, we use object names instead of the objects themselves in* $\mathbb{R}$.

Given a data model, the set of actions $\mathcal{A}$ that a user may take is

$$\begin{aligned}
&\{\mathsf{create}(C), \mathsf{delete}(C) \mid C \in \mathsf{CS}\} \cup\\
&\{\mathsf{read}(C, n), \mathsf{update}(C, n) \mid n \in \mathsf{dom}(\mathsf{attr}(C)),\ C \in \mathsf{CS}\} \cup\\
&\{\mathsf{read}(C, n), \mathsf{add}(C, n), \mathsf{remove}(C, n) \mid \exists a \in \mathsf{dom}(\mathsf{asc}), n' \in \mathsf{ID}, C' \in \mathsf{CS}.\\
&\quad \mathsf{asc}(a) = ((n, C'), (n', C)) \text{ or } \mathsf{asc}(a) = ((n', C), (n, C'))\} \cup\\
&\{\mathsf{execute}(C, n) \mid n \in \mathsf{dom}(\mathsf{meth}(C)),\ C \in \mathsf{CS}\}.
\end{aligned}$$

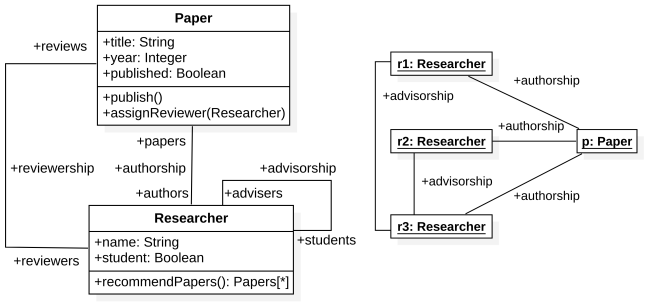

Figure 1: UML class (left) and UML object (right) diagrams of the conference management system from Example 1.

These consist of creating or deleting objects, reading or updating attributes or associations, and executing methods.

The *object constraint language* (OCL) [37] is an order-sorted first-order logic used to query object models or define properties of data models. OCL syntax has a pre-defined part used with primitive sorts (e.g., integer addition or string concatenation) and collections (e.g., **forall**, **exists**, **select**, **size**, and **excludes**), and a variable part, which is defined by a specific data model. OCL is strongly typed and every OCL expression evaluates to a value of some sort. We denote by θ_t all OCL expressions that evaluate to values of sort t. Expressions θ_{B} are called OCL constraints.

OCL's dot-operator is used to access an object's attribute (e.g., r_1.name), or an association end, i.e., the collection of objects linked with the object via an association (e.g., r_1.advisers). Collection functions are called using an arrow notation (e.g., r_1.advisers→**size**()). When used with collections, the dot-operator maps over them. OCL expressions can be written in the context of an object, and such objects can be referred to using the keyword **self**.

Example 4. *If the OCL constraint* **self**.title.size() > 0 *is given as an invariant of the* Paper *sort, it would restrict the possible object models to those whose objects have non-empty paper titles.*

The set of free variables of an OCL expression ϕ is $\mathsf{fv}(\phi)$, e.g., $\mathsf{fv}(\textbf{self}.\mathsf{papers}\textbf{→forall}(b \mid a.\mathsf{concat}(b.\mathsf{title}).\mathsf{size}() < c)) = \{a_{\mathsf{String}}, c_{\mathsf{Z}}\}$. Here variable b_{Paper} is bound by the **forall** function. Note that we subscript variables with their sorts for clarity.

Given $x_s \in \mathsf{fv}(\phi)$ and an OCL expression $t \in \theta_s$, $\phi[x_s/t]$ is obtained from ϕ by substituting all occurrences of x_s with t.

A *security model* defines which actions can be carried out on an object model by users in different roles, i.e., fine-grained access control (FGAC) policies. Our definition here corresponds to a simplified version of SecureUML [51] without action and user hierarchies.

Let $\mathcal{R}$ be a set of sorts, each representing a role and $\preceq_{\mathcal{R}}$ a partial order on $\mathcal{R}$. For example, $\mathcal{R} = \{\mathsf{Normal}, \mathsf{Committee}, \mathsf{Chair}\}$ and $\mathsf{Normal} \preceq_{\mathcal{R}} \mathsf{Committee} \preceq_{\mathcal{R}} \mathsf{Chair}$. Intuitively, if a role r_1 is larger than a role r_2 (denoted $r_2 \preceq_{\mathcal{R}} r_1$) then r_1 has all of r_2's permissions.

Given a data model $\mathcal{D}$, let $\mathcal{D}'$ extend it with a sort User representing the application's users with $\mathsf{attr}(\mathsf{User}) = \{\mathsf{role} \mapsto \mathcal{R}\}$. Also $\preceq_{\mathcal{D}'}$ may extend $\preceq_{\mathcal{D}}$ with (User, C) for any sort C that already represents application users (e.g., Researcher in Figure 1).

Authorization constraints are OCL constraints that express conditions under which a user (e.g., Alice) in some role (e.g., Chair) may

take some action (e.g., read the year attribute) of some resource (e.g., an object of the Paper sort). A constraint is written in the context of the object representing the resource, hence the **self** OCL expression evaluates to the object. The user object is represented as a free variable $\text{caller}_{\text{User}}$ in the constraint. Depending on the action, other factors may need to be account for when making the authorization decision. For example, when updating an attribute or an association end, the intended new value of some sort C can influence the decision, which is captured by the free variable value_C. Given a variables' valuation, the OCL constraint evaluates to a Boolean that encodes whether the user can carry out the action.

Given $\mathcal{D}'$, let $\text{Con}(V) = \{e \mid e \in \theta_{\text{B}}, \text{fv}(e) \subseteq V\}$ be the set of OCL constraints containing all free variables in V. Given an action $a \in \mathcal{A}$, the set of authorization constraints $C(a)$ is defined as follows:

$\text{Con}(\{\text{caller}_{\text{User}}\})$, if a is a create, delete, read, or execute;
$\text{Con}(\{\text{caller}_{\text{User}}, \text{value}_{\text{attr}(C)(n)}\})$, if a is $\text{update}(C, n)$; and
$\text{Con}(\{\text{caller}_{\text{User}}, \text{value}_{C'}\})$, if $((n, C'), (n', C))$ or $((n', C), (n, C'))$ are in asc, and a is $\text{add}(C, n)$ or $\text{remove}(C, n)$.

Now we define the security model that models FGAC policies.

Definition 3. *Given a data model $\mathcal{D}$, a* security model *is a 4-tuple $(\mathcal{D}', \mathcal{R}, \preceq_{\mathcal{R}}, \mathcal{PA})$, where $\mathcal{PA}$ is a set of permission assignments:*

$$\mathcal{PA} \subseteq \bigcup_{a \in \mathcal{A}} \mathcal{R} \times \{a\} \times C(a).$$

Intuitively, an action by a user is allowed if it is associated with a permission containing a satisfied authorization constraint, and the user's role is larger than the role in the permission. More precisely, a security model allows a user u to take an action a on an object c and (possibly) update the object with the value v iff there exists a $(r, a, \phi) \in \mathcal{PA}$ such that $r \preceq_{\mathcal{R}} u.\text{role}$ and $\phi[\text{caller}/u][\text{value}/v]$ evaluates to true in the context of c.

Example 5. *Consider the requirements from Example 1 and the data model $\mathcal{D}$ from Example 2, which we extend to $\mathcal{D}'$ with the* User *entity such that $\preceq_{\mathcal{D}'}$ is the same as $\preceq_{\mathcal{D}}$ except that* Researcher $\preceq_{\mathcal{D}'}$ User. *There are three roles $\mathcal{R}$ = {Normal, Committee, Chair} with* Normal $\preceq_{\mathcal{R}}$ Committee $\preceq_{\mathcal{R}}$ Chair. *In addition, the security model expresses the following security requirement from Example 1:* A user with role Normal can read an unpublished paper's title only if the user has no conflict with any of the paper's authors.

The triple (Normal, read(Paper, title), φ) $\in \mathcal{PA}$ *models the above requirement, where φ is the following authorization constraint:*

self.published or **self**.authors→ **forall**(a |
  caller.advisers→ **excludes**(a) and a.papers→ **forall**(p |
    not p = **self** and 2024 − p.year < 2 implies
      p.authors→ **excludes**(caller))).

4 MODELING PURPOSE AND CONSENT

Purpose limitation is the most prominent privacy requirement mandated by the GDPR. It states that personal data (namely, the data provided by the application's users) can only be used for the purposes that the data owners have consented to. Here one can distinguish between different semantics for data usage. Access control semantics considers only the initial access to the data as usage. Dataflow semantics additionally considers as usage any access to data explicitly derived (e.g., by assignment) from the initially accessed personal data. Finally, information-flow semantics further considers as usage any access to data implicitly derived (e.g., by branching on private data, or via other implicit or explicit flows) from the initially accessed personal data. We opt for an access control semantics as the other more draconian semantics substantially complicate the creation of usable applications.

We first must distinguish which sorts model personal data, so let $\text{CS}^{\mathcal{P}} \subseteq \text{CS}$ be the set of such sorts. Only the actions on these sorts are subject to purpose limitation. Hence, let $\mathcal{A}^{\mathcal{P}} \subseteq \mathcal{A}$ be the set of actions on sorts in $\text{CS}^{\mathcal{P}}$. Note that this is without loss of generality, as one can mark personal data at a more fine-grained level (e.g., a single attribute, or an association end) by modeling them as explicit sorts in the data model. Note that, for simplicity and to focus on the enforcement aspects of data protection, we here consider personal data as the subset of the data model sorts *fixed* at the design phase. Fine-grained and dynamic personal data definition is future work.

Let $\mathcal{P}$ be a set of basic purpose sorts, which induces the lattice $(2^{\mathcal{P}}, \subseteq)$ of complex purposes (i.e., sets of basic purposes). For example, *any* purpose is the set $\mathcal{P}$, *no* purpose is $\emptyset$, and *marketing* purpose is {TargetedMarketing, MassMarketing}. As in some previous work [21], we shall distinguish two general types of purposes: declared purpose and actual purpose.

A *declared purpose* is a purpose presented to the user in the form of the privacy policy. It is intended to be enforced during the application's execution, i.e., any access to the private data must be checked with respect to the declared purposes. In fact, declared purposes, the privacy policy, and the enforcement mechanism must always be synchronized. Therefore, in our model-driven approach, it is sufficient to specify the declared purposes, whereby the other two are then automatically generated. Furthermore, whenever the declared purposes change, users can receive an updated version of the correspondingly (re-)generated privacy policy, and the enforcement mechanism consistently changes to enforce the new policy.

To model declared purposes, we emulate the permission assignment relation from FGAC (Section 3):

$$\mathcal{PA}^{\mathcal{P}} \subseteq \bigcup_{a \in \mathcal{A}^{\mathcal{P}}} \mathcal{P} \times \{a\} \times C(a).$$

The relation $\mathcal{PA}^{\mathcal{P}}$ replaces the role notion in $\mathcal{PA}$ with the purpose.

Example 6. *Let the basic purposes of the conference management system be $\mathcal{P}$ =* {PublishPaper, AssignReviewer, RecommendPapers}. *Consider the following privacy policy from Example 1:* If you are a student, we will use your list of authored papers to recommend to you papers. *A declared purpose modeling the privacy policy is the tuple* (RecommendPapers, read(Researcher, papers), **self**.student).

A natural question arises: how should an enforcement mechanism decide whether some data access conforms to the declared purpose? In contract to the declared purpose, an *actual purpose* describes how data is actually being used. It is difficult to determine the actual purpose automatically by analyzing the design models we have seen so far. It must therefore be specified by annotating some of the methods in the data model with complex purposes. If a data access occurs as a part of an execution of a method, the method's annotation is the actual purpose for the data access. Given a data model $\mathcal{D}$, we explicitly define the application's interface as

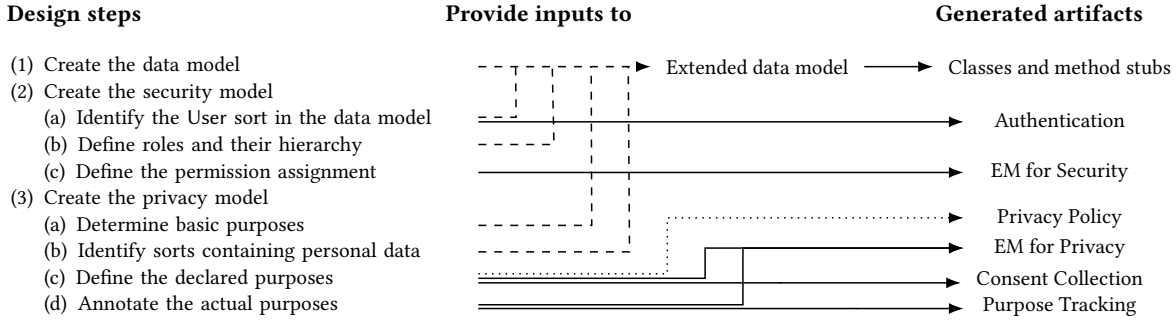

Figure 2: Model-driven privacy methodology (EM = enforcement mechanism)

a subset of methods $\mathcal{I} \subseteq \bigcup_{c \in \mathsf{CS}} \mathsf{dom}(\mathsf{meth}(c))$, and consider the function annotate : $\mathcal{I} \to 2^{\mathcal{P}}$ annotating the interface methods with (complex) purposes. We can extend the annotate function to all methods, by associating no purpose $\emptyset$ to methods outside of $\mathcal{I}$.

Example 7. *Suppose that each method in the data model from Example 2 is an interface method, i.e.,* $\mathcal{I}$ = {publish, assignReviewer, recommendPapers}. *The actual purposes can be the following:*

annotate(publish) = {PublishPaper},
annotate(assignReviewer) = {AssignReviewer}, *and*
annotate(recommendPapers) = {RecommendPapers}.

Even if declared purposes are presented to the user and enforced in the application, the GDPR further requires that personal data is only processed when the owner has provided consent. In practice, a user's consent is implicit. Typically by using a website, the user tacitly agrees with the text of the website's privacy policy. We aim to make this consent collection step explicit.

In general, users can (partially) consent to or even reject all the declared purposes. The lack of consent is treated as a rejection of that part of the privacy policy. Consent can be provided *lazily* (e.g., when a user performs an action) and can be revoked at any time.

To model consent, we must capture the relationship between a user, their personal data of some sort C, and a purpose p for each tuple (p, a, e) in the $\mathcal{PA}^{\mathcal{P}}$ relation, where the action a is taken on an object of sort C. Hence we can model consent as a relation over User $\times$ $\mathsf{CS}^{\mathcal{P}} \times \mathcal{P}$ that relates a user and a sort of personal data to a purpose that the user has consented to. For example, (Alice, Researcher, RecommendPapers) means that Alice consents that her information in the Researcher object can be used to recommend papers to her. Since collecting consent happens only once the application is running, the relation above must be maintained dynamically as a part of the data model.

Given a data model $\mathcal{D}'$ (from the security model) with identified sorts $\mathsf{CS}^{\mathcal{P}}$ containing personal data. The data model $\mathcal{D}''$ extends $\mathcal{D}'$ with a sort Consent such that attr(Consent)(purposes) = $2^{\mathcal{P}}$, attr(Consent)(user) = User, and attr(Consent)(data) = $\mathsf{CS}^{\mathcal{P}}$. A Consent object relates a user, a personal data type, and a set of purposes. Such an object exists only if the user has consented that their personal data with the appropriate type can be used for the set of purposes. The extended model also requires that every sort $p \in \mathsf{CS}^{\mathcal{P}}$ has the owner attribute, attr(p)(owner) = User, relating every instance of sort p to the user who owns it. We only model single data owners, as the treatment of shared personal data is still an open problem in GDPR [46, Problem 6]. We now combine the above notions into a privacy model.

Definition 4. *Given a data model* $\mathcal{D}$*, the privacy model is the tuple* $(\mathcal{D}'', \mathcal{P}, \mathsf{CS}^{\mathcal{P}}, \mathcal{PA}^{\mathcal{P}}, \mathsf{annotate})$.

Intuitively, an enforcement mechanism for a privacy model ensures that actual purpose always conforms to the declared purpose and that all users that own the personal data have consented to every actual purpose. More precisely, it allows a user u's action a to execute on private data d of sort D that (possibly) updates d with value v as part of the method m's execution iff annotate(m) is a subset of all declared purposes $\{p \mid (p, a, \phi) \in \mathcal{PA}^{\mathcal{P}}, \phi[\mathsf{caller}/u, \mathsf{value}/v]$ in the context of $d\}$ and there exists a Consent object c such that c.user = d.owner, c.data = D and annotate(m) $\subseteq c$.purposes.

5 METHODOLOGY

We now put all the above notions together into a model-driven methodology for developing applications that enforce privacy. Overall, an application's design is given by three models: a data model, a security model, and a privacy model. We describe each step of our methodology in detail and explain how some of the design decisions can be made. Then we show what model transformations can generate from the models. Figure 2 summarizes the methodology's design steps (left) and the outputs of the model transformations (right). Dashed arrows denote model-to-model transformations, solid arrows denote model-to-code transformations, and the dotted arrow denotes a model-to-text transformation [18].

As a first step, software engineers create a data model $\mathcal{D}$ focusing solely on the application's business logic. The data model captures the well-formed states of the application. In the next step, security engineers define the application's security model based on the data model $\mathcal{D}$. This consists of multiple sub-steps. They first pave the way for extending the data model by identifying the sort representing users of the system. They then define roles ($\mathcal{R}$), role hierarchy ($\preceq_{\mathcal{R}}$), and use them to define the permission assignment ($\mathcal{PA}$), which models FGAC policies for the data in the data model.

Finally, privacy engineers define the privacy model. They first determine the basic purposes ($\mathcal{P}$), which depend on the application's domain. One can, however, consider a taxonomy of commonly used purposes as a starting point [50]. Next, they declare

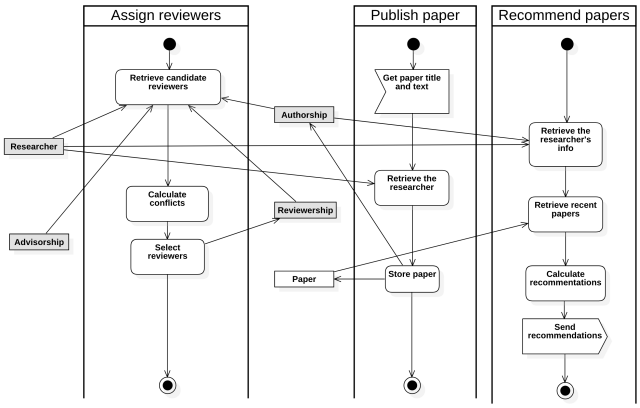

Figure 3: UML Activity diagram of the conference management system

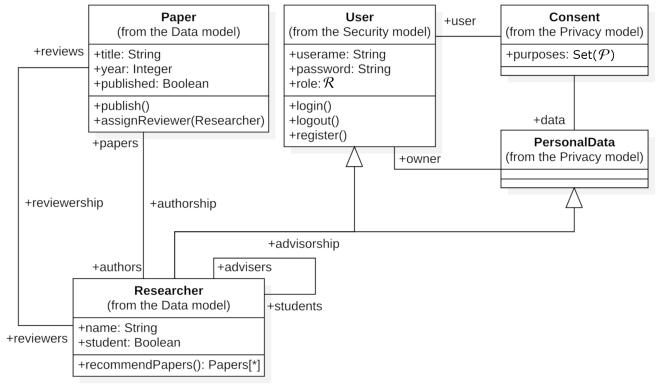

Figure 4: UML Class diagram corresponding to the extended data model of the conference management system

sorts that model personal data ($CS^{\mathcal{P}}$). Once both purposes and personal data have been identified, privacy engineers can define the declared purposes ($\mathcal{PA}^{\mathcal{P}}$). As a final step in defining the privacy model, they specify the actual purposes by annotating methods in the data model (annotate).

The annotated methods have a special role: they are the application's interface providing different business logic services. Choosing the right annotations is a creative process. One can trivially choose one basic purpose for each method in the data model. Alternatively, more precise annotations can be chosen. In general, the interface methods should correspond to the application's different business processes as modeled by a UML activity diagram [15]. In the most general case, they may call each other (in which case their actual purpose is combined in a more complex purpose), or some processes may not be annotated. For example, a general method for sorting lists is typically used as an auxiliary process and its actual purpose depends on the interface methods whose execution requires sorting. Both declared and actual purpose can be obtained by analyzing business processes, which we exemplify in the following.

Example 8. *Suppose we have access to a UML activity diagram (Figure 3) describing the business processes implemented by the methods of the conference management system described in Example 1.*

The diagram consists of three activities (shown as vertical lanes) each with input actions (⌠ symbol), process actions (⬭ symbol), and output actions (⌡ symbol) connected with directed edges describing the activities' control flow. Objects (▭ symbol) representing different sorts from the data model are shown outside of the activities' lanes. We abuse the notation and mark in grey the sorts from $CS^{\mathcal{P}}$*. Directed edges involving objects describe the data flow within the activities. Note that some objects in the activity diagram do not correspond to sorts in the data model. This can be achieved by modeling associations as explicit sorts and including them in* $CS^{\mathcal{P}}$ *(Section 4).*

Actual purposes can be derived from the diagram by observing which methods are used exclusively by other methods, as opposed to the methods that are additionally used as part of the application's interface. For the conference management system, each method is only used as an interface method and has its own actual purpose.

Declared purposes can be derived by observing the data flows from objects to process actions. For instance, the privacy policy from Example 6 is inspired by the edge that connects the Author *object to an action in the recommend papers activity.*

Identifying the purposes of a piece of code is a difficult task that cannot be readily automated. Hence, having additional manual model validation steps (either by developers, or by external privacy auditors) would help to ensure correct privacy enforcement.

Now let us consider *what* can be generated from the three models. The initial data model is first updated to account for extensions required by the security and privacy models. It is sufficient for the security engineer to identify the sort representing users by specifying its name in the security model. The data model can be automatically extended based on that information: the selected sort is declared as a subsort of the User sort. The roles and purposes can be encoded as enumerations (i.e., by using the + type constructor). A role is assigned to each user via the role attribute in the User sort, i.e., $\mathsf{attr}(\mathsf{User})(\mathsf{role}) = \mathsf{Role}$. Purposes are similarly declared (as enumerations) and associated to the Consent sort via an attribute. Distinguishing personal data sorts ($CS^{\mathcal{P}}$) from the other sorts can be achieved by declaring them as subsorts of the sort PersonalData that has the owner attribute of type User.

Example 9. *Suppose that based on the data model from Figure 1 security and privacy engineers have identified that sort* Researcher *represents both users of the system and contains personal data. Then the extended data model generated by our approach is shown in Figure 4. Namely, the diagram contains three additional classes. The* User *class identifies* Researcher *as a sort representing users and declares additional attributes and methods useful for the subsequent generation of authentication code. The* PersonalData *class identifies* Researcher *as the only sort containing personal data and declares the additional attribute* owner*. Finally, the* Consent *class captures the existing consents that users provided by declaring the attributes* user*,* data*, and* purposes*. In the diagram, attributes that refer to classes in the diagram are shown as unnamed associations.*

Once the extended model is generated, it can be used to further generate code for classes and method stubs, which still must be

implemented. Using the existing object-relational mappers, each class is also generated with persistence support defining the application's data-tier. It is also possible to use existing authentication libraries (like ASP.NET Core Identity [4] for C#, or Flask-User [2] for Python) to generate an authentication mechanism given the identified User sort. We use such a library with default settings for password-based authentication, which can further be configured.

The permission assignment ($\mathcal{PA}$) in the security model is used to generate an enforcement mechanism for the specified FGAC policies. Namely, the mechanism is called each time an attempt is made to execute any of the action from $\mathcal{A}$.

The privacy model is used to generate multiple artifacts. Firstly, privacy policy text presented to the user can be generated from the declared purposes. For example, the tuple $(p, \mathsf{read}(C, d), \phi) \in \mathcal{PA}^{\mathcal{P}}$ can be presented as the policy: *If ϕ, then we will read d of your personal data C for the purpose p*. In our concrete implementation, we provide a description field for every OCL constraint in our models, which are used when generating the privacy policy. If multiple tuples have the same action and constraint, their text can be combined and the corresponding complex purpose shown.

Similarly, the enforcement mechanism checking purpose limitation and consent is generated based on the specified declared and actual purposes and it is called whenever any of the action from $\mathcal{A}^{\mathcal{P}}$ was attempted. In addition, the consent collection mechanism that prompts users (either upon a change of declared purposes, or lazily when user data is used upon the user's request) can also be generated automatically based on the declared purposes.

Finally, a purpose tracking mechanism can be generated to track the actual purpose during the execution. It simply needs to maintain one (current) complex purpose per execution thread of the application. Whenever an interface method is invoked, the tracking mechanism adds the method's actual purpose to the current complex purpose and removes it once the method returns.

The generation process described here is technology-agnostic. In Section 7 we will describe two model transformations that we have implemented: one for ASP .NET web application and the other for Python/Flask web applications. Clearly based on the three models, only method stubs can be generated for the methods implementing custom business logic. Nevertheless, other methods that implement cross-cutting functionality like security and privacy policy enforcement, and authentication are fully implemented.

If the models' semantics allow it, the developers' implementations of the method stubs will execute successfully. Otherwise, they will throw a runtime exception. In fact, the generated enforcement mechanism would consistently throw exceptions whenever actions violate the specified models. This eliminates the risk of developers introducing implementation errors that violate users' privacy.

6 FORMAL SEMANTICS

We now provide a formal semantics for our models. Conceptually, the models formalize access control decisions of two types [16]: (1) declarative access control decisions, which depend only on static information defined in the models (e.g., roles in the permission assignment); and (2) programmatic access control decisions, which depend on the dynamic information captured by the current system's state (e.g., authorization constraint satisfaction). We extend both types of decisions to be *purpose-based*, namely, to depend on the privacy model (e.g., purposes in the declared purposes) and on the extended state of the system (e.g., existence of a Consent object).

We use first-order logic (FOL) [53] to formalize the semantics of our security and privacy models. FOL's syntax consists of logical symbols (e.g., $\neg$, $\wedge$, $\exists$), as well as non-logical ones (e.g., predicate, function, or constant symbols) defined by a *signature*. FOL's semantics is defined with respect to a valuation ν of its free variables and a *first-order structure* σ consisting of a carrier set and interpretation functions mapping non-logical symbols to predicates, functions, and constants, respectively. We write $\sigma, \nu \models \phi$ if FOL formula ϕ is satisfied by structure σ and valuation v. Instead of having a single homogeneous carrier set, *order-sorted FOL* extends FOL by considering multiple carrier sets each containing elements of the same sort.

Static information (e.g., the relations $\mathcal{PA}$ and $\mathcal{PA}^{\mathcal{P}}$) are formalized as relations in a first-order structure σ_s The dynamic information is the content of a system state (i.e., an object model of the extended data model $\mathcal{D}''$), which we also formalize as a first-order structure σ_d. The semantics of our security and privacy models is formalized by order-sorted FOL formulas φ_s and φ_p. An overall system state combines both static and dynamic information into a composite structure (σ_s, σ_d) obtained by combining σ_s and σ_d. Given a composite structure the decision to allow an action is equivalent to checking if $(\sigma_s, \sigma_d), \nu \models \varphi_s \wedge \varphi_p$ for some ν.

To define the two structures and the two formulas, we first define the respective signature Σ, partitioned into two parts Σ_s and Σ_d, representing static and dynamic information, respectively.

Signatures. Let the order-sorted signature for static information be $\Sigma_s = (\mathcal{R} \cup \mathcal{P}, \preceq_{\mathcal{R}}, \emptyset, \{\mathsf{PA}, \mathsf{PA}^{\mathsf{P}}\})$, which contains a sort for each role (in $\mathcal{R}$) and purpose (in $\mathcal{P}$), role hierarchy ($\preceq_{\mathcal{R}}$) as an order on the sorts, no function symbols, and two relation symbols (PA and PA^{P}). Intuitively, these sorts and relation symbols will be interpreted by the roles, purposes, and the (permission assignment and declared purposes) relations from our security (Section 3) and privacy (Section 4) models.

The signature for dynamic information Σ_d is built from a data model, which naturally induces an order-sorted signature with typed function and relation symbols [16]. For clarity, we further split this signature into two parts: the primitive signature Σ_P and the data model signature $\Sigma_{\mathcal{D}}$. The primitive signature is fixed and independent of any particular data model. Let $\Sigma_P = (\mathsf{PS}, \preceq_P, \Omega_P, \emptyset)$ be an order-sorted signature where the set PS contains the five primitive sorts (Section 3). The relation $\preceq_P$ is a partial order on PS such that $\mathsf{N} \preceq_P \mathsf{Z} \preceq_P \mathsf{F}$. The set Ω_P contains common function symbols on the primitive sorts, like conjunction ($\mathsf{and} : \mathsf{B} \times \mathsf{B} \to \mathsf{B}$), addition ($+ : \mathsf{F} \times \mathsf{F} \to \mathsf{F}$), string length ($\mathsf{size} : \mathsf{String} \to \mathsf{N}$), etc. The signature has no relation symbols.

Given a finite set of sorts $\mathsf{CS} \subseteq \mathsf{ID}$, where $\mathsf{CS} \cap \mathsf{PS} = \emptyset$, the set of all sorts is the smallest set S such that $\mathsf{Any} \in \mathsf{S}$, $\mathsf{CS} \cup \mathsf{PS} \subseteq \mathsf{S}$, and if $s \in \mathsf{S}$ then $\mathsf{Set}(s), \mathsf{Bag}(s), \mathsf{Sequence}(s), \mathsf{Collection}(s) \in \mathsf{S}$.[1]

A data model $\mathcal{D} = (\mathsf{CS}, \preceq_{\mathcal{D}}, \mathsf{attr}, \mathsf{meth}, \mathsf{asc})$ induces a data model signature $\Sigma_{\mathcal{D}} = (\mathsf{CS}, \preceq_{\mathcal{D}}, \Omega_{\mathcal{D}}, \mathsf{asc})$ by considering all attributes and methods as function symbols ($\Omega_{\mathcal{D}} = \bigcup_{c \in \mathsf{CS}} (\mathsf{attr}(c) \cup \mathsf{meth}(c))$), and associations as relation symbols. Finally the signature $\Sigma_d =$

[1]For simplicity, OCL types $\mathsf{Set}(T)$, $\mathsf{Bag}(T)$, and $\mathsf{Sequence}(T)$ can be encoded as $T \to \mathsf{B}$, $T \to \mathsf{N}$, and T^* types, respectively.

$(\mathsf{S}, \preceq, \Omega_P \cup \Omega_{\mathcal{D}}, \mathsf{asc})$ combines the primitive signature Σ_P and the signature $\Sigma_{\mathcal{D}}$ induced by $\mathcal{D}$. The set S is the set of all sorts. The order $x \preceq y$ on S is defined as $x \preceq_P y \vee x \preceq_{\mathcal{D}} y \vee y = \mathsf{Any} \vee \exists s.\ y = \mathsf{Collection}(s) \wedge x \in \{\mathsf{Set}(s), \mathsf{Bag}(s), \mathsf{Sequence}(s)\}$. Intuitively, it combines the sort orders from primitive and data model signatures, and introduces the (maximal) Any sort and Collection sorts.

Structures. A first-order structure $\sigma = (\llbracket\cdot\rrbracket, \{\!\{\cdot\}\!\}, \langle\!\langle\cdot\rangle\!\rangle)$ over a signature Σ consists of three interpretation functions that interpret the signature's sorts, function symbols, and relation symbols. These functions map sorts, function symbols, and relation symbols to carrier sets, functions, and relations, respectively such that:

(1) unordered sorts have disjoint carrier sets, i.e.,

$$\forall x, y \in \mathsf{S}.\ x \not\preceq y \wedge y \not\preceq x \leftrightarrow \llbracket x \rrbracket \cap \llbracket y \rrbracket = \emptyset;$$

(2) the interpretations of ordered sorts are subsets, i.e.,

$$\forall x, y \in \mathsf{S}.\ x \preceq y \leftrightarrow \llbracket x \rrbracket \subseteq \llbracket y \rrbracket;$$

(3) $\{\!\{\cdot\}\!\}$ and $\langle\!\langle\cdot\rangle\!\rangle$ map to well-typed functions and relations, i.e.,

$$\forall f : F \in \Omega.\ \forall r : R \in \rho.\ \{\!\{f\}\!\} \in \llbracket F \rrbracket \wedge \langle\!\langle r \rangle\!\rangle \in \llbracket R \rrbracket.$$

As indicated above, the structure with static information σ_s maps each role (and purpose) sort in Σ_s to a set of strings such that the sets respect $\preceq_{\mathcal{R}}$. The relation symbols in Σ_s are interpreted by the permission assignment relation ($\langle\!\langle \mathsf{PA} \rangle\!\rangle = \mathcal{PA}$) and the declared purpose relation ($\langle\!\langle \mathsf{PA}^{\mathsf{P}} \rangle\!\rangle = \mathcal{PA}^{\mathcal{P}}$), respectively.

Regarding the structure with dynamic information σ_d, we start by interpreting the primitive sorts, and then the sorts defined by the data model. Conceptually, an object model induces a first-order structure that interprets the sorts defined by the data model. The primitive sorts are interpreted by the positive 64-bit integers ($\llbracket \mathsf{N} \rrbracket = \mathbb{N}$), 64-bit integers ($\llbracket \mathsf{Z} \rrbracket = \mathbb{Z}$), IEEE-754 floating-point numbers ($\llbracket \mathsf{F} \rrbracket = \mathbb{F}$), ASCII 8-bit character strings ($\llbracket \mathsf{String} \rrbracket = \mathbb{S}$), and set of Booleans ($\llbracket \mathsf{B} \rrbracket = \mathbb{B} = \{\top, \bot\}$), respectively. The interpretation of sorts is lifted to types, (i.e., $\llbracket A \times B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket$, $\llbracket A + B \rrbracket = \llbracket A \rrbracket \cup \llbracket B \rrbracket$, $\llbracket A \to B \rrbracket = \llbracket B \rrbracket^{\llbracket A \rrbracket}$, and $\llbracket A \rightharpoonup B \rrbracket = \bigcup_{a \subseteq A} \llbracket B \rrbracket^{\llbracket a \rrbracket}$) and the interpretation of function symbols in Ω_P is as expected.

We now formally define carrier sets for sorts defined by the data model, which contain elements called objects. Recall from Section 3 that an object of a sort $c \in \mathsf{CS}$ is a pair $(o \mapsto (c, ats))$. Since attributes are just function symbols, ats is formally a set of functions $\{\{\!\{n\}\!\} \mid n \in \mathsf{dom}(\mathsf{attr}(c))\}$ interpreting the function symbols. Since object names are unique, a finite set of objects forms a (partial) function from object names to their interpretations. Hence the carrier set $\llbracket c \rrbracket$ of a sort $c \in \mathsf{CS}$ is the set of all partial functions from object names to all possible attribute interpretations, i.e.,

$$\{\!\{\mathsf{ID} \rightharpoonup (\{c\} \times \prod_{n \in \mathsf{dom}(\mathsf{attr}(c))} \llbracket \mathsf{attr}(c)(n) \rrbracket)\}\!\}.$$

An object model $(\mathbb{O}, \mathbb{M}, \mathbb{R})$ of a data model $\mathcal{D}$ induces a first-order structure where $\mathbb{O}$ defines carrier sets for $\mathcal{D}$'s sorts, $\mathbb{M}$ defines functions for $\mathcal{D}$'s methods, and $\mathbb{R}$ defines relations for $\mathcal{D}$'s associations. Otherwise, the interpretation functions are as defined above.

Model semantics. To capture the formal semantics of our models, we use the standard order-sorted FOL syntax over the signature $\Sigma = \Sigma_s \cup \Sigma_d$ additionally extended with the unary relation symbol $\mathsf{CAP} : \mathcal{P}$ modeling the current actual purposes. Since our semantics depends on authorization constraints expressed in OCL, we abuse the notation and use the OCL *constraints* as subformulas in FOL formula. This is fine as OCL is itself a FOL, only with a specialized concrete syntax. For clarity, we provide OCL's syntax and semantics in Appendix A.

The structure σ on which we evaluate the formula combines the structures σ_s and σ_d. Moreover, it is further extended to interpret the unary relation symbol $\mathsf{CAP} : \mathcal{P}$ with the set of current actual purposes, which is an abstraction representing the state of our purpose tracking mechanism. The exact content of this relation depends on the execution history of the system and it will be specified later in this section. The semantics of our models are specified for any interpretation of the CAP relation symbol.

Given a valuation ν defining a user u executing an action a on an object c (possibly) with an update value v, the formal semantics of the security model is captured by the order-sorted FOL formula φ_s:

$$\exists r'.\ \mathsf{PA}(r', a, \phi) \wedge r' \preceq_{\mathcal{R}} \mathsf{role}(u) \wedge \phi[\mathsf{caller}/u, \mathsf{value}/v],$$

where the OCL expression **self** evaluates to the object c.

Given a valuation ν defining a user u executing an action a on a personal data object d of a sort D (possibly) with an update value v, the formal semantics of the privacy model is captured by the order-sorted FOL formula φ_p:

$$\begin{aligned}&(\forall p.\ \mathsf{CAP}(p) \to \mathsf{PA}^{\mathsf{P}}(p, a, \phi) \wedge \phi[\mathsf{caller}/u, \mathsf{value}/v]) \wedge \\ &\exists c.\ \mathsf{user}(c) = \mathsf{owner}(d) \wedge \mathsf{data}(d) = D \wedge \\ &\qquad\qquad \forall p.\ \mathsf{CAP}(p) \to p \in \mathsf{purposes}(c),\end{aligned}$$

where the OCL expression **self** again evaluates to the object d.

The access control decisions can be specified based on the current system state. In contrast, the behavior of the purpose tracking mechanism (modeled by the CAP relation) depends on both the current and previous system states. Therefore, we model any system as a labeled transition system (LTS) $\Delta = (Q, \mathcal{A}', \delta)$, where the set of nodes Q contains all possible composite structures σ, edges are labeled by the actions extended with the actions corresponding to the method returns (i.e., $\mathcal{A}' = \mathcal{A} \cup \{\mathsf{return}(C, n) \mid n \in \mathsf{dom}(\mathsf{meth}(C)),\ C \in \mathsf{CS}\}$), and $\delta \subseteq Q \times \mathcal{A}' \times Q$ is transition relation. The transition relation allows only authorized actions and relates two structures based on the action taken as expected. For example, for an authorized update action, the new structure is the same as the old structure except that the new updated value is assigned, see Appendix B. The transition relation also updates the current actual purpose appropriately, i.e., δ contains:

$$\begin{aligned}&\{(q, \mathsf{execute}(C, n), q') \mid q' = q[\langle\!\langle \mathsf{CAP} \rangle\!\rangle \mapsto \langle\!\langle \mathsf{CAP} \rangle\!\rangle \cup \mathsf{annotate}(n)] \\ &\qquad\qquad \text{if } q, \nu \models \varphi_s \wedge \varphi_p\} \cup \\ &\{(q, \mathsf{return}(C, n), q') \mid q' = q[\langle\!\langle \mathsf{CAP} \rangle\!\rangle \mapsto \langle\!\langle \mathsf{CAP} \rangle\!\rangle \setminus \mathsf{annotate}(n)]\}.\end{aligned}$$

Intuitively, whenever an authorized method is executed, its actual purpose is added to the set of current actual purposes. When the method returns, the actual purpose is removed.

7 IMPLEMENTATION

In this section we describe two model transformations that implement our model-based methodology in two popular programming languages, C# and Python. In this section, we describe the code-generation process and design choices for both transformations.

Based on the three models, our model transformations create either an ASP.NET web application in C# or a Flask web application

in Python. From the extended data model, they generate a set of Entity Framework Core classes [3] in C# or SQLAlchemy [1] model classes in Python. This set of classes will be automatically mapped into an appropriate SQL database schema by the two libraries.

To handle the concept of users, roles, and the authentication process, we rely on the ASP.NET Core Identity [4] and Flask-User [2] library. We generate the user class extended with the authentication library's user class (`IdentityUser` or `UserMixin` class, respectively). Since the libraries do not support role hierarchies, we have encoded them manually in the authorization checks.

A class and the corresponding associations are also created for the purpose, personal data, and consent sorts. We configure the application such that the purposes defined in the privacy model are inserted into the database table corresponding to the purpose class whenever the database is initialized. Furthermore, a fully functional web page for collecting consent is generated in both our ASP.NET and Flask applications. The page contains the automatically generated text of the application's privacy policy and allows the user to consent to each declared purpose individually.

Each action in $\mathcal{A}$ is mapped to an existing method provided by the generated classes, e.g., for every sort $C \in \mathsf{CS}$ and attribute $n \in \mathsf{dom}(\mathsf{attr}(C))$, getter and setter methods for the property n in the class C correspond to actions $\mathsf{read}(C, n)$ and $\mathsf{update}(C, n)$, respectively. To enforce security and privacy policies, these methods need to be instrumented to call the enforcement mechanisms that implement the respective models' semantics. Such instrumentation can be achieved using different technologies. In C#, we annotate these methods with the `[Secured]` label, whose definition relies on the PostSharp C# aspect-oriented programming (AOP) library [55] to call the enforcement mechanisms. In Python, we instrument methods via a function decorator `@secured` that calls the enforcement mechanisms directly. If an action is permitted, according to both the security and privacy models, then the system executes it; otherwise, an enforcement mechanism raises an exception. If the developers do not handle the raised exception, the generated default exception handler is called that displays which permission was disallowed.

To enable the purpose tracking mechanism to track the current actual purposes, we extend the label (decorator) to include the labeled (decorated) method's actual purpose as a parameter. The parameter is a constant specified during the code generation. The definition of the label and decorator additionally includes code that pushes the supplied parameter to a thread-local stack of purposes when the interface method is called and pops it when the method returns. The stack content corresponds to the content of the CAP relation (Section 6).

Finally, our model transformations generate code for the security and privacy enforcement mechanisms. The code directly implements the set-theoretic semantics presented in Section 6. To support security and privacy policy evolution, our model transformations can also generate just the enforcement mechanisms' code, while the application's business logic remains unchanged.

Threat model and assumptions. Our approach makes as few assumptions on the system as possible. In particular, for the designers to specify an application's privacy policies (i.e., to define a privacy model), they must know the structure of the application's state (i.e., the data model) and the application's interface methods (i.e., the domain of the annotate function). Having a data model of an application at a design phase is a realistic assumption.

Our approach relies on application *designers* (i.e., software, security, and privacy engineers) to correctly specify the application's well-formed states and its security and privacy requirements. The problem of enforcing these requirements becomes extremely challenging if application designers or developers are malicious and requires some form of trusted platform technology to ensure that a correct version of enforcement mechanism code runs [56]. We therefore consider application designers to be trustworthy. Additional design validation techniques, like model checking or manual model inspection, can help justify this assumption.

In contrast, the application *developers*, while not malicious, may unintentionally deviate from the design by making implementation errors. Our methodology and tools aid well-intentioned developers in systematically and consistently enforcing security and privacy policies. As these policies have a cross-cutting effect on system execution, generating enforcement mechanism code and instrumenting relevant actions effectively connects privacy-relevant design and implementation artifacts thereby enabling privacy-by-design.

Overall, our approach helps organizations mitigate the risk of GDPR violations caused by accidental implementation errors.

8 EVALUATION

We have evaluated our model-driven approach using the generators described in Section 7. In particular, in our evaluation aims to answer the following groups of research questions (RQ):

- RQ1: Does our approach generalize across different implementation technologies and application domains?
- RQ2: How much developer effort is required to use our model-driven approach compared to a manual implementation? What is the ratio between the generated and manually written code?
- RQ3: How much runtime overhead does our approach incur? How does it compare to manual implementation and to state-of-the-art approaches? How does it scale as the size of an application's input workload and state grow?

We answer the above questions by implementing three realistic applications as case studies from different business domains (RQ1): (i) a social networking site `MiniTwit`, (ii) a conference management system `ConfMS`, and (iii) a health record manager `Hipaa`. Each application is implemented multiple times. `ConfMS` is implemented using both of our model transformations targeting different implementation technologies (RQ1). To assess the development effort (RQ2), `MiniTwit` is implemented manually twice, in addition to using our model transformation targeting Python. The first manual implementation is the original `MiniTwit` implementation [61] that does not enforce any security or privacy policy. This baseline implementation allows us to assess the (worst case) runtime overhead of our generated enforcement mechanisms (RQ3). To the best of our knowledge, no state-of-the-art approach automatically enforces privacy policies. We still compare the overhead of our approach to a state-of-the-art framework Jacqueline [64] for security policies (RQ3). The `Hipaa` application was chosen as it has an independent implementation in Jacqueline.

Table 1: Summary of the case study implementations

Name	Description	Application	Developers
`MiniTwit-baseline`	Baseline	`MiniTwit`	[61], Python
`MiniTwit-secured`	Secure, manual	`MiniTwit`	Us, Python
`MiniTwit-flask`	Secure, generated	`MiniTwit`	Us, Python
`ConfMS-flask`	Secure, generated	`ConfMS`	Us, Python
`ConfMS-asp`	Secure, generated	`ConfMS`	Us, C#
`Hipaa-jacqueline`	Secure, Jacqueline	`Hipaa`	[64], Python
`Hipaa-flask`	Secure, generated	`Hipaa`	Us, Python

Table 1 summarizes the names, descriptions, and developers of all the implementations of the three applications. The code and the deployment instructions for each of them can be found under the respective subdirectory in our publicly available artifact [45].

We now describe each application in detail and afterwards we present our evaluation results.

8.1 Case studies

Social networking site. MiniTwit is a Twitter clone with an open-source implementation [61]. In short, registered users can write new messages, follow or unfollow users, and view their own timeline. Moreover, to assess privacy policy enforcement, we extend its functionality to support in-application advertisements displayed on users' timelines.

We denote this implementation as `MiniTwit-baseline` as it forgoes security and privacy policy enforcement and implements only `MiniTwit`'s business logic. Next, we introduce MiniTwit's security and privacy policies. We denote by `MiniTwit-secured` and `MiniTwit-flask` the two MiniTwit implementations that enforce the security and privacy policies. The former extends the baseline application with a manual implementation that directly enforces the policies, while the latter follows our approach using our Python model transformations (Table 1).

The `MiniTwit` application has only one role, the registered user RegUser and the following FGAC policies:

(AC1) *Users can create new messages*, modeled as the permission assignment tuple (RegUser, create(Message), $\top$).

(AC2) *Users can initialize themselves as an author of the message and update the its publication date and text*, modeled as
(RegUser, update(Message, author), φ),
(RegUser, update(Message, pub_date), **self**.author = caller),
(RegUser, update(Message, text), **self**.author = caller).

(AC3) *Users can follow and unfollow others*, modeled as
(RegUser, add(User, follows), **self** = caller),
(RegUser, add(User, followers), value = caller),
(RegUser, remove(User, follows), **self** = caller),
(RegUser, remove(User, followers), value = caller).

(AC4) *Users can read messages (including its author, publication date, and text) posted by themselves, or by users that they follow*, modeled as
(RegUser, read(Message, author), **self**.author = caller),
(RegUser, read(Message, pub_date), **self**.author = caller),
(RegUser, read(Message, text), **self**.author = caller),
(RegUser, read(User, follows), **self** = caller).

Here φ is (**self**.author.**oclIsUndefined**() and value = caller).

We define personal data sorts $CS^{\mathcal{P}}$ as {User, Follow}, i.e., the set of users's basic information (like their gender and age) and the set of users they follow. The application has two purposes $\mathcal{P}$ = {GenerateAds, DisplayPosts} and the following privacy policy:

(DP1) *We will read your basic information, namely your age and gender, to generate and display advertisements that you may find interesting*, modeled as the declared purpose tuples
(GenerateAds, read(User, age), $\top$),
(GenerateAds, read(User, gender), $\top$).

(DP2) *We will read your followers to populate your timeline with posts*, modeled as (DisplayPosts, read(User, follows), $\top$).

The following `JSON` snippet shows the definition of the security policy for updating a message's text (as in AC2), and the privacy policy for reading the user's age for the purpose of generating advertisements (as in DP1). The concrete syntax used to define all our models is `JSON`.

```
% Security policy, permission assignment from AC2
{
  "role": "RegUser",
  "action": "update",
  "resource": {
    "class": "Message",
    "attribute": "text"
  },
  "constraint": "self.author = caller"
}

% Privacy policy, declared purpose from DP1
{
  "purpose": "GenerateRelevantMarketingEntities",
  "action": "read",
  "resources": [
    {
      "class": "User",
      "attribute": "age"
    }
  ],
  "constraint": {
    "ocl": "true",
    "desc": "true"
  }
}
```

Observe that the `JSON` input to our model transformations is isomorphic to the model definitions presented in Sections 3 and 4.

Conference management system. `ConfMS` is an extension of our running example in this paper. As described in Example 1, a registered user in this application can have either a normal role, or be on the program committee, or be the chair. Normal users can edit their profile information, search and view accepted papers, and submit new papers. Program committee users can delegate the review of a paper to some users. In addition, users with the chair role can accept papers for publication.

We define our three models for this conference management system. For space reasons, we omit the details. In a nutshell, the data model consists of 2 classes and 3 associations (Figure 1), the security model consists of 3 roles and 14 FGAC policies, and the privacy model has 3 declared and 3 actual purposes as depicted in

Examples 6, 7, and Figure 3. These models are input into our code-generators to generate C# and Python applications with method stubs. We straightforwardly implement the stubs' business logic following the activities specified in Figure 3. We denote by `ConfMS-asp` and `ConfMS-flask` the name of these two applications (Table 1).

Health record manager. `Hipaa` is a web application that allows users with different roles to view personal medical records. We define three models correspond to the underlying database and implementation of the similar case study conducted in Yang et al. [64] and compare the performance result. Due to space limitations, we only report on the sizes on the models, which are available within our artifact [45]. `Hipaa`'s data model contains 10 classes and 13 associations, its security model contains one FGAC policy, and its privacy model declares one declared and one actual purpose.

We denote by `Hipaa-jacqueline` the name of the HRM application implemented by Yang et al. [64] and `Hipaa-flask` the name of the Python application that is generated.

8.2 RQ1: Generality

To answer RQ1, we have showcased in Section 7 that our model-driven approach can target multiple different programming languages, namely C# and Python. Furthermore, we claim that any language can be targeted. Languages supporting the implementation of cross-cutting concerns like AOP [33, 34] or function decorators [42] are particularly convenient for implementing the model transformations, and such transformations produce code that is easy to maintain and evolve. Existing support for authentication and authorization also helps in this respect as the relevant libraries can be directly targeted by model transformations. Finally, by implementing the three case study applications, we show that our approach can be used in different application domains.

8.3 RQ2: Development effort

To answer RQ2, Table 2 provides statistics on the different implementations of the `ConfMS` application. At the top, the table shows the size of our three models of the `ConfMS` application. A data model's size is determined by summing the number of defined sorts, attributes, methods, and associations. A security model's size is the size of its permission assignment relation, whereas a privacy model's size is the sum of the number of declared and actual purposes. Below the model sizes, our table shows the number of generated lines of code (# LoC) for `ConfMS-flask` (Python and HTML code) and `ConfMS-asp` (C# and HTML code) implementations, as well as the lines of code needed to implement the method stubs manually. Developers need to write only 16% of the overall codebase manually in Python and 15% in C# to obtain functional web applications. The absolute difference in the number of lines of code written manually is only 202, which makes the manual effort across different technologies comparable. The difference in the ratios of manual and generated code between the two technologies is due to the Model-View-Control architecture of ASP.NET web applications, which significantly increases the codebase. We do not report numbers for the generated authentication mechanism as it is handled by the respective libraries that require only a few configuration parameters. The most significant part of the generated code is for the enforcement mechanisms: 21% of the total Python code and 30% of total C# code.

Table 2: # LoC of the ConfMS (EM = enforcement mechanism)

`ConfMS` implementations	# LoC		Total
	.json	-	
Data model	13	-	13
Security model	14	-	14
Privacy model	6	-	6
	.cs	.cshtml	
`ConfMS-asp`	1842	456	2298
a) Generated artifacts	1662	292	1954
a.1) Classes & method stubs	302	0	302
a.2) Authentication	-	-	-
a.3) EM for Security	672	0	672
a.4) EM for Privacy	10	0	10
a.5) Consent collection	54	137	191
a.6) Purpose tracking	207	0	207
a.7) Others	417	155	572
b) Manually implemented	180	164	344
	.py	.html	
`ConfMS-flask`	689	184	873
a) Generated artifacts	632	99	731
a.1) Classes & method stubs	126	0	126
a.2) Authentication	-	-	-
a.3) EM for Security	120	0	120
a.4) EM for Privacy	61	0	61
a.5) Consent collection	105	32	32
a.6) Purpose tracking	63	0	63
a.7) Others	157	67	224
b) Manually implemented	57	85	142

We also assess the overall effort when developing the `MiniTwit` application manually (`MiniTwit-secured`) compared to using our approach (`MiniTwit-flask`). The `MiniTwit-secured` application consists of 202 lines of Python and 137 lines of HTML code, whereas `MiniTwit-flask` is just 172 lines of Python and 119 lines of HTML code. The remaining `MiniTwit-flask` code is generated from models with a total size of 29 (its data model size is 15, security model size is 12, and privacy model size is 2). This provides evidence that the combined effort of the designers and developers using our approach is significantly lower than the effort of developers manually implementing the application. Furthermore, when the security or privacy policies change, our model transformation would regenerate correct enforcement mechanisms automatically, whereas a manually implemented application may require non-trivial changes.

8.4 RQ3: Runtime overhead and scalability

We deploy the three versions of the `MiniTwit` application and execute them to measure the overhead incurred by enforcing security and privacy requirements using our methodology. We then deploy the `Hipaa` implementations to compare our approach to the state-of-the-art security web frameworks. We measure the time taken between sending a request to the web application and receiving a response back. The execution time is averaged over 10 executions.

In particular, as input to the applications we use the workload where a registered user logs in and then views their own timeline (by invoking the `public_timeline` method). Each `MiniTwit` instance

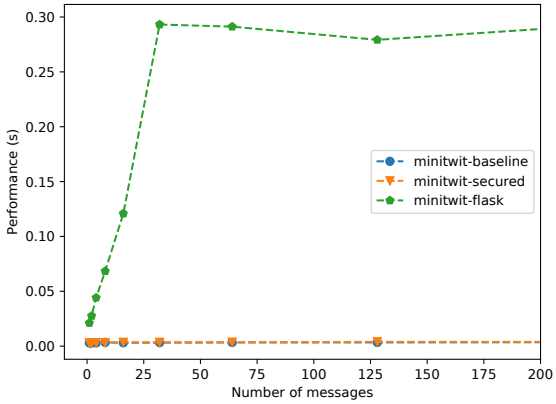

Figure 5: Performance of different `MiniTwit` implementations

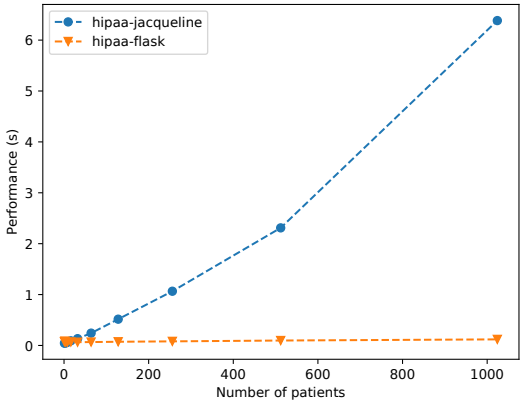

Figure 6: Performance of different `Hipaa` implementations

is deployed with 4 users and a variable number of messages in its state. Message content is randomly sampled, and its assignment to a user is randomly determined. One of the users is set to follow all the other users and we generate the workload on behalf of that user. The user's followers are considered personal data. By controlling the number of messages present in the state of the application we can assess our approach's scalability.

Figure 5 shows the execution time of the three `MiniTwit` implementations on the above workload. All the implementations scale linearly with the number of messages. Our approach clearly adds overhead as its linear factor is larger than both the baseline and the manually secured implementation. The baseline implementation does not implement any policy checks hence its execution time is the lower bound for the other two implementations. The manually secured implementation is more efficient as it performs only two checks (one for security and one for privacy policy compliance) before outputting the list of messages to the public timeline. This is prone to errors as it assumes that the subsequent code correctly queries and outputs the right messages. In contrast, our approach performs both checks whenever a message is accessed within the `public_timeline` method's implementation, ensuring that the developer only manipulates data in compliance with the policies.

Although the added overhead is comparatively high, it is unnoticeable from a user's perspective. In particular, our implementation's execution times are below one second for any number of messages. This is ensured as `MiniTwit` implements a 30-item pagination for all lists shown to the user. Therefore, our approach's performance stabilizes below 0.3 seconds for all executions involving more than 30 messages in the application's state.

When running the `Hipaa` implementations (`hipaa-jacqueline` and `hipaa-flask`), we initialize their state with a single doctor and variable number of randomly sampled patients. We then run a workload where the doctor logs in and opens the initial page of the application (by invoking the `index` method), which shows all their patients and the hospitals where they work. In this case, only the patient information is considered personal data.

Figure 6 shows the execution time of the two `Hipaa` implementations on the above workload. While our approach still scales linearly in the number of patients, Jacqueline's performance is superlinear. Both approaches perform well enough to be used with a 100-item pagination, which the `Hipaa` application does not implement. The significant difference in the performance between the two approaches is caused by the performance-intensive faceted execution [12] employed by Jacqueline. It is used to enforce stronger information-flow security policies.

9 CONCLUSIONS

In data protection, a privacy policy is an informal legal document describing how a system gathers, uses, discloses, and manages personal data. We have introduced the well-known and widely-mandated classes of privacy policies involving purpose-limitation and consent into the system design phase. Namely, we propose a privacy model that formalizes these classes of privacy policies. We define the model's semantics and implement model transformations that can generate web applications in C# and Python with complete, configured, enforcement infrastructure for their specified privacy polices. By generating this (often critical) enforcement code, our approach reduces the risk of privacy policy violations due to implementation errors. We evaluate our approach and demonstrate its broad scope and applicability, as well as its modest overhead.

In the future, we plan to extend our methodology to incorporate other data protection requirements such as data minimization (Art. 5 §1 (c) GDPR), storage limitation (Art. 5 §1 (e) GDPR), and accountability (Art. 5 §2 GDPR). We also plan extend our methodology to enforce information-flow security policies on particularly sensitive classes of personal data to prevent inference attacks. Models that have formal semantics enable the analysis of both the models themselves and the transformation functions. We plan to carry out automatic property checking of privacy models to detect and correct design errors and provide feedback to developers during method stub implementation. We also plan to fully verify the correctness of our model transformation functions.

ACKNOWLEDGMENTS

Hoang Nguyen is supported by the Swiss National Science Foundation grant "Model-driven Security & Privacy" (204796). We thank François Hublet and the anonymous reviewers on their suggestions to improve this paper.

REFERENCES

[1] 2018. Flask-SQLAlchemy. https://flask-sqlalchemy.palletsprojects.com/en/2.x/.
[2] 2018. Flask-User. http://flask-user.readthedocs.io/.
[3] 2021. MS Entity Framework Core. https://learn.microsoft.com/en-us/ef/core/.
[4] 2022. MS ASP.NET Core Identity. https://learn.microsoft.com/en-us/aspnet/identity/.
[5] Amir Shayan Ahmadian, Daniel Strüber, and Jan Jürjens. 2019. Privacy-enhanced system design modeling based on privacy features. In *34th ACM/SIGAPP Symposium on Applied Computing, (SAC)*, Chih-Cheng Hung and George A. Papadopoulos (Eds.). ACM, 1492–1499. https://doi.org/10.1145/3297280.3297431
[6] Atheer Aljeraisy, Masoud Barati, Omer F. Rana, and Charith Perera. 2021. Privacy Laws and Privacy by Design Schemes for the Internet of Things: A Developer's Perspective. *ACM Comput. Surv.* 54, 5 (2021), 102:1–102:38. https://doi.org/10.1145/3450965
[7] Thibaud Antignac and Daniel Le Métayer. 2014. Privacy Architectures: Reasoning about Data Minimisation and Integrity. In *Security and Trust Management (STM) (LNCS, Vol. 8743)*, Sjouke Mauw and Christian Damsgaard Jensen (Eds.). Springer, 17–32. https://doi.org/10.1007/978-3-319-11851-2_2
[8] Thibaud Antignac, Riccardo Scandariato, and Gerardo Schneider. 2018. Privacy Compliance Via Model Transformations. In *IEEE European Symposium on Security and Privacy Workshops, (EuroS&P Workshops)*. IEEE, 120–126. https://doi.org/10.1109/EuroSPW.2018.00024
[9] Claudio A Ardagna, Laurent Bussard, Sabrina De Capitani di Vimercati, Gregory Neven, Eros Pedrini, Stefano Paraboschi, Franz-Stefan Preiss, Pierangela Samarati, Slim Trabelsi, and Mario Verdicchio. 2009. Primelife policy language. In *W3C Workshop on Access Control Application Scenarios*. W3C.
[10] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. 2003. Enterprise privacy authorization language (EPAL). *IBM Research* 30 (2003), 31. https://www.w3.org/2003/p3p-ws/pp/ibm3.html.
[11] Colin Atkinson and Thomas Kühne. 2003. Model-Driven Development: A Metamodeling Foundation. *IEEE Softw.* 20, 5 (2003), 36–41. https://doi.org/10.1109/MS.2003.1231149
[12] Thomas H Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted execution of policy-agnostic programs. In *8th ACM SIGPLAN workshop on Programming languages and analysis for security*. 15–26.
[13] Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, Karin Bernsmed, Anderson Santana De Oliveira, and Jakub Sendor. 2014. A-PPL: an accountability policy language. In *Data privacy management, autonomous spontaneous security, and security assurance*. Springer, 319–326.
[14] Chinmayi Prabhu Baramashetru, Silvia Lizeth Tapia Tarifa, Olaf Owe, and Nils Gruschka. 2022. A Policy Language to Capture Compliance of Data Protection Requirements. In *17th International Conference on Integrated Formal Methods (IFM) (LNCS, Vol. 13274)*, Maurice H. ter Beek and Rosemary Monahan (Eds.). Springer, 289–309. https://doi.org/10.1007/978-3-031-07727-2_16
[15] David Basin, Søren Debois, and Thomas Hildebrandt. 2018. On Purpose and by Necessity: Compliance Under the GDPR. In *22nd International Conference on Financial Cryptography and Data Security (FC)*, Sarah Meiklejohn and Kazue Sako (Eds.). 20–37. https://doi.org/10.1007/978-3-662-58387-6_2
[16] David Basin, Jürgen Doser, and Torsten Lodderstedt. 2006. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.* 15, 1 (2006), 39–91. https://doi.org/10.1145/1125808.1125810
[17] David Basin, Juan Guarnizo, Srđan Krstić, Hoang Nguyen, and Martín Ochoa. 2023. Is Modeling Access Control Worth It?. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Weizhi Meng, Christian D. Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM. To appear.
[18] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. 2006. Model Transformations? Transformation Models!. In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS) (LNCS, Vol. 4199)*, Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (Eds.). Springer, 440–453. https://doi.org/10.1007/11880240_31
[19] Abhishek Bichhawat, Akash Trehan, Jean Yang, and Matt Fredrikson. 2018. ESTRELA: Automated Policy Enforcement Across Remote APIs. *CoRR* abs/1811.08234 (2018). arXiv:1811.08234 http://arxiv.org/abs/1811.08234
[20] Alan W. Brown. 2004. Model driven architecture: Principles and practice. *Softw. Syst. Model.* 3, 4 (2004), 314–327. https://doi.org/10.1007/s10270-004-0061-2
[21] Ji-Won Byun, Elisa Bertino, and Ninghui Li. 2005. Purpose based access control of complex data for privacy protection. In *10th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Elena Ferrari and Gail-Joon Ahn (Eds.). ACM, 102–110. https://doi.org/10.1145/1063979.1063998
[22] Ji-Won Byun and Ninghui Li. 2008. Purpose based access control for privacy protection in relational database systems. *VLDB J.* 17, 4 (2008), 603–619. https://doi.org/10.1007/s00778-006-0023-0
[23] Ann Cavoukian. 2009. Privacy by design. (2009).
[24] CCPA 2018. California Consumer Privacy Act. https://oag.ca.gov/privacy/ccpa.
[25] Omar Chowdhury, Limin Jia, Deepak Garg, and Anupam Datta. 2014. Temporal Mode-Checking for Runtime Monitoring of Privacy Policies. In *26th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 131–149. https://doi.org/10.1007/978-3-319-08867-9_9
[26] Lorrie Cranor. 2002. *Web privacy with P3P - the platform for privacy preferences*. O'Reilly. http://www.oreilly.de/catalog/webprivp3p/index.html
[27] Lorrie Cranor, Marc Langheinrich, Massimo Marchiori, and Joseph Reagle. 2002. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. https://www.w3.org/TR/P3P/. Obsoleted on 30 August 2018.
[28] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. 2001. The Ponder Policy Specification Language. In *International Workshop on Policies for Distributed Systems and Networks (POLICY) (LNCS, Vol. 1995)*, Morris Sloman, Jorge Lobo, and Emil Lupu (Eds.). Springer, 18–38. https://doi.org/10.1007/3-540-44569-2_2
[29] DCIA 2022. Digital Charter Implementation Act. https://www.ourcommons.ca/DocumentViewer/en/44-1/house/sitting-90/hansard.
[30] Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kirli Kaynar, and Anupam Datta. 2010. Experiences in the logical specification of the HIPAA and GLBA privacy laws. In *Workshop on Privacy in the Electronic Society (WPES)*, Ehab Al-Shaer and Keith B. Frikken (Eds.). ACM, 73–82. https://doi.org/10.1145/1866919.1866930
[31] África Domingo, Jorge Echeverría, Oscar Pastor, and Carlos Cetina. 2020. Evaluating the Benefits of Model-Driven Development - Empirical Evaluation Paper. In *32nd International Conference on Advanced Information Systems Engineering (CAiSE) (LNCS, Vol. 12127)*, Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant (Eds.). Springer, 353–367. https://doi.org/10.1007/978-3-030-49435-3_22
[32] European Parliament and Council of the European Union. 2016. General Data Protection Regulation, (EU) 2016/679. https://data.europa.eu/eli/reg/2016/679/oj
[33] Eclipse Foundation. 2021. Eclipse AspectJ. https://www.eclipse.org/aspectj/.
[34] Andreas Frömer, Alexander Lisachenko, and Kirill Nesmeyanov. 2021. Go! Aspect-Oriented Framework for PHP. https://github.com/goaop/framework.
[35] GLBA 1999. Gramm-Leach-Bliley Act. https://www.ftc.gov/legal-library/browse/statutes/gramm-leach-bliley-act.
[36] Hassan Gomaa. 2011. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press.
[37] Object Management Group. 2014. Object Constraint Language 2.4 (OCL2.4) Specification. https://www.omg.org/spec/OCL/2.4/.
[38] Michele Guerriero, Damian Andrew Tamburri, and Elisabetta Di Nitto. 2018. Defining, enforcing and checking privacy policies in data-intensive applications. In *International Conference on Software Engineering for Adaptive and Self-Managing Systems, (SEAMS)*, Jesper Andersson and Danny Weyns (Eds.). ACM, 172–182. https://doi.org/10.1145/3194133.3194140
[39] Seda Gürses, Carmela Troncoso, and Claudia Diaz. 2011. Engineering privacy by design. *Computers, Privacy & Data Protection* 14, 3 (2011), 25.
[40] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. 1976. Protection in Operating Systems. *Commun. ACM* 19, 8 (1976), 461–471. https://doi.org/10.1145/360303.360333
[41] HIPAA 1996. Health Insurance Portability and Accountability Act. https://www.hhs.gov/hipaa/.
[42] John Hunt and John Hunt. 2019. Decorators. *A Beginners Guide to Python 3 Programming* (2019), 337–351.
[43] Farzane Karami, David Basin, and Einar Broch Johnsen. 2022. DPL: A Language for GDPR Enforcement. In *35th IEEE Computer Security Foundations Symposium, (CSF)*. IEEE, 112–129. https://doi.org/10.1109/CSF54842.2022.9919687
[44] Anneke Kleppe, Jos Warmer, and Wim Bast. 2003. *MDA explained - the Model Driven Architecture: practice and promise*. Addison-Wesley. http://www.informit.com/store/mda-explained-the-model-driven-architecture-practice-9780321194428
[45] Srđan Krstić, Hoang Nguyen, and David Basin. 2024. Model-driven Privacy: Implementation and evaluation artifacts. https://anonymous.4open.science/r/model-driven-privacy-2100/.
[46] Miroslaw Kutylowski, Anna Lauks-Dutka, and Moti Yung. 2020. GDPR - Challenges for Reconciling Legal Rules with Technical Reality. In *25th European Symposium on Research in Computer Security, (ESORICS) (LNCS, Vol. 12308)*, Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider (Eds.). Springer, 736–755. https://doi.org/10.1007/978-3-030-58951-6_36
[47] Charlie Lai, Li Gong, Larry Koved, Anthony J. Nadalin, and Roland Schemers. 1999. User Authentication and Authorization in the Java(tm) Platform. In *15th Conference on Computer Security Applications (ACSAC)*. IEEE Computer Society, 285–290. https://doi.org/10.1109/CSAC.1999.816038

[48] Peifung E. Lam, John C. Mitchell, and Sharada Sundaram. 2009. A Formalization of HIPAA for a Medical Messaging System. In *6th International Conference on Trust, Privacy and Security in Digital Business (TrustBus) (LNCS, Vol. 5695)*, Simone Fischer-Hübner, Costas Lambrinoudakis, and Günther Pernul (Eds.). Springer, 73–85. https://doi.org/10.1007/978-3-642-03748-1_8

[49] Jens Leicht and Maritta Heisel. 2019. A survey on privacy policy languages: Expressiveness concerning data protection regulations. In *12th CMI Conference on Cybersecurity and Privacy (CMI)*. IEEE, 1–6.

[50] Thomas Linden, Rishabh Khandelwal, Hamza Harkous, and Kassem Fawaz. 2020. The Privacy Policy Landscape After the GDPR. *Proc. Priv. Enhancing Technol.* 2020, 1 (2020), 47–64. https://doi.org/10.2478/popets-2020-0004

[51] Torsten Lodderstedt, David Basin, and Jürgen Doser. 2002. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *5th International Conference on The Unified Modeling Language (UML) (LNCS, Vol. 2460)*, Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook (Eds.). Springer, 426–441. https://doi.org/10.1007/3-540-45800-X_33

[52] Michael J. May, Carl A. Gunter, and Insup Lee. 2006. Privacy APIs: Access Control Techniques to Analyze and Verify Legal Privacy Policies. In *19th IEEE Computer Security Foundations Workshop, (CSFW)*. IEEE Computer Society, 85–97. https://doi.org/10.1109/CSFW.2006.24

[53] Elliott Mendelson. 1987. *Introduction to mathematical logic (3. ed.)*. Chapman and Hall.

[54] Raúl Pardo and Daniel Le Métayer. 2019. Analysis of Privacy Policies to Enhance Informed Consent. In *33rd Conference on Data and Applications Security and Privacy (DBSec) (LNCS, Vol. 11559)*, Simon N. Foley (Ed.). Springer, 177–198. https://doi.org/10.1007/978-3-030-22479-0_10

[55] PostSharp. 2023. PostSharp Documentation. https://doc.postsharp.net/.

[56] Alexander Pretschner, Manuel Hilty, and David Basin. 2006. Distributed usage control. *Commun. ACM* 49, 9 (2006), 39–44. https://doi.org/10.1145/1151030.1151053

[57] Óscar Sánchez Ramón, Fernando Molina, Jesús García Molina, and José Ambrosio Toval Álvarez. 2009. ModelSec: A Generative Architecture for Model-Driven Security. *J. Univers. Comput. Sci.* 15, 15 (2009), 2957–2980. https://doi.org/10.3217/jucs-015-15-2957

[58] Mark Richters and Martin Gogolla. 2002. OCL: Syntax, Semantics, and Tools. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, Tony Clark and Jos Warmer (Eds.). LNCS, Vol. 2263. Springer, 42–68. https://doi.org/10.1007/3-540-45669-4_4

[59] Livio Robaldo, Cesare Bartolini, Monica Palmirani, Arianna Rossi, Michele Martoni, and Gabriele Lenzini. 2020. Formalizing GDPR Provisions in Reified I/O Logic: The DAPRECO Knowledge Base. *J. Log. Lang. Inf.* 29, 4 (2020), 401–449. https://doi.org/10.1007/s10849-019-09309-z

[60] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50. https://doi.org/10.1145/353323.353382

[61] Mark Sidell, David Kavanagh, and Stephan Kemper. 2022. MiniTwit: A tiny sample Flask app featuring a database and user accounts. https://github.com/Viasat/minitwit.

[62] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 615–630. https://www.usenix.org/conference/nsdi19/presentation/wang-frank

[63] Wikipedia, The Free Encyclopedia. 2023. Privacy policy. https://en.wikipedia.org/wiki/Privacy_policy [Accessed 7-Aug-2023].

[64] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, dynamic information flow for database-backed applications. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 631–647. https://doi.org/10.1145/2908080.2908098

[65] Jie Zhou, Qingguo Shen, and Yudong Xu. 2012. Research and improvement of Ponder2 policy language. In *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, Vol. 2. IEEE, 455–458.

A OBJECT CONSTRAINT LANGUAGE

The object constraint language (OCL) [58] is a formal language used to query object models or describe properties (invariants) of data models, e.g., in order to constrain their possible object models. OCL resembles an order-sorted first-order logic (FOL) with 4 logical values. Here we focus on its 2-valued variant for simplicity.

OCL is a pure language: OCL expressions do not have side effects, i.e., the object model does not change as a consequence of evaluating OCL expressions. Although OCL expressions can be used to specify many aspects, here we focus on sort invariants, where such OCL expressions are written in the context of an object of a specific sort (referenced using the **self** OCL expression).

Syntax. Let a family of sets of variables V_s indexed by a sort $s \in \mathsf{S}$ be given. The following grammar θ_s^Σ defines OCL expressions that evaluate to an element of type s over the signature Σ (possibly) extended with nullary function symbols representing constants. Below we fix the signature Σ and write just θ_s.

$$
\begin{array}{lll}
\theta_s ::= \mathbf{self} & & \\
\mid x_s & & \text{if } x_s \in V_s \\
\mid \omega(\theta_{s'_1}, \ldots, \theta_{s'_n}) & & \text{if } \omega : s_1 \times \ldots \times s_n \to s \text{ and } s'_i \preceq s_i \\
\mid \mathbf{if}\ \theta_\mathsf{B}\ \mathbf{then}\ \theta_t\ \mathbf{else}\ \theta_e\ \mathbf{fi} & & \text{if } t \preceq s \text{ and } e \preceq s \\
\mid \theta_{s'}\ \mathbf{as}\ s & & \text{if } s \preceq s' \\
\mid \theta_{\mathsf{Collection}(s')} \to \mathbf{iterate}(x_{s'},\ x_s : \theta_s \mid \theta_s) & & \text{if } x_s \in V_s \text{ and } x_{s'} \in V_{s'}
\end{array}
$$

Intuitively, the expression **self** refers to the object of sort s that is the context in which the OCL expression was written. Variables $x_s \in V_s$ stand for some element of the sort s. Function application applies the function symbol ω to a tuple of OCL expressions $\phi_1, \ldots, \phi_n$ with appropriate types. If ϕ_1 evaluates to a collection sort, then OCL allows $\phi_1 \to \omega(\phi_2, \ldots, \phi_n)$ as a syntactic sugar, otherwise $\phi_1.\omega(\phi_2, \ldots, \phi_n)$ can be used as syntactic sugar. The **if-then-else** expression evaluates to either of the two alternative expressions of a subsort of s depending on the evaluation of the OCL constraint θ_B. The **as** expression casts an expression of sort s' to s, a subsort of s'. Finally, the **iterate** expression is a *fold* on collection sorts, where the variable $x_{s'}$ binds to every element of the collection, while variable x_s serves as an accumulator that is initially set to the value of the first OCL expression θ_s and then iteratively updated for each element of the collection, based on the second OCL expression θ_s.

Recall that $\mathsf{fv}(\phi) \subseteq \bigcup_{s \in \mathsf{S}} V_s$ is the set of free variables of the OCL expression ϕ, each indexed by its sort. For $x_s \in \mathsf{fv}(\phi)$ and an OCL expression $t \in \theta_s$, $\phi[x_s/t]$ is the OCL expression obtained from ϕ by substituting all instances of x_s with t in a capture-avoiding way.

Example 10. *Consider the OCL constraint from Example 4 over the signature Σ extended with the constant 0 of type $() \to \mathsf{Z}$. It is desugared to the expression $>(\mathsf{size}(\mathsf{name}(\mathbf{self})), 0)$ defined by the above grammar.*

Semantics. Let σ be the structure induced by some object model interpreting the signature Σ. The valuation $v : V_s \to [\![s]\!]$ is a family of variable assignments, each for variables of the appropriate sort. The semantics of an OCL expression $\phi \in \theta_s$ is defined with respect to a valuation v and a context object c of the type $C \in \mathsf{S}$ using a function $\mathsf{I} : \theta_s \to V \times C \to [\![s]\!]$, where V is a set of all valuations.

$$
\begin{array}{rl}
\mathsf{I}(\mathbf{self})(v,c) = & c \\
\mathsf{I}(x)(v,c) = & v(x) \\
\mathsf{I}(\omega(\phi_1, \ldots, \phi_n))(v,c) = & \{\{\omega\}\}(\mathsf{I}(\phi_1)(v,c), \ldots, \mathsf{I}(\phi_n)(v,c)) \\
\mathsf{I}(\mathbf{if}\ \phi\ \mathbf{then}\ \phi_1\ \mathbf{else}\ \phi_2\ \mathbf{fi})(v,c) = & \text{if } \mathsf{I}(\phi)(v,c) = \top \\
& \quad \text{then } \mathsf{I}(\phi_1)(v,c) \\
& \quad \text{else } \mathsf{I}(\phi_2)(v,c) \\
\mathsf{I}(\phi\ \mathbf{as}\ s)(v,c) = & \mathsf{I}(\phi)(v,c) \text{ if } \mathsf{I}(\phi)(v,c) \in [\![s]\!] \\
\mathsf{I}(\phi \to \mathbf{iterate}(e,\ a : \phi_a \mid \phi_u))(v,c) = & \mathsf{I}(\mathsf{fold}(\phi,\ \phi_u))(v[a/\mathsf{I}(\phi_a)(v,c)], c) \\
\text{where } \mathsf{I}(\mathsf{fold}(\emptyset,\ \phi_u))(v,c) = & \mathsf{I}(a)(v,c) \\
\mathsf{I}(\mathsf{fold}(f \cdot \phi,\ \phi_u))(v,c) = & \mathsf{I}(\mathsf{fold}(\phi,\ \phi_u))(v[a/\mathsf{I}(\phi_u)(v[e/f], c)], c)
\end{array}
$$

Other more commonly used OCL expressions can be derived using the **iterate** expression. For instance, the following are the expressions the we use in this paper:

$$\begin{aligned}
\phi\rightarrow\textbf{forall}(x \mid \phi') &:= \phi\rightarrow\textbf{iterate}(x,\ a:\top \mid a \text{ and } \phi'),\\
\phi\rightarrow\textbf{exists}(x \mid \phi') &:= \phi\rightarrow\textbf{iterate}(x,\ a:\bot \mid a \text{ or } \phi'),\\
\phi\rightarrow\textbf{excludes}(x) &:= \phi\rightarrow\textbf{forall}(x' \mid \text{not } x = x'),\\
\phi\rightarrow\textbf{size}() &:= \phi\rightarrow\textbf{iterate}(x,\ a:0 \mid a\ +\ 1).
\end{aligned}$$

B SYSTEM SEMANTICS

We model any system as a labeled transition system (LTS) $\Delta = (Q, \mathcal{A}', \delta)$, where the set of nodes Q contains all σ structures (Section 6), and edges are labeled by the actions $\mathcal{A}$ (Section 3) extended with the actions corresponding to the method returns. In particular, edges are labeled with actions $\mathcal{A}'$ defined by $\mathcal{A} \cup \{\mathsf{return}(C,n) \mid n \in \mathsf{dom}(\mathsf{meth}(C)),\ C \in \mathsf{CS}\}$. The transition relation $\delta \subseteq Q \times \mathcal{A}' \times Q$ allows only authorized actions and relates two structures based on the action and is defined as follows:

$$\begin{aligned}
&\{(q, \mathsf{execute}(C,n), q') \mid q' = q[\langle\!\langle \mathsf{CAP} \rangle\!\rangle \mapsto \langle\!\langle \mathsf{CAP} \rangle\!\rangle \cup \mathsf{annotate}(n)] \\
&\qquad\qquad \text{if } q \models \varphi_s \wedge \varphi_p\} \cup\\
&\{(q, \mathsf{return}(C,n), q') \mid q' = q[\langle\!\langle \mathsf{CAP} \rangle\!\rangle \mapsto \langle\!\langle \mathsf{CAP} \rangle\!\rangle \setminus \mathsf{annotate}(n)]\} \cup\\
&\{(q, \mathsf{read}(C,n), q) \mid q' = q \text{ if } q \models \varphi_s \wedge \varphi_p\} \cup\\
&\{(q, \mathsf{update}(C,n), q) \mid q' = q[\{\!\{n\}\!\} \mapsto \{\!\{n\}\!\}[c \mapsto v]] \text{ if } q \models \varphi_s \wedge \varphi_p\} \cup\\
&\{(q, \mathsf{add}(C,n), q) \mid q' = q[\langle\!\langle r \rangle\!\rangle \mapsto \langle\!\langle r \rangle\!\rangle \cup (c,v)] \text{ if } q \models \varphi_s \wedge \varphi_p\} \cup\\
&\{(q, \mathsf{remove}(C,n), q) \mid q' = q[\langle\!\langle r \rangle\!\rangle \mapsto \langle\!\langle r \rangle\!\rangle \setminus (c,v)] \text{ if } q \models \varphi_s \wedge \varphi_p\} \cup\\
&\{(q, \mathsf{create}(C,n), q) \mid q' = q[[\![C]\!] \mapsto [\![C]\!] \cup c'] \text{ if } q \models \varphi_s \wedge \varphi_p\} \cup\\
&\{(q, \mathsf{delete}(C,n), q) \mid q' = q[[\![C]\!] \mapsto [\![C]\!] \setminus c] \text{ if } q \models \varphi_s \wedge \varphi_p\}.
\end{aligned}$$

Here $c \in [\![C]\!]$ is the object on which the action is executed, and v is the value assigned to an attribute or added to (or removed from) an association r.

Intuitively, whenever an authorized method is executed, its actual purpose is added to the set of current actual purposes. Once the method returns, the actual purpose is removed. Read actions label loop edges of the nodes where authorization constraints and privacy policies hold. An authorized update action labels an edge from an old structure to a new structure, which is the same as the old structure except that the new updated value is assigned to the attribute. Similarly for an add (remove) action where the pair of objects is added (removed) from the appropriate association. Create and delete actions add and remove objects from the structure, respectively.