Karlsruhe Institute of Technology

Software Design and Quality

Automated Consistency of Legal and Software Architecture System Specifications for Data Protection Analysis

Bachelor Thesis of

Tom Hüller

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr. Anne Koziolek
Advisor:	M.Sc. Nicolas Boltz
Second advisor:	M.Sc. Sebastian Hahner

12. June 2023 – 12. October 2023

Abstract

The years 2021 and 2022 saw the highest number of data breaches ever reported in the United States [8]. Data breaches exposing personal information mean a significant loss of customer trust and leave companies vulnerable to civil lawsuits. Many countries have also started levying severe fines in the event that protected personal data is revealed. The cost of fixing issues that could potentially cause a data breach rises exponentially the longer it takes to find and fix the mistake [9]. This makes identifying problems during the design phase an important part of keeping software development costs predictable and manageable.

In 2018, the European Union passed the General Data Protection Regulation (GDPR), which imposes obligations on every organization that collects data on EU citizens. Article 25 of the GDPR requires Data Protection by Design (DPbD) and by Default. DPbD means data protection issues must already be considered during early software development phases and not handled later in the development process. By Default means data protection measures are automatically applied without the need for user input. [33]

The GDPR focuses heavily on protecting citizens' rights to their own data; however, the exact regulations put in place are cumbersome, hard to understand, and even harder to prove compliance with in a reliable way.

In this thesis, we present approaches that allow system architects to extract legal specifications from artifacts created during system design and analyze them for GDPR compliance. Seifermann et al. [29] developed an extended Data Flow Diagram metamodel that allows the modeling of information flows and access control. We provide a model transformation between this metamodel and a GDPR metamodel developed by Boltz, which aims to model some of the complex requirements of the GDPR. The transformations work in both directions while keeping additional information to allow the architect to make changes to the system on either the architectural or legal side of the transformation. In their original publication, Seifermann et al. [29] also presented an analysis tool for their metamodel. We provide an analysis tool that is able to identify GDPR violations on the GDPR metamodel, allowing analysis on both sides of the transformation.

Zusammenfassung

In den Jahren 2021 und 2022 wurden in den USA die meisten Datenlecks bisher gemeldet [8]. Datenlecks, bei denen private Daten öffentlich zugänglich werden, bedeuten einen großen Verlust an Kundenvertrauen zusätzlich zu eventuellen Klagen. Viele Länder verhängen empfindliche Strafen, sollten persönliche Daten offengelegt werden. Die Kosten ein potentielles Datenleck zu reparieren steigen exponentiell, je länger es dauert, das Problem zu reparieren [9]. Dies führt dazu, dass Problemerkennung in der Entwurfsphase ein essenzieller Teil ist, um die Entwicklungskosten gering und überschaubar zu halten.

In 2018 hat die Europäische Union die Datenschutz-Grundverordnung (DSGVO) erlassen. Diese Verordnung stellt Anforderungen an alle Organisationen, die Daten von europäischen Bürgern sammeln und verarbeiten. Artikel 25 verlangt Datenschutz im Entwurf (DPbD) und als Standard. DPdD bedeutet, dass datenschutzrechtlich relevant Probleme bereits in der Entwurfsphase bedacht werden müssen, während als Standard bedeutet, dass Datenschutzmaßnahmen standardmäßig und ohne Einfluss des Nutzers angewendet werden. [33]

Die DSGVO ist streng auf den Schutz der Daten von EU-Bürgern fokussiert, allerdings sind die genauen Anforderungen kompliziert, schwer zu verstehen und noch schwerer zuverlässig nachzuweisen.

In dieser Arbeit stellen wir Ansätze vor, die es Softwarearchitekten ermöglichen, rechtliche Spezifikationen aus Artefakten zu extrahieren, die im Systementwurf entstanden sind. Seifermann et al. [28] haben ein erweitertes Datenflussdiagram Metamodel entwickelt, das das Modellieren von sowohl Datenflüssen als auch Zuganskontrolle erlaubt. Wir erstellen eine Modelltransformationen zwischen diesem Metamodel und einem DSGVO Metamodel von Boltz, welches erlaubt einiger der komplexen DSGVO Vorgaben zu modellieren. Die Transformationen funktionieren in beide Richtungen und speichern zusätzliche Informationen, um es dem Architekten zu ermöglichen, Änderungen am System auf beiden Seiten der Transformation vorzunehmen. In der urspünglichen Publikation stellen Seifermann et al. [29] auch ein Analysewerkzeug für ihr Metamodel vor. Wir stellen zusätzlich ein Analysewerkzeug zur Verfügung, welches DSGVO Verletzung auf dem DSGVO Metamodel identifizieren kann, um eine Analyse auf beiden Seiten der Transformation zu gewährleisten.

Contents

List of Figures

1 Introduction

With the introduction of the General Data Protection Regulation (GDPR), the European Union aimed to protect the right of its citizens to have control over all data collected about them. The GDPR imposes regulations on when and how data can be collected, how processing is to be performed with valid purposes and legal basis, and how data subjects can demand access to or the correction and deletion of the data collected about them. The GDPR allows regulators to levy heavy fines against companies and individuals who do not comply with GDPR regulations. According to CMS Hasche Sigle's enforcement tracker [25], about 4.4 billion euros in fines have been levied since the GDPR was passed into law. Their statistics show that rulings against GDPR regulations are decreed approximately every other day. A closer look, however, shows that many of these rulings only carried small fines and were made against private individuals or small companies. It can be hard for small companies without a large team of legal experts to understand the multitude of regulations imposed upon them by the GDPR.

In their paper "I'm all ears! Listening to software developers on putting GDPR principles into software development practice" Alhazmi et al. [1] analyze the responses of 20 software developers on why they struggle to include GDPR regulations in their implementation. More than half of the surveyed developers answered that they are not familiar enough with the GDPR and the techniques to implement it. Additionally, they complain about a lack of resources and guidelines on how GDPR regulations can be incorporated.

Model-Driven-Engineering [18] is one approach on how to visualize complex system dynamics and requirements. Models reduce the complexity of system design by allowing the developer to work on a higher abstraction level. Additionally, the artifacts created during Model-Driven-Engineering allow for high reusability, repairability, and intelligibility during system design and later phases of the development pipeline.

1.1 Contribution

We propose an approach that allows software architects to quickly transform and analyze artifacts created during system design for GDPR violations without extensive background knowledge of data protection law. First, we provide a consistent model transformation between a data flow diagram model instance and an instance of a GDPR-specific metamodel. This allows software architects to make changes to the system on both the architectural and legal sides. Secondly, we provide an analysis tool that can identify violations of GDPR regulations on the GDPR-specific metamodel. This allows software architects to find violations quickly and autonomously without extensive knowledge of data protection law.

Besides implementing the transformation and analysis tool, we aim to answer the following research questions as part of this thesis:

RQ1 How can instances of either model be transformed into the other without loss of information?

RQ2 What kind of GDPR violations can be modeled and found in the metamodel?

To answer question **RQ1**, we first analyze both models and try to find elements that are semantically equivalent and can be directly transformed into each other. Next, we identify model elements that can hold additional information about elements that cannot be directly transformed into an equivalent element. For elements that can neither be directly transformed nor stored in another element, we aim to provide ways of referencing previous transformations to restore the information.

To answer question **RQ2**, we extract all violations from the GDPR that we can model in the metamodel with our limited knowledge of data protection law. The analysis tool will then be designed to include all these violations.

1.2 Outline

The remainder of this thesis is structured as follows: Chapter 2 covers the foundations for the proposed thesis topic. In Chapter 3, we discuss related works in the fields of model-driven security and model transformations. In Chapter 4, we present our running example that will be employed throughout this thesis. In Chapter 5, we present the design and implementation of our analysis tool approach. Chapter 6 covers an analysis of both metamodels for similarities and the resulting implementation of the transformations. In Chapter 7, we present and discuss the results of the evaluation of our approaches. Finally, in Chapter 8, we will conclude this thesis and provide an outlook on possible future contributions.

2 Foundations

In the following chapter, we present the foundations necessary for explaining this thesis. In Section 2.1, we give a basic introduction to Model-Driven Software Development, while in Section 2.2 we give a short overview of data privacy. Section 2.3 covers Data Flow Diagrams (DFD), with special emphasis on the extended DFD Syntax by Seiferman et al. [29]. In Section 2.4, we present the legal foundations for our contribution. First, we give a short overview of the GDPR and then present the GDPR metamodel by Boltz. Section 2.5 covers model transformations and we give a short overview of the model transformation languages we considered for our contribution.

2.1 Model-Driven Software Development

Model-Driven Software Development (MDSD) [31] is one approach to designing bigger and more complex software systems. Models allow a comprehensive view of the system and its behavior before it is even implemented. Through the use of models, developers and other stakeholders can analyze and correct the software system early in the development cycle, thus significantly reducing the cost and effort to fix those errors. [9] Additionally, models that are known to work can be reused in other systems, reducing the amount of work needed for system design.

Metamodels define the concepts and rules for the creation of a model and how elements of that model interact with each other. In other words, they are the model of a model. They also allow the addition of required features to other existing models.

2.2 Data Privacy

Data privacy refers to the set of security policies and measures implemented to protect personal data from unauthorized access, processing, or disclosure. Personal information is required to be collected, processed, and kept in a transparent manner abiding by local laws. We are going to focus on data confidentiality, as it is the aspect of data privacy most important for this work. Data can be kept confidential via a multitude of approaches called confidentiality mechanisms. These include access control, encryption, information flow, or anonymization. [20]

2.3 Data Flow Diagrams

Data Flow Diagrams (DFD) are a visual representation of data flows through an information system. DFDs as originally described by DeMarco [10] consist of 4 elements. *Actors*, *processes*, *data stores*, and *data flows*. *Actors* are external entities that exist outside of system boundaries and act as data sources and sinks. *Processes* perform functions and transformations on the data entering them. *Data stores* represent repositories that save data inside of the system. *Data flows*, represented by vectors, are the pathways data takes from any of the former three to their destination.

2.3.1 Data Flow Confidentiality Metamodel

DFDs, as originally described by DeMarco, work for modeling all data flows in a system. However, they lack certain features necessary for guaranteeing data confidentiality. Seifermann et al. [29] analyze the original model as well as multiple extended DFD syntaxes. They identify three challenges the model needs to overcome to allow automated analysis.

The authors postulate that realistic applications often have nodes that receive the same type of data from multiple different paths. To detect possible issues, all valid paths need to be analyzed. Since not every possible path is also a valid path, they argue that it must be possible to specify all valid path combinations in the model.

They claim that DFDs are usually extended to allow confidentiality analysis depending on a single confidentiality mechanism. Those single-purpose models lack the flexibility to change the mechanism. The authors claim that existing analysis tools need extensive remodeling to change to confidentiality mechanism, which opens the issue of consistency.

To model realistic requirements for a system, e.g., excluding one specific node from accessing specific information, their model needs to work with custom confidentiality analysis as defined by the designer. [29]

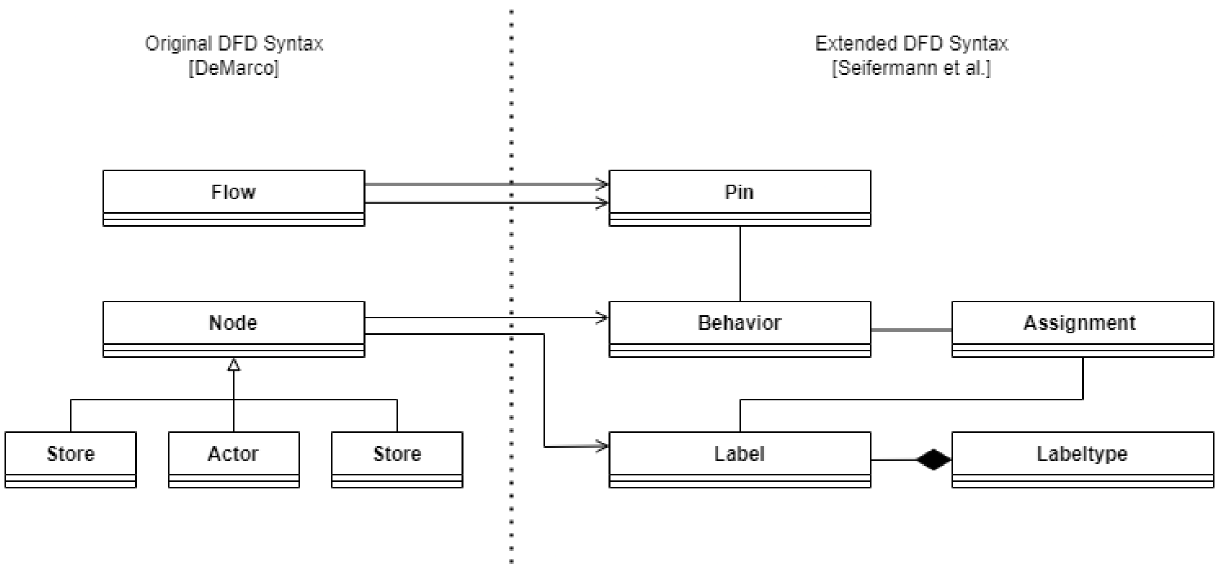

Figure 2.1: UML Diagram of Extended DFD-Metamodel by Seifermann et al.

To fulfill these requirements, the authors designed their own extended DFD syntax. Figure 2.1 shows a UML diagram of their extended DFD syntax. First, they introduce support for *Node* characteristics by adding *Labels* to every *Node* as properties. *Labels* are discrete values that represent properties like roles or classification levels. Subsequently, they add the concept of *Pins* to *Nodes*. A *Pin* can either represent a required input or an output. All data *Flows* need to start and end in a *Pin*. A *Node* can have multiple input *Pins* representing multiple required inputs. Multiple data *Flows* to or from the same *Pin* meaning multiple possible paths data can take to reach its destination. Lastly, they introduce *Behaviour* to *Nodes*. *Behaviours* visualize which data enters and leaves the *Node*. *Behaviours* contain the output and input *Pins* as well as *Assignments*, which specify which *Labels* are propagated by the *Node* when different *Labels* arrive at the input *Pins*. [29]

2.3.2 DFD Analysis

Together with the extended data flow diagram metamodel, Seifermann et al. [29] present a logic program that allows for the detection of confidentiality violations on DFD metamodel instances. The analysis tool works by building a label propagation network. *Nodes* get mapped to *Label* propagation functions, while properties get evaluated as *Labels*. After the *Label* propagation, a logical program written in Prolog is used to detect violations by finding edges with a higher classification level than the clearance level of the *Node* that receives the data.

The Prolog implementation of this analysis has since been deprecated, and a new Java-based analysis tool was presented. [27] The Java-based analysis tool currently only works for Palladio-based models, but the integration of the DFD metamodel is in progress.

2.4 Legal Foundations

This section covers the legal foundation for our contributions. First, we give an overview of the GDPR and its regulations. Then we present the GDPR metamodel developed by Boltz that aims to simplify modeling these regulations.

2.4.1 GDPR

This work is focused on the General Data Protection Regulation passed by the European Union in 2018. The GDPR requires every entity collecting data on EU citizens to comply with a set of regulations. It differentiates between controllers, who determine which and how data is processed, and processors, who process data on the controller's behalf.

The controller is responsible for ensuring compliance with the following basic principles:

Personal data is required to be collected fairly and transparently and is only allowed to be collected for a specific purpose. The collected data needs to be limited to data relevant to the aforementioned purpose and, in case it permits personal identification, is only allowed to be stored until the purpose is fulfilled. The controller must also take every practical measure to ensure that the collected data is accurate and, if required, up to date. Finally, the GDPR requires data to be processed in a secure manner that prevents data from unauthorized processing and data loss.

Processing of personal data is only lawful if the data subject has given their permission either explicitly through consent, or implicitly, e.g., through participation in a contract for which data processing is necessary. In the case of public interest or to protect the vital interests of the data subject, data may also be processed. [33]

2.4.2 GDPR Metamodel

The requirements listed in the GDPR are difficult for many software architects to understand. [1] Boltz developed a GDPR metamodel to allow modeling some of the requirements. This allows for quicker compliance analysis and reusability of previously generated models for further projects.

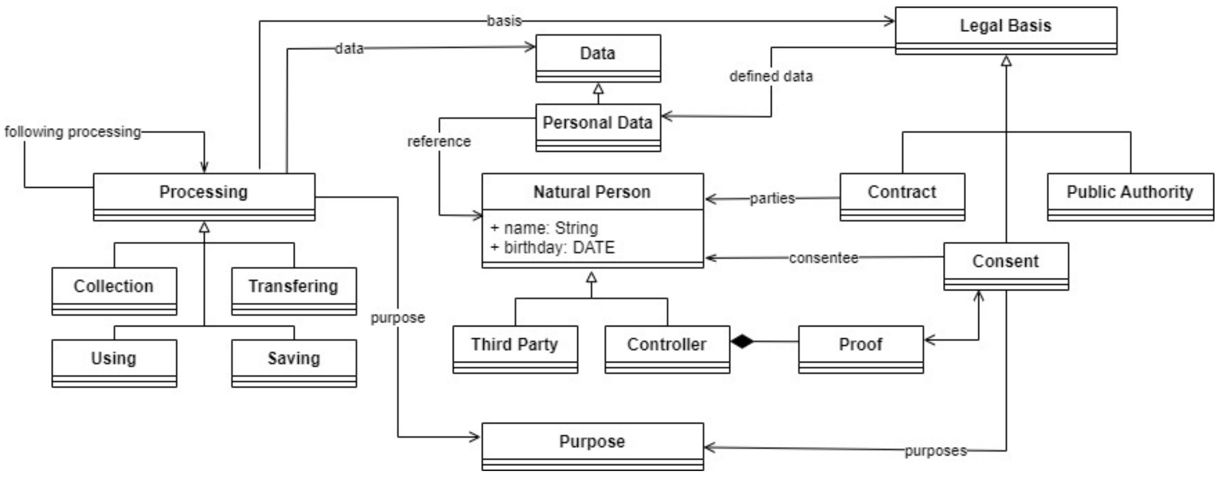

Figure 2.2: UML Diagram of GDPR Metamodel by Boltz

A UML diagram of the GDPR metamodel can be seen in Figure 2.2.

In the center of the model is the *Natural Person*, which can be either a third party, e.g., the data subject, or a *Controller*. *Personal Data* has a reference to the *Natural Person* and is itself referenced by the *Legal Basis* on which the data has been collected. A *Legal Basis* can either be the *ExerciseOfPublicAuthority*, the *PerformanceOfContract*, for which the *Natural Person* is a party, or explicit *Consent*. *Consent* requires a defined *Purpose*, a consentee, and *Proof* of *Consent* needs to be kept by the *Controller*. *Processing*, which includes *Collecting*, *Transferring*, *Usage*, and *Saving*, also requires a *Purpose* that can be set by a *Controller*. Also required is a *Legal Basis* on which the *Processing* is performed.

2.5 Model Transformations

There are several different methods and languages that allow model-to-model transformations. In the following sections, we will take a look at three different ways of achieving model transformations. Imperative approaches give a step-by-step guide on how the program state needs to change. Declarative approaches describe which goal the program needs to accomplish without specifying how. Lastly, direct manipulation allows for working directly on model instances.

2.5.1 QVT

The Meta Object Facility 2.0 Query/View/Transformation specification [22] published by the Object Management Group provides three different languages for model transformation:

Relations, Core, and Operational Mappings. The two former languages are of a declarative nature and provide support for black-box approaches. Since that would add unnecessary complexity, we will only focus on the Operational Mappings language.

The Operational Mappings language works in an imperative manner by specifying transformations. Transformations unidirectionally denote how to transform the source model into the target model. The transformation is performed through mappings, which describe how one or multiple elements of the source model are transformed into elements of the target model. Mappings consist of optional pre- and postconditions and a mapping body. In the mapping body, attributes of the source element can be used to create or update the target element. Mapping operations can also call other mapping operations to further transform attributes of the source element. The execution of mapping operations creates a trace record linking the source element and the target elements for further access.

2.5.2 Vitruv Framework

Klare et al. [19] present a whole suite of model transformation tools with their Vitruv Framework. However, we will focus on two domain-specific languages, Commonalities and Reactions, since they are the most suitable for keeping the model transformations consistent.

2.5.2.1 Reactions Language

The Reactions language allows for defining consistency relations imperatively, which describes how a change in one model needs to be transformed into the target model. Those relations work unidirectionally and change-driven. The language consists of two concepts: reactions and routines.

Reactions specify when consistency repair is necessary after a change by defining the trigger and routines. The routines will then be executed should the trigger be fulfilled. It is possible to have multiple reactions to the same change; however, execution order cannot be influenced at this point.

Routines specify how model consistency can be repaired after a change. They comprise a match, a create, and an update part. In match, preconditions are specified, and information necessary to fulfill the preconditions is retrieved. Should the preconditions prove impossible to fulfill, the update part won't be executed. Create describes which new elements need to be created, as well as their type and variable name. Update specifies which changes need to be performed to the target model to maintain consistency. Elements that were retrieved or created in the former parts can be used here.

2.5.2.2 Commonalities Language

The Commonalities language [13], in contrast, works bidirectionally and in a declarative manner through the definition of commonalities between different metamodels. A commonality describes a common concept, like classes or methods, shared by two or more metamodels. A set of one or more commonalities makes up a concept, which describes a conceptual idea, like object-oriented design, shared between the metamodels. Concepts themselves can technically be seen as a metamodel, with commonalities representing metaclasses.

A commonality consists of up to 3 elements: participations, attributes, and references. Participations describe a set of metaclasses, mapping instances of those metaclasses to instances of the commonality. It is possible to add restrictions that instances of metaclasses need to fulfill in order to be considered instances of the commonality. Commonalities can contain attributes, which in turn are mapped to attributes of the participation classes. This mapping can be done from the commonality to participation classes, from participation classes to the commonality, or bidirectionally. Optionally, attributes can be checked and converted before propagation. Additionally, a commonality can reference other commonalities, such as sub- or superclasses.

2.5.3 Java

It is also possible to implement the transformation in Java without a dedicated transformation language. Java can load the Ecore model of the source metamodel and create a model instance of the target metamodel. Then the source model instance can be parsed, and the equivalent elements of the target metamodel can be created and added to the instance. To save traces between source and target model elements, a separate metamodel has to be implemented. Java is as powerful a transformation language as the previously described languages; however, most steps need to be implemented by hand.

3 Related Work

This chapter gives an overview of the related work for this thesis. We divide this chapter by area of research. In Section 3.1, we present the concept of model-driven security. Additionally, we present extended DFD and GDPR metamodel syntaxes different from the metamodels defined by Seifermann et al. [29] and Boltz. Section 3.2 covers related work in the field of model transformation. First, we present general transformation mechanisms and then discuss a specific transformation between extended DFD syntaxes.

3.1 Model-Driven Security

Model-Driven-Security is a specialized form of the MDSD paradigm. The paradigm aims to model security-related aspects during design time. In their paper "Model Driven Security for Process-Oriented Systems", Basin et al. [6] mint that term and show how their extended UML syntax allows them to precisely model complex security policies.

3.1.1 Extended DFD-Syntax

Antignac et al. present two extended DFD syntaxes intended to model privacy requirements: the Business-Oriented DFD (B-DFD) [3] and the Privacy-Aware DFD (PA-DFD) [2] syntax. The syntax includes an additional data deletion flow that can be sent to data stores to delete previously stored information. The PA-DFD syntax is more extensive than the B-DFD syntax and aims to include annotations that allow architects to consider privacy principles during design. The authors add Data Subjects, Data Controllers, and Data Processors to the original syntax and annotate dataflows with the corresponding data subjects. Their approach is strongly linked to the GDPR and allows for modeling some of the complex GDPR requirements.

3.1.2 GDPR-Metamodels

Tom et al. [32] developed a GDPR metamodel, different from Boltz's model. Their model, while modeling GDPR entities in a similar way, focuses on additional GDPR regulations not modeled in the model by Boltz. The right of a data subject to delete or rectify their data, as well as tighter regulations for privileged data (religious beliefs, sexual preference, etc.), are included in this model. In contrast, Boltz's metamodel is restricted to normal processing of unprivileged data. The same authors also introduce an analysis method for their metamodel. [21] The analysis is designed to work on business processes by extracting an AS-IS compliance model and comparing that to a fully compliant GDPR model.

Huth [17] presented a more high-level modeling approach in his dissertation. His tool, ProPerData, can provide a comprehensive overview of GDPR-related issues and create work unit descriptions and technical overviews. However, the model is restricted to systems built in enterprise architecture and does not work with model elements taken directly from the GDPR but with higher-level abstractions more suited for the enterprise architecture approach.

Sion et al. [30] introduced a metamodel design to comply with both GDPR and the older WP29 [14] requirements. To streamline Data Protection Impact Assessments, they combine their Data Protection View (DPV) with original DFDs to model compliance scenarios more accurately. Model constraints as well as the increased input from stakeholders allow for the combination of legal matters with software engineering approaches. They explore ways to integrate DFD model instances into the DPV through correspondence rules. That allows them to keep DPV model instances consistent with DFD instances.

3.2 Model Transformation

Ruscio et al. [11] take a comprehensive look at model transformations in their paper. They classify transformation approaches into specific categories. Direct manipulation, operational, and relational approaches have already been covered in Section 2.5. Additionally, they differentiate between hybrid, graph-based, and rule-based approaches. The authors give an overview and discuss multiple different transformation approaches, including QVT. They demonstrate two different application scenarios of transformation problems: change propagation and model evolution, and show how model transformation can be successfully applied in these scenarios.

In their paper on B-DFDs, Antignac et al. [2] also present a model transformation between B-DFDs and another Privacy-Aware DFD syntax. While referencing the GDPR, the authors identify six points of interest called hotspots. They define five subtypes of processing nodes and transform each hotspot into a corresponding sequence of subtypes. The *Limit* subtype, for example, ensures that data only reaches processing nodes that operate on a purpose to which the subject has previously given consent. Knowledge of these consents is passed via a policy that can be queried from another subtype called *Request*. This allows the authors to extract complex security scenarios from quasi-normal architectural artifacts. In further works, they intend to extend and prove the GDPR compliance of their approach.

4 Running Example

In this chapter, we will introduce an example scenario that will from here on out be used as a running example. In this scenario, company X is *Collecting*, *Processing*, and *Saving* user location data based on different *Legal Bases.* The scenario has been designed to incorporate all possible elements of the GDPR metamodel and is comprised of the following users:

- Joe: Joe has consented to the usage of his data in order to use company X's services.
- Alice: Alice is using an app that uses company X's services in the background. She has therefore entered into a contract with company X that allows for the usage of her data.
- Mallory: Mallory is a criminal. A judge has ruled that company X is to collect information from Mallory and hand it over to the authorities.

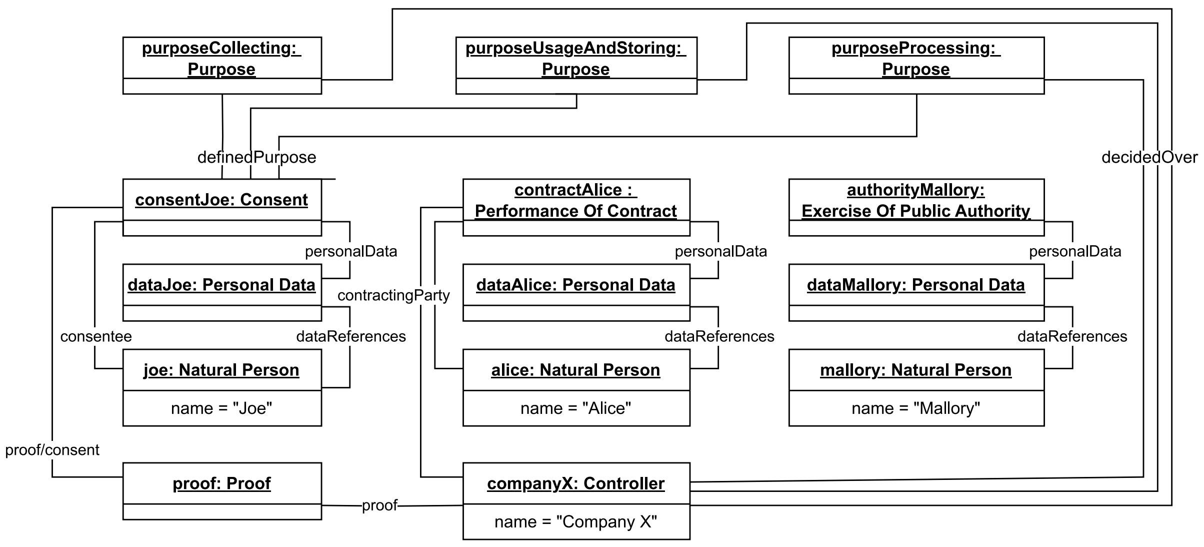

Figure 4.1: UML Object Diagram of Running Example Without Processing

Figure 4.1 shows all three individual *Natural Persons* as well as the associated *Personal Data.* It also lists the individual *Legal Bases* that are necessary for the usage of the users' data. Joe's *Consent* needs a *Proof* kept by a *Controller*, in this case, company X, who also needs to be part of *contractingParty* of Alice's *PerformanceOfContract.* Figure 4.1 also shows the *Purposes* that will be later annotated to the *Processing* elements and are decided over by the *Controller.* Joe's *Consent* is also defined on all *Purposes.*

Figure 4.2 now shows the individual *Processing* elements. Again, the scenario was designed to include all possible subtypes of *Processing.* After the collection, the data is simultaneously used and processed, and then transferred to a database.

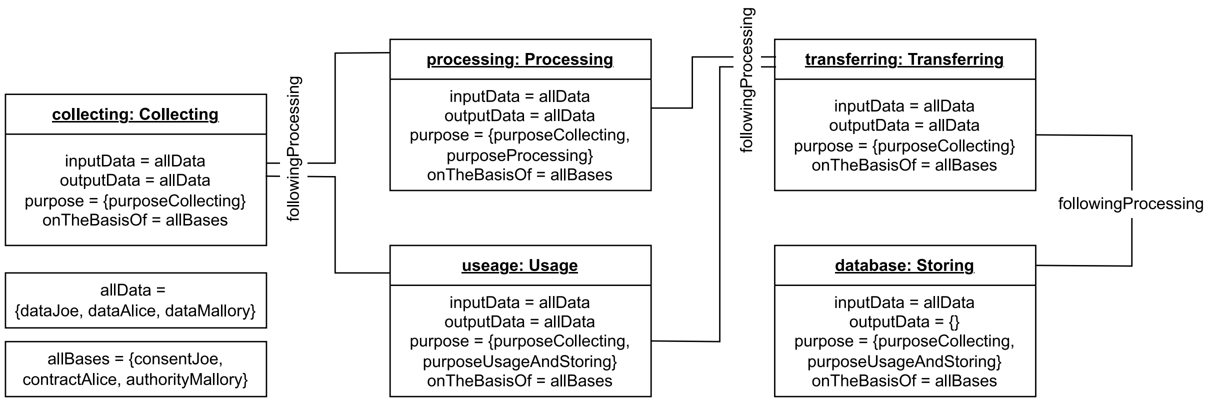

Figure 4.2: UML Object Diagram of Running Example Processing

In its current state, the scenario is compliant with the GDPR since all *Processing* is performed on valid *Legal Bases*, and the *Purpose* for the *Collecting* of data is carried through to all *followingProcessing*.

5 Analysis Tool

This chapter focuses on our proposed analysis tool approach. In Section 5.1, we analyze the GDPR for violations we are able to represent in the GDPR metamodel. Section 5.2 covers the design of the analysis tool. We describe the implementation of the developed design in Section 5.3.

5.1 Possible violations

The GDPR metamodel by Boltz aims to model specific aspects of the GDPR. The following section covers the possible violations that we can represent in the GDPR metamodel and their reference in the GDPR. [33]

No Legal Basis for Processing of Data

Article 6.1 of the GDPR [33] states that *Processing* of subject data is only lawful if done on a *Legal Basis*. The GDPR metamodel aims to include three *Legal Bases*. The first is *Consent* (Article 6.1.a) given by the data subject, allowing the *Controller* the *Processing* of their data. Secondly, the *Performance of a Contract* (Article 6.1.b), in which both the data subject and the *Controller* processing the data need to be contracting parties. The final *Legal Basis* in the GDPR metamodel is the *Exercise of Public Authority* (Articles 6.1.c and 6.1.e). The GDPR specifies two further *Legal Bases*: the protection of a data subject's vital interests (Article 6.1.d) and legitimate interests by the *Controller* that don't infringe upon a data subject's rights (Article 6.1.f). However, those *Legal Bases* are outside of the scope of the GDPR metamodel.

Therefore, the first possible violation our analysis tool needs to detect is *Processing* done without a valid *Legal Basis*.

No Proof of Consent

Should *Consent* be the *Legal Basis* for the *Processing* of a subject's data, the GDPR states that the *Controller* needs to be able to demonstrate that this *Consent* has been given (Article 7.1). This *Proof of Consent* has to be held by the *Controller* responsible for the *Processing* of data.

Our second possible violation is a missing *Proof of Consent* for *Processing* performed on the basis of said *Consent*.

Consent Not Given to all Processing Purposes

Recital 32.5 states that "When the processing has multiple purposes, consent should be given for all of them.". It is therefore, necessary for *Consent* to be defined for all *Purposes*, for which all *Processing* that uses the *Consent* as a *Legal Basis* is performed.

The third possible violation is *Consent* not defined for all necessary *Purposes*.

Proof of Consent Not Held by the Controller Deciding Over Defined Purposes

The *Processing* of data needs to be performed with a valid *Purpose* (Article 5.1). This *Purpose* is decided over by a *Controller*. Should *Consent* be used as the *Legal Basis* for *Processing* a subject's data, the *Proof of Consent* needs to be held by the *Controller* deciding over the *Purposes* for which the *Processing* is performed.

The fourth possible violation is proof of *Consent* held by the wrong *Controller*.

Violation of Purpose Limitation

Article 5.1.b states that "[Data should be] collected for specified, explicit, and legitimate purposes and not further processed in a manner that is incompatible with those purposes". The GDPR metamodel aims to model this by requiring at least a single *Purpose* being used for the collection of data to be present on every further *Processing* performed with the collected data.

The final possible violation is no *Purpose* being used for the collection of data being present on all *Processing* using the collected data.

5.2 Analysis Tool Design

We decided to implement the analysis logic in Prolog because of its backtracking capabilities. Since Prolog can't load ECore Models on its own, we additionally decided on creating a Java framework that loads the ECore Models, parses them into Prolog facts, and then calls the Prolog analysis with these additional facts.

5.2.1 Representing the Model Instances in Prolog

Since we decided to implement the analysis tool in Prolog, we need to bring model instances into a format that is understandable to the language. We do this by transforming elements of the GDPR metamodel into Prolog facts. Figure 5.1 shows how all elements are transformed into terms and facts. Underscored attributes mean the appropriate Prolog term is used, while the `[List.id]` notations means a list of all string IDs of the list elements.

Only *Processing* and its subtypes, *Controllers*, and *Collecting* are transformed into a dedicated fact, with all other information stored in those facts. *Consents*, *PerformanceOfContracts*, and *ExercisesOfPublicAuthority* are transformed into Prolog terms that hold additional information (like the consentee), if necessary. The terms are then further encapsulated by a generic *Legal Basis* term, that stores the original term as well as the data defined for *Legal Basis*. *Personal Data* is represented as a term with the IDs of all data subjects, while *Purposes* are transformed into terms holding their ID and the ID of the deciding *Controller*. *Controllers* get their own dedicated fact for *Proof* as well as *Purpose* lookups. Additionally, we created another fact for all *Collecting* elements, which will be needed for analyzing Purpose Limitation.

5.2.2 Java Framework

For the purpose of this thesis, we decided to design the analysis tool as a command-line application. Initially, the tool should ask for the location of the model instance that is to be analyzed and return an error message in the case of an invalid path. Given a valid path, it should then load the model instance at the specified location and identify all *Processing* and *Controllers*.

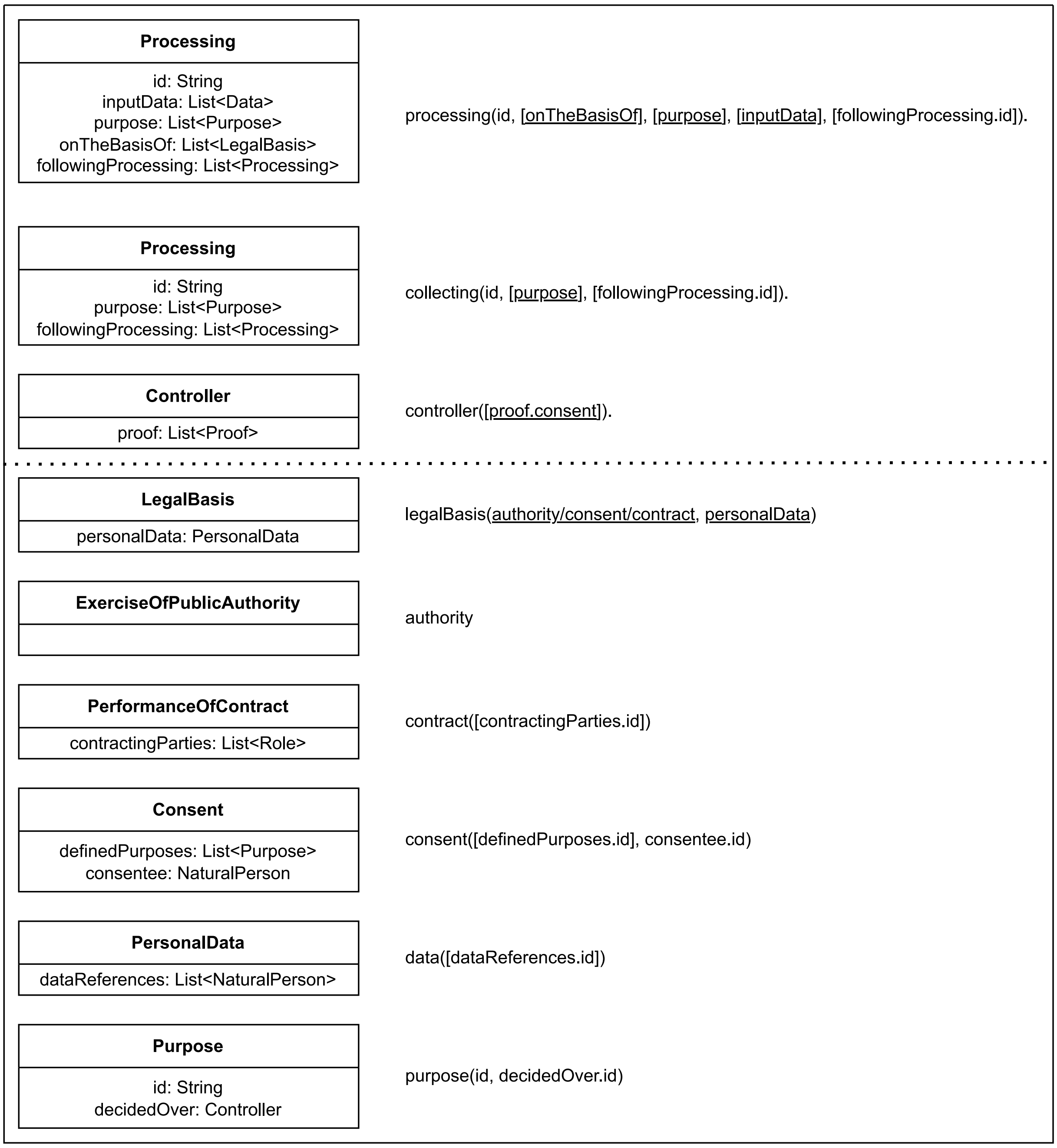
Processing
id: String
inputData: List<Data>
purpose: List<Purpose>
onTheBasisOf: List<LegalBasis>
followingProcessing: List<Processing>
processing(id, [onTheBasisOf], [purpose], [inputData], [followingProcessing.id]).
Processing
id: String
purpose: List<Purpose>
followingProcessing: List<Processing>
collecting(id, [purpose], [followingProcessing.id]).
Controller
proof: List<Proof>
controller([proof.consent]).
LegalBasis
personalData: PersonalData
legalBasis(authority/consent/contract, personalData)
ExerciseOfPublicAuthority
authority
PerformanceOfContract
contractingParties: List<Role>
contract([contractingParties.id])
Consent
definedPurposes: List<Purpose>
consentee: NaturalPerson
consent([definedPurposes.id], consentee.id)
PersonalData
dataReferences: List<NaturalPerson>
data([dataReferences.id])
Purpose
id: String
decidedOver: Controller
purpose(id, decidedOver.id)

Figure 5.1: Representation of GDPR Metamodel Elements in Prolog

To create the additional facts and execute the Prolog code, we will use the Projog framework [24]. The analysis tool should then create the additional facts for *Processing*, *Controllers*, and *Collecting* and add them to the Projog environment. It should then call the Prolog analysis code and wait for the execution to finish. Should a violation be found on a *Processing*, the ID will be returned in the console together with an error message. In the case of a violation of Purpose Limitation, the *Collecting* element whose *Purpose* was not carried will be returned. In case no violations are found, the analysis tool should return a success message. It should then wait on the location of another model instance or an exit command.

5.3 Analysis Tool Implementation

This section covers the implementation of the design concepts developed in the last section. We show the Java framework necessary for executing the analysis. Secondly, we present the implementation of the Prolog terms representing the violations found in Section 5.1.

5.3.1 Java Framework

In this section, we present the implementation of the Java framework, necessary for calling Prolog analysis terms. The framework consists of three classes. User interaction and loading the Prolog analysis rules is performed in the `main` method, while the `ModelParser` class is responsible for extracting the relevant information from the model. The `PrologBuilder` class creates the necessary Prolog facts and executes the analysis.

User Interface

For the purpose of this thesis, we decided to implement the analysis tool as a command-line application. Figure 5.2 shows the `main` method of the analysis tool. At the start of the program, the user will enter the location of the model that is to be analyzed. If the exit command is entered, the program will terminate, otherwise, it will continually ask for the next model instance after finishing the analysis. Next, a Projog [24] instance is created, and the Prolog code is loaded. Then a new `ModelParser` instance is created with the location of the model instance. The `ModelParser` will load the model and extract the elements relevant to the analysis. A new instance of `PrologBuilder` is created, which will create the necessary additional Prolog facts and execute the analysis. Should the `ModelParser` not be able to load or parse the model instance, the program will inform the user and ask for a new location. If the parsing was performed properly, the relevant elements are handed over to the `PrologBuilder` instance to create the necessary additional facts. Then the analysis is performed and the results are printed on the console.

```java
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while(true) {
            System.out.println("Enter model directory:");
            String file = scanner.next();
            if (file.equalsIgnoreCase("exit")) System.exit(0);
            Projog projog = new Projog();
            projog.consultFile(new File("src/main/resources/thesis.pl"));

            ModelParser parser = new ModelParser(file);
            PrologBuilder builder = new PrologBuilder(projog);
            if (!parser.parse()) continue;

            builder.build(parser.getProcessingList(), parser.getControllerList());
            System.out.print(builder.execute());
        }
    }
```

Figure 5.2: Analysis Tool Main Method

Model Parser

To load an Ecore model, first, a resource factory has to be registered for the relevant metamodel, or in this case, since we only load a single model type, as default. Figure 5.3 shows the methods relevant for model loading and parsing. The class instance then tries to load the model instance at the location provided during creation. In the case of a wrong model location or an unloadable model instance, "Invalid Directory" is printed to the console, and the method returns false to the `main` method. Otherwise, the list of *Processings* and *Controllers* is created for further referencing.

```java
    public boolean parse() {
        GDPRPackage.eINSTANCE.eClass();
        Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
        Map<String, Object> m = reg.getExtensionToFactoryMap();
        m.put(Resource.Factory.Registry.DEFAULT_EXTENSION, new XMIResourceFactoryImpl());

        ResourceSet rs = new ResourceSetImpl();
        try {
            Resource resource = rs.getResource(URI.createURI("file://" + file), true);
            LegalAssessmentFacts laf = (LegalAssessmentFacts) resource.getContents().get(0);
            parseLAF(laf);
            return true;
        } catch (Exception e) {
            System.out.println("Invalid Directory:");
            return false;
        }
    }

    private void parseLAF(LegalAssessmentFacts laf) {
        processingList = laf.getProcessing();
        List<Role> involvedParties = laf.getInvolvedParties();
        for (Role role : involvedParties) {
            if (role instanceof Controller) {
                controllerList.add((Controller)role);
            }
        }
    }
```

Figure 5.3: Excerpt of ModelParser Class

Prolog Builder

The `PrologBuilder` class is responsible for creating the additional Prolog facts and then subsequently executing the analysis. Before the analysis code is called, the necessary facts for *Processing*, *Collecting*, and *Controllers* need to be created. Figure 5.4 shows how this is performed. Since no changes are made to the *Processing* elements and *Controllers* this process can be performed in parallel for better performance. The Prolog facts are created according to the definition in Section 5.2.1.

```java
    public void build(List<Processing> processingList, List<Controller> controllerList) {
        processingList.stream().parallel().forEach(p -> {
            if (p instanceof Collecting)
            projog.executeQuery("assert(" + createCollectingRule((Collecting)p) + ").");
        });
        processingList.stream().parallel().forEach(p ->
            projog.executeQuery("assert(" + createProcessingRule(p) + ")."));
        controllerList.stream().parallel().forEach(c ->
            projog.executeQuery("assert(" + createControllerRule(c) + ")."));
    }
```

Figure 5.4: Prolog Builder build Method

To execute the analysis, two queries are executed by the Projog instance. Figure 5.5 shows the `execute` method, that is called by the `main` method. The `test(A)` term is fulfilled by *Processing* performed without a valid *Legal Basis*, while the `testCarriedPurpose(A)` term is fulfilled in the case of a violation of purpose limitation. If either term is fulfillable, the model instance is not in compliance with the GDPR. In the case of a non-valid *Legal Basis*, a return string is created with a violation message and the relevant *Processing* elements that violate GDPR regulations. For violations of purpose limitations, an additional message explaining the violation is created, and the *Collecting* elements whose *Purpose* were not carried are annotated. The `parseOutput` method extracts the ID from the previously created facts. In the case of a compliant model instance, the execute method returns a success message.

```java
    public String execute() {
        QueryResult carriedPurposes = projog.executeQuery("testCarriedPurposes(A).");
        QueryResult queryResult = projog.executeQuery("test(A).");
        StringBuilder result = new StringBuilder();
        if (queryResult.next()) {
            result.append("Model instance is NOT compliant.\n");
            result.append("Following processing nodes violate requirements:\n");
            result.append(parseOutput(queryResult.getTerm("A").toString()) + "\n");
            while (queryResult.next()) {
                result.append(parseOutput(queryResult.getTerm("A").toString()) + "\n");
            }
        }
        if(carriedPurposes.next()) {
            if (result.isEmpty()) result.append("Model instance is NOT compliant.\n");
            result.append("At least one collecting purpose needs to be carried through the entire flow.\n");
            result.append("Following collecting nodes purpose was not carried:\n");
            result.append(parseOutput(carriedPurposes.getTerm("A").toString()) + "\n");
            while (carriedPurposes.next()) {
                result.append(parseOutput(carriedPurposes.getTerm("A").toString()) + "\n");
            }
            return result.toString();
        }
        if (result.isEmpty()) result.append("Model instance is compliant.\n");
        return result.toString();
    }
```

Figure 5.5: Prolog Builder execute Method

5.3.2 Prolog

The analysis tool operates by analyzing each data subject of each *Data* element on each *Processing*. For each data subject, the analysis tool tests whether a valid *Legal Basis* is present. If there are no subjects left to be tested in a *Data* element, the analysis tool tests the next element. If there are no elements left, the analysis tool tests the next *Processing*. This behavior can be seen in lines 1-3, 8, 12, and 17 in Figure 5.6. A *Processing* is considered valid if all defined *Data* elements have been tested and no violations were found.

Legal Basis for Processing of Data

Since the GDPR metamodel considers three different *Legal Bases*: *Consent*, *PerformanceOfContract*, and *ExerciseofPublicAuthority*, we need to define three Prolog terms to test the validity of each *Legal Basis*.

Lines 5-8 of Figure 5.6 show how the analysis tool considers *ExerciseOfPublicAuthority* as the *Legal Basis*. First, any *Legal Basis* is taken from the list defined on the *Processing*. Then the type of the *Legal Basis* is tested. Should the *Legal Basis* be of a different type than *ExerciseOfPublicAuthority*, Prolog will try to take another *Legal Basis* from the list until an *ExerciseOfPublicAuthority* is found. If none is found, then the processing of the current subject's data is invalid under the *Legal Basis* of *ExerciseOfPublicAuthority*. It is, however, still possible that the processing is valid under another *Legal Basis*. Should *ExerciseOfPublicAuthority* be part of the *Legal Bases*, the analysis tool checks whether the data subject is part of the subjects defined on the *Legal Basis*. If this is the case, the processing of this subject's data is valid, and the analysis tool begins testing for the next subject.

Lines 14-17 show how processing on the basis of *PerformanceOfContract* is handled. The only addition to the testing of *ExerciseOfPublicAuthority* is the additional check of whether the data subject is part of the *contractingParty* list.

Ensuring the validity of *Consents* is more difficult since *Consent* has to be given to every *Purpose* the processing of data is performed for. The *Consent* to all *Purposes* may be split over multiple *Consent* elements, which further complicates the analysis. Figure 5.7 shows the `containsAllPurpose` term that gets called every time *Consent* is considered as a *Legal Basis*. To account for split *Consents*, all *Purposes* and *Legal Bases*, as well as the current data subject are handed over to the term. The tool will then take any *Purpose* of the given list and try to find a *Consent* that has been given for the specified *Purpose*. If a *Consent* is found the *Purpose* gets referenced to find the *Controller* deciding over it. Finally, it will be checked whether the relevant *Controller* holds *Proof* of the given *Consent*. This will be repeated until a valid *Consent* has been found for all processing *Purposes*.

The terms we have so far specified define whether *Processing* is considered valid. The analysis tool, however, should find *Processing* that violate the GDPR. This is achieved by telling Prolog to find a *Processing* that is considered not valid, as can be seen in Figure 5.8. The analysis tool will try to find one processing fact, as earlier defined by the Java framework, for which it cannot prove it is valid. If *Processing* whose validity can't be proved is found Prolog will return the specific fact and try to find another *Processing* in violation. If no other *Processing* is found the program terminates.

```
valid(processing(ID, _, _, [], _)).
valid(processing(ID, LegalBases, Purposes, [data([])|DataList], Following)) :-
    valid(processing(ID, LegalBases, Purposes, DataList, Following)).

valid(A):- A = processing(ID, LegalBases, Purposes, [data([Person|PersonList])|DataList], Following),
    member(LegalBasis, LegalBases), LegalBasis = legalBasis(Type, data(BasisPersonList)),
    Type = authority, member(Person, BasisPersonList),
    valid(processing(ID, LegalBases, Purposes,  [data(PersonList)|DataList], Following)).

valid(A):- A = processing(ID, LegalBases, Purposes, [data([Person|PersonList])|DataList], Following),
    containsAllPurposes(Purposes, LegalBases, Person),
    valid(processing(ID, LegalBases, Purposes, [data(PersonList)|DataList], Following)).

valid(A):- A = processing(ID, LegalBases, Purposes, [data([Person|PersonList])|DataList], Following),
    member(LegalBasis, LegalBases), LegalBasis = legalBasis(Type, data(BasisPersonList)),
    member(Person, BasisPersonList), Type = contract(Parties), member(Person, Parties),
    valid(processing(ID, LegalBases, Purposes, [data(PersonList)|DataList], Following)).
```

Figure 5.6: Excerpt of Analysis Tool Prolog Code

```
containsAllPurposes([Purpose|Purposes], LegalBases, Person):- member(LegalBasis, LegalBases),
    LegalBasis = legalBasis(Type, data(BasisPersonList)),
    Type = consent(PurposesConsent, Person), Purpose = purpose(PurposeID, ControllerID),
    member(PurposeID, PurposesConsent), member(Person, BasisPersonList),
    controller(ControllerID,ProofList), member(Type, ProofList),
    containsAllPurposes(Purposes, LegalBases, Person).

containsAllPurposes([], _, _).
```

Figure 5.7: Excerpt of Analysis Tool Prolog Code

```
test(A):- A = processing(ID, L, P, D, F), processing(ID, L, P, D, F), not(valid(A)).
```

Figure 5.8: Excerpt of Analysis Tool Prolog Code

Purpose Limitation

As described in Section 5.1 purpose limitation requires that at least one *Purpose* defined for the collection of data is present on all subsequent *Processing*. This is achieved by following the flow of *Processing* and building the intersection of processing *Purposes* of the previous and following *Processing* every time. If this intersection is non-empty on all *Processing* the collecting *Purpose* was carried through the system properly.

Figure 5.9 shows the `testCarriedPurposes` term which works similarly to the previous `test(A)` term, by trying to find a *Collecting* element whose *Purpose* has not been carried. Only *Purposes* and *followingProcessing* of each *Collecting* are handed over to the `purposeCarried` term. The term will then take one *Processing* from the *followingProcessing* list and build the intersection between the *Purpose* defined for the *Collecting* and the chosen following *Processing*. It will then proceed by calling the term for the chosen *Processing* together with the calculated intersection of *Purposes*. Additionally, the *Processing* element will be added to a list of already tested elements to prevent endless loops. Further, the tool will call the term again for the other elements in the *followingProcessing* list of the *Collecting* element. Should the term ever be called with an empty list of *Purposes* the validation will immediately fail for that *Processing*. Should no further elements be found in *followingProcessing* and the list of *Purposes* is still non-empty the *Purpose* of *Collecting* was carried correctly.

```
testCarriedPurposes(A):- A = collecting(ID, P, F), collecting(ID, P, F), not(purposeCarried(P, F, [])).

purposeCarried([], _, _):- !, fail.
purposeCarried(_, [], _).
purposeCarried(_, [ProcessingID|_], TestedNodes):- member(ProcessingID, TestedNodes).

purposeCarried(Purposes, [ProcessingID|FollowingProcessing], TestedNodes):-
    processing(ProcessingID, _, PurposesFollowing,_ ,FollowingProcessingFollowing),
    inter(Purposes, PurposesFollowing, Inter),
    purposeCarried(Inter, FollowingProcessingFollowing, [ProcessingID|TestedNodes]),
    purposeCarried(Purposes, FollowingProcessing, TestedNodes).
```

Figure 5.9: Excerpt of Analysis Tool Prolog Code

6 Transformation

This chapter covers our transformation approaches. In Section 6.1, we present the design concepts for our approaches. We first analyze both metamodels on elements that are semantically equivalent and can be directly transformed into each other. We then show a mechanism for storing GDPR metamodel information in the DFD metamodel and develop a Tracemodel for retaining DFD metamodel information. In Section 6.2, we discuss the implementation of the developed design concepts. This section is split by the direction of the transformation.

6.1 Transformation Design

We decided to implement the transformation with the QVT Operational Mappings language since it has the best documentation of all presented transformation languages. For the purpose of this thesis, the transformations will be created as QVTO standalone projects that can be executed in the Eclipse IDE directly. Those standalone transformations can also be called in Java, which means they can easily be integrated into further projects.

Transformations With Equivalent Target Element

First, we analyzed which elements of each model can be easily represented in the other model. DFD *Nodes* are equivalent to *Processing* elements in the GDPR metamodel and are therefore transformed into those and back. There are three types of *Nodes* in the DFD metamodel and all can directly be translated into GDPR *Processing* elements. *Store Nodes* are transformed into *Storing* and *Process Nodes* into GDPR *Processing*. *External Nodes* are transformed into *Collecting* since that is the closest equivalent. However, the GDPR metamodel has two more types of *Processing*: *Transferring* and *Usage*, that don't have an equivalent *Node* in the DFD metamodel. They are therefore transformed into DFD *Process Nodes*, with storage of the types for a subsequent transformation being considered later. DFD *Flows* don't have an equivalent element in the GDPR metamodel, however, they can be represented through the *followingProcessing* attribute of a *Processing*. A DFD *Flow* from *Node* A to *Node* B is represented through *Processing* B being part of *Processing* A *followingProcessing*. This transformation works in the other direction too. This, however, means that all additional information of *Flows* (name and *Pins*) is lost and storage will be considered later.

Transformations Without Equivalent Target Element

While DFD *Nodes* are equivalent to GDPR *Processing* they lack options for representing the additional aspects of *Processing*, *Personal Data*, *Legal Bases*, and *Purposes*. We decided to store the additional information as *Labels* and assign them to the properties attribute of DFD *Nodes*. Figure 6.1 shows how elements of the GDPR metamodel are transformed into elements of the DFD metamodel and back. Elements or relations of the same color are transformed into equally colored elements in the other metamodel. We created the `GDPRNode`

LabelType that will hold all *Processing* property *Labels*. *Labels* can only store information in their name string. Therefore, the *Processing* description *Labels* only store the ID of the referenced elements. An example of a *Label* name which is representing a *Purpose* would be, `GDPR::hasPurpose:GDPR::Purpose:<Purpose-ID>`. We also store the type of *Processing* in this way, e.g. `GDPR::isType:GDPR::Useage`.

In the next step, we define the *LabelType* `GDPRElements`, in which we will store *Labels* representing all the elements of the GDPR metamodel that don't have an equivalent element in the DFD metamodel. These are *Data*, *Purposes*, *Controllers*, *Natural Persons*, *Legal Bases*, and *Proof of Consent*. For each element, we create a *Label* that stores type and ID in its name like this, `GDPR::Controller:<Controller-ID>`. Those *Labels* are all added to the `GDPRElement` *LabelType* but not assigned to a *Node* or *Behaviour*.

Further, we define the `GDPRLink` *LabelType* that keeps references to attributes and the element's name. An example for the reference between *Consent* and the *Purposes* it is defined on would look like this
`GDPR::LinkConsentPurpose:GDPR::Consent:<Consent-ID>GDPR::Purpose:<Purpose-ID>`.
These *Labels* are again only added to the *LabelType* but not assigned to *Nodes*.

We decided on storing the additional information as *Labels* since they can be further used for an analysis of the DFD metamodel.

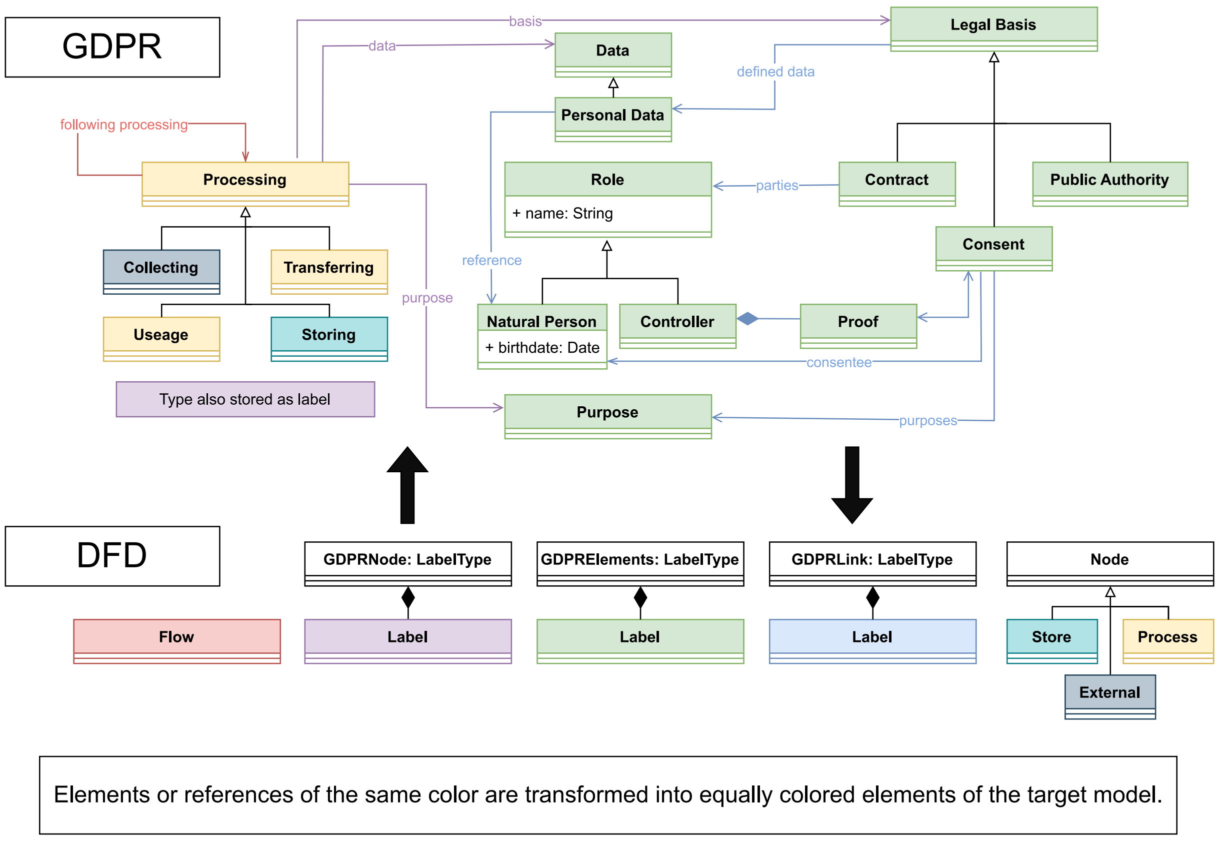

Figure 6.1: Visualisation of GDPR2DFD Transformation

Trace Model

Similarly to the GDPR metamodel, the DFD metamodel contains elements without an equivalent element in the GDPR metamodel. The only elements assignable to *Processing*, that could potentially store the information are *Purposes*. Adding additional *Purposes* would, however, defeat the purpose of the GDPR metamodel and make it harder for a software architect to employ. We decided on creating an additional Tracemodel, which can store the additional information when performing a DFD2GDPR transformation. The Tracemodel is then used in the GDPR2DFD transformation to restore the lost information. The Tracemodel stores which DFD *Node* was transformed into which GDPR *Processing*. With these properties, the *Behaviour* can be restored when transforming back. We additionally keep a list of all Information *Flows* in the Tracemodel to restore *Flow* names and *Pins*.

This inevitably means the creation of 2 transformations per direction. One transformation for initial executions without a Tracemodel or additional *LabelTypes* and one where those elements exist and need to be handled.

6.2 Transformation Implementation

In this section, we present the implementation of the concepts developed in the last section. This chapter is split by the direction of the transformation. Section 6.2.1 covers the transformation from Boltz's GDPR metamodel into the DFD metamodel by Seifermann et al. [29]. We show the implementation of the initial transformation and how the information of model instances that have previously been transformed from the DFD metamodel is restored. Section 6.2.2 covers the direction from the DFD metamodel into the GDPR metamodel. The section is equally split into the initial transformation and the transformation that restores GDPR legal specifications.

6.2.1 GDPR2DFD

This section covers the transformation of GDPR metamodel instances into DFD metamodel instances. The first paragraph covers initial transformations without the existence of a Tracemodel. We show how elements with equivalent elements are transformed and how the additional legal specifications of the GDPR metamodel are transformed into *Labels*. If a DFD metamodel instance was transformed into the GDPR metamodel a Tracemodel was created to store additional information. In the case of a transformation back into the DFD metamodel, this Tracemodel needs to be referenced to restore the information. Transformations in the context of the Tracemodel are covered in the last paragraph.

Initial Transformation

As described in paragraph 6.1, GDPR *Processing* elements are transformed into DFD *Nodes*. Figure 6.2 shows the transformation. QVTO allows the overloading of mappings as long as all input, as well as all output elements, are of the same supertype, respectively. Additionally to the `Processing:: Processing2Node() : Node` mapping, we define the `Processing2Node` Mapping for `Storing2Store` and `Collecting2External`, as described in paragraph 6.1. QVTO will always choose the most specific mapping. The DFD *Node* class, however, is abstract, which is why the result of the transformation needs to be set in the `init` section. The transformation also creates a *Behaviour* on the *Node* which will later be necessary for having *Pins* to the *Flows*. All relevant *Processing* properties are parsed into *Labels* and then added to the `properties` attribute of DFD *Nodes*. Finally, the entity name and ID of the created *Node* are set to the values of the source *Processing* for better understandability.

```
mapping Processing:: Processing2Node() : Node {
    init {
        result := new Process();
    }
    result.behaviour := new Behaviour();
    dataDictionary.behaviour += result.behaviour;
    result.properties += self.map Processing2Labels();
    parent.nodes += result;

    result.entityName := self.entityName;
    result.id := self.id;

}

mapping Storing:: Processing2Node(): Store {
    result.behaviour := new Behaviour();
    dataDictionary.behaviour += result.behaviour;

    result.properties += self.map Processing2Labels();
    parent.nodes += result;

    result.entityName := self.entityName;
    result.id := self.id;
}

mapping Processing:: Processing2Labels() : Set(Label) {
    init {
        result += self.inputData->map hasInputData();
        result += self.outputData->map hasOutputData();
        result += self.purpose->map hasPurpose();
        result += self.onTheBasisOf->map onTheBasisOf();
        result += self.map isType();
        result += self.map id();
    }
    self.followingProcessing->map Processing2Flow(self);
}
```

Figure 6.2: GDPR2DFD Processing2Node Transformation

Additionally to creating the relevant *Labels* the `Processing2Labels()` mapping also starts the transformation of *followingProcessing* into *Flows*, which can be seen in Figure 6.3. Since the mapping operation is called by each following *Processing*, the source *Processing* needs to be handed over as a parameter. The mapping first transforms the GDPR source and destination *Processing* into their DFD equivalents and saves them as variables. This is possible because when calling a mapping operation for the same element repeatedly, QVTO will return the element that has been created on the first occurrence instead of creating new elements. Next, the *Pins* are created and added to relevant *Node Behaviours* and the *Flow*, together with setting *Flow* source and destination *Nodes*.

```
mapping Processing:: Processing2Flow(in sourceProcessing:Processing) : Flow {
    var source := sourceProcessing.map Processing2Node();
    var destination := self.map Processing2Node();
    var sourcePinVar := new Pin();
    var destinationPinVar := new Pin();
    sourceNode := source;
    destinationNode := destination;
    sourcePin := sourcePinVar;
    destinationPin := destinationPinVar;
    source.behaviour.outPin += sourcePinVar;
    destination.behaviour.inPin += destinationPinVar;
    parent.flows += result;
}
```

Figure 6.3: GDPR2DFD Processing2Flow Transformation

Next, the GDPR elements without equivalent DFD elements are transformed into *Labels*. The main mapping operation `LAD2DFD()`, seen in Figure 6.4, transforms the parent elements of each metamodel, which contain all elements, into the other. It also calls the *Label* mappings for each GDPR element, that can't directly be transformed. The figure shows an exemplary mapping of a *Purpose* element into a *Label* as well as the mapping of a *Processing*'s type into a *Label*. In the final step of each mapping, the resulting *Labels* are added to the relevant *LabelType* for containment. *Purposes* have a reference to the *Controller* that is deciding over them. When a *Purpose* is transformed into a *Label* via the `Purpose2Label()` mapping, the mapping will call the mapping operation `LinkPurposeController` which takes the *Purpose* as a parameter and creates a link between the two elements as previously described.

```
mapping LegalAssessmentFacts:: LAF2DFD(): DataFlowDiagram {
    init{
        result := parent;
    }
    self.allSubobjectsOfKind(Processing)->map Processing2Node();
    self.allSubobjectsOfKind(Purpose)->map Purpose2Label();
    self.allSubobjectsOfKind(Data)->map Data2Label();
    self.allSubobjectsOfKind(Role)->map Role2Label();
    self.allSubobjectsOfKind(LegalBasis)->map LegalBasis2Label();
    self.allSubobjectsOfKind(AbstractGDPRElement)->map LinkIdEntityName();
}

mapping Purpose:: Purpose2Label() : Label {
    result.entityName := "GDPR::Purpose:" + self.id;
    gdprElementLabelType.label += result;
    gdprLinkLabelType.label += self.decidedOver.map LinkPurposeController(self);
}

mapping Controller:: LinkPurposeController(in purpose:Purpose): Label {
    result.entityName := "GDPR::LinkPurposeController:GDPR::Purpose:" + purpose.id + "GDPR::
    Controller:" + self.id;
}

mapping Storing:: isType(): Label {
    result.entityName := "GDPR::isType:GDPR::Storing";
    gdprNodeLabelType.label += result;
}
```

Figure 6.4: GDPR2DFD Elements2Labels Transformation

Transformation With Tracemodel

In the case of the GDPR source model instance having been transformed from a DFD instance earlier, the additional Tracemodel is needed to restore the information that could not be kept in the GDPR metamodel. Figure 6.5 shows the differences between the previous `Processing2Node()` mapping and the changed syntax for *Flow* reconstruction in the case of an existing Tracemodel instance. As described in paragraph 6.1 the Tracemodel keeps track of which DFD *Node* was transformed into which GDPR *Processing* element, as well as keeping a list of all *Flows* to recreate entity names and IDs. To transform a GDPR *Processing* element into a DFD *Node* the mapping operation will first reference the Tracemodel to see whether the *Processing* was earlier transformed from a DFD *Node*. If this is the case, the result of the mapping operation will be set to that DFD *Node* instead of creating a new one. A *Behaviour* only needs to be added if a new *Node* is created since the old *Node* still carries all relevant references. To restore the *Flow* information, the `createFlows` helper is called for every *Processing*. Helper operations can be called by the program itself instead of elements. The helper will then call the `resolveFlow` helper for each *followingProcessing* and add the resulting *Flow* to the Data Flow Diagram parent. The helper searches the Tracemodel *FlowList* for a *Flow* with matching source and destination *Node* ID. If a matching *Flow* is found, it is returned; otherwise a new *Flow* is created analogous to the non-Tracemodel part.

```
mapping Processing:: Processing2Node() : Node {
    init {
        var test := Tracemodel.TracesList->xselect(i| i.processing = self)->first().node;
        if (test <> null) {
            result := test;
        } else {
            result := new Process();
        }
    }
    if (test = null) {
        result.behaviour := new Behaviour();
        dataDictionary.behaviour += result.behaviour;
    };
    result.properties += self.map Processing2Assignment();
    parent.nodes += result;

    result.entityName := self.entityName;
    result.id := self.id;

}

helper createFlows(in sourceProcessing:Processing) {
    sourceProcessing.followingProcessing->forEach(i){parent.flows += resolveFlow(sourceProcessing
     , i, i.id)};
}

helper resolveFlow(in sourceProcessing:Processing, in destinationProcessing:Processing, test:
     String) : Flow{
        Tracemodel.FlowList->forEach(i){
            if (i.sourceID = sourceProcessing.id and i.destinationID  = destinationProcessing.id)
      {
                return i.flow;
            }
        };

        var flow := new Flow();
        [... Equivalent to Non-Tracemodel part ...]
        return flow;
}
```

Figure 6.5: GDPR2DFD Transformation With Tracemodel

6.2.2 DFD2GDPR

This section will cover the transformation of DFD metamodel instances into GDPR metamodel instances. This section is split into two paragraphs. The first paragraph will cover the initial transformation of DFD metamodel instances that have not earlier been transformed from GDPR metamodel instances. This means none of the additionally created *Labels* are present and need to be resolved. In the second paragraph, we will present how those additional *Labels* are used to restore GDPR metamodel information.

Initial Transformation

With no additional *Labels* being present, the transformation of DFD instances is rather straightforward and can be seen in Figure 6.6. The initial `DFD2LAF()` mapping transforms the DFD parent element into the GDPR parent element and starts inferring GDPR *Nodes* from *Flows*. Finally, all *Flows* are transformed into *FlowElements* that are kept in the Tracemodel for later reconstruction. The `Flow2Processing()` mapping first transforms source and destination *Nodes* into their GDPR counterparts and then adds the transformed destination *Processing* to the source *Processing*'s *followingProcessing* list. The `Node2Processing()` mapping first saves the trace between the source *Node* and the resulting *Processing*, sets the entity name and ID of the result to the source *Nodes* values, and finally adds the created *Processing* to the parent element for containment.

```
transformation DFD2GDPR (in dfd:DFD, out tm:Tracemodel, out gdpr:GDPR) {
...
    mapping DataFlowDiagram:: DFD2LAF(): LegalAssessmentFacts {
        init {
            result := parent;
        }
        self.flows->map Flow2Processing();
        Tracemodel.FlowList += self.flows->map Flow2FlowElement();
   }

    mapping Flow :: Flow2FlowElement() : FlowElement {
        result.flow := self;
        result.sourceID := self.sourceNode.id;
        result.destinationID := self.destinationNode.id;
    }

    mapping Flow::Flow2Processing() : Processing {
        init {result := self.sourceNode.map Node2Processing()}
        result.followingProcessing += self.destinationNode.map Node2Processing();
    }

    mapping Node::Node2Processing() : Processing {
        var trace = new Trace();
        trace.processing := result;
        trace.node := self;
        Tracemodel.TracesList += trace;

        result.id := self.id;
        result.entityName := self.entityName;

        parent.processing += result;
    }
...
}
```

Figure 6.6: DFD2GDPR Transformation Without GDPR Labels

Transformations With Information Restoring

In the case of additional GDPR *Labels* being present, the transformation first needs to find the three relevant *LabelTypes* out of the list of all *LabelTypes* in the *Data Dictionary*. The *Data Dictionary* is the parent element for all *LabelTypes* and *Behaviours*, while all other elements are kept in the *Data Flow Diagram* element. Figure 6.7 shows the `main` method of the transformation. After the *LabelTypes* are found the *Labels* contained in the *LabelType* are stored as a property (QVTO equivalent of global variables) for further use. Then all element *Labels* are transformed back into the GDPR element they were created from. After the mapping operations for the parent elements are called the GDPR *LabelTypes* are deleted from the *Data Dictionary*.

```
main() {
    var gdprNodeType := dd.objectsOfType(LabelType)->xselect(i| i.entityName.startsWith("GDPRNode
    "))->asList()->first();
    var gdprLinkType := dd.objectsOfType(LabelType)->xselect(i| i.entityName.startsWith("GDPRLink
    "))->asList()->first();
    var gdprElementType := dd.objectsOfType(LabelType)->xselect(i| i.entityName.startsWith("
    GDPRElement"))->asList()->first();


    gdprNodeLabels := gdprNodeType.label;
    gdprLinkLabels := gdprLinkType.label;
    gdprElementLabels := gdprElementType.label;

    gdprElementLabels->map Label2Purpose();
    ... [All Element Label Resolving] ...
    gdprElementLabels->map Label2Authority();

    dfd.objectsOfType(DataFlowDiagram)[DataFlowDiagram]->map DFD2Trace();
    dfd.objectsOfType(DataFlowDiagram)[DataFlowDiagram]->map DFD2LAF();

    dd.removeElement(gdprNodeType);
    dd.removeElement(gdprLinkType);
    dd.removeElement(gdprElementType);
}
```

Figure 6.7: DFD2GDPR Transformation main() With GDPR Labels

All element mapping operations are called with all element *Labels*. However, mapping operations can have an additional `when` clause that defines when the mapping is to be performed. Figure 6.8 shows how a *Label* representing a *Controller* is transformed back into a *Controller* instance. The mapping is only executed if the *Label*'s entity name starts with the prefix defined in paragraph 6.1. The *Label* however only stores the ID of the *Controller* instance, so the created Controller's ID is set to that. Afterward, the name of the *Controller* is resolved by using the `GetRoleName` helper. The helper looks through the list of linking *Labels* and finds the one linking the *Controller* ID to its name. To find the *Proofs* of *Consent* the *Controller* is holding, the `ResolveLinkControllerProof` helper first finds all *Labels* that link the *Controller* ID to the *Proof* ID. Afterward, another helper returns the *Proof Labels* associated with the found IDs. Finally, the helper gets all the *Proof* elements that have previously been created from those *Labels* and returns them.

```
mapping Label::Label2Controller(): Controller
    when {self.entityName.startsWith("GDPR::Controller:")}
    {
        var id := self.entityName.substringAfter("GDPR::Controller:");
        result.id := id;
        result.name := GetRoleName(id);
        result.entityName := GetEntityName(id);
        result.proof += ResolveLinkControllerProof(id);

        parent.involvedParties += result;
    }

helper GetRoleName(in id:String) : String {
    return gdprLinkLabels->xselect(i| i.entityName.startsWith("GDPR::LinkPersonName:GDPR::Role:"
    + id))->
            first().entityName.substringAfter("GDPR::Name:");
}

helper ResolveLinkControllerProof(in inId:String): OrderedSet(Proof) {
    var elementIds := gdprLinkLabels->xselect(i| i.entityName.startsWith("GDPR::
    LinkControllerProof:GDPR::Controller:" + inId))->
        xcollect(i| i.entityName.substringAfter("GDPR::Proof:"))->asOrderedSet();
    var outElements = new OrderedSet(Proof)();
    outElements += GetLabelsFromIdList(elementIds)->map Label2Proof();
    return outElements;
}

helper GetLabelsFromIdList(inIds:OrderedSet(String)):OrderedSet(Label) {
    var labels = new OrderedSet(Label)();
    inIds->iterate(i; acc:OrderedSet(Label) = labels| labels += GetLabelFromId(i));
    return labels;
}

helper GetLabelFromId(inId:String): Label {
    return gdprElementLabels->xselect(i|i.entityName.endsWith(inId))->first();
}
```

Figure 6.8: DFD2GDPR Transformation With GDPR Labels

7 Evaluation

This chapter will cover the evaluation of our approach. To achieve a clearly structured evaluation, the Goal Question Metric (GQM) approach [5] will be employed. In Section 7.1, we present the evaluation design and the GQM plan. We define the evaluation goals: accuracy and scalability, followed by the corresponding questions and metrics. Section 7.2 describes the evaluation setup used for evaluating the defined question, while in Section 7.3 we will discuss the result of our evaluation. In paragraph 7.4, we will discuss threats to the validity of our evaluation, while Section 7.5 covers the limitations of our approach.

7.1 Evaluation Design

For the analysis tool, we define two evaluation goals:

- EG-1: Examine the analysis tool's accuracy by comparing the results to expected results for predefined GDPR violations.
- EG-3: Examine the analysis tool's scalability for larger model instances.

For the transformation approach, we define three evaluation goals:

- EG-2.1: Examine the transformation approach's accuracy by comparing the transformation results to manually transformed model instances.
- EG-2.2: Examine the transformation approach's accuracy by transforming a model instance into the respective model and back, and comparing it to the initial state.
- EG-4: Examine the transformation approach's scalability for larger model instances.

7.1.1 Evaluation Design for Accuracy

To evaluate evaluation goal EG-1, we define the following evaluation questions:

- Q1.1: What is the analysis tool's accuracy in identifying violations in which *Processing* was defined without a valid *Legal Basis*?
- Q1.2: What is the analysis tool's accuracy in identifying violations in which no *Proof* of *Consent* is kept by the *Controller*?
- Q1.3: What is the analysis tool's accuracy in identifying violations in which the given *Consent* is not sufficient in covering all processing *Purposes*?

- Q1.4: What is the analysis tool's accuracy in identifying violations in which the *Proof of Consent* is held by a *Controller* that isn't also deciding over the *Consent*'s *Purposes*?
- Q1.5: What is the analysis tool's accuracy in identifying violations in which the *Purpose* for the collection of data is not carried through all *Processing* using the collected data?

To evaluate the accuracy of the analysis approach, the Precision-Recall and the F_1 Metric [23] will be employed:

M1.1 Precision (p) describes the ratio between correctly identified violations t_p and the sum of t_p and f_p, with f_p representing the number of false positives, or identified violations, that weren't a violation. $p = \frac{t_p}{t_p+f_p}$
M1.2 Recall (r) describes the ratio between correctly identified violations t_p and the sum of t_p and f_n, with f_n representing the number of false negatives, or violations in the scenario, that weren't found. $r = \frac{t_p}{t_p+f_n}$
M1.3 F$_1$ represents the harmonic mean of both precision and recall in one metric. $F_1 = 2\frac{p*r}{p+r}$

For EG-2 it is necessary to differentiate between initial transformations and transformations that restore information as described in 6.2. We define the following evaluation questions:

- Q2.1.1: What is the transformation approach's accuracy in transforming a DFD metamodel instance into a GDPR metamodel instance?
- Q2.1.2: What is the transformation approach's accuracy in transforming a GDPR metamodel instance into a DFD metamodel instance?
- Q2.2.1: What is the transformation approach's accuracy when transforming a GDPR metamodel instance into the DFD metamodel and back?
- Q2.2.2: What is the transformation approach's accuracy when transforming a DFD metamodel instance into the GDPR metamodel and back?

To evaluate the accuracy of the transformation approach, the Precision-Recall and the F_1 Metric [23] will be employed:

M2.1 Precision (p) describes the ratio between correctly transformed model elements t_p and the sum of t_p and f_p, with f_p representing the number of erroneous additional elements not derived from the source model instance. $p = \frac{t_p}{t_p+f_p}$
M2.2 Recall (r) describes the ratio between correctly transformed model elements t_p and the sum of t_p and f_n, with f_n representing the number of wrongly or not transformed elements of the source model instance. $r = \frac{t_p}{t_p+f_n}$
M2.3 F$_1$ represents the harmonic mean of both precision and recall in one metric. $F_1 = 2\frac{p*r}{p+r}$

7.1.2 Evaluation Design for Scalability

Our approach might be used to design and analyze large systems, consisting of many *Processing* elements, users, and *Purposes*. Therefore, evaluating scalability, with evaluation goals EG-3 and EG-4 is of importance. In this section, we take a look at the execution time of both analysis and transformation for increasingly large model instances. We use the execution time as Metric M-3 and M-4, respectively.

The execution time of the analysis tool heavily depends on the number of *Processing* that need to be tested, since each *Processing* adds a Prolog fact `processing(id, legalBases, purposes, data` that needs to be evaluated. Additionally, the number of *Natural Persons* and *Purposes* also has an impact since each *Processing* needs to be tested on all subjects and *Purposes*.

For evaluation goal EG-3, we define the following questions:

- Q3.1: How does the analysis tool's execution time scale when increasing the number of *Processing*?
- Q3.2: How does the analysis tool's execution time scale when increasing the number of *Natural Persons*?
- Q3.3: How does the analysis tool's execution time scale when increasing the number of *Purposes*?

For evaluation goal EG-4, we define the following questions:

- Q4.1: How does the transformation time scale when increasing the number of *Nodes* in the DFD model?
- Q4.2: How does the transformation time scale when increasing the number of *Processing* in the GDPR model?
- Q4.3: How does the transformation time scale when increasing the number of *Natural Persons* in the GDPR model?
- Q4.4: How does the transformation time scale when increasing the number of *Purposes* in the GDPR model?

7.2 Evaluation Setup

This section covers the setups used to answer the defined evaluation questions. We will describe the model instances used for evaluating accuracy and scalability.

7.2.1 Evaluation Setup for Accuracy

Since the GDPR metamodel has not been publicized at the point of writing this thesis, there are no other publications containing scenarios or use cases we could use for our evaluation. Therefore, we will use the running example defined in Chapter 4 for evaluating all GDPR-based questions. For questions regarding the DFD metamodel, we will use scenarios defined by Seifermann et al. in their original work on the extended DFD syntax [29]. To answer evaluation questions Q1.1 to Q1.5, we modify our running example as follows:

Setup for Question Q1.1

To answer question Q1.1, we remove the *Legal Basis* for processing from two *Processing* elements in the running example. Figure 7.1 shows an UML object diagram of the *Processing* structure

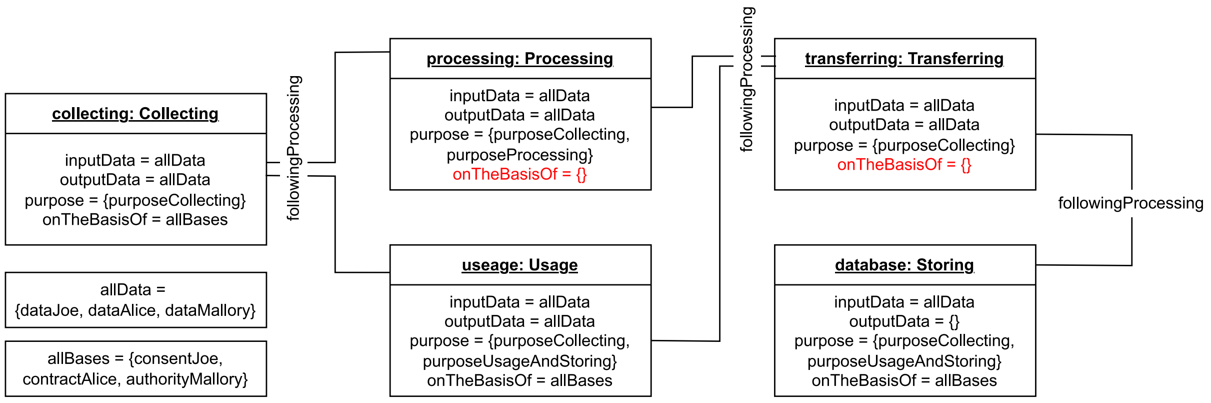

Figure 7.1: Setup for Question Q1.1

with the *Legal Bases* removed for *processing* and *transferring*. The structure of all other legal elements as seen in Figure 4.1 remains the same. We expect violations on both *Processing* without *Legal Bases.*

Setup for Question Q1.2

To answer question Q1.2, we remove the *Proof* of Joe's *Consent* from the running example. Figure 7.2 shows a UML object diagram of the running example with the *Proof* to be removed

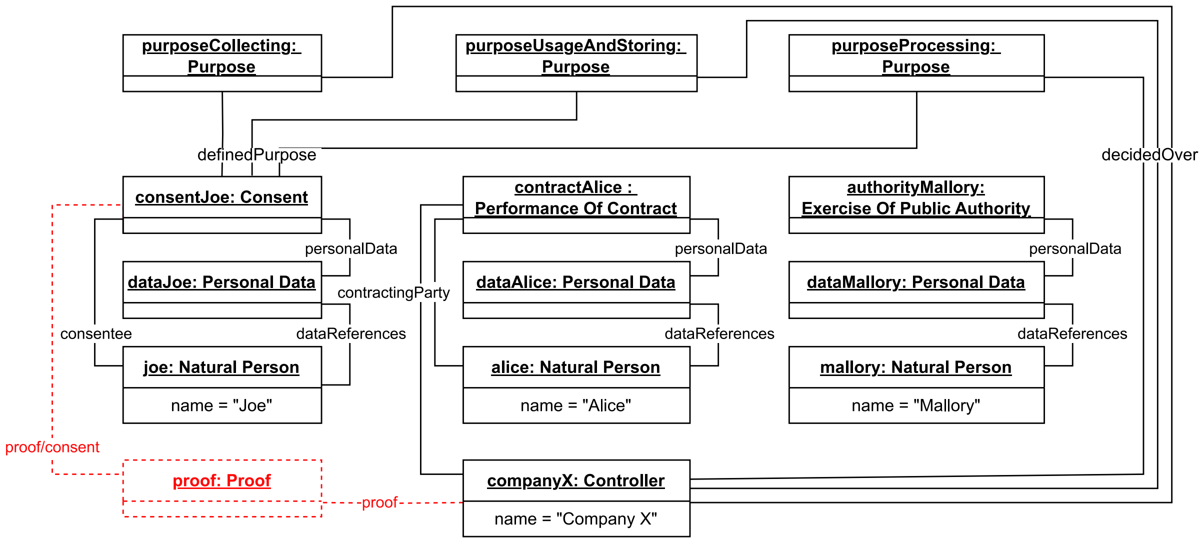

Figure 7.2: Setup for Question Q1.2

marked in red. The structure of the *Processing* elements as seen in 4.2 remains the same. This should lead to a violation on every single *Processing* since none of them have a valid *Legal Basis* for processing Joe's data now.

Setup for Question Q1.3

To answer question Q1.3, we remove the reference to *purposeUsageAndStoring* from the defined *Purposes* of *consentJoe* in our running example. Figure 7.3 shows a UML object diagram of

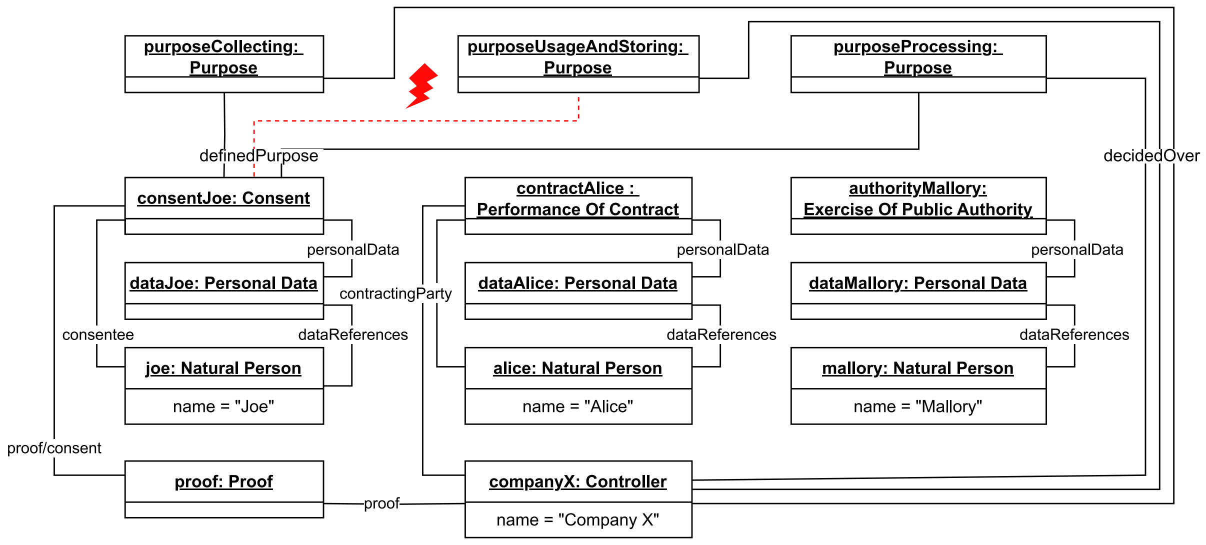

Figure 7.3: Setup for Question Q1.3

the running example with the reference to be removed marked in red. The structure of the *Processing* elements remains the same. This should lead to two violations, on *usage* and *storing*, respectively, since both *Processing* is performed for the removed *Purpose*.

Setup for Question Q1.4

To answer question Q1.4, we add a new *Controller* called *companyY*. Company Y now holds the *Proof* of Joe's *Consent*, while the *Purposes* defined on the *Consent* are decided over by *Controller companyX*.

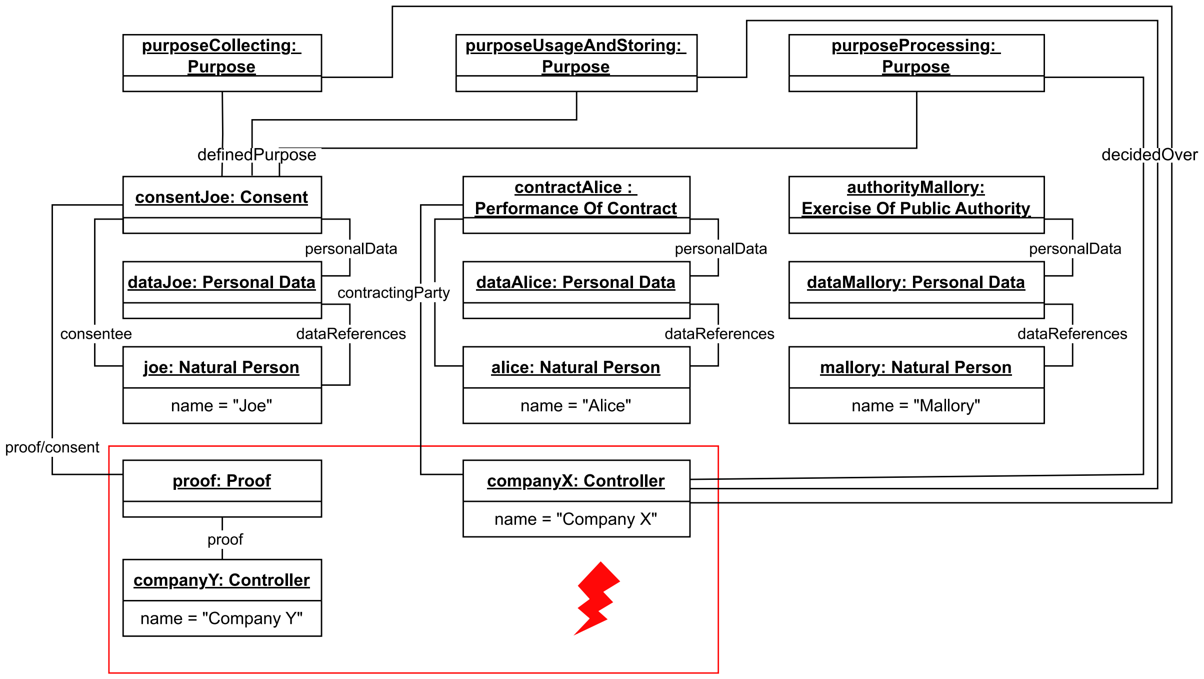

Figure 7.4: Setup for Question Q1.4

Figure 7.4 shows a UML object diagram of the modified model instance used to answer question Q1.4. This should again lead to violations on all *Processing* since *consentJoe* is no longer a valid *Legal Basis* for the processing of Joe's data.

Setup for Question Q1.5

To answer question Q1.5, we remove *purposeCollecting* from *processing* and the *database*.

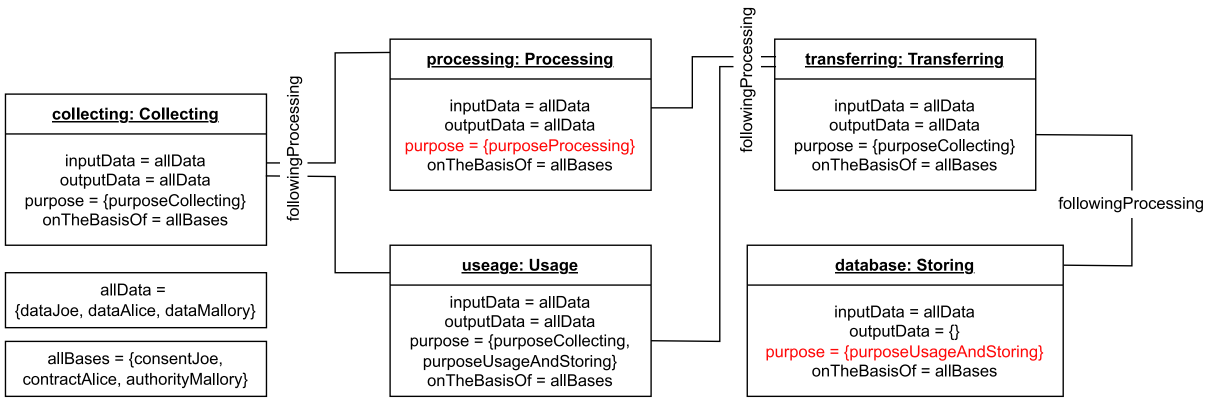

Figure 7.5: Setup for Question Q1.5

Figure 7.5 shows a UML object diagram of the modified model instances *Processing* elements. This should lead to a violation on *collecting* since its *Purpose* was not carried through all *Processing* in *followingProcessing*.

To answer evaluation questions Q2.1.1 and Q2.1.2, we will compare the results of the transformation approach to a manually created Gold-Standard. For questions Q2.2.1 and Q2.2.2, we will compare the results to the source model instances handed to the transformation.

Setup for question Q2.1.1

For determining the accuracy of the transformation, when transforming DFD-Model instances, we will reuse scenarios used by Seifermann et al. [29] to evaluate their original DFD metamodel. We decided to evaluate the transformation with the ABAC and Hospital App use cases [28].

Figure 7.6 shows the ABAC use case. Forward, join, and location changer are predefined types of *Behaviours*. Forward *Behaviour* describes the process of simply forwarding all *Labels* that arrived on the *Node* to the next *Node*. Join always takes the highest level *Label*, and sends that to the next *Node*. If both the celebrity and regular *Label* would arrive at a join *Node*, only the celebrity *Label* would be sent to the next *Node* since it is the higher-level *Label*. Location changer always sends the *Label* Asia to the next *Node*, no matter what origin *Label* arrived at the *Node*. In this use case, the *LabelTypes* Location and Role are assigned as *Node* properties while Origin and status are passed on *Flows*.

Figure 7.7 shows the Hospital App use case. Forward and join Behavior work analogous to the ABAC use case. Encrypt *Behaviour* always propagates the lowest level *Label*, in this case, Low. In this use case all *Nodes*, but the attacker, were assigned the Trust *Label*. Should data labeled as High ever reach a *Node* that isn't labeled as Trust, this would constitute a violation.

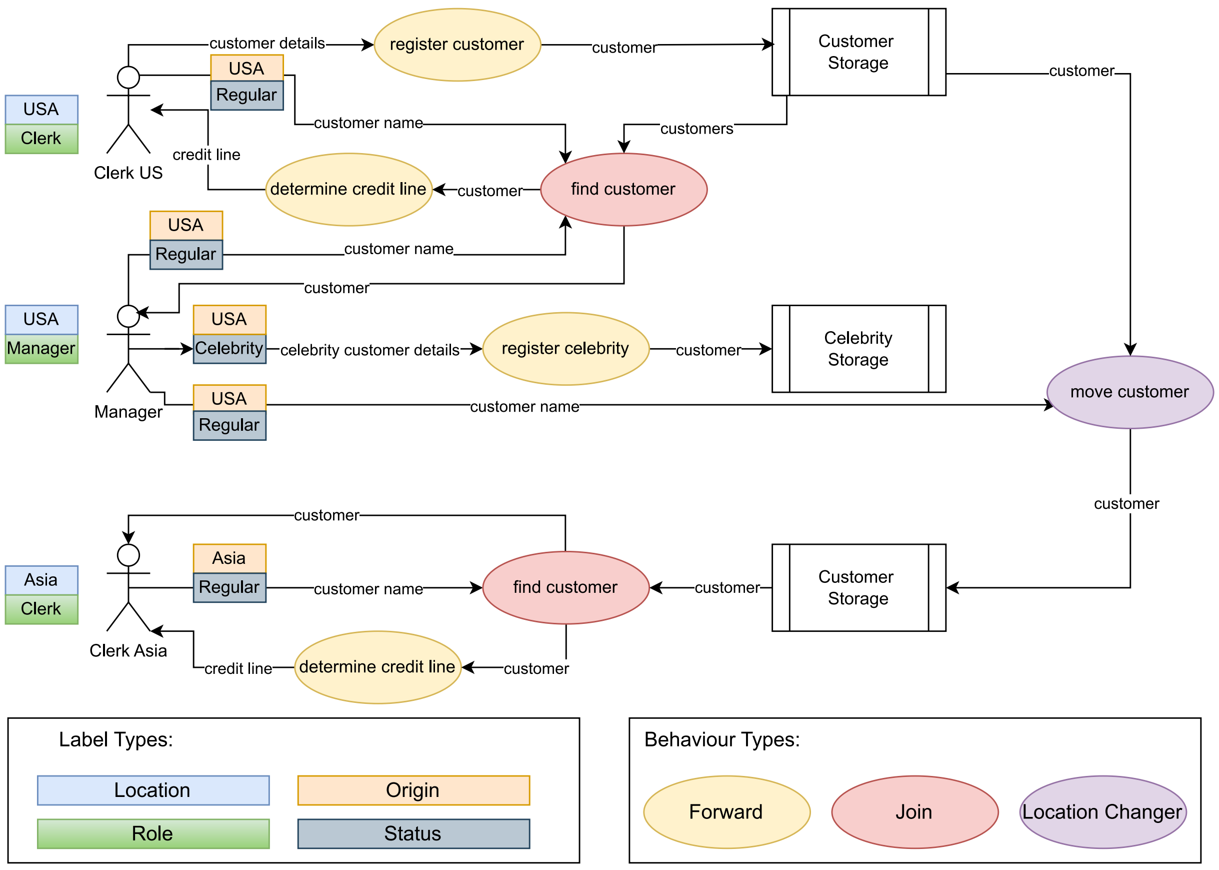

Figure 7.6: ABAC Use Case Extended Data Flow Diagram

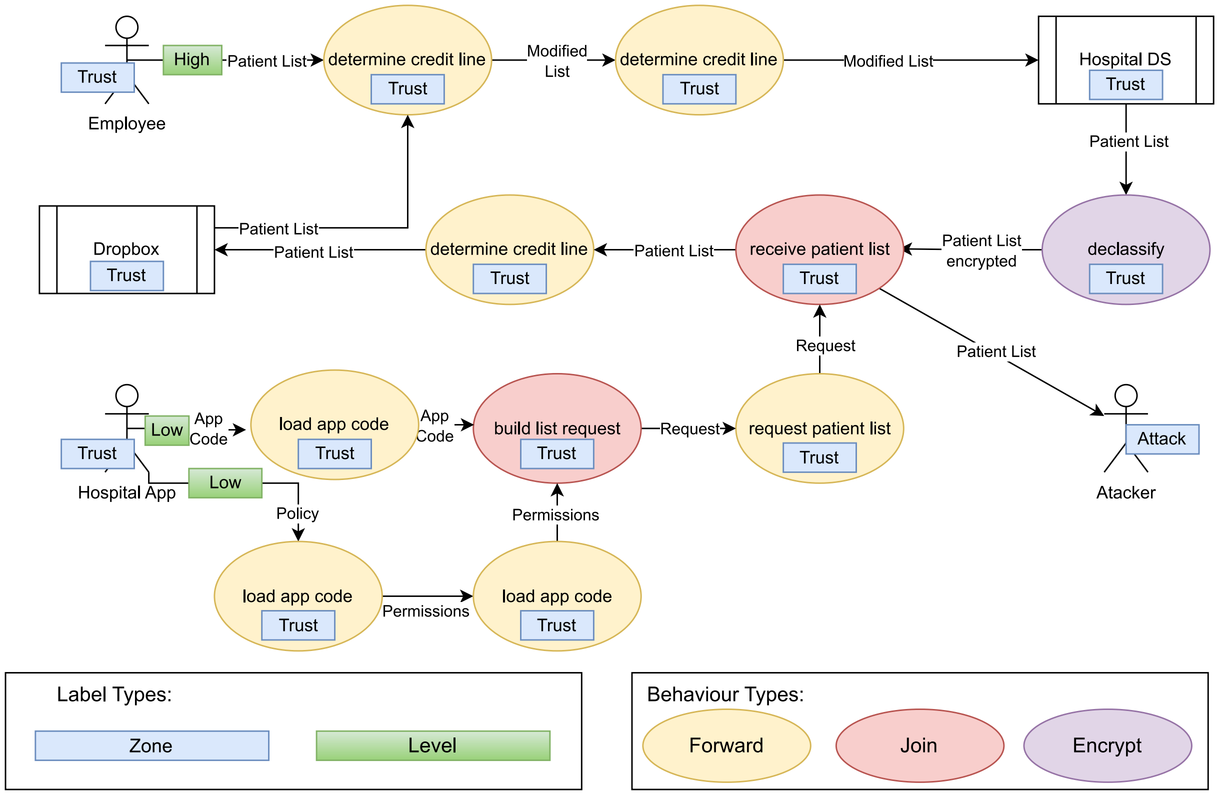

Figure 7.7: Hospital Use Case Extended Data Flow Diagram

Setup for Question Q2.1.2
To answer question Q2.1.1, we will transform the Running Example, described in Chapter 4, into a DFD metamodel instance. Then we compare it to a manually created Gold-Standard.

Setup for Question Q2.2.1
To answer question Q2.2.1, we will transform the scenarios already defined for Q2.1.1 into the GDPR metamodel and then back into the DFD metamodel. Then we will compare the results to the original model instances.

Setup for Question Q2.2.2
To answer question Q2.2.2, we will transform our Running Example, described in Chapter 4, first into the DFD metamodel and then back into the GDPR metamodel. Then we compare the results to the original model instance.

7.2.2 Evaluation Setup for Scalability

All scalability tests are run on the following configuration:

- Ryzen 7 2700X (8 core, 16 threads) @ 4.1 GHz
- 32GB RAM @ 3333MHz CL18
- 1TB NVMe PCIe3.0 SSD

- Microsoft Windows 11 Pro
- Java (Version 17.0.8 LTS)

For each question, we will run the setups by increasing the number of the specified model elements. We will start with 1 and then double the amount until we reach 4096. Each setup is then run 11 times, with the first value being discarded due to Eclipse inefficiencies, which offers a good variance and reduces the effects of the JVM.

To create large GDPR metamodel instances with minimal manual effort, we implemented a test-creation tool that can create model instances of the desired size. To create a model, one first specifies the total number of *Processing* elements and the size of *followingProcessing* of each *Processing* element. This will create a tree-like structure, with the root always being an instance of the *Collecting* class. Unless the desired number of *Processing* elements is reached, each *Processing* will be assigned the previously specified number of newly created *Processing* elements as *followingProcessing*. These are randomly created as either *Processing* or any subtype of it besides *Collecting*. The root *Collecting* and all other *Processing* also get assigned a *Purpose* for the collection of data since that needs to be carried through all *followingProcessing* of a *Collecting* instance. Next one can specify the number of *Natural Persons*, individual *Purposes*, and *Controllers*. For each *Natural Person*, all possible *Legal Bases* are created and all *Natural Persons* are assigned as data subjects for all *Processing*. *Purposes* are created according to the number specified, assigned to a random *Controller*, and then a random number of random *Purposes* is assigned to each *Processing*. For each *Natural Person* on each *Processing*, there is also a random valid *Legal Basis* assigned to the *Processing*.

Setup for Question Q3.1

The first question for evaluating scalability covers the change in execution time of the analysis tool when the amount of *Processing* is increased. We will set the number of *followingProcessing* to 3 in our test creation tool and increase the number of total *Processing* from 1 to 4096 by doubling it every step. The number of *Natural Persons*, individual *Purposes*, and *Controllers* will be set to 1.

Setup for Question Q3.2

Question Q2.2 aims to evaluate the scalability in the case of an increasing number of *Natural Persons*. We will set the total number of *Processing* to 5, the number of *followingProcessing* to 2, and increase the number of *Natural Persons* from 1 to 4096 by doubling it every step. The number of individual *Purposes* and *Controllers* will be set to 1.

Setup for Question Q3.3

To answer question Q2.3, regarding the scalability in the case of an increasing number of individual *Purposes*, we will set the total number of *Processing* to 5, the number of *followingProcessing* to 2, and increase the number of *Purposes* from 1 to 4096 by doubling it every step. The number of *Natural Persons* and *Controllers* will be set to 1.

To evaluate the scalability of our transformation approach, we will measure the execution time with and without model loading to get a better understanding of the scalability of the actual transformation.

Setup for Question Q4.1

To answer question Q4.1, we create increasingly complex DFD model instances. We increase the number of *Nodes* from 1 to 4096 by doubling each step. Each *Node* will be assigned a test *Label* as a property and an assignmentless Behavior. For each *Node*, there is a random number of *Flows* to other *Nodes*.

Setup for Question Q4.2 to Q4.4

Setups for Q4.2 to Q2.4 will be analogous to the setups for questions Q2.1 to Q2.3, respectively.

7.3 Evaluation Results

In this section, we present and discuss the results of our evaluation regarding the questions defined in the previous sections.

7.3.1 Findings and Discussion on Accuracy

This section covers the results and the discussion of our findings regarding the accuracy of both our analysis and transformation approaches. We calculate the proposed metrics precision, recall, and F_1 and discuss the meaning of the calculated values.

Findings on Q1.1

We ran the analysis tool for the proposed modified model instance. We did not falsely identify any violation of the GDPR. The analysis tool did identify two issues for the modified model instance on the *Processing* we expected. This results in $t_p = 2$ and $f_p = 0$ and a precision of $p = \frac{t_p}{t_p+f_p} = 1.0$. Since all and only expected violations have been identified, we calculate a recall of $r = \frac{t_p}{t_p+f_n} = \frac{2}{2+0} = 1.0$ and $F_1 = 2\frac{p*r}{p+r} = 1.0$. These results show that the analysis tool can identify the issue described in scenario S1.

Findings on Q1.2

The analysis tool correctly identified violations on every single *Processing*, since without *Proof*, Joe's *Consent* is not valid. This results in $t_p = 5$ and $f_p = 0$ and a precision of $p = \frac{t_p}{t_p+f_p} = 1.0$. Since all and only expected violations have been identified, we calculate a recall of $r = \frac{t_p}{t_p+f_n} = \frac{5}{5+0} = 1.0$ and $F_1 = 2\frac{p*r}{p+r} = 1.0$. These results show that the analysis tool can identify the issue of missing *Proof* of *Consent*.

Findings on Q1.3

The analysis tool correctly identified violations on the two *Processing* whose *Purpose* was removed from Joe's *Consent*. This results in $t_p = 2$ and $f_p = 0$ and a precision of $p = \frac{t_p}{t_p+f_p} = 1.0$. Since all and only expected violations have been identified, we calculate a recall of $r = \frac{t_p}{t_p+f_n} = \frac{2}{2+0} = 1.0$ and $F_1 = 2\frac{p*r}{p+r} = 1.0$. These results show that the analysis tool can identify the issue of missing *Consent* for processing *Purposes*.

Findings on Q1.4

The analysis tool correctly identified violations on all *Processing*, since Joe's *Consent* is not valid for *Purposes*, decided over by another *Controller*. This results in $t_p = 5$ and $f_p = 0$ and a precision of $p = \frac{t_p}{t_p+f_p} = 1.0$. Since all and only expected violations have been identified, we calculate a recall of $r = \frac{t_p}{t_p+f_n} = \frac{5}{5+0} = 1.0$ and $F_1 = 2\frac{p*r}{p+r} = 1.0$. These results show that the analysis tool can identify the issue of *Consent* being held by the wrong *Controller*.

Findings on Q1.5

The analysis tool correctly identified violations on the two *Processing*, which weren't defined with the *Purpose* defined for the collection of data. This results in $t_p = 2$ and $f_p = 0$ and a precision of $p = \frac{t_p}{t_p+f_p} = 1.0$. Since all and only expected violations have been identified, we calculate a recall of $r = \frac{t_p}{t_p+f_n} = \frac{2}{2+0} = 1.0$ and $F_1 = 2\frac{p*r}{p+r} = 1.0$. These results show that the analysis tool can identify the issue of the collecting *Purpose* not being carried through the entire system.

Findings on Q2.1.1

We ran the transformation for the two proposed scenarios, ABAC and HospitalApp. The transformation transformed all model elements correctly in both cases and no additional erroneous elements were created. For this transformation, we only look at the elements of the Data Flow Diagram and not the elements of the *Data Dictionary*, since they are not transformed but instead saved via the Tracemodel. Since both scenarios have the same number of DFD elements, this means $t_p = 31$ and $f_p = 0$, and therefore a precision of $p = \frac{t_p}{t_p+f_p} = 1.0$ in both cases. All model elements were transformed correctly, which leads to a recall of $r = \frac{t_p}{t_p+f_n} = \frac{31}{31+0} = 1.0$ for both scenarios and $F_1 = 2\frac{p*r}{p+r} = 1.0$. The results show that the transformation correctly transforms DFD metamodel instances into GDPR metamodel instances on initial transformations.

Findings on Q2.1.2

We ran the transformation for the proposed scenario, our running example. The scenario contains 19 elements that were all correctly transformed, be it directly or indirectly, into the DFD metamodel. No erroneous additional elements were created. We calculate a precision of $p = \frac{t_p}{t_p+f_p} = \frac{19}{19+0} = 1.0$, a recall of $r = \frac{t_p}{t_p+f_n} = \frac{19}{19+0} = 1.0$, and $F_1 = 2\frac{p*r}{p+r} = 1.0$. These results show that the transformation correctly transforms GDPR metamodel instances into DFD metamodel instances on initial transformations.

Findings on Q2.2.1

As described during the setup, we first transformed each of our scenarios into a GDPR metamodel instance and then back into a DFD metamodel instance. This transformation will create additional *Labels* and *LabelTypes* on the *Data Dictionary*, which will be considered additional erroneous elements since they were not part of the original source instance. Therefore, it is necessary to also consider the elements of the *Data Dictionary* for this question. Additionally, the transformation will add GDPR-specific *Labels* to *Node* properties. *Nodes* that get assigned additional properties (all of them) are considered wrongly transformed model elements since they deviate from the source elements. Figure 7.8 shows the results of our evaluation as well as the source model element count. Due to the "wrongly" transformed elements, we see a recall of less than 1.0, respectively. The additional *Labels* and *LabelTypes* mean a precision of 0.7921 and 0.6842, and therefore a F_1 of 0.8538 and 0.7675. This is according to our expectations since the transformation was designed this way. The additional *Labels* have no impact on an eventual analysis of the model instance and can even be used for a more comprehensive analysis. The accuracy for this question depends on the *Node* to *Flow* ratio since *Flows* are correctly transformed and more *Flows* mean more Behaviors that are also correctly transformed.

Base Scenario	DFD Elements	DD Elements	Total
ABAC	31	142	173
HospitalApp	31	88	119

Transformed Scenario	Correct DFD Elements	Erroneous DFD Elements	t_p	f_n	r
ABAC	18	13	160	13	0.9249
HospitalApp	16	15	104	15	0.8739

Transformed Scenario	Total DD Elements	Erroneous DD Elements	t_p	f_p	p	F_1
ABAC	215	42	160	42	0.7921	0.8538
HospitalApp	136	48	104	48	0.6842	0.7675

Figure 7.8: Findings on Question Q2.2.1

Findings on Q2.2.2

As described during the setup, we first transformed our running example into a DFD metamodel instance and then back into a GDPR metamodel instance. Since additional information of the DFD metamodel is stored via the Tracemodel instead of in the GDPR metamodel instance, the transformation result is equivalent to the source model instance. This means $t_p = 19$, $f_p = 0$, and $f_n = 0$ and therefore a precision, recall, and F_1 of 1.0. These results show that the transformation does work correctly when transforming GDPR model instances into the DFD metamodel and back.

7.3.2 Findings and Discussion on Scalability

In this section, we present and discuss our findings on the scalability of both our analysis and transformation approaches. For each question, we increased the number of environmental factors and measured the execution time. To provide an understandable visual representation we gathered and plotted our findings.

Findings on Question Q3.1

To answer question Q2.1, we increased the number of *Processing* elements in our GDPR metamodel instance. Figure 7.9 shows the results of our evaluation. The analysis tool's execution time shows exponential growth behavior for an increasing number of *Processing* elements. We can identify a tipping point at 64 *Processing* elements, where the execution time changes from being nearly constant to linear growth. Before that, most of the execution time is used on loading the model instances instead of creating and running the analysis. After passing 512 *Processing* elements we can identify an exponential growth behavior.

However, we believe that 4096 *Processing* elements are far beyond anything ever modeled with the GDPR metamodel. The GDPR metamodel is intended to be easily understandable and employable by software architects without extensive knowledge of data protection law. Big Data Flow Diagrams could be transformed into GDPR metamodel instances; however, it is still necessary to add the GDPR-specific elements to the model instances. We therefore reckon that model instances with more than 256 *Processing* elements will rarely be deployed, and most analyzed model instances will have far less. This results in an execution time of less than one second for most deployed model instances.

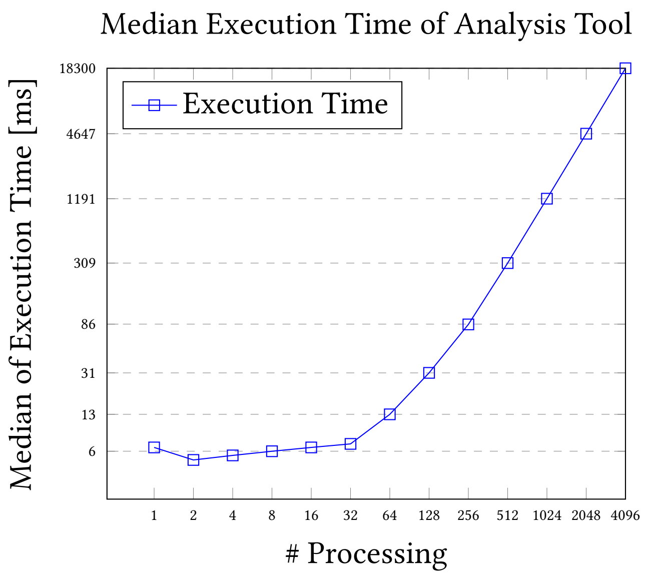

Figure 7.9: Findings on Question Q3.1

Findings on Question Q3.2

The execution time of the analysis tool for an increasing amount of *Natural Persons* modeled can be seen in Figure 7.10. Again, an exponential growth behavior can be observed, with the tipping point for linear growth being 32 and for exponential growth 128 Persons. The analysis tool's execution time scales considerably worse for *Natural Persons* than *Processing*. The execution for a model instance with 4096 *Natural Persons* takes over 5 minutes. However, we find it unlikely that a model instance with more than 32 *Natural Persons* will be employed, which means an execution time of under one second.

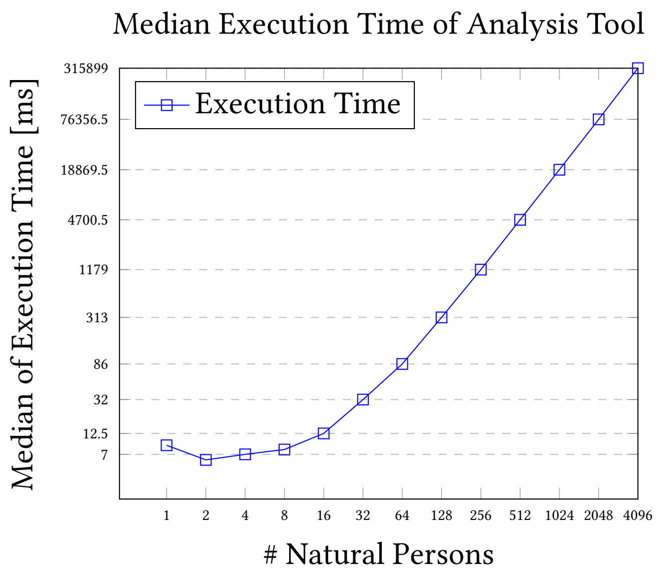

Figure 7.10: Findings on Question Q3.2

Findings on Question Q3.3

The execution time of the analysis tool for an increasing amount of *Purposes* modeled can be seen in Figure 7.11. Again, an exponential growth behavior can be observed, with the tipping point for linear growth being 32 and for exponential growth 128 *Purposes*. The analysis tool's execution time scales considerably worse for *Purposes* than *Processing*. The execution for a model instance with 4096 *Purposes* takes over 5 minutes. However, we find it unlikely that a model instance with more than 32 *Purposes* will be employed.

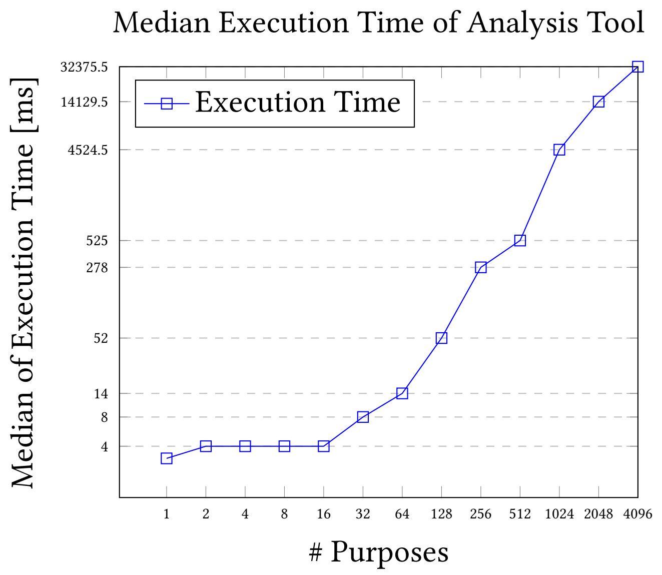

Figure 7.11: Findings on Question Q3.3

Findings on Question Q4.1

Figure 7.12 shows the execution time of the DFD2GDPR transformation for an increasing number of *Nodes*, both with and without model loading. The total execution time shows a tipping point for exponential growth behavior at 256 *Nodes*. Without model loading, the executing time stays almost constant up until 1024 *Nodes* and displays linear growth behavior afterwards. This means most of the increase in execution time can be attributed to model loading and not the transformation. We again expect most model instances transformed to have less than 256 *Nodes*, which means the transformation will be performed in under 1 second for most models deployed.

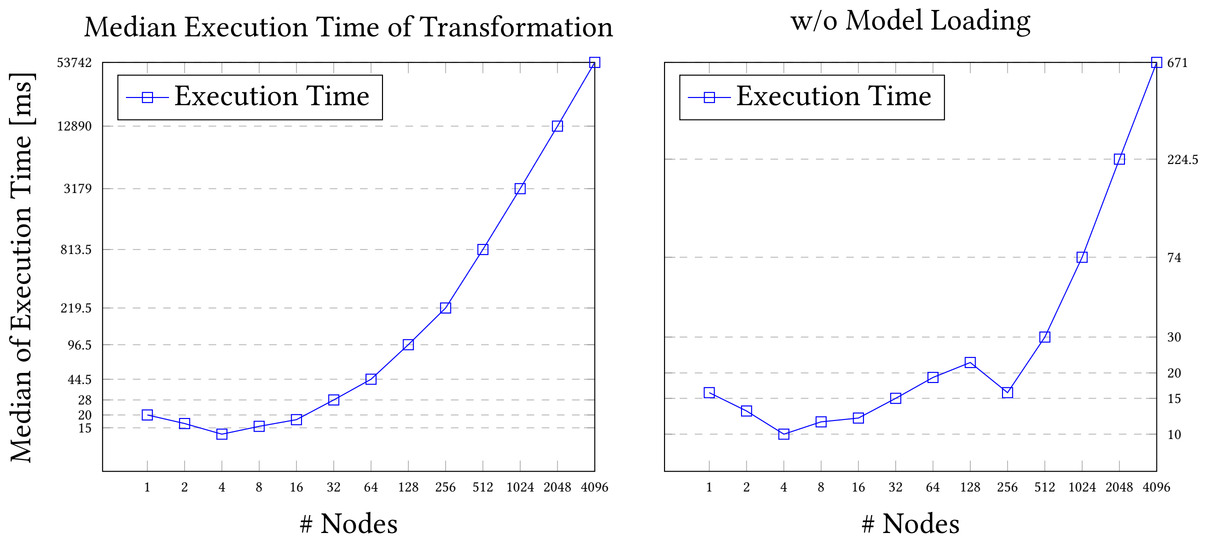

Figure 7.12: Findings on Question Q4.1

Findings on Question Q4.2

Figure 7.13 shows the execution time of the GDPR2DFD transformation for an increasing number of *Processing* elements, both with and without model loading. The total execution time shows a tipping point for exponential growth behavior at 512 *Processing* elements. Without model loading, the executing time stays almost constant up until 256 *Processing* elements and displays linear growth behavior afterwards. This again means that most of the increase in execution time can be attributed to model loading. We again expect most model instances transformed to have less than 256 *Processing* elements, which means the transformation will be performed in under 1 second for most models deployed.

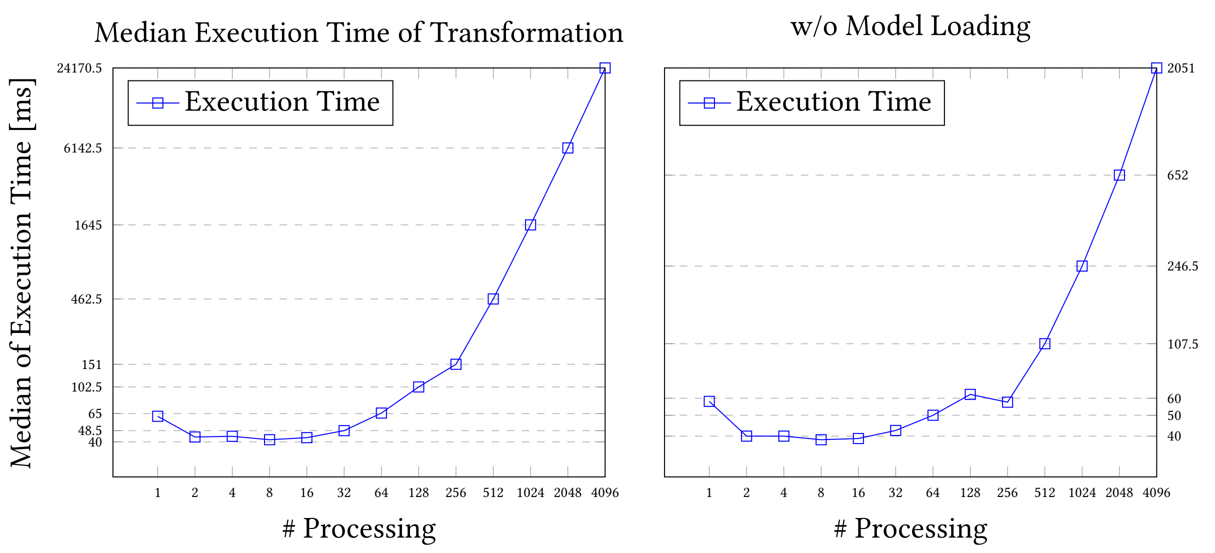

Figure 7.13: Findings on Question Q4.2

Findings on Question Q4.3

Figure 7.14 shows the execution time of the GDPR2DFD transformation for an increasing number of *Natural Persons* modeled, both with and without model loading. The total execution time shows a tipping point for exponential growth behavior at 256 *Natural Persons*. Without model loading, the executing time stays almost constant up until 128 *Natural Persons* and displays exponential growth behavior afterward. The results show that the number of *Natural Persons* has the biggest impact on the execution time of the transformation approach. This is expected since the number of Data elements as well as a *Legal Basis* is proportional to the number of *Natural Persons*. When doubling the number of *Natural Persons*, double the number of Data elements and *Legal Bases* need to be transformed as well. The transformation needs less than one second for transformations up to 256 *Natural Persons*, which is way above the number we expect from usual models.

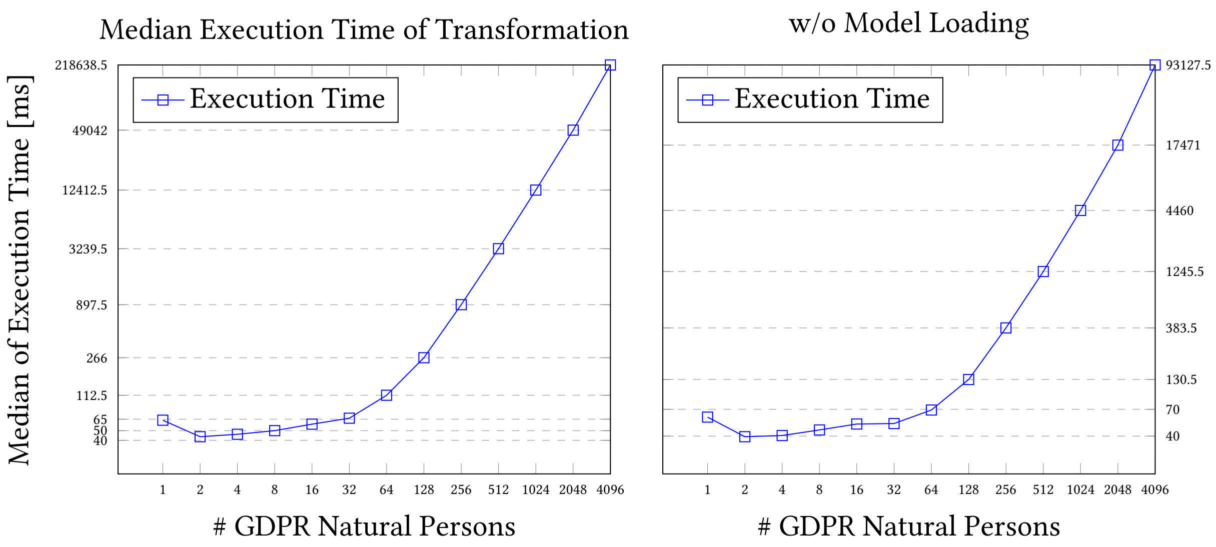

Figure 7.14: Findings on Question Q4.3

Findings on Question Q4.4

Figure 7.15 shows the execution time of the GDPR2DFD transformation for an increasing number of *Purposes*, both with and without model loading. The total execution time shows a tipping point for exponential growth behavior at 256 *Purposes*. Without model loading, the executing time stays almost constant up until 128 *Purposes* and displays linear growth behavior afterwards. After 1024 *Purposes*, exponential growth behavior can be observed. This shows that an increase in *Purposes* has a bigger impact on the execution time of both transformation and analysis than the number of *Processing* elements. This is according to expectations since *Purposes* can be attributed to multiple *Processing* elements, which increases complexity. Again, we don't expect model instances deployed to have more than 32 *Purposes*, but the transformation tools execution time will stay under 1 second for up to 256 *Purposes*.

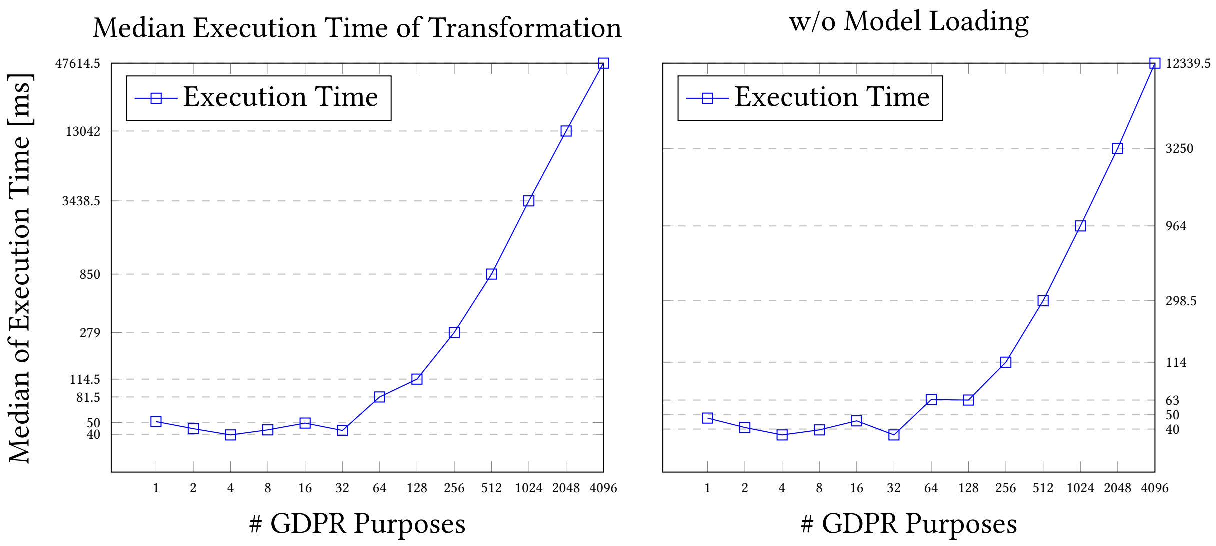

Figure 7.15: Findings on Question Q4.4

7.4 Threats to Validity

This section covers construct validity, internal validity, external validity, and reliability of our approach, as defined by Runeson et al. [26].

Construct Validity reflects whether the metrics defined are sufficient for answering the posed questions and whether the questions asked are suitable for evaluating the defined goals.

For evaluating the accuracy of our analysis tool, we chose the precision, recall, and F_1 metric. These are common for evaluating accuracy and are used in related work [4]. These metrics are also employed for evaluating the accuracy of the transformation approach and are similarly used in related work [12].

We decided to measure the execution time to evaluate the scalability of both the analysis and the transformation approach. This is common in the domain of system architecture modeling, where the system models can vary in size. This metric was used in related works for both analysis [7] and transformation [15] approaches.

Internal Validity ensures that no unknown factors affect the investigated factors, in our case accuracy and scalability.

The internal validity of our scalability evaluation might be threatened by the fact we only evaluated the change in execution time for the increase of 3 different elements of the GDPR metamodel, *Processing*, *Natural Persons*, and *Purposes*. But all other elements either depend on one of those 3 factors or don't have any impact on the execution time of the analysis approach. The amount of *Personal Data* is the same as the number of *Natural Persons*, while the number of *Legal Bases* also scales proportionally with that. The number of *Consents* also scales proportionally with the number of *Purposes*, since each *Purpose* in our test creator gets assigned to an individual *Consent*. The number of *Controllers* has a negligible impact on execution time, as long as the number of *Purposes* is not raised significantly, since *Controllers* are only referenced for verifying *Proof* of *Consent* and *Purpose*, which isn't using recursion.

For measuring the scalability of the DFD2GDPR transformation we only increased the number of *Processing* elements. This, however, is sufficient since we designed to number of *Flows* to increase with them as they would in a normal model instance. Adding more elements to the *Data Dictionary* has no impact on the execution time of only the transformation since they are not considered for initial transformations.

External Validity assures that findings are only generalized if they are applicable to other similar situations or events.

The biggest threat to the external validity of our accuracy evaluation is the limited number of violations we tested for. However, at the point of writing this thesis, these are all possible violations that we were able to extract from the GDPR and represent in the metamodel with our limited knowledge of data protection law.

The external validity of our scalability evaluation might be threatened by question Q2.3. As the analysis tool only analyses processing *Purposes* in the case of *Consent* (and collection *Purpose* carryover), the number of *Purposes* only has a significant impact on execution time if *Consent* is used as a *Legal Basis* on at least a single *Processing* element. To mitigate this threat, we ensured that *Consent* is picked as a *Legal Basis* on at least one *Processing* for every setup.

The biggest threat to the external validity of our transformation evaluation is the manually created models and gold standards used for testing. As the GDPR metamodel has not been published yet, there are no scenarios to reuse. To mitigate this threat, the Running Example in Chapter 4 was designed to include all possible elements and valid constellations of the GDPR metamodel.

7.5 Assumptions and Limitations

In this section, we discuss the assumptions made during the design and implementation, as well as the limitations of our approaches.

First, even though we designed our approaches to be used by architects without extensive knowledge of the GDPR, we still assume a basic understanding of the GDPR, especially regarding legal bases. A software architect without this understanding is still able to identify violations with the analysis tool; however, it would be difficult to understand the nature of the violation.

A limitation our approaches are facing is rooted in our limited knowledge of data protection law. We extracted all violations we could identify in the GDPR with our limited knowledge. A legal domain expert might be able to extract more violations and represent them in the GDPR metamodel.

Our approach should allow architects to quickly transform between the DFD and GDPR metamodel and allow analysis on both sides of the transformation. However, the old analysis tool by Seifermann et al. [29] was deprecated, and the integration of the DFD metamodel into the new analysis tool [27] is not completed yet.

7.6 Data Availability

All data of this thesis is made publicly available in our dataset [16]. We include all created code, model instances used for the evaluation, testing environments, and more detailed scalability results.

8 Conclusion

In this thesis, we provided approaches for transforming and subsequently analyzing system architecture artifacts into a metamodel suited for modeling the complex requirements of the GDPR. We designed and developed transformations between an extended Data Flow Diagram metamodel and a metamodel for GDPR compliance. The transformations work in both directions and without loss of information. We also provide an analysis tool that is able to identify GDPR violations.

We have developed an analysis tool for identifying violations of the GDPR. We drew the violations directly from the GDPR and designed the analysis tool to find all violations we were able to model in the GDPR metamodel with our limited knowledge of data protection law. Our analysis tool works by first loading a GDPR metamodel instance of the system, which is then parsed and transformed into a Prolog representation, as facts. The analysis tool matches the created facts against a set of Prolog terms representing GDPR violations. If the analysis tool finds a match, it generates a report that identifies the violation and the collection or processing of data that caused it.

We provided transformations between the previously presented extended DFD metamodel [29] and a GDPR metamodel developed by Boltz. We analyzed both models for semantically equivalent elements that could be directly transformed into each other. For elements without equivalent counterparts in the other model, we designed mechanisms and an additional Trace-model. This ensures the consistency of the transformations without any loss of information.

Our approach enables software architects without extensive knowledge of data protection law to extract legal specifications from regular design artifacts. After finishing a Data Flow Diagram, the architect can quickly transform the model instance into the GDPR metamodel. After annotating the *Legal Bases*, *Purposes*, and *Natural Persons*, the architect can then identify violations on the metamodel. If a violation is found, the architect can fix the system in the GDPR metamodel and then transform it back into the Data Flow Diagram metamodel. The retention of annotated legal information enables architects to efficiently transition between models while preserving the legal specifications.

In our evaluation, we discussed and presented the accuracy and scalability of our approaches. To evaluate the accuracy, we reused scenarios from other works when possible. For the approaches that lacked related literature, we devised our own scenarios and gold standards. Our approaches generally show precision, recall, and F_1 of 1.0. The only exception is DFD metamodel instances being transformed into the GDPR metamodel and back. During that process, additional model elements are created on the resulting DFD metamodel instance that store legal information only representable in the GDPR metamodel. To demonstrate the scalability of our approaches, we evaluated their ability to handle large model instances in a reasonable time. We found that even for model instances larger than those we expect to be used in practice, our approaches were able to complete in under one second.

8.1 Outlook

The approaches presented in this thesis can be deployed in their standalone form. However, we identified multiple aspects that can be further enhanced in future work.

The GDPR metamodel aims to include multiple of the most severe violations of GDPR regulations. However, there are still more possible regulations, like different handling of privileged data or processing of a minor's data, that have not yet been incorporated into the metamodel. If these changes are added to the metamodel, then the analysis tool must be extended to find these possible violations. If additional elements are necessary for modeling these violations, the transformations must also be adjusted.

Our approach provides standalone tools for the transformation and GDPR analysis. An analysis tool for DFD metamodel instances is currently in the works and will be completed soon. Long term a tool incorporating all, transformations and analysis tools, could be developed to provide a more streamlined tool for system architects.

8.2 Acknowledgements

I thank Sebastian Hahner and Maximilian Walter for their advice and suggestions while writing this thesis. Finally, I would like to express my sincere thanks to my advisor, Nicolas Boltz, for giving me the opportunity to write this thesis, as well as for his continuous availability, support, and feedback during this thesis.

Bibliography

[1] Abdulrahman Alhazmi and Nalin Asanka Gamagedara Arachchilage. “I’m all ears! Listening to software developers on putting GDPR principles into software development practice”. In: *Pers. Ubiquit. Comput.* 25.5 (Oct. 2021), pp. 879–892. ISSN: 1617-4917. DOI: 10.1007/s00779-021-01544-1.

[2] Thibaud Antignac, Riccardo Scandariato, and Gerardo Schneider. “A Privacy-Aware Conceptual Model for Handling Personal Data”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques.* Springer, Oct. 2016, pp. 942–957. DOI: 10.1007/978-3-319-47166-2_65.

[3] Thibaud Antignac, Riccardo Scandariato, and Gerardo Schneider. “Privacy Compliance Via Model Transformations”. In: *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW).* IEEE, Apr. 2018, pp. 23–27. DOI: 10.1109/EuroSPW.2018.00024.

[4] Steven Arzt et al. “FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps”. In: *SIGPLAN Not.* 49.6 (June 2014), pp. 259–269. ISSN: 0362-1340. DOI: 10.1145/2666356.2594299.

[5] Victor R. Basili and David M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *IEEE Trans. Software Eng.* SE-10.6 (Nov. 1984), pp. 728–738. DOI: 10.1109/TSE.1984.5010301.

[6] David Basin, Jürgen Doser, and Torsten Lodderstedt. “Model driven security for process-oriented systems”. In: *SACMAT ’03: Proceedings of the eighth ACM symposium on Access control models and technologies.* Association for Computing Machinery, June 2003, pp. 100–109. ISBN: 978-158113681. DOI: 10.1145/775412.775425.

[7] Nicolas Boltz. *Architectural Uncertainty Analysis for Access Control Scenarios in Industry 4.0.* 2021. DOI: 10.5445/IR/1000135847.

[8] Identity Theft Resource Center. *2022 Data Breach Report.* 2023. URL: https://www.idtheftcenter.org/publication/2022-data-breach-report/ (visited on 10/07/2023).

[9] Maurice Dawson et al. “Integrating Software Assurance into the Software Development Life Cycle (SDLC)”. In: *Journal of Information Systems Technology and Planning* 3 (Jan. 2010), pp. 49–53. URL: https://www.researchgate.net/publication/255965523_Integrating_Software_Assurance_into_the_Software_Development_Life_Cycle_SDLC (visited on 10/07/2023).

[10] Tom DeMarco. “Structure Analysis and System Specification”. In: *Pioneers and Their Contributions to Software Engineering.* Springer, 1979, pp. 255–288. DOI: 10.1007/978-3-642-48354-7_9.

[11] Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. “Model Transformations”. In: *Formal Methods for Model-Driven Engineering*. Springer, 2012, pp. 91–136. DOI: 10.1007/978-3-642-30982-3_4.

[12] Jean-Rémy Falleri et al. “Metamodel Matching for Automatic Model Transformation Generation”. In: *SpringerLink* (2008), pp. 326–340. DOI: 10.1007/978-3-540-87875-9_24.

[13] Joshua Gleitze. *A Declarative Language for Preserving Consistency of Multiple Models*. 2017. URL: https://publikationen.bibliothek.kit.edu/1000076905 (visited on 10/07/2023).

[14] *Guidelines on Data Protection Impact Assessment (DPIA) and determining whether processing is “likely to result in a high risk” for the purposes of Regulation 2016/679*. 2017. URL: https://ec.europa.eu/newsroom/article29/items/611236/en (visited on 10/07/2023).

[15] Robert Heinrich. “Architectural runtime models for integrating runtime observations and component-based models”. In: *Journal of Systems and Software* 169 (Nov. 2020), p. 110722. ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110722.

[16] Tom Hüller. *Dataset for Thesis "Automated Consistency of Legal and Software Architecture System Specifications for Data Protection Analysis"*. DOI: 10.5281/zenodo.8418730.

[17] Dominik Martin Huth. “Development of a reference process model for GDPR compliance management based on enterprise architecture”. PhD thesis. TECHNISCHE UNIVERSITÄT MÜNCHEN, 2021. URL: https://mediatum.ub.tum.de/doc/1593644/xcvtyuoiwljt4a91kz8h5qt8b (visited on 10/07/2023).

[18] Stuart Kent. “Model Driven Engineering”. In: *Integrated Formal Methods*. Springer, Apr. 2002, pp. 286–298. DOI: 10.1007/3-540-47884-1_16.

[19] Heiko Klare et al. “Enabling consistency in view-based system development — The Vitruvius approach”. In: *The Journal of Systems and Software* (171 2020). DOI: https://doi.org/10.1016/j.jss.2020.110815.

[20] Lenin Leines-Vite, Juan Carlos Pérez-Arriaga, and Xavier Limón. “CONFIDENTIALITY AND INTEGRITY MECHANISMS FOR MICROSERVICES COMMUNICATION”. In: *Computer Science and Information Technology (CS and IT)* (2021). DOI: https://doi.org/10.5121/csit.2021.111701.

[21] Raimundas Matulevičius et al. “A Method for Managing GDPR Compliance in Business Processes”. In: *Advanced Information Systems Engineering*. Springer, Aug. 2020, pp. 100–112. DOI: 10.1007/978-3-030-58135-0_9.

[22] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. June 2016. URL: https://www.omg.org/spec/QVT/1.3/PDF (visited on 10/07/2023).

[23] David M. W. Powers. “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation”. In: *International Journal of Machine Learning Technology* 2 (1 2020). DOI: https://doi.org/10.48550/arXiv.2010.16061.

[24] *Projog, An open source Prolog interpreter for Java*. URL: http://projog.org/ (visited on 10/07/2023).

[25] CMS Hasche Sigle Partnerschaft von Rechtsanwälten und Steuerberatern mbB. *GDPR Enforcement Tracker*. URL: https://www.enforcementtracker.com/?insights (visited on 10/07/2023).

[26] Per Runeson et al. *Case Study Research in Software Engineering*. Feb. 2012. ISBN: 978-1-11810435-4. DOI: 10.1002/9781118181034.

[27] Felix Schwickerath et al. *Tool-Supported Architecture-Based Data Flow Analysis for Confidentiality*. 2023. DOI: 10.48550/arXiv.2308.01645. eprint: 2308.01645.

[28] Stephan Seifermann et al. *Data Set of Publication on Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams*. URL: https://zenodo.org/record/5535599 (visited on 10/07/2023).

[29] Stephan Seifermann et al. "Detecting violations of access control and information flow policies in data flow diagrams". In: *Journal of Systems and Software* 184 (Feb. 2022), p. 111138. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111138.

[30] Laurens Sion et al. "An Architectural View for Data Protection by Design". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, pp. 25–29. DOI: 10.1109/ICSA.2019.00010.

[31] Thomas Stahl and Markus Völter. *Modellgetriebene Softwareentwicklung*. Jan. 2005. URL: https://voelter.de/data/books/mdsd_de1a.pdf (visited on 10/07/2023).

[32] Jake Tom, Eduard Sing, and Raimundas Matulevičius. "Conceptual Representation of the GDPR: Model and Application Directions". In: *Perspectives in Business Informatics Research*. Springer, Aug. 2018, pp. 18–28. DOI: 10.1007/978-3-319-99951-7_2.

[33] European Union. "General Data Protection Regulation". In: (2018). URL: https://gdpr.eu/tag/gdpr/ (visited on 10/07/2023).