A Formal Framework for Consent Management

Shukun Tokas(✉) and Olaf Owe(✉)

Department of Informatics, University of Oslo, Oslo, Norway
{shukunt,olaf}@ifi.uio.no

Abstract. The aim of this work is to design a formal framework for consent management in line with EU's General Data Protection Regulation (GDPR). To make a general solution, we consider a high-level modeling language for distributed service-oriented systems, building on the paradigm of active objects. Our framework provides a general solution for data subjects to observe and change their privacy settings and to be informed about all personal data stored about them. The solution consists of a set of predefined types for privacy related concepts, a formalization of policy compliance, a set of interfaces that forms the basis of interaction with external users for consent management, a set of classes that is used in interaction with the runtime system, and a runtime system enforcing the consented policies.

Keywords: GDPR · Data protection · Privacy policies · Policy compliance · Tagging · Runtime enforcement · Consent management

1 Introduction

In response to the emerging privacy concerns, the European Union (EU) has approved the General Data Protection Regulation (GDPR) [1] to strengthen and impose data protection rules across the EU. This regulation requires controllers that process personal data of individuals within EU and EEA, to process personal information in a "lawful, fair, and transparent manner". Article 6 and Article 9 of the regulation [1] provide the criteria for lawful processing, such as consent, fulfillment of contractual obligation, compliance with a legal obligation etc. The regulation (including several other data protection laws) recognises consent as one of the lawful principles for legitimate processing, and Article 7 sets out the conditions for the processing personal data when relying on consent.

A data subject's consent reflects his/her agreements in terms of the processing of personal data. The regulation indicates that the consent must specifically be given for the particular *purpose* of processing. It is also indicated in Recital 43 that the data subject should be given a free choice to accept or deny consent for specific purposes, rather than having one consent for several purposes. In particular our focus is on processing of personal data when *consent* is the legal

Published by Springer Nature Switzerland AG 2020
A. Gotsman and A. Sokolova (Eds.): FORTE 2020, LNCS 12136, pp. 169–186, 2020.
https://doi.org/10.1007/978-3-030-50086-3_10

ground, i.e., processing is valid only if a data subject has given consent for the specific purpose, otherwise processing of the personal data should cease. Moreover, this can be extended to incorporate other applicable legal grounds, such as vital interest, legitimate interest etc, but a discussion on this will be out of scope of this work.

Furthermore, Article 15 of the regulation prescribes that the data subject has *Right of Access*, which requires the data controllers to provide the data subject with his/her personal data, the purposes of processing, the legal basis for doing so, and other relevant information (see Article 15 [1]). WP29 recommends controllers to introduce tools, such as a privacy dashboards through which the data subject can be informed and engaged regarding the processing of their personal data [2]. The regulation also introduces an obligation for data controllers to demonstrate compliance, i.e., accountability (see Article 5(2) [1]). These requirements are likely to pose substantial administrative burden. This work is an attempt to design a pragmatic solution to address these requirements, using a formal approach. In particular, our framework covers certain aspects of *privacy principles* (Article 5), *lawfulness of processing* (Article 6), *privacy by design* (Article 25) and *data subject access request* (Article 15). Due to the nature of these requirements and space constraints, we cover these requirements partially.

The privacy requirements in the data protection regulations are defined informally, therefore, to avoid ambiguity the policy language equipped with a formal semantics is essential [3]. It is essential that the policy terminology establishes a clear link between the law and the program artifacts. For this, we let privacy policies and consent specifications be expressed in terms of several predefined names, reflecting standard terminology (allowing names to be added as needed). It is necessary that the policy terminology used towards the data subject is simple but with a formal connection to the underlying programming elements. We have previously studied static aspects of privacy policies and static checking of policy compliance from a formal point of view, a brief overview is given in [4].

The aim of this work is to design a formal framework for consent management where a data subject can change his/her privacy settings through predefined interfaces, which could be part of a library system. The data subjects are seen as external system users without knowledge of the underlying program. Data subjects may interact with the system at runtime through a user-friendly interface (e.g. a privacy dashboard), to view current privacy settings and update these settings. To make a general solution, we consider a high-level modeling language for distributed service-oriented systems, building on the paradigm of *active objects* [5,6]. The method for protecting access to personal data in this setting comprises of: tagging the data with $(subject, purpose)$ pairs; associating a *purpose* to each method accessing personal data; storing consented policies of a subject in a subject object; deriving an *effective* policy for the access from the executing method and data tags; and comparing the effective policy with the current consented policies to determine if it is a valid operation.

The main contribution of this research is a framework that consists of: *(i)* a policy and consent specification language; *(ii)* a formalization of runtime policy compliance; *(iii)* predefined interfaces and classes for consent management; *(iv)* a run-time system for dynamic checking of privacy compliance, with built-in generation of runtime privacy tags when new personal data is created. We prove a notion of runtime compliance with respect to the consented policies.

A	$::= no \mid read \mid incr \mid write \mid rincr \mid wincr \mid full$	access rights
$\mathcal{RD}$	$::=$ **purpose** R^+ [**where** Rel [**and** Rel]*]	purpose declaration
Rel	$::= R^+ < R^+$	sub-purpose declaration
P	$::= I \mid o$	principals: interface or object
p	$::= (P, R, A)$	policies
C	$::= pos(P, R, A) \mid neg(P, R, A)$	consented policies
Q	$::= C^*$	policy list

Fig. 1. BNF syntax definition of the policy language. I ranges over interface names, R over purpose names, and P over principal names. A principal is given by an object or an interface (representing all objects of that interface). Superscripts * and $^+$ denote general and non-empty repetition, respectively.

Paper Outline. The rest of the paper is structured as follows. Section 2 presents the policy and consent specification language, a formalization of policy compliance, and the core language. Section 3 introduces the functionality for consent management. Section 4 presents tag generation, dynamic checking and an operational semantics. Section 5 discusses related work, and Sect. 6 concludes the paper and discusses future work.

2 Language Setting

In order to formalize the management and processing of personal information, we introduce basic notions for privacy policies and consent in Sect. 2.1, and introduce a small language for interface and class definitions in Sect. 2.2.

2.1 Policy and Consent Specification

Privacy policies are often described in natural language statements. To verify formally that the program satisfies the privacy specification, the desired notions of privacy need to be expressed explicitly. To formalize such policies, we define a policy specification language. In our setting, a privacy policy is a statement that expresses permitted use of the personal information by the declared program entities. In particular, a policy is given by triples that put restrictions on what *principals* can access the personal data for specific *purposes* and *access-rights*. That being the case, a policy p is given by a triple (P, R, A), where:

i) P describes a principle that can access personal information and is given by an object representing a principal, or by an interface (representing all objects

supporting that interface). Interfaces are organized in an open-ended inheritance hierarchy, letting $I < J$ denote that principal I is a subinterface of J and letting $o < I$ if object o supports I. We let $\leq$ denote the transitive and reflexive extension of $<$. As an example,

$$Specialist < Doctor < HealthWorker$$

ii) The purpose name R describe the specific purpose for which personal data can be used. Such purpose names are organized in an open-ended directed acyclic graph, reflecting specialization. For instance, the declaration

purpose $treatm, health\_care$ **where** $treatm < health\_care$
policy $p_{Doc} = (Doctor, treatm, rincr)$ // *general policy*
consent $pos(p_{Doc}) = (Doctor, health\_care, write)$ // *general positive consent*
consent $neg(p_{MyDoc}) = (Dr.Hansen, treatm, full)$ // *specific negative consent*

Fig. 2. Sample purpose and policy definitions. Here $Dr.Hansen$ is a principal object.

$$\textbf{purpose}\ \ spl\_treatm, treatm\ \ \textbf{where}\ \ spl\_treatm < treatm$$

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

iii) Access rights A describe the permitted operations on personal data, and are given by a lattice, with *full* and *no* as top and bottom and with a partial ordering $\sqsubseteq_A$: *read* gives read access, *write* gives write access (without including read access), *incr* allows addition of new information but neither read nor write is included. The join of *read* and *incr* is abbreviated *rincr*, the join of *write* and *incr* is abbreviated *wincr*, while the join of *read* and *write* is *full*.

The language syntax for policies is summarized in Fig. 1, where [ ] is used as meta-parenthesis, and superscripts * and $^+$ denote general and non-empty repetition, respectively. Sample policies are given in Fig. 2.

Definition 1 (Policy Compliance). Policy compliance, $\sqsubseteq$, *is defined by*

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

Thus, a policy p' complies with p if it has the same or smaller interface, the same or more specialized purpose, and the same or weaker access rights.

In order to deal with both addition and removal of policies, we organize the policies in a list of negative and positive policies, such that the newest and most significant policy is last in the list. A positive consent has the form $pos(p)$, where p is a policy triple, meaning that access to personal data requiring p is allowed.

A negative consent has the form $neg(p)$, meaning that access to personal data requiring p is forbidden. The disjoint union of these two forms is captured by the type *Consent*. Consented policies are organized in a *Consent* list. We define compliance of policies with respect to such a list L by:

$$\begin{aligned} p \sqsubseteq \epsilon &= \mathit{false} \\ p \sqsubseteq (L; pos(p')) &= \mathbf{if}\ p \sqsubseteq p'\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ p \sqsubseteq L \\ p \sqsubseteq (L; neg(p')) &= \mathbf{if}\ p \sqsubseteq p'\ \mathbf{then}\ \mathit{false}\ \mathbf{else}\ p \sqsubseteq L \end{aligned}$$

where _ ; _ denotes list append. Thus positive or negative policies later in the list (capturing newer ones) override policies earlier in the list (capturing older ones) with smaller policy triples. This gives a simple way to upgrade and downgrade consent, and with a uniform treatment of negative as well as positive consent.

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

Fig. 3. BNF syntax of the core language, extended with purpose specifications (::R). A field is denoted w, a local variable x, a method parameter y, a class parameter z, type names N, expressions e, and expression lists $\overline{e}$. The brackets in $[T]$ and $[\overline{T}]$ are ground symbols. Function symbols f range over pre-/programmer-defined functions/-constructors with prefix/mixfix notation.

```
interface PrivacySettings {
  with User
    op PolicyList seeMyPolicies() // the current policy list is sent to the user
    op Bool addPolicy(Policy p) // add a new policy (return false if redundant)
    op Bool remPolicy(Policy p) } // remove a policy (return false if redundant)
```

Fig. 4. Interface declarations for subject's privacy settings

2.2 A High-Level Language for Active Objects

In the setting of active objects, the objects are autonomous and execute in parallel, communicating by so-called asynchronous method invocations. Object-local data structure is defined by data types. Classes are defined by an imperative language while data types and associated functions are defined by a functional language. We assume interface abstraction, i.e., remote field access is illegal and an object can only be accessed though an interface. This allows us to focus on major challenges of modern architectures, without the complications of low-level language constructs related to the shared-variable concurrency model.

```
interface Sensitive{
  with Subject
    op Void requestMySensitiveData() // a subject (the caller) requests to see
  }                                  // all personal data in the sensitive object about her

interface Subject extends PrivacySettings, Sensitive, Principal {
  with Sensitive
    op Void receiveMySensitiveData(List[TaggedData] tl)
  with User
    op Void collectMyData() // to initiate collection of personal data about the subject
    op List[TaggedData] seeMyData() // the received info is sent to subject user
  } :: all
```

Fig. 5. Interface declarations for sensitive objects and data subjects.

A strongly typed language for active objects based on [6] is given in Fig. 3. The programs we consider are defined by a sequence of declarations of interfaces (containing method declarations), classes (containing class parameters, fields, methods and class constructors), and data type definitions. Class parameters are like fields, but with read-only access. A subclass inherits class parameters, fields, and methods (and the class constructor) unless redefined. A method m may have a cointerface Co given by the with clause, **with** Co, restricting callers to objects supporting interface Co (this is checked statically and allows type-correct call backs). Each method dealing with personal data must have an associated purpose, given at the end of the method definition (:: R), if any, otherwise the one declared for the method in the interface, if any, or otherwise the one declared for the interface. Methods may declare local variables and end with a return statement. We include standard statements such as skip, assignment (:=), object creation (**new**), if- and while-statements, and we allow blocking calls ($v := o.m(\overline{e})$) where o is the callee and $\overline{e}$ is the list of actual parameters, and asynchronous calls $o!m(\overline{e})$ and broadcasts $I!m(\overline{e})$ to all objects of interface I. The incremental update $v :+ e$ extends a list v with one or more elements e.

We consider pure expressions, including products $(e_1, e_2, \ldots)$, lists, and function applications $f(\overline{e})$ where f may be a defined function or a constructor

function (including ";" for lists and constants such as *nil*, *void*, 0, 1, 2, etc.). A value is a variable-free expression with only constructor functions, such as the list $nil; 1; 2; 3$.

3 Consent Management

The policy settings of each data subject may change dynamically during runtime in interaction with the external users. In order to handle this, we define a runtime system where personal data values are tagged with specification of data subjects and processing purposes. The runtime system will check that every access to personal data complies with the consented policies. Since there could be a huge amount of personal information in a distributed system, it is essential that the information in the tags is minimized. *Our framework includes a general solution for subjects to observe and change their privacy settings.* We chose to let the information about the consented policies be stored separately from the tags. The tags are generated by the runtime system as explained in detail in Sect. 4.1. The consented policy may change dynamically, in contrast to the information in the tags, which do not change. By combining the core information in the tags with the dynamically changing consent information, we are able to keep the information in the tags relatively small.

We let interface *Principal* correspond to a system user, be it a person, an organization, or other identifiable actor. Interface *PrivacySettings* (Fig. 4) defines methods for accessing and resetting consented policies by the data subject, while the subinterface *Subject* (Fig. 5) defines methods for consent management including functionality for requesting and updating policy settings. For each data subject there is an associated object (i.e., the subject object) supporting *PrivacySettings* and *Subject*, and this object is used to manage the privacy settings and policies in interaction with an external user (for instance through an app on a mobile phone). The subject object will contain the consented policies and is used when personal data about the data subject is accessed, in order to check compliance with the consented policies as explained in the operational semantics. In addition, it is used to manage the collection of personal data from sensitive objects. Thus the class *SUBJECT*, supporting interface *Subject*, deals with handling of consented policies and collection of personal data.

The interface *PrivacySettings* specifies the interface for updating consent (Fig. 4). It includes methods for adding and removing consent by the user such that after successful addition/removal of a policy p, that policy (or a smaller) allows/denies access to personal data. There is also functionality for an user to check her current policy settings, through method *seeMyPolicies*, which returns all policies of that user.

Class *PRIVACYSETTINGS* in Fig. 6 implements *PrivacySettings* by storing the consented policies in a field variable *consented*, which is a *Consent* list (i.e., list of consented policies). The add operation $addPolicy(p)$ adds $pos(p)$ at the end of the consented list (unless $p \sqsubseteq consented$ holds already), and the remove operation $remPolicy(p)$ adds $neg(p)$ at the end of the list (unless $p \sqsubseteq consented$ gives false, in which case it is redundant). The consent list of a subject S can be initialized with some initial policies, including $(S, all, rincr)$ for self access.

One may also remove redundant consented policies in the list when new ones are added, using the following strategy: A positive policy $pos(p)$ occurring in a list L is *redundant* in the list if $p \sqsubseteq L'$ holds where L' is the list with this occurrence removed. Similarly, a negative policy $neg(p)$ occurring in L is *redundant* if $p \sqsubseteq L'$ gives false. In these cases L can replaced by L' in order to simplify future compliance tests by limiting the size of the *consented* list.

3.1 Data Collection from Sensitive Objects to Data Subjects

In order to restrict processing of personal information, we define an interface *Sensitive*, which will be the superinterface of all objects handling personal data. The interfaces *Subject* and *Sensitive* in Fig. 5 define the functionality for collection of personal information for subjects and define consent management. Class *SUBJECT* in Fig. 7 gives an implementation. A call to the method *collectMyData* on a subject object from the corresponding user will start a process to collect all personal information about the subject. The broadcast *Sensitive*!*requestMySensitiveData*() sends a *requestMySensitiveData* message to all objects implementing *Sensitive*. A sensitive object may receive a *requestMySensitiveData* request from a subject object (*caller*) and will then react by collecting the personal data tagged with the subject and send it back to the subject object through the method *receiveMySensitiveData*. This data is then collected incrementally and stored in a (tagged) list, *mydata*, which can be accessed by the corresponding user using *seeMyData* or *seeData*. This class may be used as superclass of objects supporting *Subject*. The method *requestMySensitiveData* is provided by, and implemented in, the runtime system as explained in Sect. 4.2.

```
class PRIVACYSETTINGS (User user) implements PrivacySettings {
 List[Consent] consented; // the current privacy policies
with User
   op PolicyList seeMyPolicies()
     {return (if caller = user then consented else nil fi)}
   op Bool addPolicy(Policy p) { // add a new policy (return false if redundant)
     Bool ok := (caller = user); if ok then
      if p ⊑ consented then ok := false
      else consented :+ pos(p) fi fi; return ok } // incremental update
   op Bool remPolicy(Policy p){ // add a negative policy (return false if redundant)
     Bool ok := (caller=user); if ok then
      if p ⊑ consented then consented :+ neg(p)// incremental update
      else ok := false fi fi; return ok }
...}
```

Fig. 6. The implementation of privacy settings and policy changes.

Interface *Subject* has *all* as declared purpose, and all methods in the interface and class inherit this purpose. The access to personal information in *SUBJECT* complies with the general policy $(S, all, rincr)$ for a subject object S.

4 Runtime System

The operational semantics of the considered language is given in Fig. 9. Data values are tagged with set of pairs of subject and purpose. A runtime configuration of an active object system is captured by a multiset of objects and messages (using blank-space as the binary multiset union constructor). Each object o is responsible for executing all method calls to o as well as self-calls. An object has at most one active process, reflecting the remaining part of a method execution. Objects have the form

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

where o is the object identity, δ is the current object state, and $\overline{s}$ is a sequence of statements ending with a **return**, representing the remaining part of the active process, or **idle** when there is no active process. The state of an object δ is given by a twin mapping from variable names to tagged values, written $(\alpha|\beta)$, where α is the state of the field variables $\overline{w}$ and class parameters $\overline{z}$ (including this), and β is the state of the local variables $\overline{x}$ and formal parameters $\overline{y}$ of the current process. Look-up in a twin mapping, $(\alpha|\beta)[v]$, is simply given by **if** v *in* β **then** $\beta[v]$ **else** $\alpha[v]$, where *in* is used for testing domain membership. The notation $\alpha[v \mapsto e]$ denotes map update, and the notation $(\alpha|\beta)[v := e]$ abbreviates **if** v *in* β **then** $(\alpha \mid \beta[v \mapsto (\alpha|\beta)[e]])$ **else** $(\alpha[v \mapsto (\alpha|\beta)[e]] \mid \beta)$.

```
class SUBJECT implements Subject // thereby also PrivacySettings
    extends PRIVACYSETTINGS {
  List[TaggedData] mydata; // personal data collected about subject
  op Void receiveMySensitiveData(List[TaggedData] tl){
    mydata :+ tl } // could also include info of caller, i.e., mydata :+ (caller,d)
with User
  op Void collectMyData(){
    if caller = user then
      mydata := nil; // clear list
      Sensitive!requestMySensitiveData() fi } // broadcast to all sensitive objects
  op List[TaggedData] seeMyData(){
    return (if caller = user then mydata else nil fi)}
}
```

Fig. 7. The implementation of subject.

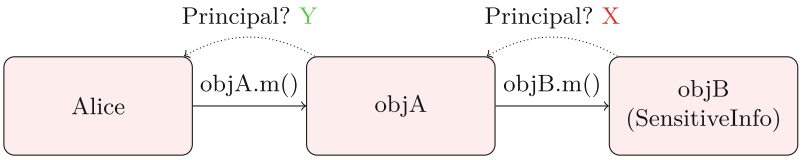

Fig. 8. Call chain. Here Alice is the principal of the method execution on objB.

In addition, the operational semantics defines the system variables *pcs* and nextFut, which appear in the state of each object (in α). The "program counter stack" *pcs* is used for storing the stack of tags on the conditions corresponding to the nesting of enclosing if/while statements, and nextFut is used for generating unique identities for calls. Furthermore, the self reference this is handled as an implicit class parameter, while myfuture and caller appear as implicit method parameters, holding the identity of a call and its caller, respectively.

Example. Consider some personal health data with the tag $\{(Alice, treatm)\}$, and assume the consented policies $(\ldots; pos(Doctor, treatm, full))$ in object *Alice*. A Doctor can then read the data since there is a matching positive policy with at least read access where Doctor is the principal and the purpose of the current method is *treatm* or less. However, for the consented list $(\ldots; pos(Doctor, treatm, full); neg(Bob, treatm, read))$, where *Bob* is a doctor object, read access will be denied due to the presence of negative policy.

4.1 Runtime Tagging of Values

The runtime checking uses two special notions: The *current purpose*, denoted $R_{current}$, is the purpose of the enclosing method, which we assume is statically specified, as in [4]. (Alternatively one could take the purpose defined in some other way, for instance by data-flow graphs as in [7].) Secondly, we define the *current principal*, denoted $P_{current}$, as the first principal object found by following the dynamic call chain from a method execution as illustrated in Fig. 8 (ignoring non-principals such as $objA$).

The runtime evaluation of an expression e gives a tagged value c of form d_l with a tag l. In a method execution the evaluation of an expression e in a state δ and with policy context *pcs* is denoted $\Delta[e]$, where the data value is evaluated (as explained in the next subsection) ignoring tags, resulting in a ground term, i.e., a term d with only constructor functions (g), and where the tag is given by the *tag* function defined below: For tagged data values, the tag function is given by $tag(d_l) = l$, and for untagged values it is given by:

$$
\begin{array}{llll}
tag() & = flatten(\delta[pcs]) & tag(d_l, \overline{c}) & = l \cup tag(\overline{c}) \\
tag(g(S)) & = \{(S, R_{current})\} \cup tag() & tag(S, \overline{c}) & = \{(S, R_{current})\} \cup tag(\overline{c}) \\
tag(g(\overline{c})) & = tag(\overline{c}), \texttt{otherwise} & tag(d, \overline{c}) & = tag(d) \cup tag(\overline{c}), \texttt{otherwise}
\end{array}
$$

Note that the tag includes $flatten(\delta[pcs])$, defined as the *union* of all tags in the stack *pcs*. An untagged product $(..., S, ...)$ will also include the tag $(S, R_{current})$. A pair (S, S') will be tagged with $\{(S, R_{current}), (S', R_{current}), flatten(\delta[pcs])\}$. An untagged constructor value $g(S, \overline{c})$ is tagged like the product $(S, \overline{c})$. When a subject S occurs as an argument to a constructor term or product, the pair $(S, R_{current})$ is added to the tag set. Note that $g(S)$, (S, S'), (S, c), and (c, S) include $(S, R_{current})$ in the tag set, but S and (S) do not, as a product must have at least two arguments. A tag (S, R) is redundant in a tag set l, and may be removed, if there is another tag (S, R') in l such that $R < R'$. Non-personal data will have an empty tag set. Policies are considered non-personal.

4.2 Runtime Checking of Privacy Compliance

The runtime system keeps track of the current consented policy list for each subject, specifying the policies for accessing personal data concerning the subject. In the runtime system there is a mapping from subjects to policy lists

$$\mathcal{M} : Subject \rightarrow PolicyList$$

given by $\mathcal{M}[S] == S.consented$ where each *consented* is maintained by the runtime system. Note that even though remote field access is not possible within the program syntax, this restriction does not apply to the runtime system.

The evaluation of expressions, $\Delta[e]$, is done depth-first, left-to-right. Thus $\Delta[f(\overline{e})]$ is $[f(\Delta\overline{e})]$, $\Delta[$**if** b **then** e **else** $e']$ is $\Delta[e]$ if $\Delta[b]$ is true and $\Delta[e']$ if $\Delta[b]$ is false, and for a value c, $\Delta[c]$ is c. (Here b is a boolean expression.) For a defined function f, $\Delta[f(\overline{c})]$ is obtained by the definition of f replacing the formal parameters by the actual values $\overline{c}$. We let the evaluation of a variable v have a built-in compliance check of read access:

$$\begin{array}{ll} \Delta[v] = \delta[v], & \textbf{if } \forall (S,R) \in tag(\delta[v]) \ : \ (P_{current}, R, read) \sqsubseteq \mathcal{M}[S] \\ \Delta[v] = error, & \textbf{otherwise} \end{array}$$

In the first line, the tag is defined by the *tag* function in Sect. 4.1. A policy $(S, all, rincr)$ is initially added to the consented list of each subject object S, to allow the data subject to read and increment his/her own data.

For write access, we define a modified state update function $\Delta[v := c]$ so that it includes the appropriate checks for assignments, and similarly for incremental assignments. Note that there is no check on local variables since they form the local work space, i.e., a method has always write access to the local variables.

$$\begin{array}{ll} \Delta[x := c] = \delta[x := c], & \\ \Delta[w := c] = \delta[w := c], & \textbf{if } \forall (S,R) \in tag(c) \ : \ (P_{current}, R, write) \sqsubseteq \mathcal{M}[S] \\ \Delta[w := c] = \delta[w := error], & \textbf{otherwise} \end{array}$$

This definition is lifted to expressions e, letting $\Delta[x := e]$ denote $\Delta[x := \Delta[e]]$. Similarly, $\Delta[v :+ c]$ requires $(P_{current}, R, incr) \sqsubseteq \mathcal{M}[S]$ for $(S,R) \in tag(c)$. Non-personal data can be accessed without restrictions since the tag is empty.

Implementation of method *requestMySensitiveData* is provided by the runtime system by making the call $caller!receiveMySensitiveData(tl)$ where tl is given by $\Delta[\overline{w}]/caller$, i.e., the tagged values of fields with *caller* in the tag.

Runtime Overhead. We have given a solution for compliance checking by a runtime system formulated at a high-level of abstraction. We here discuss the overhead in tagging and checking with this solution, and how it can be reduced. By combining the core information in the tags with the dynamically changing consent information, we are able to keep the information in the tags relatively small, and moreover the tags are not changed when the consent is changed, which is a crucial property. Thus the main overhead is in accessing the consented list for the subjects in the tag. Note that the updates on each consent list is atomic,

so there is no need for critical regions nor object synchronization at the runtime level. Thus a compliance check made by one object will not slow down the other objects. This processing can easily be made more efficient by letting each principal pull a copy of a subject's consent setting when needed. However, as this could lead to outdated consent information, one could use a version number for each subject's consent list, and let a principal check that it has the latest version before applying its local copy of a consent list. A further method of reducing overhead, would be to re-represent each consent list by means of a mapping (from principal and purpose of a given subject to access right) thereby the list traversal is reduced to direct look-up. This method has a cost whenever a consent list is updated. A further discussion is beyond the scope of this paper.

4.3 Operational Rules

Each rule in the operational semantics deals with only one object o, and possibly messages, reflecting the nature of concurrent distributed active objects, communicating asynchronously. Remote calls and replies are handled by message passing. When a subconfiguration $\mathcal{C}$ can be rewritten to a $\mathcal{C}'$, this means that the whole configuration $\ldots\mathcal{C}\ldots$ can be rewritten to $\ldots\mathcal{C}'\ldots$, reflecting interleaving semantics. The operational rules reflect small-step semantics. For instance, the rule for *skip* is given by

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

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

Each method call will have a unique identity u. A message has the form

$$\mathbf{msg}\ o \rightarrow o'.m(u, \overline{c})$$

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

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

representing a completion event where c is the returned value and u the identity of the call. In addition, $\mathbf{msg}\ o \rightarrow I.m(u, \overline{c})$ denotes a broadcast to all I objects.

The semantics in Fig. 9 formalizes the notion of idleness, and generation of objects and messages, including a rule (*no-query*) for garbage collection of unused reply messages. Generation of identities for objects and method calls is handled by underlying semantic functions and implicit attributes.

The operational semantics uses an additional *query* statement, **get** u, for dealing with the termination of call statements. A synchronous call is treated as an asynchronous call followed by a **get** query. The query **get** u is blocking while waiting for the method response with identity u.

Assignment is handled by updating the state, requiring that there is read access to any personal data (using Δ). An if-statement requires read access to personal data in the condition and the resulting tag set l is pushed on the policy stack *pcs*, ensuring that all evaluations in the taken branch implicitly includes

assign :
$$o : \mathbf{ob}(\delta, v := e; \overline{s}) \longrightarrow o : \mathbf{ob}(\Delta[v := e], \overline{s})$$

if-true :
$$o : \mathbf{ob}(\delta, \texttt{if}\ b\ \texttt{then}\ \overline{s1}\ \texttt{else}\ \overline{s2}\ \texttt{fi}; \overline{s}) \longrightarrow o : \mathbf{ob}(\delta[pcs := push(pcs, l)], \overline{s1}; pcs := pop(pcs); \overline{s}) \quad \texttt{if}\ \Delta[b] = true_l$$

if-false :
$$o : \mathbf{ob}(\delta, \texttt{if}\ b\ \texttt{then}\ \overline{s1}\ \texttt{else}\ \overline{s2}\ \texttt{fi}; \overline{s}) \longrightarrow o : \mathbf{ob}(\delta[pcs := push(pcs, l)], \overline{s2}; pcs := pop(pcs); \overline{s}) \quad \texttt{if}\ \Delta[b] = false_l$$

while :
$$o : \mathbf{ob}(\delta, \texttt{while}\ b\ \texttt{do}\ \overline{s1}\ \texttt{od}; \overline{s}) \longrightarrow o : \mathbf{ob}(\delta, \texttt{if}\ b\ \texttt{then}\ \overline{s1};\ \texttt{while}\ b\ \texttt{do}\ \overline{s1}\ \texttt{od}\ \texttt{fi}; \overline{s})$$

new :
$$o : \mathbf{ob}(\delta, v := \texttt{new}\ C(\overline{e}); \overline{s}) \longrightarrow o : \mathbf{ob}(\delta[v := o'], \overline{s})\ \ o' : \mathbf{ob}(\delta_C[\mathsf{this} \mapsto o', \mathsf{nextFut} \mapsto initialFut(o'), \overline{z} \mapsto \Delta[\overline{e}]], init_C) \quad \texttt{where}\ o' = (fresh, C),\ \text{with } fresh \text{ a fresh reference relative to } C$$

async. call :
$$o : \mathbf{ob}(\delta, a!m(\overline{e}); \overline{s}) \longrightarrow o : \mathbf{ob}(\delta[\mathsf{nextFut} := next(\mathsf{nextFut})], \overline{s})\ \ \texttt{msg}\ o \rightarrow \Delta[a].m(\Delta[\mathsf{nextFut}, \overline{e}])$$

sync. call :
$$o : \mathbf{ob}(\delta, v := a.m(\overline{e}); \overline{s}) \longrightarrow o : \mathbf{ob}(\delta, a!m(\overline{e}); v := \texttt{get}\ \delta[\mathsf{nextFut}]; \overline{s})$$

start :
$$\texttt{msg}\ o' \rightarrow o.m(u, \overline{c})\ \ o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \longrightarrow o : \mathbf{ob}((\alpha|(\beta[\mathsf{caller} \mapsto o', \mathsf{myfuture} \mapsto u, \overline{y} \mapsto \Delta[\overline{c}], pcs \mapsto nil])), \overline{s})\ \ \texttt{where}\ (m, \overline{y}, \beta, \overline{s})\ \text{is the body of } m \text{ in the class of this}$$

return :
$$o : \mathbf{ob}(\delta, \texttt{return}\ e) \longrightarrow o : \mathbf{ob}(\delta, \mathbf{idle})\ \ \texttt{msg}\ \delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{myfuture}], \Delta[e])$$

query :
$$\texttt{msg}\ o \leftarrow o'.(u, c)\ \ o : \mathbf{ob}(\delta, v := \texttt{get}\ u; \overline{s}) \longrightarrow o : \mathbf{ob}(\delta, v := c; \overline{s})$$

no-query :
$$\texttt{msg}\ o \leftarrow o'.(u, c)\ \ o : \mathbf{ob}(\delta, \overline{s}) \longrightarrow o : \mathbf{ob}(\delta, \overline{s})\ \ \texttt{if get}\ u \notin \overline{s}$$

Fig. 9. Operational rules defining small-step semantics with privacy tags. Unique future identities are ensured by functions $initialFut$, parameterized with the parent, and $next$.

l in the tag set. A while loop is handled by expanding **while** b **do** s **od** to **if** b **then** s; **while** b **do** s **od fi** upon execution of the while-statement. Void methods return the value *void*. We assume all methods end in a return statement, including class constructors, which end in **return** *void* (although omitted in the examples). An assignment of the form $v :+ e$ is treated as an atomic operation at runtime. (When lists are implemented by linked lists, this operation can be executed by a single pointer assignment, since the value of e is not affected by other objects.) Semantically, $v :+ e$ is the same as $v := v + e$, and we do not show a special rule for it. This means that a consent update can also be considered atomic. Furthermore, we assume that initial values given to fields or local variables are expanded to assignments, as described earlier.

For simplicity, rules for broadcasting (similar to that for asynchronous calls) and local synchronous calls (i.e., queries on local calls) are omitted, since such calls do not pose additional privacy challenges. In the current semantics, a query on a local call will lead to deadlock. The handling of local queries would require addition of a stack in the object state in order to be able to push and pop unfinished local method frames, for instance as in [6].

The theorem below ensures that every access to a data subject's personal information will comply with the consented policy.

Theorem 1 (Runtime Compliance). *After a policy is successfully removed, all further variable accesses that need this policy will fail by giving a runtime error until the policy, or a stronger one, is added again.*

Proof. Consider an object state δ where $(S, R) \in tag(\delta[v])$. Let policy p denote $(P_{current}, R, read)$. We must prove that a runtime look-up of v in such a state gives error after a policy p' such that $p \sqsubseteq p'$ is removed from the consented list of S and before a policy p'' such that $p \sqsubseteq p''$ is added to the consented list of S.

Every variable look-up is made through one of the operational rules, by means of δ or Δ. By inspection of these rules, we observe that all program variables are evaluated by Δ apart from caller and this in rule Return, but here the *pcs* stack is empty (since a return statement occurs last in a body), so evaluation by δ in this case is the same as by Δ. It remains to show the theorem for variable access through Δ, and for an access to v we must show that $p \not\sqsubseteq \mathcal{M}[S]$.

By induction on the length of the execution we show that $\Delta[v]$ gives error between the successful removal of p' and addition of p'' to $\mathcal{M}[\mathcal{S}]$. A successful removal must perform the atomic operation $consented :+ neg(p')$ in S. Right afterwards, $neg(p')$ is the last element in $\mathcal{M}[\mathcal{S}]$ and therefore $\Delta[v]$ gives error. If a consent $neg(p''')$ is added, $p \sqsubseteq \mathcal{M}[\mathcal{S}]$ remains false. If a consent $pos(p''')$ is added, we may assume that $p \not\sqsubseteq p'''$ (otherwise p''' can be used as p'' and there is nothing to prove) and by the induction hypothesis $p \sqsubseteq \mathcal{M}[\mathcal{S}]$ remains false. □

5 Related Work

This paper focuses on the intersection between compliance formalization and programming languages. This line of work is relatively recent, featuring several

threads of active research such as policy specification, policy enforcement, monitoring, privacy by design, language-based privacy, and role-based access control.

The work presented in [8] provides a privacy management framework for the definition of privacy agents (such as subject, controller) acting as representatives of individuals. These privacy agents play a specific role as "representative" or "proxy" of the user in order to manage personal data and ensure privacy-compliant interactions among agents. We share with [8] the objective of privacy compliant interactions, but we use an integrated style, i.e., including compliance checks within objects and actors accessing personal data. In addition, we use the same policy language for different actors and consented policies are maintained in subject objects. Cunche et al. [9] present a generic information and consent framework for IoT that allows the data subject to express privacy requirements as well as receive the information and associated privacy policy. The privacy policies for subjects and controllers are based on the PILOT semantics [10]. Privacy policies in [10] are more expressive than ours as they also encapsulate contextual information, but the semantics of policy compliance is not discussed in particular. We define fewer privacy requirements and focus on compliance formalization. The approach followed in [9] makes use of dedicated privacy agents, while we integrate the compliance checks in actor objects.

Sen et al. [11] demonstrate techniques for compliance checking in big data systems. Privacy policies are specified using a policy specification language, *LEGALEASE*, where policies can be expressed using *allow* and *deny* clauses to permit and prohibit access. Policies can be expressed using nested allow-deny rules. Policy clauses use data store, purpose, role, and data type attributes to specify information flow restrictions. Then, a data inventory tool *GROK* maps data types in code to high-level policy concepts, and the compliance checking then reduces to a form of information flow analysis. This is similar to our approach in [4] where we associate policy with the *types* carrying sensitive information, but the difference is that the type-policy mapping is integrated in the language. The policy specification language in [11] has some similarities with our work: the semantics of policies is compositional and policies are expressed as lists of positive and negative policies. However, for the sake of simplicity, we do not consider nested-policies. All information flow restrictions (policy attributes) are encoded as a lattice in [11], while in our setting that is not the case. However, in [11] the concept lattice does not seem to distinguish with information about other subjects, which we do and in addition we can generate tags at runtime when new information (involving a subject or personal information) is created.

Yang et al. [12] propose a *policy-agnostic programming* model. Sensitive data values are associated with policies and then the programmer may implement the rest of the program in a policy agnostic manner. The language's [13] runtime system enforces these policies to ensure that only policy compliant values are used in computations. In contrast, we use generalized polices for each subject (including purpose) and minimize the information in the tags.

Other examples of language-based approaches relying on information-flow control include the role-based approach in [14] and the purpose-based approach

in [15]. Myers and Liskov present a model of decentralized information flow labels, where *principals* and *labels* are the essentials of the model [14]. Principals are the entities that own, update and release (to other principals) information. A label is a set of *owner: reader* policy pairs, where *owner* is the data owner (i.e., subject in our approach), and *reader* is the principal that has read access to this data. Programs and data are annotated with such labels, and information flow restrictions are enforced by type checking. For an access to be valid, all the policy requirements of the label should be enforced, which holds in our approach as all the tags must comply with the consented policies. There are no generalized policies, and the tags will take more space than in our case. In [15], Hayati and Abadi describe an approach to model and verify aspects of privacy policies in the Jif (Java Information Flow) programming language. Data collected for a specific purpose is annotated with Jif principals and then the methods needed for a specific purpose are also annotated with Jif principals. Explicitly declaring purposes for data and methods ensures that the labeled data will be used only by the methods with connected purposes. However, this representation of purpose is not sufficient to guarantee that principals will perform actions compliant with the declared purpose. In contrast, this can be checked at runtime in our approach.

Basin et al. [7] propose an approach that relates a purpose with a business process and use formal models of inter-process communication to demonstrate GDPR compliance. Process collection is modeled as data-flow graphs which depict the data collected and the data used by the processes. Then these processes are associated with a data purpose and are used to algorithmically generate data purpose statements, i.e., specifying which data is used for which purpose and detect violation of data minimization. A main challenge tackled by this work is to automatically generate compliant privacy policies from the model. We share with this work an explicit specification of *purpose*. In [7], a purpose is associated with a process, while in our approach a method accessing personal information is tagged with a purpose and personal data is tagged with sets of $(subject, purpose)$ pairs. This tagging is useful in generating privacy policies to check compliance.

6 Conclusion

We propose a consent management framework that allows a data subject to communicate and update consent policies to the controller and to view all personal data about her in the system along with the purposes for which they are used. We have considered a core language for distributed active object systems and formalized the notion of policy compliance and given an operational semantics for the considered programming language. The runtime system ensures that every access to personal data complies with the currently consented polices.

We have illustrated the feasibility of formalizing GDPR specific privacy requirements, including *privacy by design* by providing explicit specifications of *purpose* and policy constructs; *lawfulness and transparency* of processing based

on consented purposes; *data subject access request* by providing predefined interfaces and classes to assist in providing the data subject with the personal data and purposes for which it is being processed.

Our framework includes a general solution for subjects to observe and change their privacy settings and for subjects to be informed about all personal data stored about them. The solution consists of a set of predefined types for privacy related concepts and a set of interfaces that forms the basis for interaction with external users, a set of classes that is used in interaction with the runtime system, and runtime checking of all access to personal data to ensure that it complies with the current privacy settings. The same framework can be reused for another language, as long as the assumption of interface abstraction is respected and as long as the purpose of any method handling personal data is identified.

Future Work: The framework can be extended to accommodate for other legal bases by having separate policy lists for each legal basis, and a logic to chose from these bases as required. More information can be included in the tags for a richer compliance check, for instance, the *data creator* can be recorded as the current principal of the method instance creating the data. More information can be included in the policy specification, for example restrictions on temporal validity, data collectors, and data creators. Furthermore, we could add cases of non-personal tag information as exceptions to the generated tags, for instance to deal with encryption. We can easily add more fine-grained methods for selection of policies/personal data in the interfaces/classes for privacy settings and data collection (from sensitive objects), for instance using purpose and principal to limit the selection.

References

1. European Parliament and Council of the European Union: The General Data Protection Regulation (GDPR). https://eur-lex.europa.eu/eli/reg/2016/679/oj. Accessed 24 Nov 2019
2. Article 29 Working Party: Guidelines on Consent under Regulation 2016/679. https://ec.europa.eu/newsroom/article29/item-detail.cfm?item_id=623051. Accessed 05 Feb 2020
3. Métayer, D.: Formal methods as a link between software code and legal rules. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 3–18. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24690-6_2
4. Tokas, S., Owe, O., Ramezanifarkhani, T.: Language-based mechanisms for privacy-by-design. In: Friedewald, M., Önen, M., Lievens, E., Krenn, S., Fricker, S. (eds.) Privacy and Identity 2019. IAICT, vol. 576, pp. 142–158. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-42504-3_10
5. Nierstrasz, O.: A tour of hybrid - a language for programming with active objects. In: Advances in Object-Oriented Software Engineering, pp. 67–182. Prentice-Hall, Upper Saddle River (1992)
6. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Softw. Syst. Model. **6**, 39–58 (2007)

7. Basin, David, Debois, Søren, Hildebrandt, Thomas: On purpose and by necessity: compliance under the GDPR. In: Meiklejohn, Sarah, Sako, Kazue (eds.) FC 2018. LNCS, vol. 10957, pp. 20–37. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-58387-6_2
8. Métayer, D.: A formal privacy management framework. In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 162–176. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01465-9_11
9. Morel, V., Cunche, M., Le Métayer, D.: A generic information and consent framework for the IoT. In: 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE), pp. 366–373. IEEE (2019)
10. Pardo, R., Le Métayer, D.: Analysis of privacy policies to enhance informed consent. In: Foley, S.N. (ed.) DBSec 2019. LNCS, vol. 11559, pp. 177–198. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22479-0_10
11. Sen, S., Guha, S., Datta, A., Rajamani, S.K., Tsai, J., Wing, J.M.: Bootstrapping privacy compliance in big data systems. In: 2014 IEEE Symposium on Security and Privacy, pp. 327–342. IEEE (2014)
12. Yang, J., et al.: Preventing information leaks with policy-agnostic programming. Ph.D. thesis, Massachusetts Institute of Technology (2015)
13. Yang, J., Yessenov, K., Solar-Lezama, A.: A language for automatically enforcing privacy policies. ACM SIGPLAN Not. **47**(1), 85–96 (2012)
14. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. (TOSEM) **9**(4), 410–442 (2000)
15. Hayati, K., Abadi, M.: Language-based enforcement of privacy policies. In: Martin, D., Serjantov, A. (eds.) PET 2004. LNCS, vol. 3424, pp. 302–313. Springer, Heidelberg (2005). https://doi.org/10.1007/11423409_19