Guaranteeing Privacy Policies using Lightweight Type Systems

Guaranteeing Privacy Policies using Lightweight Type Systems

Robin Adams[a], Wolfgang Schulz[b], Sibylle Schupp[c], Florian Wittner[d]*

1. Introduction

In the current state of the software industry, the design of a piece of software is never fixed, but is constantly evolving both before and after it is released onto the market. It is very rare for a piece of software or hardware to be designed once, and then built to the design that never changes: either bug fixes or feature requests need to be accommodated.

At the same time, the EU General Data Protection Regulation (GDPR)[a] – in effect since May 25 2018 – requires software that handles personal data to always adhere to the Regulation's conditions, not just when it is released on the market. This has always been true for data protection laws, and affected companies have always been bearing the responsibility of reconciling this fact with the abovementioned reality of highly dynamic software design. What has changed or more specifically will change with the GDPR are the precise obligations that controllers (as defined by Art. 4 Nr. 7) are facing when it comes to proving and ensuring ongoing compliance with the Regulation's material conditions and its accountability principle. Additionally, drastically increased amounts of possible sanctions – Art. 83 (4), (5) allow for penalties up to 20,000,000 € or 4% of a company's worldwide turnover – further heighten the importance of efficient and reliable compliance strategies. Making sure that new processes and applications remain lawful in their handling of personal data is paramount.

Among the new guidelines formulating compliance obligations Art. 24 and 25 promise to be in the center of importance. According to Art. 24 controllers shall implement appropriate technical and organisational measures to ensure and to be able to demonstrate that processing is performed in accordance with this Regulation. Art. 25, concretising these general remarks, states that those technical and organisational measures shall be designed in an effective manner to implement data-protection principles and necessary safeguards already into the architecture as well as the processing ('Data protection by design'), all the while taking into account and being oriented towards the 'state of the art'. This expression in a legal context is not new – environmental law statutes such as §3 (6) BImSchG (German Federal Immission Control Act) have been using it since the 1970s – but needs to be comprehended and interpreted independently: because of the GDPR's nature as a European legislative act and because it lacks the specification through technical appendixes that usually accompany such expressions. While Art. 24 (3) and 25 (3) do allow for or even rely on the implementation of approved certification mechanisms and codes of conducts (defined in Art. 40 and 42) as indicators for compliance, it will take some time before enough of these have been established to provide the required clarity and guidance.

It is thus not surprising that in the months and weeks leading up to the date the regulation went into effect most companies still felt unprepared and overwhelmed with the notion of adapting their practices to the new framework.[b]

a Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden,

b Hans-Bredow-Institute for Media Research, Department of Law, Hamburg, Germany.

c Institute for Software Systems, Department of Electrical Engineering, Computer Science, and Mathematics, Hamburg University of Technology (TUHH), Hamburg, Germany.

d Corresponding author. Hans-Bredow-Institute for Media Research, Department of Law, Hamburg, Germany. E-mail address: f.wittner@hans-bredow-institut.de.

Keywords: GDPR, Type Theory, Data Protection by Design, Accountability, State of the Art

a REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 27 April 2016.

What we aim at with this paper is not a clear-cut definition of these vague and as of yet undefined norms. Instead, we want to give an example of a specific technique that helps companies make sure their evolving system is compliant with specific conditions and principles set by the GDPR. We thereby want to contribute to the global development of what the GDPR calls 'state of the art'. Simultaneously we envision the usage of this technique to further the cooperation between data controlling companies and supervisory authorities by making it easier for companies to prove and for authorities to check for compliance regarding certain GDPR conditions.

We seek ways to guarantee that a given computer system satisfies a given privacy policy. Because the design of a piece of software is always changing, we are not interested in methods that require a lengthy analysis of a finished design. Best practice in software engineering now is to have a suite of automated tests that can be run after every change to the design – typically a set of fast tests that are run several times a minute, and a set of slow tests that are run once a day overnight. Yet, even extensive testing of a privacy policy directly does not yield guarantees: there are too many cases to define and run. We wish the method to be one that allows scope for the design to change, by proving that, if a set of local conditions (which are amenable to automated testing) hold, then a privacy policy is guaranteed to hold.

Specifically, we model a system as composed of *agents* or *components* which may be persons, software, or something else (e.g. companies). We seek ways to impose a *local* condition on each agent that guarantees that the system as a whole satisfies a *global* condition.

'If every component satisfies condition C, then the system as a whole satisfies condition X.'

We wish condition C to be easy to check of a given component. The software engineer is then free to modify the design of each component, the number of components, and the way they interact as they wish. Provided condition C always holds of every component, we know the policy will be satisfied by the system as a whole.

For this reason, we concentrate on the messages that are passed between components. We wish to impose restrictions on the messages that the components can send to each other, and prove results such as:

'If every agent sends and receives only messages of types $A_1, \ldots, A_n$ then it is impossible for the end user to receive the child's personal data without first having received consent from the parent.'

In the following paragraphs, we first want to explain the technical prerequisites of utilising *type theory* - a theory from computer science - for this approach. In a series of scenarios, we then want to demonstrate its flexibility when it comes to checking compliance of a computer system by applying it to a set of specific conditions and general principles of the GDPR. In the end, we shall assess and review its possibilities as well as its limitations and discuss its potential practical value.

2. Related work

The usage of technical mechanisms and approaches as a support for better understanding, defining and incorporating data protection obligations and measures has been a subject of techno-legal scientific debate for a while. So-called privacy enhancing technologies (PETs) have been developed and increasingly used already since the 1980s[a] with various priorities on e.g. data subjects' transparency[b] or identity management

b By way of example: study by Veritas as reported on by Forbes: https://www.forbes.com/sites/adrianbridgwater/2017/04/25/worldwide-climate-of-fear-over-gdpr-data-compliance-claims-veritas-study/ (Accessed on 03 September 2018).

a Even though the term itself wasn't introduced until 1995; see Registratiekamer & Information and Privacy Commissioner of Ontario, *Privacy-Enhancing Technologies: The Path to Anonymity* (Achtergrondstudies en Verkenningen 5B, vol. I & II, Rijswijk, August 1995).

b See Simone Fischer-Hübner and Stefan Berthold *Computer and Information Security Handbook* (2nd edn, Elsevier 2013) pp. 767-769 for an overview over transparency-enhancing tools (TETs).

systems.[c] Over the years, a wide variety of approaches has been offered to couple legal prerequisites and their implementation in real-life business practices and processing operations in a way that serves all actors involved. Some of these approaches focus on building entire ontologies[d] able to depict data protection principles and obligations[e], thereby trying to benefit controllers as well as Data Protection Agencies (DPAs) in comparing obligations with measures taken. Others focus on making data flows within technological systems transparent and traceable in order to enhance transparency and therefore accountability for the same actors, especially with regards to decision-making processes and the application of individual responsibilities in interconnected systems with a multitude of actors involved.[f] Others focus on the relationship between controllers and data subjects, trying to formalise the respective roles and duties for the provision or denial of data access in specific situations.[g]

In particular, there have been several approaches that use type theory in various ways as a tool to prove that systems satisfy privacy requirements. Using type theory, application developers can express general policies as well as specialised ones and detect automatically vulnerabilities or violations; accordingly, a wide range of applications for type systems exist. Most recently, type theory became popular in security research, where now a variety of domain-specific, typed languages is available that allow tracking information flow[h], even taking conditionals like prior authorization[i] into account. Interpreting passing of private data as information flow, researchers started developing type systems for privacy, addressing both specific problems like the equivalence of privacy protocols[j] and abstract frameworks.[k]

We survey some of these approaches here, and then explain how our approach differs from these past applications of type theory.

Myers and Liskov (2000)[l] give an approach that shares several features in common with ours. They give a type system that, like ours, assigns types to the data that is passed between the components of a distributed system. These types carry information about which users are allowed to access the data. They also describe a version of the programming language Java that can be used to write software in a way that provides a formal guarantee that the software will not then violate this condition and allow a user to access data whose type says they should not.

Nanevski *et al* (2011)[m] and Stewat and Aleksandar (2013)[n] describe a programming language called Relational Hoare type theory (RHTT), and show how a programmer may, while writing a program in RHTT,

c Cf. Andreas Pfitzmann and Marit Hansen, 'Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management – A Consolidated Proposal for Terminology' (2008) Version v0.31, http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.31.pdf.

d Cf. for general, not data protection related ontologies Núria Casellas, *Legal Ontology Engineering: Methodologies, Modelling Trends, and the Ontology of Professional Judicial Knowledge* (Vol. 3, Springer Science & Business Media, 2011).

e Cf. Cesare Bartolini, Robert Muthuri and Cristiana Santos 'Using Ontologies to Model Data Protection Requirements in Workflows' (JSAI International Symposium on Artificial Intelligence 2015), building on prior work such as the NEURONA ontologies, see Núria Casella and others, 'Ontological Semantics for Data Privacy Compliance: The NEURONA Project' (Proceedings of the Intelligent Privacy Management Symposium, March 2010) pp. 34-38.

f Cf. Jatinder Singh, Jennifer Cobbe and Chris Norval, 'Decision Provenance – Capturing Data Flow for Accountable Systems' (2018) arXiv:1804.05741.

g Cf. Philip WL Fong, 'Relationship-Based Access Control: Protection Model and Policy Language' (Proceedings of the first ACM Conference on Data and Application Security and Privacy, February 2011) pp. 191-202.

h Andrei Sabelfeld and Andrew C. Myers, 'Language-based information-flow security' (2003) IEEE Journal on selected areas in communications 21.1 pp. 5-19.

i Nikhil Swamy, Juan Chen and Ravi Chugh, 'Enforcing stateful authorization and information flow policies in Fine' (European Symposium on Programming Springer, 2010) pp. 529-549.

j Veronique Cortier and others, 'A type system for privacy properties' (Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security) pp. 409-423.

k Dimitrious Kouzapas and Anna Philippou, 'A typing system for privacy' (International Conference on Software Engineering and Formal Methods, Springer, Cham 2013) pp. 56-68; Georgeous Pitsiladis and Petros Stefaneas, 'Implementation of privacy calculus and its type checking in Maude' (International Symposium on Leveraging Applications of Formal Methods, Springer, Cham 2018) pp. 477-493.

l Andrew C. Myers and Barbara Liskov, 'Protecting privacy using the decentralized label model' (ACM Transactions of Software Engineering Methodology, 9, 4, October 2000) pp. 410-442.

specify a privacy-relevant property as a type of RHTT, and then provide a proof that their program satisfies this property, which may then be checked by a computer.

Laurenco and Caires (2015)[o] and Li and Zhang (2017)[p] describe a simple programming language with types that may carry *levels* of security. Two different levels, *low* and *high* (say), may be used to represent data that it is safe to make public, and data that must be kept private, respectively. It is then impossible, using their language, to write a program that will use a value whose level is *high* in a place where the level must be *low*. A typical example of such a place would be displaying the data to an unknown user.

While all these approaches share some of the goals and technical groundwork, our type-theory-based approach differs in important parts. All the attempts we have described use type theory as a tool for writing and/or analysing programs in such a way that certain privacy-relevant properties are guaranteed to hold. So applying them to a system would, in practice, mean either using them when we write the source code, or using them to analyse the source code, to verify that one component satisfies a given condition.

We are using type theory as a tool in a very different way. Our approach does not require us to be able to read or change the source code in any component. It only requires access to the messages that pass between the components. In more detail, our algorithms take a privacy policy and compute a type theory. The guarantee that, if every message has one of a given list of types in this type theory, then the privacy policy holds of the system as a whole. Our approach is therefore applicable to systems where we do not have access to the source code of some of the components, or where the source code changes frequently (for example, through so-called software updates).

It takes a set of conditions that we require to hold globally (which in the rest of this paper we call the *policies*) and produces a condition that can be checked locally: namely, any message that is passed between agents must have one of a certain number of allowed types. It does this without requiring information about the precise data flow within each agent or between agents. To the best of our knowledge, our approach is unique in this respect. Where some of those approaches aim at retroactively (al)-locating responsibilities for certain decisions within a system, our approach is to label the relevant actors within the system as well as the respective sets of data and proactively verify that only data flows in accordance with certain conditions are possible. The other approaches described could complement our approach. The legal ontologies for data protection could help the designer of a system decide which policies they will require globally, for example, or a technology for tracing data flows could provide technical means for guaranteeing the local condition.

We see two major distinguishing features in this kind of type-theory-based condition checking: its applicability to existing systems and its adaptability to future changes within those systems. We hope that the following explanations and example scenarios emphasise these novelties and will reflect on them in relation to the related works at the end of this paper.

Type theory is a sub-discipline of the field of Formal Methods, which investigates methods for building or analysing computer systems to provide mathematical proofs that the system satisfies certain technical properties. Aside from type theory, model checking and deductive verification are prominent approaches of formal methods, which, too, are used in theories and tools that help to satisfy certain legal requirements. Operating at different software representations or abstractions and at varying degrees of automation, different formal methods are orthogonal to each other and can in fact be combined.

m Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg, 'Verification of Information Flow and Access Control Policies with Dependent Types.' (Proceedings of the 2011 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA 2011) pp. 165-179.

n Stewart, Gordon, Anindya Banerjee, and Aleksandar Nanevski. 'Dependent Types for Enforcement of Information Flow Policies in Data Structures' (Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming 2013).

o Luísa Lourenço and Luís Caires, 'Dependent information flow types' (ACM SIGPLAN Notices 50.1 2015) pp. 317-328.

p Peixuan Li and Danfeng Zhang, 'Towards a Flow-and Path-Sensitive Information Flow Analysis' (30th IEEE Computer Security Foundations Symposium, IEEE 2017) pp. 53-67.

3. The Architecture Language

Rigorous checks require a formal language that a computer program can parse and process. We introduce a language for describing software *architectures*. An architecture may be thought of as the largest-scale design of a system. An *architecture* is a set of *agents* or *components* that may *create* pieces of data, *compute* new data from old, and *send* pieces of data to each other; together with the relations between these components.[a] In each scenario, we shall simply draw the diagram of the architecture and trust that the meaning is clear. Some technical details are given in the Appendix. The mathematical proofs will follow in a separate paper.

Every piece of data in an architecture has a *type*. Some types we specify when we describe the architecture (e.g. $name$, age, $address$). In addition, for any types A and B, there is always the type $A \to B$, which is the type of all *functions* from A to B; that is, the type of all programs or computations that take input of type A, and return output of type B.

In order to specify an architecture, we give:

- a set of *agents* or *components* (denoted by Greek letters, α, β,...)
- for each agent, a set of types; we say the agent *possesses* data of these types. (These represent the data that the agent can create without any external input.)
- for each pair of agents α and β, a set of types: we say that α may *send* data of these types to β

4. Scenarios

We will now exemplify the way our method works using four different scenarios covering some of the GDPR's main principles. We give a short introduction to those principles here and then elaborate in the respective scenarios.

Processing of personal data under the GDPR is generally prohibited unless explicitly authorised in each case (Art. 5 (1) (a), 6 (1)). Of the reasons for lawfulness described in Art. 6, *consent* by the data subject is arguably the most important.[a] Art. 7 elaborates on the specific conditions for effective consent, making it mandatory that it was given in an *informed* way, amongst others.

According to the principle of *purpose specification and limitation* in Art. 5 (1) (b), every processing of personal data needs to serve a specific purpose that the controller chooses beforehand. This purpose serves different functions: it gives the data subject the transparency necessary to understand what is being done with his or her data, especially when consenting to the processing[b]. It also limits the controller's freedom concerning later processing.[c] The reasons for rightfulness of processing as described in Art. 6 generally only legitimise acts in relation to that purpose.

The principle of *data minimisation* in Article 5 (1) (c) suggests that controllers should only collect the personal data that is necessary to fulfill this self ascribed purpose. Can the purpose be served in the same way without processing a certain set of personal data or by processing it in a less invading way, then it is not

a Our language is based on the framework known as *privacy by design* – see for example Thibaud Antignac and Daniel le Métayer (2014) 'Privacy Architectures: Reasoning About Data Minimisation and Integrity' (Security and Trust Management — 10th International Workshop, STM Sept. 2014) – , but is quite different in many respects. That system also includes proofs, attestations and spot-checks, and deals with the agents' beliefs about data.

a Eike Michael Frenzel, 'Art. 6' in Boris P. Paal and Daniel A. Pauly (eds), *Beck'sche Kompakt-Komentare Datenschutz-Grundverordnung*, (2nd edn, C.H. Beck 2018) para 10.

b Philipp Reimer, 'Art. 6' in Gernot Sydow (ed.), *Europäische Datenschutzgrundverordnung – Handkommentar*, (2nd edn., Nomos Verlagsgesellschaft, 2018) para 16.

c Philipp Reimer, 'Art. 5' in Gernot Sydow (ed.), *Europäische Datenschutzgrundverordnung – Handkommentar*, (2nd edn., Nomos Verlagsgesellschaft, 2018) para 24 et seq.

rightful.[d] Once a set of data that used to be necessary is no longer needed, it has to be deleted or pseudonymised.

4.1 Scenario A --- Tax return (Data minimisation)

An accountancy firm is hired to complete a company's tax return. To do this, they must collect certain information from the employees, including some personal data. In order to adhere to Art. 5 lit. b, only data that are necessary for completing the tax return can be used and should be sent to the firm. It should be noted from a legal standpoint that the accountancy firm in this scenario might be classified as a processor as per Article 4 Nr. 8, with the company hiring it - and therefore determining purpose and means of the processing - being the controller as per Article 4 Nr. 7. While Article 5 (2) only holds the controller accountable for adhering to the principles in Article 5, a similar motivation indirectly meets the processor as well: the controller will - and is obliged to according to Article 28 (1) - only choose those processors that are themselves implementing effective data protection measures akin to those formulated in Article 24, 25.[e] For our case, it is therefore irrelevant whether the accountancy firm acts as a processor or a controller.

We consider a greatly simplified model in which there is only one employee with two pieces of personal data, a of type A and b of type B. The datum a is required for the tax return, and b is not. We therefore seek a guarantee that the accountancy firm is never in possession of the datum b.

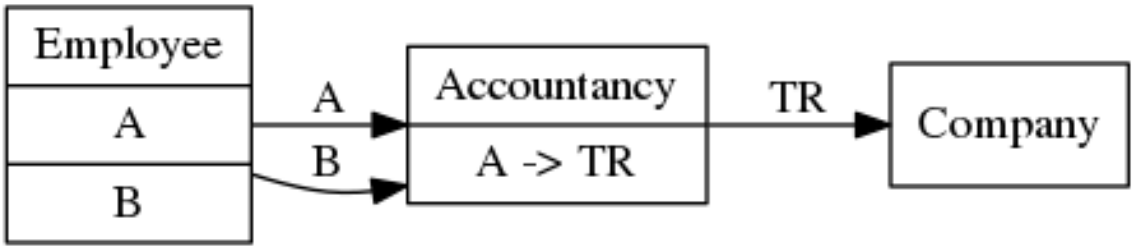

Figure 1: Tax Return Architecture

Since this is our first example, we spell out the architecture in detail. There are three agents or components, named *Employee*, *Accountancy* and *Company*. The agent *Employee* may create data of types A and B; this represents the employee creating as many copies as they want of their own personal data. The agent *Accountancy* may create a piece of data of type $A \to TR$; this represents them having a procedure for computing a tax return from a piece of data of type A. The employee may send their personal data (of type A or B) to the accountancy firm, who may send the finished tax return to the *Company*.

The privacy policy that we want in this case is:

- It is possible for the end user to receive a tax return (i.e. a piece of data of type TR).
- It is not possible for the accountancy firm or the end user to receive a piece of data of type B.

At the moment, there is nothing in the architecture that guarantees the second of these conditions. In this case, however, there is an easy way to enforce this by taking away the ability of the employee to send data of type B:

d Eike Michael Frenzel, 'Art. 5' in Boris P. Paal and Daniel A. Pauly (eds), *Beck'sche Kompakt-Komentare Datenschutz-Grundverordnung*, (2nd edn, C.H. Beck 2018) para 34 et seq.

e Mario Martini, 'Art. 24' in Boris P. Paal and Daniel A. Pauly (eds), *Beck'sche Kompakt-Komentare Datenschutz-Grundverordnung*, (2nd edn, C.H. Beck 2018) para 19.

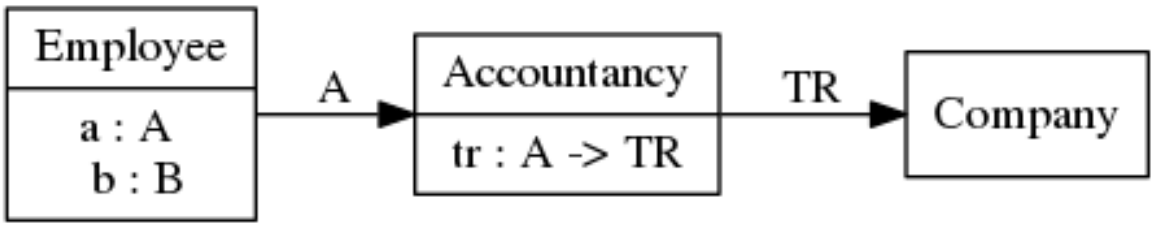

Figure 2: Safe Tax Return Architecture

4.2 Scenario B --- Tax return (Data minimisation)

Let us consider a more difficult situation. There are two kinds of tax returns that can be produced, one of which requires a and one of which requires b .

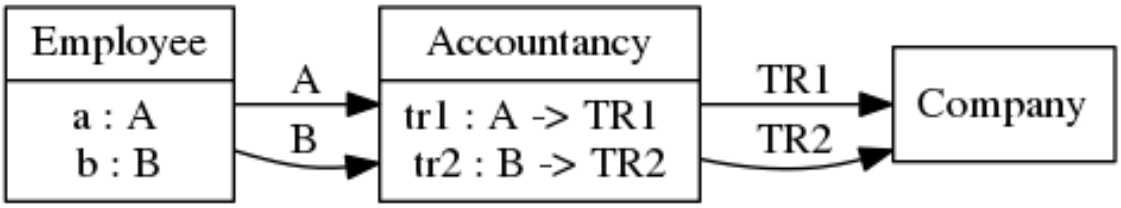

Figure 3: Another Tax Return Architecture

Article 5 (1) again stipulates that for each processing purpose (production of tax return 1 or 2, respectively) only those data that are necessary should be processed. Therefore the policy we want is:

4.2.1 Policy

- If Accountancy possesses a datum of type A , then the end user must have requested a tax return of type TR_1 .
- If Accountancy possesses a datum of type B , then the end user must have requested a tax return of type TR_2 .

We cannot get rid of any of the four possible messages without reducing the functionality of the system, so it appears we have to inspect the source code of Accountancy to have a guarantee that the policy holds. However, there is an alternative.

Consider the following architecture:

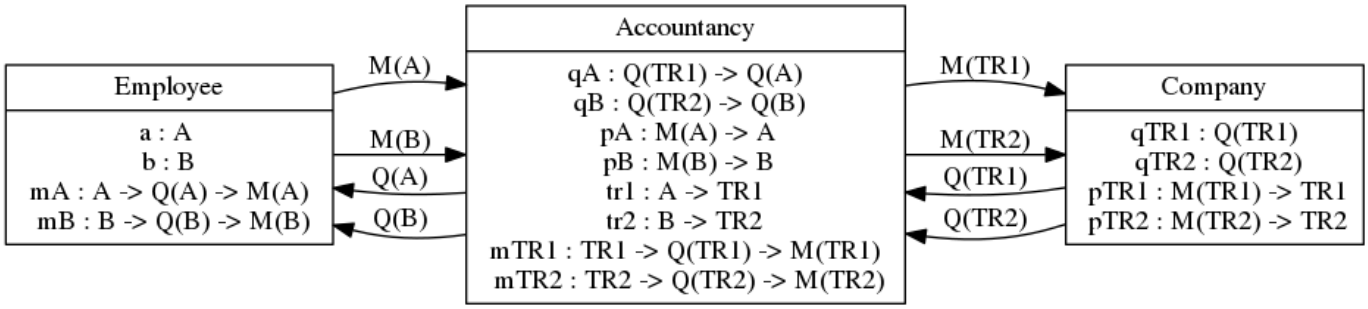

Figure 4: Another Safe Tax Return Architecture

In this architecture, we do not allow any component to send data directly. Instead, they pass *queries* and *certified messages*. A *query* is a request for a datum of a particular type; and a *certified message* consists of a query *q* requesting a datum of a particular type *A*, together with a datum *t* of that type *A*, encrypted in such a way that the datum *t* can only be read by the agent that sent the query *q*.

This allows building an architecture according to the following principles:

1. The end user may send out a query for the tax return.
2. Any other agent may only send out a query if it is for a piece of data needed to answer a query that they have received.
3. Any agent may send a piece of data only if it is to answer a query that they have received.

These principles are enforced by extending the type system as follows. For every type T in Fig. 3, we add a type $Q(T)$ of *queries* or *requests* for data of type T, and a type $M(T)$ of *messages* of type T. Note that:

- the only way to create a query for a piece of data of type A or B (that is, to create a term of type $Q(A)$ or $Q(B)$) is from a query for TR_1 or TR_2, respectively:
- the only way to create a message of type A (a term of type $M(A)$) is from a term of type A and a term of type $Q(A)$.

So we can state and prove:

'If the only messages that components can send are of type $Q(T)$ or $M(T)$, then the system as a whole satisfies the Policy above.'

In more general terms: by restricting the flow of data between the components Employee, Accountancy and End User in the way described above it becomes impossible for Accountancy or End User to gain access to data that are not necessary for the respective tax return. The system therefore integrates data minimisation as a principle right into the behavior it allows its components.

The proof is given in Section 5.3 below.

4.2.2 Implementation

The architecture in Fig. 4 works, but it requires us to modify the components from Fig. 3. For example, we must remove the ability of *Accountancy* to send and receive data of types A, B, TR_1 and TR_2, and give it the ability to send and receive data of types $M(A)$, $M(B)$, $M(TR_1)$ and $M(TR_2)$ instead.

Let us now imagine that we are given the components in Fig. 3, and cannot change them (e.g. they are services bought from a third party). Instead of changing the components themselves, we could build an *interface* to each component (see Fig. 5).

In the architecture shown in Fig. 3, the Accountancy component can receive inputs of type A and B, and give outputs of type TR_1 and TR_2. We are assuming that these details cannot be changed by us, but we can change which components the inputs come from, and which components the outputs go to.

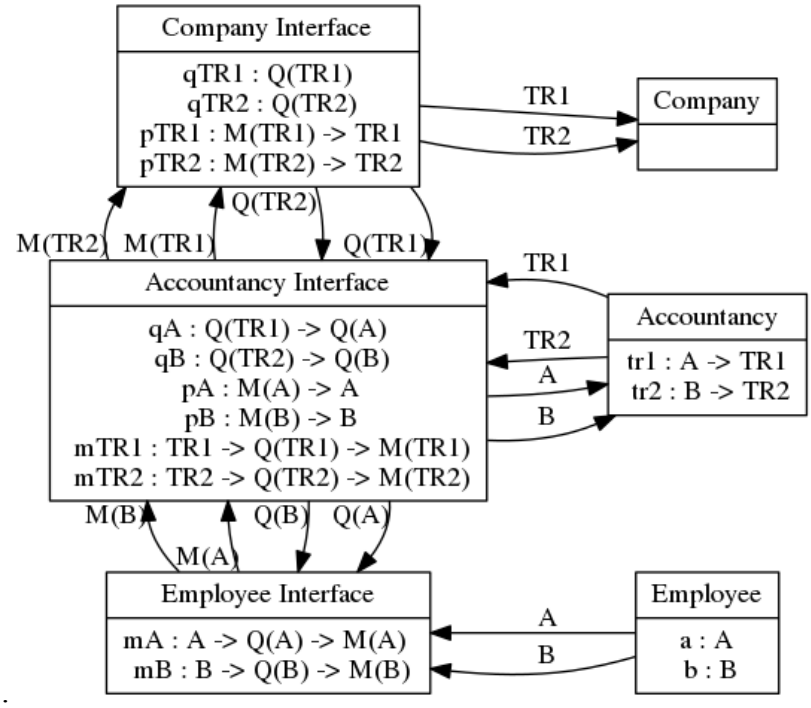

Figure 5: Implementing the Safe Architecture

For each of the three components we are given, we build a new component, called an *interface*. Each component now only communicates with its interface (that is, all inputs come from the interface, and all outputs go to the interface). The interfaces communicate with each other using only messages of type $Q(T)$ or $M(T)$.

See the Appendix for an example of how the three Interface components could be implemented.

Given the architecture shown in Fig. 5, we can prove the following property:

'It is impossible for Accountancy to have the datum a unless End User has previously sent out a query q_{TR1}, and it is impossible for Accountancy to have the datum b unless End User has previously sent out a query q_{TR2}.'

Thus, this data minimisation policy is enforced without requiring us to know anything about the inner workings of Employee or Accountancy. It furthermore shows that our method is able to work on top of and set limits for a fixed system whose components cannot or would be difficult or expensive to be changed. While the (rightful) ideal of Article 24, 25 is a system that preemptively takes data protection into account, reality - especially for companies with already well established and running systems trying to achieve compliance in time before May 2018 - might often look different. As the two figures above for scenario 2 show, our method can be applied both preemptively and subsequently.

4.3 Scenario C --- Child's consent (Article 8)

As described above, consent given by a data subject is only effective if it meets certain conditions, Art. 7. Stricter conditions apply when consent by a child is in question. Article 8 (1) therefore includes the following clause concerning the rightfulness of consent based data processing between a child and a controller offering information society services to that child:

'Where the child is below the age of 16 years, such processing shall be lawful only if and to the extent that consent is given or authorised by the holder of parental responsibility over the child.'

This means that prior to the processing of the child's data, consent must have been given by one of its parents. This consent is in turn subject to the general conditions for consent as stipulated by Article 7. Most notably the controller needs to inform the parent about its privacy policy and all relevant information concerning the processing (listed in Article 12) in order to ensure informed consent.

Last but not least, Article 7 (3) allows data subjects to revoke their consent at any time. We therefore include this here inter alia as an example of how to ensure a negative condition (the parent has *not* revoked consent) using a type system for messages.

We imagine the following setup.

Figure 6: Child's Consent --- Dangerous Architecture

At the moment, there is nothing to prevent the protected *info* being sent to the website without either *policy* or *consent* having been sent.

Using similar ideas to Scenario B, we can design an architecture that ensures that, if *Child* sends *info* to *Website*, then *Website* must have sent *policy* to *Parent*, and *Parent* must later have sent *consent* to *Website*:

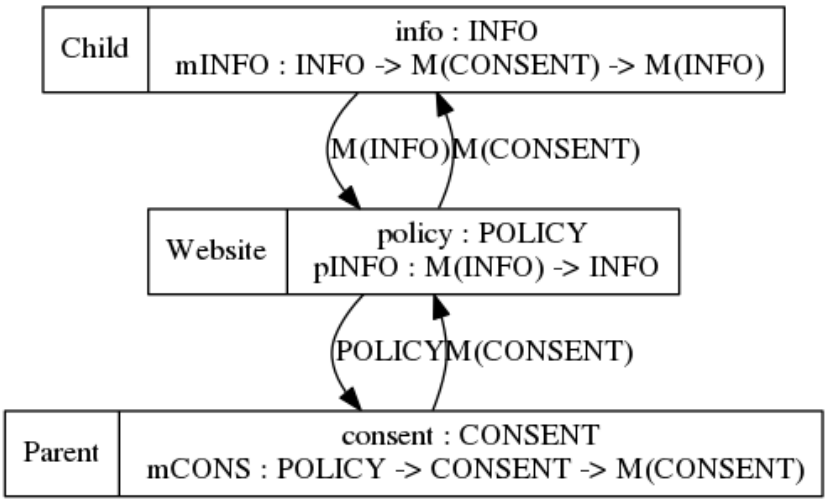

Figure 7: Child's Consent --- Safe Architecture

As before, this can be done by 'wrapping' each component from Fig. 6 in a privacy-protecting interface.

Surprisingly, it is also possible to include the condition that a revoke message has *not* been sent. This appears impossible, as it requires us to prove a negative. We prove that a message of type *CONSENT* has been sent by showing the term of type *CONSENT*, but what could count as proof that a message of type *REVOKE* has *not* been sent?

We solve this seemingly unsolvable problem as follows:

- We introduce a type of *tokens*.
- A token can only be used once.
- The first message of type *CONSENT* creates a token.

- Sending a message of type *M(INFO)* uses a token and creates a new one.
- Sending a message of type *REVOKE* uses a token and does not create a new one.

Thus, after a *REVOKE* message has been sent, no token can exist, and so a message of type *M(INFO)* cannot be sent.

To represent this, we extend our system with a new kind of datum, one which can be used only once.

Let us write $A \multimap B$ for the type of all computations which take an input of type A, return an output of type B, and *destroy* their input in the process. Thus, if $a : A$ and $f : A \multimap B$, then after we form the term $f(a)$, we cannot use a for any other purpose. In particular, if $a : A$, $f : A \multimap B$ and $g : A \multimap C$, then the terms $f(a)$ and $g(a)$ cannot both exist.

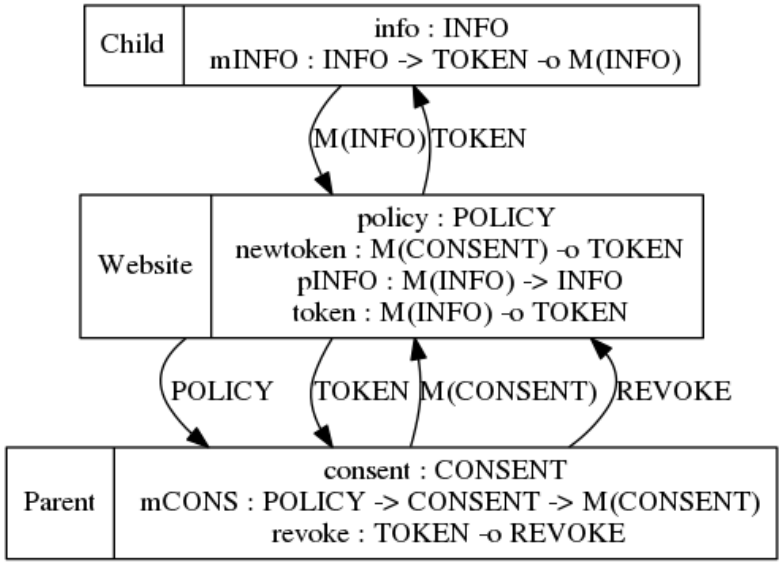

Figure 8: Child's consent --- Architecture with Revocation

We are now able to state the following theorem:

Theorem

If $\tau \vdash Website \ni t : INFO$ then τ contains an action of the form $Parent \rightarrow^{CONSENT} Website$, and does *not* contain an action of the form $Parent \rightarrow^{REVOKE} Website$.

Where $Website \ni t : INFO$ is a formal notation for "Website contains data t fo type INFO"; we explain the formal notation in Section 5.

4.4 Scenario D - Medical Research (Purpose limitation)

We explained earlier that according to the principle of *purpose limitation* the lawfulness of acts of processing only reaches as far as these acts serve a specified purpose.[f] Further processing for secondary purposes generally requires a new justification.[g] However, depending on how broad a single purpose was formulated[h], different acts of data processing can still be subsumed under it. One main challenge for controllers is therefore to correctly identify which acts of processing fall under the same purpose and to act accordingly.

f When based on a data subject's consent, this purpose is defined by the processor. When based on other legal grounds of justification, it is defined by the respective law.

g Eike Michael Frenzel, 'Art. 5' in Boris P. Paal and Daniel A. Pauly (eds), *Beck'sche Kompakt-Komentare Datenschutz-Grundverordnung*, (2nd edn, C.H. Beck 2018) para 29.

h While still within the limits of being 'specified, explicit and legitimate', Art. 5 (1) (b).

To illustrate how this principle can be enforced using our typing systems, we consider the following scenario.

A patient goes for medical treatment for a certain type of cancer. Before the treatment is begun, some personal data about him is collected that is necessary for the procedure - his medical history, the results of blood tests and other medical tests, etc. He is asked whether he wishes the data to be used only for medical treatment, or whether he consents to the data being used for medical research. If so, he may consent to it being used for cancer research, or for any form of medical research.

We can represent an architecture in which there is no safeguard ensuring purpose limitation as follows (Fig. 9). The patient may create and transmit data of type MEDICAL, representing their personal medical data. The data controller may then decide to pass them on to the patient's doctor, or to medical researchers. There is nothing in the architecture design preventing the data from being passed to a researcher who may use it for a purpose for which the patient has not given consent.

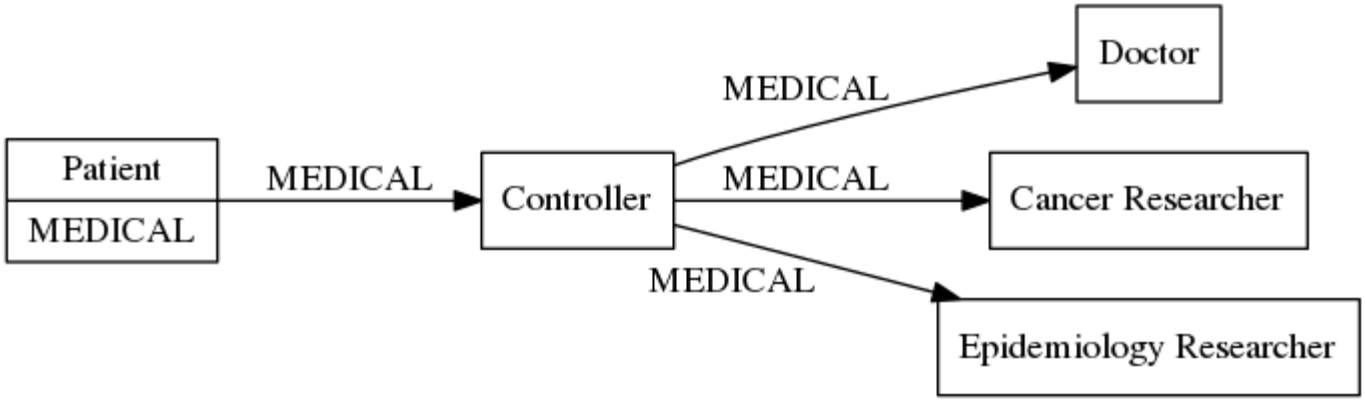

Figure 9: Medical research - dangerous architecture

In order to provide an architecture which enforces purpose limitation, consider a type system with the following types:

- TREATMENT - personal data that may be used in the medical treatment of the data subject
- RESEARCH - personal data that may be used for any form of medical research
- CANCER - personal data that may be used for cancer research
- EPID - personal data that may be used for epidemiology research

These descriptions of the types are non-exclusive - thus, if a datum has type CANCER, then we know that it is permissible to use the datum in cancer research. It may or may not be permissible to use it in epidemiology research, medical treatment, or for other purposes.

Formally, this is reflected in the fact that a datum may have more than one type. Thus, a datum may have type TREATMENT and CANCER, indicating that it is permissible to use it for the data subject's medical treatment and for cancer research.

We now introduce a *subtyping* relation between the types. We say that a type A is a *subtype* of a type B if every datum of type A is also a datum of type B. We denote this by $A \leq B$.

For this scenario, we introduce the subtyping relations $RESEARCH \leq CANCER$ and $RESEARCH \leq EPID$. These reflect the fact that, if a datum may be used for any form of research, then it may be used for cancer research; and if a datum may be used for any form of research, then it may be used for epidemiology research.

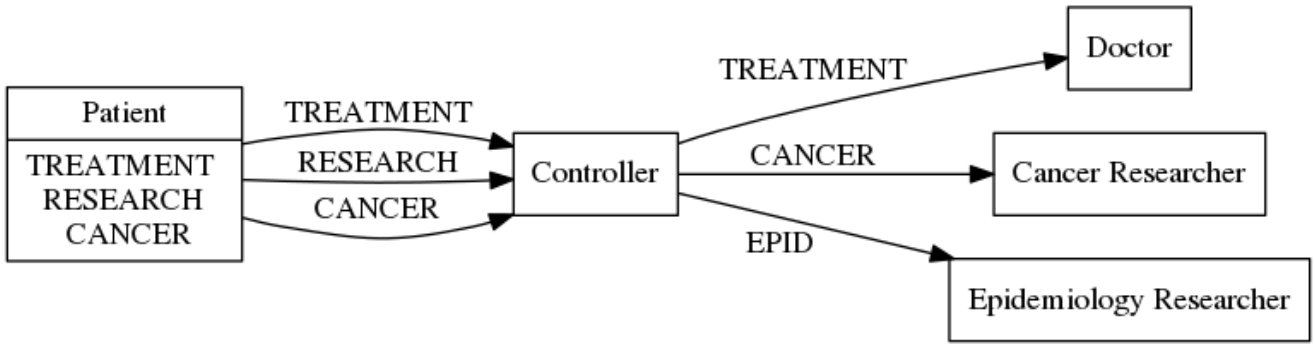

Figure 10: Medical research - safe architecture

Now, suppose in the future we wish to introduce a new category of medical research - genetic research - with the following stipulation:

- All cancer research is genetic research, but not *vice versa*.
- All genetic research is medical research, but not *vice versa*.

From this, we deduce:

- If a datum may be used for any form of medical research, then it may be used for genetic research.
- If a datum may be used for any form of genetic research, then it may be used for cancer research.

We represent this by introducing a new type into our type system:

- GENETIC - personal data that may be used for genetic research

and two new subtyping relations: RESEARCH ≤ GENETIC and GENETIC ≤ CANCER.

The new architecture shown below gives the patient four choices when his personal data is collected: it may be used for treatment alone, cancer research, genetic research, or any form of medical research.

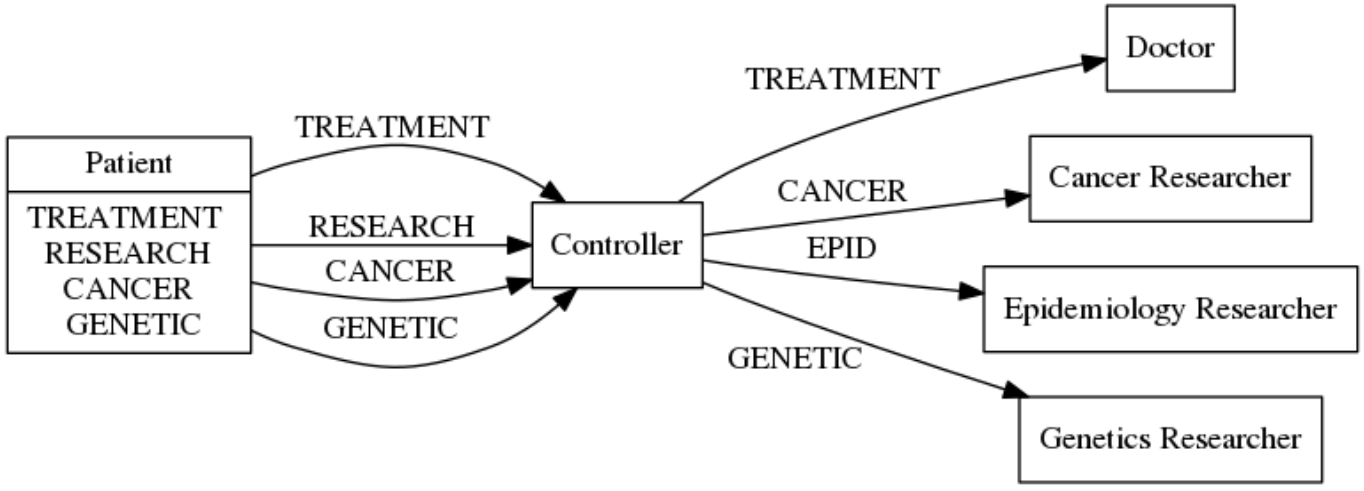

Figure 11: Medical research - safe architecture with data for genetics research

We note the following points:

1. The components of the architecture that involve the types TREATMENT, RESEARCH, CANCER and EPID are unchanged.
2. The data that was collected under the old architecture may be used immediately in the new one, without violating the new privacy policy.

Thus, imagine a hospital using the old architecture for several years, collecting a large amount of data with types TREATMENT, CANCER, EPID and RESEARCH. It should be able to adapt its system to meet the

design given by the new architecture, without having to build a whole new system from scratch. When it does so, the old data will still be usable with its old types; thus, the hospital should not need to relabel all the data in its database. We have a formally provable guarantee that we will not violate the new privacy policy with either the old or the new data (e.g. if a patient gave consent for treatment and cancer research alone in the old system, then it will not be possible to use their data for epidemiology research, or even for non-cancer-related genetic research).

This approach therefore allows new sub purposes to be introduced and defined in their relation to already existing purposes, requiring only modest changes to the software and no change to the data, in a way that preserves the guarantees of compliance with data protection regulation. This allows processors to ensure that they stay within the boundaries set through existing purposes when facing new categories and acts of processing. It also simplifies proving the sustenance of these boundaries towards authorities.

Within the context of *purpose limitation* the benefits of our approach could be utilised even further. We noted earlier that the lawfulness of acts of processing serving a specified purpose does generally not extend to secondary purposes. Art. 6 (4) limits this in one important way: processing serving a different purpose than the one formulated by the controller beforehand is still lawful if the new purpose is *compatible* with the old one. The norm gives some criteria relevant to deciding when such compatibility is indeed given, e.g. the link between the purposes, the context of the initial collection (including the relationship between data subject and controller) and possible consequences for the data subject. While it must be noted that processing for a secondary purpose should - at least where the initial processing was legitimised by the data subject's consent - remain the exception and not become the rule, since the processor could just as well have specified the initial purpose in a wider way and therefore given the subject the opportunity to give its consent[i], compatibility is nonetheless not impossible. It is furthermore worth noting that Art. 6 (4) obliges or rather allows the controller itself to ascertain whether compatibility is given or not in a specific case. On the one hand putting the decision in the controller's hands like this seems like a concession to him. On the other hand the burden of proof and by extension the risk of wrongly assuming compatibility by mistake still remains with him.[j]

A second main challenge for controllers therefore lies within correctly deciding which acts of processing serve different but nonetheless compatible purposes, and acting accordingly. Our approach as shown above can - by explicitly defining specific types and their relation as compatible or incompatible - help controllers make sure that only those new purposes that have been labeled as compatible can be used under pre existing grounds of permission.

5. Formal Description

In order to allow a computer to verify the statements we are making about the architecture, we need a precise definition of which actions are possible, and which agents have which pieces of data after a given sequence of actions. We give here a technical definition of the language used in Scenarios A, B and D above. (Scenario C makes use of linear types, which are among the advanced features of type systems and require an extension of this language, the discussion of which is beyond the scope of this paper.)

The language consists of *judgements*. An example of a judgement is

$$Employee \ni a{:}A, Employee \xrightarrow{a:A} Accountancy,$$
$$Accountancy \ni tr{:}A \to TR, Accountancy \xrightarrow{tr(a)} EndUser \vdash EndUser \ni tr(a){:}TR$$

i Eike Michael Frenzel, 'Art. 6' in Boris P. Paal and Daniel A. Pauly (eds), *Beck'sche Kompakt-Komentare Datenschutz-Grundverordnung*, (2nd edn, C.H. Beck 2018) para 49.

j Eike Michael Frenzel, 'Art. 6' in Boris P. Paal and Daniel A. Pauly (eds), *Beck'sche Kompakt-Komentare Datenschutz-Grundverordnung*, (2nd edn, C.H. Beck 2018) para 47.

On the left of the symbol $\vdash$ (which is called the *turnstile*) is a sequence of actions, called a *trace*. In this case, there are four actions: the employee creates (a copy of) a datum a of type A. The employee then sends the datum a to the accountancy component. The accountancy component creates a tax return *tr* and then sends the datum $tr(a)$ to the end user. Now, $tr(a)$ is the result of applying the calculation tr to the datum a, i.e. computing the tax return from the information provided by the employee. The datum $tr(a)$ has type TR; that is, $tr(a)$ is a tax return, which is as it should be.

On the right of the turnstile $\vdash$ is a fact about the state of the system after this sequence of events has taken place. In this case, it says that the end user is in possession of the datum $tr(a)$ (which is of type TR).

We now proceed to a complete description of all the actions and judgements that exist in our system.

An *action* is either a sequence of symbols of the form $\alpha \ni x : A$, which denotes the agent α creating a piece of data x of type A; or a sequence of symbols of the form $\alpha \rightarrow^{t:A} \beta$, which denotes the agent α sending a piece of data t of type A to the agent β. A *trace* is a sequence of actions. We use the variables σ and τ for arbitrary traces.

A *judgement* is a sequence of symbols $\tau \vdash \alpha \ni t : A$, indicating that the trace τ is a possible trace (that is, it is possible for the events in τ to occur in the given order) and, after the trace τ has finished, the agent α is in possession of the datum t, which has type A.

The system has six *rules of deduction*, which are given in Fig. 12. Each should be read as an 'if-then' statement: *if* the judgements above the line are all true, *then* the judgement below the line is true. We further stipulate that the only judgements that are true are those that can be derived using these six rules. The names of the rules are given in parentheses to the left each.

When we are given an architecture, we can apply the first three rules to any of the agents α and any of the types A which α possesses (signified by "A ∈ α" to the right of the rules). We can apply the last two rules to any of the arrows $\alpha \rightarrow^{A} \beta$ in the architecture (also signified to the right). We can apply the fourth rule without restriction.

$$(init)\frac{}{\alpha \ni x : A \vdash \alpha \ni x : A}\ (A \in \alpha)$$

$$(datum_1)\frac{\tau \vdash \beta \ni t : B}{\tau, \alpha \ni x : A \vdash \alpha \ni x : A}\ (A \in \alpha)$$

$$(datum_2)\frac{\tau \vdash \beta \ni t : B}{\tau, \alpha \ni x : A \vdash \beta \ni t : B}\ (A \in \alpha)$$

$$(func)\frac{\tau \vdash f : A \to B \qquad \tau \vdash t : A}{\tau \vdash ft : B}$$

$$(message_1)\frac{\tau \vdash \alpha \ni t : A}{\tau, \alpha \overset{t:A}{\to} \beta \vdash \beta \ni t : A}\ (\alpha \overset{A}{\to} \beta)$$

$$(message_2)\frac{\tau \vdash \alpha \ni t : A \qquad \tau \vdash \gamma \ni s : C}{\tau, \alpha \overset{t:A}{\to} \beta \vdash \gamma \ni s : C}\ (\alpha \overset{A}{\to} \beta)$$

Figure 12: Rules of Deduction. Here α and β are arbitrary agents; A, B and C are arbitrary types; x is a variable; and f, s and t are arbitrary terms.

5.1 Explanation of the Rules

We give here a detailed explanation of the rules of deduction in Fig. 12, one by one.

For the rule $(init)$: if an agent α can create pieces of data of type A , then $\alpha \ni x : A$ is a possibility for the very first action that can happen when the system begins. If this is the first action, then after it has taken place, α possesses the datum x of type A .

The rule $(datu\, m_1)$ expresses the same thing for the case where $\alpha \ni x : A$ is not the first action. If the trace τ happens and then the action $\alpha \ni x : A$ is performed, then afterwards α possesses the datum x of type A .

The rule $(datu\, m_2)$ expresses that, if an agent β (who may be α or may be a different agent) possesses the datum t before the action $\alpha \ni x : A$, then it still possesses it afterwards.

The rule (func) expresses that, given a computation $f : A \to B$ and a datum $t : A$, performing the computation f on t to obtain the datum $f(t) : B$ (short: ft: B).

The rule (message_1) expresses that, after α sends the datum $t : A$ to β , then β possesses the datum $t : A$.

The rule (message_2) expresses that, if γ possesses $s : C$ before the action $\alpha \to^{t:A} \beta$, then it still possesses $s : C$ afterwards.

5.2 Example

We show how the judgement at the beginning of this section can be derived for the architecture in Fig. 1 from these rules of deduction.

By rule $(init)$, we have

$$Employee \ni a : A \vdash Employee \ni a : A ,$$

since A is one of the possessions of $Employee$.

Applying rule $(messag\, e_1)$, we deduce

$$Employee \ni a : A , Employee \to^{a:A} Accountancy \vdash Accountancy \ni a : A$$

By rule $(d\, atu\, m_1)$, we have

$$Employee \ni a : A , Employee \to^{a:A} Accountancy , Accountancy \ni tr : A \to TR$$

$$\vdash Accountancy \ni tr : A \to TR$$

By rule $(datu\, m_2)$, we also have

$$Employee \ni a : A , Employee \to^{a:A} Accountancy , Accountancy \ni tr : A \to TR$$

$$\vdash Accountancy \ni a : A$$

Therefore, by rule $)$, we have

$$Employee \ni a : A , Employee \to^{a:A} Accountancy , Account\ ancy \ni tr : A \to TR$$

$$\vdash Accountancy \ni tr(a) : TR$$

Hence by $(messag\, e_2)$ we have

$$Employee \ni a : A, Employee \rightarrow^{a:A} Accountancy, Accountancy \ni tr : A \rightarrow TR$$

$$Accountancy \rightarrow^{tr(a):TR} EndUser \vdash EndUser \ni tr(a) : TR$$

5.3 Example

We now show how our rules of deduction can be used to formally prove the privacy policy:

If $Accountancy$ possesses the value of $a : A$, then $EndUser$ must previously have sent out a query $q_{TR1} : Q(TR1)$ for a tax return of type TR_1.

Formally, this is expressed and proven as follows:

Theorem

In the architecture of Fig. 2, if $\tau \vdash Accountancy \ni t : A$ for some trace τ and term t, then the trace τ contains an event of the form $EndUser \rightarrow^{q_{TR1}:Q(TR1)} Accountancy$ for some term q_{TR1}.

Proof

Suppose it is possible to derive $\tau \vdash Accountancy \ni t : A$. Then t must be $p_A t_1$, and $\tau \vdash Accountancy \ni t_1 : M(A)$. Therefore, τ must contain an event $Employee \rightarrow^{t_1:M(A)} Accountancy$. Let τ be equivalent to the following trace:

$$\tau = \tau_1, Employee \rightarrow^{t_1:M(A)} Accountancy, \tau_2$$

Then we must have $\tau_1 \vdash Employee \ni t_1 : M(A)$. Hence t_1 must have the form $m_A t_2 t_3$, and $\tau_1 \vdash Employee \ni t_3 : Q(A)$. Therefore, τ_1 must contain the event $Accountancy \rightarrow^{t_3:Q(A)} Employee$. Let

$$\tau_1 = \tau_3, Accountancy \rightarrow^{t_3:Q(A)} Employee, \tau_4$$

Then we must have $\tau_3 \vdash Accountancy \ni t_3 : Q(A)$. Hence $t_3 \equiv q_A(t_4)$ where $\tau_3 \vdash Accountancy \ni t_4 : Q(TR_1)$. Therefore τ_3 must contain an event $EndUser \rightarrow^{t_4:Q(TR_1)} Accountancy$, which is therefore an event in τ.

6. Conclusion

As the examples above demonstrate, our technique is capable of handling a variety of scenarios and different privacy policies when it comes to ensuring that a system is compliant with the GDPR's privacy obligations. We chose the GDPR as the relevant framework because of its high topicality and explicit openness for embracing best practices and certification schemes. Before we draw a final conclusion and paint a picture of how we envisage our technique's potential both in the context of the GDPR and other legal frameworks in today's world of Data Protection, we want to sum up and flesh out its major advantages as well as its limits.

In this spirit we first want to stress what this technique is not – namely a way of preventing fraud and deliberately malicious behaviour by controllers and processors. It does not prevent companies using it from labelling agents or pieces of data wrongly. While a lot of efforts focus on such technologies aiming at fraud detection and prevention in regards to data protection[a], we prefer to constructively offer practical application examples that can help those trying to act compliant.

We have also not specified the details of how this technique is implemented. Our approach provides a conditional guarantee: *if* only these agents possess these data initially, and only messages of these types pass

a Including the project 'Information Governance Technologies: Ethics, Policies, Architectures and Engineering' the authors are contributing to.

between agents, *then* the privacy policies are guaranteed to hold globally. We do not specify how the hypotheses of this conditional are to be ensured. This will often be a difficult problem to solve, as guarantees of properties regarding (for example) encryption are difficult to achieve. Nevertheless, it is an advantage for the designer to know which properties they are trying to ensure.

We also emphasise that our approach is intended to be implemented by the designer of a system, and not (for example) to be implemented by a data controller operating an already existing system in a particular way. In a more technical paper, two of the authors have described both how such a system may be designed, and how an already existing system could be extended to a system using our approach by adding interfaces to its components.[b]

What our approach can do, then, is twofold: make compliance easier for companies in the face of their ever-changing software systems and the inclusion of third party components while at the same time contributing to the interpretation, development and shaping of the GDPR and its conditions that are deliberately dynamic, open for interpretation and 'technologically neutral'.

On an individual level, our technique can help companies by testing their software in real time, therefore guaranteeing the integrity of privacy policies even when further developing the software. By furthermore allowing for either native integration into the software's architecture or subsequent addition on top of it, our technique is able to fulfil its purpose even when changing an already running system's components would be too difficult or expensive. This includes the highly relevant cases of software using third party services.[c]

In addition, type theory testing allows for substantial transparency towards DPAs and other invested agents with legitimate interests such as NGOs or controllers in relation to potential processors (Art. 28).[d] This can both help controllers prove and DPAs check and control compliance of a given software, while at the same time keeping publication of the content of the affected data at a minimum (see Scenario B2 above). This is especially important as conflicts between data protection principles – like accountability on the one hand and data minimization on the hand – are very common and demand for measures that promote one without restricting the other is high. Some of the similar approaches shown in Section 2 above face this exact problem.[e]

The GDPR encourages cooperation between controllers and regulators – as can be seen in Art. 35, 36, amongst others. It also encourages acts of co-regulation through codes of conducts and certification schemes, Art. 40, 41, 42. We envision that software testing through type theory could be used in such a way. Its implementation and mode of action can be examined through mathematical proofs. One example of how co-regulation could be improved through this approach can be seen in the example covered at the end of Scenario D: frequently used purposes and their respective compatibility could be pre-formulated by DPAs or other trusted authorities, thereby taking the burden of deciding on the question of compatibility at least partially off the controllers. Those would then just need to show that secondary processing for new purposes is allowed by the system's architecture only for those types labeled compatible. This would simultaneously give data subjects more legal certainty by not needing them to only trust a company's claim of compatibility.

Going further, we are confident that this technique could contribute, on a broader level, specifically to a more solid interpretation and understanding of the GDPR's conditions. Art. 25 (1) explicitly calls for organisational and technical measures that are state of the art – the scope of what is being asked of data controllers is therefore dynamically contingent on which readily available measures are most efficient as well as

b Robin Adams and Sibylle Schupp, 'Constructing Independently Verifiable Privacy-Compliant Type Systems for Message Passing Between Black-Box Components' in Ruzica Piskac and Philipp Rümmer (eds), *Verified Software. Theories, Tools, and Experiments*, (VSTTE 2018. Lecture Notes in Computer Science, vol 11294. Springer, Cham).

c See as an example for excessive data transmissions through third party tools on Smartphone apps: https://www.theguardian.com/technology/2017/nov/28/android-apps-third-party-tracker-google-privacy-security-yale-university (Accessed on 05 September 2018).

d The latter of which need to demonstrate effective safeguarding mechanisms in order to prevail against competitors.

e Jatinder Singh, Jennifer Cobbe and Chris Norval, 'Decision Provenance – Capturing Data Flow for Accountable Systems' (2018) arXiv:1804.05741 p. 6.

implementable at reasonable costs and efforts.[f] We think that our technique can further the level of current technology on all of these criteria. While many papers right now are concerned with discussing the implications and reach of these obligations on a theoretical level, we want to give a practical example of what could be state of the art at this point.

In conclusion, type theory testing is a versatile and powerful method that could potentially be used to positive effects in a variety of scenarios and legal frameworks. Its qualities lie equally in self-insurance within companies and demonstration and transparency towards other actors.

7. Acknowledgement

We thank Lies van Roessel for critical feedback. Part of this research was funded by LFF (Landesforschungsförderung) within the project "Information Governance Technologies: Ethics, Policies, Architectures and Engineering." The funders played no role in the writing of this article, or the decision to submit it for publication.

Appendix (Pseudocode)

We give here some Java-like pseudocode for the interfaces described in Fig. 5. We assume that the classes Employee and Accountancy have already been written, and have the given interfaces.

This is a naive implementation, and would not be suitable for practical purposes but rather aims at exposing the communication between components. In particular:

- It is synchronous: when an interface sends out a query, it can do nothing until the query is answered.
- It makes the personal information packaged into a message public. In practice, we would want (e.g.) MessageA.getA() to require some credential proving that the caller has the right to see that information.
- Likewise, we would want EndUserInterface to have to supply some credential in order to construct a QueryTR1 or QueryTR2 object.

```
interface Employee {
   public A getA();
   public B getB();
}

interface Accountancy {
  public TaxReturn1 getTaxReturn1(A a);
  public TaxReturn2 getTaxReturn2(B b);
}

class EmployeeInterface {
  Employee employee;

  public EmployeeInterface(Employee employee) {
    this.employee = employee;
  }

  public MessageA getMessageA(QueryA queryA) {
    A a = employee.getA();
    return new MessageA(a, queryA);
  }

  public MessageB getMessageB(QueryB queryB) {
    B b = employee.getB();
```

f Mario Martini, 'Art. 25' in Boris P. Paal and Daniel A. Pauly (eds), *Beck'sche Kompakt-Komentare Datenschutz-Grundverordnung*, (2nd edn, C.H. Beck 2018) para 39 et seq.

```
    return new MessageB(b, queryB);
  }
}

class AccountancyInterface{
  Accountancy accountancy;
  EmployeeInterface employeeInterface;

  public AccountancyInterface(Accountancy accountancy,
    EmployeeInterface employeeInterface) {
    this.accountancy = accountancy;
    this.employeeInterface = employeeInterface;
  }

  public MessageTR1 getMessageTR1(QueryTR1 queryTR1) {
    QueryA queryA = new QueryA(queryTR1);
    MessageA messageA = employeeInterface.getMessageA(queryA);
    A a = messageA.getA();
    TaxReturn1 taxReturn1 = accountancy.getTaxReturn1(a);
    MessageTR1 messageTR1 = new MessageTR1(taxReturn1, queryTR1);
    return messageTR1;
  }

  public MessageTR2 getMessageTR2(QueryTR2 queryTR2) {
    QueryB queryB = new QueryB(queryTR2);
    MessageB messageB = employeeInterface.getMessageB(queryB);
    B b = messageB.getB();
    TaxReturn2 taxReturn2 = accountancy.getTaxReturn2(b);
    MessageTR2 messageTR2 = new MessageTR2(taxReturn2, queryTR2);
    return messageTR2;
  }
}

class EndUserInterface{
  AccountancyInterface accountancyInterface;

  public EndUserInterface(AccountancyInterface accountancyInterface) {
    this.accountancyInterface = accountancyInterface;
  }

  public TaxReturn1 getTaxReturn1() {
    QueryTR1 queryTR1 = new QueryTR1();
    MessageTR1 messageTR1 = accountancyInterface.getMessageTR1(queryTR1);
    return messageTR1.getTaxReturn1();
  }

  public TaxReturn2 getTaxReturn2() {
    QueryTR2 queryTR2 = new QueryTR2();
    MessageTR2 messageTR2 = accountancyInterface.getMessageTR2(queryTR2);
    return messageTR2.getTaxReturn2();
  }
}

class QueryA{
  public QueryA(QueryTR1 queryTR1) {
  }
}

class QueryB{
  public QueryB(QueryTR2 queryTR2) {
  }
}
```

```
class QueryTR1{
  public QueryTR1() {}
}

class MessageA{
  A a;

  public MessageA(A a, QueryA queryA) {
    this.a = a;
  }

  public A getA() {
    return a;
  }
}

class MessageB{
  B b;

  public MessageB(B b, QueryB queryB) {
    this.b = b;
  }

  public B getB() {
    return b;
  }
}

class MessageTR1{
  TaxReturn1 taxReturn1;

  public MessageTR1(TaxReturn1 taxReturn1, QueryTR1 queryTR1) {
    this.taxReturn1 = taxReturn1;
  }

  public getTaxReturn1() {
    return taxReturn1;
  }
}

class MessageTR2{
  TaxReturn2 taxReturn2;

  public MessageTR2(TaxReturn2 taxReturn2, QueryTR2 queryTR2) {
    this.taxReturn2 = taxReturn2;
  }

  public getTaxReturn2() {
    return taxReturn2;
  }
}
```

Contact Details

Robin Adams, Chalmers University of Technology, Department of Computer Science and Engineering, Rännvägen 6, 41266 Gorthenborg, Sweden. Email: robinad@chalmers.se. Tel.: +768564864

Wolfgang Schulz, Hans-Bredow Institute for Media Research, Department of Law, Rothenbaumchaussee 36, 20148 Hamburg, Germany. Email: w.schulz@hans-bredow-institut.de. Tel.: +49 40 450217-0

Sibylle Schupp, Hamburg University of Technology (TUHH), Institute for Software Systems, Department of Electrical Engineering, Computer Science, and Mathematics, Schwarzenbergstraße 95, 21073 Hamburg, Germany. Email: schupp@tuhh.de. Tel.: +49 40 42878-3460

Florian Wittner, Hans-Bredow Institute for Media Research, Department of Law, Rothenbaumchaussee 36, 20148 Hamburg, Germany. Email: f.wittner@hans-bredow-institut.de. Tel.: +49 40 450217-44