DataProVe: A **Data Pr**otection P**o**licy and System Architecture **Ve**rification Tool

Abstract

In this paper, we propose a tool, called DataProVe, for specifying high-level data protection policies and system architectures, as well as verifying the conformance between them in a fully automated way. The syntax of the policies and the architectures is based on semi-formal languages, and the automated verification engine relies on logic and resolution based proofs. The functionality and operation of the tool are presented using different examples.

1 Introduction

Under the General Data Protection Regulation (GDPR) [1], personal data is defined as "any information relating to an identified or identifiable natural person"[1]. The GDPR specifies the rights for living individuals who have their personal data processed, and enforce responsibilities for the data controllers and the data processors who store, process or transmit such data.

Despite the data protection laws, there were several data breaches incidents in the past (e.g. [3–5]) and nowadays, such as the Cambridge Analytica scandal of Facebook [6], where personal data of more than 87 millions Facebook users has been collected and used for advertising and election campaign purposes without a clear data usage consent. One of the main problems was the insufficient check by Facebook on the third party applications. Google also faced lawsuit over collecting personal data without permission, and has been reported to illegally gather the personal data of millions of iPhone users in the UK [7].

The GDPR took effect in May 2018, and hence, designing compliant data protection policies and system architectures became even more important for organizations to avoid penalties. Data protection by design, under the Article 25 of the GDPR [8], requires the design of data protection measures into the development of business processes of service providers. The regulation also limits businesses from performing user profiling and demanding appropriate consents before personal data collection (Article 6 of the GDPR [9]).

Unfortunately, in textual format, the data protection laws are sometimes ambiguous and can be misinterpreted by the policy and system designers. From the technical perspective, to the best of our knowledge, only a small number of studies can be found in the literature that investigate the formal or automated method to design and verify policies and architectures in the context of data protection and privacy. The main advantage of using formal approaches during system design is that data protection properties can be mathematically proved, and design flaws can be detected at an early stage, which can save time and money.

On the other hand, using formal method for this purpose is also challenging, as abstraction is required, which is difficult in case of complex laws. In this paper, we address this problem, and model some simple data protection requirements of GDPR with regards to the data collection, usage, storage, deletion, and transfer phases. Privacy requirements are also considered such as the right to have certain data and link certain data types. We focus on the policy and architecture levels, and propose a variant of policy and architecture language, specifically designed for specifying

[1] In the US, personally identifiable information is used with a similar interpretation [2].

and verifying data protection and privacy requirements. In addition, we propose a fully automated algorithm, for verifying three types of conformance relations between a policy and an architecture specified in our language. Our theoretical methods are implemented in the form of a software tool, called DataProVe, for demonstration purposes.

The main goals of our policy and architecture languages and software tool include helping a system designer at the higher level specification (compared to the other tools that mainly focus on the protocol level), such as with the policy and architecture design, to spot any potential errors prior the concrete lower level system specification. Besides, our tool can be used for education or research purposes as well. To the best of our knowledge, this is the first work that addresses the problem of fully automated conformance check between a policy and an architecture in the context of data protection and privacy requirements.

This paper includes the following contributions:

1. We propose a variant of privacy policy language (in Section 3).
2. We propose a variant of privacy architecture language (in Section 4).
3. We propose the definition of three conformance relations between a policy and architecture (in Section 5), namely, the privacy, data protection, and functional conformance relations.
4. We propose a logic based fully automated conformance verification procedure (in Section 6) for the above three conformance relations.
5. Finally, we propose a (prototype) tool, called DataProVe, based on the theoretical foundations (in Section 8).

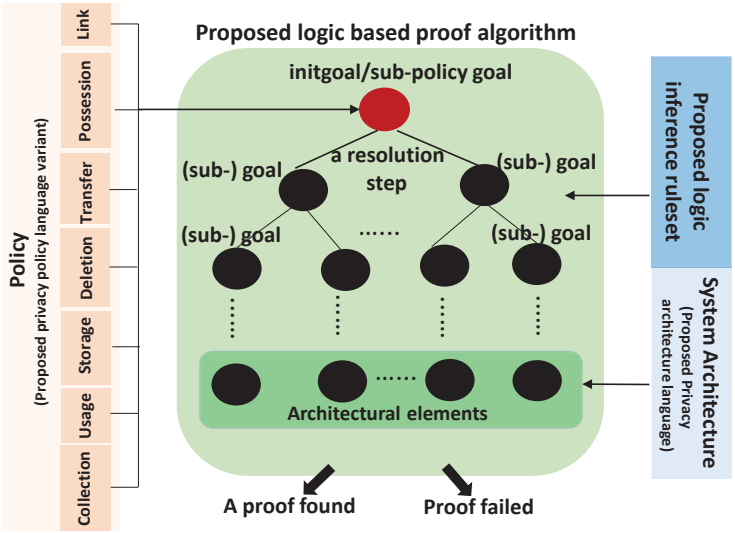

Figure 1: An overview and the intuition behind the contributions of this paper.

In Figure 1, a policy is specified using our proposed language variant, which covers seven sub-policies (data collection, usage, storage, deletion, transfer, possession and link). Each sub-policy is mapped to a logic goal that reflects the requirement in the policy. The verification engine attempts to prove each goal based on a set of logic inference rules and architectural elements (specified in our language). The proposed verification algorithm is based on a series of resolution steps, represented as a derivation tree, where the root is a goal to be proved, and the leaves are the architectural elements used to prove the goal.

The paper is structured as follows: In Section 2, we discuss the related policy and architecture languages. In Sections 3-4, we present our policy and architecture languages, respectively. The

automated conformance verification engine is detailed in Section 6. In Section 8 we present the DataProVe tool and its operation using two simple examples. Finally, we discuss the results and conclude the paper in Sections 7 and 9.

2 Related Works

2.1 Policy Languages

The Platform for Privacy Preferences (P3P) [10] enables web users to gain control over their private information on online services. On a website users can express their privacy practices in a standard format that can be retrieved automatically and interpreted by web client applications. Users are notified about certain website's privacy policies and have a chance to make decision on that. To match the privacy preferences of the users and web services, the authors proposed the Preference Exchange Language (APPEL) [11] integrated into the web clients, with which the user can express their privacy preferences that can be matched against the practices set by the online services. According to the study [12], in APPEL, users can only specify what is unacceptable in a policy. Identifying this, the authors in [12] proposed a more expressive preference language called XPref giving more freedom for the users, such as allowing acceptable preferences.

The Customer Profile Exchange (CPExchange) language [13], is a XML-based policy language, which was designed to facilitate business-to-business communication privacy policies (i.e. the privacy-enabled global exchange of customer profile information). The eXtensible Access Control Markup Language (XACML) [14] is a de-facto, XML-based policy language, specifically designed for access control management in distributed systems. The latest version was approved by the OASIS standards organization as an international standard in 2017. The Enterprise Privacy Authorisation Language (EPAL) of IBM [15] was designed to regulate an organisation's internal privacy policies. EPAL is partly similar to XACML, however, it mainly focuses on privacy policies instead of access control policies in XACML.

A-PPL [16] is an accountability policy language specifically designed for modelling data accountability (such as data retention, logging and notification) in the cloud. A-PPL is an extension of the the PrimeLife Privacy Policy Language (PPL) [17], which enables specification of access and usage control rules for the data subjects and the data controller. PPL is built upon XACML, and allows users to define the so-called sticky policies on personal data based on obligations. Obligation defines whether the policy language can trigger tasks that must be performed by a server and a client, once some event occurs and the related condition is fulfilled. This is also referred to as the Event-Action-Condition paradigm. The Policy Description Language (PDL) [18], proposed by Bell Labs, is one of the first policy-based management languages, specifically for network administration. It is declarative and is based on the Event-Action-Condition paradigm like PPL.

RBAC (Role-Based Access Control) [19] is one of the most well-known role-based access control policy languages. It uses roles and permissions in the enforced policies, namely, a subject can be assigned roles, and roles can be assigned certain access control permissions. ASL (Authorization Specification Language) [20] is another Role-based access control language based on first order logic, and RBAC components. Ponder [21] is a declarative and object-oriented policy language, and designed for defining and modelling security policies using RBAC, and security management policies for distributed systems. The policies are defined on roles or group of roles. Rei [22] is a policy language based on deontic logic, designed mainly for modelling security and privacy properties of pervasive computing environments. Its syntax involves obligation and permission, where policies are defined as constraints over permitted and obligated actions on resources.

2.2 Architecture Description Languages (ADLs)

Research on formal specification of architectures can be categorised into two groups of languages for software and hardware architectures, respectively. Darwin [23], one of the first languages for architectures, defined interaction of components through *bindings*. Bindings associate services

required by a component with the services provided by others. Its semantics is based on the π-calculus [24], a process algebra that makes Darwin capable of modelling dynamic architectures. In Wright [25], components are associated via the *connector* elements instead of bindings. Its semantics is defined in another process algebra, CSP [26], with the architecture specific *port processes* that specify external behaviour of a component, and *spec process*, the internal behaviour of a component.

Similar to Darwin, Rapide [27] defines connections between the required service and provided service "ports" of components. Similar to Wright, Rapide also supports connectors, but in a more limited way (e.g. no first class connector elements), and hence, the user can only specify explicit links between the the required and provided services. Unlike Wright, Rapide also defines the actions *in* and *out* for asynchronous communication. The semantics of Rapide is based on the event pattern language [27], and is defined as a partially ordered set of events. Among the more recent ADLs, SOFA [28] also defines connectors, which the user can specify based on four types of communication, a procedure call, messaging, streaming, and blackboard. The semantics of SOFA is based on Behaviour Protocol [29], a simplified version of CSP.

AADL [30], one of the most broadly-used ADLs, is specifically designed for embedded systems. AADL defines three groups of components, one for software architectures (including thread, process, and subprogram), the second one is for hardware architectures (such as processor, memory), and the last group is for specifying composite types. In AADL, ports and subprogram-calls are used to define interaction between components. PRISMA [31], another recent ADL, was designed to address aspect-oriented software engineering. Similar to Wright, PRISMA defines first-class connector elements, which are specified with a set of roles (i.e. components) and the behaviour of the roles is defined by aspects. The semantics of PRISMA is defined with modal logic and π-calculus. A recent attempt of architectures specification towards automation is proposed in the project, called CONNECT [32]. The semantics of this ADL is based on the FSP (finite state process) algebra [33], which allows automation and stochastic analyses of architectures. Finally, UML has also been used to specify architectures in practice, however, it is more high-level and lacks formal semantics. We note that none of these ADLs support the specification of data protection and privacy properties.

2.3 Comparison with our work

The main differences between the policy languages above and our work is that, for instance, P3P, APPEL, XPref and even PPL are mainly designed for web applications/services, and the policies are defined in a XML-based language, with restricted options for the users, while ours is designed for any type of services. In addition, our policy language variant is defined on data types (data type centred language), and supports a more systematic and fine-grained policy specification, as its syntax and semantics cover seven sub-policies capturing a representative data life-cycle (from the point the data is collected until its deletion). Our language variant is inspired by the ones proposed in [34, 35], which were proposed for biometrics surveillance systems and log design. We modified and extend those to specify different data protection requirements.

Unlike the ADLs above, our architecture language variant is designed to capture the data protection and privacy properties, and also supports cryptographic primitives. Our language is data type centred, and its semantics does not rely on process algebra like most above mentioned ADLs but instead is based on the state of all the defined data types in a system. This concept was applied in some of the previous works, such as in [36, 37]. The language variants in [36, 37] mainly focus on the computation and integrity verification of data based on trust relations. Unlike [36, 37], the language variant in this paper focuses primarily on data protection and privacy properties, rather than the data integrity perspective.

Finally, to the best of our knowledge, this is the first work that studies and proposes a fully automated conformance check between the policy and architecture levels. Our verification engine is based on the syntax of our policy and architecture language variants, and logic resolution based proofs.

3 The Specification of a Data Protection Policy

A policy is defined from the perspective of a data controller. Here, we assume that the data controllers are service providers who collect, store, use or transfer the personal data of the data subjects. The data subjects in our case are system users whose personal data is/will be collected and used by the data controller for some purposes.

3.1 Proposed Policy Syntax

A policy of a service provider, *sp*, is defined on a finite set of different entities $EntitySet^{sp}_{pol} = \{E_{i_1},\ldots, E_{i_n}\}$, and a finite set of data types $DataTypes^{sp}_{pol} = \{\theta_1,\ldots, \theta_m\}$, used by the service. An entity can be a data subject, data controller, organisations, hardware/software components.

Definition 1 *(Data Protection Policy). The syntax of the data protection policies is defined as the collection of seven sub-policies on a given data type, namely:*

$POL_{DataTypes^{sp}_{pol}} = \text{Pol}_{Col} \times \text{Pol}_{Use} \times \text{Pol}_{Str} \times \text{Pol}_{Del} \times \text{Pol}_{Fw} \times \text{Pol}_{Has} \times \text{Pol}_{Link}.$

where:

1. $\text{Pol}_{Col} = \text{Cons}_{col} \times \text{CPurp}.$ (Data Collection Sub-policy)
2. $\text{Pol}_{Use} = \text{Cons}_{use} \times \text{UPurp}.$ (Data Usage Sub-policy)
3. $\text{Pol}_{Str} = \text{Cons}_{str} \times \text{Where}_{str}.$ (Data Storage Sub-policy)
4. $\text{Pol}_{Del} = \text{FromWhere}_{del} \times \text{Del}_{delay}.$ (Data Retention Sub-policy)
5. $\text{Pol}_{Fw} = \text{Cons}_{fw} \times \text{FwTo} \times \text{FwPurp}.$ (Data Transfer Sub-policy)
6. $\text{Pol}_{Has} = \text{Who}_{canhave}.$ (Data Possession Sub-policy)
7. $\text{Pol}_{Link} = \text{Who}_{canlink}.$ (Data Connection Sub-policy)

1. The data collection sub-policy specifies whether a collection consent is required ($Cons_{col}$) and a set of collection purposes (*CPurp*). These aim to capture the consent and purposes limitation requirements in *Article 6* [9] and *Article 5(1)(b)* [38] of the GDPR.
2. The data usage sub-policy specifies whether a usage consent is required ($Cons_{use}$) for using a type of data, besides the set of purposes of the data usage (*UPurp*). These capture the *Article 6* [9] and *Article 30(1)(b))* [39] of the GDPR, respectively.
3. The data storage sub-policy specifies whether a storage consent is required ($Cons_{str}$) for storing a piece of data, and where the data can be stored ($Where_{str}$). These elements partly capture the storage limitation principle in *Article 5(1)(e)* [38] of the GDPR.
4. The data deletion sub-policy specifies from where the data can be deleted ($FromWhere_{del}$), alongside the corresponding deletion delay (Del_{del}). These elements partly capture the *Article 5(1)(e)* and *Article 17(1)(a)* [40] of the GDPR.
5. The data transfer sub-policy defines whether a transfer consent is required ($Cons_{fw}$), and all the entities (*FwTo*) to which the data can be transferred with the given purposes (*FwPurp*). These partly capture the requirement of transferring data the third-party organisations in *Article 46(1)* [41], GDPR.
6. The data possession sub-policy determines who has the right to possess a piece of data of given type.

π_θ	A policy on a data type θ, where $\pi_\theta = (\pi_{col}, \pi_{use}, \pi_{str}, \pi_{del}, \pi_{fw}, \pi_{has}, \pi_{link})$, and $\pi_\theta \in POL_{DataTypes^{sp}_{pol}}$.
π_{col}	A data collection sub-policy, $\pi_{col} \in \text{Pol}_{Col}$.
π_{use}	A usage sub-policy, $\pi_{use} \in \text{Pol}_{Use}$.
π_{str}	A storage sub-policy, $\pi_{str} \in \text{Pol}_{Str}$.
π_{del}	A retention sub-policy, $\pi_{del} \in \text{Pol}_{Del}$.
π_{fw}	A transfer sub-policy, $\pi_{fw} \in \text{Pol}_{Fw}$.
π_{has}	A data possession sub-policy, $\pi_{has} \in \text{Pol}_{Has}$.
π_{link}	A data connection sub-policy, $\pi_{link} \in \text{Pol}_{Link}$.
$\pi_\theta.\pi_*$	A sub-policy π_* of π_θ, where $* \in \{col, use, str, del, fw, has, link\}$.
$\pi_*.arg$	A reference to an argument *arg* of a sub-policy π_*.
cons	Specify if a consent is required (Y for Yes, N for No), where $cons \in Cons_{col}$, or $cons \in Cons_{use}$, or $cons \in Cons_{str}$.
upurp, cpurp, fwpurp	A set of usage, collection, and forward purposes, respectively, where each set is of the form $\{act_1{:}\theta_1, \ldots, act_n{:}\theta_n\}$, and $upurp \in UPurp$, $cpurp \in CPurp$, $fwpurp \in FwPurp$.
$act_i{:}\theta_i$	A purpose defined for a policy π_θ. It specifies that a piece of data of type θ is used (collect, forward) for an action act_i, and as a result we get a piece of data of type θ_i.
where	A set of places where a piece of data of type θ can be stored ($where \in \text{Where}_{str}$).
fromwhere	A set of places from where a piece of data of type θ can be deleted ($fromwhere \in \text{FromWhere}_{del}$).
deld	A deletion delay value, that can be *tt* or *dd* ($deld \in \text{Del}_{delay}$).
tt, dd	A non-specific time value, and a numerical time value, respectively.
fwto	A set of entities to which a piece of data can be transferred ($fwto \in \text{FwTo}$).
whocanhave	A set of entities who has the right to have a type of data ($whocanhave \in \text{Who}_{canhave}$).
whocanlink	A set that contains which entity has the right to link which pairs of types of data ($whocanlink \in \text{Who}_{canlink}$).

Table 1: The notations used in the policy syntax.

7. The data connection sub-policy determines who has the right to link two types of data.

A policy is defined on a data type (θ), specifically, let π_θ, $\pi_\theta \in POL_{DataTypes^{sp}_{pol}}$, be a policy defined on a data type θ, and on the seven sub-policies $\pi_{col} \in Pol_{Col}$, $\pi_{use} \in Pol_{Use}$, $\pi_{str} \in Pol_{Str}$, $\pi_{del} \in Pol_{Del}$, $\pi_{fw} \in Pol_{Fw}$, $\pi_{has} \in Pol_{Has}$, $\pi_{link} \in Pol_{Link}$, where

$$\pi_\theta = (\pi_{col}, \pi_{use}, \pi_{str}, \pi_{del}, \pi_{fw}, \pi_{has}, \pi_{link}).$$

Each sub-policy of π_θ is defined as follows:

1. $\pi_{col} = (cons, cpurp)$, where $cons \in \{Y, N\}$ that specifies whether consent is required to be collected from the data subjects (Y) or not (N) for a data type θ, and *cpurp* is a set of collection purposes. A purpose has the form $act_i{:}\theta_i$, which specifies that a piece of data of type θ is collected by the service provider to perform an action act_i in order to get some data of type θ_i (e.g. $\theta = name$ is collected for *creating* and *account*, i.e. the purpose is *creating*:*account*).

2. $\pi_{use} = (cons, upurp)$, with a usage consent requirement, $cons \in \{Y, N\}$, and *upurp*, a set of usage purposes.

3. $\pi_{str} = (cons, where)$, in which *where* is a set of places where a piece of data of type θ can be stored, for instance, in a client's machine ($where = \{clientpc\}$), at a third party cloud service, or in the service provider's main or backup storage places (denoted by *mainstorage*, *backupstorage*).

θ	A data type value, e.g. θ = *name*.
θ'	A data type value that we get as a result of a *service_spec_use_event* (e.g. createat or calculateat) on a piece of data of type θ and value v.
v	The value of a piece of data of type θ (e.g. v = *Peter*, for θ = *name*).
t	The time value when an event takes place.
E_{to}	An entity value to whom a piece of data is transferred/forwarded (e.g. E_{to} = *police*).
E_{from}	An entity value from which a piece of data is originated (e.g. E_{from} = *clientpc*).
place	A place where a piece of data of type θ and value v is stored. It can be *mainstorage*, *backupstorage* of a serv. provider, or some other service spec. place.

Table 2: The notations used in the policy semantics.

4. π_{del} = (*fromwhere*, *deld*), where
   - *fromwhere* contains the locations from where a piece of data can be deleted. This strongly depends on the storage locations, *where*, defined in the storage policy (point 3).
   - *deld* is the delay value for deletion. This value can be either *tt*, which refers to a "*non specific* time", or a specific "numerical" time value (e.g. 1 day, 10 mins, 5 years, etc.).
5. π_{fw} = (*cons*, *fwto*, *fwpurp*), where *cons* specifies the requirements for the data transfer consent, and *fwto* specifies a set of entities to whom the data can be transferred. Finally, *fwpurp* is a set of purposes for the data transfer.
6. π_{has} = *whocanhave*, where *whocanhave* = $\{E_1,\ldots, E_k\}$ is a set of entities in the service that have the right to have or possess a piece of data of type θ. If we forbid for a given entity to be able to have a given data type, then the entity must not have it (by any means, e.g. by intercepting, eavesdropping, or calculating, etc.).
7. π_{link} = *whocanlink*, where *whocanlink* = $\{(E_1,\theta_1),\ldots, (E_k,\theta_k)\}$, is a set of pairs of entities and data types defined in the service. Each pair (E_i, θ_i) specifies that E_i has the right to link two pieces of data of types θ and θ_i. For instance, whether a service provider has the right to link a piece of information about someone's disease with their work place.

Finally, let $\{\theta_1, \ldots, \theta_m\}$ be a set of all data types used by the service of a provider *sp*, we have:

The data protection policy of a service provider *sp* is defined by the set

$$\mathcal{PL} = \{\pi_{\theta_1}, \ldots, \pi_{\theta_m}\}.$$

3.2 Proposed Policy Semantics

3.2.1 Events

The semantics of the policy syntax can be defined using the events that capture the actions performed by different entities during an instance of a system operation. An event is defined by a tuple starting with an event name that denotes an action carried out by an entity, followed by the time of the event, and some further action-specific parameters.

Our language includes the following "built-in" events: *cconsentat*, *collectat*, *uconsentat*, *sconsentat*, *service_spec_use_event*, *storeat*, *deleteat*, *fwconsentat*, and *forwardat*, defined as follows:

Ev1*:** (cconsentat***, t, E_{from}, θ). This event specifies that a data collection consent is being collected at time t, by the service provider for a piece of data of type θ from an entity E_{from}.

E.g. *(**cconsentat**, 2020.01.21.11:18, client, personalinfo)*

Ev2*:** (collectat***, t, E_{from}, θ, v). This event specifies when a piece of data of type θ and value v is collected by the service provider from E_{from} at time t.

E.g. *(**collectat**, 2020.01.21.11:20, client, personalinfo, Peter)*

Ev3*:** (uconsentat***, t, E_{from}, θ). This event specifies that a data usage consent is collected by the service provider at time t from E_{from}.

E.g. *(**uconsentat**, 2020.01.21.11:18, client, energyconsumption)*

Ev4*:** (service_spec_use_event***, t, E_{from}, θ', θ, v). This captures a service specific event, specifically, a piece of data type θ from E_{from} is used by the service provider to obtain a piece of data type θ' after performing the action *service_spec_use_event*.

E.g. *(**createat**, 2020.01.30.15:45, client, bill, energyconsumption, 20kWh)*

Ev5*:** (sconsentat***, t, E_{from}, θ). This event specifies that a data storage consent is being collected by the service provider for a piece of data of type θ from an entity E_{from}.

E.g. *(**sconsentat**, 2020.01.30.15:45, client, sickness)*

Ev6*:** (storeat***, t, E_{from}, θ, v, *place*). This event specifies that a piece of data of type θ and value v is stored at a place *place* at time t. We note that unlike the rest events, which all defines an action carried out by a service provider, this event can capture an action by a different entity as well. For example, if *place* = *clientpc*, then event *Ev6* can refer to a storage action done by a client PC.

E.g. *(**storeat**, 2020.01.30.15:45, client, sickness, leukemia, backupstorage)*

Ev7*:** (deleteat***, t, E_{from}, θ, v, *place*). This event specifies that at some time t, a service provider deletes a piece of data of type θ and value v from a place *place*.

E.g. *(**deleteat**, 2020.01.30.15:45, client, sickness, leukemia, mainstorage)*

Ev8*:** (fwconsentat***, t, E_{to}, E_{from}, θ). This event specifies that a service provider is collecting a data transfer consent on a piece of data of type θ originated from E_{from}.

E.g. *(**fwconsentat**, 2020.01.21.11:18, insurancecompany, client, personalinfo)*

Ev9*:** (forwardat***, t, E_{to}, E_{from}, θ, v). This captures that at time t, E_{to} receives a piece of data transferred by a service provider, which has a type θ and value v, and is originally from E_{from}.

E.g. *(**forwardat**, 2020.01.21.11:18, insurancecompany, client, personalinfo, Peter)*

3.2.2 Policy-Compliant System Operation Trace

We discuss the policy compliant system operations based on the events defined in Section 3.2.1. Eleven rules (C_1-C_{11}) are defined, where each rule defines a system operation that respects a sub-policy in Definition 1 (see Figure 2 for some illustration). In the sequel, we refer to each element e of a tuple *tup* as *tup.e*, for example, we refer to π_{str} in π_θ as $\pi_\theta.\pi_{str}$. Finally, in the following rules, we assume that a piece of data of type θ has not been deleted yet between any two actions.

- C_1 (collection consent): If in $\pi_\theta.\pi_{col}$, $cons = Y$, then a consent must be collected before the collection of the data itself. Formally:

If during a system operation trace, $\exists$ *Ev1* (***collectat***, t, E_{from}, θ, v) for some time t, then $\exists$ *Ev2* (***cconsentat***, t', E_{from}, θ) for some t' in the trace, such that $t \geq t'$.

- C_2 (collection purposes): If in $\pi_\theta.\pi_{col}$, $cpurp = \{act_1{:}\theta_1, \ldots, act_n{:}\theta_n\}$, then a piece of data of type θ must not be collected for any purpose that is not in *cpurp*. Formally:

If during a system operation trace, $\exists$ (***collectat***, t, E_{from}, θ, v) for some time t, then for all instances of *Ev4*, namely, each event (act', t', E_{from}, θ', θ, v) in the trace, where $t' \geq t$ we have $act'{:}\theta' \in cpurp$. (Note that act' is a service specific event, *service_spec_use_event* in *Ev4*.)

- C_3 (usage consent): For π_θ, if $cons = Y$ in $\pi_\theta.\pi_{use}$, then consent must be collected before the usage of the data. Formally:

If during a system operation trace, $\exists$ (***service_spec_use_event***, t, E_{from}, θ', θ, v) for some time t, then $\exists$ (***uconsentat***, t', E_{from}, θ) for some t', such that $t \geq t'$.

- C_4 (usage purposes): If in $\pi_\theta.\pi_{use}$, $upurp = \{act_1{:}\theta_1, \ldots, act_n{:}\theta_n\}$, then a piece of data of type θ must not be collected for any purpose not specified in *upurp*. Formally:

If during a system operation trace, there is an instance of $Ev4$, (act', t, E_{from}, θ', θ, v) for some time t, then $act'{:}\theta' \in upurp$.

- C_5 (storage consent): If in $\pi_\theta.\pi_{str}$, $cons = Y$, then a consent must be collected before the storage of the data itself. Formally:

If during a system operation trace, $\exists$ (***storeat***, t, E_{from}, θ, v, *places*) for some time t, then $\exists$ (***sconsentat***, t', E_{from}, E, θ) for some t' in the trace, such that $t \geq t'$.

- C_6 (storage places): If in $\pi_\theta.\pi_{str}$, $where = \{place_1, \ldots, place_m\}$, then this data type must not be stored in any place that is not in *where*. Formally:

If during a system operation trace, $\exists$ (***storeat***, t, E_{from}, θ, v, *place*) for some time t, then $place \in where$.

- C_7 (deletion places): If in $\pi_\theta.\pi_{del}$, $fromwhere = \{place_1, \ldots, place_m\}$, then this data type must be deleted from all the places defined in *fromwhere*. Formally:

For all the events (***deleteat***, t_1, E_{from}, θ, v, $place_1$), …, (***deleteat***, t_n, E_{from}, θ, v, $place_n$) in a system operation trace, $\{place_1, \ldots, place_n\} = fromwhere$.

- C_8 (deletion delay): If in $\pi_\theta.\pi_{del}$, $deld = delay$, then this data type must be deleted up to the delay *delay* from the time of its collection. Formally:

If during a system operation trace, $\exists$ (***collectat***, t, E_{from}, θ, v) for some time t, and $\exists$ events (***deleteat***, t_1, E_{from}, θ, v, $places_1$), …, (***deleteat***, t_n, E_{from}, θ, v, $places_n$), for some n, then $t + delay \geq t_1 \geq t, \ldots, t + delay \geq t_n \geq t$.

- C_9 (transfer consent): If in $\pi_\theta.\pi_{fw}$, $cons = Y$, then a consent must be collected before the transfer of a piece of data of type θ. Formally:

If during a system operation trace, $\exists$ (***forwardat***, t, E_{to}, E_{from}, θ, v) for some time t, then $\exists$ (***fwconsentat***, t', E_{to}, E_{from}, θ), such that $t \geq t'$.

- C_{10} (transfer to): If in $\pi_\theta.\pi_{fw}$, $fwto = \{E_1, \ldots, E_n\}$, then a piece of data of type θ must not be transferred to any entity not defined in *fwto*. Formally:

If during a system operation trace, $\exists$ (***forwardat***, t, E_{to}, E_{from}, θ, v) for some time t, then $E_{to} \in fwto$.

- C_{11} (transfer purposes): If in $\pi_\theta.\pi_{fw}$, $fwpurp = \{act_1{:}\theta_1, \ldots, act_n{:}\theta_n\}$, then a piece of data of type θ must not be transferred for any purpose not defined in *fwpurp*. Formally:

If during a system operation trace, $\exists$ (***forwardat***, t,E_{to}, E_{from}, θ, v) for some time t, then for all instances of $Ev4$, namely, each event (act', t', E_{from}, θ', θ, v) in the trace, where $t' \geq t$ we have $act'{:}\theta' \in fwpurp$.

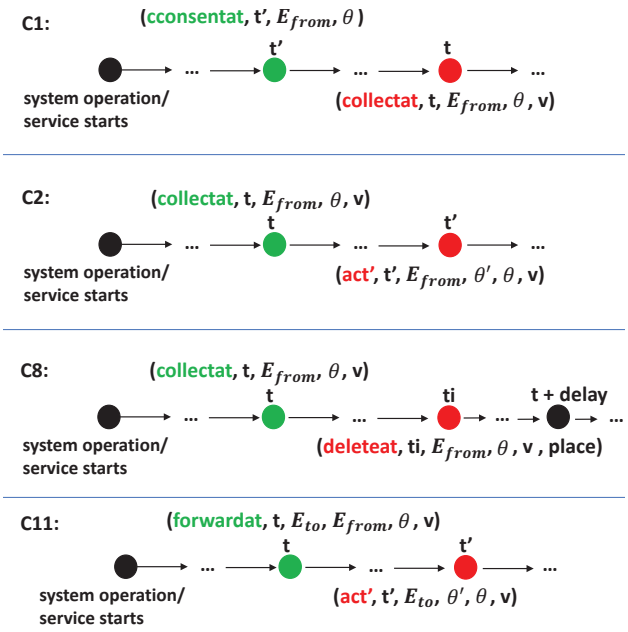

Figure 2: The illustration of some policy compliance rules.

3.2.3 The semantics of the policy events

In this section, we discuss the semantics of the policy events defined in Section 3.2.1. For this purpose, we define the so-called *data states*. We have the following assumptions before starting with the definitions:

- We assume a service provider *sp* that provides a service *serv*.
- We assume that the system that provides the service *serv* involves the entities $E_1, \ldots, E_m$ (namely, $EntitySet^{sp}_{pol} = \{E_1, \ldots, E_m\}$.).

The semantics of the policy events is defined based on the so-called *data states of an entity* and the *data states of a service*.

Data: In this context, a piece of data is defined by a pair of data type and the entity from which the data is originated, namely, $data = (\theta, E_{from})$, e.g. $data = (name, clientpc)$ or $(disease, healthapp)$.

Data value: Each piece of data, $data = (\theta, E_{from})$, takes its value, v, during a system run. For example, during a system run the data $(disease, healthapp)$ can take the value $v = coronavirus$.

Assigning a piece of data a value: We denote that data (θ, E_{from}) takes the value v by

$$(\theta, E_{from}) : v, \text{ e.g. } (disease, healthapp) : coronavirus.$$

The data state of an entity E. Given a service provider *sp* and the service *serv*, the data state of an entity E, $E \in EntitySet^{sp}_{pol}$, captures the actual values of *data*, $data = (\theta, E_{from})$, during a system run, for all $\theta \in DataTypes^{sp}_{pol}$ from the perspective of E.

Intuitively, the data state of an entity E captures how the value of a piece of data, $data = (\theta, E_{from})$, changes from the perspective of E during a system operation trace.

Formally, the data state of E is defined by the function $dstate_E$ that assigns a value (including the undefined value $\perp$) to a piece of data $data = (\theta, E_{from})$.

The data state of an entity E, $E \in \boldsymbol{EntitySet}^{sp}_{pol}$, is defined by a function:

$dstate_E : DataTuple \mapsto ValueTuple_\perp$, where

$DataTuple = (datadomain_1, \ldots, datadomain_n)$ is the tuple of all defined *data* in *serv*, where *datadomain* is the set of all *data* defined in *serv*, besides $data = (\theta, E_{from})$.

$ValueTuple_\perp = (valuedomain_1, \ldots, valuedomain_n)$ is the tuple of values of the data in *DataTuple*, where *valuedomain* denotes the domain of the values, including the undefined value $\perp$.

$DataTuple \mapsto ValueTuple_\perp$ means that each $datadomain_j$ is assigned a value $valuedomain_j$.

Finally, each $E \in EntitySet^{sp}_{pol}$ has one data state, $dstate_E$.

For example, let $E = sp$, and $DataTuple = ((disease, appOfTom), (id, appOfTom))$, namely, there are two kinds of data defined in the service, disease and an ID originated from *appOfTom*. At the very start of the service, the value of both kinds of data is undefined. Specifically, the initial data state of *sp* is $dstate^{init}_{sp} = (\perp, \perp)$. During the system run, when *sp* receives the disease information from Tom's app, the data state can change to $dstate_{sp} = (coronavirus, \perp)$. Similarly, when *sp* receives the ID *12345* from Tom's app, we can get the updated state as $dstate_{sp} = (coronavirus, 12345)$. Later, the value of $(disease, appOfTom)$ can change to *influenza*, and the state of *sp* changes to $dstate_{sp} = (influenza, 12345)$.

The data state of a service. We assume a service *serv* of a provider *sp*, where $EntitySet^{sp}_{pol} = \{E_1, \ldots, E_m\}$. *The data state of Service* is the tuple of the data states of entities $E_1, \ldots, E_m$, and a time variable.

The data state of a service *serv*

$$\delta_{serv} = (dstate_{E_1}, \ldots, dstate_{E_m}, TV).$$

The *initial data state of a service* of a provider *sp* is denoted by δ^{init}_{serv}, which is the collection of the initial states of each defined entity in $EntitySet^{sp}_{pol}$. Initially, at the start of a service, all the data have the undefined value, $\perp$.

The initial state of a service *serv*:

$$\begin{aligned} \delta^{init}_{serv} &= (dstate^{init}_{E_1}, \ldots, dstate^{init}_{E_m}, TV^{init}) \text{ with} \\ \forall i \in [1, m],\ dstate^{init}_{E_i} &= (\perp, \ldots, \perp) \\ TV^{init} &= \perp \text{ (* denoting an undefined time value *).} \end{aligned}$$

Event trace and state updates: An event trace of the operation of a service is denoted by τ, and contains a finite sequence of events defined in Figure 3.2.1, happening during a corresponding system operation. Below, we define the semantics function, denoted by S^{pol}_{Trace}, which defines how a trace τ changes the state of a service (Figure 3).

S^{pol}_{Trace} relies on the function S^{pol}_{Ev} that defines how an event in τ changes the current global state of $\mathcal{PL}$.

Semantics function (Policy)

$$S^{pol}_{Trace} : EventTrace \times ServDataStates \mapsto ServDataStates$$

$$S^{pol}_{Ev} : Event \times ServDataStates \mapsto ServDataStates$$

where *ServDataStates* denotes the "domain" of the states of services,
EventTrace denotes the "domain" of the event traces in a system run.
Event denotes the "domain" of the events.

$S^{pol}_{Trace}(emptytrace, \delta_{serv}) = \delta_{serv}$

$S^{pol}_{Trace}(event.\tau, \delta_{serv}) = S^{pol}_{Trace}(\tau, S^{pol}_{Ev}(event, \delta_{serv}))$

$S^{pol}_{Ev}((\textbf{\textit{cconsentat}}, t, E_{from}, \theta), \delta_{serv}) = \delta_{serv}[dstate_{sp}/dstate_{sp}[(cconsenttype, E_{from}) : v_{cconsent}], TV/t]$,
where $v_{cconsent}$ is the value of a collection consent.

$S^{pol}_{Ev}((\textbf{\textit{collectat}}, t, E_{from}, \theta, v), \delta_{serv}) = \delta_{serv}\ [dstate_{sp}/dstate_{sp}[(\theta, E_{from}) : v], TV/t]$

$S^{pol}_{Ev}((\textbf{\textit{uconsentat}}, t, E_{from}, \theta), \delta_{serv}) = \delta_{serv}\ [dstate_{sp}/dstate_{sp}[(uconsenttype, E_{from}) : v_{uconsent}], TV/t]$
where $v_{uconsent}$ is the value of a usage consent.

$S^{pol}_{Ev}((\textbf{\textit{createat}}, t, E_{from}, \theta', \theta, v), \delta_{serv}) = \delta_{serv}\ [dstate_{sp}/dstate_{sp}[(\theta', E_{from}) : v], TV/t]$

$S^{pol}_{Ev}((\textbf{\textit{calculateat}}, t, E_{from}, \theta', \theta, v), \delta_{serv}) = \delta_{serv}\ [dstate_{sp}/dstate_{sp}[(\theta', E_{from}) : v], TV/t]$

$S^{pol}_{Ev}((\textbf{\textit{sconsentat}}, t, E_{from}, \theta), \delta_{serv}) = \delta_{serv}[dstate_{sp}/dstate_{sp}[(sconsenttype, E_{from}) : v_{sconsent}], TV/t]$
where $v_{sconsent}$ is the value of a storage consent.

$S^{pol}_{Ev}((\textbf{\textit{storeat}}, t, E_{from}, \theta, v, place), \delta_{serv}) = \delta_{serv}\ [dstate_{place}/dstate_{place}[(\theta, E_{from}) : v], TV/t]$

$S^{pol}_{Ev}((\textbf{\textit{deleteat}}, t, E_{from}, \theta, v, place), \delta_{serv})$
$= \delta_{serv}\ [dstate_{place}/dstate_{place}[(\theta, E_{from}) : \perp, (cconsenttype, E_{from}) : \perp, (uconsenttype, E_{from}) : \perp,$
$(sconsenttype, E_{from}) : \perp, (fwconsenttype, E_{from}) : \perp], TV/t)]$.

$S^{pol}_{Ev}((\textbf{\textit{fwconsentat}}, t, E_{to}, E_{from}, \theta), \delta_{serv}) = \delta_{serv}\ [dstate_{sp}/dstate_{sp}\ [(fwconsenttype, E_{from}) : v_{fwconsent}], TV/t]$
where $v_{fwconsent}$ is the value of a transfer consent.

$S^{pol}_{Ev}((\textbf{\textit{forwardat}}, t, E_{to}, E_{from}, \theta, v), \delta_{serv}) = \delta_{serv}\ [dstate_{E_{to}}/dstate_{E_{to}}[(\theta, E_{from}) : v], TV/t]$

Figure 3: The semantics of the policy events, where *createat* and *calculateat* are the two instances of *service_spec_use_event* (*Ev4*).

Figure 3 summarises the semantics of the events defined in Section 3.2.1. Each event can either change the global state or leave it unchanged. To capture the modification made by an event at time t on the state of the variable (θ, E_{from}) from the perspective of an entity E we write $\delta_{serv}[dstate_E/dstate_E[(\theta, E_{from}) : v], TV/t]$ (or $\delta_{serv}[dstate_E/dstate_E[(\theta, E_{from}) : \perp], TV/t]$ in case of the undefined value, e.g. when a piece of data has been deleted). Intuitively, this notation captures that the old state $dstate_E$ is replaced with the new state $dstate_E[(\theta, E_{from}) : v]$ ($dstate_E[(\theta, E_{from}) : \perp]$), in which the variable (θ, E_{from}) has been given the value v (or the undefined value $\perp$) as a result of the event, the time variable TV is given the time value t.

3.3 Well-formed Policies

A policy $\mathcal{PL}$, $\mathcal{PL} = \{\pi_{\theta_1}, \ldots, \pi_{\theta_m}\}$, is well-formed if for each data type $\theta_1, \ldots, \theta_n$ (where $\pi_{\theta_j} = (\pi_{col}, \pi_{use}, \pi_{str}, \pi_{del}, \pi_{fw}, \pi_{has}, \pi_{link})$), there is not any pair of sub-policies which are conflicting.

$\perp$	An undefined data and time value.
$serv$	A service.
$dstate_E$	The data state of an entity E.
δ_{serv}	The data state of a service $serv$.
$dstate_E^{init}$	The initial data state of an entity E.
δ_{serv}^{init}	The initial data state of a service $serv$.
τ	An event trace.
TV	A time variable.
$\delta_{serv}[dstate_E/dstate_E[(\theta, E_{from}) : v], TV/t]$	A change made on the state of a service $serv$ as a result of an event. Here, inside δ_{serv}, the data (θ, E_{from}) is assigned a value v inside the data state of E ($dstate_E$). Besides, TV is assigned the time value t.
$dstate_E/dstate_E[(\theta, E_{from}) : v]$	The current data state of E is updated with a new one in which the data (θ, E_{from}) is assigned a value v.

Table 3: The notations used in the semantics of the policy events.

For example, the pair (π_{col}, π_{has}) or (π_{str}, π_{has}) is conflicting if the collection (π_{col}) or storage (π_{str}) sub-policy specifies that an entity E can collect or store a type of data (θ_j), but in π_{has}, E does not have the right to have/posses this type of data.

4 The Corresponding Architecture Level

System architectures describe how a system is composed of components and how these components relate to each other (which is abstracted away from the policy), however, they abstract away from the implementation details, such as the cryptographic algorithms, the specific order and timing of the messages (e.g. we only define that *sp* can receive a sickness record from a *health app*, but we do not specify the authentication, key exchange or communication protocol behind that).

4.1 Proposed Architecture Syntax

In line with the policy specification, a system architecture is defined on a set of entities (components) and data types. For a service provider *sp*, we define a finite set of entities, $EntitySet_{arch}^{sp} = \{E_{i_1}, \ldots, E_{i_n}\}$. Let $DataTypes_{arch}^{sp} = \{\theta_1, \ldots, \theta_m\}$ be the set of all the data types defined in an architecture. We assume the finite sets of data variables Var, ($X_\theta \in Var$), time variables ($TT \in TVar$), data values Val ($V_\theta \in Val$), respectively. Finally, we define the finite sets of the time and deletion delay values ($t \in TVal$, and $dd \in DVal$), respectively.

Terms: As shown in Figure 4, a term, denoted by T, can be:

- A variable (X_θ) that represents some data of type θ, and a data constant or value (V_θ) of type θ.
- A special term *ds* that specifies the real identity value of a data subject (this will be used for modelling pseudonyms).
- A term can be an entity E that specifies any software or hardware component, organisations, a data controller, or a data subject.
- A special function (*SpecFunc*) that specifies the time, pseudonyms and four types of consents.
- Finally, a term can be a time value (Ti).

A variable $X_\theta \in Var$ represents a piece of data of type θ supported by *sp*, such as the users' personal information, photos, videos, energy data, insurance number, etc. X_θ can be a non-function/simple data D_θ of type θ, a cryptographic or meta function (*CryptoFunc*), and finally, any other service specific function.

Terms:

$T ::= X_\theta \mid V_\theta \mid ds \mid E \mid SpecFunc \mid Ti.$

$X_\theta ::= D_\theta \mid CryptoFunc \mid Service\_spec\_fun(X_{\theta_1}, \ldots, X_{\theta_n}).$
(where TYPE($CryptoFunc$) $= \theta$, TYPE($Service\_spec\_fun(X_{\theta_1}, \ldots, X_{\theta_n})$) $= \theta$).

$Ti ::= dd \mid TT.$

$SpecFunc ::= \boldsymbol{Time}(Ti) \mid \boldsymbol{P}(ds) \mid \boldsymbol{Cconsent}(Data) \mid \boldsymbol{Uconsent}(Data).$
$\mid \boldsymbol{Sconsent}(Data) \mid \boldsymbol{Fwconsent}(Data, E_{to})$

$Data ::= (X_\theta, E_{from})$ where E_{from} is an entity who originally sent the data X_θ.

$CryptoFunc ::= \boldsymbol{Sk}(X_{pkeytype}) \mid \boldsymbol{Senc}(X_\theta, X_{keytype}) \mid \boldsymbol{Aenc}(X_\theta, X_{pkeytype}).$
$\mid \boldsymbol{Hash}(X_\theta) \mid \boldsymbol{Mac}(X_\theta, X_{keytype}) \mid \boldsymbol{Meta}(X_\theta).$

Destructor application on terms:

$G(T_1, \ldots, T_n) \rightarrow T$

Function that returns a type of a term T:

TYPE(T) $= \theta$, where $\theta \in DataTypes^{sp}_{arch}$.

Function *HasAccessTo*:

HasAccessTo: $E_i \in EntitySet^{sp}_{arch} \rightarrow \{E_j \in EntitySet^{sp}_{arch}\}$.

Figure 4: Terms, Destructors and Types.

Functions: The two groups of functions *SpecFunc* and *CryptoFunc* are defined as follows:

- Function ***Time***(Ti) specifies the time with either a non-specific time value TT or a numerical delay value, *dd*. While *dd* captures a numerical time value such as 3 years, 2 months, etc., the value TT is not numerical, and is used to express the informal term "at some point/time". Function $\mathbf{P}(ds)$ specifies a pseudonym of a real identity *ds*.
- ***Cconsent***$(Data)$, ***Uconsent***$(Data)$ and ***Sconsent***$(Data)$, besides $Data = (X_\theta,\ E_{from})$, specify a piece of data of type collection, usage, and storage consent, respectively, on a piece of data X_θ that is originally sent by E_{from}. Finally, ***Fwconsent***$(Data,\ E_{to})$ specifies a transfer consent on *Data*, alongside an entity to whom the data can be transferred (E_{to}).
- **Meta**(X_θ) defines the metadata (information about other data), or information located in the header of the packets (e.g. IP address). For simplicity, they are both modelled by **Meta**.
- The basic cryptographic functions:
  - **Sk**$(X_{pkeytype})$: This function defines a type of private key used in asymmetric key encryption algorithms. Its argument has a type of public key (pkeytype).
  - **Senc**$(X_\theta,\ X_{keytype})$: This defines a type of symmetric key encryption, and has two arguments, a piece of data (of type θ) and a symmetric key (of type keytype).
  - **Aenc**$(X_\theta,\ X_{pkeytype})$: This defines a type of the cipher text resulted from an asymmetric key encryption, and has two arguments, a piece of data and a public key (pkeytype).
  - **Mac**$(X_\theta,\ X_{keytype})$: This defines a type of the message authentication code that has two arguments, a piece of data and a symmetric key.
  - **Hash**(X_θ): This defines a type of the cryptographic hash that has one argument, a piece of data of type θ.

Values: A variable X_θ will be given a specific data value V_θ during an instance of a system run (see Section 4.2). V_θ can be the value of both a simple (non-function) data or a function, and it can also be $\perp$, which denotes an undefined value (every data variable X_θ has the value $\perp$ at the start of a service).

Destructor: This represents an evaluation of a function, used to model a verification procedure. For instance, if $X_{enc} = Senc(X_{name},\ X_{Skey})$ that represents the encryption of data X with the server key X_{Skey}, and X_{Skey} represents a symmetric key, then $G(X_{enc}, X_{Skey}) \rightarrow X$ is $Dec(Senc(X_{name},\ X_{Skey}),\ X_{Skey}) \rightarrow X_{name}$. Note that not all functions have a corresponding destructor, e.g., in case X_{hash} is a one-way cryptographic hash function, $X_{hash} = Hash(X_{password})$, then due to the one-way property there is no destructor (reverse procedure) that returns $X_{password}$ from the hash X_{hash}.

HasAccessTo: This is a function that expects an entity as input and returns a set of other entities defined in the same architecture. It specifies which entity can have access to the data handled/stores/collected by other entities. For example, if E_m and E_p represent a smart meter, and a digital panel, respectively, and we want to specify that the service provider, *sp*, can have access to the panel and the meter, then, we define the relation $HasAccessTo(sp) = \{E_m,\ E_p\}$. It is used for verifying the data possesion and link policies.

4.1.1 System Architecture

The definition of a system architecture: An architecture $\mathcal{PA}$ is defined as a set of *actions* (denoted by $\{\mathcal{F}\}$). The formal definition is given as follows:

- Action $OWN(E,\ X_\theta)$ captures that E can own the data variable X of type θ (during a service regardless of time). Note that X_θ is the originally owned data (not the data obtained/received by E).

$\mathcal{PA} ::= \{\mathcal{F}\}$

$\mathcal{F} ::= OWN(E,\ X_\theta)$
$\quad | \ CALCULATEAT(E,\ X_\theta,\ \textbf{\textit{Time}}(TT))$
$\quad | \ CREATEAT(E,\ X_\theta,\ \textbf{\textit{Time}}(TT))$
$\quad | \ RECEIVEAT(E,\ Data,\ \textbf{\textit{Time}}(TT))$
$\quad | \ RECEIVEAT(E,\ \textbf{\textit{Cconsent}}(Data), \textbf{\textit{Time}}(TT))$
$\quad | \ RECEIVEAT(E,\ \textbf{\textit{Uconsent}}(Data), \textbf{\textit{Time}}(TT))$
$\quad | \ RECEIVEAT(E,\ \textbf{\textit{Sconsent}}(Data),\ \textbf{\textit{Time}}(TT))$
$\quad | \ RECEIVEAT(E,\ \textbf{\textit{Fwconsent}}(Data,\ E_{to}),\ \textbf{\textit{Time}}(TT))$
$\quad | \ STOREAT(E,\ Data,\ \textbf{\textit{Time}}(TT))$
$\quad | \ DELETEWITHIN(E,\ Data,\ \textbf{\textit{Time}}(dd))$
$\quad | \ CALCULATE(E,\ X_\theta)$
$\quad | \ CREATE(E,\ X_\theta)$
$\quad | \ RECEIVE(E,\ Data)$
$\quad | \ STORE(E,\ Data)$

Where $Data = (X_\theta,\ E_{from})$, X_θ is originally sent by E_{from}.

Figure 5: The table shows the syntax of a system architecture with the defined actions between components/entities.

- $CALCULATEAT(E,\ X_\theta,\ \textbf{\textit{Time}}(TT))$ specifies that an entity E can calculate the variable X_θ based on an equation $X_\theta = T$, for some term T at non-specific time TT (e.g. θ = bill, and X_θ = *Bill*(*energyconsumption, tariff*)).
- $CREATEAT(E,\ X_\theta,\ \textbf{\textit{Time}}(TT))$ specifies that E can create a piece of data of type θ, based on an equation $X_\theta = T$ (e.g. θ = account, and X_θ = *Account(name, address)*). The actions *create* and *calculate* merely differ in the nature of T, for example, we calculate a bill, while create an account.
- $RECEIVEAT(E,\ Data,\ \textbf{\textit{Time}}(TT))$ means that E can receive $Data$ (i.e. $(X_\theta,\ E_{from})$) at some non-specific time TT.
- $RECEIVEAT(E,\ \textbf{\textit{Cconsent}}(Data),\ \textbf{\textit{Time}}(TT))$, $RECEIVEAT(E,\ \textbf{\textit{Uconsent}}(Data),\ \textbf{\textit{Time}}(TT))$, and $RECEIVEAT(E,\ \textbf{\textit{Sconsent}}(Data),\ \textbf{\textit{Time}}(TT))$ specify that a collection, usage and storage consent on $Data$, $Data$=$(X_\theta,\ E_{from})$, respectively, can be received by E at time TT.
- $RECEIVEAT(E,\ \textbf{\textit{Fwconsent}}(Data,\ E_{to}),\ Time(TT))$ specifies that a transfer consent on $Data$ and E_{to} can be received by E at time TT.
- $STOREAT(E,\ Data,\ \textbf{\textit{Time}}(TT))$ specifies that $Data$ can be stored at some non-specific time TT in a place E. A place can be ***mainstorage*** and ***backupstorage***, which represent a collection of main storage places such as main servers, and a collection of backup storage places (e.g. backup servers) of a service provider, respectively, or any service specific place (e.g., *clientPC*).
- $DELETEWITHIN(E,\ Data,\ \textbf{\textit{Time}}(dd))$ specifies that $Data$ must be deleted from a place E within a certain time delay dd (where dd is a numerical time value, e.g. 10 years).
- The last four CALCULATE, CREATE, RECEIVE and STORE actions at the end are the corresponding versions of the previous four but without the **Time**() construct. They capture the correspinding actions regardless of time, and are defined for convenient purposes, offering the user an option to specify the simpler actions if they only want to reason about privacy properties. The actions with the **Time**() construct are mainly used for reasoning about data protection properties and requirements (e.g. whether a consent has been collected before

collection, usage, or transfer). The semantics of these four actions are the same as the previous four.

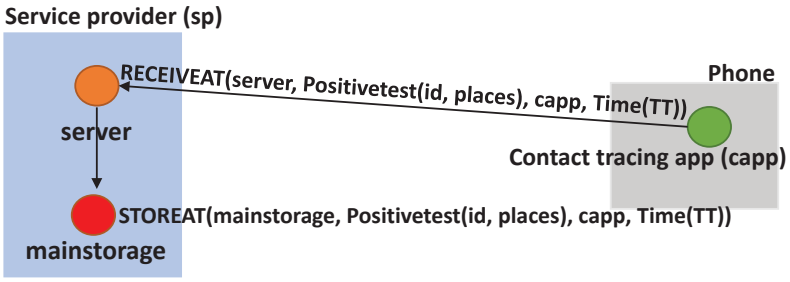

Figure 6: A simple example architecture, where $Data = (X_\theta, E_{from}) = (Positivetest(id, places), capp)$.

An example architecture is shown in Figure 6, where a service provider collects positive (virus) test records sent by contact tracing apps. A record contains an unique ID and a set of places where the phone has been brought to, and the record is stored in the main storage place(s) of *sp*. Here, we also define HasAccessTo(*sp*) = {*server*, *mainstorage*} so that *sp* can have access to *server* and *mainstorage*.

4.2 Proposed Architecture Semantics

Like the policy case, the semantics of an architecture is based on events and system run traces. A trace Γ is a sequence of high-level events $Seq(\epsilon)$ taking place in during a service, as presented in Figure 7.

$\Gamma ::= Seq(\epsilon)$
$\epsilon ::= own(E, X_\theta{:}V_\theta, t)$, for all t in any traces during a service
$\quad | \; calculateat(E, X_\theta{:}T, t)$
$\quad | \; createat(E, X_\theta{:}T, t)$
$\quad | \; receiveat(E, Data{:}V_{TYPE(Data)}, t)$
$\quad | \; receiveat(E, \boldsymbol{Cconsent}(Data){:}V_{cconsent}, t)$
$\quad | \; receiveat(E, \boldsymbol{Uconsent}(Data){:}V_{uconsent}, t)$
$\quad | \; receiveat(E, \boldsymbol{Sconsent}(Data){:}V_{sconsent}, t)$
$\quad | \; receiveat(E, \boldsymbol{Fwconsent}(Data){:}V_{fwconsent}, t)$
$\quad | \; storeat(E, Data{:}V_{TYPE(Data)}, t)$
$\quad | \; deletewithin(E, Data{:}V_{TYPE(Data)}, dd, t).$

Where $Data = (X_\theta, E_{from})$, X_θ is originally sent by E_{from}.

Figure 7: Events defined for architectures.

An event can be seen as an instance of an action defined in Figure 5 that happens at some specific time t (e.g. *2020.01.30.15:45*) during a system run trace. Events are given the same names as the corresponding actions but in lower-case letters in order to avoid confusion.

- Event $own(E, X_\theta{:}V_\theta, t_{all})$ captures that E owns X_θ with a value V_θ at time t_{all} (where t_{all} denotes "all the time" during a service). $X_\theta{:}V_\theta$ means that the variable X_θ is assigned a value V_θ[2].

[2] V_θ can be a name, e.g. Peter, that is assigned to the X_θ during a service/system operation.

- *calculateat*(E, X_θ:T, t) captures that at some time t, E calculates a piece of data of type θ that is equal to a term T (based on the equation X_θ=T, e.g. $X_{hash} = Hash(X_{password})$.).
- *createat*(E, X_θ:T, t) captures that at some time t, E creates a piece of data of type θ that is equal to a term T (e.g. $X_\theta = Account(X_{name}, X_{address})$).
- *receiveat*(E, *Data*:$V_{TYPE(Data)}$, t) specifies that E receives a piece of data of type *TYPE(Data)* and value $V_{TYPE(Data)}$ at some specific time t.
- Events *receiveat*(E, ***Cconsent***(*Data*):$V_{cconsent}$, t), *receiveat*(E, ***Uconsent***(*Data*):$V_{uconsent}$, t), *receiveat*(E, ***Sconsent***(*Data*):$V_{sconsent}$, t), and *receiveat*(E, ***Fwconsent***(*Data*):$V_{fwconsent}$, t) specify that E receives a (collection, usage, storage, or transfer) consent on *Data* with a value V_θ, where θ is a corresponding type of consent ($\theta \in \{cconsent, uconsent, sconsent, fwconsent\}$).
- *storeat*(E, *Data*:$V_{TYPE(Data)}$, t) says that a piece of data of type *TYPE(Data)* is stored in a place E.
- *deletewithin*(E, *Data*:$V_{TYPE(Data)}$, dd, t) specifies that at time t, a piece of data of type *TYPE(Data)* is deleted from a place E, where $t \leq t_{collect} + dd$, where the data was collected at $t_{collect}$[3].

The semantics of each architecture event above is defined by the semantics function, S_T, which specifies the impact made by each event on the states of the data variables (i.e. how the values of X_θ, for all $\theta \in DataTypes^{sp}_{arch}$, changes after an event takes place). For example, let $DataTypes^{sp}_{arch}$ =$\{name, bill\}$, the two types supported by *sp*, and $Entity^{sp}_{arch}$ =$\{sp, client\}$. At the start of the service, the variable states of both *sp* and *client* are ($X_{name} = \perp$, $X_{bill} = \perp$), where $\perp$ is an undefined (initial) value. As a result of an event *own*(*client*, X_{name}:*Peter*, t_{all}), the variable state of *sp* remains unchanged, while the state of *client* has changed to ($X_{name} = Peter$, $X_{bill} = \perp$).

4.3 The Semantics of Architecture Events

States: The semantics of events is defined based on *local states* and the *global state* of the data types defined in a system. Given a service provider *sp*, a local state captures the values of (a data variable) X_θ, for all $\theta \in DataTypes^{sp}_{arch}$ from the perspective of an entity (component) E. Intuitively, a local state of E captures how the value of X_θ, $\theta \in DataTypes^{sp}_{arch}$, changes from the perspective of an E during a system operation.

Formally, a local state of E is a function $State_V$ that assigns a value (including the undefined value $\perp$) to each variable.

Local state of E (denoted by μ_E)

$State_E : Var \mapsto Val_\perp$, where Var is a set of all possible data variables and $Val_\perp$ a set of all possible values, including the undefined value $\perp$.

Assume that there are m entities $E_1, \ldots, E_m$ defined in an architecture. The *global state* of an architecture is the collection of all the local states in a system. A global state is denoted by μ, where $\mu = (\mu_{E_1}, \ldots, \mu_{E_m}, TT)$.

Global state of an architecture (denoted by μ)

$State : State^m_E \times TVar.$

The *initial* (*global*) *state* for an architecture $\mathcal{PA}$ is denoted by σ^{init}, and is the collection of the initial states of each defined entity. Initially the values of all the variables defined in the architecture (including the time variable) have the undefined value, $\perp$.

[3]This can be extended to the time of any other action (e.g. the time when the data is stored).

μ^{init}: **Initial Global State**

$$\mu^{init} = (\mu^{init}_{E_1}, \ldots, \mu^{init}_{E_m}, TT^{init}) \; with$$
$$\forall i \in [1, m], \; \mu^{init}_{E_i} = (\bot, \ldots, \bot)$$
$$TT^{init} = \bot.$$

Event trace and state updates: An event trace of an architecture $\mathcal{PA}$ is denoted by $\tau_{\mathcal{PA}}$, and contains a finite sequence of events defined in Figure 7, happening during a system operation. Below we define the semantics function, denoted by S_T, which defines how a trace $\tau_{\mathcal{PA}}$ changes the global state of an architecture (Figure 8).

S_T makes use of the function S_E, which defines how each event in $\tau_{\mathcal{PA}}$ changes the current global state of $\mathcal{PA}$.

Semantics function

$$S_T : EventTrace \times State \mapsto State$$
$$S_{Ev} : Event \times State \mapsto State$$

Definition 2 (The semantics of architectures) *The semantics of an architecture $\mathcal{PA}$ is defined as a set of global states that can be reached from the initial global state :*

$$\{\mu \in State \,|\, \exists \; \tau_{\mathcal{PA}}, S_T(\tau_{\mathcal{PA}}, \mu^{init}) = \mu\}.$$

$S_T(emptytrace,\, \mu) = \mu$ $\qquad$ $S_T(event.\tau_{\mathcal{PA}},\, \mu) = S_T(\tau_{\mathcal{PA}},\, S_{Ev}(event,\, \mu))$

$S_{Ev}(own(E,\, X_\theta{:}V_\theta,\, t),\, \mu) = \mu[\mu_E/\mu_E[X_\theta/V_\theta], TT/t]$

$S_{Ev}(calculateat(E,\, X_\theta{:}T,\, t),\, \mu) = \mu\; [\mu_E/\mu_E[X_\theta/eval(T,\, \mu_E)],\, TT/t]$

$S_{Ev}(createat(E,\, X_\theta{:}T,\, t),\, \mu) = \mu\; [\mu_E/\mu_E[X_\theta/eval(T,\, \mu_E)],\, TT/t]$

$S_{Ev}(receiveat(E,\, Data{:}V_{TYPE(Data)},\, t),\, \mu) = \mu[\mu_E/\mu_E[Data/V_{TYPE(Data)}],\; TT/t]$

$S_{Ev}(receiveat(E,\, \boldsymbol{Cconsent}(Data){:}V_{cconsent},\, t),\, \mu) = \mu[\mu_E[\boldsymbol{Cconsent}(Data)/V_{cconsent}], TT/t]$

$S_{Ev}(receiveat(E,\, \boldsymbol{Uconsent}(Data){:}V_{uconsent},\, t),\, \mu) = \mu[\mu_E[\boldsymbol{Uconsent}(Data)/V_{uconsent}], TT/t]$

$S_{Ev}(receiveat(E,\, \boldsymbol{Sconsent}(Data)/V_{sconsent},\, t),\, \mu) = \mu[\mu_E[\boldsymbol{Sconsent}(Data)/V_{sconsent}], TT/t]$

$S_{Ev}(receiveat(E,\, \boldsymbol{Fwconsent}(Data)/V_{fwconsent},\, t),\, \mu) = \mu[\mu_E[\boldsymbol{Fwconsent}(Data)/V_{fwconsent}], TT/t]$

$S_{Ev}(storeat(E,\, Data{:}V_{TYPE(Data)},\, t),\, \mu) = \mu\; [\mu_E/\mu_E[X_\theta/V_\theta],\, TT/t]$

$S_{Ev}(deletewithin(E,\, Data{:}V_{TYPE(Data)},\, dd,\, t),\, \mu)$
$= \mu\; [\mu_E/\mu_E[X_\theta/\bot,\, \boldsymbol{Cconsent}(Data)/\bot,\, \boldsymbol{Uconsent}(Data)/\bot,\, \boldsymbol{Sconsent}(Data)/\bot,$
$\boldsymbol{Fwconsent}(Data)/\bot],\, TT/t)].$

Figure 8: The semantics of architectural events.

Each event can either change the global state or leave it unchanged. To capture the modification made by an event at time t on (only) the variable state of an entity E we write $\mu[\mu_E/\mu_E[X_\theta/V_\theta], TT/t]$ (or $\mu[\mu_E/\mu_E[X_\theta/\bot], TT/t]$ in case of the undefined value, e.g., when a variable has been deleted). Intuitively, this denotation captures that the old state μ_E is replaced with the new state $\mu_E[X_\theta/V_\theta]$ ($\mu_E[X_\theta/\bot]$), in which the variable X_θ has been given the value V_θ (or the undefined value $\bot$) as a result of the event, the time variable TT is given the value t. $eval(T,\, \mu_E)$ is a function that evaluates the variables in T with μ_E.

4.4 Well-formed Architectures

An architecture $\mathcal{PA}$ is well-formed if:

- Whenever $STORE(E, Data) \in \mathcal{PA}$ or $STOREAT(E, Data, \text{Time(TT)}) \in \mathcal{PA}$, we have
  - $OWN(E, \theta) \in \mathcal{PA}$, $RECEIVE(E, Data) \in \mathcal{PA}$ or $RECEIVEAT(E, Data, \textbf{Time}(\text{TT})) \in \mathcal{PA}$, or
  - $CREATE(E, Data) \in \mathcal{PA}$ or $CREATEAT(E, Data, \textbf{Time}(\text{TT})) \in \mathcal{PA}$, or
  - $CALCULATE(E, Data) \in \mathcal{PA}$ or $CALCULATEAT(E, Data, \textbf{Time}(\text{TT})) \in \mathcal{PA}$.

  In case of *OWN*, *CREATE/AT* or *CALCULATE/AT*, instead of *Data*, there can be any function of θ in *Data* (where $Data = (\theta, E_{from})$), except for *Cconsent*, *UConsent*, *Sconsent*, or *Fwconsent*. In case of *RECEIVE/AT*, instead of *Data*, there can be any function of θ except for cryptographic functions and the "consent functions".

- Whenever $DELETEWITHIN(E, Data, \textbf{Time}(\text{dd})) \in \mathcal{PA}$, we have
  - $STORE(E, Data) \in \mathcal{PA}$ or $STOREAT(E, Data, \text{Time(TT)}) \in \mathcal{PA}$.

5 The Conformance Between Policies and Architectures

We propose three types of conformance relation: (i) privacy conformance, (ii) conformance with regards to data protection properties (which we refer to as DPR conformance in this paper), and (iii) functional conformance. Privacy conformance compares a policy and an architecture based on the privacy properties. Specifically, if we do not give an entity the right to have or link certain types of data, then in the architecture this entity cannot have or link those types of data.

Definition 3 *(Proposed privacy conformance definition)*

1. *If in a policy π_θ an entity E does not have the right to have any data of type θ, then E cannot have this type of data in the corresponding architecture.*
2. *If in a policy π_θ an entity E does not have the right to link two types of data, θ_1 and θ_2, then E cannot link these types of data in the corresponding architecture.*

The DPR conformance relation deals with the data protection requirements (specified in the sub-policies), such as appropriate consent collection, satisfaction of the defined deletion/retention delay, appropriate storage and transfer of a given type of data.

Definition 4 *(Proposed DPR conformance definition):*

1. *If in a policy π_θ, the collection of a (collection, usage, storage, or transfer) consent is required for a piece of data of a given type, then in the architecture the reception of a consent can happen before or at the same time with the reception of the data itself.*
2. *If in an architecture there is an action **act** (createat or calculateat) defined on a data type θ, then in the policy π_θ, there is a (collection, usage, storage, or transfer) purpose **act**:θ' defined for the type θ (besides some θ').*
3. *If in an architecture a piece of data of type θ can be stored in some storage place, **strplace**, then in the policy π_θ, **strplace** $\in \pi_{str}$.**where** (see Table 1 for notations).*
4. *If in the policy π_θ, **delplace** $\in \pi_{del}$.**fromwhere**, then in the corresponding architecture the same data type can be deleted from the place **delplace**.*
5. *If in an architecture, a piece of data of type θ can be deleted within a delay **dd** (from collection), then in the corresponding policy π_θ, **dd** $\leq \pi_{del}$.**deld**. In other words, the retention delay defined in the policy must be respected in the architecture.*

6. *If in an architecture, a piece of data of type θ can be transferred to an entity E, then in the policy π_θ, $E \in \pi_{fw}.\textbf{\textit{fwto}}$ (again, see Table 1 for notations).*

Finally, functional conformance compares a policy and an architecture from the perspective of functionality or effectiveness. This conformance can help a system designer to find an appropriate trade-off between functionality and privacy as in real life, a system is expected to be able to provide certain services.

Definition 5 *(Proposed functional conformance definition)*

1. *If in a policy π_θ, an entity E has the right to have a type of data, θ, then E can have this type of data in the corresponding architecture.*
2. *If in a policy π_θ, an entity E has the right to link two types of data, θ_1 and θ_2, then E can link these types of data in the corresponding architecture.*
3. *If in a policy π_θ, the collection of a (collection, usage, storage, or transfer) consent is **not** required, then **no** corresponding consent can be received in the corresponding architecture.*
4. *If in a policy π_θ, there is a (collection, usage, storage, or transfer) purpose $\textbf{\textit{act}}{:}\theta'$ defined, then in the corresponding architecture there is an action **act** defined on a data type θ (besides some θ').*
5. *If in a policy π_θ, ($\textbf{\textit{strplace}} \in \pi_{str}.\textbf{\textit{where}}$) for some storage place **strplace**, then in the corresponding architecture this type of data can be stored in **strplace**.*
6. *If in an architecture a piece of data of type θ can be deleted from a storage place, **delplace**, then in the corresponding policy π_θ, we have ($delplace = \pi_{del}.\textbf{\textit{fromwhere}}$).*
7. *If in the policy π_θ, $E \in \pi_{fw}.\textbf{\textit{fwto}}$, then in the corresponding architecture, the same type of data can be transferred to the same entity E.*

6 The proposed automated verification engine

The verification engine is based on logic and resolution based proofs. Below, we define the inference rules that will be used in the proof process in Algorithm 1. See Table 4 for the notations used in this section.

Definition 6 *An inference rule R is denoted by $R = H \vdash T_1, \ldots, T_n$, where H is the head of the rule and $T_1, \ldots, T_n$ is the tail of the rule. Each element T_i of the tail is called a fact (or condition), and a head is called a "consequence". The rule R reads as "if $T_1, \ldots, T_n$, then H".*

Figure 9 presents the proposed rules used in the verification of the DPR conformance relations. For instance:

- *D1* specifies that if an entity EV can receive a transfer consent on *Data*, *Data* $=(\theta V, EV_{from})$, to EV_{to} at some non-specific time TV, and EV_{to} can receive this at the same time (or later[4]), then we say that EV can collect the transfer consent on θV to EV_{to}.
- Rule *D2* is defined for data collection consent, rules *D3-D4* are for usage consent collection, and *D5* is for the storage consent.
- Rules *D6* and *D7* are the corresponding version of *D1* and *D2*, respectively, where θV is inside another compound data type[5].

[4]This is modelled in an abstract way by using the same non-specific time value TV.

[5]For example, Anytypeinccrypto[θV] can be *Sicknessrec*(θV,...), *Sicknessrec*(Anytypeinccrypto1[θV],...), **Senc**(θV, K), or **Senc**(Anytypeinccrypto1[θV], K), etc

An inference rule	$H \vdash T_1, \ldots, T_n$.
The head of a rule	H (in $H \vdash T_1, \ldots, T_n$).
The tail of a rule	$T_1, \ldots, T_n$ (in $H \vdash T_1, \ldots, T_n$). A T_i is called as a (sub-)goal in a proof.
A fact	Any of $H, T_1, \ldots, T_n$.
A predicate	Each fact has the form of PREDICATE($Argument_1, \ldots, Argument_m$).
θV	A variable that can be mapped to a data type θ in the policy/architecture.
EV	A variable that can be mapped to an entity E in the policy/architecture.
DD	A variable that can be mapped to a deletion delay dd in the policy/arch.
TV	A variable that can be mapped to a non-specific time value TT in the arch.
TT, dd	A non-specific time value (TT), and a numerical time value (dd).
K, PK	The variables that can be mapped to a type of symmetric and public key.
ds	A value that specifies a real identity of a living individual.
P(ds)	The pseudonym of the real identity ds.
GoalsToBeProved	The set of goals (sub-goals) to be proved.
initgoal	A goal to be proved, which is generated from/captures a sub-policy.
nextgoal	The (next) goal in the set *GoalsToBeProved* to be proved.
previousgoal	The goal in *GoalsToBeProved* was proved right before *nextgoal*.
$\mathbb{AG}$	A set of all possible *initgoal*s (covers all the seven sub-policies).
C/U/FwPurpSet	A set of facts that capture the collection/usage/transfer purposes, respectively.
UniqueTypes	A set of facts that capture the unique data types (UNIQUE(θ)).
TrivialHASLINKFacts	A set of "trivial" HAS, LINK, LINKUNIQUE facts generated from architectural actions (See Figures 14-15).
Anytype(arg1,..., argn)	A piece of (compound) data Anytype that has n arguments (Anytype $\notin \{Senc, Aenc, Mac, Hash\}$).
Anytype[θV]	A piece of data Anytype that contains a piece of data of type θV. (Anytype $\notin \{Senc, Aenc, Mac, Hash\}$). θV may be an argument of another compound data inside Anytype, and so on.
Anytypeinccrypto[θV]	Similar to above, but Anytypeinccrypto can also be *Senc*, *Aenc*, *Mac*, or *Hash*. θV may be an argument of another compound data inside Anytypeincrypto, and so on.
σ	A unifier or mapping, e.g. $\sigma = \{EV \mapsto E, \theta V \mapsto \theta, DD \mapsto dd, TV \mapsto TT\}$, where E is an entity value (e.g. client), θ is a type value (e.g. name), dd (e.g. 6 years).
$T\sigma$	Apply the mapping σ to the variables in T.
nextgoalσ	Apply the mapping σ to the variables in *nextgoal*.
Data	(θV, EV_{from}), θV is a type of a piece of data, EV_{from} is who originally sent this data.
isSuccessful[(rule, goal)]	A dictionary used in e.g. the Python language, with (rule, goal) as the key.

Table 4: The notations used in the automated verification engine.

Figure 10 shows the proposed rules used in the verification of the privacy conformance relation (i.e. a HAS/HASUPTO data possession property). For instance:

- Rule *P1* says that if an entity EV can store $Data$, $Data = (\theta V, EV_{from})$, and can delete $Data$ within a time delay DD, then the entity can have this data[6] up to DD time.
- Rule *P2* says that if a trusted authority/organisation has any data that contains a pseudonym ($\mathbf{P}(ds)$), alongside some other data, then the trusted authority can also have the same data that contains the "real" identity ds.
- *P3* says that if EV can own a type of data (regardless of time), then it can have this type of data.
- Rule *P4* says that if EV can receive $Data$ at some non-specific time TV, then it can have this data. The rest rules can be interpreted in a similar way.
- Finally, rules *P8-P10* capture the decryption of the cryptographic data types. *P8* says that if EV can have an encryption of $Data$ using a symmetric key K, and it can also have K, then it can have $Data$. Similarly, *P9-P10* deal with the decryption of a message authentication code, and the asymmetric decryption process, respectively.

D1. FWCONSENTCOLLECTED(EV, θV, EV_{to}) $\vdash$
RECEIVEAT(EV, **Fwconsent**($Data$, EV_{to}),**Time**(TV)), RECEIVEAT(EV_{to}, $Data$, **Time**(TV))

D2. CCONSENTCOLLECTED(EV, θV) $\vdash$
RECEIVEAT(EV, **Cconsent**($Data$), **Time**(TV)), RECEIVEAT(EV, $Data$, **Time**(TV))

D3. UCONSENTCOLLECTED(EV, θV) $\vdash$
RECEIVEAT(EV, **Uconsent**($Data$),**Time**(TV)), CREATEAT(EV, Anytype[θV], EV_{from}, **Time**(TV))

D4. UCONSENTCOLLECTED(EV, θV) $\vdash$
RECEIVEAT(EV, **Uconsent**($Data$), **Time**(TV)), CALCULATEAT(EV, Anytype[θV], EV_{from}, **Time**(TV))

D5. STRCONSENTCOLLECTED(EV, θV) $\vdash$
RECEIVEAT(EV, **Sconsent**($Data$), **Time**(TV)), STOREAT(EV, $Data$, **Time**(TV))

D6. FWCONSENTCOLLECTED(EV, θV, EV_{to}) $\vdash$
RECEIVEAT(EV, **Fwconsent**(θV, EV_{from}, EV_{to}),**Time**(TV)),
RECEIVEAT(EV_{to}, Anytypeinccrypto[θV], EV_{from}, **Time**(TV))

D7. CCONSENTCOLLECTED(EV, θV) $\vdash$
RECEIVEAT(EV, **Cconsent**(θV, EV_{from}), **Time**(TV)),
RECEIVEAT(EV, Anytypeinccrypto[θV], EV_{from}, **Time**(TV))

Where Data = (θV, EV_{from}) (θV represents a data type, and EV_{from}, an entity that originally sent this data).

Figure 9: The proposed inference rules for DPR conformance check. The predicates and arguments of the heads and tails in the rules are in line with the architecture syntax in Figure 5.

Figure 11 includes the proposed rules used in the verification of the privacy conformance relation (for the LINK property). For instance:

- Rule *L0* says that if an entity (specified by the variable) EV can have two pieces of data of types θV_1 and θV_2, inside any compound data types with the same metadata, then this entity can link θV_1 and θV_2.

[6]More precisely, it can have the corresponding type of data (θV) in $Data = (\theta V, EV_{from})$.

P1. HASUPTO(EV, θV, **Time**(DD)) $\vdash$
STOREAT(EV, $Data$, **Time**(TV)), DELETEWITHIN(EV, $Data$, **Time**(DD))

P2. HAS(**trusted**, Anytype(ds, θV)) $\vdash$ HAS(**trusted**, Anytype(θV, **P**(ds))),
where Anytype is not a crypto function (Anytype $\notin$ {$Senc$, $Aenc$, Mac, $Hash$}).

P3. HAS(EV, θV) $\vdash$ OWN(EV, θV)

P4. HAS(EV, θV) $\vdash$ RECEIVEAT(EV, $Data$, **Time**(TV))

P5. HAS(EV, θV) $\vdash$ STOREAT(EV, $Data$, **Time**(TV))

P6. HAS(EV, θV) $\vdash$ CREATEAT(EV, θV, **Time**(TV))

P7. HAS(EV, θV) $\vdash$ CALCULATEAT(EV, θV, **Time**(TV))

P8. HAS(EV, θV) $\vdash$ HAS(EV, **Senc**(θV, K)), HAS(EV, K)

P9. HAS(EV, θV) $\vdash$ HAS(EV, **Mac**(θV,K)), HAS(EV, K)

P10. HAS(EV, θV) $\vdash$ HAS(EV, **Aenc**(θV, PK)), HAS(EV, **Sk**(PK))

P11. HASUPTO(EV, θV,**Time**(DD)) $\vdash$
STORE(EV, Data), DELETEWITHIN(EV, Data, **Time**(DD))

P12. HAS(**trusted**, Anytype(ds, θV)) $\vdash$ HAS(**trusted**, Anytype(**P**(ds), θV))

P13. HAS(**trusted**, Anytype(θV, ds)) $\vdash$ HAS(**trusted**, Anytype(θV, **P**(ds)))

P14. HAS(**trusted**, Anytype(θV, ds)) $\vdash$ HAS(**trusted**, Anytype(**P**(ds), θV))

P15. HAS(EV, θV) $\vdash$ RECEIVE(EV, Data)

P16. HAS(EV, θV) $\vdash$ STORE(EV, Data)

P17. HAS(EV, θV) $\vdash$ CREATE(EV, θV)

P18. HAS(EV, θV) $\vdash$ CALCULATE(EV, θV). [a]

[a] No rule is defined for the trivial HAS, LINK, LINKUNIQUE properties (e.g. if *sp* can receive *Bill*(*name*, *address*), then it can have *name*, *address*, and can link them, but the facts HAS(sp, name),. . . , LINK(sp, name, address), LINKUNIQUE(sp, name, address) are generated directly from the architectural actions/facts). See Figure 14 for details on how these facts are generated.

Figure 10: Inference rules for privacy conformance check (HAS and HASUPTO property). *P8-P10* capture the cryptographic verification/decryption process, i.e. the destructor application defined in Figure 4.

L0. LINK(EV, θV_1, θV_2) $\vdash$
HAS(EV, Anytype1(θV_1, θV, **Meta**(θV_3))), HAS(EV, Anytype2(θV_2, $\theta V'$, **Meta**(θV_3)))

L1. LINK(EV, θV_1, θV_2) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_2, $\theta V'$, θV_3)).

L2. LINK(EV, θV_1, θV_2) $\vdash$
HAS(EV,Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_3, $\theta V'$, θV_2))

L3. LINK(EV, θV_2, θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_2, $\theta V'$, θV_3))

L4. LINK(EV, θV_2, θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_3, $\theta V'$, θV_2))

L5-L8 are similar to L1-L4, respectively, but with HAS(EV, Anytype1(θV_3, θV, θV_1))
instead of HAS(EV, Anytype1(θV_1, θV, θV_3)) to capture the different order of the data types.

U1. LINKUNIQUE(EV, θV_1, θV_2) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_2, $\theta V'$, θV_3)), UNIQUE(θV_3)
where Anytype1 and Anytype2 are not crypto functions.

U2. LINKUNIQUE(EV, θV_1, θV_2) $\vdash$
HAS(EV, Anytype1(θV_1,θV,θV_3)), HAS(EV, Anytype2(θV_3,$\theta V'$,θV_2)), UNIQUE(θV_3)

U3. LINKUNIQUE(EV, θV_2,θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_2, $\theta V'$,θV_3)), UNIQUE(θV_3)

U4. LINKUNIQUE(EV, θV_2, θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_2, $\theta V'$, θV_3)), UNIQUE(θV_3)

U5-U8 are similar to U1-U4, respectively, but with HAS(EV, Anytype1(θV_3, θV, θV_1))
instead of HAS(EV, Anytype1(θV_1, θV, θV_3)) to capture the different order of the data types.

Figure 11: Inference rules for the privacy conformance check (basic linkability and unique linkability).

C1. CRYPTHAS(EV, θV) $\vdash$ HAS(EV, **Senc**(θV, K)), HAS(EV, K)

C2. CRYPTHAS(EV, θV) $\vdash$ HAS(EV, **Mac**(θV,K)), HAS(EV, K)

C3. CRYPTHAS(EV, θV) $\vdash$ HAS(EV, **Aenc**(θV, PK)), HAS(EV, **Sk**(PK))

Figure 12: Inference rules for CRYPTHAS check. The three rules *C1-C3* in Figure 12 are similar to *P8-P10*, and we define them with verify whether a piece of data of type θV_1 can be obtained by a decryption step. We intentionally differentiate between CRYPTHAS and HAS to deal with the linkability rules in Figure 13.

L1/b. LINK(EV, θV_1, θV_2) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_2, $\theta V'$, Anytypeincrypto[θV_3])),
CRYPTHAS(EV, θV_3)

L1/c. LINK(EV, θV_1, θV_2) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(Anytypeincrypto[θV_2], $\theta V'$, θV_3)),
CRYPTHAS(EV, θV_2)

L1/d. LINK(EV, θV_1, θV_2) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)),
HAS(EV, Anytype2(Anytypeincrypto1[θV_2], $\theta V'$, Anytypeincrypto2[θV_3])),
CRYPTHAS(EV, θV_2), CRYPTHAS(EV, θV_3)

L2/b. LINK(EV, θV_1, θV_2) $\vdash$
HAS(EV,Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_3, $\theta V'$, Anytypeincrypto[θV_2])))
CRYPTHAS(EV, θV_2)

L2/c. LINK(EV, θV_1, θV_2) $\vdash$
HAS(EV,Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(Anytypeincrypto[θV_3], $\theta V'$, θV_2)))
CRYPTHAS(EV, θV_3)

L2/d. LINK(EV, θV_1, θV_2) $\vdash$
HAS(EV,Anytype1(θV_1, θV, θV_3)),
HAS(EV, Anytype2(Anytypeincrypto1[θV_3], $\theta V'$, Anytypeincrypto2[θV_2])))
CRYPTHAS(EV, θV_3), CRYPTHAS(EV, θV_2)

L3/b. LINK(EV, θV_2, θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_2, $\theta V'$, Anytypeincrypto[θV_3])),
CRYPTHAS(EV, θV_3)

L3/c. LINK(EV, θV_2, θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(Anytypeincrypto[θV_2], $\theta V'$, θV_3))
CRYPTHAS(EV, θV_3)

L3/d. LINK(EV, θV_2, θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)),
HAS(EV, Anytype2(Anytypeincrypto1[θV_2], $\theta V'$, Anytypeincrypto2[θV_3])),
CRYPTHAS(EV, θV_3), CRYPTHAS(EV, θV_2)

L4/b. LINK(EV, θV_2, θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(θV_3, $\theta V'$, Anytypeincrypto[θV_2])),
CRYPTHAS(EV, θV_2)

L4/c. LINK(EV, θV_2, θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)), HAS(EV, Anytype2(Anytypeincrypto[θV_3], $\theta V'$, θV_2)),
CRYPTHAS(EV, θV_3)

L4/d. LINK(EV, θV_2, θV_1) $\vdash$
HAS(EV, Anytype1(θV_1, θV, θV_3)),
HAS(EV, Anytype2(Anytypeincrypto1[θV_3], $\theta V'$, Anytypeincrypto2[θV_2])),
CRYPTHAS(EV, θV_3), CRYPTHAS(EV, θV_2).

For each rule in **L5-L8** and **U1-U8** we define a corresponding three rules (/b, /c, /d) with Anytypeincrypto[θV_2] and Anytypeincrypto1[θV_3] in the same manner, respectively.

Figure 13: Inference rules for linkability and unique linkability (part 2). These rules are the extension to the basic rules in Figure 11, in order to capture the case of compound data types for the completeness property. E.g. Anytypeincrypto[θV_3] represents a piece of compound data type (which may be a crypto function) that contains a type θV_3 (note Anytypeincrypto[θV_3] means that θV_3 can be an argument of another compound data inside Anytypeincrypto, and so on).

- Rule *L1* says that if the entity EV can have any data that contains two pieces of data of types θV_1, and θV_3 besides any other data (denoted by θV and $\theta V'$), and any data that contains two pieces of data of types θV_2 and θV_3, then EV can link θV_1 and θV_2. Note that this is not "unique" linkability, meaning that EV cannot be sure that the data of types θV_1 and θV_2 belong to the same individual (although it can narrow down the set of possible individuals to some extent).
- Extending *L1*, rule *U1* also says that if a type θV_3 is unique (e.g. public IP addresses), then EV can "uniquely" link the data of types θV_1 and θV_2, namely, it can also be sure that they belong to the same individual.

In Figure 13, for example, rule L1/b says that if an entity EV can have θV_1 inside a compound data type (Anytype1), and θV_2 in some compound data type, then it can link θV_1 and θV_2. The main difference between L1/b and L1 is that in the first case, θV_3 is also inside a compound data type (which can be a type of a cryptographic function). We use the fact CRYPTHAS(EV, θV_3) to capture that EV can have θV_3 by decrypting the cryptographic function that contains θV_3[7].

The forward search strategy: In order to speed up the verification process and avoid an infinite loop of resolution steps during a proof, let us consider the "trivial" HAS, LINK and LINKUNIQUE properties, namely, if an entity EV can have Anytype(θV_1, ..., θV_n) for some n, then:

- EV can have each of θV_1, ..., θV_n.
- EV can link and uniquely link all the possible pairs among θV_1, ..., θV_n.

For these "trivial" properties, instead of defining the inference rules

- HAS(EV, θV_1) $\vdash$ HAS(EV, Anytype(θV_1, ..., θV_n)), ...,
- LINK(EV, θV_1, θV_2) $\vdash$ HAS(EV, Anytype(θV_1, ..., θV_n)), ...,
- LINKUNIQUE(EV, θV_{n-1}, θV_n) $\vdash$ HAS(EV, Anytype(θV_1, ..., θV_n)),

we generate the HAS, LINK and LINKUNIQUE facts directly from the architectural actions.

Formally, the rule for generating the "trivial" HAS, LINK and LINKUNIQUE facts is given in Figure 14.

If a resulted HAS fact from point 1 in Figure 14 still contains a compound data type (e.g. in HAS(EV, θV_1), θV_1 = Anytype$'$($\theta V_1'$, ..., $\theta V_n'$)), then we recursively generate the "trivial" HAS, LINK and LINKUNIQUE facts from it.

Facts generation example: If RECEIVE(*sp*, *Sicknessrec*(*Personalinfo*(name,address),disease)[8] $\in \mathcal{PA}$ (the same is valid for RECEIVEAT), then the following set of HAS, LINK and LINKUNIQUE facts will be generated:

TrivialHASLINKFacts = {HAS(*sp*, *Personalinfo*(name,address)), HAS(*sp*, disease), HAS(*sp*, name), HAS(*sp*, address), LINK(*sp*, *Personalinfo*(name, address), disease), LINK(*sp*, name, address), LINK(*sp*, name, disease), LINK(*sp*, address, disease)}

6.1 Proposed Automated Conformance Check Algorithm

The automated conformance verification is based on the execution of *resolution* steps and backward search. Resolution is well-known in logic programming and is widely supported in logic programming languages. The formal definition of resolution is based on the so-called substitution and unification steps. A substitution binds some value to some variable, and we denote it by σ in this paper.

[7]Namely, CRYPTHAS(EV, θV_3) is defined to deal with the case when θV_3 is inside a cryptographic function in the second HAS fact.

[8]We intentionally do not use any space character between the arguments as this is the way how we need to specify an action in the software tool

The generation of trivial HAS, LINK, LINKUNIQUE facts (part 1):

From RECEIVE(EV, Anytype($\theta V_1, \ldots, \theta V_n$)) or RECEIVEAT($EV$, Anytype($\theta V_1, \ldots, \theta V_n$), **Time**($TV$)), where Anytype $\notin$ {*Senc*, *Mac*, *Aenc*, *Hash*}, the following facts are generated and put into the set called *TrivialHASLINKFacts* for use during the verification (see point 1 of Algorithm 1):

1. HAS(EV, θV_1), ..., HAS(EV, θV_n),
2. LINK(EV, θV_1, θV_2), ..., LINK(EV, θV_{n-1}, θV_n),
3. LINKUNIQUE(EV, θV_1, θV_2), ..., LINKUNIQUE(EV, θV_{n-1}, θV_n).
4. If θV_1 = Anytype1($\theta V_1'$, ..., $\theta V_n'$), and θV_2 = Anytype2($\theta V_1''$, ..., $\theta V_m''$), then the LINK and LINKUNIQUE facts are also generated for each $\theta V_1'$, ..., $\theta V_n'$ against each $\theta V_1''$, ..., $\theta V_m''$. Specifically,
   (a) LINK(EV, $\theta V_1'$, $\theta V_1''$), ..., LINK(EV, $\theta V_{n-1}'$, $\theta V_n''$),
   (b) LINKUNIQUE(EV, $\theta V_1'$, $\theta V_1''$), ..., LINKUNIQUE(EV, $\theta V_{n-1}'$, $\theta V_n''$).
5. The same fact generation rule in point 4 is applied for the rest θV_j cases, and recursively on the arguments of θV_j (if any).

The same HAS, LINK, LINKUNIQUE fact generation rule is applied to the action OWN(EV, Anytype(θV_1, ..., θV_n)), CREATE(EV, Anytype(θV_1, ..., θV_n), CALCULATE(EV, Anytype(θV_1, ..., θV_n). If there is an overlap between the data in the RECEIVE(AT), CREATE(AT), CALCULATE(AT) actions, then all the redundancies will be eliminated in the set *TrivialHASLINKFacts*. We do not need to generate facts in case of STORE(AT) as data can only be stored if an entity can receive or own this data.

Figure 14: Generate “trivial" HAS, LINK, LINKUNIQUE facts from the arch. actions.

The (recursive) generation of trivial HAS, LINK, LINKUNIQUE facts (part 2):

From HAS(EV, Anytype$'$($\theta V_1'$, ..., $\theta V_n'$)), where Anytype$'$ $\notin$ {*Senc*, *Mac*, *Aenc*, *Hash*}, the following facts are generated and added into the set *TrivialHASLINKFacts* for use during the verification (see point 1 of Algorithm 1):

1. HAS(EV, $\theta V_1'$), ..., HAS(EV, $\theta V_n'$),
2. LINK(EV, $\theta V_1'$, $\theta V_2'$), ..., LINK(EV, $\theta V_{n-1}'$, $\theta V_n'$),
3. LINKUNIQUE(EV, $\theta V_1'$, $\theta V_2'$), ..., LINKUNIQUE(EV, $\theta V_{n-1}'$, $\theta V_n'$).
4. If $\theta V_1'$ = Anytype1(θV_1^1, ..., θV_n^1), and $\theta V_2'$ = Anytype2(θV_1^2, ..., θV_m^2), then the LINK and LINKUNIQUE facts are also generated for each θV_1^1, ..., θV_n^1 against each θV_1^2, ..., θV_m^2. Specifically,
   (a) LINK(EV, θV_1^1, θV_1^2), ..., LINK(EV, θV_{n-1}^1, θV_n^2),
   (b) LINKUNIQUE(EV, θV_1^1, θV_1^2), ..., LINKUNIQUE(EV, θV_{n-1}^1, θV_n^2).
5. The same fact generation rule in point 4 is applied for the rest θV_j cases, and recursively on the arguments of θV_j (if any).

Figure 15: Generate HAS, LINK, LINKUNIQUE facts from a HAS fact).

Definition 7 *A substitution σ is the most general unifier of a set of facts $\mathbb{F}$ if it unifies $\mathbb{F}$,and for any unifier μ of $\mathbb{F}$, there is a unifier λ such that $\mu = \lambda\sigma$.*

Definition 8 *Given a goal (fact) F, and a rule $R = H \vdash T_1,\ldots, T_n$, where F is unifiable with H with the most general unifier σ, then the resolution $F \circ_{(F,H)} R$ results in $T_1\sigma, \ldots, T_n\sigma$.*

Definition 9 *The function that generates initial (verification) goals is defined as:*

$$\mathbb{G} : Policy_{DataType^{sp}_{pol}} \rightarrow \{ColG \cup UseG \cup StoreG \cup DelG \cup TransfG \cup HasG \cup LinkG\}.$$

$\mathbb{G}$ expects a policy as input and returns a set of seven subsets of goals to be proved in a conformance check. Each subset contains the goals capturing each sub-policy in Section 3.1. For a data type θ, we have:

$$\mathbb{G}(\pi_\theta) = \{\mathcal{G}^\theta_{col} \cup \mathcal{G}^\theta_{use} \cup \mathcal{G}^\theta_{str} \cup \mathcal{G}^\theta_{del} \cup \mathcal{G}^\theta_{fw} \cup \mathcal{G}^\theta_{has} \cup \mathcal{G}^\theta_{link}\}$$

The goals generation rules: In the following, we provide the rules for goals generation based on the specific values of the sub-policies inside π_θ, namely, (π_{col}, π_{use}, π_{str}, π_{del}, π_{fw}, π_{has}, π_{link}):

1. For $\pi_{\mathbf{col}}$ with the collection purpose values $\{cp_1{:}\theta'_1,\ldots, cp_n{:}\theta'_n\}$, the following verification goals are generated:

   $\mathcal{G}^\theta_{\mathbf{col}} = \mathcal{G}^\theta_{\mathbf{ccons}} \cup \mathcal{G}^\theta_{\mathbf{cpurp}}$, where $\mathcal{G}^\theta_{\mathbf{ccons}} = \{\text{CCONSENTCOLLECTED(sp, } \theta)\}$,
   $\mathcal{G}^\theta_{\mathbf{cpurp}} = \{\text{CPURPOSE}(\theta'_1, cp_1), \ldots, \text{CPURPOSE}(\theta'_n, cp_n)\}$.

   If $cons = N$, then CCONSENTCOLLECTED(sp, θ) $\notin \mathcal{G}^\theta_{\mathbf{use}}$.

2. For $\pi_{\mathbf{use}}$ with the usage purpose values $\{up_1{:}\theta'_1,\ldots, up_n{:}\theta'_n\}$, the following verification goals are generated:

   $\mathcal{G}^\theta_{\mathbf{use}} = \mathcal{G}^\theta_{\mathbf{ucons}} \cup \mathcal{G}^\theta_{\mathbf{upurp}}$, where $\mathcal{G}^\theta_{\mathbf{ucons}} = \{\text{UCONSENTCOLLECTED(sp, } \theta)\}$,
   $\mathcal{G}^\theta_{\mathbf{upurp}} = \{\text{UPURPOSE}(\theta'_1, up_1), \ldots, \text{UPURPOSE}(\theta'_n, up_n)\}$.

   Again, if the first argument of $\pi_{\mathbf{use}}$ is $cons = N$, then UCONSENTCOLLECTED(sp, θ) $\notin \mathcal{G}^\theta_{\mathbf{use}}$.

3. For $\pi_{\mathbf{str}}$ with the storage place values $\{E_1,\ldots, E_n\}$, the next verification goals are generated:

   $\mathcal{G}^\theta_{\mathbf{str}} = \mathcal{G}^\theta_{\mathbf{scons}} \cup \mathcal{G}^\theta_{\mathbf{places}}$, where $\mathcal{G}^\theta_{\mathbf{scons}} = \{\text{STRCONSENTCOLLECTED(sp,}\theta)\}$,
   $\mathcal{G}^\theta_{\mathbf{places}} = \{\text{STORE}(E_1, \theta, EV_{from}), \ldots, \text{STORE}(E_n, \theta, EV_{from}), \ldots, \text{STOREAT}(E_n, \theta, EV_{from}, \mathbf{Time}(TT))\}$.

4. If $\pi_{\mathbf{del}} = (\{E_1,\ldots, E_n\}, dd)$, where $E_1,\ldots, E_n$ are the values of the deletion places, and dd is the value of the deletion delay, then:

   $\mathcal{G}^\theta_{\mathbf{del}} = \mathcal{G}^\theta_{\mathbf{hasupto}} \cup \mathcal{G}^\theta_{\mathbf{within}}$, where
   $\mathcal{G}^\theta_{\mathbf{hasupto}} = \{\text{HASUPTO}(E_1, \theta, \mathbf{Time}(dd)), \ldots, \text{HASUPTO}(E_n, \theta, \mathbf{Time}(dd))\}$,
   $\mathcal{G}^\theta_{\mathbf{within}} = \{\text{DELETEWITHIN}(E_1, \theta, EV_{from}, \mathbf{Time}(dd)), \ldots, \text{DELETEWITHIN}(E_n, \theta, EV_{from}, \mathbf{Time}(dd))\}$.

5. If $\pi_{\mathbf{fw}} = (cons, \{E_1,\ldots, E_n\}, \{fwp_1{:}\theta'_1,\ldots, fwp_m{:}\theta'_m\})$, where $E_1,\ldots, E_n$ are the entities who can receive the transferred data, and $fwp_1,\ldots, fwp_m$ are the transfer purpose values, then:

   $\mathcal{G}^\theta_{\mathbf{fw}} = \mathcal{G}^\theta_{\mathbf{fwcons}} \cup \mathcal{G}^\theta_{\mathbf{fwto}} \cup \mathcal{G}^\theta_{\mathbf{fwpurp}}$, where
   $\mathcal{G}^\theta_{\mathbf{fwto}} = \{\text{RECEIVE}(E_1, \theta, EV_{from}), \text{RECEIVE}(E_n, \theta, EV_{from}), \ldots, \text{RECEIVEAT}(E_n, \theta, EV_{from}, \mathbf{Time}(TT))\}$,
   $\mathcal{G}^\theta_{\mathbf{fwcons}} = \{\text{FWCONSENTCOLLECTED(sp, } \theta, E_1),\ldots, \text{FWCONSENTCOLLECTED(sp, } \theta, E_n)\}$,
   $\mathcal{G}^\theta_{\mathbf{fwpurp}} = \{\text{FWPURPOSE}(\theta'_1, fwp_1), \ldots, \text{FWPURPOSE}(\theta'_n, fwp_m)\}$.

6. For $\pi_{\mathbf{has}}$, if $\{E_1,\ldots, E_n\}$ is the set of all defined entities in an architecture, then:

   $\mathcal{G}^\theta_{\mathbf{has}} = \{\text{HAS}(E_1, \theta),\ldots, \text{HAS}(E_n, \theta)\}$.

7. For $\pi_{\mathbf{link}}$, if $\{E_1,\ldots, E_n\}$ is the set of all defined entities in an architecture, and $\{\theta_1,\ldots, \theta_m\}$ is

a set of all defined data types (different from θ), then :

$\mathcal{G}^{\theta}_{\mathbf{link}} = \{\text{LINK}(E_1, \theta, \theta_1), \text{LINK}(E_1, \theta_1, \theta), \ldots, \text{LINK}(E_n, \theta, \theta_n), \ldots, \text{LINKUNIQUE}(E_n, \theta_m, \theta)\}$.

Finally, let us denote the set of all goals to be proved during a conformance verification by $\mathbb{AG}$, namely:

$$\mathbb{AG} = \bigcup_{\forall \theta \in DataTypes^{sp}_{pol}} \mathbb{G}(\pi_\theta),$$

where $DataTypes^{sp}_{pol}$ is a set of all data types defined in the policy for a service provider *sp*.

The generation of purpose-facts in architectures: Besides the actions defined in Figure 5, to verify the DPR conformance regarding the (collection, usage, or forward) purposes, the so-called purpose-facts are generated. This is based on the following purposes-fact generation rules, for a given architecture $\mathcal{PA}$:

1. If $CREATEAT(E, X_\theta, \mathbf{Time}(TT)) \in \mathcal{PA}$, then CPURPOSE($\theta$, createat) $\in$ *CPurpSet*.
2. If $CALCULATEAT(E, X_\theta, \mathbf{Time}(TT)) \in \mathcal{PA}$, then UPURPOSE($\theta$, calculateat) $\in$ *UPurpSet*.
3. If $RECEIVEAT(E, \mathbf{Fwconsent}(X_\theta, E_{to}), \mathbf{Time}(TT)) \in \mathcal{PA}$, and $CREATEAT(E_{to}, X_\theta, \mathbf{Time}(TT)) \in \mathcal{PA}$, then FWPURPOSE($\theta$, createat) $\in$ *FwPurpSet*.
4. If $RECEIVEAT(E, \mathbf{Fwconsent}(X_\theta, E_{to}), \mathbf{Time}(TT)) \in \mathcal{PA}$, and $CALCULATEAT(E_{to}, X_\theta, \mathbf{Time}(TT)) \in \mathcal{PA}$, then FWPURPOSE($\theta$, calculateat) $\in$ *FwPurpSet*.

These rules define how the facts for the collection (point 1), usage (point 2), and transfer (points 3-4) purposes are generated from the architectural actions, and added into the sets *CPurpSet*, *UPurpSet*, and *FwPurpSet*, respectively, to be used in Algorithm 1.

To speed up the verification process, the actions defined in an architecture are divided into four subsets, specifically, *ArchTime*, *ArchPseudo*, *ArchMeta*, and *Arch*. *ArchTime* includes the actions that contain the **Time**() construct, *ArchPseudo* includes the actions that contain the **P**() construct for pseudonym, *ArchMeta* includes the actions that contain the **Meta**() construct for metadata, and finally, *Arch* is a set of actions without any specific construct above (see rules *P15-P18*).

Finally, if the set of unique data types[9] defined in the policy is $\{\theta_1, \ldots, \theta_n\}$, $\{\theta_1, \ldots, \theta_n\} \subseteq DataType^{sp}_{pol}$, then we have the corresponding set of facts, *UniqueTypes*, which can be used to prove the unique linkability properties (see rule *U1* in Figure 11):

$$UniqueTypes = \{\text{UNIQUE}(\theta_1), \ldots, \text{UNIQUE}(\theta_n)\}.$$

Let us define the following rule sets that we will use in the inference algorithms, namely:

- $DPRRules = \{D1, \ldots, D5, D6, D7\}$,
- $HasUpToRules = \{P1, P2\}$,
- $HasRules = \{P3, \ldots, P18\}$,
- $CryptHasRules = \{C1, C2, C3\}$,
- $LinkRules = \{L0, L1$ (inc. L1/b-L1/d), $\ldots, L8$ (inc. L8/b-L8/d)$\}$, and
- $LinkUniqueRules = \{U1$ (inc. U1/b-U1/d), $\ldots, U8$ (inc. U8/b-U8/d)$\}$.

[9]Unique data types are types that can be used to uniquely identify a living individual, e.g. passport numbers.

Algorithm 1 defines the process of checking whether the input architecture *Architecture* is fulfilling the "initial" goal, *initgoal*, and returns either 1 if the proof is successful, or 0 if failed.

Algorithm 1: ConformanceCheck(initgoal, Architecture, Rulesets, N)

```
/* (* Backward search strategy *)                                          */
/* Note:  If the proof has failed for initgoal for an (original) entity E, the algorithm
   will attempt the proof initgoal in which E is replaced with the entities in the set
   HasAccessTo(E).                                                         */
```

Result: Proof found (1) /Proof not found (0) (* see Table 4 for the used notations *)

Inputs:

1. Rulesets = {*DPRRules*, *HasUpToRules*, *HasRules*, *CryptHasRules*, *LinkRules*, *LinkUniqueRules*, *TrivialHASLINKFacts*}.
2. Architecture = {*ArchTime*, *ArchPseudo*, *ArchMeta*, *Arch*}.
3. ArchPurposes = {*CPurpSet*, *UPurpSet*, *FwPurpSet*}.
4. UniqueTypes.
6. Goal: *initgoal*, where *initgoal* $\in \mathbb{AG}$.
7. Allowed layers of nested crypto functions: N.

if *initgoal* $\in \mathcal{G}^{\theta}{}_{\boldsymbol{places}} \cup \mathcal{G}^{\theta}{}_{\boldsymbol{within}} \cup \mathcal{G}^{\theta}{}_{\boldsymbol{fwto}}$ **then**
  for *arch in Architecture* **do**
    if *(initgoal* $\circ_{(initgoal,arch)}$ *arch) is successful* ***or*** *(initgoal == arch)* **then**
      return 1
    end
    return 0
  end
else
  if *the predicate of initgoal matches the predicate of a purpose-fact in AP, AP* $\in$ *ArchPurposes* **then**
    for *purp in AP* **do**
      if *(initgoal* $\circ_{(initgoal,purp)}$ *purp) is successful* ***or*** *(initgoal == purp)* **then**
        return 1
      end
      return 0
    end
  else
    if ***VerifyAgainstRuleset****(initgoal, Architecture, UniqueTypes, Rulesets, N) == 1* **then**
      return 1
    else
      return 0
    end
  end
end

Algorithm 2: VerifyAgainstRuleset(goal, Architecture, UniqueTypes, Rulesets, N)

if *the predicate of* ***goal*** *matches the predicate of a head of a rule in RS, RS* $\in$ *Rulesets* **then**
  for *rule in RS* **do**
    isSuccessful[(rule, goal)] = **VerifyRule**(*rule, goal, Architecture, UniqueTypes, Rulesets, N*)
  end
  if *for all rule in RS: isSuccessful[(rule, goal)] == 0* **then**
    return 0
  else
    return 1
  end
end

Algorithm 3: VerifyUniqueTypes(*rule, goal, UniqueTypes*)

for *unique in UniqueTypes* **do**
  if *(goal* $\circ_{(goal,\ unique)}$ *unique) is successful* ***or*** *(goal == unique)* **then**
    /* A dictionary entry with the key of (*rule, goal, unique*). The proof of goal with arch was successful. */
    Derivation_Unique_Successful[(rule,goal,unique)] = 1
  else
    /* The proof of goal with unique and rule has failed. */
    Derivation_Unique_Successful[(rule, goal, unique)] = 0
  end
end
if *for all unique in UniqueTypes: Derivation_Unique_Successful[(rule, goal, unique)] == 0* **then**
  return 0
else
  return 1
end

Algorithm 4: VerifyAgainstArch(*rule, goal, AS*)

for *arch in AS* **do**
  if *(goal* $\circ_{(goal,\ arch)}$ *arch) is successful with* σ_{arch} ***or*** *(goal == arch)* **then**
    /* A dictionary entry with the key of (*rule, goal, arch*). The proof of goal with arch was successful. */
    Derivation_Arch_Successful[(rule, goal, arch)] = 1
    /* σ_{arch} is the mapping that proves *goal* with *arch*. Several *arch*s can prove *goal* with different σ_{arch}-s. *Mappings[goal]* contains all σ_{arch}-s that prove goal. */
    add the mapping σ_{arch} into the set *Mappings[goal]*
  else
    /* The proof of goal with arch and rule has failed. */
    Derivation_Arch_Successful[(rule, goal, arch)] = 0
  end
end
if *for all arch in AS: Derivation_Arch_Successful[(rule,goal,arch)] == 0* **then**
  return 0
else
  return 1
end

Algorithm 6 defines a verification process of *initgoal* via the sub-goals resulted from the resolution steps.

Algorithm 5: **CaseNoPreviousGoal**(rule, nextgoal, Architecture, UniqueTypes, Rulesets, N)

```
/* In this case, there is no previousgoal right before nextgoal. Hence, the proof will be
   attempted on nextgoal instead of nextgoalσ (like in Algorithm 6).                    */
if nextgoal is an action, and matches the Time/P/Meta construct in AS, AS ∈ Architecture then
    if VerifyAgainstArch(rule, nextgoal, AS) ==1 then
        /* The proof of nextgoal with rule and the subset AS was successful.           */
        isSuccessful[(rule, nextgoal)] = 1
    else
        /* The proof of nextgoal with rule and the subset AS failed.                   */
        isSuccessful[(rule, nextgoal)] = 0
    end
else
    if the predicate of nextgoal matches a fact in UniqueTypes then
        if VerifyUniqueTypes(rule, nextgoal, UniqueTypes) ==1 then
            isSuccessful[(rule, nextgoal)] = 1
        else
            isSuccessful[(rule, nextgoal)] = 0
        end
    else
        if VerifyAgainstRuleset(nextgoal, Architecture, UniqueTypes, Rulesets, N) == 1 then
            isSuccessful[(rule, nextgoal)] = 1
        else
            isSuccessful[(rule, nextgoal)] = 0
        end
    end
end
```

Algorithm 6: **VerifyRule**(*rule, goal, Architecture, UniqueTypes, Rulesets, N*)

```
/* Note: The variable arguments in the inference rules are renamed before they are used in a
   resolution.                                                                              */
GoalsToBeProved = {goal};
if goal ∘(goal, head of rule) rule is successful then
    /* Check for the limit of nested layers of crypto functions.                           */
    if ∃ fact in (goal ∘(goal, head of rule) rule) that contains more than N nested layers of crypto
     functions and rule ∈ {P8, P9, P10} then
        return 0;
    else
        remove goal from GoalsToBeProved ;
        add the facts in (goal ∘(goal, head of rule) rule) to the start of GoalsToBeProved;
        for nextgoal in GoalsToBeProved do
            /* For all mappings (σ) that can be used to prove previousgoal (Algorithm 4).  */
            if there exists previousgoal examined just before nextgoal in GoalsToBeProved then
                for σ in Mappings[previousgoal] do
                    if nextgoalσ is an action in AS, AS ∈ Architecture then
                        if VerifyAgainstArch(rule, nextgoalσ, AS) ==1 then
                            /* The proof of nextgoal with rule and σ was successful.       */
                            isSuccessfulMapping[(rule, nextgoal, σ)] = 1
                        else
                            /* The proof of nextgoal with rule and σ was unsuccessful.     */
                            isSuccessfulMapping[(rule, nextgoal, σ)] = 0
                        end
                    else
                        if the predicate of nextgoalσ matches a fact in UniqueTypes then
                            if VerifyUniqueTypes(rule, nextgoalσ, UniqueTypes) ==1 then
                                isSuccessfulMapping[(rule, nextgoal, σ)] = 1
                            else
                                isSuccessfulMapping[(rule, nextgoal, σ)] = 0
                            end
                        else
                            if VerifyAgainstRuleset(nextgoalσ, Architecture, UniqueTypes,
                             Rulesets, N) == 1 then
                                isSuccessfulMapping[(rule, nextgoal, σ)] = 1
                            else
                                isSuccessfulMapping[(rule, nextgoal, σ)] = 0
                            end
                        end
                    end
                end
            else
                CaseNoPreviousGoal(rule, nextgoal, Architecture, UniqueTypes, Rulesets, N)
            end
            if for all σ in Mappings[previousgoal]: isSuccessfulMapping[(rule, nextgoal, σ)] == 0
             then
                isSuccessful[(rule, nextgoal)] == 0
            else
                isSuccessful[(rule, nextgoal)] == 1
            end
        end
        if for all nextgoal in GoalsToBeProved: isSuccessful[(rule, nextgoal)] == 1 then
            return 1
        else
            return 0
        end
    end
end
```

Algorithm Explanation. Algorithm 1 expects as input the set of inference rules (*Rulesets*, which also contains a set of "trivial" HAS, LINK, LINKUNIQUE facts generated from the architectural actions (*TrivialHASLINKFacts*)), a set of facts that capture the actions in an architecture (*Architecture*), a set of purposes defined in an architecture (*ArchPurposes*), a set of unique data types (*UniqueTypes*), and a verification goal, *initgoal*. N is a defined number that denotes the maximum layers of nested cryptographic functions in a piece of data that the verification engine examines. A finite N is used to ensure the termination of the proof process.

1. First of all, if *initgoal* $\in \mathcal{G}^{\theta}{}_{\textbf{places}} \cup \mathcal{G}^{\theta}{}_{\textbf{within}} \cup \mathcal{G}^{\theta}{}_{\textbf{fwto}}$ (see points 3-5 in Definition 9), then we check whether *inigoal* can be unified with or equal to a fact in *Architecture*. The algorithm returns 1 if the proof was successful, and 0 otherwise.

2. If *initgoal* is not an action fact, then we check if the (collection, usage, or transfer) purposes in an architecture is in line with the policy, namely, whether *initgoal* is in *ArchPurposes*. The algorithm returns 1 if the proof was successful, and otherwise, 0.

3. If *initgoal* is not a purpose-fact (e.g. *initgoal* = HAS(sp, name)), then we try to prove it using the inference rule set and the given architecture. If a proof or a derivation was found for *initgoal*, then 1 is returned, otherwise, 0.

4. In **VerifyRule**(*rule, goal, Architecture, UniqueTypes, Rulesets, N*), inside algorithm 2, we attempt to carry out resolution steps between *initgoal* and each rule in an appropriate *RS*, *RS* $\in$ *Rulesets*. If the proof has failed for all rules in *RS*, then 0 is returned (proof failed). Otherwise, if at least one rule can be used to prove the goal, then 1 is returned.

5. In algorithm 6, a step *goal* $\circ_{(goal,\ head\ of\ rule)}$ *rule* can be successful or unsuccessful (in case there is no unifier σ for *goal* and the head of *rule*). This step results in the new (sub-)goals to be proved. If there is a new (sub-)goal that contains more than N layers of nested cryptographic functions (*Senc*, *Aenc*, *Mac*, *Hash*), then we return 0, and this "branch" of the proof was unsuccessful[10]. If there is a new (sub-)goal which corresponds to an architectural action, then we attempt to prove it using the facts in *Architecture*.

6. Algorithm 4 specifies a proof attempt using the (action) facts in *Architecture*. If there is no matching action for a goal, then this branch of the proof was unsuccessful. Otherwise, this branch of the proof has been successful.

7. Finally, algorithm 3 checks *goal* against the set *UniqueTypes*. If there is no matching, then this branch of the proof was unsuccessful. Otherwise, this branch of the proof has been successful.

Example 1. Let *Architecture* = {RECEIVEAT(*sp*, *name*, *client*, **Time**(TT))} and *initgoal* = HAS(*sp*, *name*), namely, we want to prove that *sp* can have *name*. This can be proven with rule *P4* in Figure 10 and a resolution step in Definition 8.

- **Step 1**: *initgoal* $\circ_{(initgoal,\ HAS(EV,\theta V))}$ *P4* = RECEIVEAT(*sp*, *name*, *client*, **Time**(TT))), as *initgoal* can be unified with HAS(EV, θV), the head of rule *P4*, with the unifier $\sigma = \{EV \mapsto sp, \theta V \mapsto name, EV_{from} \mapsto client, TV \mapsto TT\}$. We have RECEIVEAT($EV$, θV, E_{from}, **Time**(TV))σ as a result, which is equal to RECEIVEAT(*sp*, *name*, *client*, **Time**(TT))).
- **Step 2**: As RECEIVEAT(*sp*, *name*, *client*, **Time**(TT))) $\in$ *Architecture*, therefore, we get **ConformanceCheck**(*initgoal, Architecture, Rulesets, N*) == 1, for any natural N.

Example 2. Let *Architecture* = {RECEIVEAT(*sp*, **Senc**(*name*,key), *client*, **Time**(TT)), OWN(sp, key)} and *initgoal* = HAS(*sp*, *name*). This can be proven with rules *P8*, then *P3*, *P4* as shown in Figure 16.

[10] A proof can be seen as a derivation tree, with *initgoal* in the root and the facts in *Architecture* are the leaves.

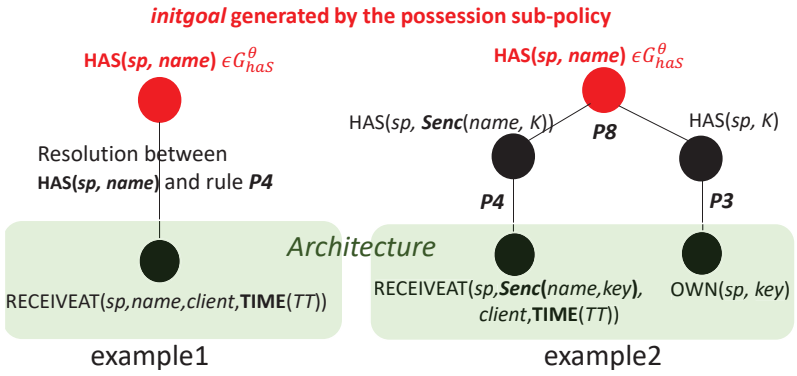

Figure 16: Two example proofs (without and with encryption, respectively).

6.1.1 Properties

Property 1 *(Correctness)*
We distinguish several cases based on the value of initgoal:

1. *If initgoal* $\in$ *{HAS(E, θ), HASUPTO(E, θ,* ***Time****(dd))}, and* $E \in \pi_\theta.\pi_{has}$ *at the policy level, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, Architecture functionally conforms with this requirement of the policy.*

2. *If initgoal* $\in$ *{HAS(E, θ), HASUPTO(E, θ,* ***Time****(dd))}, and* $E \notin \pi_\theta.\pi_{has}$*, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, Architecture does not privacy conform with the policy.*

3. *If initgoal* $\in \mathcal{G}^{\theta}_{link}$ *and* $(E_i, \theta_i) \in \pi_\theta.\pi_{link}$*, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture functionally conforms with this link policy.*

4. *If initgoal* $\in \mathcal{G}^{\theta}_{link}$ *and* $(E_i, \theta_i) \notin \pi_\theta.\pi_{link}$*, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture does not privacy conform with the policy.*

5. *If initgoal* $\in \mathcal{G}^{\theta}_{ccons} \cup \mathcal{G}^{\theta}_{ucons} \cup \mathcal{G}^{\theta}_{scons} \cup \mathcal{G}^{\theta}_{fwcons}$*, and* $\pi_{col}.cons = Y$*,* $\pi_{use}.cons = Y$*,* $\pi_{str}.cons = Y$*, or* $\pi_{fw}.cons = Y$ *in* π_θ*, respectively, then the architecture DPR conforms with the actual sub-policy whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1.*

6. *If initgoal = CPURPOSE(θ, cp) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{cpurp}$*), and (cp:θ* $\in \pi_{use}.cpurp$*), then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture functionally conforms with the policy.*

7. *If initgoal = CPURPOSE(θ, cp) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{cpurp}$*), and (cp:θ* $\notin \pi_{use}.cpurp$*), then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture does not DPR conform with the policy.*

8. *If initgoal = UPURPOSE(θ, up) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{upurp}$*), and (up:θ* $\in \pi_{use}.upurp$*), then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture functionally conforms with the policy.*

9. *If initgoal = UPURPOSE(θ, up) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{upurp}$*), and (up:θ* $\notin \pi_{use}.upurp$*), then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture does not DPR conform with the policy.*

10. *If initgoal = FWPURPOSE(θ, fwp) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{fwpurp}$*), and (fwp:θ* $\in \pi_{fw}.fwpurp$*), then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture functionally conforms with the policy.*

11. *If initgoal = FWPURPOSE(θ, fwp) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{fwpurp}$*), and (fwp:θ* $\notin \pi_{fw}$*.fwpurp), then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture does not DPR conform with the policy.*

12. *If initgoal = {STORE(E, θ,* EV_{from}*), STOREAT(E, θ,* EV_{from}*,* ***Time****(TT))} (i.e. initgoal* $\in \mathcal{G}^{\theta}_{places}$*), and* ***(E*** $\in \pi_{str}$***.where)****, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture functionally conforms with the policy.*

13. *If initgoal = {STORE(E, θ,* EV_{from}*), STOREAT(E, θ,* EV_{from}*,* ***Time****(TT))} (i.e. initgoal* $\in \mathcal{G}^{\theta}_{places}$*), and* ***(E*** $\notin \pi_{str}$***.where)****, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture does not DPR conform with the policy.*

14. *If initgoal = DELETEWITHIN(E, θ,* EV_{from}*,* ***Time****(dd)) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{within}$*) and* ***(dd*** $\leq \pi_{del}$***.deld)*** *and* ***(E*** $\in \pi_{del}$***.fromwhere)****, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture functionally conforms with the policy.*

15. *If initgoal = DELETEWITHIN(E, θ,* EV_{from}*,* ***Time****(dd)) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{within}$*) and* ***(dd*** $\geq \pi_{del}$***.deld)*** *and* ***(E*** $\in \pi_{del}$***.fromwhere)****, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture does not DPR conform with the policy.*

16. *If initgoal = RECEIVE(E, θ,* EV_{from}*) (initgoal* $\in \mathcal{G}^{\theta}_{fwto}$*), and E* $\in \pi_{fw}$*.fwto, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture functionally conforms with the policy.*

17. *If initgoal = RECEIVE(E, θ,* EV_{from}*) (initgoal* $\in \mathcal{G}^{\theta}_{fwto}$*), and E* $\notin \pi_{fw}$*.fwto, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 1, the architecture does not DPR conform with the policy.*

Proof: **ConformanceCheck**(*initgoal, Architecture, Rulesets, N*) == 1 means that a proof of *initgoal* can be found with *Architecture*. Whenever *initgoal* can be proved with a rule *rule* = *H* ⊢ T_1,...,T_n (in Algorithm 6), there is at least one fact in *Architecture* that can be used to prove the sub-goals T_1,..., T_n. Besides, since *Data* includes the entity who originally sent it, i.e. *Data* = (*θV*, EV_{from}), we can avoid that in the rules for consent collections (e.g. D1-D2), the consent contains a different data from the one can be received. In addition, in rules *D1-D7*, *P1*, *P4-P5*, *P11*, and *P15-P16*, *Data* is a pair of a data type and the entity who originally sent it, i.e. *Data* = (*θV*, EV_{from}), which can be differentiated from the other data pairs.

Therefore, in case of points 1 and 3 (of Property 1), the first two points of Definition 5 are satisfied, respectively. In case of points 2 and 4, the two points of Definition 3 are unsatisfied, respectively. In case of point 5, **ConformanceCheck**(*initgoal, Architecture, Rulesets, N*) == 1 means that the first point of Definition 4 is satisfied. Points 6, 8, 10 of Property 1 correspond to the satisfaction of point 4 of Definition 5, while points 7, 9, 11 mean that point 2 in Definition 4 is unsatisfied. Point 12 of Property 1 correspond to the satisfaction of point 5 of Definition 5, while point 13 correspond to (the unsatisfied) point 3 in Definition 4. Point 14 of Property 1 correspond to the satisfaction of point 6 of Definition 5, while point 15 correspond to (the unsatisfied) points 4-5 in Definition 4. Point 16 corresponds to the satisfactory of point 7 of Definition 5. Finally, point 17 corresponds to (the unsatisfied) point 6 in Definition 4. □

Property 2 *(Termination up-to N) Let N be the maximum number of nested layers of cryptographic functions that the verification engine will examine. Assume that the nested layers of the defined data types are finite, beside a finite N, the proof process never gets into an infinite loop.*

Proof: The verification engine performs resolution steps between the goals and the rules in *Rulesets*, as well as the (action) facts in *Architecture*. If there is an infinite loop in the proof process, then we would have an infinite number of resolution steps. We will show that the number of resolution steps is always finite during the proof of *initgoal*.

As a result of a resolution step $goal \circ_{(goal,\ head\ of\ rule)} rule$, where $rule \in \{P8, P9, P10\}$, we get the two new (sub-)goals in the tails of the rules (e.g. $goal \circ P8 = \text{HAS}(EV,\mathbf{Senc}(\theta V,K))\sigma$, $\text{HAS}(EV,K)\sigma$). Since the verification engine does not prove/examine any goal with more than N layers of cryptograpghic functions (e.g. HAS(sp,**Senc**(**Senc**,...(**Mac**(name,key))),...,key),key), there are maximum N recursive calls of the resolution step $goal \circ_{(goal,\ head\ of\ rule)} rule$, beside $rule \in \{P8, P9, P10\}$. Each recursive call produces two (sub-)goals, hence, N recursive calls result in at most 2^N (sub-)goals to be proved. In the worst case scenario, this would mean $2^N*|Rulesets|$ resolution steps (between each goal and rule pair, where $|Rulesets|$ is the number rules in *Rulesets*).

In case *rule* is one of *P3-P7* or *P15-P18*, a resolution step $goal \circ_{(goal,\ head\ of\ rule)} rule$ would generate a single goal (e.g. $goal \circ_{(goal,\ head\ of\ P4)} P4 = \text{RECEIVEAT}(EV, Data, \mathbf{Time}(TT))\sigma$). Then, the resulted (sub-)goals will be checked against the facts in *Architecture*, which yields $|Architecture| + 1$ resolution steps for each rule (where $|Architecture|$ is the number elements in *Architecture*).

In case *rule* is one of *D1-D7* or $rule \in \{P1, P11\}$, $2*|Architecture| + 1$ resolution steps are carried out. For $rule \in \{P2, P12, P13, P14\}$, a step $goal \circ_{(goal,\ head\ of\ rule)} rule$ generates a single (sub-)goal. The (sub-)goals are then be checked against the rule set (*Rulesets*), including rules *P3-P7* (or *P15-P18*), which yields $2*|Architecture| + 1$ resolution steps in each case. In addition, when these (sub-)goals are checked against *P8-P10*, it yields $2^N*|Rulesets|$ resolution steps in each case. We note that in rules *P2* and *P12-P14*, *ds* is a value and *P(ds)* is a function on *ds* that represents the pseudonym. Therefore, we cannot have an infinite number of recursive resolution steps between these rules and the resulted sub-goals, because *ds* cannot be unified with *P(ds)*, and θV cannot be unified with either *ds* or *P(ds)* being of different types.

In case *rule* is one of *L1-L8*, a resolution step $goal \circ_{(goal,\ head\ of\ rule)} rule$ generates two (sub-)goals. Each (sub-)goal will be examined against every rule (in *Rulesets*), but a resolution step can only be successful in case of *P3-P10*. The resolution with each of these rules results in a finite number of further resolution steps (as we argued above). Similarly, the case of *U1-U8* only yields a finite number of resolution steps.

□

The completeness property can be stated as a consequence of the termination property (Property 2), as follows:

Property 3 *(Completeness)*
If all the data types specified in Architecture contain at most N layers of nested cryptographic functions, for some finite N, and all the defined data types contain a finite number of layers of other data types, then:

1. *If initgoal ∈ {HAS(E,θ), HASUPTO(E,θ,Time(dd))}, and $E \in \pi_\theta.\pi_{has}$ at the policy level, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 0, the architecture does not functionally conform with the policy.*

2. *If initgoal $\notin \mathcal{G}^\theta_{link}$ and $(E,\theta') \in \pi_\theta.\pi_{link}$, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 0, Architecture does not functionally conform with the policy.*

3. *If initgoal $\in \mathcal{G}^\theta_{ccons} \cup \mathcal{G}^\theta_{ucons} \cup \mathcal{G}^\theta_{scons} \cup \mathcal{G}^\theta_{fwcons}$, and $\pi_{col}.cons = Y$, $\pi_{use}.cons = Y$, $\pi_{str}.cons = Y$, $\pi_{fw}.cons = Y$ in π_θ, respectively, then the architecture does not DPR conform with the policy whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 0.*

4. *If initgoal = CPURPOSE(θ, cp) (i.e. initgoal $\in \mathcal{G}^\theta_{cpurp}$), and (cp:θ $\in \pi_{use}.cpurp$), then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 0, the architecture* ***does not*** *functionally conform with the policy.*

5. *If initgoal = UPURPOSE(θ, up) (i.e. initgoal $\in \mathcal{G}^\theta_{upurp}$), and (up:θ $\in \pi_{use}.upurp$), then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 0, the architecture* ***does not*** *functionally conform with the policy.*

6. *If initgoal = FWPURPOSE(θ, fwp) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{fwpurp}$*), and (fwp:θ* $\in \pi_{fw}$*.fwpurp), then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 0, the architecture* ***does not*** *functionally conform with the policy.*

7. *If initgoal = {STORE(E, θ,* EV_{from}*), STOREAT(E, θ,* EV_{from}*,* ***Time****(TT))} (i.e. initgoal* $\in \mathcal{G}^{\theta}_{places}$*), and* ***(E*** $\in \pi_{str}$***.where)****, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 0, the architecture* ***does not*** *functionally conform with the policy.*

8. *If initgoal = DELETEWITHIN(E, θ,* EV_{from}*,* ***Time****(dd)) (i.e. initgoal* $\in \mathcal{G}^{\theta}_{within}$*) and* ***(dd*** $\leq \pi_{del}$***.deld)*** *and* ***(E*** $\in \pi_{del}$***.fromwhere)****, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 0, the architecture* ***does not*** *functionally conforms with the policy.*

9. *If initgoal = RECEIVE(E, θ,* EV_{from}*) (initgoal* $\in \mathcal{G}^{\theta}_{fwto}$*), and E* $\in \pi_{fw}$*.fwto, then whenever* ***ConformanceCheck****(initgoal, Architecture, Rulesets, N) == 0, the architecture* ***does not*** *functionally conform with the policy.*

Property 3 says that completeness can only be "achieved" up to the maximum allowed nested layers of cryptographic functions, N.

Proof: If **ConformanceCheck**(*initgoal, Architecture, Rulesets, N*) == 0, then *initgoal* cannot be proved by any fact in *Architecture* provided that all facts in *Architecture* contain at most N nested layers of functions *Senc*, *Aenc*, and *Mac*, and nested layers of other data types. The latter assumption is required for a resolution step to be successful, while the first is required to make the verification terminates. Otherwise, if there is a set of facts in *Architecture*, which can be used to prove *initgoal*, then there would be a derivation tree meaning that **ConformanceCheck**(*initgoal, Architecture, Rulesets, N*) == 1.

Therefore, point 1 of Property 3 does not satisfy the first point of Definition 5. Similarly, point 2 of Property 3 does not satisfy the second point of Definition 5. Point 3 of Property 3 does not satisfy the first point of Definition 4. Points 4-6 of Property 3 correspond to point 4 of Definition 5. Points 7, 8, and 9 of Property 3 correspond to points 5, 6, and 7 of Definition 5, respectively.

Moreover, we show that the inference rules cover all the possible data types (format) may be defined in the architectural actions.

- In case of consents, rules D1-D5 capture the case when an action RECEIVE(AT) is defined on a data type θV and the corresponding consent is also defined on this data type.
- In case of forward and collection consents, the fact Anytypeinccrypto[θV] in rules D6 and D7 covers all data types that contains θV.
- Similarly, in the LINK and LINKUNIQUE cases, the rules in Figures the rules in Figure 13 are defined on data types of form Anytypeinccrypto[θV].
- For the "trivial" HAS, LINK and LINKUNIQUE facts, the set *TrivialHASLINKFacts* contains all the possible generation of data types (as shown in Figures 14-15).

□

7 Discussion

As most of the laws and articles in the GDPR are complex, formally specifying them without simplification is either cumbersome or impossible. In this paper, we attempt to capture some basic requirements in an abstract way. There are several ways to improve or extend the proposed formal sepcifications. For instance, practically, depending on the context of a consent (e.g. healthcare or education contexts), a consent may contain different pieces of information that need to be

modelled. Furthermore, in our languages we do specify the deletion of a consent, but only when the data itself is deleted (see the last rule in Figure 8). A more detailed study of the consent revocation process can be addressed in the future, for example, when the collected data has not been deleted yet, but the consent for transfer has been revoked. This could be addressed by changing the last rule in Figure 8 such that only the consent in question is deleted.

There are areas to improve regarding the transfer sub-policy as well, for example, the GDPR covers the case when personal data is transferred to a third country or an international organisation, and appropriate agreement and arrangement must be done prior data transfer [41]. This agreement could be specified in the form of a sticky policy between a service provider and an international organisation. Sticky policies are used in PPL [17] to match the expectation of a client and the obligation offered by a service provider. Regarding the deletion sub-policy, in the GDPR, the data subject also has the right to request a deletion for their collected data. This can be modelled with an event/action that captures the reception of a deletion request (e.g. *recvdelreq*(θ, *place*, t)) and a corresponding deletion event within a specified delay. Finally, transparency is also an important part of the GDPR as it captures the "right to be informed", which can be defined by the event/action "notify" that happens before the data collection, usage, storage and transfer.

To capture the (collection, usage, or transfer) purposes, for simplicity, the architecture language proposed in this paper relies on only the two basic actions *create* and *calculate*. In the same way, additional actions can be added to specify purposes such as "send some type of data" (defined by *send*:θ such as *send*:*bill*), or "notify about some type of data" (e.g. *notify*:θ such as *notify*:*energyconsumption*).

Besides the simplified data protection requirements, the strength of our approach is the data possession and data connection policies, as well as the automated verification of these. Although at the policy and architecture levels the verification process may seem to be simpler than in case of verifying a program code, it is relevant to detect any design flaws at these higher levels. Manual and informal reasoning can be error-prone, especially when there are many complex data types and entities in the system.

8 Implementation

8.1 The System Architecture Specification Page

After launching the tool, as depicted in Figure 17, the default page can be seen, where the user can specify a system architecture. DataProVe supports two types of components, the so-called main components, and the sub-components. The main components can represent an entire organisation, system or entities that consists of several smaller components, such as a service provider, a customer, or authority (trusted third-party organisation). Sub-components are elements of a main component, for example, a service provider can have a server, a panel, or storage place. A main component usually has access to the data handled by its own sub-components, but this is not always the case, for instance, two main components can share a sub-component and only one main-component has access to its data. This can happen, for example, when a service provider operates a device of a trusted third party, but it does not have free access to the content of the data stored inside the device.

In the first version of DataProVe (v0.9), main components are represented by rectangular shapes, while sub-components are represented by circles. Examples can be seen in Figures 18-21.

In this report, we will interchange between the two terms entity and component, because the term entity has been used in our theoretical papers, while the tool uses the term component more. They refer to the same thing in our context.

In DataProVe one can specify which main-component can have access to which sub-component. An example can be seen in Figure 22, where we specified the relation between sp and server, meter, as well as between the authority auth and meter, socialmediapage.

In Figure 23, a new text box is created with the name *recvdmsg1*, which denotes that the server receives a message called *msg1*. Its content (depicted in Figure 24) says that sp can receive

Figure 17: After launching DataProVe, the system architecture specification page can be seen.

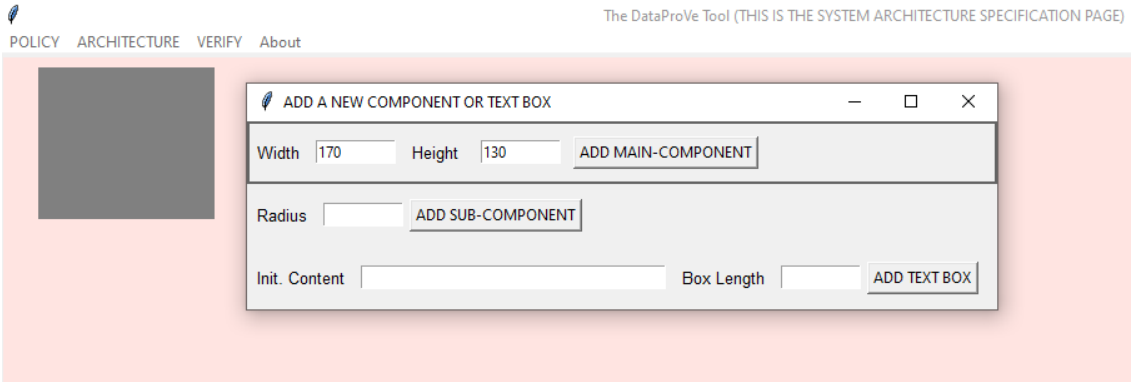

Figure 18: Adding a main component of size 170x130.

a reading that contains the energy consumption (energy) and the customer ID (*custID*).

In the architecture level, we distinguish entity/component, actions and data, where actions specify what an entity/component can do on a piece of data (it may not perform this action eventually during a low-level system run, but there are instances of the system run that where this action happens), except for DELETEWITHIN, as we will see later.

8.1.1 ACTIONS

Based on the definition of actions and architectures in Figure 5, we propose their corresponding formats that can be given in the text boxes/text editor in DataProVe.

Actions are words/string of all capital letters, and DataProVe supports the actions "OWN", "RECEIVE", "RECEIVEAT", "CREATE", "CREATEAT", "CALCULATE", "CALCULATEAT", "STORE", "STOREAT", "DELETE", "DELETEWITHIN". The syntax of each action in DataProVe is as follows.

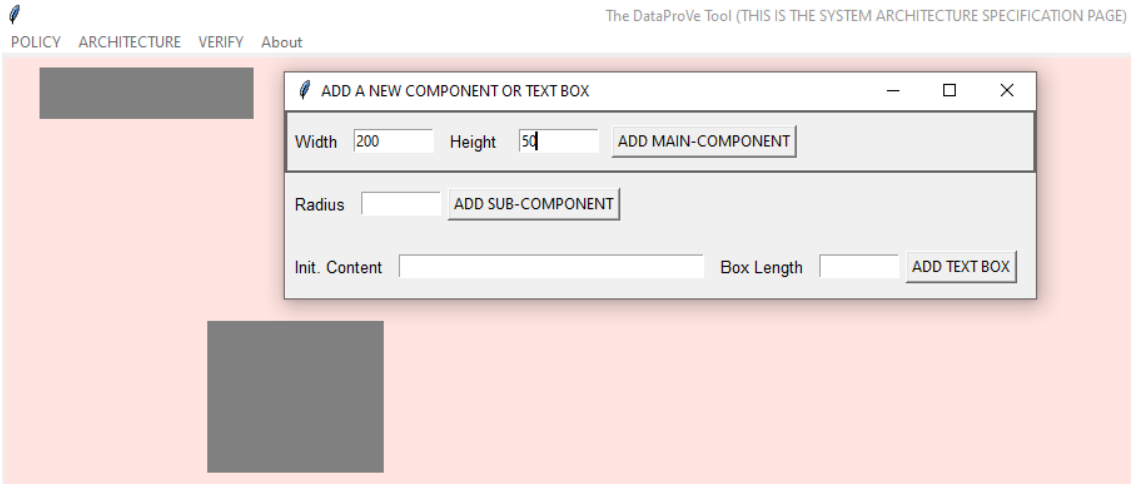

Figure 19: Adding a new main component of size 200x50.

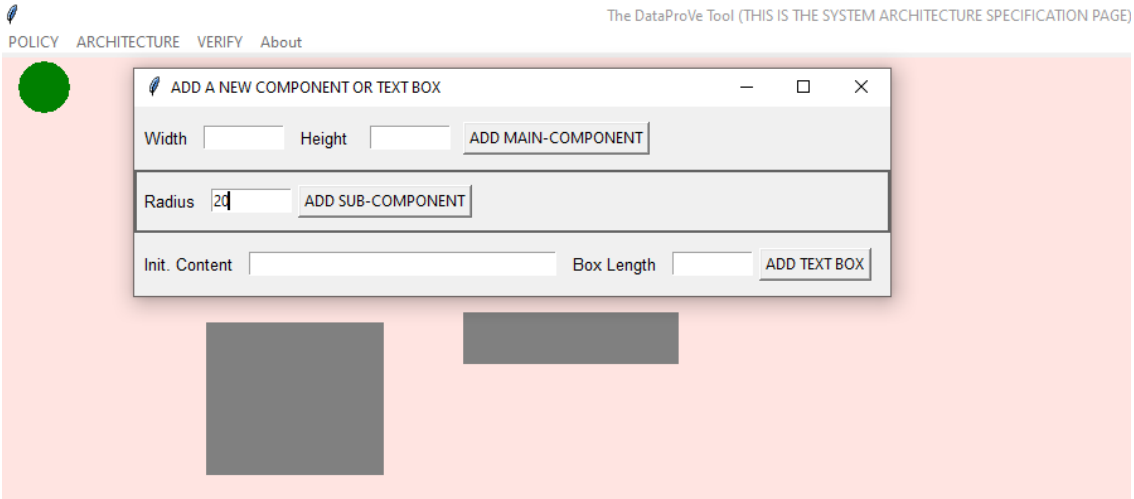

Figure 20: Adding a new sub-component with a radius size of 20.

Note: no space character is allowed when specifying the actions in the bullet points below.

The reserved/pre-defined keywords are highlighted in bold, while the non-bold text can be freely defined by the user:

- **OWN**(component,Datatype) :

  This action defines that a component (e.g., **sp**, auth, server, meter etc.) can own a piece of data of type Datatype. For example, **OWN**(server,spkey) say they server can own the a piece of data of type service provider key (spkey).

- **RECEIVE**(component,Datatype):

  This action defines that a component can receive a piece of data of type Datatype, for example, **RECEIVE**(server,Sicknessrecord(name,insurancenumber)) says that server can receive a sickness record that contains a piece of data of type name and insurance number.

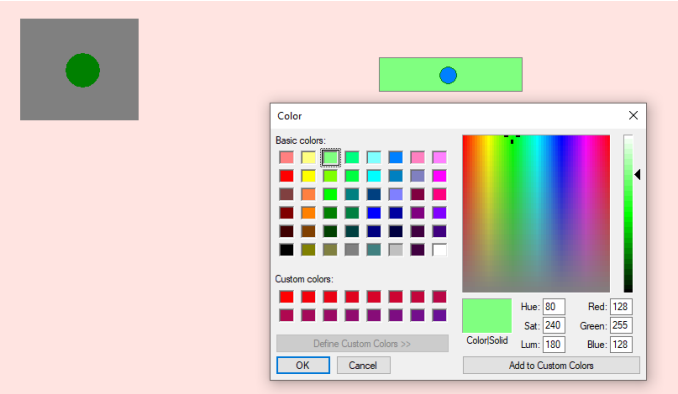

Figure 21: Choosing the color for a component.

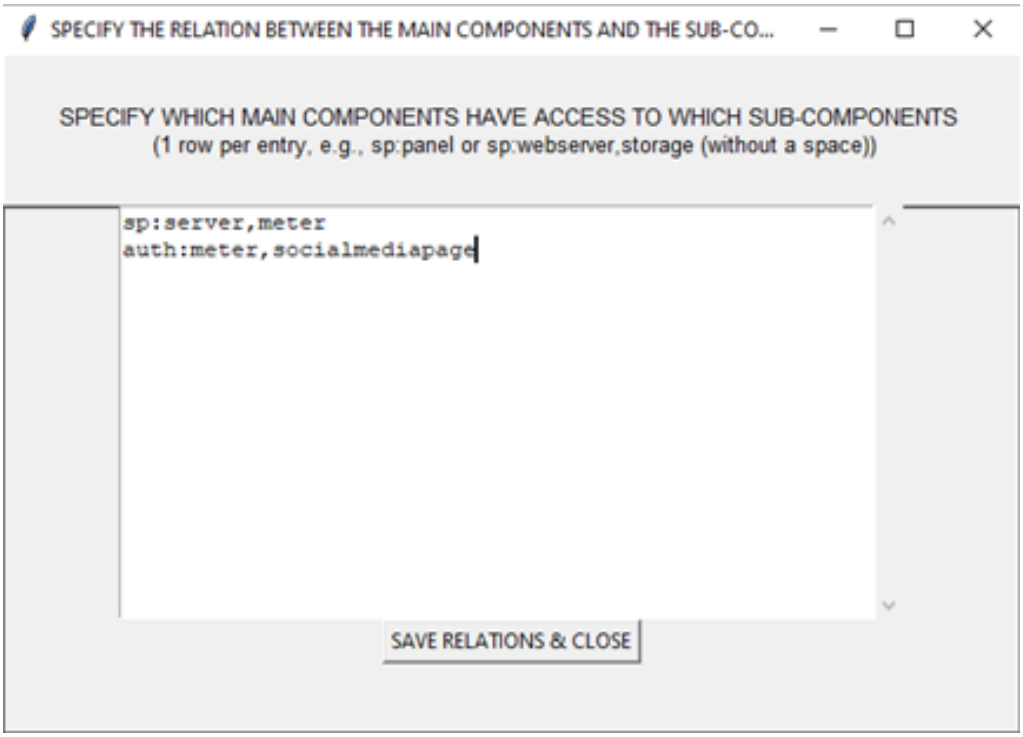

Figure 22: Specify which main component has access to the data in which sub component (sp has access to server and meter, while auth has access to meter and socialmediapage).

- **RECEIVEAT**(component,Datatype,**Time(t)**):

  This action is similar to the previous one, except that here we also need to define the time when the data can be received. Since at the architecture level we do not intent to specify the concrete time value, the generic time construct, denoted by the keyword **Time(t)** specifies that component can receive a piece of data of type Datatype at some (not specific) time **t**. **RECEVEAT** is used to define when a consent (**Cconsent**(Datatype), **Uconsent**(Datatype), **Sconsent**(Datatype), **Fwconsent**(Datatype)) is received.

- **CREATE**(component,Datatype):

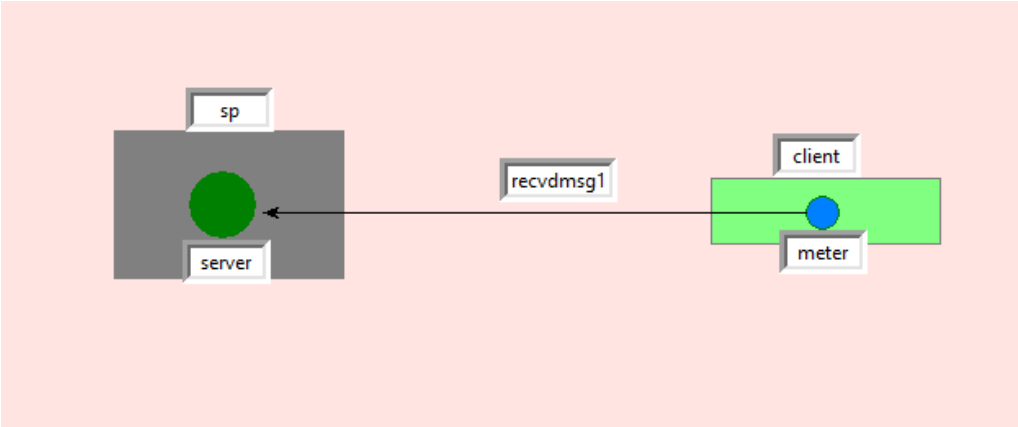

Figure 23: Draw an arrow from the component meter to server.

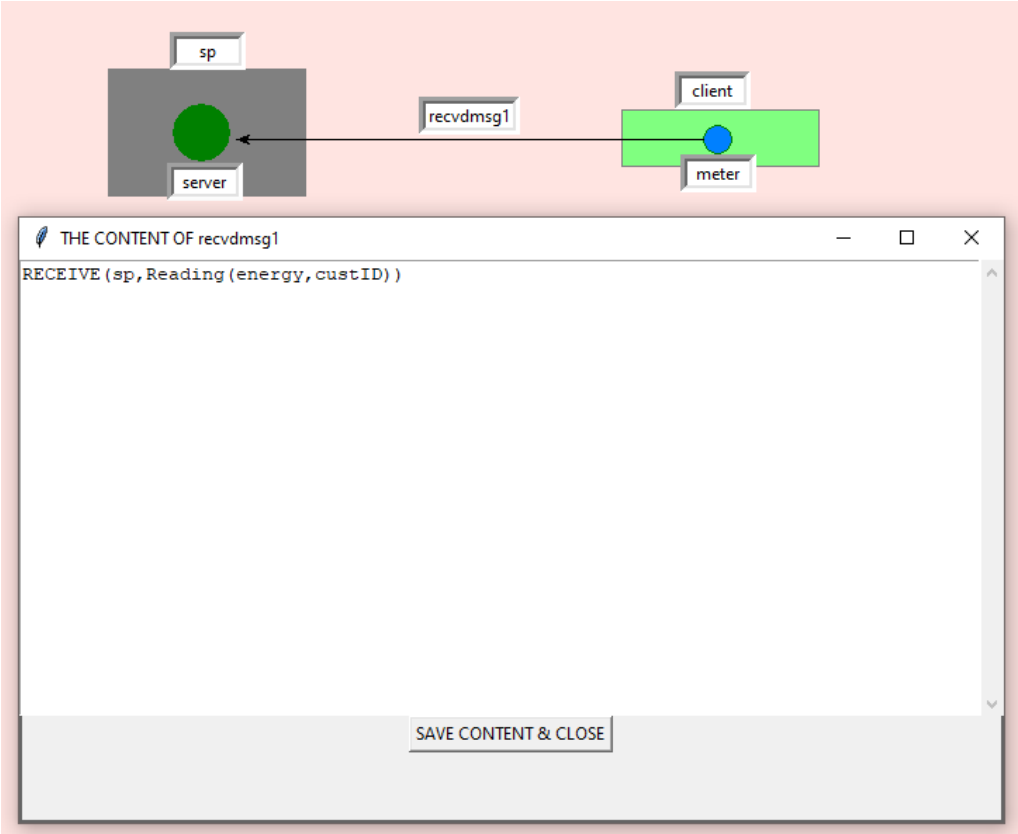

Figure 24: Specify the message content of recvdmsg1 (through the action RECEIVE).

This action defines that a component can create a piece of data of type Datatype, for instance, **CREATE**(**sp**,Account(name,address,phone)) defines that a service provider **sp** can create an account that contains three pieces of data of types name, address and phone number.

- **CREATEAT**(component,Datatype,**Time(t)**):
  This action defines that a component can create a piece of data of type Datatype at some

(not specific) time **t**. For example, **CREATE**(**sp**,Account(name,address,phone),**Time(t)**).

- **CALCULATE**(component,Datatype):

  This action defines that a component can calculate a piece of data of type Datatype, for instance, **CALCULATE**(**sp**,Bill(energyconsumption)) defines that a service provider **sp** can calculate a bill using a piece of data of type energy consumption.

- **CALCULATEAT**(component,Datatype,**Time(t)**):

  This action defines that a component can calculate a piece of data of type Datatype at some (not specific) time **t**. For example, **CALCULATE**(**sp**,Bill(energyconsumption),**Time(t)**).

- **STORE**(storageplace,Datatype):

  This action defines that a service provider can store a piece of data of type Datatype in storageplace, where storageplace can be **mainstorage**, **backupstorage**. These reserved keywords define a collection of storage place(s) that can be seen as “main" storage, or “backup" storage, respectively.

  For example, **STORE**(**mainstorage**,Account(name,address,phone)) defines that a service provider can store an account that contains name, address and phone number in its main storage place(s).

- **STOREAT**(storageplace,Datatype,**Time(t)**):

  This action defines that a component can store a piece of data of type Datatype in the place(s) storageplace at some (not specific) time **t**.

  For example, **STORE**(**mainstorage**,Account(name,address,phone),**Time(t)**) defines that an account with a name, address and phone number can be stored in the main storage of the service provider at some time **t**.

- **DELETE**(storageplace,Datatype):

  The action delete is closely related to the action store, as it defines that a piece of data of type Datatype can be deleted from storageplace.

  For example, **DELETE**(**mainstorage**,Account(name,address,phone)) captures that a service provider can.

- **DELETEWITHIN**(storageplace,Datatype,**Time**(tvalue)):

  This action captures that once the data is stored, a component must delete a piece of data of type Datatype within the given time value tvalue (tvalue is a data type for time values). Unlike the non-specific **Time(t)**, which is a predefined construct, tvalue is defined by the user, and takes specific time values such as 3 years or 2 years 6 months.

  For example, **DELETE**(**mainstorage**,Account(name,address,phone),**Time**(2y)) defines that the service provider must delete an account from its main storage within 2 years.

8.1.2 COMPONENTS/ENTITY

A component can be specified by a string of all lower case, for example, a service provider can be specified by **sp**, or a third-party authority by auth (obviously they can be specified with any other string).

DataProVe supports some pre-defined or reserved components/entities, such as **sp**, **trusted**, **mainstorage**, **backupstorage**.

- **sp**: this reserved keyword defines a service provider. DataProVe only allows a single service provider at a time (in the specification of a policy and architecture).

- **trusted**: this reserved keyword defines a trusted authority that is able to link a pseudonym to the corresponding real name.

- **mainstorage**: this reserved keyword defines the collection of main storage places of a service provider.
- **backupstorage**: this reserved keyword defines the collection of backup storage places of a service provider.

> *Note: An entity/component is always defined as the first argument of an action.*

8.1.3 DATA TYPES

DataProVe supports two groups of data types, the so-called compound data types, and simple data types.

- **Simple data types** do not have any arguments, and they are specified by strings of all lower cases, without any space or special character. Example simple data types include name, address, phonenumber, nhsnumber, etc.
- **Compound data types** have arguments, and they are specified by strings that start with a capital letter followed by lower cases (again without any space or special character). For example, Account(name,address,phone) is a compound data type that contains three simple data types as arguments. Another example compound data type can be Hospitalrecord(name,address,insurance). Any similar compound data types can be defined by the user. We note that the space character is not allowed in the compound data types.

  Nested compound data types are compound data types that contain another compound data types. For instance, Hospitalrec(Sicknessrec(name,disease),address,insurance) captures a hospital record that contains a sickness record of a name and disease, and an address, and finally, an insurance number.

> *Note: The first version of DataProve (v0.9) supports three layers of nested data types.*

DataProVe has pre-defined or reserved data types, such as

- The types of consents: **Cconsent**(Datatype), **Uconsent**(Datatype), **Sconsent**(Datatype), **Fwconsent**(Datatype).

  We do not differentiate among the different consent format, it can be e.g. written consent, or online consent form, or some other formats.

  - **Cconsent**(Datatype): This is a type of collection consent on a piece of data of type Datatype. For example, **Cconsent**(illness), **Cconsent**(Account(creditcard,address)) capture the collection consent on the illness information, and the account containing a credit card number and address.
  - **Uconsent**(Datatype): A type of usage consent on a piece of data of type Datatype. For example, **Uconsent**(Energy(gas,water,electricity)), **Uconsent**(address).
  - **Sconsent**(Datatype): A type of storage consent on a piece of data of type Datatype. For example, **Sconsent**(personalinfo), **Sconsent**(Account(creditcard,address)) defines the types of storage consent on a type of personal information and account, respectively.
  - **Fwconsent**(Datatype,component): A type of forward/transfer consent on a piece of data of type Datatype, and a component to whom the data is forwarded/transfered. E.g. **Fwconsent**(personalinfo,auth), **Fwconsent**(Account(creditcard,address),auth) defines the type of forward consent on the type of personal information and account, respectively, as well as a third party authority (auth) to which the given data is forwarded.

- The types of time and time value: **Time(t)** or **Time**(tvalue), where **Time**() is a time data type, while the pre-defined special keyword t denotes a type of non-specific time, and tvalue is a type of time value (such as 5 years, 2 hours, 1 minute, etc.). tvalue is a (recursive) type and takes the form of

  $$tvalue ::= y \mid mo \mid w \mid d \mid h \mid m \mid numtvalue \mid tvalue + tvalue$$

  where y specifies a year, mo a month, w a week, d a day, h an hour and m a minute. Further, numtvalue is the a number (num) before tvalue, for example if num = 3 and tvalue = y, then numtvalue is 3y (i.e. 3 years). Additional examples include tvalue = 5y + 2mo + 1d + 5m.

  It is important to note that **Time**(tvalue) can only be used in the action **DELETEWITHIN**, **RECEIVEAT**, **CREATEAT**, **CALCULATEAT**, **STOREAT** must contain the non-specific time **Time(t)**.

  For example, the actions

  - **DELETEWITHIN**(**sp**,**mainstorage**,Webpage(photo,job),**Time**(10y+6mo)) Any webpage must be deleted from the main storage of the service provider within 10 years and 6 months.
  - **RECEIVEAT**(**sp**,**Cconsent**(illness),**Time(t)**) The service provider can receive a collection consent on illness information at some non-specific time **t**.
  - **RECEIVEAT**(**sp**,**Uconsent**(Webpage(photo,job)),**Time(t)**) The service provider can receive a usage consent on a webpage at some non-specific time **t**.
  - **STOREAT**(**sp**,**backupstorage**,Webpage(photo,job),**Time(t)**) The service provider can store a webpage in its back up storage places at some non-specific time **t**.
  - **CREATEAT**(server,Account(name,address),**Time(t)**): The service provider can create an account that contains a name and address in at some non-specific time **t**.
  - **CALCULATEAT**(**sp**,Bill(tariff,Energy(gas,water,electricity)),**Time(t)**): The service provider can create an account that contains a name and address in at some non-specific time **t**.

- The type of metadata and meta values: **Meta**(Datatype).

  This data type defines the type of metadata (information about other data), or information located in the header of the packets, the meta information often travels through a network without any encryption or protection, which may pose privacy concern. Careful policy and system design are necessary to avoid privacy breach caused by the analysis of metadata or header information.

  > *Note:* ***Meta****(Datatype) is always defined as the last argument in a piece of data.*

  Example application of metadata includes:

  - **RECEIVE**(sp,Sicknessrec(name,disease,**Meta**(ip))):
    This action defines that the service provider can receive a packet that containing a name and disease, but the packet also includes the metadata IP address of the sender computer. We note that this syntax is simplified in terms that it aims to eliminate the complexity of nested data type. Specifically, this syntax abstracts away from the definition of the so-called packet data type, an “abbreviation" of the lengthy **RECEIVE**(**sp**,Packet(Sicknessrec(name,disease),**Meta**(ip)))).

- **RECEIVE**(**sp**,Sicknessrec(name,disease,**Meta**(**Enc**(ip,k)))): This action is similar to the previous one, but now the metadata IP address is encrypted with a key k.
- **RECEIVEAT**(**sp**,Sicknessrec(name,disease,**Meta**(ip)),**Time(t)**):
  This action is similar to the first one, but it includes the time data types at the end. It defines that the service provider receives the sickness record along with the IP address of the sender device, at some non-specific time **t**.

Obviously, any metadata can be defined instead of IP address in the examples above.

- The type pseudonymous data: **P**(Datatype | component).

  This data type defines the type of pseudonymous data, for example, a pseudonym. The argument can be either a data type or a component [11]. Pseudonym is a means for achieving a certain degree of privacy in practice as the real identity/name and the pseudonym can only be linked by a so-called trusted authority. DataProVe also captures this property, namely, only the component trusted can link the pseudonym to the real name/identity.

  For example,

  - **RECEIVE**(**sp**,Sicknessrec(**P**(name),disease)):
    This action defines that a service provider can receive a sickness record, but this time, the name in the record is not the real name but a pseudonym, hence, the service provider cannot link a real name to a disease.
  - **RECEIVE**(**trusted**,Sicknessrec(**P**(name),disease)):
    This is similar to previous case, but the trusted authority can receive a sickness record instead of the service provider.
  - **RECEIVE**(**sp**,Sicknessrec(**P**(name),disease,**Meta**(ip))): Again, this is similar to the first case, but with metadata.
  - **RECEIVEAT**(**sp**,Sicknessrec(name,disease,**Meta**(ip)),**Time(t)**): This is similar to previous case, but also include the time data type.

- The types of cryptographic primitives and operations: DataProVe supports the basic cryptographic primitives for the architecture. Again, we provide the reserved keywords in bold.

  - Private key: **Sk**(Pkeytype):
    This data type defines the type of private key used in asymmetric encryption algorithms. Its argument has a type of public key (Pkeytype). We note that public key is not a reserved data type.
  - Symmetric encryption: **Senc**(Datatype,Keytype):
    This is the type of the cipher text resulted from a symmetric encryption, and has two arguments, a piece of data and a symmetric key (Keytype).
    For example,
    - **RECEIVE**(**sp**,**Senc**(Account(name,address),key)):
      This specifies that a service provider can receive a symmetric key encryption of an account using a key of type key.
    - **RECEIVE**(**sp**,**Senc**(Account(**Senc**(name,key),address),key)):
      This specifies that a service provider can receive a symmetric key encryption of an account that contains another encryption of a name, using a key of type key.
    - **OWN**(**sp**,key):
      This specifies that a service provider can own a key of type key.

[11] This would be in the versions above v0.9. In the version 0.9, DataProVe preserves the keyword (all small letters) **ds** for data subject, and the user can define P(**ds**) to specify that the real data subject/identity has been pseudonymised.

– Asymmetric encryption: **Aenc**(Datatype,Pkeytype):

  This is the type of the cipher text resulted from an asymmetric encryption, and has two arguments, a piece of data and a public key (Pkeytype).

  For example,

  * **RECEIVE**(**sp**,**Aenc**(Account(name,address),pkey)):
    This specifies that a service provider can receive an asymmetric key encryption of an account using a public key of type pkey.
  * **CALCULATE**(**sp**,**Sk**(pkey)):
    This specifies that a service provider can calculate a private key corresponding to the public key (of type pkey).
  * **OWN**(**sp**,pkey):
    This specifies that a service provider can own a public key of type pkey.

– Message authentication code (MAC): **Mac**(Datatype,Keytype):

  This is the type of the message authentication code that has two arguments, a piece of data and a symmetric key (Keytype).

  For example,

  * **RECEIVE**(**sp**,**Mac**(Account(name,address),key)):
    This specifies that a service provider can receive a message authentication code of an account using a key of type key.

– Cryptographic hash: **Hash**(Datatype):

  This is the type of the cryptographic hash that has only one argument, a piece of data.

  For example,

  * **RECEIVE**(server,**Hash**(password)):
    This specifies that a server can receive a hash of a password.
  * **STORE**(**sp**,**mainstorage**,**Hash**(password)):
    This specifies that a service provider can store a hash of a password in its main storage place(s).

8.2 The Data Protection Policy Specification Page

On the data protection policy specification page, we can define a high-level data protection policy (as shown in Figure 25).

8.2.1 Entities/Components (the top part)

The policy page has three parts, the top part is to specify the entities/components in the system, such as authority, client etc. On the left side, the user is expected to provide a short notation, and on the right side, the full name/description to help identifying the meaning of the notation. For instance, in Figure 25, the notation is auth, and the description is third party authority. After adding a new entity, it will appear in the drop-down option menu in the bottom part. Note that **the entity sp (service provider) is a pre-defined entity** that is already added by default (hence, the user does not need to add). The user can specify any other entities.

8.2.2 Data groups/Data types (the middle part)

The middle part in the policy specification page is for defining the data groups and data types. As shown in Figure 26, the user can define a group of data types, for instance, a data group denoted by *personalinfo* is defined which includes four data types, name, address, *dateofbirth*, and *phonenumber*.

The option menu in the middle (called “IS THIS UNIQUE") expects the user to provide if the data group together with its data types can be used to uniquely identify an individual. For

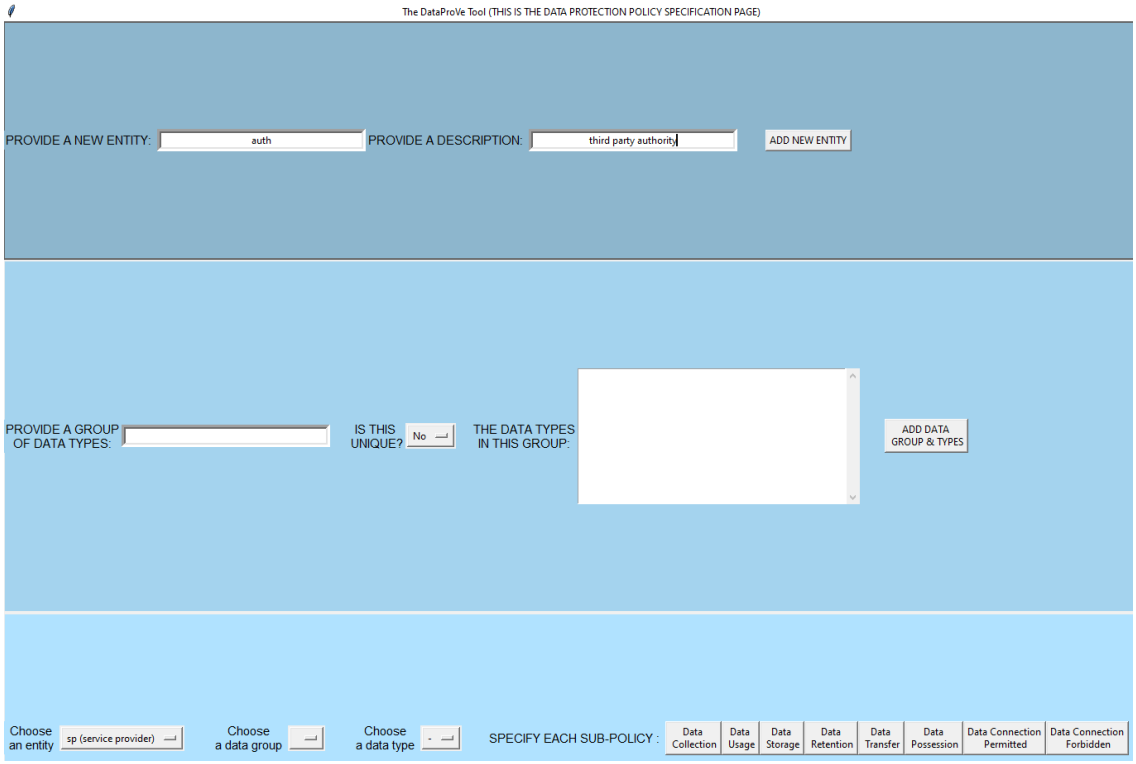

Figure 25: The Policy Specification Page.

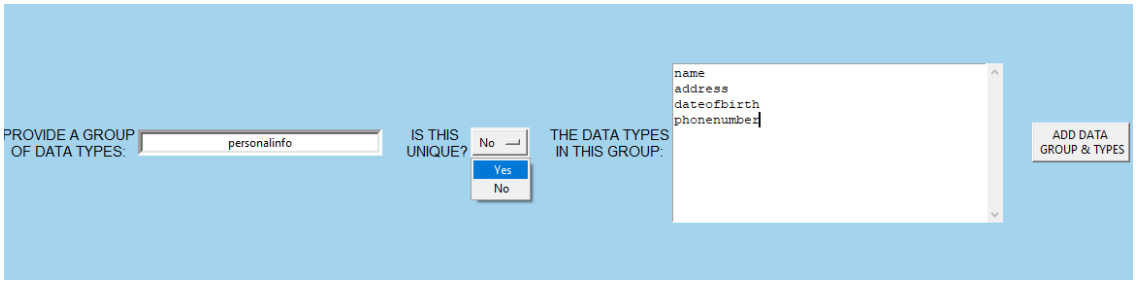

Figure 26: Specifying data groups (personalinfo) and its data types.

instance, a name alone cannot be used to unique identify an individual, but a name together with an address, date of birth and phone number, can be, so the option "Yes" was chosen. Another example is shown in Figure 27, with the data group called energy (refers to energy consumption) and its data types, gas, water, and electricity consumption. This type group together with its types cannot be used to uniquely identify an individual, hence, the option "No" was chosen.

8.2.3 Policy specification (the bottom part)

Based on the syntax of the policy language given in Section 3.1, we follow the seven sub-policies. However, here to avoid confusion we divide the last sub-policy, the data connection policy, into two categories,the data connection permit and data connection forbid policies. In the first one the user can specify which data link they allow, while in the second one for which they forbid.

A data protection policy is defined on a data group/type and an entity. In DataProVe, each

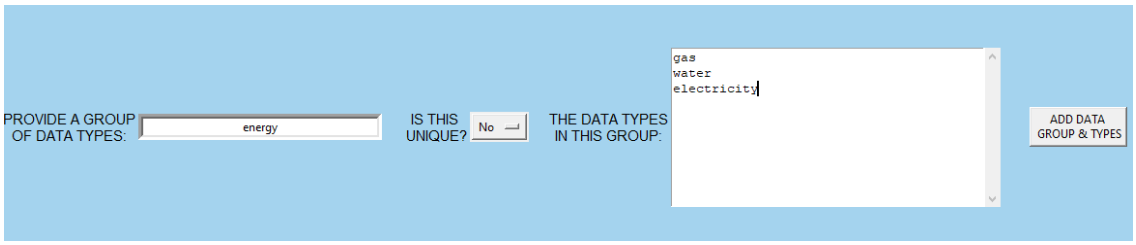

Figure 27: Specifying data groups (energy) and its data types.

policy consists of eight sub-policies, to achieve a fine-grained requirement specification (Figure 28). The users do not have to define all the eight sub-policies, but they can if it is necessary. Both the policies and architectures can be saved, and opened later to modify or extend.

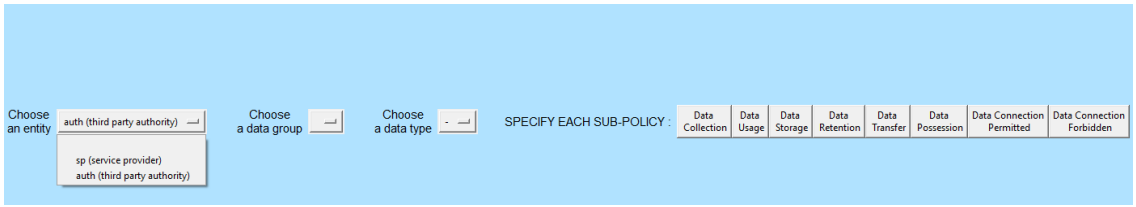

Figure 28: The Policy Specification Page (entities).

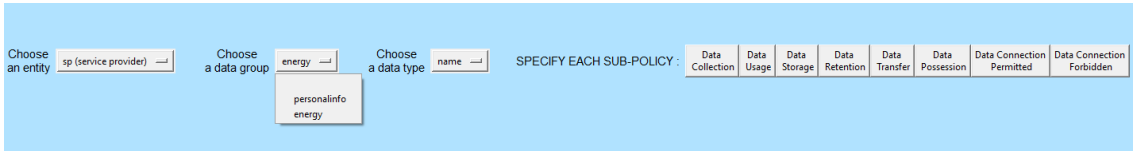

Figure 29: The Policy Specification Page (data groups).

The first five sub-policies (collection, transfer) are defined *only from the service provider's* perspective. For the rest three sub-policies (data possession and the two data connections policies), the user can specify from any entity's perspective.

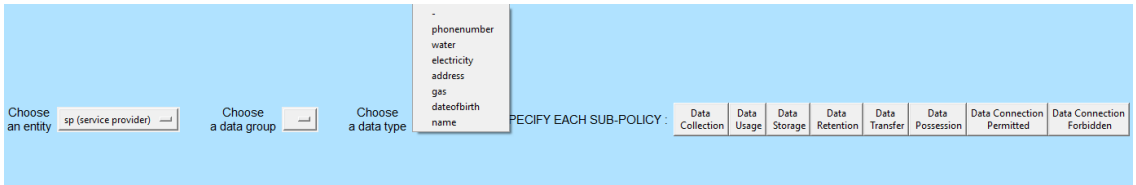

Figure 30: The Policy Specification Page (choosing among data types).

The eight sub-policies are data collection, data usage, data storage, data retention, data transfer, data possession and the two data connection sub-policies.

The data collection sub-policy: In the data collection sub-policy window, for a given entity and data group the user can specify whether consent is required to be collection when the selected entity collect a selected data group (Y for Yes/N for No), and then specify the collection purposes.

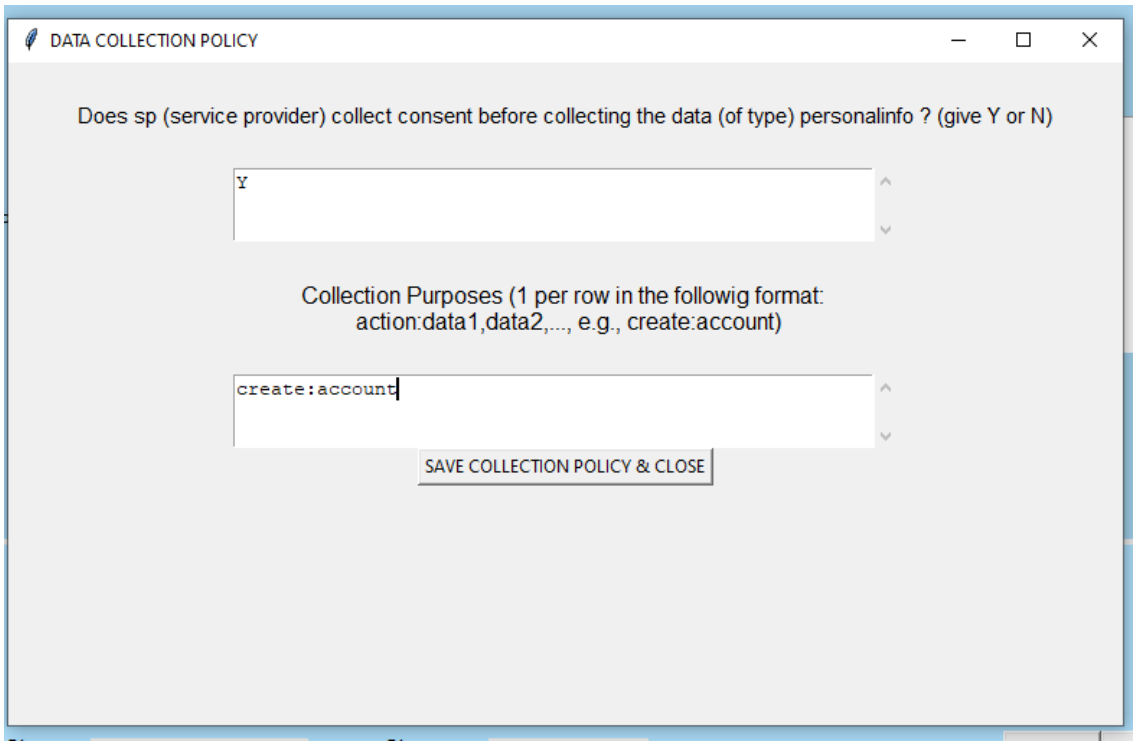

Figure 31: The data collection sub-policy.

The collection purposes can be given row by row, each row with a different action in the format of:

$$\boxed{action1{:}data1,data2,\ldots,data\_n}$$

where action1 can be any action, while data1,. . . , data_n are compound data types (note that these compound data types do not need to be specified/added in the policy). For example, in Figure 31, the user sets that consent is required to be collected when the service provider collects the personal information. Then, the collection purpose for personal information is to create an account. The compound type account does not need to be defined in the policy.

The data possession sub-policy:

The data possession sub-policy defines who can have/possess a piece of data of a given group. The users only need to specify who are allowed to have or possess a given data group, DataProVe automatically assumes that the rest entities/components are not allowed to have/possess the selected type of data.

The data connection permitted sub-policy: This sub-policy specifies which entity is permitted to connect or link two types/groups of data.

In the second drop-down option menu, the user can specify further if the selected entity is permitted to be able to link two pieces of data uniquely, meaning that it will be able to deduce that the two pieces of data belongs to the same individual.

For example, in Figure 33, we specified that the service provider is permitted to be able to link the data group energy and the data group personalinfo. However, we do not allow the service provider to be able to uniquely link the two data groups. Obviously, if personalinfo was defined as unique, then unique link would be possible, so there is chance that the architecture always violates this requirement of the policy.

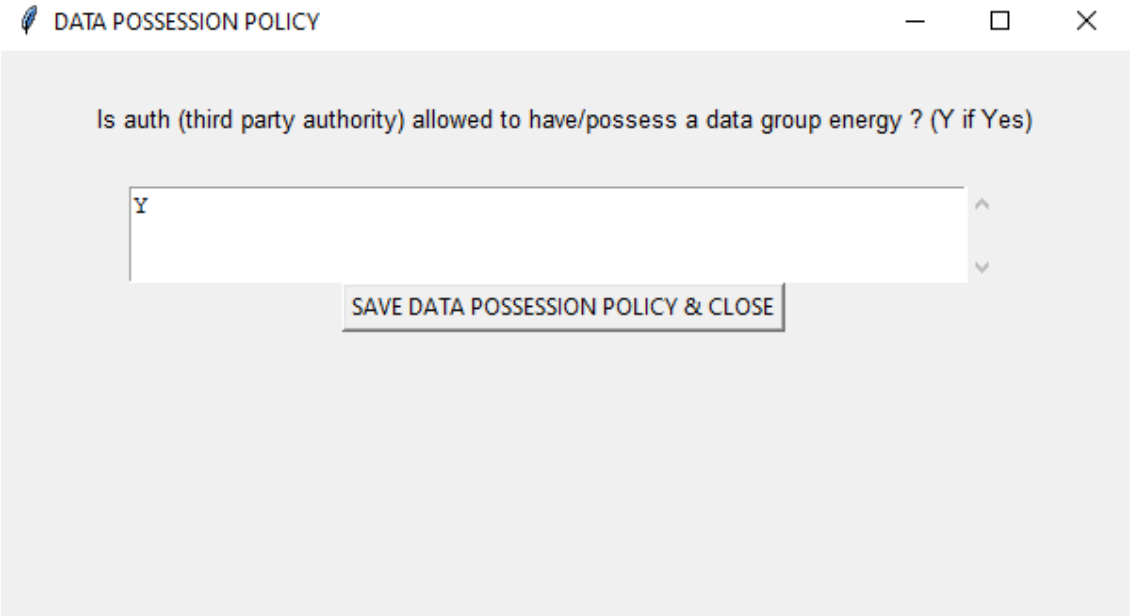

Figure 32: The data possession sub-policy.

The data connection forbidden sub-policy: This sub-policy is the counterpart of the permitted policy. While in case of the data possession policy, the user only needs to specify which entity is allowed to have or possess certain type of data, and DataProVe automatically assumes that the rest are not allowed, here the user needs to explicitly specify which pair of data types/groups are an entity is forbidden to be able to link together.

For example, in Figure 34, we forbid for the third-party authority to be able to link the data group personalinfo with the data group energy. Here, we forbid the unique link-ability of these two data groups for the third-party authority.

If we choose "No" (Figure 35), then it means that any ability to link any two pieces of data of the given data groups, is forbidden (not just unique link). Hence, this option is stricter than the previously one.

8.3 Conformance verification

We define three types of conformance, namely, functional conformance, privacy conformance and the so-called DPR conformance.

8.3.1 Functional conformance

The functional conformance captures if an architecture is functionally conforming with the specified policy. Namely:

1. If in the policy, we allow for an entity to be able to have a piece of data of certain data type/group, then in the architecture the same entity can have a piece of data of the same type/group.
2. If in the policy, we allow for an entity to be able to link/uniquely link two pieces of data of certain types/groups, then in the architecture the same entity can link/uniquely link two pieces of data of the same types/groups.
3. If in the policy, the (collection, usage, storage, transfer) consent collection is not required for a piece of data of given type/group, then in the architecture there is no consent collection.

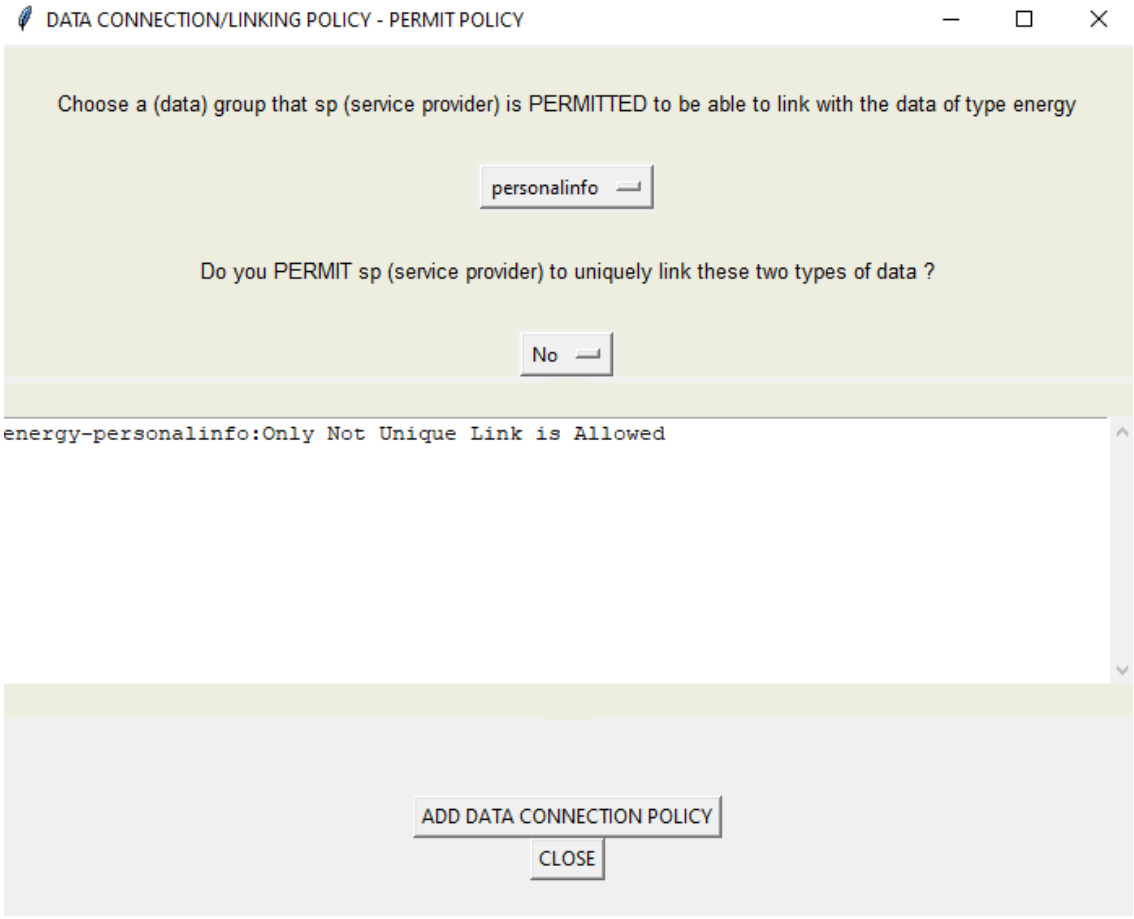

Figure 33: The data connection permission sub-policy.

4. If in the policy, we define

   (a) a storage option “Main and Backup Storage" for a piece of data of certain type/group, then in the architecture there is a **STORE** or **STOREAT** action defined for both **mainstorage** and **backupstorage**, and for the same data type/group;

   (b) a storage option “Only Main Storage", then in the architecture there is a **STORE** or **STOREAT** action defined for only **mainstorage**, and for the same data type/group.

   (c) If in the policy, we allow a piece of data of certain type/group, data, to be transferred to an entity ent, then in the architecture there is **RECEIVEAT**(ent,data,**Time(t)**) or **RECEIVE**(ent,data).

8.3.2 Violation of the functional conformance

1. In the policy, we allow for an entity to be able to have a piece of data of certain data type/group, but in the architecture the same entity cannot have a piece of data of the same type/group.

2. In the policy, we allow for an entity to be able to link/uniquely link two pieces of data of certain types/groups, but in the architecture the same entity cannot link/uniquely link two pieces of data of the same types/groups.

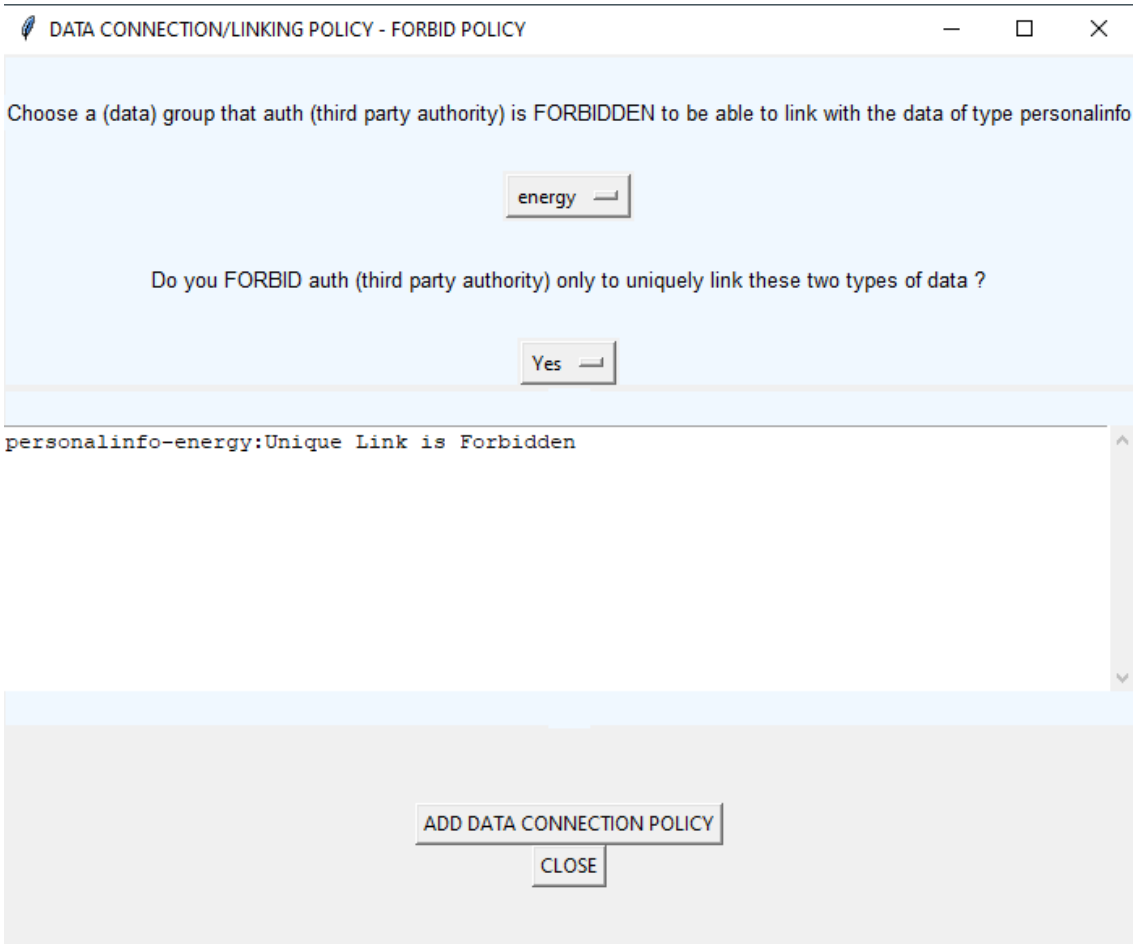

Figure 34: The data connection permission sub-policy. The case when only unique link is forbidden.

3. In the policy, the (collection, usage, storage, transfer) consent collection is not required for a piece of data of given type/group, but in the architecture there is a consent collection, namely, an action
   - **RECEIVEAT**(**sp**,**Cconsent**(data),**Time(t)**), or
   - **RECEIVEAT**(**sp**,**Sconsent**(data),**Time(t)**), or
   - **RECEIVEAT**(**sp**,**Uconsent**(data),**Time(t)**), or
   - **RECEIVEAT**(third,**Fwconsent**(data,third),**Time(t)**).
4. In the policy, we define
   (a) a storage option "Main and Backup Storage" for a piece of data of certain type/group, but in the architecture, there is **STORE** or **STOREAT** action defined for only either **mainstorage** or **backupstorage**, or no store action defined at all, for the same data type/group;
   (b) a storage option "Only Main Storage", but in the architecture there is no **STORE** or **STOREAT** action defined at all, for the same data type/group.

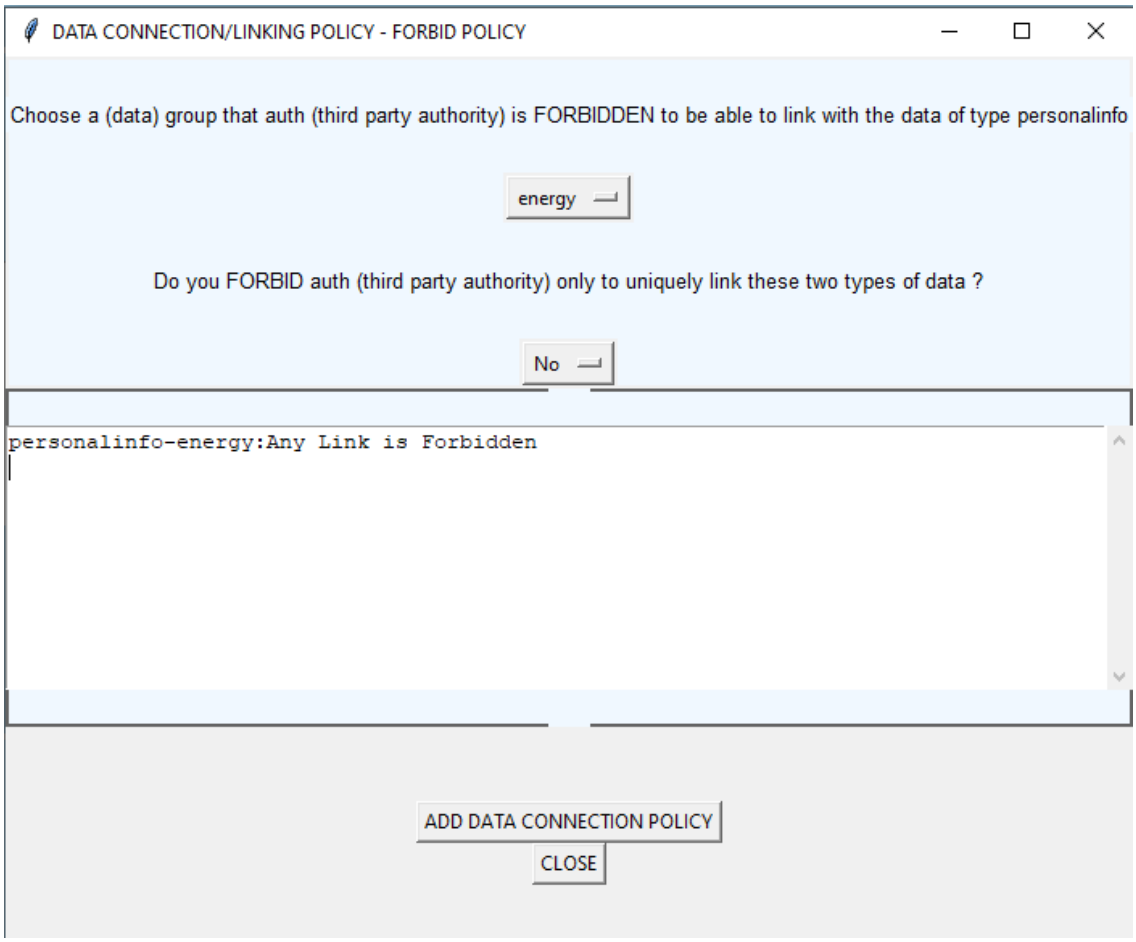

Figure 35: The data connection permission sub-policy.

5. In the policy, we allow a piece of data of certain type/group, data, to be transferred to an entity ent, but in the architecture there is no **RECEIVEAT**(ent,data,**Time(t)**) or **RECEIVE**(ent,data) defined (i.e., data is not transferred to the entity ent).

8.3.3 Privacy conformance

The privacy conformance captures if an architecture satisfies the privacy requirements defined in the policy. Namely:

1. If in the policy, we forbid for an entity to be able to have or possess a piece of data of certain type/group, then in the architecture the same entity cannot have or possess a piece of data of the same type/group.
2. If in the policy, we forbid for an entity to be able to link/uniquely link two pieces of data of certain types/groups, then in the architecture the same entity cannot link/uniquely link two pieces of data of the same types/groups.

8.3.4 Violation of the privacy conformance

1. In the policy, we forbid for an entity to be able to have or possess a piece of data of certain type/group, but in the architecture the same entity can/is be able to have or possess a piece of data of the same type/group.

Figure 36: To verify the conformance between the specified system architecture and policy.

2. In the policy, we forbid for an entity to be able to link/uniquely link two pieces of data of certain types/groups, but in the architecture the same entity can/is be able to link/uniquely link two pieces of data of the same types/groups.

8.3.5 DPR conformance

The privacy conformance captures if an architecture satisfies the data protection requirements defined in the policy. Namely:

1. If in the policy, the (collection, usage, storage, transfer) consent collection is required for a piece of data of given type/group, then in the architecture there is a collection for the corresponding consent.

2. If in the policy, we define a (collection, usage, storage) purpose action:data for a piece of data of certain type/group, then in the architecture there is the action action defined on a compound data type data.

8.3.6 Violation of the DPR conformance

1. In the policy, the (collection, usage, storage, transfer) consent collection is required for a piece of data of given type/group, but in the architecture, there is no collection for the corresponding consent.

2. In the policy, we define a (collection, usage, storage) purpose action:data for a piece of data of certain type/group, but in the architecture there is not any action action defined on a compound data type data, or besides action, there are also other actions defined in the architecture on data that are not allowed in the policy.

3. In the policy, we define

   (a) a storage option “Main and Backup Storage" for a piece of data of certain type/group, but in the architecture there is a STORE or STOREAT action defined for some storage place, different from **mainstorage** and **backupstorage**, for the same data type/group;

(b) a storage option “Only Main Storage", but in the architecture there is a **STORE** or **STOREAT** action defined for some storage place, different from **mainstorage**, for the same data type/group.

4. In the policy, we define

   (a) a deletion option “From Main and Backup Storage" for a piece of data of a certain data type/group, data, but in the architecture there is not any of the action
   - **DELETE**(**mainstorage**,data) or
   - **DELETEWITHIN**(**mainstorage**,data,**Time**(tvalue)), or
   - **DELETE**(**backupstorage**,data) or
   - **DELETEWITHIN**(**backupstorage**,data,**Time**(tvalue));

   (b) a deletion option “Only From Main Storage" for a piece of data of a certain data type/group, data, but in the architecture there is no action **DELETE**(**mainstorage**,data) or **DELETEWITHIN**(**mainstorage**,data,**Time**(tvalue)).

5. In the policy, we allow a piece of data of certain type/group, data, to be transferred to an entity ent, but in the architecture there is also an action **RECEIVEAT**(ent1,data,**Time(t)**) or **RECEIVE**(ent1,data) defined for some ent1 to whom we do not allow data transfer in the policy.

8.4 Application Examples

In this section, we highlight the operation of DataProVe using two very simple examples.

8.4.1 Example 1 (Data retention policy)

In this example, in the policy we specify a data group (a group of data types) called *personalinfo*, which is stored centrally at the main storage places of the service provider. In the storage sub-policy, we also set that storage consent is required before the storage of *personalinfo*. Finally, we do not give service provider (sp) the right to have the data of group/type *personalinfo*. In the deletion policy, we set the retention delay in the main storage to 8 years (i.e. 8y in Figure 37).

In the architecture level, we add an action that says a piece of data of type personalinfo must be deleted from the main storage within 10 years (action DELETEWITHIN, in the last line).

Content of *spmessages*: **RECEIVEAT**(**sp**,**Sconsent**(personalinfo),**Time**(t))
Content of *storagemessages*: **RECEIVEAT**(**mainstorage**,personalinfo,**Time**(t))
Content of *storemain*: **STOREAT**(**mainstorage**,personalinfo,**Time**(t))
Content of *deletion*: **DELETEWITHIN**(**mainstorage**,personalinfo,**Time**(10y)).

In the architecture shown in Figure 38, the service provider (sp) can receive a storage consent for *personalinfo* at some non-specific time *t*. The main storage places of sp can receive the data at some non-specific time and store it. The data of this type/group is deleted within 10 years from the main storage places.

As a verification result (Figure 39), we got that the architecture violates the privacy conformance, as the architecture allows for sp to have the data of type personalinfo after 8 years, however, in the policy we set it to only 8 years. In the last line of the verification result window, we can also see a DPR conformance property, namely, sp collects storage consent before the data is stored.

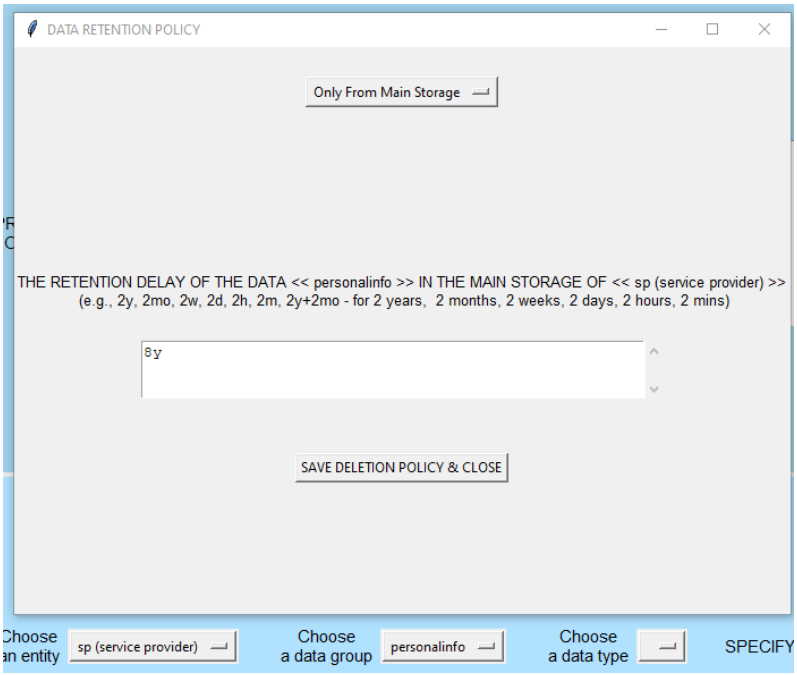

Figure 37: We set that the data of type/group personal information must be deleted from the main storage places of the service provider within 8 year.

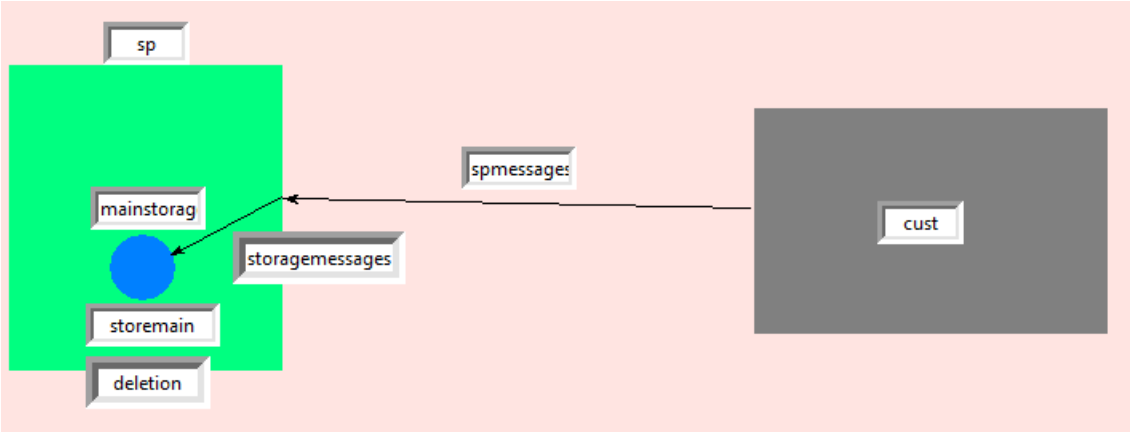

Figure 38: The service provider (sp) stores the personal information in its main storage places.

8.5 Example 2 (Data possession and connection policy)

In the second simple example, we focus on the data possession and data connection sub-policies. We present the receive action with the Meta construct (metadata or "packet" header data such as IP address, source, destination addresses, etc.).

In the policy, we define four data groups, nhsnumber (National Health Service number), name, photo, and address (see Figure 40).

Then, we forbid (any kind of link-ability, not only unique link) for the service provider to be able to link two pieces of data of types nhsnumber, and photo (see Figure 41). Again, we also forbid for the service provider to be able to have all the four data types/groups.

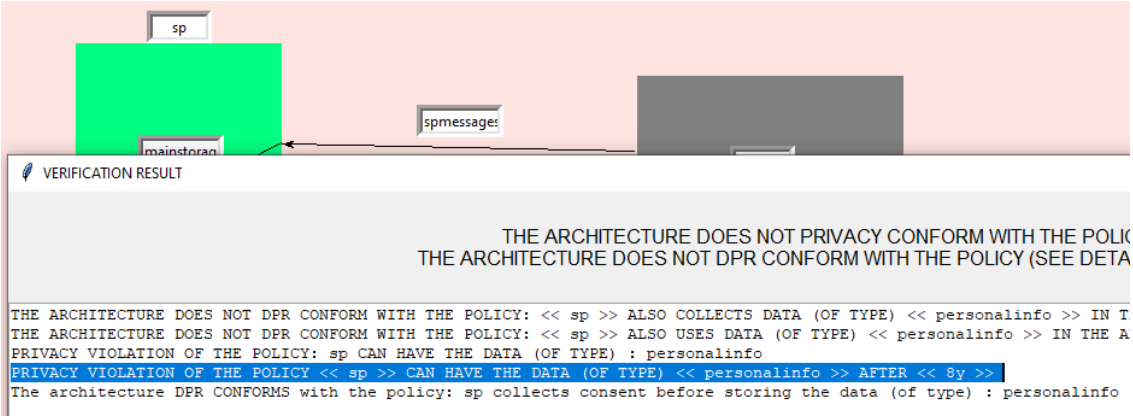

Figure 39: The verification results show the violation of the privacy and DPR conformance properties. We also got the first two lines of DPR conformance because in this example, we did not specify the collection and usage sub-policies (we left them blank).

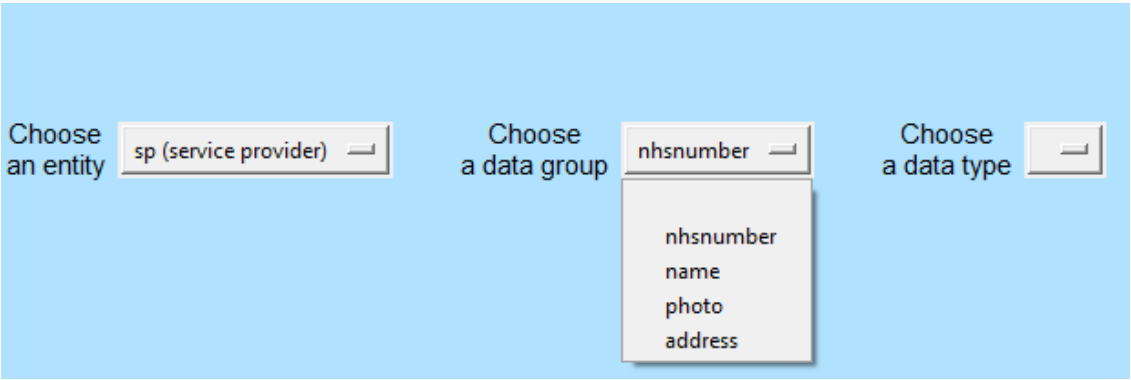

Figure 40: The policy level with the four data types/groups.

In the architecture, a service provider collects data from two phone applications (Figure 42). The "HealthXYZ" app sends the service provider a sickness record with a public IP address (an unique IP of a phone) other app, called, "SocialXYZ" also sends the social profile with the same ip address (same phone). Both data types are encrypted (using symmetric key encryption) with the service provider keys (and sp owns the two keys).

Content of *spmessage1* in Figure 42:
RECEIVE(**sp**,**Senc**(Sicknessrecord(nhsnumber,name,**Meta**(ip)),spkey1))

Content of *spmessage2* in Figure 42:
RECEIVE(**sp**,**Senc**(Socprofile(photo,address,**Meta**(ip)),spkey2))

Content of *spowned* in Figure 42:
OWN(**sp**,spkey1)
OWN(**sp**,spkey2)

As a result (Figure 43), we got that the service provider not only be able to link the data of

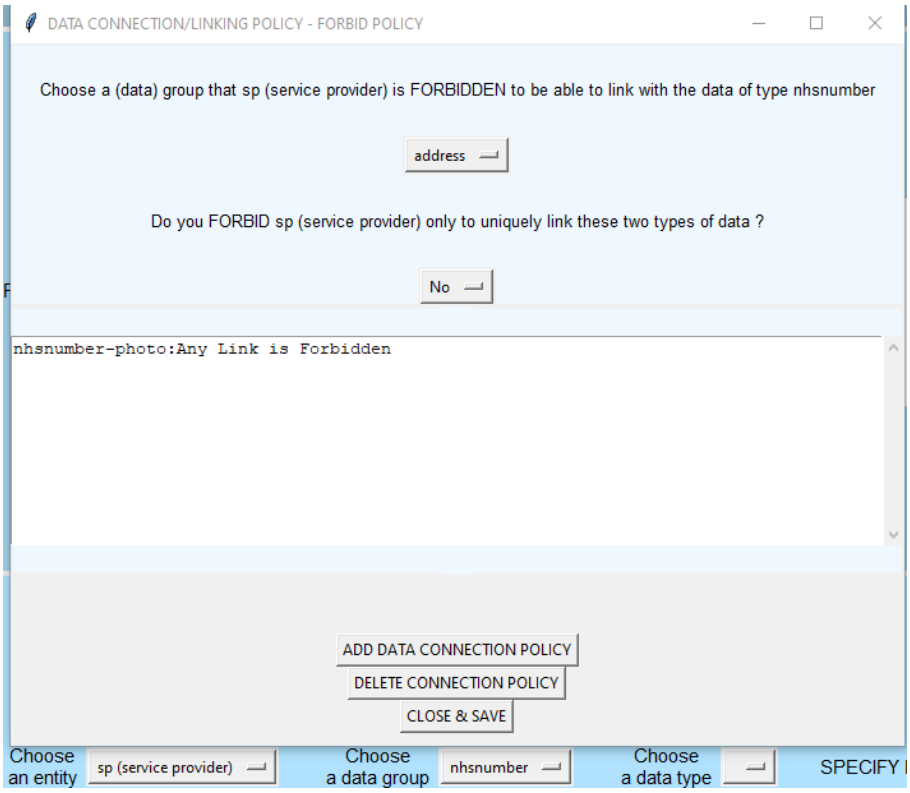

Figure 41: The specified data connection sub-policy for example 2.

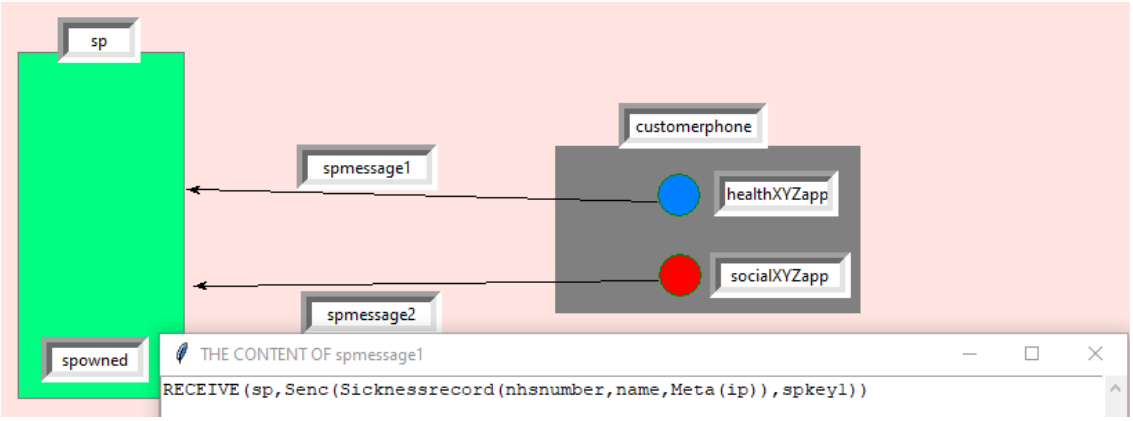

Figure 42: The specified architecture for example 2.

types nhsnumber with the data of type photo, but it also has all the data of types nhsnumber, name, photo and address. The reason is that sp will be able to decrypt both messages and link, have the data inside them. Note that we only have linkability but not unique link, because the Apps can be used by different people in one family, so the set of possible individuals can be narrowed down, but *sp* cannot be sure that *nhsnummber* and *photo* belong to the same individual.

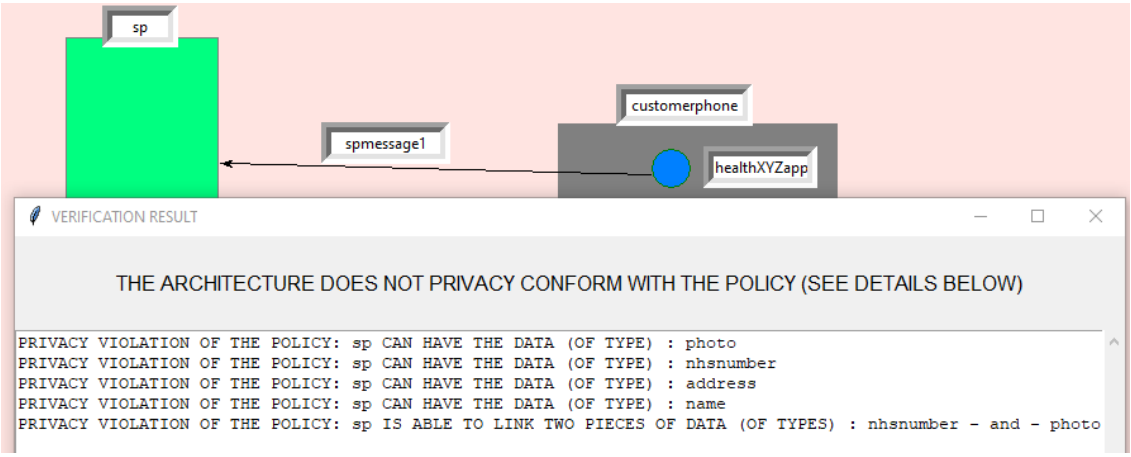

Figure 43: The verification result for example 2.

9 Conclusion and Future Work

We addressed the problem of formal specification and automated verification of data protection requirements at the policy and architecture levels. Specifically, we proposed a variant of policy and architecture languages to specify a simple set of data protection requirements based on the GDPR. In addition, we proposed DataProVe, a tool based on the syntax of our languages and a logic based verification engine to check the conformance between a policy and an architecture. In this paper, our language variants and tool only cover a limited set of data protection requirements in an abstract way, hence, there are many possibilities to extend and improve their syntax and semantics to specify more complex laws. Regarding the conformance check of the privacy properties (the right to have and link data), a possible extension would be including the behaviour of the hostile attackers (e.g. steal personal data) in the verification. Finally, we plan to improve the effectiveness of the conformance check algorithm for the data types with a large number of nested layers.

References

[1] General Data Protection Regulation (GDPR). Article 4. https://gdpr-info.eu/art-4-gdpr/.

[2] Erika McCallister, Tim Grance, Karen Scarfone. Guide to Protecting the Confidentiality of Personally Identifiable Information (PII). Natinonal Institute of Standards and Technology. US Department of Commerce, SP 800-122, 1995.

[3] Karen Kullo. Facebook sued over alleged scanning of private messages. Bloomberg, 2 January 2014. http://www.bloomberg.com/news/articles/2014-01-02/facebook-sued-over-alleged-scanning-of-private-messages.

[4] Samual Gibbs. Belgium takes Facebook to court over privacy breaches and user tracking. The Guardian, 15 June 2015. http://www.theguardian.com/technology/2015/jun/15/belgium-facebook-court-privacy-breaches-ads.

[5] Sean Buckley. Deleting Google Photos won't stop your phone from uploading pictures. Engaget.com, 13 July 2015. http://www.engadget.com/2015/07/13/deleting-google-photos-wont-stop-your-phone-from-uploading-pict/.

[6] Facebook and Cambridge Analytica: What You Need to Know as Fallout Widens. The New York Times, 19 March 2018. https://www.nytimes.com/2018/03/19/technology/facebook-cambridge-analytica-explained.html.

[7] Google faces UK suit over alleged snooping on iPhone users. Financial Times, 30 November 2017. https://www.ft.com/content/9d8c7136-d506-11e7-8c9a-d9c0a5c8d5c9.

[8] General Data Protection Regulation (GDPR). Article 25. https://gdpr-info.eu/art-25-gdpr/.

[9] General Data Protection Regulation (GDPR). Article 6. https://gdpr-info.eu/art-6-gdpr/.

[10] The Platform for Privacy Preferences. P3P, 2012. http://www.w3.org/P3P/.

[11] The Platform for Privacy Preferences (P3P). APPEL 1.0, 2012. http://www.w3.org/TR/2002/WD-P3P-preferences-20020415/.

[12] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Xpref: a preference language for p3p. *Computer Networks*, 48(5):809 – 827, 2005. Web Security.

[13] Kathy Bohrer and Bobby Holland. Customer Profile Exchange (CPExchange) Specification Version 1.0, 2000. http://xml.coverpages.org/cpexchangev1_0F.pdf.

[14] OASIS Open. Extensible access control markup language (xacml) version 3.0, 2017. http://docs.oasis-open.org/xacml/3.0/errata01/os/xacml-3.0-core-spec-errata01-os.html.

[15] P. Ashley, S. Hada, G. Karjoth, C. Powers and M. Schunter. Enterprise Privacy Authorization Language (EPAL 1.2), 2000. http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/.

[16] Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, Karin Bernsmed, Anderson Santana De Oliveira, and Jakub Sendor. A-ppl: An accountability policy language. In Joaquin Garcia-Alfaro, Jordi Herrera-Joancomartí, Emil Lupu, Joachim Posegga, Alessandro Aldini, Fabio Martinelli, and Neeraj Suri, editors, *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance*, pages 319–326, Cham, 2015. Springer International Publishing.

[17] S Trabelsi, Akram Njeh, Laurent Bussard, and Gregory Neven. Ppl engine: A symmetric architecture for privacy policy handling. *W3C Workshop on Privacy and data usage control*, pages 1–5, 04 2010.

[18] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proceedings 16th National Conference on Artificial Intelligence*, AAAI-99, pages 291–298, Orlando, USA, 1999. ACM.

[19] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[20] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 31–46, Washington, DC, USA, 1997. IEEE Computer Society.

[21] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 18–38, London, UK, UK, 2001. Springer-Verlag.

[22] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '03, pages 63–, Washington, DC, USA, 2003. IEEE Computer Society.

[23] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In Wilhelm Schäfer and Pere Botella, editors, *Software Engineering — ESEC '95*, pages 137–153, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[24] A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992.

[25] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transaction on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[26] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[27] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[28] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: architecture for component trading and dynamic updating. In *Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No.98EX159)*, pages 43–51, 1998.

[29] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[30] R. B. Franca, J. Bodeveix, M. Filali, J. Rolland, D. Chemouil, and D. Thomas. The aadl behaviour annex – experiments and roadmap. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 377–382, 2007.

[31] J. Perez, I. Ramos, J. Jaen, P. Letelier, and E. Navarro. Prisma: towards quality, aspect oriented and dynamic software architectures. In *Third International Conference on Quality Software, 2003. Proceedings.*, pages 59–66, 2003.

[32] Valérie Issarny, Amel Bennaceur, and Yérom-David Bromberg. *Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability*, pages 217–255. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[33] Amelia Bădică and Costin Bădică. Fsp and fltl framework for specification and verification of middle-agents. *Int. J. Appl. Math. Comput. Sci.*, 21(1):9–25, March 2011.

[34] Vinh-Thong Ta, Denis Butin, and Daniel Le Métayer. Formal accountability for biometric surveillance: A case study. In Bettina Berendt, Thomas Engel, Demosthenes Ikonomou, Daniel Le Métayer, and Stefan Schiffner, editors, *Privacy Technologies and Policy*, pages 21–37, Cham, 2016. Springer International Publishing.

[35] Denis Butin and Daniel Le Métayer. Log Analysis for Data Protection Accountability. In *19th International Symposium on Formal Methods (FM 2014)*, volume 8442 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2014.

[36] Vinh-Thong Ta and Thibaud Antignac. Privacy by design: On the conformance between protocols and architectures. In Frédéric Cuppens, Joaquin Garcia-Alfaro, Nur Zincir Heywood, and Philip W. L. Fong, editors, *Foundations and Practice of Security*, pages 65–81, Cham, 2015. Springer International Publishing.

[37] Thibaud Antignac and Daniel Le Métayer. Privacy architectures: Reasoning about data minimisation and integrity. In Sjouke Mauw and Christian Damsgaard Jensen, editors, *Security and Trust Management*, pages 17–32, Cham, 2014. Springer International Publishing.

[38] General Data Protection Regulation (GDPR). Article 5. https://gdpr-info.eu/art-30-gdpr/.

[39] General Data Protection Regulation (GDPR). Article 30. https://gdpr-info.eu/art-30-gdpr/.

[40] General Data Protection Regulation (GDPR). Article 17. https://gdpr-info.eu/art-17-gdpr/.

[41] General Data Protection Regulation (GDPR). Article 46. https://gdpr-info.eu/art-46-gdpr/.