DPL: A Language for GDPR Enforcement

Farzane Karami
Dept. of Informatics
University of Oslo
Oslo, Norway
farzanka@ifi.uio.no

David Basin
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
basin@inf.ethz.ch

Einar Broch Johnsen
Dept. of Informatics
University of Oslo
Oslo, Norway
einarj@ifi.uio.no

2022 IEEE 35th Computer Security Foundations Symposium (CSF) | 978-1-6654-8417-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/CSF54842.2022.9919687

***Abstract*—The General Data Protection Regulation (GDPR) regulates the handling of personal data, including that personal data may be collected and stored only with the data subject's consent, that data is used only for the explicit purposes for which it is collected, and that is deleted after the purposes are served. We propose a programming language called DPL (Data Protection Language) with constructs for enforcing these central GDPR requirements and provide the language's runtime operational semantics. DPL is designed so that GDPR violations cannot occur: potential violations instead result in runtime errors. Moreover, DPL provides constructs to perform privacy-relevant checks, which enable programmers to avoid these errors. Finally, we formalize DPL in Maude, yielding an environment for program simulation, and verify our claims that DPL programs cannot result in privacy violations.**

***Index Terms*—GDPR enforcement, runtime checking**

I. Introduction

The General Data Protection Regulation (the GDPR) [1] regulates the processing of personal data and is now part of European Union law. The GDPR mandates transparent data processing, where data is collected with the data subject's consent and used only for the purposes for which it was collected. Moreover, the GDPR requires the right to be forgotten, where data must be deleted on request or after its purposes are served. It is an open question how systems processing sensitive data should be built to satisfy these requirements.

We approach this problem from a programming language perspective: how can one design a language to prevent data-protection violations? Conventional programming languages do not support the features essential to enforce GDPR compliance. For example, they lack an explicit representation of purpose, and there is no control over data collection and usage based on purposes and consent. The state-of-the-art generally checks purpose-based privacy compliance in programs using privacy labels and static information-flow analysis [2]–[4], where the enforced policies are determined by policy labels specified by the programmer at compile time. This has its limitations. For example, users' consent and deadlines for data deletion cannot be expressed in policies. Moreover, temporal aspects cannot be handled when consent is dynamically granted or revoked or retention deadlines are reached.

In this paper, we enforce GDPR requirements at runtime. Our approach allows us to enforce richer policies than those enforced statically. For example, we can express temporal requirements on data deletion, and the data will be deleted from the system when the deadline arrives. In contrast to static approaches, our runtime approach is also more exact, since users' consent, given at runtime, determines for what purposes their data can be used, and these purposes are added to the policies. However, in contrast to static approaches, our approach comes at the price of runtime overheads.

The GDPR requirements that we handle concern data usage (see Section II-A), and we present language features to enforce these requirements. Our focus is on object-oriented and service-oriented languages, where objects are entities, method calls are processes, which may use personal data, and return values are the outputs of processes. The language involves the following features: 1) Object databases with commands for data storage and retrieval. 2) Interface encapsulation, which enforces programming to interfaces and prevents remote access to fields and methods. 3) Language constructs to build and manipulate privacy policies including **policy** to create privacy policies, **opt-in** for granting consent, and **collect** for collecting data. The policies describe to whom data belongs, for which purposes data can be used, and when data must be deleted. 4) Runtime checking to ensure that processes only access data as authorized by their privacy policies.

We define DPL, a data protection language enriched with the above features. Additionally, DPL supports the users' rights to opt-out of policies or delete their data at any time. Table I summarizes the GDPR requirements we handle, the object-oriented (OO) features exploited in DPL, and our specific extensions for privacy-related operations. While we present DPL as a stand-alone language, our language features could also be implemented as an extension of existing languages. Its core is inspired by the concurrent active object language ABS [5], which achieves encapsulation by typing objects by interfaces (instead of classes). One could also use Java directly, which supports encapsulation using the private modifier for defining local methods and fields in classes, whereas methods in interfaces are defined with the public modifier. We believe that our approach could be carried out with other Java-like languages such as Java 8 [6], Scala/Akka [7], [8], and ASP [9].

In DPL, interfaces represent purposes, and their declared methods are the processes used to achieve the purpose. Personal data is collected after a user gives consent, and the identity of the corresponding privacy policy is attached to the data, which is then called *sensitive data*. DPL supports dynamic policy changes, where users grant or withdraw consent,

DOI 10.1109/CSF54842.2022.00008

GDPR requirements	OO features	Added features
consent		**opt-in**, **opt-out**
purpose limitation	interface encapsulation message passing	**policy**, **collect** runtime checking
storage limitation	object model	**store**, **retrieve** objects' databases
right to be forgotten		delete

Table I
GDPR REQUIREMENTS AND ASSOCIATED FEATURES.

and hence the policy evolves over time. Moreover, policies expire and are deleted from the system when deadlines arrive or the user deletes her data. We define DPL's runtime system that tracks the flow of sensitive data and performs runtime checking (see Section III-D). A process is allowed to access sensitive data if its intended usage complies with the purpose in the privacy policy, and the policy has not expired. In DPL, the failure to comply to a policy will not result in a privacy violation but rather a runtime error. Moreover, DPL provides constructs to perform privacy-relevant checks so that programmers can write programs that avoid actions that would lead to policy violations rather than throwing runtime errors.

We formalize DPL's operational semantics in rewriting logic [10], which is supported by the Maude system [11]. This provides a prototype interpreter[1] for executing DPL programs. We also use Maude's model checker to complement our pen-and-paper proofs with model-checked examples that support our claims.

In summary, our contributions are as follows. We define DPL, an object-oriented language extended with features that support data protection. We map the GDPR data usage requirements to our language, and define an executable formal model for DPL, given by an SOS-style operational semantics. We provide a pen-and-paper proof that DPL programs satisfy the GDPR data usage requirements. DPL provides exact enforcement of privacy policies, it is user-centric in that users' consent is reflected in policies, and it enforces richer policies than those enforced by static approaches. We also formalize DPL in rewriting logic, use Maude's model-checker to verify our data protection properties on concrete programs, and illustrate on examples how privacy violations cannot occur. Overall, DPL is the first programming language designed for developing programs that comply to GPDR data usage requirements.

II. Background

A. GDPR requirements

Our focus is on the following requirements, which are central to the GDPR's restrictions on data usage.

Purpose limitation: *"[Personal data shall be] collected for specified, explicit and legitimate purposes and not further processed in a manner that is incompatible with those purposes"* [1, Article 5, Sec. 1 (b)]. Data is considered to be personal data if it can be used, directly or indirectly, to identify a person [12]. To comply with this, the purposes for which personal data is collected must be made explicit, and the collected data must be used only for those purposes.

Consent: *Personal data is collected only if the data subject's consent is granted. In order to give consent, a data subject should be aware of the identity of the controller and the purpose of processes in which her data is used* [1, Article 6]. *Moreover, a data subject has the right to withdraw her consent at any time* [1, Article 7]. To comply with this, data collection requires consent, and personal data must no longer be processed after consent is withdrawn.

Storage limitation: *"[Personal data shall be] kept in a form which permits identification of data subjects for no longer than is necessary for the purposes for which the personal data are processed"* [1, Article 5, Sec. 1 (e)]. To comply with this, personal data shall be deleted after the purpose of processing is fulfilled [13]. For example, a credit card number is collected to make a purchase, and if a data subject consents, this information can be stored for subsequent purchases. The storage period shall be limited to a strict minimum, and a controller shall establish time limits for data erasure [1, Rec. 39].

Right to be forgotten: *"The data subject shall have the right to [...] the erasure of personal data concerning him or her [...] and the controller shall have the obligation to erase personal data without undue delay [...]"* [1, Article 17]. To comply with this, data must be promptly deleted on request.

B. An example

To illustrate our methodology in subsequent sections, we use an example taken from [13]. The example features an online retailer whose core processes are:

Register customer: A customer provides her credit card information, her e-mail, and her postal address.

Purchase: A registered customer purchases a product from the retailer's online shop using her registered credit card. This process produces the customer's order along with an invoice, which is sent to her address.

Mass Marketing: A customer's email or postal address is used to send untargeted advertisements.

Targeted Marketing: A customer's email or postal address and her shopping history are used to send targeted advertisements.

The GDPR requires *consent statements* for processes using personal data. A consent statement describes what data is used for which purposes. For example, the consent statement for **Mass Marketing** is "we use your customer information (name and email address) for mass marketing."

C. System model

Our system model formalizes how distributed applications process users' personal data. It features users, objects, and databases, where objects share data through message passing. We assume that all communication is cryptographically protected, e.g., using TLS, and focus on data protection in this distributed setting. Later we present DPL, which formalizes these distributed systems and their behavior.

[1] The Maude model is available at https://github.com/maude-gdpr/maude.

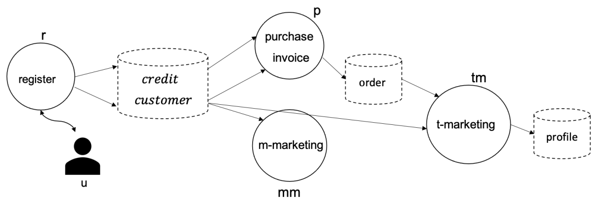

Figure 1. The online-retailing example in our system model (after [13]).

Figure 1 shows the on-line retailing example in our system model. In this example, the personal data is credit card information, customer information, the order, and the user profile. Circles represent objects with the names r, *p*, *mm*, and *tm* and contain the name of the processes that use the personal data. Dashed cylinders represent the objects' databases. The curved bidirectional arrow represents interaction with a user to collect data over a user interface. For example, the method *register* of the object r collects credit card information and customer information from the user u and stores this data in its database. Arrows from databases to objects represent database data used by the objects' methods. Arrows from objects to databases represent that method results are stored in the databases.

III. DPL: A CALCULUS FOR DATA PROTECTION

We now describe the principles behind DPL as well as its syntax and semantics. The complete formalization of the syntax and semantics in Maude, along with all auxiliary functions and examples, can be found at https://github.com/maude-gdpr/maude.

A. Language design principles

We now return to the GDPR requirements from Section II-A and explain DPL's design principles and language features used to enforce the requirements.

In what follows, we will use the following notation. Let $x_1, x_2, ..., x_n$ represent a sequence of n terms, where ϵ is the empty sequence. We use the notation $\overline{x}$ to range over sequences, possibly empty, and $\overline{x}^1$ to emphasize that the sequence has at least one element. We write lists using ":" and the empty list as []. We also employ standard list notation and write a list like $x1 : (x2 : [])$ as $[x1, x2]$.

a) Purpose limitation: To enforce this requirement, the purposes for which personal data are collected and used should be made explicit. In DPL, instead of using interfaces, we explicitly define purposes by a declaration **purpose** $P\{\overline{Sig}^1\}$, with P a name and $\overline{Sig}^1$ the method signatures (the methods' return types and parameters) required to achieve the purpose. Moreover, encapsulation is achieved by typing objects by purposes. Objects created from classes implementing purposes provide methods to achieve the purposes. We assume that programs are well-typed, which could be enforced by adapting a standard type system for interfaces (e.g., [5]), that all methods using personal data are declared in some purposes, and that each purpose contains exactly the methods needed to achieve it. For example, we can define the purpose *Purchase* as follows.

```
purpose Purchase {
 Order purchase(String credit, String customer);
 Invoice invoice(Order order, String customer); ... }
```

We propose a mechanism that prevents personal data from going to objects that implement no purpose, the wrong purpose, or even the right purpose when it has not been consented to by the data owner. First, we define *contracts* declaring which object may use the methods associated with a purpose. Afterwards, we define *privacy policies*, which are attached to collected data and contain sets of contracts. Contracts can be added to or removed from policies when a user opts in or opts out, respectively. The contracts define the objects' access rights; i.e., if a contract belongs to a policy, then the associated object can use the data.

Definition 1 (Contract)**.** A contract is an expression **contract**(P, e), where P is a purpose and e is an object that belongs to a class implementing P.

For example, the expression **contract**$(Purchase, p)$ defines a contract for the *Purchase* declaration, where p is an object of type *Purchase*. We say that an object's contract *complies to a policy* if the contract belongs to the runtime representation of that policy, defined in the following.

Definition 2 (Privacy policy)**.** A privacy policy is a runtime element represented as a five-tuple (u, cp, cm, b, t), where u is the identity of the user whose data is collected, *cp* is a set of persistent contracts, *cm* is a set of mutable contracts that are updated when opting in or out, $b \in \{true, false\}$ denotes whether the collected data should be stored persistently, and $t \in \mathcal{N}$ is a natural number representing a timestamp specifying when the collected data should be deleted.

The terms u, *cp*, and *cm* in a privacy policy are initialized when a user logs in, the policy is created, and the user gives consent using **opt-in** statements, respectively. The Boolean b and the timestamp t capture retention and deletion requirements, respectively. Time is an integral part of our model. We will later specify a system clock that decrements the timestamps in all policies; data deletion is triggered when deadlines arrive.

We require a privacy policy for all personal data collected. For example, since credit card and customer data are used for different purposes, we define a new policy for each kind of data. Note that a policy can be used for different types of data if the data is used for the same purposes.

b) Consent: Here we explain how to write strings for consent statements to accurately represent intended purposes. In the declaration **purpose** P $\{\overline{Sig}^1\}$, if the method parameters in $\overline{Sig}^1$ contain personal data such as credit card and customer data, then the consent statements are "We use credit card data for P" and "We use customer data for P". Moreover, we can use the identifier X of the entity that will use the data instead of "we" and extend the statement with our retention and deletion policies. For example, the consent statement for *Purchase* becomes: "X uses your credit card number for purchasing, and your data is stored for one year" and similarly, for customer data. There is a correspondence between the consent statement and the contract of a purpose declaration. A consent statement is used to collect a user's consent, and the corresponding contract is used to control objects' access to the user's data.

c) Right to withdraw consent: In DPL, a contract can be removed from a policy anytime, and the data associated with that policy is no longer used for the withdrawn purpose. This models the user action for withdrawing consent.

d) Storage limitation: In DPL, objects encapsulate their states. An object state consists of a substitution for process-local variables, mapping variables to data, a substitution for fields, and a substitution for the object's database. Local variables are deleted after process termination, when a purpose is served. We do not allow assigning sensitive data to fields because fields store data as long as the object is alive, and all of the object's methods may have access to the fields.

Data must sometimes be stored for a time period captured by a time value t. For data storage, we integrate databases into DPL's object model, where for simplicity, our databases are just key-value stores. An object can store and retrieve data in its database, and remote access to databases is prohibited. In DPL, when the system clock advances, the timestamp of every policy is decremented. When the timestamp reaches one, the policy is deleted from the system, and data associated with the policy is also deleted from databases. Note that sensitive data in local variables can no longer be used since the policy is deleted.

e) Right to be forgotten: In DPL, a policy can be deleted anytime, and the data associated with that policy is deleted from objects' databases. This models a user action for data deletion. Deletion is also triggered when deadlines arrive. Handling these restrictions without undue delay is nontrivial, as there can be race conditions involving the time of (authorization) check and the time of use. For example, after we check compliance, a method is authorized to use data, but prior to its use, a deletion deadline may arrive or consent may be withdrawn.

In DPL, errors will be thrown if expired data is used. But race conditions between time-of-check and time-of-use mean that user-provided checks are insufficient to prevent all errors from arising. We observe though that such race-conditions are generally not critical in practice since data protection is not a hard real-time requirement. When data is deleted (or alternatively consent is revoked) "undue delay" does not mean that everything aborts and the data is instantly deleted, but rather, as soon as reasonably possible, the system will no longer process the data and it will be removed. When we use the services of Google or Facebook, our expectations for deletion are on the order of minutes or hours, not seconds.

To reflect this in DPL, we introduce *compliance scopes* where compliance is checked and assumed to remain valid within the scope. Namely, within a scope, we allow a finite number of program steps to proceed without checking compliance since the compliance was checked when entering the scope. The finite steps provide an abstract representation of the temporal notion "without undue delay." Of course, we must avoid non-terminating processes as otherwise compliance would not be checked for an arbitrary amount of time. Hence loops and recursive calls are omitted from our compliance scopes.

$$
\begin{aligned}
B &::= \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{String} \\
T &::= B \mid \mathit{PI} \mid \mathsf{Policy} \mid \mathsf{Contract} \mid \mathsf{User} \mid \mathsf{CStmt} \mid \mathsf{Key} \\
&\quad\; \mid \mathsf{Sensitive}\langle B, \mathsf{Policy}\rangle \\
\mathit{PI} &::= \textbf{\textsf{purpose}}\ P\{\overline{\mathit{Sig}}^1\} \\
\mathit{PR} &::= \overline{\mathit{PI}}\ \overline{\mathit{CL}}\ \textbf{\textsf{main}}\{\overline{T\ x}; s\} \\
\mathit{CL} &::= \textbf{\textsf{class}}\ C(\overline{T\ x})\ \textbf{\textsf{implements}}\ \overline{P}^1\{\overline{T\ x};\ \overline{M}^1\} \\
\mathit{Sig} &::= T\ m(\overline{T\ x}) \\
M &::= \mathit{Sig}\ \{\overline{T\ x};\ s\} \\
s &::= s; s \mid x := \mathit{rhs} \mid \textbf{\textsf{return}}\ e \mid \textbf{\textsf{skip}} \mid \textbf{\textsf{log-in}}() \\
&\quad\; \mid \textbf{\textsf{store}}(k, e)\ \textbf{\textsf{else}}\ \{s\} \mid \textbf{\textsf{opt-in}}(cs, cn, l) \mid \textbf{\textsf{log-out}}() \\
&\quad\; \mid \textbf{\textsf{if-comply}}(cn, \overline{e})\ \{s\}\ \textbf{\textsf{else}}\ \{s'\} \\
&\quad\; \mid \textbf{\textsf{if-consent}}(cn, l)\ \{s\}\ \textbf{\textsf{else}}\ \{s'\} \\
&\quad\; \mid \textbf{\textsf{retrieve}}(k, x)\ \{s\}\ \textbf{\textsf{else}}\ \{s'\} \mid \textbf{\textsf{collect}}(cn, l, x) \\
\mathit{rhs} &::= e \mid e.m(\overline{e}) \mid \textbf{\textsf{new}}\ C(\overline{x}) \mid \textbf{\textsf{policy}}(b, t) \\
d &::= \textbf{\textsf{contract}}(P, e) \mid \textbf{\textsf{cstmt}}(\mathit{str}) \mid \textbf{\textsf{key}}(u, \mathit{str}) \mid \mathit{str} \mid \mathit{bool} \mid \mathit{nat} \\
e &::= d \mid x \mid e\ \mathit{op}\ e \\
\mathit{op} &::= + \mid - \mid \ldots
\end{aligned}
$$

Figure 2. DPL's grammar; to simplify the presentation, let b range over Boolean expressions, u users, cs consent statements, cn contracts, l policies, P purposes, t timestamps, and k keys.

B. Syntax

We define DPL's grammar in Fig. 2. The types T are base types B, purposes *PI*, as well as types for policies, contracts, users, consent statements (CStmt), keys, and sensitive data with associated policies. A program *PR* includes purposes, classes, and a main block. A class may implement one or more purposes $\overline{P}^1$ and has methods $\overline{M}^1$. A method signature consists of a return type T, a method name m, and typed parameters declarations $\overline{T\ x}$. A method definition M consists of a signature and a body with local variables and statements. Statements s include sequential composition, assignment, and privacy-specific constructs to be discussed shortly. There is no surface syntax for directly constructing sensitive data; this happens indirectly via **collect**-statements. Right-hand-side expressions *rhs* include (pure) expressions e and method calls, as well as object and policy creation expressions, which create references at runtime. Although method calls $e.m(\overline{e})$ and return statements are standard, they can transfer sensitive data between objects. Data d consists of contracts $\textbf{\textsf{contract}}(P, e)$, where P is a purpose and e an object implementing that purpose, consent statements $\textbf{\textsf{cstmt}}(\mathit{str})$, constructed from strings, and keys $\textbf{\textsf{key}}(u, \mathit{str})$, where u is a user and *str* a string used as a tag to denote a particular attribute associated with u, in addition to strings, Boolean values, and natural numbers. Expressions e consist of data d, variables x, and operations *op* on e (e.g., logical and arithmetic operators).

DPL has the following non-assignable *reserved variables*: the self reference *this*, the self contract *cnThis*, the caller reference *caller*, the caller's contract *cnCaller*, and the reference *user* for a logged-in user.

In DPL, data can be collected only within a session, which starts with **log-in**() and ends with **log-out**(), and a user can grant or deny consent using **opt-in**(cs, cn, l) statements. Moreover, data collection is not a primitive, but rather composed from atomic statements such as:

$$\textsf{\textbf{log-in}}();\ l := \textsf{\textbf{policy}}(b, t);\ \textsf{\textbf{opt-in}}(cs, cn, l);\\ \textsf{\textbf{if-consent}}(cn, l)\{\ \textsf{\textbf{collect}}(cn, l, x)\ \};\ \textsf{\textbf{log-out}}();$$

Here, a user logs-in and a new privacy policy is created by **policy**(b, t), which returns a unique policy identity, assigned to l. Then, if the user gives consent to the consent statement *cs*, the contract *cn* is added to the policy l. The condition **if-consent**(cn, l) checks if consent has been granted for *cn* in the policy l, in which case data is collected from the user interface by **collect**(cn, l, x), under the contract *cn*; the policy l is attached to the data, and the resulting sensitive data is assigned to x. Note that we allow syntactic sugar where we omit the **else**-branches when they are not needed.

The statement **store**(k, e) checks compliance for storage and, when compliant, the data e is stored in the database, with the key k. Otherwise, the data is not stored, and the **else**-branch is executed. Conditional constructs enable the programmer to make checks to ensure GDPR compliance. These checks may be omitted, in which case the failure to comply will result in a runtime error in DPL rather than a privacy violation in non-DPL systems.

The statement **if-comply**$(cn, \overline{e})$ checks that all elements in the list $\overline{e}$ are GDPR compliant with respect to the contract *cn* and **if-consent**(cn, l) checks that consent has been granted to a contract *cn* under a policy l. The conditional constructs open and close compliance scopes in the **if**-branch. The statement **retrieve**(k, x) checks if the key k is in a database, in which case data is retrieved and assigned to x. In all three of these statements, if the check succeeds then the success branch (s) is executed and otherwise the else branch (s') is executed.

Note that to analyze the GDPR compliance of DPL programs in this paper, we capture the user actions for withdrawing consent and data deletion directly in DPL's operational semantics, such that they may occur at any time, instead of programming them explicitly in the program's surface syntax.

C. Example

We illustrate how DPL provides the essential ingredients needed to develop a GDPR compliant system, as discussed in Section II-A, by implementing the online-retailing example of Section II-B. Figure 3 presents the DPL code, focusing on consent statements, contracts, and data collection. We extend this code in Appendix D to show how DPL enforces *purpose limitation* and *storage limitation* in remote objects receiving sensitive data. User actions for data deletion and for withdrawing consent may occur at any time, reflecting the user's *right to be forgotten* and *right to withdraw consent*.

```
 1 purpose Purchase {
 2  Order purchase(String credit, String customer); ... }
 3 purpose MassMarketing{ String m-marketing(String customer);}
 4 purpose Registration{ ⟨String, Policy, Policy⟩ register(...);
 5  String getCredit(User u); String getCustomer(User u);}
 6 // Definitions of classes Purchase–c, MMarketing–c, Order, ...
 7 class Registration-c implements Registration () {
 8  ⟨String, Policy, Policy⟩ register(Contract cn1, CStmt cs1, CStmt cs2,
 9                       Contract cn2, CStmt cs3) {
10   String credit; String customer; Nat t; t := 31,536,000; // Seconds
11   log-in(); // binds a user identifier to the variable user
12   // Privacy policies for collected data items
13   Policy l1 := policy (true, t); Policy l2 := policy (true, t);
14   opt-in(cs1, cn1, l1); opt-in(cs2, cn1, l2); opt-in(cs3, cn2, l2);
15   // Data collection and storage
16   if-consent(cn1, l1) { collect(cn1, l1, credit);
17    store(key(user,“credit”), credit) };
18   if-consent(cn1, l2) { collect(cn1, l2, customer);
19    store(key(user,“customer”), customer) };
20   log-out();
21   return (⟨user, l1, l2⟩);
22  }
23  String getCredit(User u) { String credit;
24   retrieve(key(u,“credit”), credit) {
25    if-comply(cnCaller, credit){return(credit)} else { return(“0”)}
26   else { return(“0”)} }
27  String getCustomer(User u) { ... }
28 }
29 main{ String credit, customer;
30  User u; Policy l1; Policy l2; Purchase p := new Purchase-c();
31  MassMarketing mm := new MMarketing-c();
32  Registration r := new Registration-c();
33  // Consent statements and contracts
34  CStmt cs1 := cstmt(“X uses your credit card number for
35   purchasing and your data is stored for one year.”);
36  CStmt cs2 := cstmt(“X uses your customer information for
37   purchasing and your data is stored for one year.”);
38  CStmt cs3 := cstmt(“X uses your customer data for
39   mass marketing.”);
40  Contract cn1 := contract(Purchase, p);
41  Contract cn2 := contract(MassMarketing, mm);
42  // Call the register method for data collection
43  ⟨u, l1, l2⟩ := r.register(cn1, cs1, cs2, cn2, cs3);
44  if-consent(cn1, l1) { credit := r.getCredit(u) };
45  if-consent(cn1, l2) { customer := r.getCustomer(u) };
46  if-comply(cn1, (credit, customer)) {
47   Order order := p.purchase(credit, customer) };
48  if-comply(cn2, customer) {mm.m-marketing(customer)} ... }
```

Figure 3. Online-retailing example in DPL.

Lines 1–5 define purposes for Purchase, MassMarketing, and Registration. We omit the details of classes Purchase-c, MMarketing-c, and Order, and focus on the class Register, which implements data collection, and its register method (line 8). First, a user is logged-in and two privacy policies l1 and l2 are created, which both expire at a given time t (here one year, written in seconds). The **opt-in** statements (line 14) let the user grant (or deny) consent to the consent statements cs1, cs2, and cs3, in which case the associated contracts are added to the policies. Credit card and customer data are collected from the user (lines 16 and 18), returning data credit and customer of types Sensitive⟨String,l1⟩ and Sensitive⟨String,l2⟩, respectively. Keys are constructed with the tags “credit” and “customer” and

$$
\begin{aligned}
\mathit{Cfg} &::= \{\mathit{cfg}\}\\
\mathit{cfg} &::= \emptyset \mid \mathit{obj} \mid \mathit{msg} \mid \mathit{policy} \mid \mathit{class} \mid \mathit{error} \mid \mathit{cfg}\ \mathit{cfg}\\
\mathit{obj} &::= o(a, p, \mathit{db})\\
\mathit{msg} &::= m(\overline{v}, o', o, \mathit{cn}) \mid \mathit{com}(v, o) \mid n(\mathit{process})\\
\mathit{policy} &::= l(u, \mathit{cp}, \mathit{cm}, b, t)\\
\mathit{error} &::= \mathit{errorU}(o, \mathit{cn}) \mid \mathit{errorC}(o, \mathit{cn}) \mid \mathit{error}(o)\\
p &::= \mathit{process} \mid \mathit{idle}\\
\mathit{process} &::= (\sigma, s@V)\\
v &::= o \mid l \mid d \mid \mathit{sensitive}(d, l)\\
s &::= m? \mid \mathit{cScope}(S) \mid \mathit{cont}(n) \mid \ldots
\end{aligned}
$$

Figure 4. The runtime elements, where S is a set of policy-contract pairs.

are used to store the sensitive data in the database (lines 17 and 19). The **log-out**() statement ends the registration, which returns a tuple with the user identifier and the two policies. (We go slightly beyond the defined syntax here by directly returning a tuple instead of creating a result object.)

In the main block, remote calls initiate the processing of the users' data. The main block first creates objects r, p, and mm, typed by purposes. Then consent statements (lines 34–39) and contracts (lines 40–41) are defined. The consent statements and contracts are passed to the method r.register (line 43), which returns a user identifier and policies (for simplicity, we simultaneously assign to all three variables u, $l1$, and $l2$ instead of going via a result object). Then, the methods r.getCredit and r.getCustomer are called to retrieve the user's credit card and customer data, respectively (lines 44–45). Afterwards, the purchase and m-marketing methods are called (lines 47–48).

The example uses conditional constructs to avoid runtime errors. For example, in line 44, **if-consent**(cn1, l1) checks if consent for Purchase is granted, in which case the method getCredit is called. In line 46, **if-comply**(cn1, (credit, customer)) checks if the contract cn1 complies to the privacy policies of the variables credit and customer, which are passed as parameters to the method purchase. Here, the method is only called if the compliance check holds. In line 16, if consent is not granted, data is neither collected nor stored. In line 25, **if-comply**(cnCaller, credit) checks compliance with respect to the caller's contract before the **return**-statement; if compliance fails, some default value "0" is sent instead of the credit card number. Moreover, if **retrieve**(**key**(u,"credit"), credit) in line 24 fails, a default "0" is sent in line 26. We omit further error handling in this example and do not test against these default values, but remark that they play the role of ad hoc option-types, suggesting ways to further enrich DPL.

D. Operational semantics

We define DPL's operational semantics using multiset rewriting [10]. Although DPL is presented as a single-threaded system, multiple rewrite rules may be simultaneously enabled, and the execution of a program gives rise to multiple transition sequences (traces). For example, time can always advance, consent be withdrawn, or data deletion be triggered. These interleavings can give rise to race conditions between time-of-check and time-of-use for GDPR compliance.

a) Runtime elements: DPL's runtime syntax is shown in Fig. 4. A global configuration *Cfg* is a bracketed multiset of runtime elements: objects, messages, privacy policies, and classes. An *object* $o(a, p, db)$ has an identifier o, a substitution a, which maps the object's fields to values, an active process p, which may be *idle*, and a database *db*, which maps keys to data. A *process* combines a substitution σ, which maps process-local variables to values, with a runtime statement $s@V$, where s is a statement and V a compliance scope.

Messages represent process invocation, completion, and suspension. In an *invocation message* $m(\overline{v}, o', o, cn)$, m is the name of the called method, $\overline{v}$ the actual parameters, o' the callee, o the caller, and *cn* the caller's contract. A *completion message com(v,o)* contains a result value v and a receiver object o. In a *suspension message* $n(\mathit{process})$, n is a name and *process* a suspended process. *Policies* were already defined in Def. 2. Classes are simple look-up tables for method definitions, and are omitted here. We use white space to denote the composition of configurations and $\emptyset$ for the empty configuration, which is the identity for the composition.

We consider three kinds of errors for a process executing in an object o: $\mathit{errorU}(o, \mathit{cn})$ is a *data usage error* expressing that data usage in o is not compliant with the contract *cn*; $\mathit{errorC}(o, \mathit{cn})$ is a *data collection error* expressing that consent associated with the contract *cn* is not granted; and $\mathit{error}(o)$ represents other errors.

Values v include identifiers o and l for objects and privacy policies, data d, and sensitive data $\mathit{sensitive}(d, l)$. We extend the statements s of Fig. 2 as follows: $m?$ blocks an object after calling a method, *cScope* marks the end of a compliance scope; and $\mathit{cont}(n)$ schedules a suspended process n.

Substitutions bind fields, process-local variables, or keys to values, respectively. Thus, a substitution θ is a finite map, written $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. We write $\theta(x)$ to lookup the variable x in θ, and $\theta[x \mapsto v]$ to update θ with the binding $[x \mapsto v]$. In the composition $\theta \circ \theta'$, the bindings in θ' shadow those in θ, so $\theta \circ \theta'(x) = \theta'(x)$ if $x \in \mathit{dom}(\theta')$ and $\theta \circ \theta'(x) = \theta(x)$ otherwise. The notation $\theta|_C$ denotes domain restriction of θ to a set C of variables and $[]$ denotes the empty substitution.

b) Compliance: At runtime, execution happens in the context of compliance scopes. DPL features specific condition constructs to interact with these scopes. The dynamic checking of compliance is formalized by a predicate *comply* that captures our notion of compliance between policies and a contract, either by inspecting the compliance scope of the executing process or by a direct dynamic check of the configuration. Since the scope is only extended by performing a compliance check, the scope introduces a delayed effect for compliance checking in that execution may continue within the scope even after consent has been withdrawn.

Let V be a compliance scope and *ls* a list of policies. The *comply* predicate is defined inductively over a list of policies with an auxiliary predicate performing the runtime *check*:

$$\begin{aligned}
&\mathit{comply}([], \mathit{cn}, \mathit{cfg}, V) = \mathit{true}\\
&\mathit{comply}((l\!:\!\mathit{ls}), \mathit{cn}, \mathit{cfg}, V) =\\
&\quad (\langle l, \mathit{cn}\rangle \in V \ \vee\ \mathit{check}(l, \mathit{cn}, \mathit{cfg}))\ \wedge\ \mathit{comply}(\mathit{ls}, \mathit{cn}, \mathit{cfg}, V)
\end{aligned}$$

$$\mathit{check}(l, \mathit{cn}, \mathit{cfg}) = \begin{cases} \mathit{cn} \in (\mathit{cp} \cup \mathit{cm}) & \text{if } l(u, \mathit{cp}, \mathit{cm}, b, t) \in \mathit{cfg} \\ \mathit{false} & \text{otherwise} \end{cases}$$

c) The transition system: DPL's operational semantics is defined using multiset rewrite rules [10], which define a transition relation on configurations. Rewrite rules may be conditional, and we present them as inference rules with zero or more conditions as premises and a labelled transition $\mathit{lhs} \xrightarrow{l} \mathit{rhs}$ between patterns *lhs* and *rhs* as the conclusion. The rule can be applied to a sub-multiset *cfg* of a configuration if *lhs* matches *cfg* for some substitution θ and the premises hold. The rule's effect is to replace *cfg* in the global configuration with *rhs*, to which the substitution θ is applied. Matching is modulo associativity and commutativity in the multiset and hence no structural rules are needed to reorder runtime elements in configurations.

We present the rewrite system in three parts: rules for user interaction, rules for storage, deletion and scopes, and rules for the standard execution of statements. Our focus here is on sensitive data; standard rules for non-sensitive data are given in Appendix B. We have formalized the rewrite system in Maude [11], and this formalization is further described in Appendix C.

The *evaluation of expressions* is formalized by a function $[\![e]\!]_\theta$ from expressions e and substitutions θ to values. For example, **contract**(P, e) evaluates to **contract**(P, o) where o is an object reference. Evaluation is untyped: we assume that programs are well-typed such that evaluation does not get stuck and produces meaningful values. We also employ other auxiliary functions, which are briefly explained the first time they are referenced.

Article 20 of the GDPR states that the processing of data shall not adversely affect the rights and freedoms of others. Therefore, we disallow binary operations on two sensitive data items where the policies belong to different users. For simplicity, binary operations are also disallowed if the policies belong to the same user but different data types. This avoids complications regarding the withdrawal of consent from one policy. We leave open more permissive solutions, e.g., involving the intersection of (consented) purposes in policies, as future work.

d) User interaction: Figure 5 presents rewrite rules involving sessions, policies, consent, data collection, and data deletion. Labels on the transition relation represent input data from a user interface or a "tick" from an external clock. A transition is unlabeled if no input is required.

Data can only be collected and contracts only added to a privacy policy within a *session*. We first consider the effects of **log-in()** on the active process of an object. If no user is currently logged-in, the reserved variable *user* is bound to the user identifier u in LOG-IN, starting a session. Otherwise, ERROR-LOG-IN triggers an error. Rule LOG-OUT removes *user* from the local variables, ending the session. If no user is logged-in, ERROR-LOG-OUT triggers an error.

Rule POLICY creates a *privacy policy* with identifier *l*. The predicate *fresh*(*l*) expresses that the name *l* is unique in the global configuration. The persistent contracts are initialized to the contracts of o and *ob*(*main*), and the set of mutable contracts is initially empty. These sets reflect the policy for data collection; only mutable contracts can be removed by the user. If no user is logged-in, ERROR-POLICY triggers an error.

A user may grant consent to the policy for a given contract. In OPT-IN, the user accepts a consent statement *cs*, represented by the label "*yes*", and the associated contract *cn* is added to the mutable contracts *cm* of the policy *l*. If no user is logged-in, the policy does not exist in *cfg*, or the user denies consent, NO-OPT-IN formalizes that the **opt-in** has no effect. Here, *idExist*(*l*, *cfg*) expresses that a policy with identifier *l* is found in the configuration. To ensure the correct application of this predicate, the rule pattern matches over the global configuration. Note that NO-OPT-IN does not throw an error; doing so would require defining sanity checks for **opt-in** to avoid runtime errors. Instead, errors are triggered in subsequent commands if they try to use data for purposes associated to the missing contracts. In OPT-OUT, a mutable contract *cn* is removed from the policy *l*.

Data can be collected from the user under a given contract. In COLLECT, the user provides data d if the contract *cn* is compliant with the policy *l*. The data is paired with the policy and stored in the local substitution σ. In ERROR-COLLECT, a data collection error is triggered when no user is logged-in or the *comply* predicate is false.

Time advancing is captured by TICK. It applies to global configurations and is always enabled to reflect that GDPR compliance is independent of execution speed. The function *dec* decrements each policy's timestamp by one. When a policy's timestamp is one and TICK fires, *dec* deletes the policy from *cfg* and associated sensitive data from the objects' databases. Rule DELETE captures that a user can request policy deletion at any time, and the policy is deleted from the configuration. The function *del* deletes the data associated with *l* from the objects' databases. The functions *dec* and *del* are formally defined in Appendix A.

e) Storage, deletion, and scopes: Figure 6 presents rules for storage, retention, and conditional constructs. Sensitive data is stored in the object's database in STORE if the object's contract complies to the policy *l* and data storage is allowed (i.e., $b = \mathit{true}$). Expired data can never be stored, so policy compliance is checked in STORE regardless of the compliance scope. Rule NO-STORE applies when policy compliance does not hold or data storage is not allowed (the latter is captured by the negated auxiliary predicate *dataStorage*(*l*, *cfg*)).

Data from the database can be fetched to local variables in RETRIEVE, which selects the success branch s when the object's contract complies to the associated policy *l*. Otherwise the **else**-branch is selected in NO-RETRIEVE. Data can never be retrieved with an expired policy, which is reflected by the empty scope in the *comply* predicate in NO-RETRIEVE. Note

$$\textsc{log-in}\quad\dfrac{user \notin dom(\sigma)}{\begin{array}{c}o(a,(\sigma,(\textbf{log-in}();s)@V),db)\\ \xrightarrow{\text{“}u\text{”}} o(a,(\sigma[user\mapsto u],s@V),db)\end{array}}$$

$$\textsc{error-log-in}\quad\dfrac{user \in dom(\sigma)}{\begin{array}{c}o(a,(\sigma,(\textbf{log-in}();s)@V),db)\\ \rightarrow error(o)\end{array}}$$

$$\textsc{policy}\quad\dfrac{\begin{array}{c}v_1=[\![b]\!]_{a\circ\sigma}\quad v_2=[\![t]\!]_{a\circ\sigma}\\ user\in dom(\sigma)\quad u=\sigma(user)\quad fresh(l)\end{array}}{\begin{array}{c}o(a,(\sigma,(x:=\textbf{policy}(b,t);s)@V),db)\\ \rightarrow o(a,(\sigma,x:=l;s@V),db)\\ l(u,\{contract(main,ob(main)),\\ a(cnThis)\},\emptyset,v_1,v_2)\end{array}}$$

$$\textsc{log-out}\quad\dfrac{user \in dom(\sigma)}{\begin{array}{c}o(a,(\sigma,(\textbf{log-out}();s)@V),db)\\ \rightarrow o(a,(\sigma|_{dom(\sigma)\setminus\{user\}},s@V),db)\end{array}}$$

$$\textsc{error-log-out}\quad\dfrac{user \notin dom(\sigma)}{\begin{array}{c}o(a,(\sigma,(\textbf{log-out}();s)@V),db)\\ \rightarrow error(o)\end{array}}$$

$$\textsc{error-policy}\quad\dfrac{user \notin dom(\sigma)}{\begin{array}{c}o(a,(\sigma,(x:=\textbf{policy}(b,t);s)@V),db)\\ \rightarrow error(o)\end{array}}$$

$$\textsc{opt-in}\quad\dfrac{\begin{array}{c}cs=[\![e_1]\!]_{a\circ\sigma}\qquad l=[\![e_3]\!]_{a\circ\sigma}\\ cn=[\![e_2]\!]_{a\circ\sigma}\qquad user\in dom(\sigma)\end{array}}{\begin{array}{c}o(a,(\sigma,(\textbf{opt-in}(e_1,e_2,e_3);s)@V),db)\ l(u,cp,cm,b,t)\\ \xrightarrow{\text{“}yes\text{”}} o(a,(\sigma,s@V),db)\ l(u,cp,cm\cup\{cn\},b,t)\end{array}}$$

$$\textsc{no-opt-in}\quad\dfrac{\begin{array}{c}l=[\![e_3]\!]_{a\circ\sigma}\\ user\notin dom(\sigma)\vee\neg idExist(l,cfg)\vee x=\text{“}no\text{”}\end{array}}{\begin{array}{c}\{o(a,(\sigma,(\textbf{opt-in}(e_1,e_2,e_3);s)@V),db)\ cfg\}\\ \xrightarrow{x}\{o(a,(\sigma,s@V),db)\ cfg\}\end{array}}$$

$$\textsc{collect}\quad\dfrac{\begin{array}{c}user\in dom(\sigma)\qquad cn=[\![e_1]\!]_{a\circ\sigma}\qquad l=[\![e_2]\!]_{a\circ\sigma}\\ sd=sensitive(d,l)\qquad(\langle l,cn\rangle\in V\ \ \vee\ \ cn\in(cp\cup cm))\end{array}}{\begin{array}{c}o(a,(\sigma,(\textbf{collect}(e_1,e_2,x);s)@V),db)\ l(u,cp,cm,b,t)\\ \xrightarrow{\text{“}d\text{”}} o(a,(\sigma[x\mapsto sd],s@V),db)\ l(u,cp,cm,b,t)\end{array}}$$

$$\textsc{error-collect}\quad\dfrac{\begin{array}{c}cn=[\![e_1]\!]_{a\circ\sigma}\qquad l=[\![e_2]\!]_{a\circ\sigma}\\ user\notin dom(\sigma)\vee\neg comply(l,cn,cfg,V)\end{array}}{\begin{array}{c}\{o(a,(\sigma,(\textbf{collect}(e_1,e_2,x);s)@V,db)\ cfg\}\\ \xrightarrow{\text{“}d\text{”}}\{errorC(o,cn)\ cfg\}\end{array}}$$

$$\textsc{opt-out}\quad\begin{array}{c}l(u,cp,\{cn,\overline{cn}\},b,t)\\ \rightarrow l(u,cp,\{\overline{cn}\},b,t)\end{array}\qquad\textsc{tick}\quad\{cfg\}\xrightarrow{tick}\{dec(cfg)\}\qquad\textsc{delete}\quad\{l(u,cp,cm,b,t)\ cfg\}\rightarrow\{del(l,cfg)\}$$

Figure 5. Rewrite rules for user interactions.

that an object storing data cannot retrieve the data if the policy is deleted or consent is withdrawn.

Policy compliance is checked dynamically in IF-CONSENT. If the contract *cn* complies to the policy *l*, the compliance scope is extended and the **if**-branch is selected. Here, *cScope* marks the scope's end. Otherwise, the scope is unchanged and the **else**-branch is selected in NO-CONSENT. Rule CLOSE-SCOPE reduces the compliance scope at scope's end.

Compliance between expressions and a contract is checked in IF-COMPLY, where policies are extracted from the evaluated expressions and added to the compliance scope and execution continues with the **if**-branch before closing the scope. Otherwise, NO-COMPLY selects the **else**-branch. The function $policyIn(\overline{v})$ returns a list of policies from the sensitive data in $\overline{v}$ and $pairs(ls, cn)$ pairs each policy in the list of policies *ls* with the contract *cn* and returns the set of policy-contract pairs.

f) Standard Rules: Figure 7 presents the rules for standard statements, augmented to dynamically check compliance. Sensitive data can be assigned to local variables by ASSIGN-LOCAL if compliance between the object's contract and the policy is guaranteed by the scope. Otherwise, ERROR-ASSIGN triggers an error. Since sensitive data cannot be assigned to fields (see Sec. III-A), ERROR-ASSIGN-FIELD triggers an error. Object creation initialises an object with an empty database in NEW, but triggers an error in ERROR-NEW if the constructor's actual parameters contain sensitive data. The function $atts(C, \overline{v}, o)$ returns the initial substitution a for fields, where the formal parameters are bound to $\overline{v}$, the reserved variable *this* to the identifier o, and the reserved variable *cnThis* to $contract(P, o)$, where P is the purpose implemented by C.

In CALL, method calls may only occur if the called method's actual parameter values are compliant with the callee's contract (accessed via an auxiliary function $contract(o')$, where o' is the callee). In this case, a message is sent to the callee. This message includes the caller's contract $a(cnThis)$, such that the callee can check compliance before returning the method's result to the caller. The caller is blocked until it receives the result. Since the waiting time is unknown, the compliance scope is emptied and data will need to be rechecked once computation resumes. If the actual parameter

STORE

$$\frac{\begin{array}{c} sd = \llbracket e \rrbracket_{a \circ \sigma} \qquad sd = \mathit{sensitive}(d, l) \\ cn = a(\mathit{cnThis}) \qquad cn \in (cp \cup cm) \qquad b = \mathit{true} \end{array}}{\begin{array}{c} o(a, (\sigma, (\textsf{\textbf{store}}(k, e)\ \textsf{\textbf{else}}\{s\}; s')@V), db)\ l(u, cp, cm, b, t) \\ \to o(a, (\sigma, s'@V), db[k \mapsto sd])\ l(u, cp, cm, b, t) \end{array}}$$

NO-STORE

$$\frac{\begin{array}{c} \neg \mathit{dataStorage}(l, \mathit{cfg})\ \vee\ \neg \mathit{comply}(l, cn, \mathit{cfg}, \emptyset) \\ \mathit{sensitive}(d, l) = \llbracket e \rrbracket_{a \circ \sigma} \qquad cn = a(\mathit{cnThis}) \end{array}}{\begin{array}{c} \{\mathit{cfg}\ o(a, (\sigma, (\textsf{\textbf{store}}(k, e)\ \textsf{\textbf{else}}\{s\}; s')@V), db)\} \\ \to \{\mathit{cfg}\ o(a, (\sigma, (s; s')@V), db)\} \end{array}}$$

RETRIEVE

$$\frac{\begin{array}{c} sd = db(k) \qquad sd = \mathit{sensitive}(d, l) \\ x \in \mathit{dom}(\sigma) \qquad cn = a(\mathit{cnThis}) \qquad cn \in (cp \cup cm) \end{array}}{\begin{array}{c} o(a,(\sigma, (\textsf{\textbf{retrieve}}(k, x)\{s\}\ \textsf{\textbf{else}}\{s'\}; s'')@V),db) \\ l(u, cp, cm, b, t) \\ \to o(a, (\sigma[x \mapsto sd], s; s''@V), db) \\ l(u, cp, cm, b, t) \end{array}}$$

NO-RETRIEVE

$$\frac{\begin{array}{c} x \notin \mathit{dom}(\sigma)\ \vee\ k \notin \mathit{dom}(db)\ \vee\ cn = a(\mathit{cnThis}) \\ \mathit{sensitive}(d, l) = db(k) \qquad \neg \mathit{comply}(l, cn, \mathit{cfg}, \emptyset) \end{array}}{\begin{array}{c} \{\mathit{cfg}\ o(a,(\sigma, (\textsf{\textbf{retrieve}}(k, x)\{s\}\ \textsf{\textbf{else}}\{s'\}; s'')@V),db)\} \\ \to \{\mathit{cfg}\ o(a, (\sigma, (s'; s'')@V), db)\} \end{array}}$$

IF-CONSENT

$$\frac{cn = \llbracket e_1 \rrbracket_{a \circ \sigma} \qquad l = \llbracket e_2 \rrbracket_{a \circ \sigma} \qquad cn \in (cp \cup cm)}{\begin{array}{c} o(a, (\sigma, (\textsf{\textbf{if-consent}}(e_1, e_2)\{s\}\ \textsf{\textbf{else}}\{s'\}; s'')@V), db)\quad l(u, cp, cm, b, t) \\ \to o(a, (\sigma, (s; \mathit{cScope}(\langle l, cn \rangle); s'')@V \cup \{\langle l, cn \rangle\}), db)\quad l(u, cp, cm, b, t) \end{array}}$$

CLOSE-SCOPE

$$\begin{array}{c} o(a, (\sigma, (\mathit{cScope}(S); s)@V), db) \\ \to o(a, (\sigma, s@V \backslash S), db) \end{array}$$

IF-COMPLY

$$\frac{\overline{v} = \llbracket \overline{e} \rrbracket_{a \circ \sigma} \quad \mathit{ls} = \mathit{policyIn}(\overline{v}) \quad \mathit{comply}(\mathit{ls}, cn, \mathit{cfg}, \emptyset) \quad S = \mathit{pairs}(\mathit{ls}, cn)}{\begin{array}{c} \{\mathit{cfg}\ o(a, (\sigma, (\textsf{\textbf{if-comply}}(cn, \overline{e})\{s\}\ \textsf{\textbf{else}}\{s'\}; s'')@V), db)\} \\ \to \{\mathit{cfg}\ o(a, (\sigma, (s; \mathit{cScope}(S); s'')@V \cup S), db)\} \end{array}}$$

NO-CONSENT

$$\frac{\begin{array}{c} \neg \mathit{comply}(l, cn, \mathit{cfg}, \emptyset \\ cn = \llbracket e_1 \rrbracket_{a \circ \sigma} \qquad l = \llbracket e_2 \rrbracket_{a \circ \sigma}) \end{array}}{\begin{array}{c} \{\mathit{cfg}\ o(a, (\sigma, (\textsf{\textbf{if-consent}}(e_1, e_2)\{s\} \\ \qquad\qquad \textsf{\textbf{else}}\{s'\}; s'')@V), db)\} \\ \to \{\mathit{cfg}\ o(a, (\sigma, (s'; s'')@V), db)\} \end{array}}$$

NO-COMPLY

$$\frac{\overline{v} = \llbracket \overline{e} \rrbracket_{a \circ \sigma} \qquad \mathit{ls} = \mathit{policyIn}(\overline{v}) \qquad \neg \mathit{comply}(\mathit{ls}, cn, \mathit{cfg}, \emptyset)}{\begin{array}{c} \{\mathit{cfg}\ o(a, (\sigma, (\textsf{\textbf{if-comply}}(cn, \overline{e})\{s\}\textsf{\textbf{else}}\{s'\}; s'')@V, db)\} \\ \to \{\mathit{cfg}\ o(a, (\sigma, (s'; s'')@V), db)\} \end{array}}$$

Figure 6. Rewrite rules for data storage, deletion and scopes.

values are not compliant with the contract, ERROR-CALL triggers a data usage error. The callee receives the message in CALLEE-INVC. The function *class*(o) returns the class of object o and *bind*$(o, C, m, \overline{v}, o', cn)$ creates a new process $(\sigma, s@\emptyset)$, where the reserved variable *caller* is bound to o' in σ, the reserved variable *cnCaller* to the caller's contract *cn*, and the formal parameters to $\overline{v}$. Moreover, s is the method body of m in the class C and the compliance scope is empty. Data may have expired and needs to be checked using the appropriate conditional constructs in the method body.

Upon method completion with sensitive data as the return value, RETURN sends a completion message to the caller if the caller's contract, which is stored in the callee's local variable *cnCaller*, complies with the policy *l*. Otherwise, ERROR-RETURN triggers a data usage error. Observe that the error can be avoided by testing the caller's contract; e.g., **if-comply**$(\mathit{cnCaller}, e)\{$**return**$(e)\}$. The caller receives the completion message and gets unblocked in GET-DATA. The data might have expired before the message is received, potentially triggering an error in subsequent statements.

Self calls are supported by SELF-CALL. (Observe that cyclic call chains give rise to deadlock in our semantics; these could be handled with scheduling messages by adapting the pattern of SELF-CALL to blocked objects with incoming calls.) In SELF-CALL, the compliance scope is emptied so the calling process will need to recheck compliance when it resumes. The new process, also with an empty compliance scope, ends with a *cont(n)* statement and the old process is wrapped in a scheduling message. These are matched to resume execution in SELF-RETURN, provided that the object's contract, found in the field *cnThis*, complies with the policy. Otherwise, ERROR-SELF-RETURN triggers a data usage error.

The initial state is derived from the main block **main**$\{\overline{T\ x}; s\}$ by creating an object *ob(main)*$([\mathit{cnThis} \mapsto \mathit{contract}(\mathit{main}, \mathit{ob(main)})], ([], s@\emptyset), [])$, with identity *ob(main)* and contract *contract*$(\mathit{main}, \mathit{ob(main)})$. Note that this contract must be added to created policies so that the *ob*(main) object can access sensitive data. In the object, the local substitution and the database are empty, and the active process $([], s@\emptyset)$ corresponds to the activation of **main**'s statements s.

ASSIGN-LOCAL

$$\frac{\begin{array}{c} cn = a(\mathit{cnThis}) \quad x \in \mathit{dom}(\sigma) \\ \langle l, cn\rangle \in V \ \vee \ cn \in (cp \cup cm) \\ sd = \llbracket e \rrbracket_{a\circ\sigma} \quad sd = \mathit{sensitive}(d,l) \end{array}}{\begin{array}{c} o(a, (\sigma, (x := e; s)@V), db) \\ l(u, cp, cm, b, t) \rightarrow \\ o(a, (\sigma[x \mapsto sd], s@V), db) \\ l(u, cp, cm, b, t) \end{array}}$$

ERROR-ASSIGN-FIELD

$$\frac{\begin{array}{c} x \in \mathit{dom}(a) \\ \mathit{sensitive}(d,l) = \llbracket e \rrbracket_{a\circ\sigma} \end{array}}{\begin{array}{c} o(a, (\sigma, (x := e; s)@V), db) \\ \rightarrow \mathit{error}(o) \end{array}}$$

NEW

$$\frac{\begin{array}{c} \overline{d} = \llbracket \overline{e} \rrbracket_{a\circ\sigma} \qquad a' = \mathit{atts}(C, \overline{d}, o') \\ \mathit{fresh}(o') \qquad \mathit{contract}(P, o') = a'(\mathit{cnThis}) \end{array}}{\begin{array}{c} o(a, (\sigma, (x := \textbf{new } C(\overline{e}); s)@V), db) \rightarrow \\ o(a, (\sigma, (x := o'; s)@V), db) \ \ o'(a', \mathit{idle}, []) \end{array}}$$

ERROR-ASSIGN

$$\frac{\begin{array}{c} x \in \mathit{dom}(\sigma) \quad \mathit{sensitive}(d,l) = \llbracket e \rrbracket_{a\circ\sigma} \\ cn = a(\mathit{cnThis}) \quad \neg\mathit{comply}(l, cn, \mathit{cfg}, V) \end{array}}{\begin{array}{c} \{o(a, (\sigma, (x := e; s)@V), db) \ \mathit{cfg}\} \\ \rightarrow \{\mathit{errorU}(o, cn) \ \mathit{cfg}\} \end{array}}$$

ERROR-NEW

$$\frac{sd \in \llbracket \overline{e} \rrbracket_{a\circ\sigma} \qquad sd = \mathit{sensitive}(d,l)}{\begin{array}{c} o(a, (\sigma, (x := \textbf{new } C(\overline{e}); s)@V), db) \\ \rightarrow \mathit{error}(o) \end{array}}$$

CALL

$$\frac{\begin{array}{c} o' = \llbracket e \rrbracket_{a\circ\sigma} \quad \overline{v} = \llbracket \overline{e} \rrbracket_{a\circ\sigma} \quad cn = \mathit{contract}(o') \\ ls = \mathit{policyIn}(\overline{v}) \quad \mathit{comply}(ls, cn, \mathit{cfg}, V) \end{array}}{\begin{array}{c} \{o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db) \ \mathit{cfg}\} \rightarrow \\ \{o(a, (\sigma, (x := m?; s)@\emptyset), db) \ m(\overline{v}, o', o, a(\mathit{cnThis})) \ \mathit{cfg}\} \end{array}}$$

ERROR-CALL

$$\frac{\begin{array}{c} o' = \llbracket e \rrbracket_{a\circ\sigma} \quad \overline{v} = \llbracket \overline{e} \rrbracket_{a\circ\sigma} \quad cn = \mathit{contract}(o') \\ ls = \mathit{policyIn}(\overline{v}) \quad \neg\mathit{comply}(ls, cn, \mathit{cfg}, V) \end{array}}{\begin{array}{c} \{o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db) \ \mathit{cfg}\} \\ \rightarrow \{\mathit{errorU}(o, cn) \ \mathit{cfg}\} \end{array}}$$

RETURN

$$\frac{\begin{array}{c} sd = \llbracket e \rrbracket_{a\circ\sigma} \quad sd = \mathit{sensitive}(d,l) \quad o' = \sigma(\mathit{caller}) \\ cn = \sigma(\mathit{cnCaller}) \quad \langle l, cn\rangle \in V \ \vee \ cn \in (cp \cup cm) \end{array}}{\begin{array}{c} o(a, (\sigma, \textbf{return } e \ @V), db) \ l(u, cp, cm, b, t) \rightarrow \\ o(a, \mathit{idle}, db) \ l(u, cp, cm, b, t) \ \mathit{com}(sd, o') \end{array}}$$

CALLEE-INVC

$$\frac{\begin{array}{c} (\sigma, s@\emptyset) = \mathit{bind}(o, C, m, \overline{v}, o', cn) \\ C = \mathit{class}(o) \quad \sigma(\mathit{caller}) = o' \quad \sigma(\mathit{cnCaller}) = cn \end{array}}{\begin{array}{c} \{o(a, \mathit{idle}, db) \ m(\overline{v}, o, o', cn) \ \mathit{cfg}\} \\ \rightarrow \{o(a, (\sigma, s@\emptyset), db) \ \mathit{cfg}\} \end{array}}$$

ERROR-RETURN

$$\frac{\begin{array}{c} \mathit{sensitive}(d,l) = \llbracket e \rrbracket_{a\circ\sigma} \quad o' = \sigma(\mathit{caller}) \\ cn = \sigma(\mathit{cnCaller}) \quad \neg\mathit{comply}(l, cn, \mathit{cfg}, V) \end{array}}{\begin{array}{c} \{o(a, (\sigma, \textbf{return } e \ @V), db) \ \mathit{cfg}\} \rightarrow \\ \{\mathit{errorU}(o, cn) \ \mathit{cfg}\} \end{array}}$$

SELF-CALL

$$\frac{\begin{array}{c} o = \llbracket e \rrbracket_{a\circ\sigma} \quad \overline{v} = \llbracket \overline{e} \rrbracket_{a\circ\sigma} \\ (\sigma', s'@\emptyset) = \mathit{bind}(o, C, m, \overline{v}, o, a(\mathit{cnThis})) \\ \mathit{fresh}(n) \quad ls = \mathit{policyIn}(\overline{v}) \quad \mathit{comply}(ls, a(\mathit{cnThis}), \mathit{cfg}) \end{array}}{\begin{array}{c} \{o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db) \ \mathit{cfg}\} \rightarrow \\ \{o(a, (\sigma', (s'; \mathit{cont(n)})@\emptyset), db) \ n(\sigma, (x := m?; s)@\emptyset) \ \mathit{cfg}\} \end{array}}$$

SELF-RETURN

$$\frac{\begin{array}{c} a(\mathit{cnThis}) \in (cp \cup cm) \\ o = \sigma(\mathit{caller}) \quad v = \llbracket e \rrbracket_{a\circ\sigma} \end{array}}{\begin{array}{c} o(a, (\sigma, (\textbf{return } e; \mathit{cont(n)})@V), db) \\ n(\sigma', (x := m?; s)@V') \ l(u, b, cp, cm, t) \rightarrow \\ o(a, (\sigma'[x \mapsto v], s@V'), db) \ l(u, b, cp, cm, t) \end{array}}$$

GET-DATA

$$\begin{array}{c} o(a, (\sigma, (x := m?; s)@V), db) \ \mathit{com}(v, o) \\ \rightarrow o(a, (\sigma[x \mapsto v], s@V), db) \end{array}$$

ERROR-SELF-RETURN

$$\frac{\begin{array}{c} \neg\mathit{comply}(l, cn, \mathit{cfg}, V) \quad cn = a(\mathit{cnThis}) \\ \mathit{sensitive}(d,l) = \llbracket e \rrbracket_{a\circ\sigma} \quad o = \sigma(\mathit{caller}) \end{array}}{\begin{array}{c} \{o(a, (\sigma, (\textbf{return } e; \mathit{cont(n)})@V), db) \ \mathit{cfg}\} \\ \rightarrow \{\mathit{errorU}(o, cn) \ \mathit{cfg}\} \end{array}}$$

ERROR-SELF-CALL

$$\frac{\begin{array}{c} \neg\mathit{comply}(ls, a(\mathit{cnThis}), \mathit{cfg}, V) \\ o = \llbracket e \rrbracket_{a\circ\sigma} \quad \overline{v} = \llbracket \overline{e} \rrbracket_{a\circ\sigma} \quad ls = \mathit{policyIn}(\overline{v}) \end{array}}{\begin{array}{c} \{o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db) \ \mathit{cfg}\} \\ \rightarrow \{\mathit{errorU}(o, cn) \ \mathit{cfg}\} \end{array}}$$

Figure 7. Rewrite rules for standard statements.

IV. Correctness

DPL's operational semantics gives rise to a transition system, where states are configurations and transitions correspond to rule applications. There are infinitely many initial states reflecting the starting configurations of infinitely many programs. Moreover, a program may give rise to a nonterminating application of rules and, therefore, also infinitely many states.

We reason about this infinite state transition system and prove that DPL programs cannot lead to GDPR violations with respect to the requirements given in Section II-A. Namely, we formalize properties that ensure purpose limitation, consent, the right to withdraw consent, storage limitation, and the right to be forgotten. Formal definitions and proofs are given in Appendix A.

In our formalization, we define a trace as a sequence of configuration and action pairs $(cfg_0, R_0), (cfg_1, R_1), \cdots$, where an action is the name of the rule that is fired at the configuration (i.e., $cfg_i \rightarrow cfg_{i+1}$ by applying the rule R_i).

Auxiliary formulas	Explanation
$use(o, \langle d, l\rangle, cn)$	the object o uses the data $\langle d, l\rangle$ for a purpose associated with the contract cn
$complyTo(l, cn)$	the policy l exists and the contract cn complies to the policy
$checked\text{-}scope(o, \langle l, cn\rangle)$	the pair $\langle l, cn\rangle$ is in o's compliance scope
$noExecIn(o)$	the current execution step is not in o
$errorU(o, cn)$	a data usage error associated with the contract cn occurred in the object o
$errorC(o, cn)$	a data collection error in o when consent associated with the contract cn is not granted
$collect(o, cn, l)$	o is executing a **collect**(cn, l, x) statement
$optedIn(l, cn)$	the contract cn is added to the policy l
$optedOut(l, cn)$	the contract cn is removed from the policy l
$expired(l)$	the policy l is deleted from the configuration
$dbDel(l)$	sensitive data associated with the policy l is deleted from databases
$deleted(l)$	the policy l is deleted

Table II
EXPLANATION OF PREDICATES.

To formalize and reason about temporal properties of DPL programs, we use linear temporal logic (LTL) [14], [15] with the standard temporal operators: $\bigcirc$ (next), $\Box$ (always), $\Diamond$ (sometime), and *Until* and W, which are the strong and weak until operators respectively. The notation $\models \varphi$ denotes that the LTL property φ holds for all traces of our transition system.

Table II shows the state formulas we use and their informal interpretation. Note that some of our definitions state that predicates must eventually hold and for this to be the case we require a fair transition system. Since the rules TICK, DELETE, and OPT-OUT can fire infinitely often and at anytime, we specify strong fairness for our transition system, where if a rule is enabled infinitely often, then it fires infinitely often. We express strong fairness for our transition system as follows.

$$fair = \Box\Diamond enabled_1 \Rightarrow \Box\Diamond fired_1 \;\wedge\; \cdots \;\wedge\; \Box\Diamond enabled_i \Rightarrow \Box\Diamond fired_i.$$

In this formula, the index i ranges over the names of our rules, $enabled_i$ is true (i.e., satisfied at a given point in a trace) when the rule i is enabled namely, the premises of the rule are true and the left-hand-side of the rule matches the current configuration, and $fire_i$ is true when the rule i is fired. We shall assume strong fairness when proving all our properties.

The following property $\mathscr{P}$ formalizes purpose limitation, where compliance is checked for any data usage. Namely, an object cannot use data for a purpose that is not compliant with the policy, and an error arises if the object attempts a non-compliant usage. In DPL, the only statements using data are assignments, calls, and **return**-statements. The *use* formula is true if one of these three statements is the first statement in the active process of an object. The property $\mathscr{P}_1$ says that if whenever *use* is true, then *checked-scope* is true (i.e., if compliance is checked prior to the usage in the appropriate conditional construct), then a data usage error never arises. Note that $\mathscr{P}_1$ holds regardless of whether the *complyTo* formula is true or false. The property $\mathscr{P}_2$ says that if data is used and the formulas *complyTo* and *checked-scope* are false, then in the next step, execution does not continue in the object until a data usage error arises or the user opts in.

$$\begin{aligned}
&\mathscr{P} = \mathscr{P}_1 \;\wedge\; \mathscr{P}_2\\
&\mathscr{P}_1 = \forall o, d, l, cn.\\
&\quad \Box(use(o, \langle d, l\rangle, cn) \Rightarrow checked\text{-}scope(o, \langle l, cn\rangle))\\
&\quad\; \Rightarrow \Box\neg errorU(o, cn)\\
&\mathscr{P}_2 = \forall o, d, l, cn.\\
&\quad \Box((use(o, \langle d, l\rangle, cn) \;\wedge\; \neg complyTo(l, cn)\\
&\quad\; \wedge\; \neg checked\text{-}scope(o, \langle l, cn\rangle))\\
&\quad\; \Rightarrow \bigcirc(noExecIn(o)\ Until\ (errorU(o, cn) \;\vee\; optedIn(l, cn))))
\end{aligned}$$

Theorem IV.1 (Purpose limitation)**.** *The property $\mathscr{P}$ holds for all traces of our transition system.*

The proof of this theorem and all theorems in this section can be found in Appendix A.

The following property $\mathscr{C}$ formalizes that data is collected only if consent has been granted. The formula *collect* is true if there is a **collect**(cn, l, x) statement in the active process of an object as the first statement. The property $\mathscr{C}_1$ says that if **collect**(cn, l, x) is always in a conditional construct that checks compliance between a policy l and a contract cn, then a data collection error never arises. The property $\mathscr{C}_2$ says that when collecting data, if the formulas *complyTo* and *checked-scope* are false, then in the next step, execution does not continue in the object until a data collection error arises or the user opts in.

$$\begin{aligned}
&\mathscr{C} = \mathscr{C}_1 \;\wedge\; \mathscr{C}_2\\
&\mathscr{C}_1 = \forall o, l, cn.\\
&\quad \Box(collect(o, l, cn) \Rightarrow checked\text{-}scope(o, \langle l, cn\rangle))\\
&\quad\; \Rightarrow \Box\neg errorC(o, cn)\\
&\mathscr{C}_2 = \forall o, l, cn.\\
&\quad \Box((collect(o, l, cn) \;\wedge\\
&\quad \neg complyTo(l, cn) \;\wedge\; \neg checked\text{-}scope(o, \langle l, cn\rangle))\\
&\quad\; \Rightarrow \bigcirc(noExecIn(o)\ Until\ (errorC(o, cn) \;\vee\; optedIn(l, cn))))
\end{aligned}$$

Theorem IV.2 (Consent)**.** *The property $\mathscr{C}$ holds for all traces of our transition system.*

The following property $\mathscr{W}$ formalizes the right to withdraw consent: A user can withdraw consent, and the corresponding purpose is removed from the policy, which prevents subsequently using the data for that purpose. The formula $optedOut(l, cn)$ is true when the contract cn is removed from the policy l. The formula $optedIn(l, cn)$ is true when the contract cn is added to the policy l. The property $\mathscr{W}$ says that when $optedOut(l, cn)$ is true, then compliance with respect to the policy and the contract remains false until $optedIn(l, cn)$ is true (if it ever becomes true, hence we use LTL's weak-until operator).

$$\begin{aligned}
&\mathscr{W} = \forall l, cn.\\
&\quad \Box(optedOut(l, cn) \Rightarrow (\neg complyTo(l, cn)\ W\ optedIn(l, cn)))
\end{aligned}$$

Theorem IV.3 (Right to withdraw consent)**.** *The property $\mathscr{W}$ holds for all traces of our transition system.*

The following property $\mathscr{S}$ formalizes storage limitation: Data is deleted from the objects' databases when the deadline for data deletion arrives. It says that if the formula *expired*(l) is true (due to policy expiration or the DELETE rule), then data with that policy is deleted from the objects' databases. In the rules TICK and DELETE, policy deletion and data deletion are specified using functions and equations. Thus when a policy is deleted, its data is deleted in the same state as well.

$$\mathscr{S} = \forall l.\ \Box(\mathit{expired}(l) \Rightarrow \mathit{dbDel}(l))$$

Theorem IV.4 (Storage limitation)**.** *The property $\mathscr{S}$ holds for all traces of our transition system.*

The following property $\mathscr{F}$ formalizes the right to be forgotten: A user can request to delete her data, and the data is then deleted from the objects' databases. The predicate *deleted*(l) is true when the policy l is deleted by the rule DELETE. The property $\mathscr{F}$ says that when a policy is deleted, the formula *expired* is true for that policy. Moreover, by Theorem IV.4, data with that policy is deleted from databases.

$$\mathscr{F} = \forall l.\ \Box(\mathit{deleted}(l) \Rightarrow \mathit{expired}(l))$$

Theorem IV.5 (Right to be forgotten)**.** *The property $\mathscr{F}$ holds for all traces of our transition system.*

As mentioned, attempted GDPR violations do not succeed; they instead produce runtime errors. These errors can be systematically avoided by the following *hygienic measures*, which amount to good GDPR practice: i) all methods that use personal data are specified in the corresponding purpose declarations; ii) for each purpose declaration, the corresponding consent statement and contract are specified; iii) the pattern in Section III-B is used for data collection, which requires a logged-in user, a privacy policy for the user, opt-in options, and the appropriate **if-consent** construct prior to data collection; and iv) the construct **if-comply**$(cn, \overline{e})$ is used prior to the commands that use sensitive data in $\overline{e}$. That these hygienic measures are sufficient follows by careful inspection of the rules, and we also confirm this for concrete programs by model checking hygienic and non-hygienic programs (see Appendix C for some examples).

V. Maude formalization

We have formalized DPL's operational semantics in Maude [11]. This yields a prototype environment for simulating DPL programs and thereby provides us some confidence in our rules. Maude supports multiset rewriting logic, which we use to model the non-deterministic behavior of our system, where enabled rules can be interleaved at any point after each rewrite. We use a user object to non-deterministically give "*yes*" or "*no*" input to **opt-in** statements.

Our Maude formalization provides a prototype verification environment for DPL programs. It can be used to check all our properties on finite-state programs, such as the program given in Fig. 3. (Technically, the programs may be infinite state, e.g., involve data from unbounded domains, provided only finitely many states are reachable.) We also validate statements about "hygienic programs" given in Section IV. We check $\mathscr{P}_1$, $\mathscr{C}_1$, $\mathscr{W}$, $\mathscr{S}$, and $\mathscr{F}$ for the hygienic program given in Fig. 3, where all the usage constructs are protected with conditional constructs. Maude verifies that errors never arise and all specified properties hold. We checked the properties $\mathscr{P}_2$ and $\mathscr{C}_2$ concerning programs raising errors on the program where different combinations of the conditional constructs are removed. Maude verifies that the corresponding errors arise and the properties hold. We present our results with Maude in Appendix C.

VI. Related Work

Purpose-based access control mechanisms [16], [17] have been proposed to control access to data in databases based on intended purpose information associated with the data. Users must state their access purposes when requesting data and can access the data if their stated purposes comply with the data's intended purpose. Both kinds of purposes are organized hierarchically, and compliance is defined based on a partial-order relation. In contrast, we enforce GDPR requirements using programming language constructs and runtime checks. In DPL, intended purposes are added to policies when users give consent, and policies can change over time when data subjects consent to new purposes or withdraw their consent.

Privacy by Design (PbD) [18] is a framework that introduces principles that should be considered when designing a system architecture. Schneider [19] explains that since a model specification can be very different from the implementation, PbD, by itself, cannot in general guarantee privacy unless it also encompasses the implementation. We believe that languages like DPL have an important role to play in building privacy-by-design systems as DPL directly supports many of the principles espoused there. For example, it is a proactive approach that provides privacy by default.

Another design-oriented approach is [13], where GDPR compliance is checked at the design level. Business processes represent one or more purposes, and formal models of inter-process communication identify data collection and data usage points [13]. In order to identify purposes and data usage points, we build on the approach proposed in [13]. However, instead of business processes, the methods that implement a process are grouped together in a purpose declaration. Moreover, we provide language support for policy enforcement, whereas [13] only supports data protection through audits.

Researchers have used information-flow analysis [20], [21] to check privacy policy compliance in programs. There, types are annotated with privacy policy labels, and a notion of policy compliance is defined. We also track the flow of sensitive data, where left-hand-side expressions get the policy identity of right-hand-expressions. We expand upon the most closely related work here in the following.

In [22], the authors propose a decentralized label model for Jif to enforce role-based access control in programs. Jif principals represent entities with specific roles, which have a hierarchical structure. Program variables are annotated with policy labels, where a label contains the principal that owns the data and a set of readers who can read the data. Jif's type checking enforces information-flow control and protects principals' privacy. Moreover, a principal may declassify the label of the data that it owns. Declassification of roles requires runtime checking to determine whether a process is authorized to declassify data. However, most of a program can be certified statically with no overhead. In contrast, we enforce richer data protection policies, as required by the GDPR, such as the necessity of providing consent, the right to withdraw consent, purpose limitation, storage limitation, and the right to be forgotten. GDPR temporal requirements, where users can withdraw consent or data is automatically deleted when deadlines arrive cannot be enforced statically. In DPL, we enforce these requirements by runtime checking.

In [3], the authors enforce purpose-based and storage-based restrictions in Jif. Jif's principals represent purposes, ordered hierarchically. Data is annotated with the principal that owns the data, representing the purpose for which the data is collected. Methods that are needed for a purpose are annotated with the corresponding principal. By means of Jif's type checking, compile-time errors arise if data is used for non-compliant purposes. Similarly, for enforcing retention restrictions, Jif's principals represent retention labels. In [3], storage restrictions are limited to retention labels (in the sense of P3P), intended purposes are fixed, and access purposes are associated by the programmer. In DPL, users' consent determines the intended purposes in policies, and access purposes are automatically associated with created objects. Moreover, in DPL, instead of policies themselves, references to policies are attached to data, and thus if a policy changes due to the user actions or the passage of time, then this change is automatically enforced on any usage or storage of data with the policy reference. In addition, this requires less storage overhead at runtime and we can also enforce deletion policies.

In [4], the authors propose a static approach to check privacy policy compliance in Bing, where privacy policies are specified in LEGALEASE, and GROK maps data types in a code to policies and tracks the flow of information. By taking a static approach, there are no runtime overheads. However, their approach cannot be used to enforce the GDPR requirements such as consent, the right to withdraw consent, the right to be forgotten, and temporal requirements for data deletion, where a runtime approach would be required.

VII. Conclusion

We have presented DPL, a programming language designed for data protection with provable guarantees. DPL and our Maude-based simulation environment are prototypical and allow us to simulate programs and experiment with our new language features. Our initial experience supports the thesis that custom language support can play an important role in building systems meeting strict data protection requirements like those of the GDPR.

Our work constitutes a first significant step in building a robust, usable language with formal GDPR guarantees. This work could be strenghtened in several ways, which suggest interesting directions for future work: (1) Large-scale case studies are needed to better assess the language's usability and further evaluate the runtime overheads involved. (2) For stronger correctness results, our pen-and-paper proofs (in Appendix A) could be further formalized in a theorem prover such as Coq or Isabelle. (3) A type and effect system for DPL could be used to enforce the correct use of scopes, with an associated type preservation theorem. This would make programming in DPL easier by eliminating runtime errors for well-typed programs. We have also highlighted other possibilities for future work in this paper's body. This includes defining fine-grained compliance scopes in programs, developing a more permissive solution for binary and general operations on data items with different policies, and, finally, building real language support.

Acknowledgment

This work was partly funded by the Research Council of Norway through *IoTSec* (project no. 248113).

References

[1] "Regulation 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)," *Official Journal of the European Union*, p. L119:1–88, April 2016.

[2] N. N. Kumar and R. Shyamasundar, "Realizing purpose-based privacy policies succinctly via information-flow labels," in *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. IEEE, 2014, pp. 753–760.

[3] K. Hayati and M. Abadi, "Language-based enforcement of privacy policies," in *International Workshop on Privacy Enhancing Technologies*. Springer, 2004, pp. 302–313.

[4] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing, "Bootstrapping privacy compliance in big data systems," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 327–342.

[5] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen, "ABS: A core language for abstract behavioral specification," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2010, pp. 142–164.

[6] V. Serbanescu, C. Nagarajagowda, K. Azadbakht, F. de Boer, and B. Nobakht, "Towards type-based optimizations in distributed applications using ABS and Java 8," in *International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. Springer, 2014, pp. 103–112.

[7] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2-3, pp. 202–220, 2009.

[8] D. Wyatt, *Akka concurrency*. Artima Incorporation, 2013.

[9] D. Caromel and L. Henrio, *A Theory of Distributed Objects: Asynchrony-Mobility-Groups-Components*. Springer, 2005.

[10] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical computer science*, vol. 96, no. 1, pp. 73–155, 1992.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Springer, 2007.

[12] P. Voigt and A. von dem Bussche, *The EU general data protection regulation (GDPR): A Practical Guide*. Springer, 2017.

[13] D. Basin, S. Debois, and T. Hildebrandt, "On purpose and by necessity: compliance under the gdpr," in *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 20–37.

[14] N. Markey, "Temporal logic with past is exponentially more succinct," *Bulletin-European Association for Theoretical Computer Science*, vol. 79, pp. 122–128, 2003.

[15] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[16] J.-W. Byun, E. Bertino, and N. Li, "Purpose based access control of complex data for privacy protection," in *Proceedings of the tenth ACM symposium on Access control models and technologies*, 2005, pp. 102–110.

[17] J.-W. Byun and N. Li, “Purpose based access control for privacy protection in relational database systems,” *The VLDB Journal*, vol. 17, no. 4, pp. 603–619, 2008.

[18] A. Cavoukian, “Privacy by design: origins, meaning, and prospects for assuring privacy and trust in the information era,” in *Privacy protection measures and technologies in business organizations: aspects and standards*. IGI Global, 2012, pp. 170–208.

[19] G. Schneider, “Is privacy by construction possible?” in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 471–485.

[20] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[21] A. Russo and A. Sabelfeld, “Dynamic vs. static flow-sensitive security analysis,” in *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, 2010, pp. 186–199.

[22] A. C. Myers and B. Liskov, “Protecting privacy using the decentralized label model,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 4, pp. 410–442, 2000.

Appendix

A. Proofs

We prove that the properties in Section IV hold for all strongly fair runs of our transition system. To define the fairness and some of our formulas, we must track the current rule that is executed and the object’s identity that the rule is applied to. We define a trace τ as a sequence of tuples, such as $\tau = (\mathit{cfg}_0, R_0, \mathit{Id}_0), (\mathit{cfg}_1, R_1, \mathit{Id}_1), \cdots$, that in addition to the configuration *cfg* and the rule label R, also tracks the identity *Id* of the object, which can be an object’s identity or *other*, when the rules TICK, DELETE, and OPT-OUT are fired. We define formulas on a state denoted by $\phi(\mathit{cfg})$, or on tuples, denoted by $\phi(\mathit{cfg}, R, \mathit{Id})$ as needed. We formalize the formulas in Table II as follows.

$$\begin{aligned}
&\mathit{use}(o, \mathit{sensitive}(d,l), \mathit{cn})(\mathit{cfg}) = \exists a, \sigma, x, e, e', \overline{e}, m, s, s', V, \mathit{db}, o'.\\
&\quad o(a, (\sigma, (s; s')@V), \mathit{db}) \in \mathit{cfg} \;\wedge\\
&\quad (s = x := e \;\wedge\; \mathit{sensitive}(d,l) = [\![e]\!]_{a\circ\sigma} \;\wedge\; \mathit{cn} = a(\mathit{cnThis})) \;\vee\\
&\quad (s = \textbf{return}(e) \;\wedge\; \mathit{sensitive}(d,l) = [\![e]\!]_{a\circ\sigma} \;\wedge\; o' = \sigma(\mathit{caller})\\
&\quad\quad \wedge\; \mathit{cn} = \sigma(\mathit{cnCaller})) \;\vee\\
&\quad (s = \textbf{return}(e) \;\wedge\; \mathit{sensitive}(d,l) = [\![e]\!]_{a\circ\sigma} \;\wedge\; o = \sigma(\mathit{caller})\\
&\quad\quad \wedge\; \mathit{cn} = \sigma(\mathit{cnThis})) \;\vee\\
&\quad (s = x := e'.m(\overline{e}) \;\wedge\; \mathit{sensitive}(d,l) \in [\![\overline{e}]\!]_{a\circ\sigma} \;\wedge\; o' = [\![e']\!]_{a\circ\sigma}\\
&\quad\quad \wedge\; \mathit{cn} = \mathit{contract(o’)}) \;\vee\\
&\quad (s = x := e'.m(\overline{e}) \;\wedge\; \mathit{sensitive}(d,l) \in [\![\overline{e}]\!]_{a\circ\sigma} \;\wedge\; o = [\![e']\!]_{a\circ\sigma}\\
&\quad\quad \wedge\; \mathit{cn} = a(\mathit{cnThis}))
\end{aligned} \tag{1}$$

$$\begin{aligned}
&\mathit{complyTo}(l, \mathit{cn})(\mathit{cfg}) = \exists u, \mathit{cp}, \mathit{cm}, b, t.\\
&\quad l(u, \mathit{cp}, \mathit{cm}, b, t) \in \mathit{cfg} \;\wedge\; \mathit{cn} \in (\mathit{cp} \cup \mathit{cm})
\end{aligned} \tag{2}$$

$$\begin{aligned}
&\mathit{checked\text{-}scope}(o, \langle l, \mathit{cn}\rangle)(\mathit{cfg}) = \exists a, \sigma, s, V, \mathit{db}.\\
&\quad o(a, (\sigma, s@V), \mathit{db}) \in \mathit{cfg} \;\wedge\; \langle l, \mathit{cn}\rangle \in V
\end{aligned} \tag{3}$$

$$\begin{aligned}
&\mathit{errorU}(o, \mathit{cn})(\mathit{cfg}) = \mathit{errorU}(o, \mathit{cn}) \in \mathit{cfg}\\
&\mathit{errorC}(o, \mathit{cn})(\mathit{cfg}) = \mathit{errorC}(o, \mathit{cn}) \in \mathit{cfg}
\end{aligned} \tag{4}$$

$$\begin{aligned}
&\mathit{collect}(o, l, \mathit{cn})(\mathit{cfg}) = \exists a, \sigma, e_1, e_2, e_3, s, s', V, \mathit{db}.\\
&\quad o(a, (\sigma, (s; s')@V), \mathit{db}) \in \mathit{cfg} \;\wedge\\
&\quad s = \textbf{collect}(e_1, e_2, e_3) \;\wedge\; \mathit{cn} = [\![e_1]\!]_{a\circ\sigma} \;\wedge\; l = [\![e_2]\!]_{a\circ\sigma}
\end{aligned} \tag{5}$$

$$\begin{aligned}
&\mathit{optedOut}(l, \mathit{cn})(\mathit{cfg}, R, \mathit{Id}) = \exists u, b, \mathit{cp}, \mathit{cm}, t.\\
&\quad l(u, b, \mathit{cp}, \mathit{cm}, t) \in \mathit{cfg} \;\wedge\; R = \text{OPT-OUT}(l) \;\wedge\; \mathit{cn} \notin \mathit{cm}
\end{aligned} \tag{6}$$

$$\begin{aligned}
&\mathit{optedIn}(l, \mathit{cn})(\mathit{cfg}, R, \mathit{Id}) = \exists u, b, \mathit{cp}, \mathit{cm}, t.\\
&\quad l(u, b, \mathit{cp}, \mathit{cm}, t) \in \mathit{cfg} \;\wedge\; R = \text{OPT-IN}(l) \;\wedge\; \mathit{cn} \in \mathit{cm}
\end{aligned} \tag{7}$$

$$\mathit{expired}(l)(\mathit{cfg}) = \exists u, b, \mathit{cp}, \mathit{cm}, t.\; l(u, b, \mathit{cp}, \mathit{cm}, t) \notin \mathit{cfg} \tag{8}$$

$$\begin{aligned}
&\mathit{dbDel}(l)(\mathit{cfg}) = \exists o, a, p, \mathit{db}, d.\\
&\quad o(a, p, \mathit{db}) \in \mathit{cfg} \;\wedge\; \mathit{sensitive}(d, l) \notin \mathit{db}
\end{aligned} \tag{9}$$

$$\mathit{deleted}(l)(\mathit{cfg}, R, \mathit{Id}) = R = \text{DELETE}(l) \tag{10}$$

$$\mathit{noExecIn}(o)(\mathit{cfg}, R, \mathit{Id}) = \mathit{Id} \neq o \tag{11}$$

In the following, we prove the properties in Section IV. The initial state of the program $\overline{CL}\ \overline{PI}$ **main**$\{\overline{T\ x}; s\}$ is a multiset including the object *ob(main)*, the classes $\overline{CL}$, and the purposes $\overline{PI}$, where $\mathit{cfg}_0 = \{\mathit{CL}\ \mathit{PI}\ \mathit{ob(main)}([\mathit{cnThis} \mapsto \mathit{contract}(\mathit{main}, \mathit{ob(main)})], ([], s@\emptyset), [])\}$.

$$\begin{aligned}
&\mathscr{P} = \mathscr{P}_1 \;\wedge\; \mathscr{P}_2\\
&\mathscr{P}_1 = \forall o, d, l, \mathit{cn}.\\
&\quad \square(\mathit{use}(o, \mathit{sensitive}(d,l), \mathit{cn}) \Rightarrow \mathit{checked\text{-}scope}(o, \langle l, \mathit{cn}\rangle))\\
&\quad \Rightarrow \square\neg \mathit{errorU}(o, \mathit{cn})\\
&\mathscr{P}_2 = \forall o, d, l, \mathit{cn}.\\
&\quad \square((\mathit{use}(o, \mathit{sensitive}(d,l), \mathit{cn}) \;\wedge\; \neg \mathit{complyTo}(l, \mathit{cn})\\
&\quad \wedge\; \neg \mathit{checked\text{-}scope}(o, \langle l, \mathit{cn}\rangle))\\
&\quad \Rightarrow \bigcirc(\mathit{noExecIn}(o)\ \mathit{Until}\ (\mathit{errorU}(o, \mathit{cn}) \;\vee\; \mathit{optedIn}(l, \mathit{cn}))))
\end{aligned}$$

Theorem A.1 (Purpose limitation)**.** *The property $\mathscr{P}$ holds for all traces of our transition system.*

Proof. First, we prove $\mathscr{P}_1$ using a proof by contradiction. Let τ be a fair trace. The premise of $\mathscr{P}_1$ says that in every state in τ, for all o, d, l, and cn, whenever $\mathit{use}(o, \mathit{sensitive}(d,l), \mathit{cn})$ holds, then $\mathit{checked\text{-}scope}(o, \langle l, \mathit{cn}\rangle)$ holds. To achieve a contradiction, assume that $\mathit{errorU}(o, \mathit{cn})$ holds in a reachable state in τ and consider the first such state cfg' that it holds. This cannot be the initial state in τ (per definition) so there must be a transition from a predecessor state *cfg* to cfg' adding $\mathit{errorU}(o, \mathit{cn})$ to the configuration. The only rules that could have added $\mathit{errorU}(o, \mathit{cn})$ are the following:

- ERROR-ASSIGN: In $o(a, (\sigma, (x := e; s)@V), \mathit{db})$, let $\mathit{sensitive}(d,l) = [\![e]\!]_{a\circ\sigma}$ and $\mathit{cn} = a(\mathit{cnThis})$. Since the current program statement in *cfg* is an assignment, then $\mathit{use}(o, \mathit{sensitive}(d,l), \mathit{cn})$ holds in this state. Therefore, $\mathit{checked\text{-}scope}(o, \langle l, \mathit{cn}\rangle)$ holds in *cfg*, and hence also $\langle l, \mathit{cn}\rangle \in V$. Thus, in the rule ERROR-ASSIGN, the premise *comply* holds, and this rule cannot fire in *cfg*. Thus, $\mathit{errorU}(o, \mathit{cn})$ does not hold in cfg'.
- ERROR-CALL: In $o(a, (\sigma, (x := e.m(\overline{e}); s)@V), \mathit{db})$, let $o' = [\![e]\!]_{a\circ\sigma}$, $\mathit{sensitive}(d,l) \in [\![\overline{e}]\!]_{a\circ\sigma}$, and $\mathit{cn} = \mathit{contract}(o')$. Since the current program statement in *cfg* is a call, then $\mathit{use}(o, \mathit{sensitive}(d,l), \mathit{cn})$ holds in this state. The rest of this case is identical to the ERROR-ASSIGN case.
- ERROR-SELF-CALL: In $o(a, (\sigma, (x := e.m(\overline{e}); s)@V), \mathit{db})\}$, let $o = [\![e]\!]_{a\circ\sigma}$, $\mathit{sensitive}(d,l) \in [\![\overline{e}]\!]_{a\circ\sigma}$, and $\mathit{cn} = a(\mathit{cnThis})$. Since the current program statement in *cfg* is a self-call, then $\mathit{use}(o, \mathit{sensitive}(d,l), \mathit{cn})$ holds in this state. The rest of this case is identical to the ERROR-ASSIGN case.
- ERROR-RETURN: In $o(a, (\sigma, \textbf{return}\ e@V), \mathit{db})$, let $\mathit{sensitive}(d,l) = [\![e]\!]_{a\circ\sigma}$, $o' = \sigma(\mathit{caller})$, and

$cn = \sigma(\mathit{cnCaller})$. Since the current program statement in *cfg* is a **return**, then $\mathit{use}(o, \mathit{sensitive}(d, l), cn)$ holds. The rest of this case is identical to the ERROR-ASSIGN case.

- ERROR-SELF-RETURN: In $o(a, (\sigma, (\textbf{return}\ \ e; \mathit{cont(n)})@V), db)$ $n(\sigma', (x := m?; s)@\emptyset)$, let $\mathit{sensitive}(d, l) = [\![e]\!]_{a \circ \sigma}$, $o = \sigma(\mathit{caller})$, and $cn = a(\mathit{cnThis})$. Since the current program statement in *cfg* is a self **return**, then $\mathit{use}(o, \mathit{sensitive}(d, l), cn)$ holds. The rest of this case is identical to the ERROR-ASSIGN case.

In all these cases, we conclude that $\neg \mathit{errorU}(o, cn)$ holds, so $\mathscr{P}_1$ holds.

Next, we prove $\mathscr{P}_2$. Given a fair trace τ, consider any reachable state *cfg* where the premise of $\mathscr{P}_2$ holds, i.e., for some o, d, l, and cn, $\mathit{use}(o, \mathit{sensitive}(d, l), cn)$, $\neg \mathit{checked\text{-}scope}(o, \langle l, cn \rangle)$, and $\neg \mathit{complyTo}(l, cn)$ hold. Let cfg' be the next state in τ. We show that there exists a state cfg'' after cfg' where $\mathit{errorU}(o, cn) \vee \mathit{optedIn}(l, cn)$ holds, and $\mathit{noExecIn}(o)$ holds for all states from cfg' up to (but not necessarily including) cfg''.

For the rest of the proof, we use the fact that $\mathit{noExecIn}(o)$ holds when enabled rules not involving the object o fire. There are four such kinds of rules: 1) TICK, 2) DELETE, 3) OPT-OUT, and 4) all enabled rules that apply to other objects than o.

Now, since $\mathit{use}(o, \mathit{sensitive}(d, l), cn)$ holds in *cfg*, there are five possible program statements that can execute within o. These correspond to the five disjunctions in the definition of *use* (Equation 1). We consider the Assignment case below and four other cases for Call, Self-Call, Return, and Self-Return are analogous.

For the assignment case, consider how execution can progress after an assignment, from the next state cfg'. Either a program statement in the object o fires, or one of the four kinds of rules fire not involving o. This yields the following two cases: 1) The only enabled rule involving o is ERROR-ASSIGN, which produces a state cfg'' where $\mathit{errorU}(o, cn)$ holds. 2) For the other four kinds of rules that can be enabled, as discussed previously, (i) for all of these rules, ERROR-ASSIGN remains enabled except (ii) when the rule OPT-IN, enabled in another object o', fires and thereby adds the contract cn to the policy l. If 2(i) holds, then $\mathit{noExecIn}(o)$ holds in the successor state. We can only repeat this case finitely often since ERROR-ASSIGN remains enabled, and by fairness, it must eventually fire, leading to state cfg'' where $\mathit{errorU}(o, cn)$ holds. If 2(ii) applies (to any successor state), we then reach a state cfg'' where $\mathit{optedIn}(l, cn)$ holds. We conclude that from cfg' onwards, $\mathit{noExecIn}(o)$ holds until we reach cfg'' when either $\mathit{errorU}(o, cn)$ or $\mathit{optedIn}(l, cn)$ holds. This establishes $\bigcirc(\mathit{noExecIn}(o)\ \mathit{Until}\ (\mathit{errorU}(o, cn) \vee \mathit{optedIn}(l, cn)))$. □

$$
\begin{aligned}
&\mathscr{C} = \mathscr{C}_1 \ \wedge \ \mathscr{C}_2\\
&\mathscr{C}_1 = \forall o, l, cn.\\
&\quad \Box(\mathit{collect}(o, l, cn) \Rightarrow \mathit{checked\text{-}scope}(o, \langle l, cn \rangle))\\
&\quad\ \Rightarrow \Box \neg \mathit{errorC}(o, cn)\\
&\mathscr{C}_2 = \forall o, l, cn.\\
&\quad \Box((\mathit{collect}(o, l, cn) \ \wedge\\
&\quad \neg \mathit{complyTo}(l, cn) \ \wedge \ \neg \mathit{checked\text{-}scope}(o, \langle l, cn \rangle))\\
&\quad\ \Rightarrow \bigcirc(\mathit{noExecIn}(o)\ \mathit{Until}\ (\mathit{errorC}(o, cn) \ \vee \ \mathit{optedIn}(l, cn))))
\end{aligned}
$$

Theorem A.2 (Consent)**.** *The property $\mathscr{C}$ holds for all traces of our transition system.*

Proof. The proof of $\mathscr{C}_1$ is analogous to our previous proof of $\mathscr{P}_1$. Namely, we prove $\mathscr{C}_1$ using a proof by contradiction. Let τ be a fair trace. The premise of $\mathscr{C}_1$ says that in every state in τ, for all o, l, and cn, whenever *collect(o,l,cn)* holds, then *checked-scope* holds. To achieve a contradiction, assume that $\mathit{errorC}(o, cn)$ holds in a reachable state in τ and consider the first such state cfg' that it holds. This cannot be the initial state in τ (per definition) so there must be a transition from the predecessor state *cfg* to cfg' adding $\mathit{errorC}(o, cn)$ to the configuration. The only rule that could have added $\mathit{errorC}(o, cn)$ is ERROR-COLLECT. In $o(a, (\sigma, (\textbf{collect}(e_1, e_2, e_3); s)@V), db)$, let $cn = [\![e_1]\!]_{a \circ \sigma}$ and $l = [\![e_2]\!]_{a \circ \sigma}$. Since the current program statement in *cfg* is $\textbf{collect}(e_1, e_2, e_3)$, then $\mathit{collect}(o, l, cn)$ holds. Therefore, $\mathit{checked\text{-}scope}(o, \langle l, cn \rangle)$ holds in *cfg*, and hence also $\langle l, cn \rangle \in V$. Thus, in the rule ERROR-COLLECT, the premise *comply* holds and this rule cannot fire in *cfg*. Thus, $\mathit{errorC}(o, cn)$ does not hold in cfg', so $\mathscr{C}_1$ holds.

The proof of $\mathscr{C}_2$ is analogous to our proof of $\mathscr{P}_2$. In particular, given a fair trace τ, consider any reachable state *cfg*, where the premise of $\mathscr{C}_2$ holds, i.e., for some o, l, and cn, the formulas *collect(o,l,cn)*, $\neg \mathit{complyTo}(l, cn)$, and $\neg$*checked-scope* hold. Let cfg' be the next state in τ. We show that there exists a state cfg'' after cfg' where $\mathit{errorC}(o, cn) \ \vee \ \mathit{optedIn}(l, cn)$ holds, and $\mathit{noExecIn}(o)$ holds for all states from cfg' up to (but not necessarily including) cfg''.

As with the previous proof of $\mathscr{P}_2$, we will use the fact that $\mathit{noExecIn}(o)$ holds when enabled rules not involving the object o fire. There are four such kinds of rules: 1) TICK, 2) DELETE, 3) OPT-OUT, and 4) all enabled rules that apply to other objects than o.

Since *collect(o,l,cn)* holds in *cfg*, there is only one possible program statement that can execute according to the definition of *collect* in Equation 5, which is $\textbf{collect}(cn, l, x)$. Now consider how execution can progress from the next state cfg'. Either a program statement in the object o fires, or one of the four kinds of rules fire not involving o. This yields the following two cases: 1) The only enabled rule involving o is ERROR-COLLECT, which produces a state cfg'' where $\mathit{errorC}(o, cn)$ holds. 2) For the other four kinds of rules that can be enabled, as discussed previously, (i) for all of these rules, ERROR-COLLECT remains enabled except (ii) when the rule OPT-IN, enabled in another object o', fires and thereby adds the contract *cn* to the policy *l*. If 2(i) holds, then $\mathit{noExecIn}(o)$ holds in the successor state. We can only repeat this case finitely often since ERROR-COLLECT remains enabled, and by fairness, it must eventually fire, leading to state cfg'' where $\mathit{errorC}(o, cn)$ holds. If 2(ii) applies (to any successor state), we then reach a state cfg'' where $\mathit{optedIn}(l, cn)$ holds. We conclude that from cfg' onwards, $\mathit{noExecIn}(o)$ holds until we reach cfg'' when either $\mathit{errorC}(o, cn)$ or $\mathit{optedIn}(l, cn)$ holds. This establishes $\bigcirc(\mathit{noExecIn}(o)\ \mathit{Until}\ (\mathit{errorC}(o, cn)\ \vee\ \mathit{optedIn}(l, cn)))$. □

$$\begin{aligned}&\mathscr{W} = \forall l, cn.\\ &\quad \Box(\mathit{optedOut}(l, cn) \Rightarrow (\neg \mathit{complyTo}(l, cn)\ W\ \mathit{optedIn}(l, cn)))\end{aligned}$$

Theorem A.3 (Right to withdraw consent)**.** *The property $\mathscr{W}$ holds for all traces of our transition system.*

Proof. Given a fair trace τ, consider any reachable state *cfg* where $\mathit{optedOut}(l, cn)$ holds, i.e., according to the definition *optedOut* (in Equation 6), the rule OPT-OUT has been fired and the contract *cn* does not belong to the policy *l*. We show that if there exists a state cfg' in τ, where $\mathit{optedIn}(l, cn)$ holds, then $\neg \mathit{complyTo}(l, cn)$ holds from *cfg* up to cfg'. Moreover, if $\mathit{optedIn}(l, cn)$ never holds in any state in τ, then $\neg \mathit{complyTo}(l, cn)$ holds forever from *cfg* onwards.

Since $\mathit{optedOut}(l, cn)$ holds in *cfg*, the contract *cn* does not belong to the policy *l*, thus the formula $\neg$*complyTo(l,cn)* also holds in *cfg*. Note that $\neg \mathit{complyTo}(l, cn)$ is invariant over all the rules except OPT-IN. So either 1) the rule OPT-IN eventually fires yielding a state cfg' where $\mathit{optedIn}(l, cn)$ holds, and $\neg \mathit{complyTo}(l, cn)$ holds from *cfg* up to this point, or 2) OPT-IN never fires and then $\neg \mathit{complyTo}(l, cn)$ continuously holds from *cfg*. □

For deletion, we formalize the auxiliary functions *dec* and *del* (in Section III-D) in the following. The function $\mathit{delData}(db, l)$ deletes sensitive data associated with the policy *l* from the database *db*, which is a substitution. Equations are applied in order, top-down.

$$\begin{aligned}&\mathit{dec(l(u,cp,cm,b,t)\ cfg)} = l(u, cp, cm, b, t-1)\ \mathit{dec(cfg)}\quad \mathit{if}\ t > 1\\ &\mathit{dec(l(u,cp,cm,b,1)\ cfg)} = \mathit{dec(del(l,\ cfg))}\\ &\mathit{dec(cfg)} = \mathit{cfg}\end{aligned} \tag{12}$$

$$\begin{aligned}&\mathit{del(l,o(a,p,db)\ cfg)} = o(a, p, \mathit{delData}(db, l))\ \mathit{del(l,cfg)}\\ &\mathit{del(l,\ cfg)} = \mathit{cfg}\\ &\mathit{delData}([\mathit{sensitive}(d, l'), \overline{v}], l) = \mathit{delData}([\overline{v}], l)\quad \mathit{if}\ l = l'\\ &\mathit{delData}([\mathit{sensitive}(d, l'), \overline{v}], l) =\\ &\qquad\qquad [\mathit{sensitive}(d, l'), \mathit{delData}([\overline{v}], l)]\quad \mathit{if}\ l \neq l'\\ &\mathit{delData}([], l) = []\end{aligned} \tag{13}$$

$$\mathscr{S} = \forall l.\ \Box(\mathit{expired}(l) \Rightarrow \mathit{dbDel}(l))$$

Theorem A.4 (Storage limitation)**.** *The property $\mathscr{S}$ holds for all traces of our transition system.*

Proof. Given a fair trace τ, consider any reachable state *cfg* where $\mathit{expired}(l)$ holds by Equation 8, a policy *l* does not exist in the state *cfg*. We show that $\mathit{dbDel}(l)$ also holds in *cfg*, where data with the policy *l* is deleted from all objects' databases. Note that the state *cfg* cannot be the initial state because in the initial state there is no policy. In the predecessor state of *cfg*, the only rules that could have deleted a policy *l*, yielding $\mathit{expired}(l)$ in *cfg*, are the following:

- Tick: The TICK rule was applied and the timestamp of the policy was one. In this case, the *dec* function (in Equation 12), deletes the policy, where $\mathit{expired}(l)$ holds in *cfg*, and the function *del* (in Equation 13) deletes sensitive data associated with the policy *l* from databases, so that $\mathit{dbDel}(l)$ holds in *cfg*. Hence, $\mathscr{S}$ holds.
- Delete: The rule DELETE was applied. In this case, the policy is deleted, where $\mathit{expired}(l)$ holds in *cfg*, and the function *del* deletes sensitive data associated with the policy *l* in databases, so that $\mathit{dbDel}(l)$ holds in *cfg*. Hence, $\mathscr{S}$ holds.

□

$$\mathscr{F} = \forall l.\ \Box(\mathit{deleted}(l) \Rightarrow \mathit{expired}(l))$$

Theorem A.5 (Right to be forgotten)**.** *The property $\mathscr{F}$ holds for all traces of our transition system.*

Proof. In a fair trace τ, consider any reachable state *cfg* where $\mathit{deleted}(l)$ holds by Equation 10, the rule DELETE was applied to a policy *l*. We show that $\mathit{expired}(l)$ also holds in *cfg*, and by Theorem A.4 data with that policy is deleted from databases. Note that the state *cfg* cannot be the initial state because in the initial state there is no policy. Since $\mathit{deleted}(l)$ holds in *cfg*, then in the predecessor state of *cfg*, the rule DELETE was applied, which deletes the policy, yielding $\mathit{expired}(l)$ in *cfg*, so $\mathscr{F}$ holds. □

B. *Rewrite rules for non-sensitive data*

Figure 8 shows the rewrite rules for non-sensitive data, omitted from Sect. III-D. These rules mirror their non-starred counterparts for sensitive data, but without compliance checks. Rule STORE* stores non-sensitive data d in the database, ASSIGN-LOCAL* assigns non-sensitive data d to local variables, ASSIGN-FIELD* assigns non-sensitive data d to fields, and RETURN* returns non-sensitive data d to the caller.

C. *Maude formalization*

a) Maude model: We specify DPL's operational semantics in Maude, which gives us a prototype environment for program simulation and verification.

In our Maude model, a program is written in a main block with a multiset of classes: $\mathit{main}\{L,\ SL\}\ \mathit{cfg}$, where *L* is an initial state (substitution), *SL* is a list of statements, and *cfg* specifies the classes. An object is represented as

$$\langle\ O : C \mid \mathit{Att{:}\ S,\ Pr{:}\ (L,\ SL),\ Lcnt{:}\ N,\ Db{:}\ DB}\ \rangle$$

consisting of the object name *O*, the class name *C*, attributes *S*, the active process *(L, SL)* with local variables *L* and statements *SL*, a counter for creating unique identities *N*, and the database *DB*. The counter is a technical device, omitted from the previous sections, used to create fresh identifiers, corresponding to the use of the *fresh* predicate in the operational semantics.

$$\overset{\text{STORE*}}{\dfrac{d = [\![e]\!]_{a\circ\sigma}}{\begin{array}{c}o(a,(\sigma,(\textbf{store}(k,e)\ \textbf{else}\{s\};s')@V),db)\\ \to o(a,(\sigma,s'@V),db[k\mapsto d])\end{array}}}
\quad
\overset{\text{ASSIGN-LOCAL*}}{\dfrac{x \in dom(a) \quad d = [\![e]\!]_{a\circ\sigma}}{\begin{array}{c}o(a,(\sigma,(x := e;s)@V),db)\\ \to o(a,(\sigma[x\mapsto d],s@V),db)\end{array}}}
\quad
\overset{\text{ASSIGN-FIELD*}}{\dfrac{x \in dom(a) \quad d = [\![e]\!]_{a\circ\sigma}}{\begin{array}{c}o(a,(\sigma,(x := e;s)@V),db) \to\\ o(a[x\mapsto d],(\sigma,s@V),db)\end{array}}}$$

$$\overset{\text{RETURN*}}{\dfrac{d = [\![e]\!]_{a\circ\sigma} \quad o' = \sigma(caller)}{\begin{array}{c}o(a,(\sigma,\textbf{return}\ e\ @V),db) \to\\ o(a,idle,db)\ com(d,o')\end{array}}}
\quad
\overset{\text{SELF-RETURN*}}{\dfrac{o = \sigma(caller) \quad d = [\![e]\!]_{a\circ\sigma}}{\begin{array}{c}o(a,(\sigma,(\textbf{return}\ e;cont(n))@V),db)\\ n(\sigma',(x := m?;s)@V') \to\\ o(a,(\sigma'[x\mapsto d],(s)@V'),db)\end{array}}}$$

Figure 8. Rewrite rules for operations on non-sensitive data.

For simplicity, class names represent purposes, and an object created from a class *C* gets the purpose *C*.

A policy is represented as

$$PL\ \langle\ U,\ Cp,\ Cm,\ B,\ T\ \rangle$$

where *PL* is the policy identity, *U* is a user identity, *Cp* is a set of persistent contracts, *Cm* is a set of mutable contracts, *B* is true if data is allowed to be stored persistently, and *T* is the timestamp.

For **opt-in** options, we assign a user object to consent to or deny an **opt-in** option. The user object sends the message *optInMsg(true, 'contract, 'policy)* to consent and *optInMsg(false, 'contract, 'policy)* to deny consent. In Maude's syntax, a name is represented by *'name*. Input data for data collection is given in the *collect* command; i.e., *collect(int(1), 'contract, 'policy, 'x)*, where *int(1)* is the input data, and the function *int(1)* creates data of type Integer. Moreover, a userId is given in the command *logIn(str('u))*, where the function *str()* creates data of type String.

The rules TICK, DELETE, and OPT-OUT can be interleaved at any point in the program reflecting clock ticks and user actions. To specify fairness and some of the formulas in Table II, we add the element $ruleLabel(Id, R)$ to the configuration, where *Id* is the identity of the current object that is executing, and R is the current rule's label that is fired. When a rule fires, the parameters of *ruleLabel* change accordingly.

b) A rewrite rule example: Here, we present the formalization of the DELETE rule in Maude. Note that in this paper, we omit the *ruleLabel(Id, R)* from the rules, where *Id* is the identity of the object that is executing, and *R* is an event constructed from a rule label and possibly parameters. In our Maude model, *ruleLabel(Id, R)* is added to all the rules. In the following, when the rule *delete* fires, *Id* changes to *Other*, which is a constant of type *Identity*, and *R* to *delete(PL)* of type *Event*. Note that if an object *O* is executing, then *Id* changes to *O*, which is also of type *Identity*.

```
rl [delete] :
{ PL ⟨ U, Cp, Cm, B, T ⟩ ruleLabel(Id, R) Cfg }
 =>
{ del(PL, ruleLabel(Other, delete(PL)) Cfg) } .
```

c) Model checking results for programs: We define the LTL formulas in Section IV in Maude and use Maude's model checker to verify the GDPR properties that we previously formalized on the online retailer example in Fig. 3.

Hygienic programs: We verify the properties $\mathscr{P}_1$, $\mathscr{C}_1$, $\mathscr{W}$, $\mathscr{S}$, and $\mathscr{F}$ on the (hygienic) online retailer program. Maude verifies that errors cannot occur and returns true for each of these properties.

Non-hygienic programs: To check the properties concerning programs giving rise to errors, we model check different scenarios for non-hygienic programs for Fig.3. Namely, we remove different combinations of the statements to systematically check the necessity of the all conditions for hygienic programs given in Section IV.

The scenarios are as follows: i) For the Purchase purpose, we do not define the consent statements cs1 and cs2 and the contract cn1. Therefore, in the register method, credit card data and customer data are collected with the MassMarketing's consent statement and contract (cs3, cn2). Maude throws a data usage error when the method p. purchase(credit, customer) is called. To avoid this error, the **if-comply**(cn1, (credit, customer)) is required (line 45). ii) The pattern for data collection, in Section III-B, is not followed. In this case, we check two scenarios: 1) We skip the log-in() (line 11). Maude throws an error when creating a policy (line 13). 2) We remove the **if-consent** in line 16, to check that the appropriate error arises when collecting data for the Purchase purpose. Maude verifies that a data collection error arises. We check the consent property $\mathscr{C}_2$ for the credit data with the policy l1 with respect to the Purchase contract. Maude verifies the property $\mathscr{C}_2$. iii) We remove the **if-comply** for the call mm.m-marketing (line 47), and check $\mathscr{P}_2$. Maude verifies that the appropriate error arises when making the call m-marketing. Note that we check the purpose limitation property $\mathscr{P}_2$ on the customer data with respect to the 'MassMarketing contract.

Here, we present Maude's model checker results for a non-hygienic variant of the program in Fig.3, where the **if-consent** in line 16 and the **if-comply** for the call mm.m-marketing (line 47) are removed. In the following commands, *init* is the initial configuration built from the program's main block. The term *sensitive(str('mail), policy(2))* is sensitive data associated with the customer data and the policy *policy(2)*. The term *contract(str('MassMarketing), ob('MassMarketing0))* is the contract for the object *MassMarketing0*. The term *contract(str('Purchase), ob('Purchase0))* is the contract for the object *Purchase0*. The term *policy(1)* is the policy for the credit data. In the following, Maude verifies the properties $\mathscr{P}$, $\mathscr{C}$, $\mathscr{W}$, $\mathscr{S}$, and $\mathscr{F}$, respectively:

```
red modelCheck(init, purposeLimitationconf
    (init, sensitive(str('mail), policy(2)),
        contract(str('MassMarketing), ob('MassMarketing0)))) .
result Bool: true
```

```
red modelCheck(init, consentconf(init,
    policy(1), contract(str('Purchase), ob('Purchase0)))) .
result Bool: true
```

```
red modelCheck(init, withdraw(init,
    contract(str('Purchase), ob('Purchase0)), policy(1))) .
result Bool: true
```

```
red modelCheck(init, storageLimitationconf(init, policy(2))) .
result Bool: true
```

```
red modelCheck(init, forget(init, policy(2))) .
result Bool: true
```

D. Case study extension

We expand on the example from Section III-C and show that DPL prevents illegal data usage and storage in objects receiving sensitive data. These objects can only store or process sensitive data if their contracts comply with the policy, i.e., if consent for the object's purpose is given. When a user withdraws consent or requests data deletion, the corresponding policy changes. Thus, prior to any data usage, conditional constructs are needed to avoid errors. Moreover, objects cannot illegally send sensitive data to other objects since errors occur.

We present the classes Purchase-c and MMarketing-c. The corresponding objects p and mm receive the credit and customer data. In class Purchase-c, the method purchase stores the customer data if storage is allowed and the user's consent for the Purchase purpose is given. In line 7, the object's contract is checked for compliance, then data processing continues; otherwise, the default value "0" is returned. In line 11, the caller's contract is checked for compliance before returning the method result. In the class MMarketing-c, we create a new object tm for targeted marketing and try to illegally send customer data to this object via a method call (line 28). The call triggers an error since tm's contract is not defined and does not comply with the customer's policy. Moreover, if we define the corresponding contract/consent statement, we still need a session and the user's consent to add the contract to the policy. In line 34, the m-marketing method checks if the object's contract complies with the policy, then continues processing the data. A message with the given text is sent to the customer and a copy of the message is returned to the caller, which is our main object.

Similarly, we run Maude's model checker for the program in Fig. 9, and Maude verifies all the properties in Section IV.

```
 1 class Purchase-c implements Purchase() {
 2   Order purchase(String credit, String customer){
 3     user = ... // Make user accounts for data storage
 4     // Store data if storage is allowed, otherwise skip:
 5     store(key(user, "customer"), customer) else { skip; }
 6     // Check if data usage is allowed:
 7     if-comply(cnThis, credit, customer){
 8       ...
 9       Order order=...
10       // Check if the caller is allowed to receive the result
11       if-comply(cnCaller, order) { return(order); }
12       else{ return 0; }
13     } // End of if-comply(cnThis, credit, customer)
14     else {return 0; }
15   }// End of the purchase method
16 } // End of class
17
18
19 purpose TargetedMarketing {
20   String targetedMarketing(String customer){...}}
21
22 class TargetedMarketing-c implements TargetedMarketing() {...}
23
24 class MMarketing-c implements MassMarketing() {
25   TargetedMarketing-c tm = new TargetedMarketing-c();
26   String m-marketing(String customer){
27     // Let us send data illegally to tm
28     tm.targetedMarketing(customer); // This results in an error
29
30     user = ... // Create user accounts for data storage
31     // Store data if storage is allowed, otherwise skip
32     store(key(user, "customer"), customer) else { skip; }
33
34     if-comply(cnThis, customer){ // Self-sanity check
35       String text=...
36       ... // Send MassMarketing text to the customer
37       return text;
38     } // End of if-comply
39     else {return 0; }
40   } // End of m-marketing method
41 } // End of class
```

Figure 9. Extension of the online-retailing example in DPL from Fig. 3.