Domain-specific Language for Data-driven Design Time Analyses and Result Mappings for Logic Programs

Master's Thesis of

Sebastian Hahner

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Stephan Seifermann
Second advisor:	M.Sc. Frederik Reiche

January 15, 2020 – August 17, 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, August 17, 2020

. . . . . .

(Sebastian Hahner)

Abstract

In today's connected world, exchanging data is essential to many business applications. With the increase in connectedness and the growing volume of data, ensuring security, privacy and conformance to legal restrictions becomes increasingly critical. In order to cope with these requirements early, design time data flow analyses have been proposed. By explicitly modeling data and their characteristics, the architectural model can be automatically tested against formulated data flow constraints. These verification approaches transform the modeled architecture into underlying formalisms such as logic programs. In order to enhance the expressiveness of constraints, they often have to be formulated in the underlying formalism as well. This requires architects to know about the formalism, the transformed architecture and the verification environment.

We aim to bridge the gap between the architectural domain and the underlying formalism which occurs in constraint formulation. We propose a domain-specific language (DSL) which enables architects to define constraints while defining the architecture. By utilizing the terminology which is used to model the architecture, individualized constraints can be formulated without knowledge on the verification process. We provide a mapping of constraints formulated in our DSL from the architectural domain to the underlying formalism. Analysis results are mapped back into the architectural domain to ease their interpretation.

The DSL is based on the generalization of existing constraints from real-world case studies. We evaluate the DSL's expressiveness, usability and space efficiency for different sized data flow restrictions. Approximately 75% of examined constraints can be expressed using the first version of our DSL while requiring up to 10 times less code. Besides the basics of data flow modeling and the modeling environment, no further knowledge on the transformation and verification mechanism is required. Additionally, we investigate the equivalence of analysis results of transformed constraints defined in our DSL with constraints formulated in the underlying formalism. In our tests, the transformed constraints achieve 100% recall while maintaining 90% precision.

Zusammenfassung

In der vernetzten Welt von Heute ist der Austausch von Daten für viele Anwendungen unerlässlich. Mit der zunehmenden Vernetzung und dem wachsenden Datenaufkommen wird die Gewährleistung von Sicherheit, Datenschutz und die Einhaltung rechtlicher Vorgaben immer wichtiger. Um diesen Anforderungen frühzeitig gerecht zu werden, können Datenflussanalysen zur Entwurfszeit eingesetzt werden. Durch explizite Modellierung der Daten und ihrer Eigenschaften kann das Architekturmodell automatisch gegen Datenflussbeschränkungen getestet werden. Diese Verifikationsansätze transformieren die modellierte Architektur in ihnen zugrunde liegende Formalismen wie z.B. logische Programme. Um die Aussagekraft der Beschränkungen zu erhöhen, müssen diese oft ebenfalls unter Nutzung des Formalismus ausgedrückt werden. Dies erfordert von den Architekten Kenntnisse über den Formalismus, die transformierte Architektur und die Verifikationsumgebung.

Unser Ziel ist es, die Lücke zwischen der architektonischen Domäne und dem zugrundeliegenden Formalismus zu schließen, die bei der Formulierung von Datenflussbeschränkungen auftritt. Wir schlagen eine domänenspezifische Sprache (DSL) vor, die es Architekten ermöglicht, Einschränkungen bereits während der Definition der Architektur festzulegen. Durch die Verwendung der selben Terminologie, die auch zur Modellierung der Architektur eingesetzt wird, können individualisierte Beschränkungen ohne Kenntnisse des Überprüfungsprozesses formuliert werden. Zusätzlich stellen wir eine Abbildung der in unserer DSL formulierten Einschränkungen von der Architekturdomäne in den Formalismus vor. Analyseergebnisse werden in die Architekturdomäne zurück abgebildet, um deren Interpretation zu erleichtern.

Die DSL basiert auf der Sammlung und Generalisierung bestehender Einschränkungen aus realen Fallstudien. Wir bewerten die Aussagekraft, Nutzbarkeit und Kompaktheit der DSL für Datenflussbeschränkungen unterschiedlicher Größe. Ungefähr 75% der untersuchten Beschränkungen können mit der ersten Version unserer DSL ausgedrückt werden, wobei bis zu 10-mal weniger Quelltext benötigt wird. Neben den Grundlagen der Datenflussmodellierung und Wissen über die Modellierungsumgebung sind keine weiteren Kenntnisse über den Transformations- oder Verifikationsmechanismus erforderlich. Zusätzlich untersuchen wir die Äquivalenz der Analyseergebnisse von Beschränkungen, die in unserer DSL formuliert wurden mit Beschränkungen, welche direkt unter Nutzung des Formalismus ausgedrückt wurden. In unseren Tests erreichen Beschränkungen, welche mit Hilfe unserer DSL formuliert wurden, eine 100%ige Ausbeute bei einer Präzision von 90%.

To Alina, Beate and Markus

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1
- 1.1. Contribution . . . 2
- 1.2. Outline . . . 2

2. Foundations 5
- 2.1. Data Flow Diagrams . . . 5
- 2.2. Data-Centric Palladio . . . 6
- 2.3. Domain Specific Languages . . . 7
- 2.4. Logic Programming . . . 8

3. Related Work 11
- 3.1. Alternative Analysis for PCM . . . 11
- 3.2. Security Modeling . . . 12
- 3.3. Data Flow Oriented Modeling . . . 12
- 3.4. Comparison . . . 13

4. Example Scenarios 15
- 4.1. Geolocation Constraints . . . 15
- 4.2. Access Control . . . 16

5. Supporting the Existing Analysis Process 21

6. Language Requirements 23
- 6.1. Case Studies . . . 23
- 6.2. Underlying Formalism . . . 27

7. Language Syntax 31
- 7.1. Overview . . . 31
- 7.2. Types and Imports . . . 33
- 7.3. Selection . . . 35
- 7.4. Conditions . . . 38
- 7.5. Constraints . . . 44
- 7.6. Example Instances . . . 46
  - 7.6.1. Geolocation Constraints . . . 46
  - 7.6.2. Access Control . . . 49

8. Language Semantics **53**
8.1. Overview . . . 53
8.2. Selection . . . 57
8.3. Conditions . . . 62
8.4. Constraints . . . 67
8.5. Result Mapping . . . 70
8.6. Example Transformations . . . 72
8.6.1. Geolocation Constraints . . . 72
8.6.2. Access Control . . . 76

9. Evaluation **79**
9.1. Goals and Questions . . . 79
9.2. Evaluation Design . . . 81
9.2.1. Studied scenarios . . . 81
9.2.2. G1 - Expressiveness . . . 83
9.2.3. G2 - Usability . . . 84
9.2.4. G3 - Space Efficiency . . . 87
9.2.5. G4 - Equivalence of Analysis . . . 88
9.3. Results and Discussion . . . 91
9.3.1. G1 - Expressiveness . . . 91
9.3.2. G2 - Usability . . . 96
9.3.3. G3 - Space Efficiency . . . 102
9.3.4. G4 - Equivalence of Analysis . . . 106
9.3.5. Summary . . . 112
9.4. Threats to Validity . . . 113
9.5. Assumptions and Limitations . . . 114
9.6. Data Availability . . . 115

10. Conclusion **117**
10.1. Summary . . . 117
10.2. Future Work . . . 118
10.3. Acknowledgements . . . 119

Bibliography **121**

A. Appendix **125**
A.1. Well-formedness of Conditions . . . 125
A.2. Correctness Evaluation . . . 128
A.3. Mapped Prolog Code Examples . . . 135
A.4. Meta-Model . . . 137

List of Figures

2.1. Simple data flow scenario of user input data processing 6
2.2. SEFF with data flow annotation 7
2.3. A simple class diagram 8

4.1. Deployment diagram of the Geolocation Constraints example scenario . 16
4.2. Deployment diagram of the Travel Planner example 18
4.3. Sequence diagram of the Travel Planner example 19

5.1. Extended development process with Data-Centric Palladio 22

7.1. Simplified abstract syntax of constraints in our DSL 32
7.2. Abstract syntax of top-level elements 34
7.3. Abstract syntax of data and destination selection as well as classes 35
7.4. Abstract syntax of conditions 39
7.5. Abstract syntax of exemplary operations and references to variables . . . 40
7.6. Abstract syntax of condition operations (simplified) 41
7.7. Abstract syntax of constraints and their contained elements 44
7.8. Geolocation constraints example instance using the DSL 47
7.9. Geolocation constraints example instance with characteristic classes . . 48
7.10. Travel Planner access control example instance using the DSL 50

8.1. Data flow diagram of the transformation and solving process 54
8.2. Object diagram of four different CharacteristicTypeSelectors 58
8.3. Abstract syntax tree of a mapped characteristic selector 58
8.4. Abstract syntax tree of an exemplary condition in the DSL 64
8.5. Abstract syntax tree of the transformation to Prolog of an exemplary condition 65

9.1. Adapted diagram of Morris on compiler correctness proofs 89

A.1. Complete abstract syntax of the meta-model of our DSL 138

List of Tables

3.1. Comparison of related work 14

4.1. Constraint violation examples in access control systems 17
4.2. Actors and roles in the Travel Planner example 17

7.1. All CharacteristicSetOperations which can be used in conditions 42
7.2. All BooleanOperations which can be used in conditions 43

8.1. Mapping of operations of a condition to Prolog predicates 63
8.2. Possible arguments created by the mapping of constraints 69

9.1. Expressiveness of predicates of the Constraint Query API 93
9.2. Collected operations on elements and sets 95
9.3. Change effort of different DSL change scenarios 96
9.4. Usage of modeling concepts in case study constraints 98
9.5. Required knowledge to define constraints with or without the DSL . . . 101
9.6. Number of model elements of scenarios' constraints 105
9.7. Change in number of elements in change scenarios 106
9.8. Number of found violations of different constraints 111

List of Listings

2.1. PlantUML DSL code of a simple class diagram 8
2.2. Simple Prolog example with facts and rules 9
2.3. Simple Prolog queries and results 9

6.1. Basic predicates of the Constraint Query API 28
6.2. Advanced predicates of the Constraint Query API 29

7.1. Simplified Geolocation Constraints example given in concrete syntax . . 33
7.2. Concrete syntax of type and import statements 35
7.3. Concrete syntax of selectors and characteristic classes 37
7.4. Concrete syntax of conditions using operations 44
7.5. Full-fledged constraint given in concrete syntax 45
7.6. Concrete syntax of the Geolocation Constraints example instance 48
7.7. Concrete syntax of the Access Control example instance 50

8.1. Simplified concrete syntax of the DSL prior to its transformation 56
8.2. Excerpt of the DSL transformation result represented as Prolog code . . . 56
8.3. Simplified textual representation of a violation after the result mapping . 57
8.4. Prolog code of the transformed four characteristic selector types 59
8.5. Prolog code of an transformed attribute selector and a property selector . 59
8.6. Return value and call state predicates 60
8.7. Concrete syntax of characteristic class and a attribute class selector . . . 61
8.8. Prolog code of a transformed characteristic class and its referencing . . . 61
8.9. Selector defining a characteristic set variable and the mapped result . . . 62
8.10. Nested operations and their Prolog counterpart 64
8.11. Complete structure of transformed constraints 68
8.12. Raw Prolog result of a single constraint violation 70
8.13. Complete textual representation of a mapped constraint violation 72
8.14. Shortened concrete syntax of the Geolocation Constraints example scenario 73
8.15. Excerpt of the DSL transformation result of the first Geolocation constraint 74
8.16. Excerpt of the DSL transformation result of the second Geolocation constraint 74
8.17. Prolog solving result of the Geolocation Constraints scenario 75
8.18. Mapped result of the Geolocation Constraints scenario 76
8.19. Shortened concrete syntax of the Access Control example scenario . . . 76
8.20. Excerpt of the DSL transformation result of the Access Control constraint 77
8.21. Exemplary Prolog solving result of the Access Control scenario 78
8.22. Mapped exemplary result of the Access Control scenario 78

9.1. Excerpt from the Xtext grammar which defines the DSL 99
9.2. Excerpt from the grammar which defines encapsulated operations 100
9.3. Concrete syntax of a minimal constraint 102
9.4. Concrete syntax of a large constraint 103
9.5. Concrete syntax of the UMLsec Secure Links constraints 104

A.1. Examples for conditions which consist of multiple operations 125
A.2. Shortened concrete syntax of constraints of both exemplary scenarios . . 135
A.3. Complete transformation result of the Geolocation Constraints scenario 136
A.4. Complete transformation result of the Access Control scenario 137

1. Introduction

In today's connected world, exchanging data is vital for a lot of business applications. To ensure software quality attributes like performance and reliability, the effect of data processing has to be considered early in the design process. By only considering the control flow of applications, some of these effects do not become visible. Data-driven analyses try to answer design questions by modeling the flow of data explicitly.

Data-driven architecture description is especially helpful in terms of security and privacy reasoning, because "people usually talk about confidentiality in terms of data rather than in terms of processes." [41]. To ensure properties such as confidentiality, the modeled data flow has to be analyzed. Automated data flow analyses are needed because "detecting confidentiality issues manually is not feasible" [41]. The prerequisite of automated analyses are formalized architectural models.

In order to aid the software architect, *Data-Centric Palladio* [41], an extension of the *Palladio Component Model (PCM)* [37] has been proposed. Explicitly modeled architectures allow verification at design time through techniques like simulation [37], model checking, and theorem proving [15]. *Data-Centric Palladio* uses the logic programming language *Prolog* to test modeled data flows against specified constraints [41]. An example is the restriction of confidential data like personal user details to never leave a specified part of a system, e.g. an internal, protected server.

A transformation chain enables automatic verification by mapping the modeled architecture to executable Prolog code [41, 25]. After the execution, constraint violations indicate security issues and can be traced back. In order to define specific constraints on the system's behavior and flow of processed data, the constraints have to be explicitly expressed by an architect using Prolog.

Here, multiple problems occur: First, it cannot be assumed, that software architects know the underlying formalism well enough and are able to express constraints without additional training. Second, knowledge about the automated transformation and the target meta-model is required since constraints are expressed against this meta-model and the resulting code from the transformation. Third, the analysis results represented as Prolog solution and are not mapped back to the architectural model which makes their interpretation more difficult.

This gap between the architectural domain and the underlying formalism can also be observed in related work [24, 26, 49, 15, 17, 23, 21, 34]. With higher analysis variability, more knowledge about the underlying formalism is required. For instance, using logical formulas [49, 15] is more difficult than using previously defined metrics [24, 26] or annotations [18, 34, 21].

In the following, we outline the contribution and give an overview of the thesis' structure and content.

1.1. Contribution

We propose an approach to enable modeling of constraints using the terminology and abstraction of the architectural domain without the need of knowledge about the underlying formalism. We aim to define a *domain-specific language (DSL)* to express constraints and analysis goals at design time. Using this language, an architect shall be capable of defining generic or individualized data flow constraints for a modeled system without knowledge of the underlying formalism Prolog. Afterwards, our approach transforms the constraint into executable Prolog code which checks whether the data restrictions hold for the modeled architecture. Last, possible constraint violations are transformed back to the architectural domain to ease the interpretation through the architect.

Besides the engineering aspect of providing ready-to-use tooling, we answer the following research questions as part of this thesis:

RQ1 Which architecture-level constructs and language elements are necessary in order to define data flow constraints?

RQ2 How is the mapping of constraints and results between architecture and formalism?

We aim to minimize the risk of rebuilding the formalism in another domain without respect to real-world use cases. Thus, we collect case studies from related work to answer these questions. We focus on analysis goals in order to find a generalized approach to express data flow constraints. Additionally, we consider the analysis capabilities of the underlying formalism and its usage in *Data-Centric Palladio* [41, 25] as influential factor regarding the expressiveness of our domain-specific language. We present a list of requirements for such a language.

To answer **RQ1**, we derive modeling concepts from these requirements and formalize them in a detailed meta-model in the architectural domain. Please note, that this does not imply a feature-finished language with the same possibilities as Prolog but rather an approach to enable to formulation of analysis goals for the majority of analyzed case studies.

We answer **RQ2** by extending the existing transformation approach from *Data-Centric Palladio* with a mapping from the newly defined domain-specific language to compliant Prolog code that reuses existing analysis capabilities [25]. Additionally, we define mappings for the analysis results from the executed Prolog program back to the architectural domain. Please note, that this step is not supposed to be equivalent to round-trip-engineering as seen in model synchronization because we expect different concepts in the analysis results than defined for the domain-specific language.

1.2. Outline

The remainder of this thesis is structured as follows: In Chapter 2, we discuss fundamentals like data flow modeling and domain specific languages. In Chapter 3, we present related work in the domain of transformation and analysis of architectural models. In Chapter 4, we summarize example scenarios used in the remainder of this thesis. We discuss the

integration of our approach in the existing analysis process of *Data-Centric Palladio* in Chapter 5. Afterwards, we collect language requirements in Chapter 6 which arise from case studies and the underlying formalism. We derive our domain-specific language's syntax from these requirements in Chapter 7. Then, we discuss the language's semantics, its transformation to Prolog code and the mapping of results in Chapter 8. In Chapter 9, we evaluate our approach. We conclude with Chapter 10 and provide a short outlook on future work.

2. Foundations

In this chapter, we introduce fundamentals needed to explain this thesis. First, in Section 2.1, we present the basic terminology used in data flow modeling. Then, in Section 2.2, we briefly explain *Data-Centric Palladio* [41], an extension of the *Palladio Component Model (PCM)* [37] to model and to analyze data flows. In Section 2.3, we focus on *domain-specific languages (DSLs)*. Last, the concept and terminology of logic programming is summarized in 2.4.

2.1. Data Flow Diagrams

A *Data Flow Diagram (DFD)* represents a systems structure by modeling interfaces and data flows inbetween [9]. The flow of control (e.g. one component calling another) is not in focus. DeMarco highlights that these diagrams "present the workings of a system as seen by the data, not as seen by the data processors" [9] which helps abstracting from individual operations. Data flows are often described using graphical notations. DeMarco provides rather informally semantics in natural language.

The following conceptual elements can be used inside of *Data Flow Diagrams* [9]:

- *Processes*, represented by named circles
- *Data Flows*, represented by annotated arrows between processes
- *Files*, represented by straight lines
- *Data Sources*, represented by boxes with only outgoing data flows
- *Data Sinks*, represented by boxes with only incoming data flows

A flow of data starts at a source - e.g. a user's input - is processed inbetween and is terminated in a data sink, e.g. stored on a server. We visualize this simple scenario with exemplary processes in Figure 2.1. Here, the input from the user is verified and then filtered using filter settings which originate from a file. Last, the filtered input is stored.

DeMarco defines data flows as "pipeline through which packets of information of known composition flow" [9]. This means that there is no upper limit of possible pipelines between two processes if data of different kind or with different purposes is sent. Through denoting multiple flows when applicable, the interface becomes more distinct. Data flow names shall represent what is known about the data and its state, e.g. by differentiating User Input and Filtered User Input in Figure 2.1. Naming data flows from files is optional since the file description is considered sufficient [9].

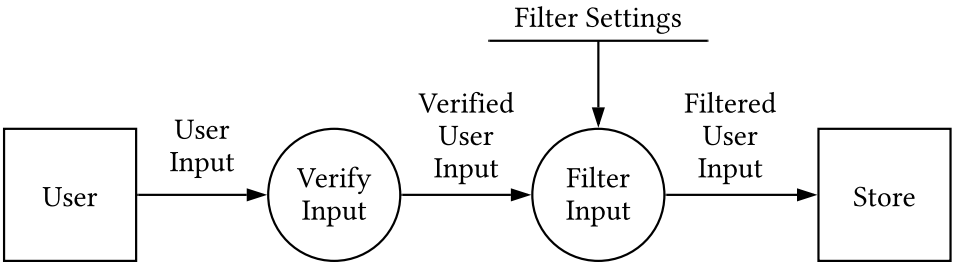

Figure 2.1.: Simple data flow scenario of user input data processing

Processes transform incoming data flows into outgoing data flows. As discussed above, any number of incoming and outgoing data flows are valid. DeMarco defines files as "temporary repository of data" [9] and thereby differentiates them from data sinks. Additionally, data sources or sinks are outside of the context of the modeled system. The Store-Element in Figure 2.1 is considered outside of the diagram's focus and thus modeled as sink. Otherwise, denoting it as file would be more accurate. Boxes in data flow diagrams can represent both a data source and a data sink simultaneously. Being outside the modeled system, data sources and sinks are not allowed to modify data. Only processes are allowed to transform data and are modeled explicitly inside the systems boundaries.

2.2. Data-Centric Palladio

Palladio is an approach to model software architectures and predict quality attributes such as performance and reliability [37]. The *Palladio Component Model (PCM)* is a domain-specific language for "specifying and documenting software architectural knowledge" [37]. Through analysis methods like simulation, complex software systems performance can be predicted early in the design process.

To achieve this, we specify several information: We encapsulate Component behavior in *Service Effect Specifications (SEFFs)*, which are stored in the *Component Repository Model*. We wire these components together using the *System Model*. We describe in the *Component Allocation Model* how the system is deployed on hardware. The hardware is defined in the *Execution Environment Model*. Last, the *Usage Model* describes the users behavior while interacting with the system. Combined, these five models allow the prediction of quality attributes like performance [37].

To allow further analysis regarding software quality attributes like security or privacy, *Data-Centric Palladio* has been proposed by Seifermann et al. [41]. Based on a "conventional software architecture model" [41], data-driven constraints can be added by the architect through custom annotations.

The meta-model of *Data-Centric Palladio* allows the specification of data flow descriptions using data, sources and sinks, characteristics and processing operations. We explain data flow modeling with sources, sinks and processes in Section 2.1.

Additionally, data and processes can hold characteristics. Characteristics are "named finite sets of values" [41]. These are used e.g. to express roles or access rights on data. The data flow analysis compares the characteristics of data and processes. Here, characteristics can be modeled explicitly by the architect or inferred implicitly. An example for this are data characteristics depending on the data's source. Thus, characteristics of data may change when beeng processed in the system. We use processing operations to express the transformation of data flows from source to sink. Examples are data filtering or aggregation [41]. To exchange data between components, we define data transmissions.

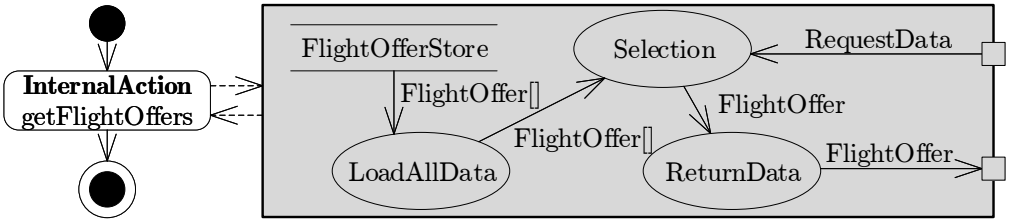

Figure 2.2.: SEFF with data flow annotation

Using the meta-model of *Data-Centric Palladio*, *Palladio*'s SEFFs can be annotated with data flow information, as shown in Figure 2.2 [41]. All specified data and processes can hold characteristics. The action getFlightOffers is described using a data flow from the FlightOfferStore to the user. Based on the requested data, results are processed (selection).

After modeling data flows, we can evaluate specified constraints automatically. For this analysis, the model is transformed into executable Prolog code. The output of the program provides insight whether or not the modeled architecture holds the specified requirements.

2.3. Domain Specific Languages

Several definitions of domain-specific languages can be found in other work [1, 20, 28, 29, 46]. Van Deursen et al. define them as follows: "A *domain specific language (DSL)* is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [46]. This definition holds multiple aspects of DSLs, we want to briefly discuss in the following.

DSLs offer appropriate notations for end-users and can also be called "end-user programming" [46]. Thus, the end-user needs no in-depth knowledge of formalisms and only needs to understand the DSL to "be more capable of handling more complex tasks." [33]. This can have a positive impact on the user's productivity [28]. Other opportunities of DSLs are self-documentation through meaningful notations, consistency, proper level of abstraction and enhanced maintainability [46]. Use cases are analysis, verification and optimization [28].

In comparison, a *general-purpose language (GPL)* offers a more generic approach: "A DSL offers appropriate domain-specific notations from the start" [28]. Additionally, GPLs tend

to require more boilerplate code. Conciseness shall be considered throughout development in order to avoid accidental complexity.

DSLs don't have to be directly executed [28] and can rather be transformed into an executable representation. To create a DSL, domain-specific knowledge has to be combined with language development skills which is considered "hard" [28].

In the following, we present PlantUML as an example for a DSL which shows both the enhanced intuitiveness and the restrictiveness to a defined domain. *PlantUML* [38] is a open-source DSL to textually define UML-diagrams. Figure 2.3 shows a simple class diagram, modeling the (simplified) relation of an *Object* and an *ArrayList* [38]. We show the textual representation in Listing 2.1. Even without seeing the diagram, PlantUML code can be understood due to its intuitiveness. However, this is not the case for all textual representations of UML diagrams, e.g. XML is worse readable.

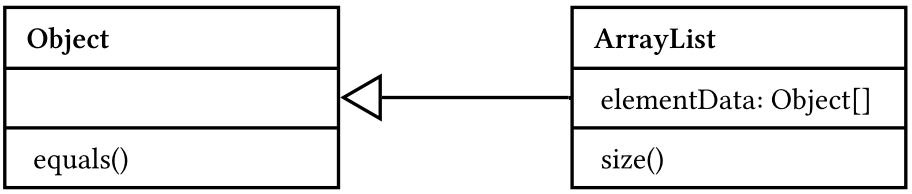

Figure 2.3.: A simple class diagram

```
1  @startuml
2  class Object {
3    equals()
4  }
5
6  class ArrayList {
7    elementData: Object[]
8    size()
9  }
10
11 Object <|-- ArrayList
12 @enduml
```

Listing 2.1: PlantUML DSL code of a simple class diagram

2.4. Logic Programming

Logic programs follow a different paradigm than "classic" procedural programming languages like Java or C++ [3]. The procedural paradigm consists of statements, which are executed from start to end. These can be encapsulated inside control structures like condi-

tional constructs or loops. In comparison, logic programs follow the declarative paradigm. They consist of facts and rules which can be combined and solved by the programming environment. The programmer does not specify *how* to solve the program.

A well known logic programming language is *Prolog*, which stands for "*Programming in Logic*" [3]. Using the Prolog environment, questions (called "queries") can be asked to a previously defined system of facts and rules. We provide a short example based on Bramer et al. [3] in Listing 2.2. The first two lines are facts which declare that fido and henry are dogs. The third line is a rule, specifying that anything can be called an animal, if it's a dog; or in more natural language: "All dogs are animals".

```
1   dog(fido).
2   dog(henry).
3   animal(X) :- dog(X).
```

Listing 2.2: Simple Prolog example with facts and rules

With this simple set of facts and rules, some first questions can be asked. To answer these, the Prolog environment solves the underlying logic program. We show queries and their results in Listing 2.3. The first query is straightforward: We ask if fido is a dog. Because of the first fact in Listing 2.2, this is evaluated to *true*. The second query asks if fido is an animal. Because of our rule in the last line of Listing 2.2, this is also *true*. In the last query, we ask if there can be found any facts about dogs, which evaluate to *true*. This is the case for both fido and henry. Based on these building blocks, complex logical relations can be formalized and solved. A detailed description how Prolog approaches such problems using unification and backtracking can be found in [3, Chapter 3].

```
1   ?- dog(fido).
2   true.
3   ?- animal(fido).
4   true.
5   ?- dog(X).
6   X = fido;
7   X = henry
```

Listing 2.3: Simple Prolog queries and results

In order to expand this basic understanding of the Prolog environment, we summarize important language elements and terminology [43].

Terms are a basic building block of Prolog *programs*. By adding *arguments*, they become *Compound Terms*. The number of arguments is specified by their *arity*. The next bigger building block is the *clause*, also called sentence, or rule [3]. It consists of a *head* and a *body*. The head represents the identity of the rule and can be expressed using an *atom* (a textual constant) or a *compound term* with arguments. The *body* itself is a *clause* and thus a *term*. The semantics are straightforward: The *body* implicates the *head*. In line 3 of Listing 2.2, we see such a *rule*: `animal(X) :- dog(X)`. The head and body are built up from *compound terms* with one argument and thus with an *arity* of one.

Facts are clauses without *body*. Thus, they have no condition which must be satisfied. In Listing 2.2, we can see such facts for fido and henry. A *predicate* represents a collection of clauses with the same *functor*, which means matching name and *arity* (like dog). A Prolog *program* is a collection of *predicates*.

After defining the *program*, the Prolog engine can answer questions called *goals* or *queries*, as shown in Listing 2.3. If a *goal* succeeds, a *solution* with bound variables is returned. The query fails if Prolog is unable to find a solution which satisfies the question.

3. Related Work

In this chapter, we give an overview of related work for this thesis. The common research field of all related work is the design time analysis of software architectures. Related work discusses the use of different formalisms to analyze software quality attributs like performance or to ensure security. In order to perform analyses, these approaches transform to formalisms which allow automated verification.

This includes approaches based on the *Palladio Component Model (PCM)* [24, 27] in Section 3.1, security modeling approaches like *UMLsec* [18] and *Software Architecture Models* [15, 49] in Section 3.2, and data flow oriented approaches like *Secure Data Flow Diagrams* [34] and *iFlow* [21] in Section 3.3. We close with a short comparison of related work and our approach in Section 3.4.

3.1. Alternative Analysis for PCM

The *Palladio Component Model* [37] is a DSL to document and evaluate software quality attributes like performance and reliability (please see Section 2.2 for fundamentals). To analyze these models, different approaches were proposed. We summarize the analysis using *Layered Queueing Networks (LQN)* [24] and *Queueing Petri Nets (QPN)* [27]. Both approaches use metrics which limit the expression possibilities of analysis goals of an architect.

Koziolek et al. present **A Model Transformation from the Palladio Component Model to Layered Queueing Networks** [24]. They propose an automated model transformation from the architectural layer to a connected LQN solver. Their goal is to create a reusable transformation approach with performance advantage over discrete-time simulation. The transformation is based on a defined mapping between PCM and LQN. The model is serialized afterwards and sent to the LQN solver for performance analysis. Users can change parameters of the architecture to quickly evaluate their influence on performance attributes. A limitation of their approach is the lack of solver feedback. Thus, analysis results are not processed but rather printed to the screen and have to be interpreted manually by the architect.

Meier et al. propose an **Automated Transformation of Component-based Software Architecture Models to Queueing Petri Nets** [27]. They use a model-to-model transformation from PCM to lower level QPNs and simulate the modeled system afterwards. The mapping is provided using QVTO. To enable user input and specification of desired analysis results, a "Instrumentation UI and Instrumentation Model" [27] is defined. Together with result integration, this approach lets the user view simulation results and identify performance problems without leaving the architectural abstraction level. The user can define the level of detail of desired results from simple aggregations to detailed

histograms [26]. This influences the amount of collected data and the simulation time. Although the results are mapped back to architectural domain, they are only "printed to the console and to a results file" [26] without further processing.

3.2. Security Modeling

The field of *Model Driven Security (MDS)* tries "to solve security-related questions at design time using model-driven approaches" [13]. Through early modeling of critical system aspects, costly changes in later steps of the software development process can be avoided. These approaches are related to our work because they also handle transformations from the architectural domain and analysis goal specification. We present the modeling approaches *Software Architecture Model (SAM)* [15, 49] and *UMLsec* [18]. These have already been discussed in previous work [13].

Yu and He et al. propose the usage of **Software Architecture Models** for security analysis of distributed systems [49] and as formal basis for analyzing software architectural specifications [15]. SAM is based on a "dual-formalism consisting of petri nets and temporal logic" [13]. The approach allows hierarchical decomposition of software architectures and verifying constraints automatically using model checking or theorem proving. To analyze an formalized architecture, it is transformed into a finite state system [15]. User input constraints, formalized in *Linear Temporal Logic (LTL)*, are transformed into *Computation Tree Logic (CTL)*. The analysis returns whether the constraint holds for the given system. If not, the failure is demonstrated by an execution sequence. This approach offers a higher analysis variability compared to other related work. Still, the architect has to use the formalism LTL to specify constraints.

Jürjens et al. present the **UMLsec** [18] approach which extends the *Unified Modeling Language (UML)* with "security engineering related techniques" [13]. Through the usage of stereotypes and constraints, UML diagrams can be annotated with security attributes, e.g. by defining data as confidential. To verify modeled systems, a transformation into *Abstract State Machines* is provided. These can be model-checked against an adversary using *SPIN* and the Promela-language [18]. The adversary's behavior is based on the previously annotated diagrams. If a vulnerability has been found, a trail file "which records the sequence of actions of the potential attack" [18] is produced. This file is handed over to the error analyzer, which creates a report containing the problems found. Through this transformation, the architect can model the systems security and view the analysis results on the same abstraction level. However, Jürjens states that there is still lack of "feedback from the model checker back into the UML model" [18] which makes the result interpretation more difficult.

3.3. Data Flow Oriented Modeling

In this section, we summarize related work providing approaches for data flow oriented analysis. This includes *Security Data Flow Diagrams (SecDFD)* [34] and *iFlow* [21, 23]. Please note, that these approaches could also be considered as security modeling (see

Section 3.2). We chose to list them separately because of their strong relation to data flow oriented analyses. Both approaches use annotations which limit the architects analysis capabilities compared to a DSL.

Peldszus et al. present **Secure Data-Flow Compliance Checks between Models and Code based on Automated Mappings** [34]. The mapping is created (semi-) automatically through combining design-level SecDFD-models with corresponding implementations. SecDFD is a graphical representation of data flow diagrams, extended by security annotations. Their goal is to aid the architect by discovering "secure data-flow compliance violations" [34]. Peldszus et al. state the importance of tool-assistance because manual mappings are "inefficient and error-prone" [34]. They propose a semi-automated mapping between model and code based on heuristics like matching structures and signatures. The user has to verify these mappings afterwards in an iterative process. The final mapping can be used for compliance checks, e.g. by applying security metrics. The verification of modeled data flow contracts without the help of external analysis tools is considered future work.

Katkalov et al. propose the **Model-Driven Development of Information Flow-Secure Systems with IFlow** [23]. IFlow provides an environment for modeling of flow-sensitive applications. These models can be be used to "automatically generate deployable app and web service code as well as a formal model" [23]. Based on an annotated UML diagram, restrictions can be formally verified. Therefore, the model is transformed into an *Abstract State Machine* and used as input for theorem proving. Proof goals are generated automatically based on security level annotations by the user, e.g. which data is considered to be kept private. The proving step yields whether the rules of non-inference hold for the specified system and restrictions. Its not discussed if there is any other processing of the analysis results.

3.4. Comparison

To conclude, we compare the different approaches listed before. We focus on aspects which are relevant in the scope of the thesis. This includes the user's influence on the analysis goals and their domain of definition, the analysis method, and the processing of results. The comparison is summarized in Table 3.1.

All approaches have the common goal to aid the architect on design decisions by analyzing architectural models. First, we discuss the domain of analysis goals. Possible domains are the architectural domain if no knowledge is required about the underlying formalism. Except for SAM, all approaches use the architectural domain to formulate analysis goals. The variability of requirements, constraints or analysis goals depends on the provided formalization. Related work uses metrics, logical statements or annotations for modeled software systems. We consider the selection of metrics having a lower analysis variability than annotating models. Using the underlying formalism or a DSL is supposed to have the highest variability.

A common gap is mapping of analysis results. Only *UMLsec* and *Palladio QPN* offer a transformation of analysis output back into the architectural domain. Still, this mapping

Approach	Analysis variability	Goal formulation	Constraint domain	Analysis method	Result mapping
Palladio LQN [24]	low	metrics	architecture	analytical solver	no
Palladio QPN [27]	low	metrics	architecture	simulation	yes
SAM [49]	high	temporal logic	formalism	theorem proving	no
UMLsec [18]	medium	annotations	architecture	model checking	yes
SecDFD [34]	medium	annotations	architecture	theorem proving	no
iFlow [23]	medium	annotations	architecture	theorem proving	no
Data-Centric Palladio	high	logic program	formalism	logic program	no
Aspired goal	high	domain specific language	architecture	logic program	yes

Table 3.1.: Comparison of related work

is limited and does not allow result interpretation in the same view which has been used for modeling.

We added a comparison to the current state of *Data-Centric Palladio* as well as the aspired goal of this thesis. *Data-Centric Palladio* relies on the formulation of constraints using a formalism in addition to annotating the architectural model. Thus, we consider it having high analysis variability. The goal of this work is to enable the architect to define constraints in the architectural domain using a DSL instead of the formalism. Additionally, we use result mapping to transform analysis results back in the architectural domain which eases the interpretation of constraint violations.

4. Example Scenarios

In this chapter, we introduce two example scenarios. These scenarios are used for the requirements analysis, the language design (**RQ1**) and the mapping to the underlying formalism (**RQ2**). Both scenarios originate from other work in the field of design time analysis [23, 47].

4.1. Geolocation Constraints

This example scenario is based on work by Seifermann et al. [41], Kunz [25] and Weimann [47]. It has been previously used to develop parts of the transformation of *Data-Centric Palladio*. We simplify the scenario to clearly show the usage of characteristics for data flow analyses.

The scenario is based on geolocation restrictions implied e.g. from the government. An example are the *General Data Protection Regulations (GDPR)*, which do only allow personal data "to be processed or stored within the European Union or certain countries with equivalent privacy regulations" [25]. Thus, this example represents a data flow restriction based on the kind of data and the processing location. We define the characteristics of this scenario as follows:

- **Privacy level**: Either personal or anonymous. Data is considered to be personal if it can be associated with a person (e.g. name, address, phone number). Otherwise, it is considered to be anonymous (e.g. product lists, stock quantities, item descriptions).
- **Location**: In this example scenario either EU, USA, or Asia. Thus, a place can be inside the EU or not.

As mentioned previously, personal data shall be protected in conformance to legal restrictions: personal data with its origin inside the EU is not allowed to be processed in a non-EU-location. In order to implement this scenario using *Data-Centric Palladio*, we use the data's privacy level (personal or anonymous) and the location of a service (EU, USA, or Asia) as characteristics. We annotate a modeled system with these characteristics.

A common instance of this scenario are (international) online shops which have to synchronize shop information across borders but are not allowed to do so with personal data. In Figure 4.1, we show the diagram of a simple online shop. The Shop Server contains the Web Store component, which is used to view and buy products. If the user views products, the Web Store calls an recommender to get additional information and products to show to the user while browsing. This component is deployed on a separate Recommendation System. When the user buys a product, its personal data is stored in a database, which is deployed on a Database Server.

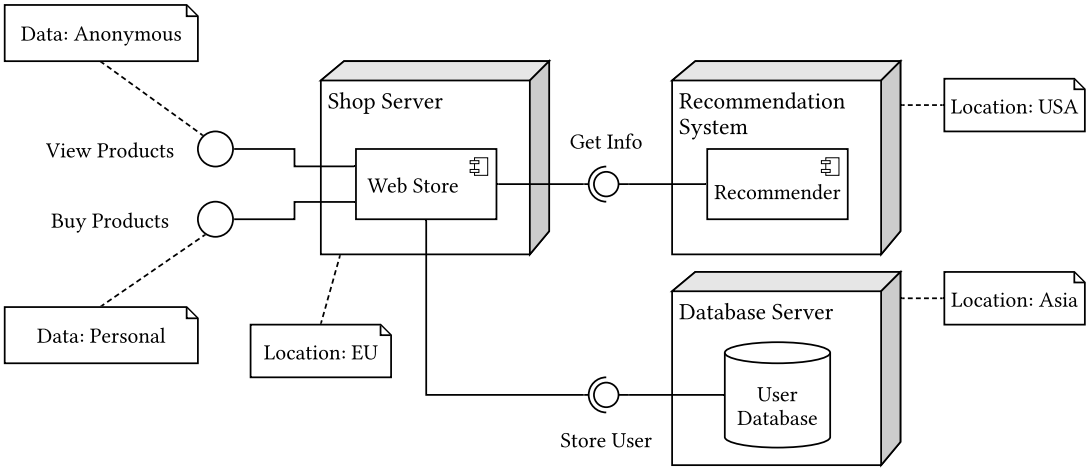

Figure 4.1.: Deployment diagram of the Geolocation Constraints example scenario

We annotate all servers with the *location* characteristic which represents the location of their deployment and we annotate all user endpoints with the *privacy level* characteristic. Trivially, USA and Asia are non-EU-locations. With this information, we can test if the constraint *"No personal information is processed outside of the EU"* holds for every data flow.

In the example shown in Figure 4.1, the data flow from the user to the web store to the recommender fulfills this constraint. Although the recommender is deployed with the Recommendation System in the *USA* (a non-EU-location) only anonymous data is processed which is excluded from the restriction. This is not the case for the buy products data flow. Personal information flows through the web store first which is no problem since it is located inside the *EU*. However, storing the user data in the user database violates the constraint since the Database Server is located in Asia.

Please note, that this example represents a very basic scenario. Still, it is prototypical for real-world problems regarding legal regulations and one can simply imagine that complex, international online shops cannot be checked against data flow constraints by hand without tool-support.

4.2. Access Control

This scenario represents another popular use case of data flow restrictions. Access control systems manage security by assigning access rights to sensitive information. Prior to allowing data access of actors, the system checks their authorization. A well known example of this behavior is the Unix file system [25].

The data flow constraint of access control systems can be formulated as follows: *Data access is only permitted if the actor's role is authorized.* We model this behavior using the actor's roles and the data access rights as characteristics. Every datum has a defined list of authorized roles. A constraint violation occurs if an actor gets access to a datum which does not have the actors role in its authorized role list. We illustrate this in Table 4.1 using

Authorized roles	Actor roles	Intersection of both sets	Constraint violation
$\{A, B, C\}$	$\{C\}$	$\{C\}$	no
$\{A, B\}$	$\{C\}$	$\emptyset$	yes
$\{A, B, C\}$	$\{C, D\}$	$\{C\}$	no
$\{A, B\}$	$\{C, D\}$	$\emptyset$	yes

Table 4.1.: Constraint violation examples in access control systems

Actor	Role
TravelPlanner App	*User*
CreditCardCenter App	*User*
TravelAgency	*TravelAgency*
Airline	*Airline*

Table 4.2.: Actors and roles in the Travel Planner example

four simple roles. In theory, this represents a test for emptiness of the intersection of two sets.

We consider a simplified version of the TravelPlanner case study [23, 22] which has been previously used in other work [41, 25]. In this case study, users install a travel planning application on their smartphone. They have also a CreditCardCenter App installed which manages their credit card details. The TravelPlanner app is "sponsored and developed by a travel agency that acts as a broker for the airline" [22]. Typical usage scenarios are requesting flight offers and booking flights. Since the latter requires the exchange of sensitive information, we model the roles explicitly in Table 4.2 [25]. Possible roles are *User*, *TravelAgency* and *Airline*. Both, the TravelAgency and the Airline have their corresponding roles. The TravelPlanner App and the CreditCardCenter App have the role *User* since both are installed on the user's smartphone.

We show the deployment diagram of the simplified scenario (with according role annotations) in Figure 4.2 and the sequence diagram in Figure 4.3. In the following, we discuss the occurring data flows and the access rights of exchanged data. First, the user requests flight offers using the TravelPlanner App. The app forwards the request to the TravelAgency which forwards it again to the Airline. The list of flight offers is then sent back through the TravelAgency to the TravelPlanner App and displayed to the user. All exchanged data (the flight offer request and the resulting offer) authorize all existing roles (*User*, *Travel Agency* and *Airline*). Trivially, no constraint violation occurs.

The user chooses a flight offer and initializes the booking process through the TravelPlanner App. This time, the request is sent directly to the Airline. Thus, the datum holding the selected flight only authorizes the roles *User* and *Airline*. A constraint violation would occur if information about the selected flight would flow to the TravelAgency. Additionally, the TravelPlanner App transmits credit card data to the airline. Initially, this data only authorizes the role *User* since it is considered sensitive. In order to be passed to the airline,

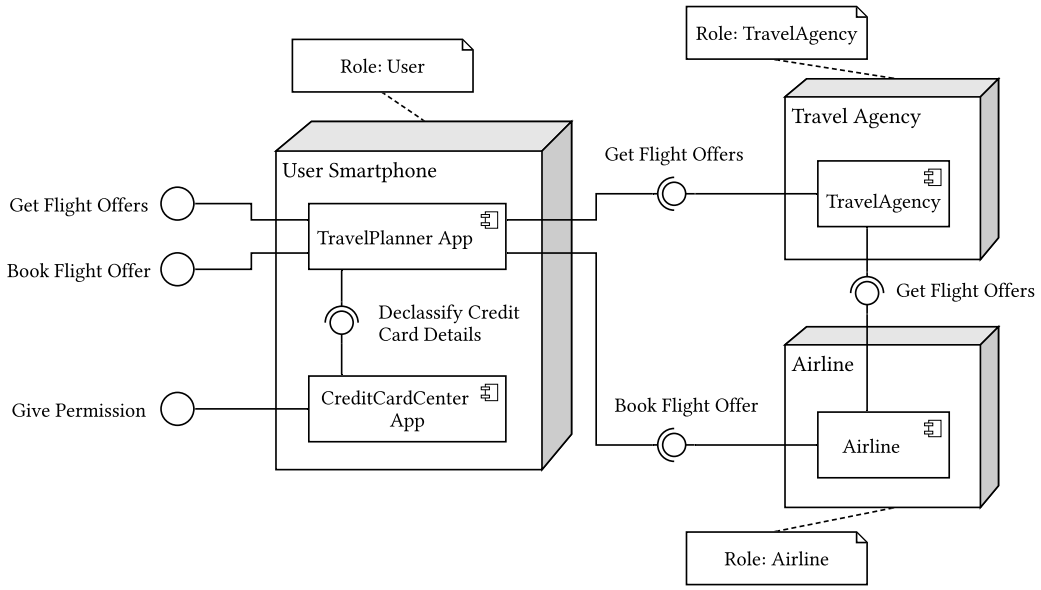

Figure 4.2.: Deployment diagram of the Travel Planner example

it has to be declassified first using the CreditCardCenter App which includes a permission request shown to the user. By accepting this request, the role *Airline* is added to the credit card data. This allows the airline to access selected information about the credit card without a constraint violation.

In comparison with the geolocation constraint scenario shown in Section 4.1, this data flow analysis requires additional effort. The characteristics of data which flows from the CreditCardCenter App are not fixed but rather depend on user input. Additionally, characteristics are represented by sets instead of single values.

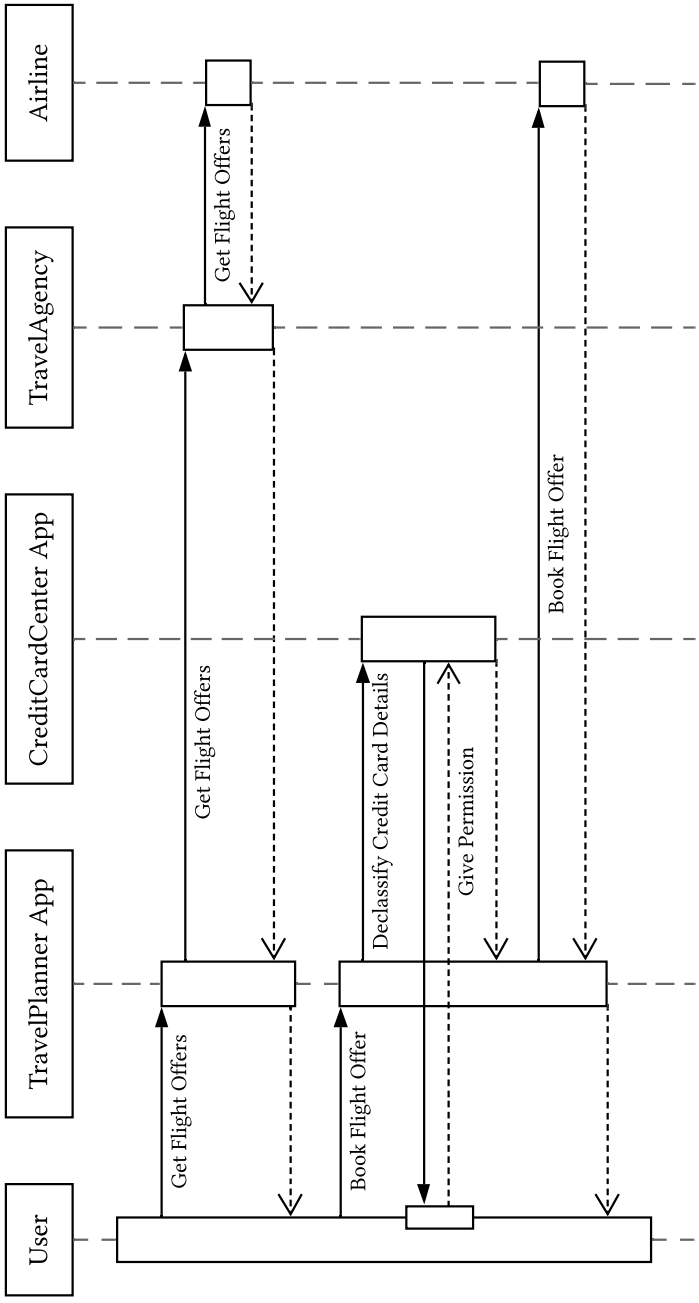

Figure 4.3.: Sequence diagram of the Travel Planner example

5. Supporting the Existing Analysis Process

In this chapter, we discuss the integration of our approach into the existing development process of *Data-Centric Palladio* proposed by Seifermann et al. [41].

As discussed in Chapter 1, the status-quo suffers from multiple problems: There is no abstraction enabling the architect to define analysis goals without knowledge of the underlying formalism and the representation of the transformed architectural model. Analysis results are also affected by this domain gap. We propose the usage of a *domain-specific language (DSL)* to bridge this gap and inspect high-level constructs (**RQ1**) and their mapping to Prolog (**RQ2**).

Figure 5.1 shows the extended development process. White elements represent existing process elements while gray elements are part of our contribution. Elements with dashed borders annotated with gear icons are hereby fully automated without user interference. All other activities are performed by the architect.

The starting point of *Data-Centric Palladio* is a software architecture description based on the *Palladio Component Model* which does "not consider data and its processing explicitly" [41]. In order to enable the analysis of data flows, the architects defines data and its processing explicitly. In order to do that, he defines characteristics and annotates the existing software architecture model. These characteristics represent e.g. the sensitivity of data as shown in Section 4.1. For more information on the usage of *Data-Centric Palladio*, please refer to Section 2.2.

After the data flows under consideration are defined, *Data-Centric Palladio* transforms the model and its annotations to the Operation Model. This intermediate step is proposed "because it decouples the analysis from the ADL [architectural description language] used to model the system" [41]. The Operation Model is a "simplified analysis model" [41] consisting of operations and their processing of data. Afterwards, the generated model is transformed to executable Prolog code. Both steps can be executed automatically.

The definition of analysis goals is independent from this transformation. The architect uses our proposed DSL to define data flow constraints. He defines restrictions based on the characteristics from the previous step. An exemplary constraint from Section 4.1 is the restriction of sensitive user data to not flow to a specified location.

Afterwards, the formulated constraint is transformed to executable Prolog code and combined with the previously generated code from the Operation Model. The result of this step is a Prolog program which is used to find data flow issues by evaluating every defined constraint.

The result of this analysis is a list of violations which refer to the Operation Model. In order to aid the architect, these violations are mapped back into the architectural domain. This enables the architect to interpret the results in the last step without the need to leave

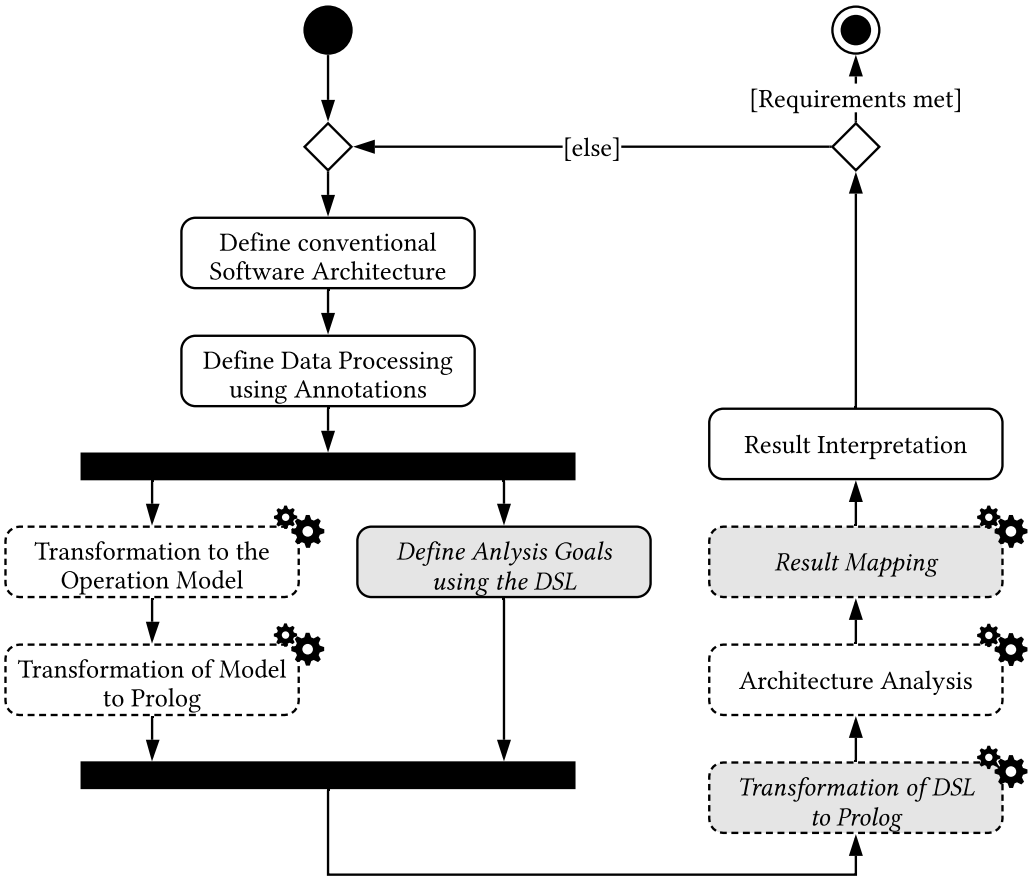

Figure 5.1.: Extended development process with Data-Centric Palladio

his domain. All steps from the transformation of the DSL to Prolog to the Architecture Analysis and the Result Mapping are automated.

Data-Centric Palladio favors an iterative approach. If the modeled architecture does not meet all requirements it can be adjusted based on the analysis results. Previously defined data flow constraints can also be changed or refined after each execution.

6. Language Requirements

In this chapter, we collect requirements for the language design. In order to discuss which constructs and high-level elements are necessary for our DSL (**RQ1**), we consider two influential factors. First, we show case studies and analysis approaches from the security modeling domain (please see Section 3.2) in section Section 6.1. Then, we summarize the existing analysis approach from *Data-Centric Palladio* which uses Prolog and the *Constraint Query API* from Kunz [25] in Section 6.2.

This method is motivated by the *Feature-Oriented Domain Analysis (FODA)* [19] which "supports reuse at the functional and architectural levels" [19]. They identify four major factors which influence a generic domain model: The capabilities from the end-user's perspective, the applications environment, the domain technology and the implementation techniques. This can be summarized as user-centered (or top-down) and technology-centered (or bottom-up). Thus, we both generalize existing case studies (top-down) and abstract from existing constraints formulated in the underlying formalism (bottom-up).

The results from the case studies can be interpreted as lower limit of requirements which the language has to satisfy in order to define applicable data flow constraints. The underlying formalism, namely the *Constraint Query API* is the upper limit of functionality since we cannot define any constraint which is not translatable into it. Trivially, using Prolog without our DSL represents this upper limit but lacks the desired abstraction.

6.1. Case Studies

We collect data flow scenarios from case studies [47, 23] and analysis approaches from other work [15] in the security modeling domain. The studies have already been used in previous *Data-Centric Palladio* related work from Kunz [25]. We extract information about data flows and their case-related restrictions. Then, we show similarities and generalize the constraint requirements. This represents the lower-limit of language requirements.

The **Geolocation Scenario** was used in previous work from Kunz [25] and Weimann [47]. It describes data flow restrictions which originate from the legal framework of international online shops. We describe a simplified version of this scenario in Section 4.1 which only divides the sensitivity of data in "fully sensitive" and "fully anonymous" without finer gradations. The restriction of data flows rather depends on their processing and the environment. We repeat the definition of three confidentiality levels from the work of Weimann [47]:

- *Type-0 (Personal information)*: Data which is directly related to personal information (e.g. name, adress or credit card number).

- *Type-1 (Personal identifiable information)*: Data which does not directly contain personal information. If this data is combined with other *Type-0* or *Type-1* data, personal information can be reconstructed.
- *Type-2 (Anonymous data)*: Data which does not reveal any personal information even combined with other data

In the following, we discuss the constraint formulation. The simplified scenario shown in Section 4.1 focuses on *Type-0* and *Type-2* data. As *Type-2* does not hold any personal information, its data flows are not restricted. *Type-0* data holds sensitive information and thus always needs to be protected. We only allow the processing of *Type-0* data flows in *safe* locations. In the context of a legal framework, this is e.g. the same country the data originates from. In Section 4.1, we describe the constraint as: *"No personal information is processed outside of the EU"*. By using confidentiality levels, this can be generalized to:

"No Type-0 information is allowed to flow outside of a safe location"

Here, the term *flow* implicitly covers processing or storing of data. Please note that we do not define *safe* locations here but rather leave this to the concrete analysis.

The restrictions on *Type-1* need further information about the context of data processing. Here, the joining of two different data flows from from different sources has to be prohibited because this would enable the retrieval of personal identifiable information. Kunz models the source of data as data type for "illustration purposes" [25]. However, he points out that an alternative would be use of "an additional attribute to the data which specifies the information source" [25]. In order to prevent the joining of data flows, the constraint can be formulated as follows:

"No Type-1 data is allowed to flow to a location where other Type-1 data is processed if both have different data types"

This first scenario reveals multiple requirements for formulating data flow constraints and thus language elements. We collect our findings in the following:

R1 Constraints restrict the flow of data between source, processes and sinks (please see Section 2.1 for details on the terminology). They abstract from concrete processing operations, parameters and control flow

R2 Data can be selected by its characteristics (e.g. *Type-0* data)

R3 Data can be selected by its data type

R4 Groups of processes and sinks can be selected by their characteristics (e.g. the location)

R5 Processes and sinks can be selected by their identity (e.g. restricting multiple data flows from being handled in one process)

R6 Characteristic selection can be inverted (e.g. every location except the `EU`)

R7 Characteristic selection accepts multiple values (e.g. Asia and USA)

R8 Multiple forms of a characteristic can be combined to named classes (e.g. *safe* locations)

R9 Multiple data flows can be considered in one constraint (e.g. two distinct *Type-1* data flows)

The **TravelPlanner** case study originates from the Iflow project [23]. It represents an access control system. This system only permits an actor to gain access if the requested datum is open for the actor's role. A simple example are system critical files of an operating system: A system administrator is allowed to access these files while normal users are not. The actor's roles and the authorized roles of a datum can be both modeled using characteristics.

The travel planner case study presents a typical scenario of planing a travel by plane. We explained this case study in detail in Section 4.2. The involved actors are the user, a travel agency and an airline. The user books a flight offer and transmits credit card details which is considered to be sensitive. Thus, only the airline is permitted to gain access after the user explicitly declassified the information. This can be explicitly modeled by controlling the roles which are allowed to gain access on the credit card data. The list of roles is not static but changes over time, e.g. when the data is declassified and sent to the airline. Other information such as data about flight offers is not restricted at all and can be viewed by all actors.

This case study represents only one case of the usage of access control systems. In order to define a generalized constraint, we do not consider example data (e.g. credit card data) but rather restrict the data access using the characteristics *actor's roles* and *authorized roles*. Both characteristics are modeled using sets rather than single values. If a datum's list of *authorized roles* does not contain at least one of the accessing *actor's roles*, a constraint violation occurs. There is no differentiation if an actor actively asks for the datum (in the sense of a call in the control flow) or if the datum passifly flows to the actor like in the example presented above. We define the constraint as follows:

"Information is only allowed to flow to an actor if the intersection of its authorized roles and the actor's roles is not empty"

Intuitively, a non-empty intersection of both sets implicates that the *authorized roles* contain at least one of the *actor's roles*. We prefer the set opertation since it can be used in a broader context.

The analysis of this case study reveals additional requirements. We extend our list as follows:

R10 Characteristics can be variable (e.g. an *authorized role*) and thus require further restrictions

R11 Variable Characteristics can be represented by single values or sets (e.g. a single or multiple roles)

R12 Variable characteristic can be analyzed using set operations (e.g. the intersection of two sets or a test for emptiness of a set)

Formal **Software Architecture Models (SAM)** are related work to this thesis. Yu et al. [49] present another exemplary travel planner to explain their use of formalisms for defining requirement specifications. Please see Section 3.2 for more information.

The travel planner receives travel information and then books an appropriate flight, reserves a hotel room and rents a car. The flight ticket is bought from Delta Airlines if their offer is lower than $400, otherwise the ticket is purchased at Continental Airlines. This internal decision is the critical point of the system: If the agent which returns ticket prices for Delta Airlines gains knowledge about the $400 threshold it could always return $399. This would ensure that they always sell their own ticket while still maximing their profit per ticket. To avoid this problem, they define the following constraint (simplified):

"An agent's decisive data is not allowed to be leaked to another agent"

Here, the decisive data can be an arbitrary combination of input data, e.g. the price, user information, selected route or travel time. In the example above, the ticket price represents decisive data. The sensitivity of data has to be labeled for every agent.

This constraint introduces one additional requirement:

R13 Data and processes can be selected by combining the selection of multiple characteristics

In the following, we summarize the findings from this section. Our goal is to define a lower limit of functional requirements for the language design. Besides defining language elements, we seek to find similarities and a common constraint shape. We start by enumerating the previously determined data flow constraints:

(1) *No Type-0 information is allowed to flow outside of a safe location*

(2) *No Type-1 data is allowed to flow to a location where other Type-1 data is processed if both have different data types*

(3) *Information is only allowed to flow to an actor if the intersection of its authorized roles and the actor's roles is not empty*

(4) *An agent's decisive data is not allowed to be leaked to another agent*

Despite the fact that these constraints originate from different case studies and related work they show several similarities. First, they restrict the flow of data by *not allowing* it or *only allowing* it under certain circumstances (**R1**). The modality of these restrictions is fixed, *never* allowing a data flow or leakage.

Second, these restrictions only apply for selected cases and not for all data flows. Examples are **(1)** only considering *Type-0* information or **(4)** only restricting decisive data. There is no information on data which does not fit the selection. In the geolocation

scenario, *Type-2* data is allowed to flow to any *safe* or *unsafe* location. The data flow constraints follow a blacklisting approach.

Data is selected based on its characteristics (**R2**) or other attributes like its data type (**R3**). The selection can also be inverted (**R6**) or combined with other characteristics (**R13**). Multiple values (**R7**) and classes of characteristics (**R8**) are also possible, e.g. all *safe* locations. In some cases, the static value of a characteristic is not relevant but rather its value in comparison with other data characteristics values, e.g. in **(3)** which requires more detailed restrictions (**R10** and **R11**).

Besides the selection of data based on its characteristics, nodes like processes and sinks are selected. These are sometimes also called entities, actors, agents or locations. Nodes are selected by their characteristics (**R4**) or by their identity (**R5**).

The restriction of data flows in **(2)** and **(3)** is coupled on an additional condition. This condition uses previously defined characteristic variables and evaluates them, e.g. by testing for distinct sets in **(3)**. Other variable comparisons and set operations are imaginable (**R12**). All constraints consider one data flow at a time except for **(2)** which restricts the joining of distinct data flowing from multiple sources (**R9**).

6.2. Underlying Formalism

In order to discuss the upper limit of of functionality, we summarize the *Operation Model* and the *Constraint Query API* [25] which are used by *Data-Centric Palladio* [41]. Because we extend this approach (please see Chapter 5 for details) the analysis capabilities of this underlying formalism represent the upper limit of possible constraints. Last, we discuss abstraction possibilities and gaps between the architectural domain and the analysis domain.

The *Operation Model* is used as intermediate model between the architectural description of a system and lower-level Prolog code. It can be seen as "simplified analysis model" [41]. The transformations between the modeled architecture, the *Operation Model* and Prolog code are fully automated. Because our proposed DSL is also transformed into Prolog code which refers to this model, it's viable to consider it early in the design process.

In the following, we show important elements form the *Operation Model*:

- *Operations* are central structural elements and represent processes which manipulate data, e.g. a database store operation
- *Call Parameters* and *Return values* are used to exchange data between operations, sources and sinks, e.g. the data to store in a database
- *State Variables* represent globally accessible data which can be also used by operations
- *Properties* represent characteristics of an operation, source or sinks. Properties are static and set in the transformation, e.g. the location of a database
- *Attributes* represent characteristics of data flowing between operations. Attributes are dynamic and can be manipulated from operations, e.g. the sensitivity of data

- *ValueSetTypes* are containers for characteristics values or literals. Both properties and attributes refer to these types, e.g. the characteristics location and sensitivity

The *Operation model* does not work with single data instances but rather classes of data with matching characteristics. *Data-Centric Palladio* doesn't specify the flow of data instances either; the use of characteristics to select classes of data is in focus. Characteristics are "named finite sets of values" [41] represented by *ValueSetTypes* and used in *properties* or *attributes*. This distinction is due to semantic differences between constant operation characteristics and variable data characteristics [25]. Every entry of an *ValueSetType* is represented as boolean value. E.g. setting a data's privacy characteristic to private results in the data types attribute privacy.private set to true.

Using the *Constraint Query API* "the system can then be queried for constraint violations" [25]. Prior to the development of our DSL, this was the proposed way of interacting with the *Operation Model*. We summarize the predicates which are provided by the API. For a full reference, please refer to [25, p. 41].

Listing 6.1 shows basic predicates of the API which can be used to bind typing information. This can be seen as first step to more complex queries "for which one would normally use the universal or the existential quantors" [25]. Line 1 to 3 shows predicates for the names of atomic building elements, namely *properties*, *attributes* and *operations*. Using the predicates from line 5 to 7, we can combine this information to ask if a named value, *attribute* or *property* belongs to a *ValueSetType*. With the predicates from line 9 to 11, the relation of *operations* with *parameters*, *return values* and *state variables* can be queried.

```
1   isProperty(P)    % true if P is a property
2   isAttribute(A)   % true if A is a attribute
3   isOperation(OP)  % true if OP is an operation
4
5   valueSetMember(T,V) % true if V is a value which belongs to the ValueSetType T
6   attributeType(A,T)  % true if A is an attribute and T is its ValueSetType
7   propertyType(P,T)   % true if P is a property and T is its ValueSetType
8
9   opertationParameter(OP,P)   % true if OP is an operation with a parameter P
10  opertationReturnValue(OP,R) % true if OP is an operation with a return value R
11  opertationState(OP,ST)   % true if OP is an operation with a state variable ST
```

Listing 6.1: Basic predicates of the Constraint Query API

Based on this information, we can use more advanced predicates which query the flow of data and their characteristics. We summarize these predicates in Listing 6.2. The *Operation Model* uses a stack to represent the control flow through the system. Every *operation* and each call is put onto this stack. The most upper element is always the current *operation* under consideration, the most lower element of a correct call stack is the *SystemUsage* which can be seen as entry point to any valid call sequence. The predicate *stackValid* checks if an arbitrary list represents such a correct call stack.

The predicates in line 3 and 4 are used to determine if an *operation* has a *property* (or a *property* with a given value). As stated before, *properties* represent characteristics

of processes in the data flow. E.g. hasProperty(OP,location) evaluates to true for every operation if the location would be modeled for all operations.

```
1   stackValid(S) % true if the list S represents a correct call sequence
2
3   hasProperty(OP,PR)         % true if the operation OP has a property PR
4   operationProperty(OP,PR,V) % true if the operation OP has a property PR
5                                which is set to value V
6
7   callArgument(S,P,A,V) % true if the operation on the top of stack S has a
8                           parameter P with value V of the attribute A
9   returnValue(S,R,A,V)  % true if the operation on the top of stack S has a
10                          return value R with value V of the attribute A
11
12  preCallState(S,OP,ST,A,V)  % true if the operation OP with its call on top of
13                               stack S has a state variable ST with value V of
14                               attribute A before its execution
15  postCallState(S,OP,ST,A,V) % true if the operation OP with its call on top of
16                               stack S has a state variable ST with value V of
17                               attribute A after its execution
```

Listing 6.2: Advanced predicates of the Constraint Query API

Characteristics of data flowing through the system in the form of *parameters* or *return values* are modeled using *attributes*. The predicates from line 7 and 9 are used to check if the *operation* on the top of the call stack has *call parameters* or *return values* with their *attributes* set to the requested value. Is e.g. the sensitivity characteristic of data modeled explicitly, every ingoing *parameter* and outgoing *return value* of any operation in the call stack can be tested for sensitivity.

This is also the case for state variables. Since state variables have to be unambiguously identified, the containing operation which uses these variables must also be specified in order to use the predicates from line 12 and 15. The choice of predicate (`pre` or `post`) determines if the variable is considered before or after the operation call.

In order to discuss both lower and upper limits, we combine the findings from this section with the discussion on higher-level language requirements from the previous section. The comparison of the interaction with the *Constraint Query API* and the way we formulated constraints in Section 6.1 shows multiple domain gaps which have to be resolved.

First, a gap in the representation of model elements exists. All constraints refer to architectural model elements (e.g. components) or higher-level entities (e.g. agents or actors). In the context of data flow modeling these are represented by sources, processes or sinks (please see Section 2.1). These elements are selected by characteristics (**R4**) or their identity (**R5**). Fortunately, the existing transformation of *Data-Centric Palladio* transforms these elements from the architectural domain to the *Operation Model* while retaining their characteristics (or *properties* in the terminology of the *Operation Model*).

This is also the case when speaking about data. All constraints select data, e.g. by their characteristics (**R2**) or meta-attributes (**R3**). Also multiple characteristics (**R13**) or characteristics with multiple accepted values (**R7**) are possible. The transformation to the *Operation Model* transforms characteristics to *ValueSetTypes*. Using predicates like `callArgument` and `returnValue`, this information can be querried. Combinations of multiple characteristics or values are also possible using Prologs built-in predicates to combine multiple terms.

Some constraints require additional conditions (**R10**, **R11** and **R12**), e.g. comparing two sets of roles for access control systems. The realization of these constraints using the *Constraint Query API* requires additional steps, e.g. introducing new variables and evaluating them separately. This also can be achieved using Prolog.

Besides selecting data and processes, all constraints show a similar structure. They restrict the flow of data to e.g. never allowing it under certain circumstances (**R1**). Constraints following this approach can be expressed by using the predicates from the *Constraint Query API* shown in this chapter and bridging the domain gaps discussed before. We select data (*parameters*, *return values* and *state variables*) and *operations* from the call stack. The execution of the formulated query then returns violations which match our selection.

7. Language Syntax

In this chapter, we present the syntax of our *domain-specific language (DSL)*. **RQ1** asks which elements are necessary to define data flow constraints. We discuss architecture-level constructs and language elements which are used to satisfy the requirements discussed in Chapter 6. We start by giving a brief conceptual overview of the DSL in Section 7.1. In Section 7.2 to Section 7.4, we discuss the language elements in detail and show their relation to the previously collected requirements. Last, we formulate constraints for the example scenarios shown in Chapter 4 and explain the usage of the DSL in Section 7.6.

We focus on the abstract syntax of our DSL which is represented by the language's meta-model. Our contribution to **RQ1** is the analysis of domain elements rather than their concrete representation. Furthermore, an abstract syntax can be represented by multiple concrete syntaxes. We show our concrete syntax at the end of each section but refer to it as *exemplary* concrete syntax.

7.1. Overview

The purpose of our DSL is to define constraints which restrict the data flow. In order to aid the architect to reason about these data flows, we only use concepts from the architectural domain. In Section 6.1, we discuss that all analyzed constraints follow a common pattern. They restrict the flow of selected data without considering the control flow **(R1)**. They use the modality of *never* allowing a data flow. Furthermore, conditions can be used to precisely specify forbidden behavior.

In Figure 7.1, we show the simplified abstract syntax of constraints in our DSL. We annotate the elements with notes to show how the syntax parts relate to the exemplary constraint from Section 4.1: *"No personal information is processed outside of the EU"*. Please note, that this is only a simplification. For the complete meta-model, refer to Section A.4.

A Constraint is a named entity defined by the architect. Each Constraint contains a Rule which defines the desired restriction on data flows. The statement sets the modality of of the rule. As discussed before, all constraints gathered in Section 6.1 require a data flow to *never flow* under certain circumstances.

A constraint only subjects selected data and selected destinations. The selection step is required because restricting all data flows without limitation is not considered to return viable results. Thus, DataSelector and DestinationSelector are obligatory elements of each Rule. We include multiple ways to select data and destinations. First, they can be selected using characteristics which have been introduced by *Data-Centric Palladio*. Our DSL is integrated in the analysis process and references characteristics on previously defined architectural models (for more information on the complete process, see Chapter 5). A characteristic selection can be used both as DataSelector and DestinationSelector. In the

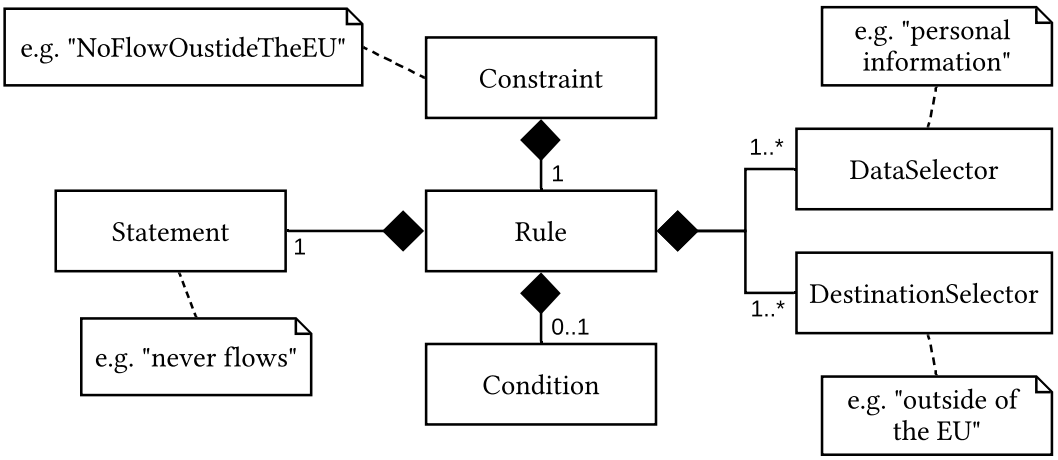

Figure 7.1.: Simplified abstract syntax of constraints in our DSL

case of the *Geolocation Constraints* scenario, we select data which contains *"personal information"* (using a characteristic) and select destinations *"outside of the EU"* (using another characteristic).

Besides specifying single characteristic literals, also the selection of multiple literals and the exclusion of literals are possible. Another way of using characteristics is to define characteristic classes which encapsulate the characteristic selection. By using classes, selections can be predefined by an architect and referenced in multiple constraints. This indirection is considered useful in more advanced scenarios. Last, DestinationSelectors can refer to elements of the architectural model created with *Data-Centric Palladio*. Using this identity selection, restrictions can be defined more precise and tailored to specific operations or components.

Some constraints do not rely on fixed characteristics (or sets of characteristics) but rather compare the values of data and destination characteristics. An example are role-based approaches where processes emit data with dynamically set authorization roles. To formulate such constraints, we add characteristic variables and set variables. These variables are defined using DataSelectors and DestinationSelectors. We add Conditions which restrict possible variable values with regard to other variables. The simplest case is a value comparison of characteristic variables of data flowing to a destination with other characteristic variables. More complex scenarios which compare characteristic sets using set operations or the conjunction of multiple conditions are also supported.

The language elements Statement, DataSelector and DestinationSelector are obligatory. The Condition of a rule is optional and only required if a DataSelector or DestinationSelector defines a characteristic variable.

Regarding the concrete syntax, we aim to provide a modular design which can be easily read by an architect without prior knowledge. Because most of the language elements shown above are non-optional, every Rule has the same structure:

⟨DataSelector⟩ ⟨Statement⟩ ⟨DestinationSelector⟩ ⟨Condition⟩

This structure is motivated by the constraints formulated in natural language: "*Data with certain characteristics never flows to destinations with certain characteristics where...*". Listing 7.1 shows a possible constraint for the *Geolocation Constraints* example which prohibits personal data to flow outside of the EU using our DSL. We describe each compartment in the following sections.

```
1  constraint NoFlowOutsideTheEU {
2    data.attribute.privacy.personal NEVER FLOWS node.property.location.!EU
3  }
```

Listing 7.1: Simplified Geolocation Constraints example given in concrete syntax

7.2. Types and Imports

The development process with our DSL is integrated in *Data-Centric Palladio*. We reference characteristics and architectural model elements. Prior to using these to define data flow constraints, they have to be properly included from the environment explicitly.

We identify the following external information to be relevant:

- *Characteristic Types*: *Data-Centric Palladio* defines containers which encapsulate the type's name and possible values called literals. They are represented as enumeration. E.g. for a characteristic called Privacy level which models the privacy of personal data, possible literals are personal and anonymous.

- *Palladio Models*: In order to refer to specific nodes or destinations of data flows, we need information about the architectural model and its deployment. The latter is stored in the *Allocation Model* which references the *Execution Environment Model*, the *System Model* and transitively the *Component Repository Model*. The *UsageModel* describes the users behavior and provides insights on entry points of control- and data flow. Combined, these models represent modeled information of the architecture for which we define data flow constraints. For more information on the *Palladio Component Model*, see Section 2.2.

In the following, we describe how this information is included and processed in order to be used to define data flow constraints. Figure 7.2 shows the abstract syntax of the root container Model and contained language elements such as constraints discussed before. The Include element is used to import the information specified above from other models. These models are serialized and store in XMI-files. Thus, the Include-statement references these. If constraints only rely on characteristics and don't relate to specific architectural elements, only including *Characteristics Types* is sufficient. Otherwise, all relevant Palladio models have to be also included as stated above.

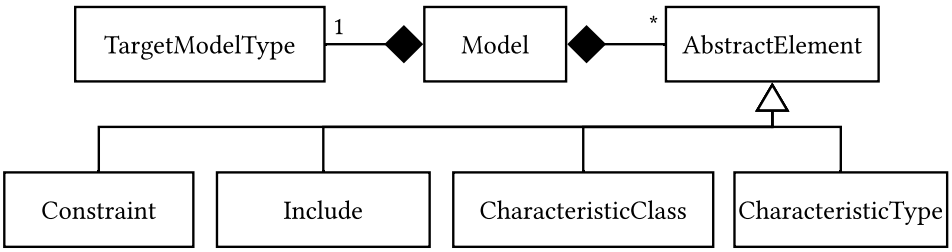

Figure 7.2.: Abstract syntax of top-level elements

Every Model requires a definition of one TargetModelType. We use this explicit definition to separate the constraint formulation from the architectural description which is represented by the TargetModelType. Possible types are *Data-Centric Palladio* and the underlying *Operation Model*. We support both model types since the choice only has minor impact on the DSL complexity while giving the architect more flexibility. Besides the target type, architects state which imported *Characteristic Types* and *Palladio Models* shall be used to resolve references.

Characteristics have to be explicitly declared before they can be used inside constraints. Using the CharacteristicType language element, the architect binds types to included *Characteristics Types*. This indirection of type definition has multiple advantages: First, *Characteristic Types* can be renamed for the usage inside of constraints which can enhance the understandability of the formulated restriction. Second, not all included *Characteristic Types* are sometimes needed, e.g. if only a subset of these are required in the defined constraints. Third, this indirection is considered to be more generic regarding the desired target model type. Using separate characteristic type definitions, multiple target models can be supported without the need of rewriting constraints.

Other language elements which are directly contained in the root Model are Constraints and Characteristic Classes. As outlined in Section 7.1, Characteristic Classes encapsulate one or more characteristic selections. They can be referenced throughout all constraints in the same file and are thus defined independently. We discuss their syntax in Section 7.3. Constraints are also directly contained in the Model. They are the main language element to formulate data flow restrictions and are explained in Section 7.5.

Last, we show our *exemplary* concrete syntax in Listing 7.2. We use the `import`-keyword to include external information like *Characteristic Types* and *Palladio Models* in line 3 to 5. Defining import-statements in the head of a code file is common practice in many programming languages. We start each file with the definition of the TargetModelType, as shown in line 1. We use the keyword `target` followed by the model type (`DataCentricPalladio` or `OperationModel`). Additionally, included *Characteristic Types* and *Palladio Models* which shall be used to resolve references prior to the analysis are added to a comma-separated list. We reference them by their file name without file name extension. The list starts after the `using`-keyword. At least one source with *Characteristic Type* definitions is required. We use the `type`-keyword to indicate a declaration of CharacteristicTypes. A type declaration

contains the desired type name for the usage inside Constraints and Characteristic Classes followed by the name of the included characteristic. An example is shown in line 7.

```
1  target DataCentricPalladio using characteristicTypes, allocationModel, usageModel
2
3  import "characteristicTypes.xmi"
4  import "allocationModel.allocation"
5  import "usageModel.usagemodel"
6
7  type internalTypeName : OriginalTypeName
```

Listing 7.2: Concrete syntax of type and import statements

7.3. Selection

With our DSL, we formulate restrictions to data flows in software systems. These restrictions are always applicable to only a subset of all possible data flows. Disallowing all data flows would not yield viable results because any function call would be interpreted as constraint violation. Thus, the selection of data and data flow destinations is crucial. Using *Data-Centric Palladio*, the proposed way of selection is by characteristics.

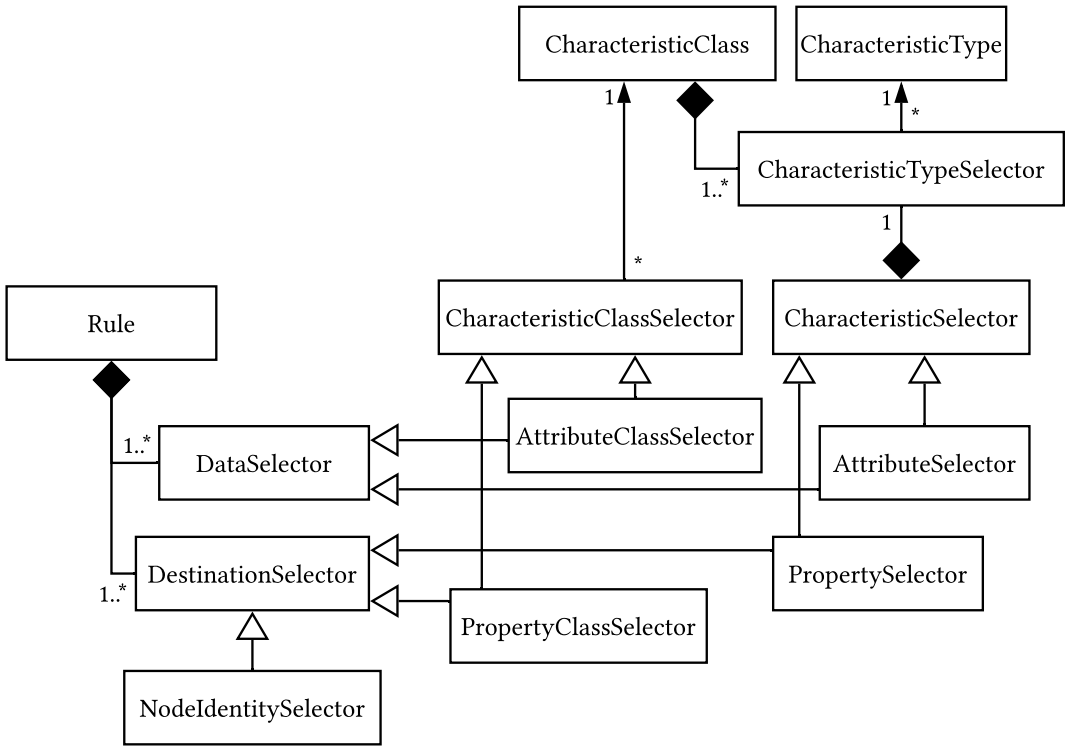

Figure 7.3.: Abstract syntax of data and destination selection as well as classes

In Figure 7.3, we show the abstract syntax of data- and destination selections, called *Selectors*. The majority of these *Selectors* work with characteristics and thus with the CharacteristicType element, discussed in Section 7.2. The CharacteristicTypeSelector is used to select literals from a referenced CharacteristicType. In the following, we enumerate all possible selections using the CharacteristicTypeSelector. For each selection, we give an example with the characteristic type Privacy level which models the privacy of personal data. Possible literals are personal and anonymous.

- *Single literals*: The most basic case of characteristic selection is to select one single literal (**R2**, **R4**). This is the case if e.g. only one type of data is considered in a constraint, e.g. only personal data.
- *Multiple literals*: This characteristic selection is used if more than one literal are under consideration (**R7**). There is no maximum number of selected literals. Thus, this selection is only limited by the number of defined literals in the characteristic type, e.g. personal and anonymous.
- *Inversion*: One or multiple literals can be selected by selecting any literals except one (**R6**). This is considered to be a inversion, e.g. selecting every literals which is not anonymous (which results in the selection of the personal literal in our example).
- *Variables*: Variables are used to evalaute multiple selections, e.g. by comparing them (**R10**). Variables do not restrict the selection to a defined subset of possible literals but work as a placeholder. They require the architect to make further assumptions in the Condition part of a rule. We discuss this case in Section 7.4.

This selection is embedded in the CharacteristicSelector. We use the selection of characteristic literals in two parts of an constraint's Rule. We select data using the DataSelector and we select the data flow destination using the DestinationSelector. In Figure 7.3, we use multiple inheritance to indicate the combined meaning of *Selectors* being part of a Rule and their specific selection goal. AttributeSelectors represent both CharacteristicSelectors and DataSelectors which are used to define which data is restricted using characteristics. Accordingly, PropertySelectors represent CharacteristicSelectors and DestinationSelectors, determining the restricted data flows destination using characteristics. We use the terminology of the *Operation Model* which differentiates between *properties* (characteristics of operations) and *attributes* (characteristics of data). For more information, see Section 6.2.

Besides selecting characteristic literals directly inside a Rule, CharacteristicClasses can be defined (**R8**). These classes are not part of a Rule and thus can be referenced in any constraint. The use of CharacteristicClasses offers multiple benefits: First, it's possible to share characteristic selections among multiple constraints. This does not only speed up development by reusing selections but also reduces the change impact to a single source rather than changing all affected constraints. Second, multiple selections can be encapsulated into a named class. This enables the architect to name a specific combination of selections and thus enhances the understandability.

CharacteristicClasses contain one or multiple CharacteristicTypeSelectors. Thus, the selection of characteristics is independent from the later usage in DataSelectors or DestinationSelectors. We define two concrete *Selectors* following the same pattern discussed

above. AttributeClassSelectors reference CharacteristicClasses and are used as DataSelector. Accordingly, PropertyClassSelectors reference CharacteristicClasses and are used as DestinationSelector.

Last, we discuss NodeIdentitySelectors. These represent the only *Selectors* which do not depend on characteristics. Instead, they reference architectural elements like components directly (**R5**). There are two possibilities to identify destinations. If the constraint is defined for the TargetModelType *OperationModel* (see Section 7.2), the name of the operation is sufficient. Otherwise, if the TargetModelType is set to *Data-Centric Palladio*, the destination is identified using elements from *Palladio Models* which have to be included in advance. For a fully qualified reference, the architect needs to specify the *Assembly Context* (from an included *System Model* or *Repository Model*), a *Component* from this context and a contained *Service Effect Specification (SEFF)*.

We present our *exemplary* concrete syntax of *Selectors* and CharacteristicClasses in Listing 7.3. Please note, that line 5 to 14 do not represent complete constraints but are intended to show multiple usage scenarios of *Selectors*. For this example, we use the already introduced characteristic Privacy with its literals personal and anonymous. Additionally, we use the characteristic Location with its literals EU, USA and Asia. Both characteristics have been introduced in Section 4.1.

```
1   class NonPersonalData {
2     privacy.!personal
3   }
4
5   data.attribute.privacy.personal
6   data.attribute.privacy.!personal
7   data.class.NonPersonalData
8
9   node.property.location.EU
10  node.property.location.!EU
11  node.property.location.[USA, Asia]
12
13  node.identity.name."OperationName"
14  node.identity.ExemplaryAssemblyContext.ExemplaryComponent.ExemplarySEFF
```

Listing 7.3: Concrete syntax of selectors and characteristic classes

Line 1 to 3 show the definition of a CharacteristicClass. Using this class, all literals except personal are selected in line 2. In our example, thats only true for the literal anonymous. The concrete syntax of such CharacteristicTypeSelectors starts with the characteristic name (`privacy`) followed by the literal selection (*not* `personal`). The class name `NonPersonalData` reflects this selection. Line 5 and 6 show a characteristic selection used as DataSelector. Therefore, they are introduced by the keyword `data`. The keyword attribute indicates the CharacteristicSelector type. Line 5 selects only data with the characteristic `privacy` set to `personal`. Line 6 indicates the opposite. Line 7 shows the data selection using a AttributeClassSelector and referencing the CharacteristicClass defined in line 1 to 3. This is indicated by the `class`-keyword. Because the referenced characteristic class selects

all literals which are *not* `personal`, using this class as DataSelector is equivalent to the selection in line 6.

Line 9 to 11 show PropertySelectors which are both CharacteristicSelector and DestinationSelectors. This is indicated by the keyword `node` and `property`, respectively. All three PropertySelectors reference the characteristic `location`. The first statement in line 9 selects all destinations with the characteristic `location` set tot `EU`. The second statement in line 10 selects all destinations with the characteristic `location` set *not* to `EU`. The third statement in line 11 selects all destinations with characteristic `location` set to either `USA` or `Asia`. In order to simplify the syntax, the inversion of literals and the selection of multiple literals cannot be combined. The conjunction of multiple (inverted) selections can be achieved by utilizing multiple DataSelectors. Because in our example, the three possible literals are `USA`, `Asia` and `EU`, the selections from line 10 (*not* `EU`) and line 11 (all location literals except `EU`) are equivalent. Please note, that this does not indicate a semantic equivalence. If another characteristic was added afterwards, the selection of both lines would become different again.

Line 13 and 14 show NodeIdentitySelectors as indicated by the keyword `identity`. If the TargetModelType is set to *Operation Model*, destinations are selected bei their name using an arbitrary string. In order to identity destinations using *Data-Centric Palladio*, the architect specifies the *Assembly Context* (e.g. `ExemplaryAssemblyContext`), the *Component* (e.g. `ExemplaryComponent`) and the *Service Effect Specification* (e.g. `ExemplarySEFF`).

7.4. Conditions

For some data flow restrictions such as the role-based access control scenario presented in Section 4.2, defining rules with fixed values is not sufficient. In order to extend the architects' capabilities, we introduce characteristic variables and characteristic set variables in Section 7.3. These variables represent placeholders for possible literals in the analysis process (**R10**). The initial usage of variables does not restrict the data flow in any way. Thus, architects need to restrict possible values of multiple characteristic variables by evaluating them (**R12**). This is achieved using the Condition compartment of a Rule.

Conditions are defined to work with variables and sets. In order to make restrictions on their allowed values, we define operations. These operations reference variables and combine them or evaluate them to boolean statements. Simple examples are testing if two variables are equal or checking if a variable's value is element of another set variable. It's also possible to create combinations of variables, e.g. by unifying two set variables' literals. Last, operations can be nested which enables architects to formulate more complex Conditions.

Figure 7.4 shows the abstract syntax of Conditions. In order to enhance the readability, we omit concrete Operations and show these separated in Figure 7.6. As discussed in Section 7.3, CharacteristicTypeSelectors refer to defined CharacteristicTypes. Instead of binding a selection to specific literals, architects can postpone the restriction to the Condition-part of a Rule by using a CharacteristicVariableType. They either define a new CharacteristicVariable or a new CharacteristicSet variable. Both are required to have an unique name. CharacteristicVariables represent placeholders for exactly one characteristic literal, CharacteristicSet variables

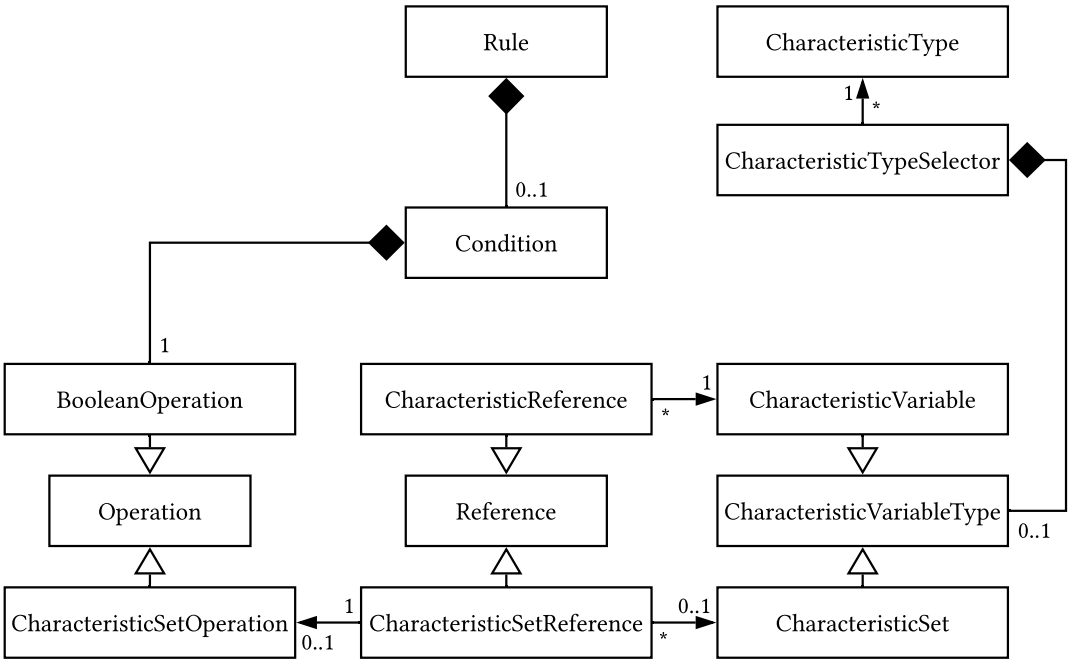

Figure 7.4.: Abstract syntax of conditions

represent placeholders for a set of characteristic literals. Thus, multiple literals, one literal or none (empty set) are possible instances.

In order to reference these variables in Operations, we use References which point on CharacteristicVariableTypes defined in CharacteristicTypeSelectors. We use CharacteristicReferences to reference CharacteristicVariables and we use CharacteristicSetReferences to reference CharacteristicSets. These references are used from concrete operations e.g. a comparison operation which checks two distinct variables for equality using two CharacteristicReferences. In Figure 7.4, we omit all containment relations of References because these belong to concrete Operations which inherit from either BooleanOperation or CharacteristicSetOperation.

Operations follow the functional paradigm strictly. They accept one or more arguments (represented by References) and return a single value without any side effects. We classify Operations depending on their *return type*. CharacteristicSetOperations return a CharacteristicsSet. Thus, CharacteristicSetReferences can refer to these operations (nesting) as well as to concrete CharacteristicSet variables. BooleanOperations return a boolean value, either *true* or *false*. They can be nested in other BooleanOperations, e.g. the conjunction of two values. We don't define a separate CharacteristicOperation because we don't realize any operation which has a CharacteristicVariable *return type*. A Condition is represented by exactly one BooleanOperation. Which concrete operation is used and wether or not multiple operations are nested is up to the architect.

We show three exemplary operations and their usage of Refernces in Figure 7.5. The three concrete operations are highlighted gray. The CreateSetOperation is used to convert

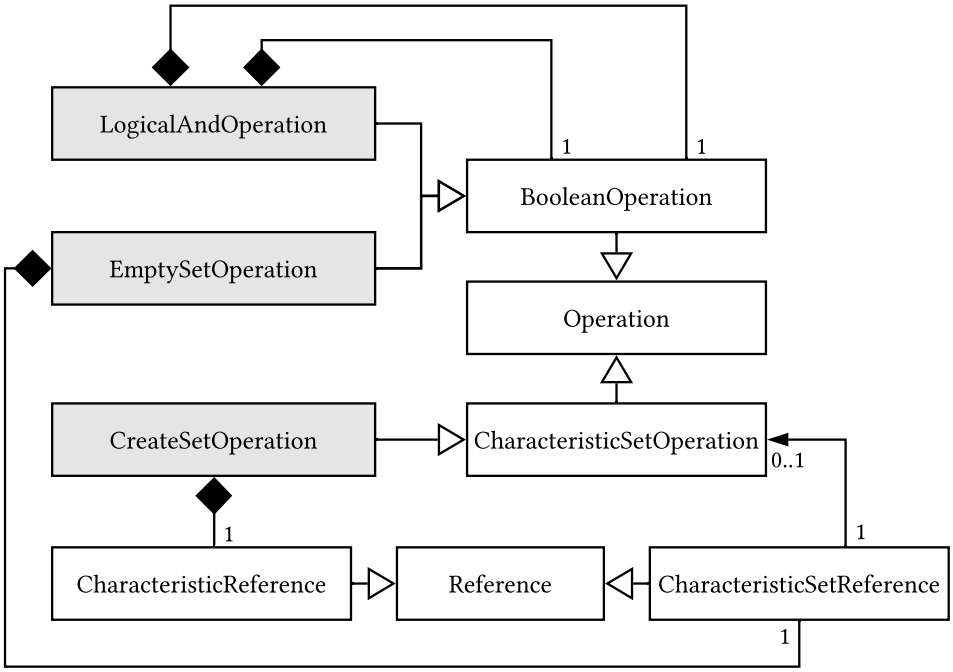

Figure 7.5.: Abstract syntax of exemplary operations and references to variables

a single CharacteristicVariable to a CharacteristicSet with a single entry (the variables value). Thus, the operation refers to one CharacteristicReference. The CreateSetOperation is a CharacteristicSetOperation and can thus be referenced from other operations which accept a CharacteristicSetReference. An example for such an operation is the EmptySetOperation which tests a referenced CharacteristicSet for emptiness. Because a set can either be empty or not, this represents a BooleanOperation. The LogicalAndOperation accepts two arbitrary BooleanOperations and returns the conjunction of their boolean values. Thus, it represents a BooleanOperation itself. We don't use References for BooleanOperations because these only exist in the Condition and can be nested directly. This is not the case for CharacteristicVariables and CharacteristicsSets which originate from DataSelectors or DestinationSelectors and thus have to be referenced.

The containment relations in Figure 7.5 show how multiple Operations can be nested. For instance, a CreateSetOperation can be used as argument in a EmptySetOperation using a CharacteristicSetReference. Another example is the conjunction of multiple EmptySetOperations using the LogicalAndOperation. Also LogicalAndOperations can be nested in other LogicalAndOperations. This approach gives the architect the freedom to formulate complex Conditions while only accepting syntactically correct nesting. We discuss this well-formedness of nested Condition operations using the lambda-calculus in Section A.1.

In total, we define a base set of 11 Operations which are shown in Figure 7.6. These operations originate from basic literature on set theory [5, 10]. The *intersection*, *union* and *subtraction* of sets are considered to be basic operations to create new sets [10, p. 46]. We add an operation to *convert* an element to a set which contains only this element, since

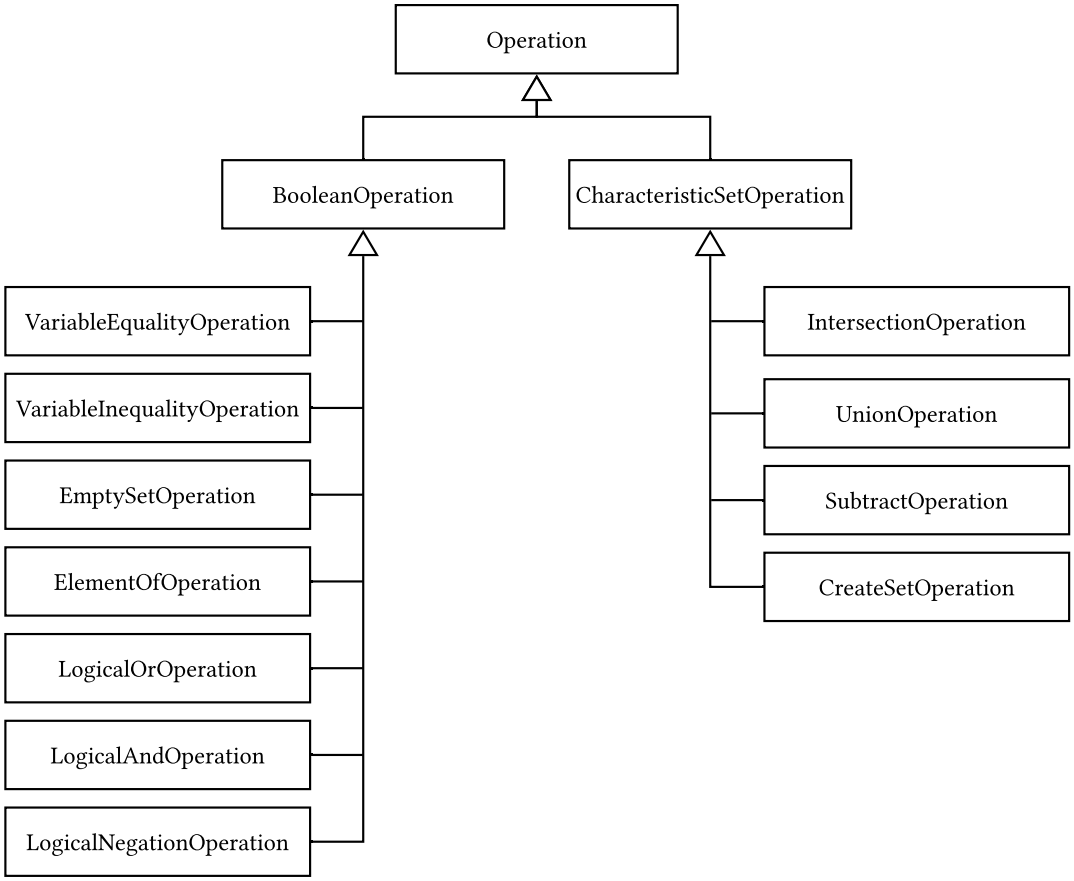

Figure 7.6.: Abstract syntax of condition operations (simplified)

elements and sets are not equivalent [10, p. 39]. Other operations check if a set *contains* a specified element and if a set is *empty* [10, p. 40]. We add operations to test the *equality* and *inequality* of variables [5, p. 23]. In order to create more complex Conidtions, we add three boolean operations: Conjunction (And), Disjunction (Or) and Negation (Not). These represent a complete operator system. The selected Operations are sufficient to express all constraints from the case studies presented in Section 6.1.

We list our selection of CharacteristicSetOperations in Table 7.1 and our selection of BooleanOperations in Table 7.2. For each operation, we show the expected arguments, discuss the operation's result and give a short example. In order to simplify the notation, we use numbers instead of characteristic literals. However, this has no impact on the semantics of these operations. Please note, that the examples do not represent our concrete syntax but rather use mathematical and logical symbols and notations.

In order to simplify the base set of Operations of our DSL, we exclude several Operations which can be expressed by combining included Operations. We discuss four examples in the following. Here, upper case letters (like A) represent set variables, lower case letters (like x) represent variables.

Operation	Arguments	Result	Examples
Intersection	Two set variables	A set variable containing all literals which occur in both arguments	$A = \{1, 2, 3\}$ $B = \{2, 3, 4\}$ $A \cap B = \{2, 3\}$
Union	Two set variables	A set variable containing all literals which occur in at least one of the arguments	$A = \{1, 2, 3\}$ $B = \{2, 3, 4\}$ $A \cup B = \{1, 2, 3, 4\}$
Subtract	Two set variables	A set variable containing all literals which occur in the first but not in the second argument	$A = \{1, 2, 3\}$ $B = \{2, 3, 4\}$ $A \setminus B = \{1\}$
Create Set	One variable	A set variable which only contains the argument	$A = 1$ $set(A) = \{1\}$

Table 7.1.: All CharacteristicSetOperations which can be used in conditions

- *Test for disjoint sets*: A common use case is to test whether or not two sets contain no matching items. We exclude this operation since it can be easily expressed by testing if the intersection of two sets contains zero elements and thus represents the empty list. Formally: $distinct(A, B) \Leftrightarrow (A \cap B) = \emptyset$
- *Equality of sets*: The equality of sets can be expressed by testing whether or not both sets become empty if subtracted from another: $A = B \Leftrightarrow (A \setminus B = \emptyset) \,\&\, (B \setminus A = \emptyset)$
- *Adding an element to a set*: We do not include an operation which adds an element to a set. Instead we use the CreateSet-operation together with the Union operation: $addElement(A, x) = A \cup set(x)$
- *Removing an element from a set*: We do not include an operation which removes an element from a set. We use the CreateSet-operation together with the Subract-operation: $removeElement(A, x) = A \setminus set(x)$

Please note, that this enumeration is not comprehensive. This is also the case for our 11 selected Operations. In order to cope with this problem, we minimize the change impact of adding new operations to the DSL. Figure 7.5 shows that all concrete operations inherit from either BooleanOperation or CharacteristicSetOperation. Other than that, they only have containment relations to References or Operations which are used as arguments. Thus, we can add or remove Operations easily without changing the meta-model at another point.

Last, we discuss our *exemplary* concrete syntax of Operations in the Condition part of a Rule. Listing 7.4 shows multiple operations which use References on one characteristic variable in line 1 and two characteristics set variables in line 2 and 3. Please note, that the conditions starting in line 5 do not represent complete constraints but are used to show multiple usage scenarios of Operations.

Operation	Arguments	Result	Examples
Variable Equality	Two variables	*true*, if both variables' literals are equal. *false*, otherwise.	$(3 = 3) = true$ $(1 = 4) = false$
Variable Inequality	Two variables	*true*, if both variables' literals are unequal. *false*, otherwise.	$(3 \neq 3) = false$ $(1 \neq 4) = true$
Empty Set	One set variable	*true*, if the set variable contains zero elements. *false*, otherwise.	$(\{3\} = \emptyset) = false$ $(\{\ \} = \emptyset) = true$
Element of	One variable and one set variable	*true*, if the set variable contains the others literal *false*, otherwise.	$(3 \in \{3, 5\}) = true$ $(3 \in \{4, 5\}) = false$
Logical Or	Two boolean operations	*true*, if at least one of the operations evaluates to *true*. *false*, otherwise.	$(true \vee false) = true$ $(false \vee false) = false$
Logical And	Two boolean operations	*true*, if both operations evaluate to *true*. *false*, otherwise.	$(true \wedge false) = false$ $(true \wedge true) = true$
Logical Negation	One boolean operation	*true*, if the operation evaluates to *false*. *false*, otherwise.	$\neg true = false$ $\neg false = true$

Table 7.2.: All BooleanOperations which can be used in conditions

```
1  data.attribute.privacy.$A
2  data.attribute.privacy.$B{}
3  data.attribute.privacy.$C{}
4
5  isEmpty(B)
6  isEmpty(intersection(B,C))
7  elementOf(A,B)
8  isEmpty(B) & !(isEmpty(C))
```

Listing 7.4: Concrete syntax of conditions using operations

In order to represent a valid Condition, the most-outer operation of line 5 to 8 has to be a BooleanOperation. This is the case because EmptySetOperations, ElementOfOperation and LogicalAndOperation inherit from BooleanOperation. The Condition in line 5 evaluates to *true* if the set represented by the characteristic set variable *B* has zero literals. Line 6 evaluates to *true* if the intersection of the characteristic set variables *B* and *C* is empty. The Condition in line 7 returns *true* if the characteristic literal represented by the variable *A* is also contained in the characteristic set *B*. And line 8 is *true* if the set variable *B* has zero elements but the set variable *C* has at least one and is thus not empty. Using this simple notation, architects can formulate even complex conditions in a straightforward manner.

7.5. Constraints

We motivated the structure of constraints in Section 7.1. In this section, we discuss further details of constraints and show their concrete syntax using the building blocks presented in Section 7.2 to Section 7.4.

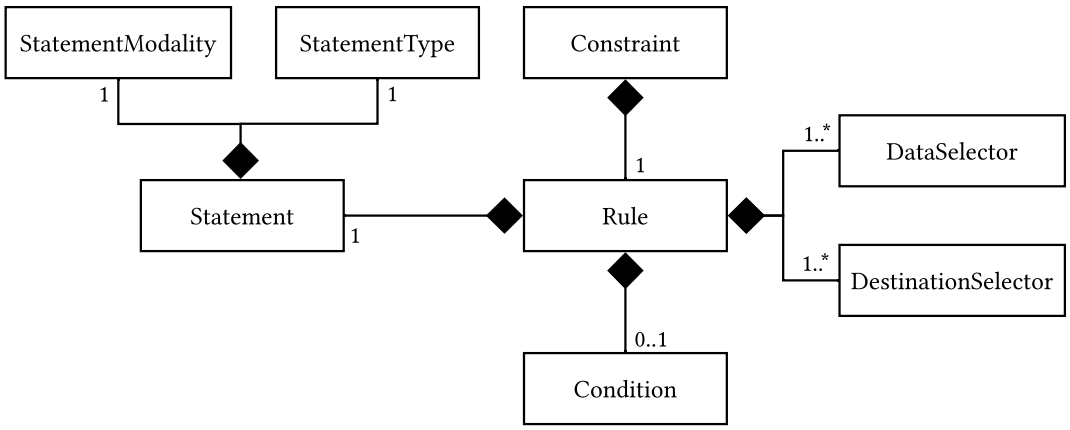

Figure 7.7.: Abstract syntax of constraints and their contained elements

We show the abstract syntax of Constraints in Figure 7.7. A constraint has an unique name and exactly one Rule. We model the Rule explicitly to enhance the expandability of

our DSL, e.g. by adding the possibility to combine multiple rules. Each Rule has exactly one Statement which holds details on the concrete flow restriction. The restriction is defined by its StatementModality and its StatementType. We also model these explicitly to enable further refinement in future versions of the DSL. Currently, the only possible StatementType restricts the *flowing* property of data. The only possible StatementModality is *never*. Following the foundations of modal logic, other modalities like *always* are imaginable. Combined, the statement of Rules in the current DSL restrict the data to *never flow*.

We select data which is restricted to *never flow* using DataSelectors and the destination of such flows using DestinationSelectors. Every rule is required to have at least one DataSelector and at least one DestinationSelector. It's also possible to select data using multiple DataSelectors (**R13**) in a single Rule. This is not equivalent to selecting multiple literals from a single characteristic but rather enables the architect to apply further filtering on the selected data by combining more than one characteristic selection. The same is possible for DestinationSelectors. A simple example is considering data security: if e.g. personal data is only allowed to flow outside of an internal server if it's encrypted, this can be achieved by combining a DataSelector on the *Privacy* characteristic of the data with a DataSelector on a characteristic which represents the data's encryption status. More information on *Selectors* can be found in Section 7.3.

In some cases, selecting data or destinations by specifying concrete characteristic literals is not sufficient. We defined characteristic variables which represent placeholders for possible characteristics literals in Section 7.3. If architects use variables, they are required to make further assumptions in the Condition part of a Rule. Thus, it depends on the data and destination selection whether a Condition is required. We discussed the structure and restriction capabilities of Conditions in Section 7.4.

We combine the parts of the concrete syntax of DataSelectors, DestinationSelectors, Statements and Conditions to formulate complete Rules and Constraints. Listing 7.5 shows the concrete syntax of multiple exemplary constraints. We use the characteristic Location (with the literals EU, USA and Asia).

```
1  constraint SimpleConstraint {
2    data.attribute.location.EU NEVER FLOWS node.property.location.USA
3  }
4
5  constraint WithCondition {
6    data.attribute.location.$DataLocation
7      NEVER FLOWS node.property.location.$DestinationLocation
8      WHERE DataLocation != DestinationLocation
9  }
```

Listing 7.5: Full-fledged constraint given in concrete syntax

Every Constraint starts with the `constraint`-keyword followed by the name of the constraint. The constraint's Rule is surrounded by brackets. Both Rules contain the required elements: DataSelector, Statement and DestinationSelector. The second constraint additionally uses a Condition in line 8. We aim to enable the architect to read constraints

like natural language. Thus, we use speaking names as keywords and structure Rules following the data flow from source to destination. We use upper case letters for some of the keywords like NEVER, FLOWS and WHERE to show a simple partitioning at first glimpse. This approach is influenced by the common declarative database language *SQL*.

The first constraint in line 1 to 3 restricts data which has a characteristic `location` set to `EU` to never flow to a destination with this characteristic set to `USA`. The second condition in line 5 to 9 is formulated less precisely and therefore uses a Condition. Data with a variable `location` is never allowed to flow to a destination with its characteristic `location` set to another literal. Thus, the second constraint implies the first one because the literals of the `location` characteristic differ in the DataSelector and DestinationSelector. However, the opposite is not the case. The second constraint e.g. also restricts data with its `location` set to `Asia` to never flow to a destination with its `location` set to `EU`.

7.6. Example Instances

In this section, we present the application of our DSL in real-world examples. We use the language elements presented in the previous sections to describe the constraints from the example scenarios presented in Chapter 4. For each scenario, we summarize its use case and all defined characteristics with their belonging literals. We analyze the constraints and show how they can be formalized using our meta-model. This includes a discussion of the usage of our *exemplary* concrete syntax.

7.6.1. Geolocation Constraints

The *Geolocation Constraints* scenario in Section 4.1 is motivated by legal restrictions, e.g. from the European *General Data Protection Regulations (GDPR)*. They restrict the flow of personal data and disallow data to be passed to non-whitelisted countries. This constraint does e.g. affect the architecture of (international) online shops which have to synchronize data across borders. However, this is not allowed for personal data. The example instance of the *Geolocation Constraints* scenario uses a Shop Server deployed in the EU while the Database Server is located in Asia.

Using *Data-Centric Palladio*, the following characteristics have been defined to model this scenario:

$$\begin{aligned} \mathit{privacy} &= \{\, \texttt{personal}, \texttt{anonymous} \,\} \\ \mathit{location} &= \{\, \texttt{EU}, \texttt{USA}, \texttt{Asia} \,\} \end{aligned}$$

The privacy characteristic represents the privacy level of data and is personal if it can be associated with a person. The location characteristic represents the location of deployment of servers and components. The constraint is formulated as follows: *"No personal information is processed outside of the EU"*. Following the legal restriction of the *GDPR*, this can also be formulated in a broader sense: *"In unsafe locations, it is never permitted to process or store [personal] data"* [25]. The second constraint uses an additional

classification for *safe* locations (e.g. EU) and *unsafe* locations (e.g. Asia) and is thus more expandable. We model both constraints in the following.

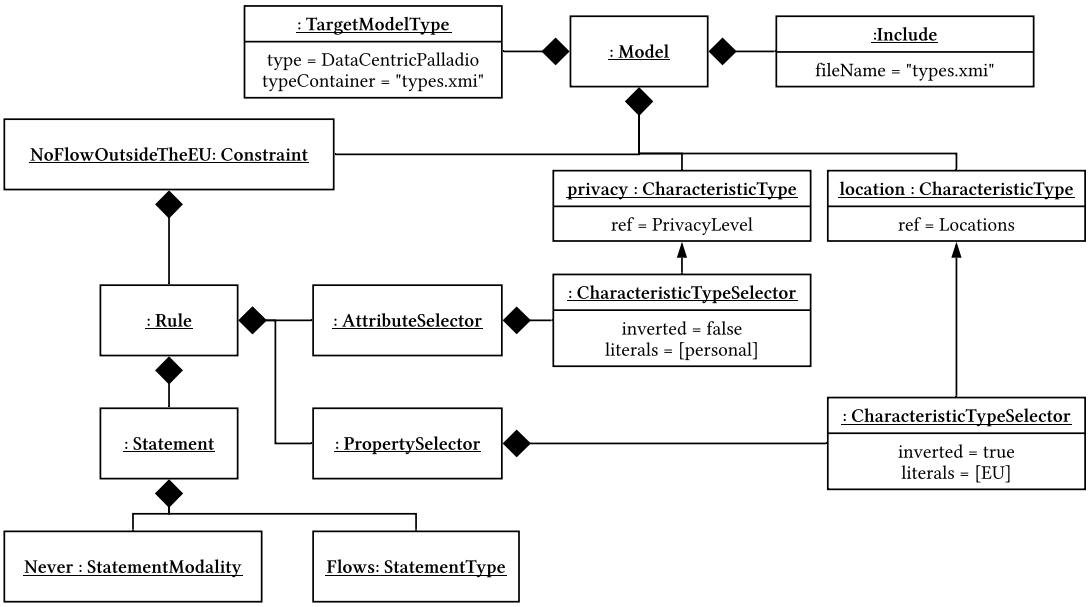

Figure 7.8.: Geolocation constraints example instance using the DSL

Figure 7.8 shows the structure of the first Constraint which prohibits the flow of personal data to destinations outside of the EU. We include an external file *types.xmi* which contains the characteristics of the architectural model created with *Data-Centric Palladio*. We use these characteristics to create CharacteristicTypes called privacy and location. These are referenced in CharacteristicTypeSelectors. The AttributeSelector selects data with the characteristic privacy set to personal. The PropertySelector selects destinations with the characteristic location *not* set to EU. This is represented by the inverted-flag set to *true*. Both selectors are used in a Rule with the Statement *never flows*. The rule is contained in a Constraint with the name NowFlowOutsideTheEU.

The alternative constraint from Kunz [25] expands this constraint by classifying locations to be either *safe* or *unsafe*. We model this scenario using CharacteristicClasses. Figure 7.9 shows the structure of this constraint. We use a CharacteristicClass called UnsafeLocations to embed the CharacteristicTypeSelector used to select all destinations which have a characteristic location not set to EU. We then reference this class in the destination selection part of the Rule using a PropertyClassSelector.

The major benefit of using CharacteristicClasses in this scenario is the interchangeability. Architects can add or remove characteristics and literals in the class without having to change any constraints which reference the class. An exemplary use case in this scenario would be to check data flows to various locations. Additionally, they can reference the class in multiple Constraints since it is an independent element.

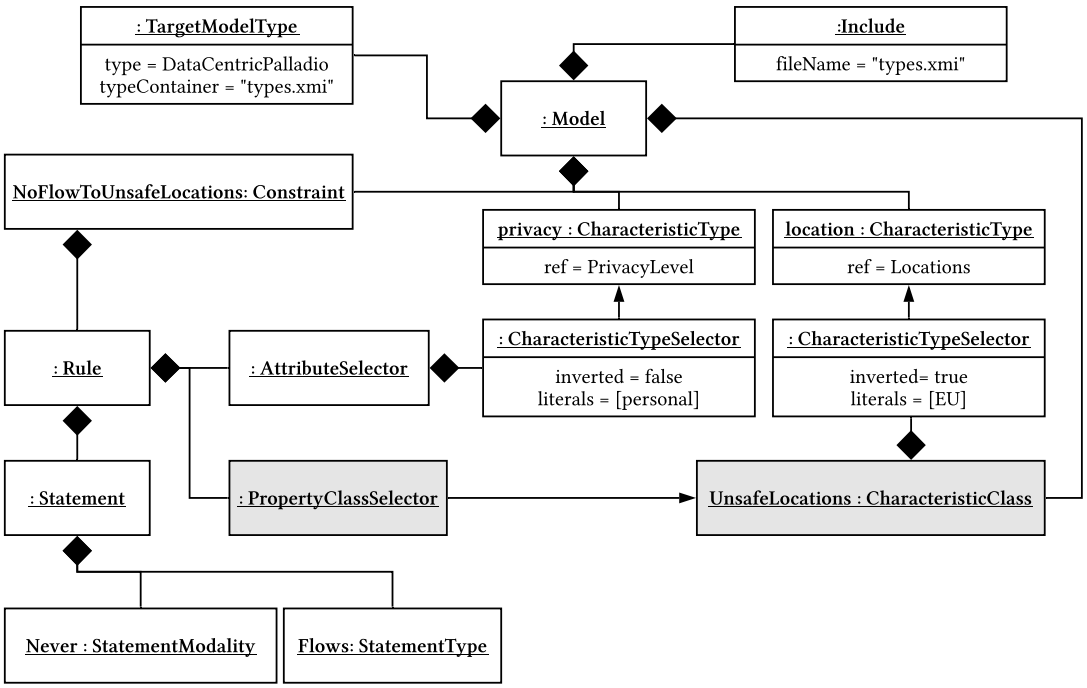

Figure 7.9.: Geolocation constraints example instance with characteristic classes

```
1  target DataCentricPalladio using types
2  import "types.xmi"
3
4  type privacy : PrivacyLevel
5  type location : Locations
6
7  class UnsafeLocations {
8    location.!EU
9  }
10
11 constraint NoFlowOutsideTheEU {
12   data.attribute.privacy.personal NEVER FLOWS node.property.location.!EU
13 }
14
15 constraint NoFlowToUnsafeLocations {
16   data.attribute.privacy.personal NEVER FLOWS node.class.UnsafeLocations
17 }
```

Listing 7.6: Concrete syntax of the Geolocation Constraints example instance

We show the formulation of both constraints using our *exemplary* concrete syntax in Listing 7.6. We define the TargetModelType and include the characteristics definition file in line 1 and 2. We define the CharacteristicTypes which are used in the constraints in line 3 and

4. The first constraint which doesn't use CharacteristicClasses is shown in line 11 to 13. The second constraint in line 15 to 17 uses a reference to the CharacteristicClass defined in line 5 to 7. In this example, the selection of both DestinationSelectors is identical despite the fact that the first constraint directly defines the literals while the second constraint references a class. It's up to the architect whether or not the definition of a CharacteristicClass is worth the additional effort.

7.6.2. Access Control

The second example scenario is based on *Access Control Systems* like the Unix file system. Prior to accessing any data, the actor's role is checked against the list of all authorized roles of the requested data. We illustrated this scenario in Section 4.2 using a simplified version of the TravelPlanner case study [23, 22]. In this example, users request flight offers from a TravelAgency. They choose an offer and book it by directly contacting the offering Airline. This booking process includes the transfer of Credit Card Details which has to be permitted by the user explicitly. Otherwise, this would represent a privacy violation.

Using *Data-Centric Palladio*, the following characteristics have been defined to model this example:

$$\begin{aligned} \textit{roles} &= \{\texttt{User}, \texttt{TravelAgency}, \texttt{Airline}\} \\ \textit{accessRights} &= \{\texttt{User}, \texttt{TravelAgency}, \texttt{Airline}\} \end{aligned}$$

Both characteristics enumerate the same set of literals. Possible literals are the `User` which books the flight, the `TravelAgency` which collects flight offers and the `Airline`. We use the characteristic roles to model an actor's role and the characteristic accessRights to model all actor's roles which are authorized to access a specific datum, e.g. the user's Credit Card Details. The generic constraint for *Access Control Systems* like the presented one can be formulated as follows: *"Data access is only permitted if the actor's role is authorized"*. An actor is authorized, if one of its roles is contained in the accessRights characteristic set of the accessed data.

Figure 7.10 shows the structure of this constraint. The usage of TargetModelType, Include and Statement is equivalent to the *Geolocation Constraints* example explained in Subsection 7.6.1. We import characteristics types and refer to them using CharacteristicTypes. In this scenario, further restrictions to the data flows are defined which cannot be expressed using concrete literals in AttributeSelectors or PropertySelectors. Thus, we declare two CharactersticSet variables which are called rights and roles. These variables are defined as part of the CharacteristicTypeSelectors whose attribute isVariableSelector is set to true. This indicates that variables are used which have to be restricted in the Condition of the Rule.

As discussed before, actors (and thus destinations) are only authorized to access data if at least one of their roles is also contained in the data's access rights set. This can be evaluated by the intersection of both sets. If this intersection is not empty, at least one role exists in both sets which implicates a permitted data access. If the intersection is empty, a violation is found. We model this by using a EmptySetOperation together with a IntersectionOperation. The latter uses two CharacteristicSetReferences to reference the CharacteristicSet variables

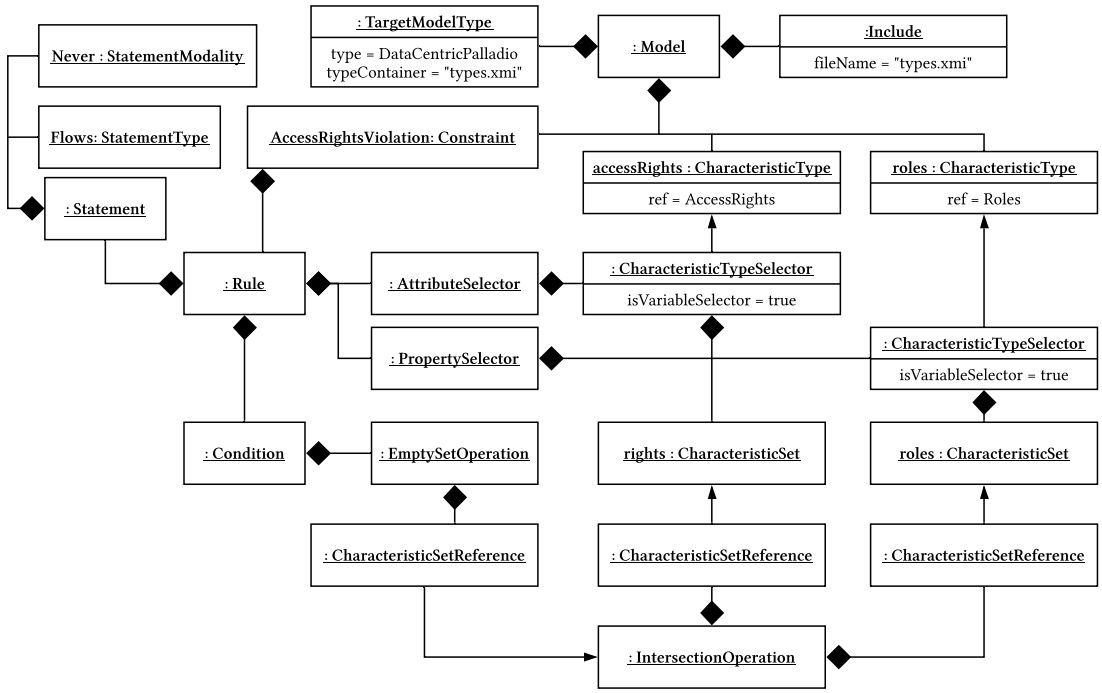

Figure 7.10.: Travel Planner access control example instance using the DSL

created from the AttributeSelector and the PropertySelector. The EmptySetOperation uses a CharacteristicSetReference to refer to the result of the IntersectionOperation. Figuratively speaking, this represents nesting both operations. The EmptySetOperation is contained by the Condition which expects a boolean results; in this case if the intersection is empty or not.

```
1  target DataCentricPalladio using types
2  import "types.xmi"
3
4  type accessRights : AccessRights
5  type roles : Roles
6
7  constraint AccessRightsViolation {
8   data.attribute.accessRights.$rights{}
9     NEVER FLOWS node.property.roles.$roles{}
10    WHERE isEmpty(intersection(rights,roles))
11 }
```

Listing 7.7: Concrete syntax of the Access Control example instance

Listing 7.7 shows the formulation of this constraint using our *exemplary* concrete syntax. Line 1 and 2 are identical to the *Geolocation Constraints* example shown in Subsection 7.6.1: we define the target model and import the characteristics from *Data-Centric Palladio*. In line 3 and 4, we define the CharacteristicTypes needed for the Constraint. The Constraint is called

`AccessRightsViolation` and contains the required elements Statement, DataSelector (in the form of an AttributeSelector) and DestinationSelector (in the form of a PropertySelector). Both selectors do not refer to concrete characteristic literals but define CharacteristicSet variables, called `rights` and `roles`. The Condition refers to these variables by first applying an IntersectionOperation on both sets. The intersection is then checked on it's emptiness using a EmptySetOperation.

CharacteristicSetReferences are only used in the abstract syntax but not visible in the concrete syntax. Their main purpose is to connect an operation to its arguments. These arguments can be concrete variables as seen e.g. in the IntersectionOperation of this example. Alternatively, the arguments are results of other, nested operations. While this difference is important for the abstract representation, it is not for the user. Thus, we decide to use a simple concrete syntax motivated by the functional paradigm which omits the internals. To conclude, we want to point out the similarity of the constraint formulated in our DSL with the constraint written in natural language. We consider architects to be capable of understanding the constraint even with no or only little training in data flow modeling.

8. Language Semantics

In this chapter, we discuss the semantics of our *domain-specific language (DSL)*. The semantics of our language are defined by the mapping from language elements presented in Chapter 7 to parts of executable Prolog programs. This mapping bridges the gap between the architectural domain and the underlying formalism used for the analysis. Additionally, we discuss the mapping of results from the analysis back into the architectural domain. By providing a detailed description of the mapping between both domains, we strive to answer **RQ2**.

We start by giving an overview of the mapping in Section 8.1. This includes a discussion of additional dependencies other than the constraints formulated with our DSL. In Section 8.2 to Section 8.4, we present details of the semantics of individual language elements. We discuss the mapping of analysis results back in the architectural domain in Section 8.5. Last, we show two example transformations based on the example scenarios from Chapter 4 in Section 8.6.

8.1. Overview

Our DSL is used to define data flow constraints. In order to analyze whether or not modeled architectures violate these constraints, they have to be mapped to the underlying formalism Prolog. This mapping is realized using a model-to-model transformation from our meta-model presented in Chapter 7 to a Prolog meta-model developed by Seifermann [40]. Generated prolog models can be serialized afterwards.

As discussed in Chapter 5, this transformation is embedded in the process of *Data-Centric Palladio*. Thus, we use the existing analysis capabilities of the *Operation Model* together with the *Constraint Query API* summarized in Section 6.2. Prior to the definition of our DSL, architects had to manually define constraints by writing Prolog code. By using our DSL together with *Data-Centric Palladio*, both the modeled architecture and its defined constraints are transformed into Prolog code automatically.

The Prolog engine performs the analysis in the underlying formalism by solving the defined constraints. The results, a list of zero or more data flow constraint violations, are transformed back into the architectural domain. The benefit of this additional mapping is the simplified interpretation without the need of knowing the underlying formalism. The result mapping only transforms the results and not the constraints or the architectural model. Thus, our approach is not considered to be round-trip engineering.

In order to generate executable Prolog code which interacts with the *Data-Centric Palladio* environment, we must consider additional aspects as input to our transformation. Strembeck et al. [42] consider the platform of the DSL execution as "software building blocks that provide functions to implement the DSL's semantics" [42]. Examples are

"programming languages and frameworks" [42], e.g. Prolog or the *Operation Model*. In our scenario, additional dependencies occur since the transformation of the DSL also relies on the architectural model and its transformation to the platform.

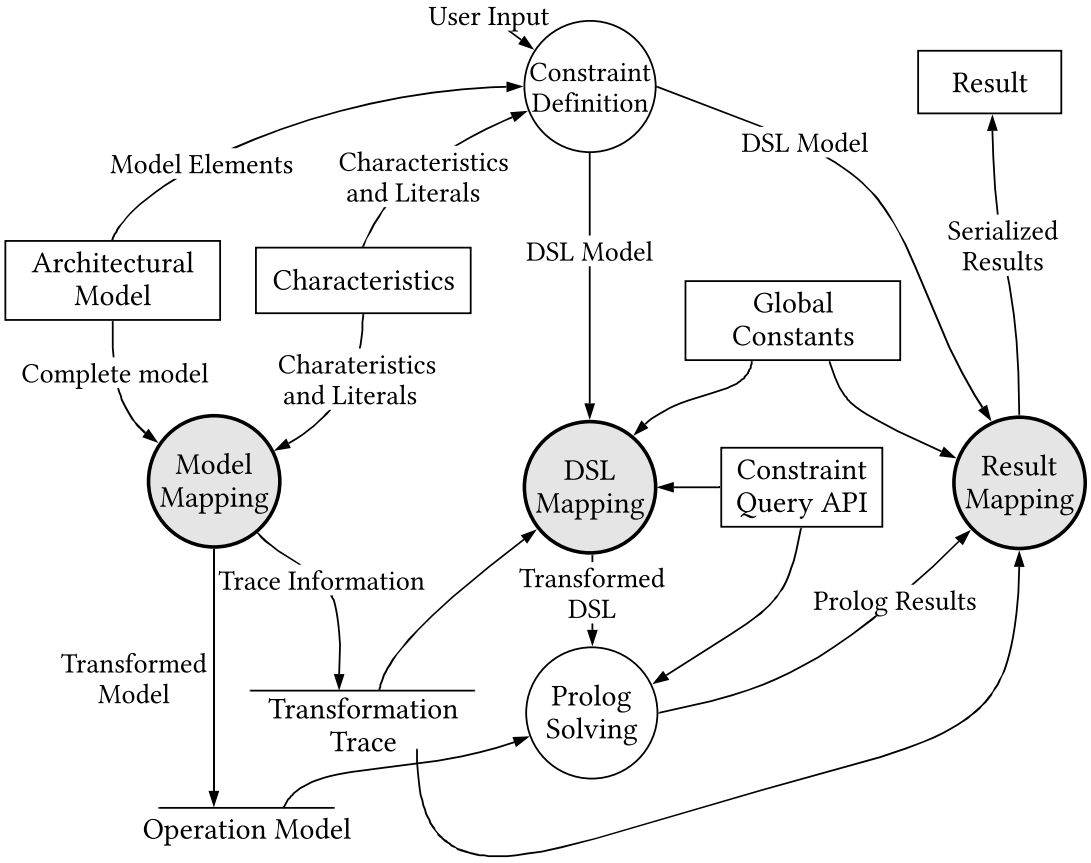

Figure 8.1.: Data flow diagram of the transformation and solving process

Figure 8.1 shows the data flows and thus dependencies between the architectural model and the constraint definitions, their transformation and the underlying formalism. We use the notations of data flow diagrams presented in Figure 2.1. We model the Architectural Model, Characteristics and the Constraint Query API as data sources because they are "lying outside the context of [our] system" [9] and are considered to be available throughout the whole transformation process. Global Constants represent another data source which holds naming conventions for different language elements. These conventions can be chosen freely but need to be the same in the DSL Mapping and Result Mapping process.

We model the Result as data sink which receives the constraint violations mapped back into the architectural domain. We show the definition of constraints using our DSL as process which receives information about the Architectural Model and its Characteristics and returns a DSL model as described in Chapter 7. This is the only step in the transformation process which requires User Input, e.g. an architect defining the constraints. The Prolog Solving process takes both the mapped results from the Model Mapping as well as the DSL

Mapping and uses the Constraint Query API to analyze the constraints and return Prolog Results.

There are three major mappings in Figure 8.1 which are highlighted. The Model Mapping takes information about the Architectural Model and the Characteristics and transforms these into the Operation Model. This process is defined by *Data-Centric Palladio* [41, 25]. Additionally, a Transformation Trace is generated which holds transformation instance specific mapping information. With this information, elements in the transformation output can be traced back to their origin in the Architectural Model.

The DSL mapping is the first transformation we describe in this chapter. We use this mapping to create a Prolog representation of the constraints which can be analyzed afterwards. The execution of this transformation depends on multiple artefacts:

- *DSL Model*: This is the major input of the mapping. This model is an instance of the meta-model described in Chapter 7 and holds all information about the constraints formulated by an architect.
- *Constraint Query API*: The transformation generates Prolog code which calls the Constraint Query API to analyze the transformed architectural model. Thus, the resulting code depends on the definition of the API.
- *Transformation Trace*: An architect can select data flow destinations by their identity (see Section 7.3). The Transformation Trace is used to resolve the transformed element identities, e.g. elements of the architectural model
- *Global Constants*: In order to simplify the Result Mapping, we use constants which exist throughout the solving process and in the Prolog Results. The constants are need to identify specific variables in the Prolog Results, e.g. by using defined variable prefixes

The result of the DSL mapping is the Transformed DSL which is serialized as Prolog code and used for the Prolog Solving process. The results of the analysis represent zero or more constraint violations which are mapped back into the architectural domain to be interpreted by an architect. As with the DSL mapping, the execution of this transformation depends on multiple artefacts:

- *Prolog Results*: The results of the Prolog Solving process represent constraint violations in the form of bound variables, e.g. the specific element where a violation occurs. These violations are the major input of this mapping.
- *Transformation Trace*: The results of the analysis are based on the Operation Model. In order to map them back into the architectural domain, the originating elements have to be resolved. This step is the counterpart to the usage of the Transformation Trace in the DSL mapping.
- *Global Constants*: These constants have been introduced by the DSL Mapping. They can be used to identify relations in the Prolog Results which are not represented by the underlying formalism, e.g. the meaning of a variable

- *DSL Model*: We use the information on the originating constraint definition to enhance the representation of mapped results. By combining information about the constraint and its violations, we aim to simplify the interpretation by an architect.

In the following, we present simplified results of the transformation in order to show the different artefacts on which the DSL Mapping and the Result Mapping depend. We use the *Geolocation Constraints* example scenario with the characteristics `privacy` (`personal` or `anonymous`) and `location` (`EU`, `Asia` or `USA`) presented in Section 4.1. For the complete transformation, please refer to Section 8.6. Listing 8.1 shows the constraint of this scenario. Data which is considered to be `personal` is not allowed to flow to any location which is not considered to be in the `EU`. Both selections use characteristics.

```
1  constraint NoFlowOutsideTheEU {
2    data.attribute.privacy.personal NEVER FLOWS node.property.location.!EU
3  }
```

Listing 8.1: Simplified concrete syntax of the DSL prior to its transformation

Listing 8.2 shows serialized Prolog code which represents the result of the DSL Mapping. Please note that this is only an excerpt; the complete generated code can be found in Section A.3. The clause which starts in line 1 exists to check if the transformed Operation Model contains Call Arguments which violate the constraint discussed above. Call Arguments represent the flow of data to a process as arguments of an operation. The predicate in line 6 represents the selection of `personal` data using the *Constraint Query API*. Line 7 represents the selection of operations with the characteristic `location` not set to `EU`.

```
1  constraint_NoFlowOutsideTheEU_CallArgument(QueryType, OP, S, P) :-
2    QueryType = 'CallArgument',
3    S = [OP | _],
4    stackValid(S),
5    operationParameter(OP, P),
6    callArgument(S, P, 'privacy', 'personal'),
7    \+ operationProperty(OP, 'location', 'EU').
```

Listing 8.2: Excerpt of the DSL transformation result represented as Prolog code

This snippet shows three of the four influences modeled in Figure 8.1. First, the DSL Model contains the information on the constraints and influences the usage of selections and characteristics. Second, the Constraint Query API is used to query the transformed Operation Model and thus influences the structure of the clause and the used predicates. Third, variable names as `QueryType` or `OP` and prefixes as `constraint` originate from the Global Constants. The Transformation Trace is not used in this example because no operations are selected by their identity.

The generated Prolog code from the Operation Model is combined with the generated code from the DSL. Using the latter predicates, goals can be formulated for the Prolog Solving process. If a prohibited data flow exists, the goal succeeds and a solution with bound variables is returned which represents a constraint violation. This violation is mapped back to the architectural domain using Result Mapping.

Listing 8.3 shows the textual representation of an exemplary violation. In line 1 to 4, the `Constraint Details` are summarized. This information depends on the DSL Model input of the Result Mapping. Line 6 - 8 show a detected constraint violation. The variables in line 7 and 8 originate from the Prolog Results and are assigned using the common knowledge of the Global Constants. Line 8 shows the call stack which contains all operations and operation calls which lead to the data flow violation. If the target model (see Section 7.2) is set to *Data-Centric Palladio*, the Transformation Trace is used to map operations from the Operation Model back to their counterpart in the modeled architecture.

```
1  CONSTRAINT DETAILS
2  Data Characteristics: "privacy" set to "personal"
3  Statement: NEVER FLOWS
4  Destination Characteristics: "location" not set to "EU"
5
6  CONSTRAINT VIOLATIONS
7  Parameter "customer" is not allowed to be call argument in operation "store".
8    - Call Stack: "store", "storeUser", "ShopServer_buy", "buy", "usage"
```

Listing 8.3: Simplified textual representation of a violation after the result mapping

8.2. Selection

The selection of data and data flow destinations is a key aspect of our language. Every Rule defined in our DSL is required to have at least one DataSelector and one DestinationSelector (see Section 7.3). Most selectors refer directly or indirectly (using CharactersiticClasses) to characteristics. In order to explain the mapping of those selectors, we first discuss the representation of characteristics in the *Operation Model* and the querying of these using the *Constraint Query API*.

As described in Section 6.2, characteristics are represented as *ValueSetTypes* which are "named finite sets of values" [41]. In the *Operation Model*, data *attributes* and operation *properties* refer to these *ValueSetTypes*. The assignment of concrete characteristic literals depends on the architectural model.

In order to query the *Operation Model* for data and destinations with certain characteristics, the *Constraint Query API* defines five relevant predicates. Data is represented as *call argument* or *return value* of an operation or as globally accessible *call state*. We use the predicates `callArgument`, `returnValue`, `preCallState` and `postCallState` to test for flowing data with specific *attributes*. Destinations are represented as *operations*. We use the predicate `operationProperty` to test for operations with specific *properties*.

In order to combine the selection of data and destinations, we use *call stacks* which are used by the *Operation Model* to represent every possible control flow through the system. The `stackValid` predicate ensures that a *call stack* is correct which is the case if it represents a possible control flow through the system, starting at the *usage* of an user. By utilizing Prolog's backtracking mechanism, we can test all possible control flows and thus all occurring combinations of data and destinations in the *Operation Model*.

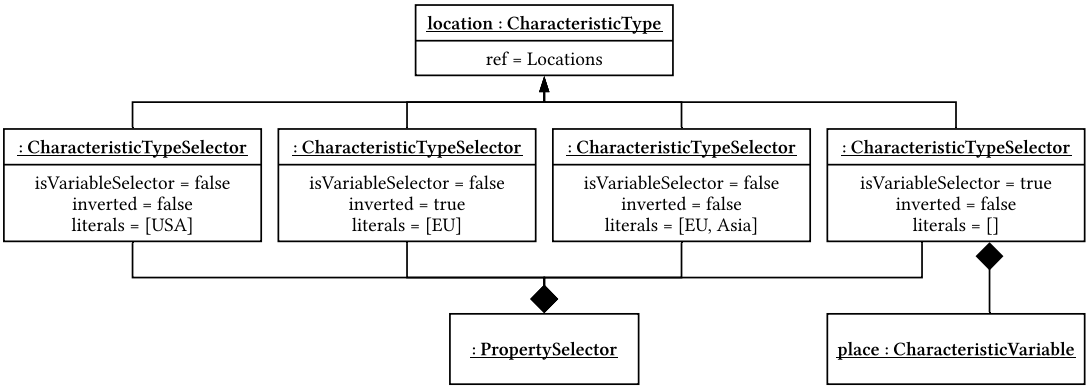

Figure 8.2.: Object diagram of four different CharacteristicTypeSelectors

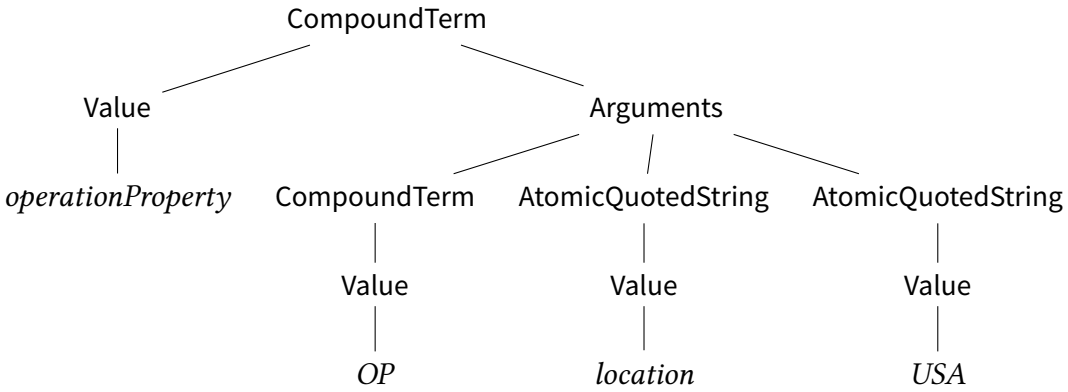

Figure 8.3.: Abstract syntax tree of a mapped characteristic selector

In our DSL, CharacteristicTypeSelectors represent the selection of characteristic literals and are referenced by AttributeSelectors and PropertySelectors. They can select single literals, multiple literals, use inverted selections or define variables. Figure 8.2 shows these four possible combinations which are referenced from a PropertySelector. The left-most CharacteristicTypeSelector only selects destinations with the *property* location set to USA. The next selects only destinations with their location not set to EU. The thirds selects destinations with location set to either EU or Asia. The right-most does not define any literals but uses a characteristicVariable named place instead.

We use the `operationProperty` predicate to specify the relationship between an operation from the *Operation Model* and a characteristic literal. The predicate is used with three arguments: The operation, the characteristic and it's literal. If the specified operation has set the specified literal, the predicate succeeds. Figure 8.3 shows the abstract syntax tree of the generated Prolog code. We use the terminology from the Prolog meta-model [40]. The generated CompoundTerm contains its name (the value operationProperty) and the three arguments. OP represents a variable and is thus represented by another CompoundTerm with zero arguments. location and USA are fixed values represented by AtomicQuotedStrings.

Listing 8.4 shows the Prolog code of the transformed selector shown in Figure 8.3 in line 1. Line 3 to 8 show the other three transformed selectors from Figure 8.2. We omit the remaining three abstract syntax trees for the sake of brevity.

```
1  operationProperty(OP, 'location', 'USA').
2
3  \+ operationProperty(OP, 'location', 'EU').
4
5  (operationProperty(OP, 'location', 'EU') ;
6   operationProperty(OP, 'location', 'Asia')).
7
8  operationProperty(OP, 'location', Var_place).
```

Listing 8.4: Prolog code of the transformed four characteristic selector types

The second transformed selector in line 3 specifies operations which do not have the characteristic `location` set to `EU`. This is realized using Prolog's "\+" predicate which evaluates to true if its argument cannot be proven. The third selector accepts multiple literals from the same characteristic. This is realized in line 5 and 6 using Prolog's `or`-predicate which evaluates to true if one of its two arguments evaluates to true. The last mapped selector in line 8 does not specify a literal but declares a new variable. Further details of this variable behove to the mapped Condition part. In order to identify the variable as characteristicVariable, it is prefixed with the global constant `Var`.

AttributeSelectors are mapped in a similar way using the predicates discussed above: `callArgument`, `returnValue`, `preCallState` and `postCallState`. Since we analyze a data flow as a whole and not its direction, the transformed result uses them all. Additionally, a call stack variable is generated and tested for validity using the `stackValid` predicate.

```
1  S = [OP | _],
2  stackValid(S),
3  operationProperty(OP, 'location', 'USA'),
4  callArgument(S, P, 'privacy', 'personal'),
5  operationParameter(OP, P).
```

Listing 8.5: Prolog code of an transformed attribute selector and a property selector

Listing 8.5 shows the combination of a mapped AttributeSelector together with a mapped PropertySelector and the generated call stack variable. The call stack variable is unified in line 1. Using Prolog's list syntax, we bind the variable `OP` to the head of a list `S`. The underscore indicates that the remainder of the list remains unspecified. The `stackValid` predicate in line 2 ensures that the list `S` represents a correct call stack. The `operationProperty` predicate in line 3 tests the operation at the top of the call stack `OP` for its characteristic `location`. This represents the left-most selector shown in Figure 8.2. Line 4 uses the `callArgument` predicate to test if a parameter `P` holds the literal `personal` of the characteristic `privacy`. The predicate therefore uses the operation at the head of the list `S`. We ensure that this parameter `P` belongs to the operation `OP` using the predicate

operationParameter in line 5. All these predicates are combined using Prolog's built-in conjunction predicate ",".

In addition to the callArgument predicate, other data related predicates returnValue, preCallState and postCallState have to be used because we do not consider the data flows direction only its existence. However, this does not alter the structure of the code snippet presented in Listing 8.5. We show other predicates in Listing 8.6 which can be used instead of the callArgument and operationParameter predicates in line 4 and 5 of Listing 8.5. The arguments of these predicates are similar. In order to cover all data flows, these have to be combined using multiple clauses.

```
1  operationReturnValue(OP, P),
2  returnValue(S, P, 'privacy', 'personal').
3
4  operationState(OP, ST),
5  postCallState(S, OP, ST, 'privacy', 'personal').
6
7  operationState(OP, ST),
8  preCallState(S, OP, ST, 'privacy', 'personal').
```

Listing 8.6: Return value and call state predicates

Another selector type are NodeIdentitySelectors. These selectors don't select destinations by their characteristic but rather by their identity which is represented by the destination's name. This can be represented in Prolog by unifying the operation variable OP with a AtomicQuotedString which holds the name, e.g. "OP = 'destinationName'". However, this simple mapping is only possible if the defined DSL constraint targets the *Operation Model*. If the TargetModelType (see Section 7.2) references *Data-Centric Palladio*, the following additional steps are required to correctly resolve the targeted operation.

If a constraint is defined for *Data-Centric Palladio*, destinations are referenced by a *Component*, its *Assembly Context* and a *Service Effect Specification (SEFF)*. All these elements are contained in the architectural model for which the data flow constraint is defined. In order to reference the correctly named operation in the Operation Model, the transformation of the originating architectural model has to be traced. This is achieved using the *Transformation Trace* which is an additional output of the *Model Mapping* (see Figure 8.1). The *Transformation Trace* holds all mappings from a concrete architectural model to the *Operation Model*. It can be queried using an interface of *Data-Centric Palladio* and returns the name of the mapped operation in the Operation Model. This name is then used for the Prolog unification, e.g. "OP = 'ResourceDemandingSEFF (GDFtwHKJEeq9tYpRa9lb6Q) - AC q7weoHKJEeq9tYpRa9lb6Q'".

This additional resolution step is also needed for the names of characteristics and their literals used by AttributeSelectors and PropertySelectors. Figure 8.1 shows that characteristics are another input of the *Model Mapping*. If constraints are defined for *Data-Centric Palladio* the mapping of characteristics and literals has to be queried using the *Transformation Trace*.

Besides referencing characteristics directly to select data and destinations, CharacteristicClasses can be used. They also select *attributes* inside an AttributeClassSelector and

properties inside a PropertyClassSelector but define the selection independent from the referencing constraint. Thus, they represent an indirection. For more information on CharacteristicClasses see Section 7.3. Listing 8.7 shows a simple example of a CharacteristicClasses which selects locations which are literals as `EU` or `USA` using the `location` characteristic in 1 to 3. Additionally, a PropertyClassSelector in line 5 refers to this class.

```
1  class EUorUSA {
2    location.[EU,USA]
3  }
4
5  node.class.EUorUSA
```

Listing 8.7: Concrete syntax of characteristic class and a attribute class selector

We show the generated Prolog code from the transformation in Listing 8.8. Please note that this snippet doesn't represent a completely transformed constraint but is only used to illustrate the mapping of CharacteristicClasses. These classes are mapped to separate clauses like `characteristicsClass_EUorUSA` in line 3. For each selection of a class, two terms are generated: First, the `valueSetMember` term in line 4. This belongs to the *Constraint Query API* and is used to ensure that a variable represents a literal from the specified characteristic, e.g. `location`. Second, we add a predicate in line 5 which is used to select the specified literals. Line 1 and 2 are facts which represent this selection. Using Prolog's backtracking mechanism, we ensure that one of them exists and is member of the characteristic `location` in order to make this clause succeed.

```
1  characteristicsClass_EUorUSA_location_0('EU').
2  characteristicsClass_EUorUSA_location_0('USA').
3  characteristicsClass_EUorUSA(ClassVar_EUorUSA_location) :-
4    valueSetMember('location', ClassVar_EUorUSA_location),
5    characteristicsClass_EUorUSA_location_0(ClassVar_EUorUSA_location).
6
7  operationProperty(OP, 'location', ClassVar_EUorUSA_location),
8  characteristicsClass_EUorUSA(ClassVar_EUorUSA_location).
```

Listing 8.8: Prolog code of a transformed characteristic class and its referencing

CharacteristicClasses are defined independently in our DSL and are also transformed independently to their use in constraints. They are defined for characteristics in general and neither bound to *attributes* or *properties*. This binding is defined later in line 7. This line represents an DestinationSelector as described above. The major difference is, that instead of specifying a concrete literal (e.g. `EU`), a variable is inserted. This variable is then used together with the predicate from the transformed CharacteristicClass in line 8.

If the Prolog engine is requested to prove the term `operationProperty` in line 7 for a specified operation `OP` the following steps happen: The unbound variable with the name `ClassVar_EUorUSA_location` is detected and tried to resolve using the term in line 8 which refers to the clause from the transformed CharacteristicClass in line 3. Using the `valueSetMember` predicate, the range of possible values is reduced to all literals of the the

`location` characteristic, e.g. `EU`, `USA` and `Asia`. Then, line 5 further reduces possible values to `EU` and `USA`. By utilizing backtracking, both values are tried to bind to the variable in order to prove the `operationProperty` term. This succeeds if the specified operation `OP` is labeled with one of this characteristics, as stated in the original DSL selector.

The example shown in Listing 8.8 uses several prefixes and postfixes defined in the *Global Constants*. We use these to identify the usage of variables and to avoid naming conflicts. Without this decoration, a characteristic called e.g. `OP` might collide with the identical named term which represents an operation. Additionally, we use an iterator for all transformed selectors of a CharacteristicClass because there is no restriction how many selectors can be contained by a class for a single characteristic. This is similar to Rules which can contain any number of DataSelectors but at least one.

8.3. Conditions

Besides specifying concrete characteristic literals, DataSelectors and DestinationSelectors can define CharacteristicVariables which represent single literals or sets of literals. If such a variable is defined, an additional Condition is required which defines further restrictions on the variable's value. In order to define these restrictions, a number of Operations can be used. Examples are testing if a variable's literal is contained in another CharacteristicSet or if a CharacteristicSet is empty. For more information, see Section 7.4.

In order to define a mapping from Conditions to Prolog, three domain gaps have to be bridged: The mapping of CharacteristicVariables, the transformation of single Operations to Prolog predicates and the representation of the structure of nested Operations of our abstract syntax. The mapping of variables which are introduced in characteristic selectors is similar to the mapping of CharacteristicClasses. Instead of using a *Constraint Query API* like `operationProperty` with a constant literal, a new variable is introduced. We've shown this above in Listing 8.4. We use simple variables to represent CharacteristicVariables and lists to represent CharacteristicSets. This is motivated by the strong support for lists by the Prolog language. For our usage scenario, lists without duplicate entries are close enough to the semantics of sets. Prolog's documentation also refers to sets as "unordered list without duplicates" [44].

```
1  node.property.location.$A{}
2
3  findall(IteratorTemplate, operationProperty(OP, 'location', IteratorTemplate),
       VarSet_A)
```

Listing 8.9: Selector defining a characteristic set variable and the mapped result

To obtain all possible literals of an *attribute* or *property*, we uses Prolog's built-in `findall` predicate. Listing 8.9 shows a DestinationSelector with a CharacteristicSet variable in line 1 and the transformed result in line 3. The `findall` predicate accepts a template variable, a goal to solve and a list to store all results. The predicate tries to satisfy the goal (in this case the `operationProperty` predicate) using backtracking. All successful unifications with the template variable `IteratorTemplate` are stored in the list `VarSet_A`. The prefix `VarSet`

Operation	Prolog predicate	Example
Intersection	`intersection/3`	`?- intersection([1,2],[1,3],X).` `X = [1].`
Union	`union/3`	`?- union([1,2,3],[3,4],X).` `X = [1, 2, 3, 4].`
Subtract	`subtract/3`	`subtract([1,2,3,4],[1,4],X).` `X = [2, 3].`
Create Set	`unification/2`	`?- =(X,[1]).` `X = [1].`
Variable Equality	`unification/2`	`?- 3 = 3.  % true` `?- 3 = 4.  % false`
Variable Inequality	`unification/2`, `not/1`	`?- \+ 3 = 4.  % true` `?- \+ 3 = 3.  % false`
Empty Set	`length/2` *(with length 0)*	`?- length([],0).  % true` `?- length([1,2],0).  % false`
Element of	`memberchk/2`	`?- memberchk(1,[1,2,3]).  % true` `?- memberchk(4,[1,2,3]).  % false`
Logical Or	`or/2`	`?- true ; false.  % true` `?- false ; false.  % false`
Logical And	`and/2`	`?- true , true.  % true` `?- true , false.  % false`
Logical Negation	`not/1`	`?- \+ false.  % true` `?- \+ true.  % false`

Table 8.1.: Mapping of operations of a condition to Prolog predicates

originates from the *Global Constants* and is used to mark this variable as CharacteristicSet variable.

A Condition consists of operations which have to be mapped. Fortunately, Prolog's *lists*-library [44] already contains several common predicates, e.g. `intersection` or `union`. These predicates are defined for sets ("list[s] without duplicates" [44]) and thus suppress duplicates. An example is the `union` predicate which filters duplicates while two lists are combined. Table 8.1 shows all operations which can be used inside Conditions and the predicate they are mapped to. We provide a short example for each predicate.

Although the mapping of operations to Prolog's built-in predicates is straightforward, the mapping of the structure of nested operations needs additional effort. The Condition in our DSL follows the functional paradigm strictly. Operations accept one or multiple arguments and return exactly one return value. There exist no states or side effects. Additional variables other than the CharacteristicVariables defined in selectors cannot be

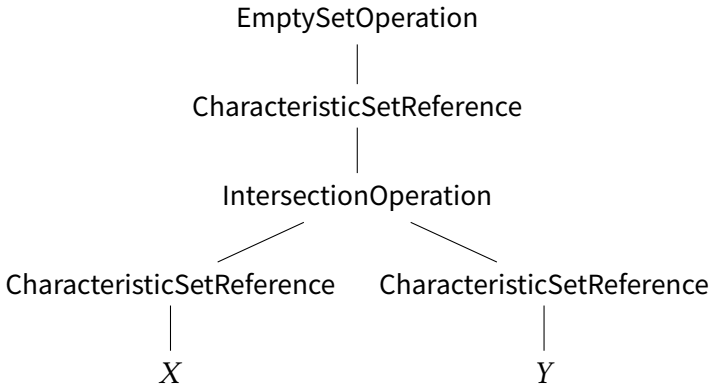

Figure 8.4.: Abstract syntax tree of an exemplary condition in the DSL

declared. On the other hand, Prolog follows the logic paradigm. Using facts and rules, goals can be proven. This is realized by binding all variables in a term. A goal succeeds if all variables can be bound without conflict. The term `X = 3, X = 4.` (*"Unify x with 3 and unify x with 4"*) will always fail because there is no possible value for `X` which satisfies both terms. The term `X = 3; X = 4` (*"Unify x with 3 or unify x with 4"*) always succeeds. Prolog doesn't have concepts like return values from functional programming. Goals can either be proven and all variables bound, or the goal is not provable. Cabot et al. also encounter this gap while mapping the *Object Constraint Language (OCL)* to Prolog [6].

In order to map nested operations from the functional paradigm to Prolog, we have to make up for the missing concept of return values. To do so, we introduce new Prolog variables for every transformed operation. The only exceptions are LogicalAnd, LogicalOr and LogicalNot which can be transformed directly.

We discuss a simple example for the usage of these variables in the following. Listing 8.10 shows two nested operations in line 1 and the transformed Prolog code in line 3. The Condition term tests if the intersection of two CharacteristicSets is empty. This is mapped to a `length` predicate and `intersection` predicate as described above in Table 8.1. The CharacteristicSets `X` and `Y` are mapped to Prolog variables using the `VarSet` prefix from the *Global Constants* to mark them as set variables. In order to combine the statement of both predicates in line 3, a new variable `Temp_0` is introduced. The Prolog engine unifies the set variables and binds the result to the new variable. Using this variable, the `length` predicate is tried so satisfy which only succeeds if the variable represents an empty list.

```
1  isEmpty(intersection(X,Y))
2
3  intersection(VarSet_X, VarSet_Y, Temp_0), length(Temp_0, 0).
```

Listing 8.10: Nested operations and their Prolog counterpart

We illustrate this gap using abstract syntax trees. Figure 8.4 shows the tree of the condition formulated in our DSL. The nesting is realized using a CharacteristicSetReference in the EmptySetOperation. These references can either refer to nested operations or to concrete CharacteristicVariables. Thus, an architect can nest an arbitrary number of operations.

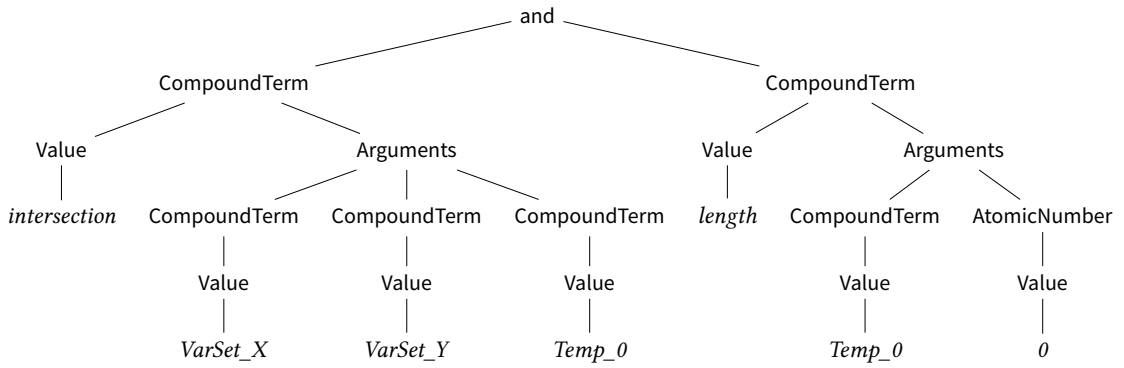

Figure 8.5.: Abstract syntax tree of the transformation to Prolog of an exemplary condition

Figure 8.5 shows the tree of the transformation result in Prolog. The CompoundTerms which represent the `intersection` and `length` predicate are not nested in each other but rather combined using a `and` predicate. This also shows why logical operations (namely LogicalAnd, LogicalOr and LogicalNot) don't need additional variables: They can be directly mapped to their Prolog counterpart.

In the following, we summarize the process of transforming nested operations to Prolog terms. Algorithm 1 shows the pseudo code of our algorithm. We present the mapping of three different operations: LogicalAndOperation which represents boolean operations with boolean arguments, EmptySetOperation which represents boolean operations with a set as argument and IntersectionOperation which represents operations with sets as argument and return type. All other mappings are defined similar to these examples despite the fact that they have differing numbers of arguments.

The algorithm accepts a BooleanOperation as input, which is contained by the Condition element (see Section 7.4). The output is a stack of Prolog CompoundTerms which can be combined using `and` predicates. The mapping is started by calling the `map()` function in line 4. In the shown excerpt, three functions with different arguments types exist. The LogicalAndOperation mapping in line 5 to 11 transforms a conjunction in the DSL to a conjunction in Prolog. Therefore, the left and right subtrees of the conjunction are mapped independently. The call of a `map()` function always results in the transformed prolog term lying on top of the stack E. After both subtrees are mapped and stored in variables, the conjunction of both is pushed back to the stack E to be used in the surrounding operation or returned as result.

The EmptySetOperation mapping is shown in line 12 to 16. This operation accepts a reference to a CharacteristicSet variable or a reference to a nested operation which returns such a variable. This is realized using the `resolve()` operation which expects a Reference. If this is a reference to a real variable used in a selector, the variable is pushed to the variable stack V in line 28. Otherwise, a variable is pushed to the stack in 32 and the operation mapped next which binds this variable. This represents the nesting of operations. Calling the `resolve()` function at the start of the mapping like in line 13, 18 or 19 ensures that either the variable itself lies on the variable stack V or a additional variable lies on the stack V together with the operations which bind this variable lying on the stack E.

Input: A BooleanOperation b which represents the top-level element of a Condition
Output: A stack of Prolog terms E which can be conjuncted

```
1  E ← Empty stack of Prolog terms;
2  V ← Empty stack of variables;
3  i ← 0;
4  map(b);
5  Function map(o : LogicalAndOperation):
6      map(o.left);
7      left ← E.pop();
8      map(o.right);
9      right ← E.pop();
10     term ← AndTerm{left ← left, right ← right};
11     E.push(term);
12 Function map(o : EmptySetOperation):
13     resolve(o.value);
14     variable ← V.pop();
15     term ← Term{value ← "length", arguments ← [variable, 0]};
16     E.push(term);
17 Function map(o : IntersectionOperation):
18     resolve(o.left);
19     resolve(o.right);
20     right ← V.pop();
21     left ← V.pop();
22     temporal ← V.head();
23     term ← Term{value ← "intersection", arguments ← [left, right, temporal]};
24     E.push(term);
25 Function resolve(r : Reference):
26     if r references a variable then
27         variable ← r.variable;
28         V.push(variable);
29     else
30         temporal ←TemporalVariable{value ← i};
31         i ← i + 1;
32         V.push(temporal);
33         map(r.operation);
34     end
```

Algorithm 1: Mapping algorithm of Conditions from the DSL to Prolog

Afterwards, this variable is popped from the stack in line 14 and used to create the CompoundTerm with the `length` predicate in line 15. This term is then pushed to stack E. The mapping of the IntersectionOperation in line 17 to 24 works similarly. Instead of one CharacteristicSet reference, this operation expects two references which explains both `resolve()` calls in line 18 and 19. However, an IntersectionOperation does not return a boolean value but a characteristic set which has to be further evaluated. Thus, the variable

which is bound to the result of the intersection is not popped but left on the stack in line 22. The variable has to be used by the surrounding operation which nested the IntersectionOperation. It is ensured that such an operation exists because only BooleanOperations are allowed as entry point of a Condition.

We illustrate the usage of this algorithm by discussing the single transformation steps of the Condition shown in Listing 8.10 and Figure 8.4. The algorithm starts with the EmptySetOperation as input and calls the accurate `map()` function in line 12. This mapping calls the `resolve()` function in line 13 which recognizes the reference to an IntersectionOperation and thus maps this nested operation first using the accurate `map()` function in line 17. Prior to executing this mapping, an additional variable is generated an pushed on the variable stack V in line 32. The mapping of the IntersectionOperation starts by resolving the variables by calling `resolve()` twice in line 18 and 19. Because these reference real CharacteristicSet variables, no further nesting is required and both variables are pushed to the stack onto the existing variable. Afterwards, both variables are popped from the variable stack V and used to create the first CompoundTerm with the `intersection` predicate in line 23. Additionally, the variable (which is now again head of the stack) is referenced but not popped from it. Last, the created CompoundTerm is pushed to the stack E in line 23 and the function returns the control flow to the calling `resolve()` function which returns to the mapping of the surrounding EmptySetOperation in line 13. In line 14, the variable which was left on the variable stack V is popped from it and used afterwards to create a CompoundTerm with the `length` predicate in line 15. This term is pushed to the stack E in line 16. Then, the control flow returns to line 4 and the algorithm ends with two CompoundTerms in the stack E which are combined afterwards using Prolog's `and` predicate. The result is shown as abstract syntax tree in Figure 8.5 and serialized in line 3 of Listing 8.10.

8.4. Constraints

After Constraints have been defined using our DSL, they are mapped to Prolog code. Afterwards, the generated code is combined with the transformed architectural model as input for the Prolog Solving process. In previous sections, we discussed the mapping of single parts of a Constraint, e.g. the selection of data and destinations Section 8.2 and the definition of conditions in Section 8.3. In order to be analyzed, these individual elements have to be combined after the transformation. The result is a Prolog clause which can be queried in the analysis process and combines all mapped predicates.

In Section 8.2, we discussed that multiple predicates of the *Constraint Query API* can be used to query data attributes: `callArgument`, `returnValue`, `preCallState` and `postCallState`. To create a comprehensive result of constraint violations, all predicates have to be evaluated separately. Thus, we map each CharacteristicSelector four times in separate rules.

To identify which query predicate returned a constraint violation, we add an additional `QueryType` variable to each rule. This variable is unified with a string representing its type, e.g. `"QueryType = 'CallArgument'"`. This variable is combined with the result of the selector mapping using Prolog's `and` predicate. If CharacteristicVariables are used to select data or destinations, the mapped Condition is also added. The four clauses are

then combined using Prolog's `or` predicate in one clause which represents the complete Constraint. To associate violations to their originating Constraints, another unification of a variable with the Constraint name is added. The final predicate can then be satisfied by the Prolog engine.

Listing 8.11 shows the general structure of the complete Prolog code generated from a single Constraint (called `ExemplaryConstraint` in this example). Defined Characteristic-Classes are transformed independently and also added to the result as shown in line 1. Moreover, Listing 8.11 shows several prefixes and postfixes which originate from the *Global Constants*. They are used to prevent any naming conflict of the transformed rules.

```
1   characteristicsClass_ExemplaryClass(...) :- ...
2
3   constraint_ExemplaryConstraint(...) :-
4     ConstraintName = 'ExemplaryConstraint',
5     (constraint_ExemplaryConstraint_ReturnValue(...);
6      constraint_ExemplaryConstraint_PostCallState(...);
7      constraint_ExemplaryConstraint_CallArgument(...);
8      constraint_ExemplaryConstraint_ReturnValue(...)).
9
10  constraint_ExemplaryConstraint_ReturnValue(...) :- ...
11
12  constraint_ExemplaryConstraint_PostCallState(...) :- ...
13
14  constraint_ExemplaryConstraint_CallArgument(...) :- ...
15
16  constraint_ExemplaryConstraint_ReturnValue(...) :- ...
```

Listing 8.11: Complete structure of transformed constraints

The Statement *"never flows"* which is used by all Constraints is implicitly mapped. The semantics of this Statement is to blacklist certain data flows. The analysis selects the blacklisted data flows and returns violations of these restrictions. Thus, no further negation or inversion of complete Constraints is necessary. This discussion is supported by the fundamentals of temporal logic. Statements use the modality *never*. The negation of "*never*" ("always not") can be formulated as "*at least once*" (or eventually).

More formally:

$$\neg(\Box\neg P) \Leftrightarrow \Diamond P$$

Thus, finding violations which represent evidence to the contrary of the modality "*never*" is the proposed analysis approach. This relationship between formulated constraints and analysis results has also been identified by Kunz [25].

In order to interpret the results of the Prolog Solving process, all relevant variables have to be declared as arguments in the head of the generated rules. Although Prolog goals only succeed if all variables can be bound without conflicts, the variables' values are not shown to the user if they are not declared as arguments in the queried predicate. Moreover, we do not only use a static set of variable names like `QueryType` or `OP` but rather create

Argument	Appearance	Description
OP	*always*	The OperationModel operation where the constraint violation occurred
S	*always*	The complete call stack of the constraint violation
P	In CallArgument and ReturnValue queries	The datum which caused the constraint violation
ST	In CallState queries	The call state which caused the constraint violation
ConstraintName	*always*	The name of the constraint for which a violation was found
QueryType	*always*	The type of the query which found the violation (possible values are: *CallArgument, ReturnValue, PreCallState and PostCallState*)
ClassVar_ ...	For each referenced CharacteristicClass	Arguments with this prefix indicate the state of an characteristic selected by a referenced CharacteristicClass
Var_ ...	For each defined CharacteristicVariable	Arguments with this prefix indicate the state of a variable which leads to the constraint violation
SetVar_ ...	For each defined CharacteristicSetVariable	Arguments with this prefix indicate the state of a set variable which leads to the constraint violation

Table 8.2.: Possible arguments created by the mapping of constraints

custom variables for CharacteristicVariables and CharacteristicClasses. To obtain all relevant violation details, we gather variables used in the generated rules. This enhances the Prolog results and simplifies the interpretation for the architect after the Result Mapping. Often, variable names are used in more then one rule. For instance, all rules shown in line 10 to 16 in Listing 8.11 use the variable OP which represents the operation under consideration. When these rules are used together in line 5 to 8, it's sufficient to include the variables' name as argument in the surrounding rule once.

This collection approch is embedded in the mapping of Constraints to Prolog Clauses. When a predicate is used inside a clause's body, all arguments are collected and used as arguments of the surrounding clause as well. This ensures that the predicate which represents the entry point to the Prolog Solving process contains all necessary arguments. In Listing 8.11, the clause's arguments in line 3 are represented by the union of the arguments of all used predicates in line 5 to 8 together with the variable ConstraintName.

We show possible arguments in Table 8.2. We also state if the arguments' appearance depends on other properties of the mapped Constraint. For instance, variables which represent the selection of a CharacteristicClass are only generated if a Selector references such a class. For each constraint violation, the arguments' values are returned after the Prolog Solving process.

8.5. Result Mapping

The Prolog Code which results from the DSL mapping is combined with the Code generated from the transformation of the architectural model. The Prolog Solving process yields zero or more constraint violations. In order to ease the interpretation, these violations are mapped back into the architectural domain by the Result Mapping. In Section 8.1, we summarized the influences of this mapping: Besides the raw Prolog Results, information on the Transformation Trace, Global Constants and the constraints originating from the DSL Model is necessary. In the following, we discuss these influences and how they are used to abstract from the underlying formalism.

The Prolog Solving is used to prove a *goal* which is represented by the main predicate which is created by mapping a Constraint. The Prolog environment tries to prove this *goal* which succeeds if at least one constraint violation is found. Otherwise, the environment yields `false` which is mapped to the simple statement: *"No constraint violation was found"*. If the *goal* succeeds, the predicates arguments' bound values are returned. A simple Prolog result which is represented by bound variables is shown in Listing 8.12. We consider this to be the main input of the Result Mapping.

```
1  ConstraintName = 'SampleConstraintName',
2  QueryType = 'CallArgument',
3  OP = 'ResourceDemandingSEFF (_GDFtwHKJEeq9tYpRa9lb6Q) - AC
       _q7weoHKJEeq9tYpRa9lb6Q',
4  S = ['ResourceDemandingSEFF (_GDFtwHKJEeq9tYpRa9lb6Q) - AC
       _q7weoHKJEeq9tYpRa9lb6Q', opCall_f7b0423c |...],
5  ST = 'DB.store.param.input_STATE_5e4e3009',
6  ClassVar_personal = 'EnumCharacteristicLiteral true (_6MC8YHKKEeq9tYpRa9lb6Q)'.
```

Listing 8.12: Raw Prolog result of a single constraint violation

This result shows the values of all arguments discussed in Section 8.4. This state leads to a constraint violation. The Prolog output contains information about the Constraint in line 1 to 2 and about the data flow in line 3 to 6. The values in line 3 to 6 represent elements from the Operation Model. This model is only used by the underlying formalism. Using the Transformation Trace, we can resolve the originating elements from the architectural domain. In this example, line 3 to 5 represent elements from the modeled architecture and the value in line 6 represents a characteristic literal.

The names of the variables in line 1 to 5 and the variable's prefix in line 6 originate from the Global Constants. By using the same constant names in the DSL mapping and the Result mapping, we simplify the identification of the variables' meaning. These constants are

required to match in both mapping executions. The prefix `ClassVar_` in line 6 indicates that this constraint references a CharacteristicClass. However, the Prolog result does not offer any information about the usage or details of this class. We use the original DSL Model as additional input to the Result Mapping to fill this gap.

We identify 4 different tasks in the Result Mapping. We summarize these in the following:

1. *Mapping basic variables*: Variables like `OP` and `ST` represent the point in the system where the constraint violation occurred. Mapping these variables is considered to be the minimal information which is needed by an architect in order to interpret the analysis result. The mapping of these variables depends on the Global Constants as well as the Transformation Trace.

2. *Mapping the call stack*: The call stack represents the stack of operations in the Operation Model and is used internally to query possible data flows. However, this information can also be useful to trace back the call sequence which leads to the constraint violation. Thus, it's mapped back using the Transformation Trace.

3. *Mapping class variables*: For each referenced CharacteristicClass, a variable is generated which holds the selected literal which leads to the constraint violation. The variable is identified by its prefix from the Global Constants. Afterwards, information about the CharacteristicClass is gathered using the variables name and the DSL Model. Combined, this mapping yields the parameter, its literal and the referenced class.

4. *Mapping characteristic variables*: For each CharacteristicVariable which is used in a Selector, a variable is created. After the Prolog Solving process, this variable is bound to the literal which leads to the constraint violation. We map each variables' value using the Transformation Trace.

In addition to the mapped Prolog results, we also process all constraints from the DSL Model and summarize their restrictions. We consider the combination of information about the constraint (e.g. which data and destinations are selected) and information about the constraint's violations (e.g. literal combinations and the call stack) to be easier to interpret.

After the Result Mapping, we serialize the results and show them to the architect. In order to minimize the change impact of different concrete representations of mapped results, we decoupled the mapping and the serialization process. Currently, we support the creation of plain text files as well as the markdown format. Using the latter, information like variable assignment and call stacks can be presented as tables. Additionally, the presentation of variables' values benefits from more formatting options. We implemented the conversion of the abstract Result Mapping results to the concrete representation using the *Template*-Pattern. Thus, other representations can be integrated easily. A conceivable possibility would be the graphical representation integrated into diagrams of the architectural model.

Listing 8.13 shows a exemplary serialized result mapping. First, general information about the scenario, the constraint and the violations is shown in line 1 to 7. In line 9 to 12, we show constraint details. In this example, data is selected using two characteristics and the destination is selected using a CharacteristicClass. Last, all constraint violations are enumerated. For each violation, a short summary is displayed. Afterwards, additional

information like the call stack or selected literals from CharacteristicClasses are shown. Any information which is not used by the constraint under observation is omitted. For instance, no CharacteristicVariables or Conditions are used by the constraint in this example. The Result Mapping and the serialization adapts to the formulated constraint.

```
1  GENERAL
2  Case name: "example-scenario"
3  Constraint count: 1
4
5  CONSTRAINT
6  Constraint name: "SampleConstraint"
7  Violations found: 2
8
9  CONSTRAINT DETAILS
10 Data Characteristics: "sensitivity" set to "high", "encrypted" set to "false"
11 Statement: NEVER FLOWS
12 Destination Classes: "isNotSafe"
13
14 CONSTRAINT VIOLATIONS
15 1. Parameter "details" is not allowed to be call argument in operation "save".
16    - Call Stack: "save", "saveData", "Server_saveData", "usage"
17    - Characteristic Classes: Parameter "location" (Class "isNotSafe") set to "USA"
18 2. Parameter "info" is not allowed to be return value in operation "send".
19    - Call Stack: "send", "analyze", "External_Server", "usage"
20    - Characteristic Classes: Parameter "location" (Class "isNotSafe") set to "Asia"
```

Listing 8.13: Complete textual representation of a mapped constraint violation

8.6. Example Transformations

In this section, we show the application of the mapping in real-world examples. We use the scenarios presented in Chapter 4 which were already used to discuss the language's syntax in Section 7.6. We discuss the constraints and their transformation results and compare them to constraints formulated using the underlying formalism. Additionally, we show exemplary constraint violations and the mapping of these results back to architectural domain.

8.6.1. Geolocation Constraints

The *Geolocation Constraints* scenario deals with the flow of personal data motivated by legal restrictions. The example scenario models an online shop which operates inside the EU but uses servers which are deployed in Asia. The constraint prohibits personal data to flow outside the European Union. Therefore, two characteristics have been defined. The privacy characteristic states whether data is personal or anonymous. The location

characteristic represents the deployment location. In this scenario, possible locations are EU, USA and Asia.

Listing 8.14 repeats the constraint using our *exemplary* concrete syntax. In addition to the basic constraint shown in line 5 to 7, a more generic approach is shown in line 9 to 11. Instead of prohibiting personal data to flow out of the EU, *unsafe* locations are defined using a characteristics class. This class is shown in line 1 to 3. Although both constraints represent different analysis goals, they yield the same constraint violations in this scenario.

```
1  class UnsafeLocations {
2    location.!EU
3  }
4
5  constraint NoFlowOutsideTheEU {
6    data.attribute.privacy.personal NEVER FLOWS node.property.location.!EU
7  }
8
9  constraint NoFlowToUnsafeLocations {
10   data.attribute.privacy.personal NEVER FLOWS node.class.UnsafeLocations
11 }
```

Listing 8.14: Shortened concrete syntax of the Geolocation Constraints example scenario

We transform both constraints to the underlying formalism using our DSL mapping. We use the Operation Model as target model in order to simplify the generated Prolog code. Listing 8.15 shows an excerpt from the transformation result of the first constraint. For the sake of brevity, we only show the main predicate in line 1 to 4 and the clauses generated to handle *call arguments* in line 5 to 11 and *return values* in line 14 to 20. We leave out the generated code which handles *call states*. The full generated Prolog code can be found in Section A.3.

The clause in line 1 to 4 unifies the `ConstraintName` variable with the original name of the constraint. This information is needed in the Result Mapping step to associate single results with their originating constraints. Afterwards, all predicates for handling data flows are referenced. In this simplified example, only *call arguments* and *return values* are under consideration. Both clauses contain a unification of the `QueryType` variable followed by the use of the call stack. Afterwards, data is selected using the `callArgument` predicate in line 11 and the `returnValue` predicate in line 19 together with the selection of the data flow destination using the `operationProperty` predicate.

In Listing 8.16, we show an excerpt from the transformation result of the second constraint. Here, we only consider the handling of *call arguments*. Line 1 to 4 show the transformation result of the CharacteristicClass which is referenced by the second constraint. The location EU is denoted as fact in line 1 which is then used in line 4 inside the predicate which represents the class. This predicate is referenced in line 17 together with the selection of the destination using the `operationProperty` predicate in line 16.

```
constraint_NoFlowOutsideTheEU(ConstraintName, QueryType, OP, S, P) :-
  ConstraintName = 'NoFlowOutsideTheEU',
  (constraint_NoFlowOutsideTheEU_CallArgument(QueryType, OP, S, P);
  constraint_NoFlowOutsideTheEU_ReturnValue(QueryType, OP, S, P)).

constraint_NoFlowOutsideTheEU_CallArgument(QueryType, OP, S, P) :-
  QueryType = 'CallArgument',
  S = [OP | _],
  stackValid(S),
  operationParameter(OP, P),
  callArgument(S, P, 'privacy', 'personal'),
  \+ operationProperty(OP, 'location', 'EU').

constraint_NoFlowOutsideTheEU_ReturnValue(QueryType, OP, S, P) :-
  QueryType = 'ReturnValue',
  S = [OP | _],
  stackValid(S),
  operationReturnValue(OP, P),
  returnValue(S, P, 'privacy', 'personal'),
  \+ operationProperty(OP, 'location', 'EU').
```

Listing 8.15: Excerpt of the DSL transformation result of the first Geolocation constraint

```
characteristicsClass_UnsafeLocations_location_0_NEG('EU').
characteristicsClass_UnsafeLocations(ClassVar_location) :-
  valueSetMember('location', ClassVar_location),
  \+ characteristicsClass_UnsafeLocations_location_0_NEG(ClassVar_location).

constraint_NoFlowToUnsafeLocations(ConstraintName, QueryType, OP, S, P,
    ClassVar_location) :-
  ConstraintName = 'NoFlowToUnsafeLocations',
  constraint_NoFlowToUnsafeLocations_CallArgument(QueryType, OP, S, P,
      ClassVar_location).

constraint_NoFlowToUnsafeLocations_CallArgument(QueryType, OP, S, P,
    ClassVar_location) :-
  QueryType = 'CallArgument',
  S = [OP | _],
  stackValid(S),
  operationParameter(OP, P),
  callArgument(S, P, 'privacy', 'personal'),
  operationProperty(OP, 'location', ClassVar_location),
  characteristicsClass_UnsafeLocations(ClassVar_location).
```

Listing 8.16: Excerpt of the DSL transformation result of the second Geolocation constraint

The additional variable `ClassVar_location` which holds the selected literal is used as argument in both predicates in line 6 and 10. This enables the Result Mapping to display the variables concrete value to the architect in the case of a constraint violation.

We compare the semantics of the transformed constraints to the original constraint from Kunz [25]. In natural language, he describes the Prolog constraint as follows:

*"Does an operation **o** exist within any call sequence, where for any parameter **p** of **o** the [**privacy**] is [**personal**] and the deployment specified by the **location** property of **o** specifies an unsafe location."*

This definition is equivalent to the predicate shown in line 10 to 17. Using Prolog's backtracking mechanism together with the `stackValid` predicate in line 13, any possible call sequence is queried. The selection of any parameter with the matching characteristics is performed by the `callArgument` predicate in line 15 and the `location` is selected using the `operationProperty` predicate together with the `UnsafeLocations` CharacteristicClass in line 16 and 17.

As discussed before, both constraints yield the same violations because they select the same data and destinations. In the following, we discuss the Prolog result and its mapping back to the architectural domain. Listing 8.17 shows an exemplary constraint violation represented by the raw Prolog Solving output of the second constraint using CharacteristicClasses. This result holds the unification for data flow related variables like `OP` and `S` but also mapping related information like the constraint and type of query where the violation was found.

```
1   ConstraintName = 'NoFlowOutsideTheEU',
2   QueryType = 'CallArgument',
3   OP = 'UserDB_store',
4   S = ['UserDB_store', storeUser, 'ShopServer_buy', buy, usage],
5   P = customer,
6   ClassVar_location = 'Asia'.
```

Listing 8.17: Prolog solving result of the Geolocation Constraints scenario

Listing 8.18 shows the mapped result. First, information about the constraint is presented in line 1 to 12. Then, all constraint violations are listed. Therefore, the result from the analysis is processed. In this example, the violation occurs because of a `store` operation on a database which is deployed in `Asia` and which handles personal `customer` data.

```
1  GENERAL
2  Case name: "geolocation-constraints"
3  Constraint count: 1
4
5  CONSTRAINT
6  Constraint name: "NoFlowOutsideTheEU"
7  Violations found: 1
8
9  CONSTRAINT DETAILS
10 Data Characteristics: "privacy" set to "personal"
11 Statement: NEVER FLOWS
12 Destination Classes: "UnsafeLocations"
13
14 CONSTRAINT VIOLATIONS
15 1. Parameter "customer" is not allowed to be call argument in operation
       "UserDB_store".
16   - Call Stack: "UserDB_store", "storeUser", "ShopServer_buy", "buy", "usage"
17   - Characteristic Classes: Parameter "location" (Class "UnsafeLocations") set to
         "Asia"
```

Listing 8.18: Mapped result of the Geolocation Constraints scenario

8.6.2. Access Control

The second scenario is based on *Role-based Access Control*. In this case study, the flow of sensitive data from a *Travel Planner* smartphone application to a travel agency and an airline is modeled. We use characteristics which represent the roles of these actors and access rights to different types of data. For instance, the access to credit card details is only permitted to roles authorized by the user. This can be formulated by comparing an actor's roles and the access rights for each datum. If the intersection is empty, no authorized role can be found and thus a constraint violation occurs. We repeat the constraint using our *exemplary* concrete syntax in Listing 8.19. We use CharacteristicSetVariables which represent `roles` and access `rights` in line 2 and 3. In line 4, we reference these variables to create Condition which tests the emptiness of the intersection of both sets.

```
1  constraint AccessRightsViolation {
2    data.attribute.accessRights.$rights{}
3      NEVER FLOWS node.property.roles.$roles{}
4      WHERE isEmpty(intersection(rights,roles))
5  }
```

Listing 8.19: Shortened concrete syntax of the Access Control example scenario

In Listing 8.20, we show an excerpt from the transformation result using our DSL mapping. The predicates contain two additional arguments: `VarSet_rights` and `VarSet_roles`. These represent the CharacteristicSetVariables from the modeled constraint. In line 10 and

11, Prolog's `findall` predicate is used to collect all possible roles for the system state under consideration. The intersection of these roles is then tested for emptiness using Prolog's list and set operations in line 12.

```
1  constraint_AccessRightsViolation(ConstraintName, QueryType, OP, S, P,
       VarSet_rights, VarSet_roles) :-
2    ConstraintName = 'AccessRightsViolation',
3    constraint_AccessRightsViolation_CallArgument(QueryType, OP, S, P,
       VarSet_rights, VarSet_roles).
4
5  constraint_AccessRightsViolation_CallArgument(QueryType, OP, S, P, VarSet_rights,
       VarSet_roles) :-
6    QueryType = 'CallArgument',
7    S = [OP | _],
8    stackValid(S),
9    operationParameter(OP, P),
10   findall(IteratorTemplate, callArgument(S, P, 'authorizedRoles',
       IteratorTemplate), VarSet_rights),
11   findall(IteratorTemplate, operationProperty(OP, 'accessRoles',
       IteratorTemplate), VarSet_roles),
12   intersection(VarSet_rights, VarSet_roles, Temp_0), length(Temp_0, 0).
```

Listing 8.20: Excerpt of the DSL transformation result of the Access Control constraint

We compare the semantics of the transformed constraints with constraints formulated by Kunz [25]. He describes a Prolog constraint which finds access control violations as follows:

> *"Does an operation **o** exist within any call sequence, where for any parameter **p** of **o** with the attribute authorizedRoles, no role **r** exists so that both **p.authorizedRoles.r** and **o.roles.r** are true."*

The iteration over operations in possible call sequences is achieved using the call stack from the Operation Model in line 7 and 8. We collect `authorizedRoles` of the parameter under consideration with the first `findall` predicate in line 10. The `accessRoles` are collected with the second `findall` predicate in line 11. *"Being true"* causes the appearance of a role in one of the two sets. Thus, the lack of existence of at least one role in both sets can be formulated as test for the emptiness of the intersection of both sets as described in line 12.

We show an exemplary Prolog result after the Prolog sovling in Listing 8.21. In addition to the unification of general variables like the constraint name and the call stack, the assignment of both CharacteristicSetVariables is shown in line 6 and 7. In the result shown here, an actor with the role `Airline` tries to access data which is only allowed to be accessed from actors with the role `User`. The intersection of both sets represents the empty set and thus a constraint violation. This can expressed as: $roles \cap rights = \emptyset$.

Listing 8.22 shows the mapped result. After the general information in line 1 to 7, we show the constraint details in line 8 to 13. Both the data and destination selection display the use of CharacteristicSetVariables. The Condition is printed as defined in the DSL. We

choose to display the function in its original shape because we consider a description of the Condition in natural language to become confusing quickly.

```
1  ConstraintName = 'AccessRightsViolation',
2  QueryType = 'CallArgument',
3  OP = 'requestDetails',
4  S = ['requestDetails', fetchDetails, 'TravelPlanner_bookFlight', book, usage],
5  P = request,
6  VarSet_roles = ['Airline'],
7  VarSet_rights = ['User'].
```

Listing 8.21: Exemplary Prolog solving result of the Access Control scenario

Afterwards, the constraint violations are displayed in line 15 to 18. For each violation, the values of all CharacteristicVariables are displayed in addition to the basic information like the call stack or which operation call caused the violation. This information is especially helpful after the problem location has been identified in order to understand the properties of the data causing it.

```
1  GENERAL
2  Case name: "travel-planner-access-control"
3  Constraint count: 1
4
5  CONSTRAINT
6  Constraint name: "AccessRightsViolation"
7  Violations found: 1
8
9  CONSTRAINT DETAILS
10 Data Characteristics: "authorizedRoles" set to variable "rights"
11 Statement: NEVER FLOWS
12 Destination Characteristics: "accessRoles" set to variable "roles"
13 Condition: "isEmpty(intersection(authRoles,accessRoles))"
14
15 CONSTRAINT VIOLATIONS
16 1. Parameter "request" is not allowed to be call argument in operation
       "requestDetails".
17   - Call Stack: "requestDetails", "fetchDetails", "TravelPlanner_bookFlight",
       "book", "usage"
18   - Characteristic Variables: Variable "rights" set to "User", variable "roles"
       set to "Airline"
```

Listing 8.22: Mapped exemplary result of the Access Control scenario

The serialization to plain text files prints all variables an their values as list for each constraint violation. Alternatively, we implemented a serialization which creates markdown files. Here, variables and their values are displayed as simple table which is considered to enhance the readability especially for constraints which utilize many different CharacteristicVariables.

9. Evaluation

In this chapter, we present the evaluation of our approach. We evaluate the *domain-specific language (DSL)* as well as the mapping between the architectural domain and the underlying formalism. In order to minimize the risk of "collecting unrelated, meaningless data" [2] we use a *Goal-Question-Metric*-Plan [2]. Basili et al. propose defining goals first and then deriving questions of interest. While goals are on a conceptual level, questions are on a operational level and try to characterize the "object of measurement" [7]. Metrics are then used to collect (quantitative) data to evaluate questions and measure the fulfillment of previously defined goals.

There is already work on DSL quality [20, 33] and the development of DSLs available [1, 28, 46]. This work identifies several quality attributes of DSLs such as simplicity, uniqueness, consistency, space economy and the choice of the right abstraction level.

We present goals and resulting questions in Section 9.1. In Section 9.2, we explain the evaluation design which is used to answer these questions. We present and discuss the evaluation results in Section 9.3. Afterwards, we discuss threats to validity in Section 9.4 and limitations of our approach in Section 9.5. We close with information on the availability of the data to reproduce our evaluation in Section 9.6.

9.1. Goals and Questions

The goal of this evaluation is to show whether or not the research questions presented in Section 1.1 have been answered satisfactorily. In research question **RQ1**, we ask about architecture-level constructs are necessary to define data flow constraints. Research question **RQ2** asks for a mapping of these constraints from the architectural domain into the underlying formalism as well as a mapping of analysis results.

Based on these research questions, we derive evaluation goals. We evaluate the expressiveness of the DSL (**RQ1**) and the usability and space efficiency as proposed by other work presented above. In order to evaluate the mapping between the architectural domain and the formalism (**RQ2**), we evaluate the equivalence of the analysis. We define the following evaluation goals:

G1 **Expressiveness**: The domain specific language shall provide concepts which allow the versatile constraint specification for different applications.

G2 **Usability**: The domain specific language shall improve an architects analysis productivity by reducing the tasks complexity.

G3 **Space Efficiency**: The domain specific language shall be concise without verbose elements or structures for small and larger problem specifications.

G4 **Equivalence of analysis**: The transformation of the domain specific language to the native formalism shall preserve the semantics.

Goal **G1** evaluates the expressiveness of our DSL. A high expressiveness indicates that many architecture-level constructs are supported and thus many data flow constraints can be expressed using our DSL. In order to discuss the expressiveness, we analyze the formulation of different constraints derived from related work (**Q1**). Additionally, we discuss the usage of well-known terminology and the analysis environment provided by *Data-Centric Palladio*. This includes an evaluation whether the supported concepts to formulate constraints on variables and sets are sufficient (**Q2**).

To evaluate **G1**, we define the following questions:

Q1 Are the supported DSL concepts sufficient to formulate constraints in the context of data flow modeling?

Q2 Does the DSL constraints offer required concepts to reason about characteristics variables and characteristic sets?

Paige et al. [33] stress the importance of simplicity and usability of a DSL. The expressiveness of a DSL has to be paired with decent usability in order to add value to the architect's analysis process. Thus, we evaluate the usability with goal **G2**. First, we discuss the abstraction from the formalism which is used for the analysis (**Q3**). Next, we discuss the DSL's conciseness and generalization (**Q4**). A too low degree of conciseness makes the practical usage of the DSL more difficult while a too high degree of generalization might lead to more abstract concepts which are hard to understand.

Additionally, we evaluate typical language properties like orthogonality (**Q5**) and uniqueness (**Q6**). Orthogonality increases the understandability of constraints because language features do not overlap and thus can be considered one at a time. The concept of uniqueness ensures that multiple language elements cannot be used to express the same constraint which can lead to confusion while learning the language as well as while reading code written by other architects. Ultimately, we consider the knowledge required to use our DSL in comparison to the underlying formalism (**Q7**).

To evaluate **G2**, we define the following questions:

Q3 Does the DSL provide abstraction from the underlying formalism to the architectural domain?

Q4 Is the DSL concise without a too high degree of generalization?

Q5 Does the DSL provide orthogonality, avoiding overlapping features?

Q6 Does the DSL follow the rules of uniqueness, avoiding multiple ways to express the same constraint or analysis goal?

Q7 How much and which knowledge is required to use the DSL for defining data flow constraints?

Goal **G3** evaluates the space efficiency of our DSL. First, we consider lower and upper bounds regarding complexity. The DSL shall be applicable for simple analysis without requiring much boilerplate code (**Q8**). This would cause simple analysis goals to be hard to define and hard to read which also harms the usability. On the other hand, the DSL shall be applicable for larger constraints and analysis goals with multiple goals as well (**Q9**). Last, we compare the space efficiency of the DSL with the underlying formalism (**Q10**). Here, we consider not only the absolute difference in effort to write equivalent constraints but also the scalability of the DSL compared to the underlying formalism with growing number of referenced elements and defined constraints.

To evaluate **G3**, we define the following questions:

Q8 Can the DSL be applied to formulate simple analyses without requiring much boilerplate code?

Q9 Can the DSL be applied to formulate large analyses without being verbose?

Q10 Does the DSL improve space efficiency compared to the underlying formalism?

Last, we evaluate the equivalence of the analysis results in goal **G4**. Without doubt, expressiveness, usability and space efficiency are important properties of a DSL. However, a language created to define constraints becomes less useful if the mapping to the formalism used for the analysis is error-prone. An erroneous transformation might lead to differing analysis results in comparison to constraints defined directly using the underlying formalism. Due to the importance of this topic, we choose to analyze the preservation of semantics by performing a correctness proof (**Q11**) as advised in the domain of compiler construction [30]. Additionally, we compare analysis results from different constraints formulated both using our DSL and the underlying formalism (**Q12**).

To evaluate **G4**, we define the following questions:

Q11 Does the transformation preserve semantics?

Q12 Does using the native formalism compared to the transformed DSL yield the same constraint violations and thus analysis results?

9.2. Evaluation Design

In this section, we discuss the evaluation design and how we strive to answer the questions presented above. In order to avoid overfiting of our DSL, we use additional scenarios other than the exemplary scenarios introduced in Chapter 4. We briefly present all studied scenarios in Subsection 9.2.1. Afterwards, we discuss the evaluation design for each goal in Subsection 9.2.2 to Subsection 9.2.5.

9.2.1. Studied scenarios

Prior to the definition of our DSL, we collected requirements from scenarios from related work which define constraints on the flow of data. We described this approach in Section 6.1.

In total, we identified 7 different scenarios. We choose two scenarios as input for the requirements analysis process. First, the *Geolocation Constraints* [25, 47] scenario which uses two different constraints to restrict the flow of *Type-0* and *Type-1* data. We consider them from now on separately due to the major difference in their goal formulation (see Section 6.1). Second, we analyzed the *Travel Planner* case study [22]. In the following, we briefly describe all 8 scenarios which will be used throughout the evaluation whenever applicable.

Geolocation (Type-0 data). This scenario describes an online shop which handles personal data. Personal "Type-0" data is not allowed to be stored in unsafe locations. For more information, see Section 6.1.

Geolocation (Type-1 data). This scenario also described an online shop which handles personal data. Personal identifiable information is not allowed to be joined with data from other origins. For more information, see Section 6.1.

Geolocation Constraints using encryption. This scenario was defined by Kunz [25] by altering the original *Geolocation Constraints* scenario. The context remains the same describing an online shop which handles personal data. Additionally, a characteristic is defined which describes the encryption status of data. Instead of using a location characteristic to describe the deployment of services, the constraint prohibits the flow of personal but unencrypted data to a specific element of the architecture.

Travel Planner. This case study originates from the iFlow approach [21] and has been also used to evaluate previous work in the context of *Data-Centric Palladio* [41]. It describes the flow of data through a travel planning smartphone application. For more information, see Section 6.1.

ContactSMSManager. This scenario also originates from the iFlow approach [21] and describes the interaction of two smartphone apps: a ContactManager and a SMSManager. The latter is only permitted to access a subset of personal contact data when authorized by the ContactManager. This constraint is another instance of the *Access Control* restriction presented in Section 4.2.

DistanceTracker. This case study also originates from the iFlow approach [21] and describes a fitness app which tracks the GPS position of the user. Afterwards, this data is sent to another service for further evaluation. However, this service is not allowed to gather information about the user's current position. This constraint can also be realized based on the *Access Control* example.

UMLsec Secure Links. The scenario originates from UMLsec [17] and is used to validate the secrecy of selected data. Therefore, an attacker with a set of capabilities (e.g. read or delete) is modeled. Additionally, the transmission of the data under consideration is described. The constraint analyzes different combinations of attackers and transmission types. For instance, an attacker might delete an encrypted data transmission, but cannot ready the data.

UMLsec Secure Dependencies. This scenario also originates from UMLsec [17] and describes a structural security analysis. This analysis verifies whether or not specified security requirements such as secrecy or integrity are fulfilled by all classes which depend on a selected interface. Therefore, the analysis checks whether all connectivity of dependent classes satisfies the requirements.

9.2.2. G1 - Expressiveness

The first goal of this evaluation is the investigation of the DSL's expressiveness. An expressive DSL enables the user to formulate many different scenarios without being limited by the number or power of language concepts and elements. In Chapter 6, we described two approaches: first, analyzing real-world case studies as lower limit of the desired expressiveness. We named this the *top-down* approach. Second, the analysis capabilities of the underlying formalism as upper limit of the power of our DSL. We described this approach being *bottom-up*.

Question Q1. First, we discuss whether or not the current state of our DSL satisfies the collected requirements sufficiently and covers all required concepts. We use all scenarios described in Subsection 9.2.1 to complete our selection for the *top-down* approach. This enables us to test whether or not the expressiveness is sufficient to formulate constraints for 8 scenarios in total. Please note, that this selection is not exhaustive but represents all scenarios gathered in this thesis.

Second, in order to provide a *bottom-up* evaluation, we compare the expressiveness of our DSL with the *Constraint Query API* by Kunz [25]. This API is used by the underlying formalism to validate constraints against the transformed architecture. We briefly described its capabilities in Section 6.2. Our mapping to the underlying formalism uses this API as well as all constraints directly formulated as Prolog code without the use our DSL. Thus, this represents the upper limit of analysis possibilities of the current version of *Data-Centric Palladio*. This API consists of 24 Prolog predicates in total. We evaluate whether each of the predicates can be expressed in our DSL explicitly or implicitly. We consider a predicate to be expressed implicitly if it is either used by the API together with an explicitly expressed predicate or if it is implicitly created by our mapping but not explicitly formulated by an architect.

Third, we discuss which of our requirements from Chapter 6 have been fulfilled. Requirements which have been gathered from real-world use cases but have not been fulfilled indicate a limited expressiveness of our DSL. Thus, we analyze the coverage of requirements.

Question Q2. The highest degree of freedom in goal formulation is offered by the `Condition` part of our DSL described in Section 7.4. Architects can define precise conditions using operations from set theory. We discuss the completeness of the set of operations supported by our DSL. In order to answer this question, we performed a literature research on set theory [5, 10] to collect operations on elements and sets. We also include operations from the `list`-library of SWI-Prolog [44] which are applicable on sets. For each collected operation, we show whether it was included or excluded. For all excluded operations, we show whether or not they can be expressed by combining included operations.

In order to combine multiple operations on sets and elements besides nesting, we added boolean operations to the `Condition` part. We discuss the completeness of selected boolean operations with regard to propositional logic separately.

9.2.3. G2 - Usability

The second goal evaluates the language's usability. Often, the evaluation of usability-related topics includes the execution of user studies. Due to the effort of executing such studies with a representative selection of subjects, we choose to evaluate usability by objective measures. This evaluation is focused on the abstract syntax since this represents the major contribution of Chapter 7. However, we include a discussion of our *exemplary* concrete syntax whenever applicable.

Question Q3. One major goal of DSLs is to provide a sound abstraction from the underlying formalism. With our DSL, we abstract from the Prolog programming language and the *Constraint Query API* [25] which is used for the analysis of *Data-Centric Palladio*.

In order to evaluate the abstraction, we analyze the required effort of different change scenarios in our DSL compared to native Prolog code. Here, less effort indicates good abstraction because the constraint formulation is closer to the real problem under consideration.

In order to enable a fair comparison between the DSL and Prolog, we use handwritten Prolog code instead of generated code from our DSL mapping process. Otherwise, we would run into danger to evaluate the DSL mapping instead of the abstract syntax. We only consider change scenarios which originate from the architectural domain because these are closer to changes in real constraint specifications.

We collected change scenarios by iterating over all elements of the abstract syntax. We omit too strongly related scenarios, e.g. changing variable references in different types of operations which has similar change impacts independent from the selected operation. Furthermore, we do not consider scenarios where elements are deleted since these yield the same numerical results as equivalent scenarios where elements are added to constraints. We collected the following 15 change scenarios:

S0 Constraint creation

S1 Change a Constraint name

S2 Change a characteristic literal of a CharacteristicTypeSelector

S3 Change a characteristic type of a CharacteristicTypeSelector

S4 Change a condition variables name in a CharacteristicTypeSelector

S5 Change a characteristic literal's inversion in a AttributeSelector

S6 Change a characteristic literal's inversion in a DestinationSelector

S7 Change a AttributeSelector to a reference in a CharacteristicClassSelector

S8 Change a NodeIdentitySelector name

S9 Change a CharacteristicSetOperation

S10 Change a BooleanOperation

S11 Add a characteristic literal to a CharacteristicTypeSelector

S12 Add a DataSelector to a Constraint

S13 Add a DestinationSelector to a Constraint

S14 Add a CharacteristicClass and a CharacteristicTypeSelector

S15 Add a Condition and a BooleanOperation

The first scenario **S0** is no real change scenario but included for reference. In this scenario, a minimal constraint with one AttributeSelector and one PropertySelector is created. This constraint's name is changed in scenario **S1**. The scenarios **S2** to **S8** represent basic change scenarios in different DataSelectors and DestinationSelectors. The scenarios **S9** and **S10** represent simple change scenarios in a constraint's Condition. Last, the scenarios **S11** to **S15** represent the expansion of the basic constraint. They range from minimal additions like a new literal in **S11** to larger changes like the addition of a new CharacteristicClass in **S14** or a Condition in **S15**.

We measure the change effort of these scenarios by counting the required atomic steps and distinguish between minor and major changes to the formulated constraints. We consider the change of an attribute value and adding or removing arguments as minor change impact while we consider adding or removing domain elements as major change. We count the changes using the abstract syntax for our DSL and the Prolog meta-model [40] for constraints formulated within the formalism. Additional effort which is required to prepare the constraints to the change scenario such as **S0** is excluded. The measurement always starts with semantically equivalent constraints in both domains.

Question Q4. We discuss the DSL's conciseness. A major strength of DSLs is the language's capabilities to create specifications with less verbosity than the underlying formalism. However, there is an upper limit of the usability of concise language concepts. If the degree of generalization is too high, the language becomes hard to learn and hard to use. In order to measure the DLS's conciseness, we analyze which language concepts are used how often in total throughout all scenarios described in Subsection 9.2.1. Language concepts which are seldomly used indicate a too high degree of generalization and shall be examined more closely. However, this is no comprehensive review because we do not consider our selection of constraints to be exhaustive in the first place.

The choice of evaluated concepts is based on the abstract syntax of our DSL. The more often a concept is utilized, the better. We evaluate how often the following language concepts are used:

C1 Constraints and Rules

C2 CharactersticTypes and CharacteristicTypeSelectors

C3 CharacteristicClasses and CharacteristicClassSelectors

C4 AttributeSelectors

C5 PropertySelectors

C6 NodeIdentitySelectors

C7 Conditions

C8 BooleanOperations

C9 CharacteristicSetOperations

Question Q5. A language benefits from a high degree of orthogonality because this allows the user to interpret language concepts independently. Language orthogonality can also be described as independence of modeling concepts. Orthogonality violations might cause unwanted dependencies between seemingly unrelated elements which leads to worse maintainability and understandability. We evaluate the orthogonality of constraints formulated in our DSL based on the structure presented in Section 7.1: DataSelectors, DestinationSelectors, Statement and Condition. The discussion is based on relations in the abstract syntax which indicate coupling between different parts of the DSL.

Question Q6. Languages which satisfy uniqueness avoid providing more then one way to express semantically identical specifications. Paige et al. [33] state: "By avoiding duplication of features, the language is kept smaller and more explainable". However, uniqueness shall not be mixed-up with the sacrifice of features; it only favors smaller subsets of "powerful features that may be useful in more than one context" [33]. In some cases, uniqueness and expressiveness become mutually exclusive. This requires the careful consideration by the languages designer.

Our evaluation of uniqueness is twofold: First, we analyze the mapping between concrete syntax and abstract syntax. Here, a uniqueness violation would cause confusion which element from the concrete syntax is the correct or best one to describe a constraint. Second, we analyze the uniqueness of different concepts of our language; also by taking the semantics of different elements into consideration. Here, the tolerance of uniqueness violations is higher due to the trade-off between expressiveness and uniqueness. Still, every violation must be weighted up carefully.

Question Q7. We discuss the knowledge which is required to define constraints using our DSL in comparison to the underlying formalism. We distinguish between complexity which is essential to a problem and complexity which is added by accident because of the choice of abstraction or environment [4]. An abstraction like the use of a DSL benefits from minimizing accidental knowledge. This enables new users to learn the DSL more quickly and understand constraints written by other architects more easily. However, it's hard to quantify knowledge. Therefore, we focus on *which* knowledge is required and discuss the amount qualitatively.

We gathered different domain concepts and terminology which influences the process of defining constraints with or without the use of our DSL. This includes knowledge on the syntax and the modeled architecture under consideration but also the analysis environment. As discussed above, complexity which is introduced by the environment and requires knowledge can be essential or accidental. Based on the analysis process described in Chapter 5 and the transformation overview discussed in Section 8.1, we identify the following domains and concepts:

K1 *Data flow diagrams and architectural modeling*, e.g. knowing how to define architectures and use the terminology proposed by data flow diagrams. This is essential to the modeling domain.

K2 *Data-Centric Palladio*, e.g. knowing how to use the modeling environment and define concrete models using *Palladio*. This is essential to use both approaches.

K3 *The architectural model in use and it's characteristics*, e.g. knowing about modeled data and the characteristic's names and literals. This essential to the architect.

K4 *The DSL mapping and result mapping*, e.g. knowing details about the transformation and the internal processing. This adds to the accidental complexity.

K5 *Global constants*, e.g. knowing about the names and usage of these constants which also adds to the accidental complexity.

K6 *Operation Model and Constraint Query API*, e.g. knowing about the structure of the model or how to use single API predicates. This is also considered to be accidental complexity when using the DSL.

K7 *Transformation and trace*, e.g. knowing about the resolution of transformed elements or how to manually resolve them. Manually resolving traces is considered to be accidental complexity and thus avoidable.

K8 *Prolog*, e.g knowing about the logic paradigm, about the basic syntax or about common predicates. This is essential when using the underlying formalism.

K9 *The DSL*, e.g. knowing about the structure of constraints and the concrete syntax. This is essential when using the DSL.

Additionally, we discuss which knowledge is required in order to interpret the analysis results after the result mapping. Ideally, no further knowledge is required other than the knowledge needed to define the constraints using our DSL.

9.2.4. G3 - Space Efficiency

The third goal evaluates the language's space efficiency. A space efficient language can reduce the effort required by "modelers and tools [...] in order to maintain the models" [33]. As already discussed in usability-related questions, there is an upper limit for space efficiency to improve the language. A too high degree of space efficiency harms the usability by impairing the readability.

To evaluate space efficiency, we consider constraints of different sizes which could also be named *scalability*. However, we choose the term *space efficiency* to clarify that the scaling factor under consideration is the number of language elements required to describe a certain specification. This includes discussing the concrete syntax of these elements. We do not consider analysis or transformation performance since both are out of scope and not considered to become critical.

Question Q8. We discuss the formulation of simple analysis goals using our DSL. We construct a minimal constraint which only uses mandatory parts of the Rule. We analyze the space efficiency regarding the abstract and concrete syntax and discuss the trade-off between maximizing space efficiency while retaining the same readability.

Question Q9. This question discusses the opposite of question **Q8**. We consider large analyses such as big constraints with many elements. We investigate vertical composition by adding more elements to an constraint and horizontal composition by adding more

constraints. To evaluate the verbosity of bigger constraints, we construct a constraint which contains all types of elements of the abstract syntax and discuss its readability. Additionally, we discuss the UMLsec secure links scenario which is considered to have a large constraint.

Question Q10. We compare the DSL's space efficiency with the space efficiency of constraints formulated in the underlying formalism. Thereto, we recreate constraints formulated with the DSL using handwritten Prolog code. We don't use generated Prolog code from the DSL mapping in order to exclude the mapping performance from the consideration and thus to provide a fair comparison to the native formalism. This question is related to the discussion of abstraction in question **Q3**. In question **Q10**, we don't consider change effort but count the concrete number of elements of different constraints. Therefore, we use the abstract syntax of the DSL and the Prolog meta-model [40].

We utilize all expressible constraints from the scenarios described in Subsection 9.2.1. We also include the minimal constraint from question **Q8** and the comprehensive constraint from question **Q9**. We evaluate the absolute number of model elements as well as the growth of these numbers for bigger constraints. Since all constraints differ in their structure and use different language elements with different intensities, this enables us to evaluate the space efficiency of these elements compared to native Prolog code. In order to gain additional insights, we perform the change scenarios **S11** to **S15** from questions **Q3** which add different elements to the constraint under consideration.

9.2.5. G4 - Equivalence of Analysis

The fourth goal evaluates the equivalence of the analysis using the DSL compared to the underlying formalism. We define the equivalence as receiving the same analysis results which represent the same constraint violations. Formulating constraints using our DSL shall yield the same results as defining similar constraints using Prolog code and the *Constraint Query API*. This goal is important, because differing analysis results can significantly decrease the benefit gained by using the DSL as abstraction to the underlying formalism.

Question Q11. Due to the importance of this topic, we use a formal correctness proof to evaluate the preservation of semantics. A transformation preserves semantics if the semantics of the language's elements are equivalent to the semantics of the transformed result. This approach is motivated by Narayanan et al. which checked "whether the semantics of the input model were preserved in the output model of a transformation" [32] for graph transformations.

Figure 9.1 shows the structure of correctness proofs performed e.g. in the domain of compiler construction. This approach is based on Morris' advice [30] as well as the discussion by Thatcher et al. [45] and Dyber [11]. The figure shows the transformation γ of a source language L into a target language T. In our scenario, the source language L is represented by the DSL, the target language T by Prolog Code using the *Constraint Query API* and the transformation γ by the DSL mapping. Both languages use semantics θ and ψ which map an arbitrary well-formed language construct to a meaning. This mapping depends on the language contexts C and K which are transformed by π. In our scenario, this context is defined by *Data-Centric Palladio* and its *Operation Model*, respectively. Last,

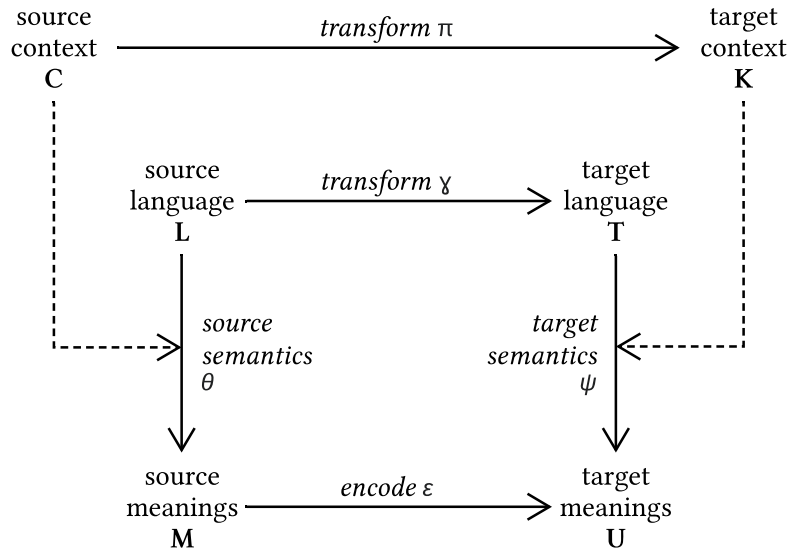

Figure 9.1.: Adapted diagram of Morris on compiler correctness proofs

the source meanings M which are defined by the source semantics θ shall imply the target meanings U defined by the target semantics ψ. If this encoding is erroneous, the interpretation or execution of code in the target language T would not behave as defined in the source language L which would break the correctness. In our scenario, this would yield a differing analysis result.

In order to prepare this proof, we formalize several aspects of our DSL. This includes the source language L, the source semantics θ and the transformation γ. Other parts of this proof originate from other work: The *Constraint Query API* as part of the target language T and its semantics ψ have been defined by Kunz [25]. The source context C and target context K have been defined by Seifermann et al. [41]. In order to construct this proof, we present the formalization of these concepts. Ultimately, we perform a structural induction to show the encoding ϵ for varying language combinations L but fixed transformation rules γ.

We use universal algebra to formalize the language and the transformation as proposed by Jackson et al. [16] to reason about semantics of DSLs. Universal algebra enables us to define "a set of n-ary function symbols for encoding the modeling concepts" [16] which can be nested, "naturally expressing relations over relations" [16]. Jackson et al. define a domain by enumerating all concepts, components and primitives in a structure Υ. The set of all possible model realizations is denoted by R_Υ. In the following we only consider well-formed model realizations.

Transformation rules as defined by our DSL mapping pattern match against the input model realization and generate an output model realization. Accordingly, a transformation γ is a three tuple of disjoint signatures Υ_L and Υ_T of both domains and a set of transformation rules τ. The signatures Υ_L and Υ_T are derived by enumerating all domain concepts of the

source language L and target language T, respectively. A transformational interpretation $[\![\,]\!]^{\gamma}$ is a mapping between model realizations following the rules defined in γ:

$$\gamma = \langle \Upsilon_L, \Upsilon_T, \tau \rangle,$$
$$[\![\,]\!]^{\gamma} : R_{\Upsilon_L} \rightarrow R_{\Upsilon_T}$$

Last, we consider the semantics of our language θ by defining a mapping val_{θ} based on propositional logic. This interpretation evaluates given domain elements from the source language L to *true* or *false*. Accordingly, the mapping val_{ψ} evaluates domain elements from the target language T:

$$val_{\theta} : R_{\Upsilon_L} \rightarrow \{true, false\}$$
$$val_{\psi} : R_{\Upsilon_T} \rightarrow \{true, false\}$$

We use these mappings to describe the structural induction. Our goal is to show the encoding ϵ for every possible realization R_{Υ_L} of the domain Υ_L which is implied by the source language L. We denote the implication between the source meanings M and target meaning U as follows:

$$\forall r \in R_{\Upsilon_{DSL}} : val_{\theta}(r) \implies val_{\psi}([\![r]\!]^{\gamma})$$

The source meanings M are denoted by applying the mapping val_{θ} on a arbitrary source model realization r. This mapping originates from the semantics θ of the source language L. The transformational interpretation $[\![\,]\!]^{\gamma}$ is used to transform this realization to a target model realization $[\![r]\!]^{\gamma}$ based on the transformation γ. Using the semantics ψ of the target language T, we denote the meanings of this transformed realization as $val_{\psi}([\![r]\!]^{\gamma})$. By testing this implication for every source model realization r using a structural induction, we proof the encoding ϵ of meanings.

Question Q12. We compare the analysis results of transformed constraints from our DSL with constraints formulated in the underlying formalism. Therefore, we gathered specified constraints from other work [25, 31] and reformulated the constraints using our DSL. Since the collected constraint specifications originate from other, evaluated theses, we treat them as gold program. We classify each result of our transformed constraints as *true-positive* (accepted a valid violation), *true-negative* (rejected an invalid violation), *false-positive* (accepted an invalid violation) or *false-negative* (rejected a valid violation). This enables us to use the metrics of binary classifications [36]:

$$Precision = \frac{true\text{-}positive}{true\text{-}positive \; + \; false\text{-}positive} \qquad Recall = \frac{true\text{-}positive}{true\text{-}positive \; + \; false\text{-}negative}$$

Please note, that question **Q11** and **Q12** not only use different approaches but also evaluate different aspect of the language semantics. Question **Q11** focuses on the defined

semantics of our language while question **Q12** includes the formulation of constraints based on real examples from other work. Thus, the latter also evaluates the applicability of the DSL's semantics while the formal proof only discusses its correctness.

9.3. Results and Discussion

In this section, we present and discuss the results of our evaluation. In Subsection 9.3.1 to Subsection 9.3.4, we discuss the results for each evaluation goal separately. Afterwards, we summarize the results in Subsection 9.3.5.

9.3.1. G1 - Expressiveness

The first goal **G1** evaluates the expressiveness of our DSL. Question **Q1** discusses whether the DSL covers all required concepts while question **Q2** focuses on the expressiveness of the Condition part of a constraint.

Question Q1. This question is threefold: First, we evaluate the expressiveness in a *top-down* approach by utilizing constraints from other work. Afterwards, we discuss the upper limit of the expressiveness by analyzing the mapping to the underlying formalism and the *Constraint Query API* in a *bottom-up* manner. Last, we discuss whether all requirements presented in Chapter 6 have been satisfied.

As discussed in Subsection 9.2.1, we collected a total of 8 scenarios which can be used to answer this question. Three of these scenarios have already been used as input to the requirements process. In the following, we show whether or not the scenarios can be expressed using our DSL. If a constraint cannot be expressed, we discuss the missing elements. If a constraint can be expressed, we show a short excerpt how it could be formulated using our *exemplary* concrete syntax:

1. **Geolocation Constraints, Type-0 data** [25] ✓
   This constraint can be expressed by formulating a simple Rule with basic CharacteristicTypeSelectors and a CharacteristicClass:
   `data.attribute.level.Type0 NEVER FLOWS node.class.isNotSafe`

2. **Geolocation Constraints, Type-1 data** [25] ✗
   This constraint is formulated by joining two different data flows (see Section 6.1) which cannot be expressed in the current version of our DSL.

3. **Geolocation Constraints using encryption** [25] ✓
   This constraint can be expressed by combining multiple DataSelectors which work with different characteristics with a NodeIdentitySelector:
   `data.attribute.origin.EU & data.attribute.personalInformation.isTrue & data.attribute.encrypted.!isTrue NEVER FLOWS node.name."storeInDB"`

4. **TravelPlanner** [21] ✓
   This scenario was used to gather requirements for the Condition part of constraints. Its constraint can be expressed by using CharacteristicVariables and two Operations:

```
data.attribute.rights.$rights NEVER FLOWS node.property.roles.$roles
WHERE isEmpty(intersection(rights,roles))
```

5. **ContactSMSManager** [21] ✓
   This scenario is another instance of role-based access control systems. It can be expressed by using the same constraint as shown for the **TravelPlanner** scenario.

6. **DistanceTracker** [21] ✓
   This scenario is another instance of role-based access control systems. It can be expressed by using the same constraint as shown for the **TravelPlanner** scenario.

7. **UMLsec Secure Links** [17] ✓
   This scenario models an attacker with varying capabilities which tries to attack differently protected parts of a system. It can be expressed by using multiple CharacteristicClasses and constraints, e.g.:
   `data.class.AllSensitivities NEVER FLOWS node.property.Link.Internet`

8. **UMLsec Secure Dependencies** [17] ✗
   This analysis verifies security requirements of dependent elements of a software system. It cannot be expressed due to the lack of support for the `lnot` predicate of the *Constraint Query API* in the DSL mapping. However, it would be possible to support this predicate without changing the abstract syntax of our DSL.

The DSL is capable of expressing 6 out of 8 constraints or 75%. Next, we discuss our *bottom-up* approach. We collected all 24 predicates of the *Constraint Query API* defined by Kunz [25] and evaluated whether these can be expressed using our DSL. Thereto, we review the DSL mapping and discuss the explicit and implicit usage of these predicates. We consider a predicate to be expressed implicitly if it is either used by the API together with an explicitly expressed predicate or if it is implicitly created by our mapping but not explicitly formulated by an architect.

Table 9.1 shows the result of this analysis. Most of the predicates from the *Constraint Query API* can be expressed. However, most of them are implicitly generated or used inside the API in order to aid the constraint analysis. 5 predicates cannot be expressed at all. In total, 19 of 24 predicates or approximately 80% of all predicates can be expressed. The high number of implicitly expressed predicates can be explained with the additional abstraction of our DSL. Especially the predicates in the *type information* category "can be used for both testing or as generators" [25] when writing constraints using the underlying formalism. Because of the fixed structure of constraints in our DSL which *never* allow the flow of data there is no need for an architect to specify these generators by hand.

Last, we discuss whether or not the requirements presented in Chapter 6 have been satisfied. We used these requirements to define a goal for the expressiveness of our language. Thus, not satisfying a requirement can be considered to be a lack of expressiveness. In total, we defined 13 language requirements. We realized the structure of constraints (**R1**), the selection of data (**R2**) and destinations (**R4**) using characteristics, the selection of elements by their identity (**R5**), the inversion of selections (**R6**), the selection of multiple

Predicate name	Explicitly	Implicitly	Not expressible
Type information			
`isProperty`			✗
`isDataType`			✗
`isAttribute1`		✗	
`isOperation`		✗	
`isSystemUsage`		✗	
`valueSetMember`		✗	
`attributeType`		✗	
`propertyType`			✗
`dataTypeAttribute`		✗	
`operationParameter`		✗	
`operationParameterType`		✗	
`operationReturnValue`		✗	
`operationReturnValueType`		✗	
`operationState`		✗	
`operationStateType`		✗	
Operations, Usage and Calls			
`callArgument`	✗		
`returnValue`	✗		
`stackValid`		✗	
`preCallState`	✗		
`postCallState`	✗		
`defaultState`			✗
`operationCall`		✗	
`hasProperty`			✗
`operationProperty`	✗		

Table 9.1.: Expressiveness of predicates of the Constraint Query API

characteristic literals (**R7**) and the creation of classes (**R8**). Additionally, we created CharacteristicVariables (**R10**), CharacteristicSetVariables (**R11**), Conditions (**R12**) and enabled the combination of multiple selectors (**R13**).

However, two requirements were not realized. We did not add a DataSelector which is able to select data by its type (**R3**) because this can also be modeled using characteristics which has already been discussed by Kunz: "An alternative representation would be to add an additional attribute to the data" [25]. Second, we did not realize the consideration of multiple data flows in one constraint (**R9**). Together, these gaps explain why the DSL is not capable of expressing the *Type-1* constraint from the *Geolocation Constraints* example explained above.

To sum up, these results indicates an overall good expressiveness. 75% of tested constraints can be expressed, utilizing 80% of the predicates offered by the *Constraint Query API*. This has been achieved by realizing 11 of 13 language requirements. Please note that this is no exhaustive answer. It cannot be assumed that this collection of scenarios or constraints is comprehensive. By extending the DSL mapping, these numbers could be improved. However, in order to receive optimal results, the syntax of our DSL has to be altered by adding support for multiple data flows. This is considered to be a major change but feasible due to the orthogonal structure of constraints (see question **Q5** for details).

Question 2. The Condition part of our DSL offers multiple operations to define precise conditions on CharacteristicVariables. Due to the high degree of freedom in the Condition part, we evaluate its expressiveness separately. First, we analyze the selection of operations on elements and sets. Afterwards, we discuss the nesting and combination of these operations using propositional logic.

Table 9.2 shows 18 operations which we collected from literature [5, 10] and the `list`-library of SWI-Prolog [44]. We denote whether an operation is included or excluded. If an operation is excluded, we state whether it is expressible using other, included operations.

We included a set of basic operations which enable the architect to express most of the collected operations. With the 8 operations included, we can express 16 of 18 operations or approximately 90%. We discussed the possibility of expressing excluded operations by combining included operations in Section 7.4. However, the set of selected operations is not minimal. For instance, we included operations for both the equality and inequality of variables which can also be expressed using logical negation. The discussion whether or not a language should only include a minimal set of operations is not considered to be expedient. This is also supported by the fact, that most modern programming languages support more than the absolute minimum of required operations.

We excluded the operation *power set*, because the concept of a set containing other sets is not expressible using our type system and we found no real constraint which would profit from this feature. This is also the case for the operation *element count*. We consider testing for emptiness of sets to be sufficient for most constraints. However, adding other operations on existing types, e.g. testing for disjoint sets is considered to have a low change impact due to the use of polymorphisms.

In order to combine multiple operations on sets and elements beside nesting, we added boolean operations. The combination of AND, OR and NOT represent a functionally complete set of operations thus allowing architects to express any combination of operations using propositional logic. It should be noted that this set is not minimal [48].

Operation	Included	Expressible	Excluded
Element of Set	✗		
Not element of set		✗	
Equivalence of sets		✗	
Empty set	✗		
Subset		✗	
Power set			✗
Subset		✗	
Disjoint set		✗	
Union	✗		
Intersection	✗		
Set difference	✗		
Symmetric difference		✗	
Create set	✗		
Variable equality	✗		
Variable inequality	✗		
Add element to set		✗	
Remove element of set		✗	
Element count			✗

Table 9.2.: Collected operations on elements and sets

#	Description	DSL		Prolog		Difference	
		Major	Minor	Major	Minor	Major	Minor
0	Constraint created	9	0	19	0	10	0
1	Constraint renamed	0	1	0	1	0	0
2	Literal changed	0	1	0	1	0	0
3	Characteristic type changed	0	1	0	1	0	0
4	Condition variable renamed	0	1	0	1	0	0
5	Data selection inverted	0	1	4	0	4	-1
6	Destination selection inverted	0	1	1	0	1	-1
7	Class reference replaced	3	0	2	1	-1	1
8	Node identity changed	0	1	0	1	0	0
9	Set operation changed	0	1	0	1	0	0
10	Boolean operation changed	0	1	0	1	0	0
11	Literal added	0	1	2	0	2	-1
12	Data selector added	2	0	2	0	0	0
13	Destination selector added	2	0	2	0	0	0
14	Class and selector added	2	0	4	0	2	0
15	Condition added	4	0	2	0	-2	0
	Sum 1 - 15	13	10	19	8	6	-2

Table 9.3.: Change effort of different DSL change scenarios

9.3.2. G2 - Usability

The second goal **G2** evaluates the usability of our DSL. Question **Q3** evaluates the abstraction from the underlying formalism, question **Q4** discusses the conciseness and question **Q5** evaluates the language's orthogonality. Question **Q6** checks whether the language satisfies uniqueness and questions **Q7** asks which knowledge is required to use our approach compared to the underlying formalism.

Question Q3. This questions ask whether our DSL provides a significant abstraction from the underlying formalism. Therefore, we analyze the change effort for one constraint creation scenario and 15 different change scenarios. We measure atomic change operations and categorize them into major changes (e.g. adding new elements) and minor changes (e.g. change an existing value).

Table 9.3 shows the change effort of these scenarios. First, we compare the effort to create a constraint in our DSL compared to the underlying formalism. Both constraints shall yield the same analysis results. The effort to create this constraint using Prolog and the *Constraint Query API* is twice as much as using the DSL.

The scenarios **S1** to **S10** represent simple change scenarios, e.g. renaming a constraint or changing the selected literal. Scenarios **S1** to **S4** and **S8** to **S10** show no difference in effort at all. Other scenarios require less effort using the DSL. Only the replacement

of class references in scenario **S7** requires more effort when using the DSL. This is the case because the Prolog representation only requires the replacement of terms while the DSL requires a new selector. In the scenarios **S11** to **S15**, elements are added to the constraint. The difference between DSL and Prolog code is similar to the change scenarios discussed before. However, scenario **S15** shows doubled effort using the DSL in comparison to Prolog. This can be explained by the representation of operations and references to CharacteristicVariables in the DSL. Prolog is well suited for clauses which include the processing of elements and lists.

In summary, the change effort of the evaluated change scenarios is smaller on average. There are less changes required which are also smaller. This can be seen when comparing the difference in major and minor changes. This indicates good abstraction of the DSL from the underlying formalism. Prolog is a high-level programming language with much more formulation possibilities than provided of our DSL. Thus, achieving large differences in the required effort is considered to be difficult.

Question Q4. We evaluate the DSL's conciseness regarding the degree of generalization. A too low degree of conciseness lowers the language abstraction while a too high degree of generalization impairs the language's readability and usability. We measure how often different concepts of our language are used in the constraints considered in this thesis. Here, low numbers indicate a possibly too high degree of generalization and shall be discussed.

Table 9.4 shows the relation between concepts of our DSL and their usage in the scenarios. Concepts **C1** and **C2** are used in every scenario which is implied by them beeing required parts of every constraint. AttributeSelectors (**C4**) and PropertySelectors (**C5**) are also commonly used among all scenarios. Condition related scenarios (**C7** to **C9**) are used whenever variable characteristic literals are under consideration.

The most infrequently used concepts are CharacteristicClasses (**C3**) and NodeIdentitySelectors (**C6**). However, we don't consider them to be a threat to the language's conciseness. CharacteristicClasses are motivated by constraints written in natural language. In more than one case, architects use characteristics as patterns e.g. by defining *unsafe* locations. NodeIdentitySelectors are used to identify elements from the architectural model. In theory, this can always be achieved by defining a new characteristic and applying a literal only to one element. However, this workaround violates the semantics of selecting an element which can be achieved more directly using NodeIdentitySelectors. As discussed previously, our collection of scenarios is not comprehensive. Thus, these numbers can only be seen as indicators.

Question Q5. This question evaluates the language's orthogonality. A high orthogonality is indicated by independent language concepts. This influences other desried properties like readability, maintainability and scalability. The discussion of orthogonality is based on the abstract syntax.

In Section 7.1, we presented an overview of the abstract syntax of our DSL. Here, we discussed the different parts of a constraint: Its Statement, DataSelector, DestinationSelector and Condition. This structure is also shown by the concrete syntax:

⟨DataSelector⟩ ⟨Statement⟩ ⟨DestinationSelector⟩ ⟨Condition⟩

#	**Modeling concept**	Geolocation Constraints	Geolocation w. Encryption	TravelPlanner	ContactSMSManger	DistanceTracker	UMLsec Secure Links
1	Constraints and Rules	✓	✓	✓	✓	✓	✓
2	CharacteristicClasses and CharacteristicClassSelectors	✓	✓	✓	✓	✓	✓
3	CharactersticTypes and CharacteristicTypeSelectors	✓	✗	✗	✗	✗	✓
4	AttributeSelectors	✗	✓	✓	✓	✓	✓
5	PropertySelectors	✗	✗	✓	✓	✓	✓
6	NodeIdentitySelectors	✗	✓	✗	✗	✗	✗
7	Conditions	✗	✗	✓	✓	✓	✗
8	BooleanOperations	✗	✗	✓	✓	✓	✗
9	CharacteristicSetOperations	✗	✗	✓	✓	✓	✗

Table 9.4.: Usage of modeling concepts in case study constraints

The division of constraints in different parts already indicates good orthogonality. However, in order to give a comprehensive review, we discuss all relations between these parts and the rest of the abstract syntax. An overview of the complete abstract syntax can be found in Figure A.1. The Statement of a rule is defined completely independent to other parts of the constraint and thus preserves orthogonality. DataSelectors and DestinationSelectors are defined mostly independent. Both can relate to CharacteristicTypeSelectors and CharacteristicClasses. These contain other CharacteristicTypeSelectors which ultimately refer to CharacteristicTypes. CharacteristicTypeSelectors which are contained directly in DataSelectors or DestinationSelectors are independent from each other. They only might share common references to CharacteristicTypes. None of these references are cyclic. Additionally, CharacteristicClasses are defined independently from their use inside of CharacteristicClassSelectors. In summary, all relations are unidirectional and none of them are considered to be strongly coupled. Thus, all elements can scale independently and the orthogonality is preserved.

Another interaction between parts of a constraint are CharacteristicVariables. These variables are defined and contained in CharacteristicTypeSelector which are used by AttributeSelectors and PropertySelectors. The variables are referenced from inside the Condition part to specify the restriction of the variables' values. In order to decouple this relation, we introduced CharacteristicReferences. These references are used by operations inside the Condition part to either reference variables or other operations. They represent the only way to interact with variables and are defined unidirectional.

In summary, all parts of Constraints have a different concern, do not overlap and are only loosely coupled. Thus, we assume a high degree of orthogonality.

Question Q6. Languages with a high degree of uniqueness avoid the duplication of features which enhance the language's understandability. First, we discuss the mapping between concrete and abstract syntax and whether elements of the abstract syntax can be created by different elements from the concrete syntax. This would violate the uniqueness. Second, we discuss the mapping between our DSL and the underlying formalism.

We realized the DSL by defining a grammar using Xtext [12]. This grammar maps to the abstract syntax. In order to discuss the uniqueness of this relation, we discuss whether this represents a one-to-one mapping where every element of the abstract syntax is related to exactly one element from the concrete syntax. Following the mapping approach, elements from the abstract syntax are created by rules denoted in the grammar. Listing 9.1 shows an excerpt from our Xtext grammar. Line 1, 4 and 11 define new rules. These rules determine the concrete syntax. For instance, line 2 defines the concrete syntax of constraints which start with the keyword `constraint` followed by the constraint's name and its rule which is defined separately. It's also possible to define inheritance relationships. For instance, line 11 to 12 specify AttributeSelectors and AttributeClassSelectors as children of DataSelectors. All these elements are rules of the concrete syntax and also part of the abstract syntax.

```
1   Constraint:
2     'constraint' name=ID '{' rule=Rule '}';
3
4   Rule:
5     dataSelectors+=DataSelector ('&' dataSelectors+=DataSelector)*
6     statement=Statement
7     destinationSelectors+=DestinationSelector
8       ('&' destinationSelectors+=DestinationSelector)*
9     (condition=Condition)?;
10
11  DataSelector:
12    AttributeSelector | AttributeClassSelector;
```

Listing 9.1: Excerpt from the Xtext grammar which defines the DSL

The grammar consists of 43 rules in total. 10 of these rules implicate an inheritance relation like DataSelector and cannot be directly instantiated using the DSL. The abstract syntax contains 42 elements. Each of these rules refer to exactly one element of the abstract syntax and thus preserve the uniqueness. There is only one exception: in order to enhance the readability, we enable architects to encapsulate parts of the Condition using parenthesis. Listing 9.2 shows the excerpt from the grammar where this rule is defined. Using the `return` keyword, we indicate that this rule returns a BooleanOperation and does not create an EncapsulatedLogicalOperation element in the abstract syntax. This represents a threat to the uniqueness because the encapsulation of operations is optional enabling architects to define the same constraint in two different ways. However, allowing the use of parenthesis to enhance the readability or alter the semantics is common among most modern programming languages. As discussed before, uniqueness and expressiveness

can become mutually exclusive when followed blindly. In this case, we prefer higher expressiveness.

```
1   EncapsulatedLogicalOperation returns BooleanOperation:
2     LogicalOrOperation |
3     '(' LogicalOrOperation ')';
```

Listing 9.2: Excerpt from the grammar which defines encapsulated operations

Next, we discuss the mapping between the DSL and the underlying formalism. Here, it's not sufficient to consider the mapping of single elements. Instead, we analyze the semantics of groups of elements and discuss combinations which express similar restrictions. A uniqueness threat occurs whenever a constraint with identical analysis results can be defined using different elements of the abstract syntax. We exclude the reordering of elements which does not introduce new features and thus cannot violate uniqueness.

In the current version of the DSL, we identify two possible threats to the uniqueness: First, the set of operations which can be used inside the Condition part of an constraint is not minimal. Thus, operations can be substituted which violates the uniqueness. As discussed with question **Q2**, this is intentional and can be found in most modern programming languages. As stated above, in case of doubt we prefer expressiveness over uniqueness.

The second threat is the concept of CharacteristicClasses. Everything which can be selected by defining and referencing CharacteristicClasses can also be selected directly by utilizing AttributeSelectors oder PropertySelectors, respectively. This is due to the common containment relation to CharacteristicTypeSelectors. However, the meaning of both language concepts is different: the use of CharacteristicClasses enables architects to define selections of characteristics independently of their use inside of constraints. As discussed in question **Q4**, this is motivated by constraints formulated in natural language. Again, we prefer expressiveness over uniqueness.

In summary, there are only few concerns regarding uniqueness. All possible threats represent a trade-off decision between expressiveness and uniqueness and have been chosen intentionally.

Question Q7. We discuss which knowledge is required to use our DSL compared to using the underlying formalism. Based on the analysis process, we identified 9 different concepts or knowledge categories. Table 9.5 shows which knowledge we consider to be required in order to define constraints in either of both domains.

Some knowledge requirements are independent from the choice whether or not the DSL shall be used. This includes basic knowledge on data flow modeling (**K1**), *Data-Centric Palladio* (**K2**) and the architectural model in use (**K3**). It's also independent of the approach how *much* knowledge is needed because the architectural model and its characteristics have to be defined using both approaches.

When defining constraints without the DSL, knowledge about the formalism is required. This includes understanding the *Operation Model* and the *Constraint Query API* (**K6**). In order to reference elements from the architectural model and its characteristics, architects have to retrieve information from the transformation trace (**K7**). Additionally, at least basic knowledge on the logic paradigm, the syntax and common Prolog predicates is required to define constraints in Prolog code (**K8**).

#	**Concept**	Without the DSL	With the DSL
1	Data flow diagrams and modeling	**Yes,** but only basic knowledge on data flow modeling is required, no details on data flow diagrams.	
2	Data-Centric Palladio	**Yes,** knowledge on defining architectural models, characteristics and *Palladio* is required.	
3	The architectural model in use and it's characteristics	**Yes,** knowledge about the model under consideration and its use of defined characteristics and their meanings is required.	
4	The DSL mapping and result mapping	**No**, because this concept only exists in the approach using the DSL.	**No**, because the mapping is fully transparent to the architect without manual interference.
5	Global Constants	**No**, because this concept only exists in the approach using the DSL.	**No**, because they are added while mapping to Prolog and removed in the result mapping. Thus, they are fully transparent to the architect.
6	Operation Model and API	**Yes**, because constraints are defined against them.	**No**, because they are hidden for the architect and only visible to the automated transformation.
7	Transformation and Trace	**Yes**, because constraints are written using transformed elements such as characteristics.	**No**, because this information is only used in the mapping process.
8	Prolog	**Yes**, at least basic knowledge is required because the constraints are written in Prolog.	**No**, because constraints are written in the DSL and transformed to Prolog automatically.
9	The DSL	**No**, because this concept only exists in the approach using the DSL.	**Yes**, knowledge on the structure of constraints, the concrete syntax and its semantics is required.

Table 9.5.: Required knowledge to define constraints with or without the DSL

When defining constraints by using the DSL, knowledge on the DSL is required (**K9**). This includes at least knowing the structure of constraints, the concrete syntax and common elements. Knowing the internals of the mapping and result mapping (**K4**) or global constants (**K5**) is not required as well as knowing about the *Operation Model* (**K6**), the transformation trace (**K7**) or the Prolog language (**K8**). This is also the case when discussing the mapped results: They reference the architectural model and its characteristics (**K3**) and we consider basic knowledge on data flow modeling (**K1**) to be helpful when interpreting constraint violations.

This discussion yields several results: First, both approaches require common base knowledge (**K1**, **K2** and **K3**). Beside this common knowledge, the specific knowledge required to define constraints using Prolog (**K6**, **K7** and **K8**) and using the DSL (**K9**) is mutually exclusive. This supports the fact that the DSL provides a real abstraction over the underlying formalism. In order to define constraints without the DSL, we count 6 concepts to be relevant. Using the DSL, we only count 4. This indicates that the use of the DSL might simplify the process by requiring less diverse knowledge. Using the DSL hides accidental complexity because the definition of constraints only depends on essential concepts (e.g. knowledge about *Palladio* and the architectural model) and not on transformation artefacts (e.g. knowledge about the *Operation Model* or the transformation trace). However, its hard to quantify knowledge. Thus, we cannot proof this assumption without extensive user studies.

9.3.3. G3 - Space Efficiency

The third goal evaluates the language's space efficiency. We discuss simple analysis goals with question **Q8** and larger analyses with question **Q9**. Last, we compare the DSL's space efficiency compared to constraints formulated in the underlying formalism in question **Q10**.

Question Q8. This question evaluates the space efficiency of simple constraints. Therefore, we discuss the amount of elements in the abstract syntax and the amount of boilerplate code in the concrete syntax of a minimal constraint. This constraint only consists of required elements, namely a `Rule`, a `Statement`, a `DataSelector` and a `DestinationSelector`. We use an exemplary characteristic `type` which consists of the literals `A` and `B`. Listing 9.3 shows the concrete syntax of the minimal constraint.

```
1  constraint MinimalConstraint {
2    data.attribute.type.A NEVER FLOWS node.property.type.B
3  }
```

Listing 9.3: Concrete syntax of a minimal constraint

We require architects to name constraints, e.g. `MinimalConstraint`. The name of a constraint has no effect on the analysis process. However, named constraints are easier to be recognized when reading DSL code from other architects or interpreting mapped analysis results. Next, we discuss the keywords of the abstract syntax, namely `data`, `attribute`, `node`, `property` and `NEVER FLOWS`. These keywords are only examples which we chose in order to close the gap between the concrete syntax of the DSL and constraint

formulated in natural language. Still, they introduce some boilerplate code. `attribute` and `property` are used to determine that the constraint selects Characteristics. `data` and `node` don't serve this purpose and can only be justified by arguing with readability. This is a conscious trade-off. We consider less or shorter keywords would benefit the space efficiency not enough to justify the loss of understandability. In order to fully understand the effect of several keywords, user studies are required.

Next, we discuss the abstract syntax. DataSelectors and DestinationSelectors are well integrated into our language and serve multiple purposes, e.g. selecting Characteristics, classes or nodes by their identity. In the current version of the DSL, Rules and Statements are less important. Rules have to be contained exactly once in every constraint and are implicitly created in the concrete syntax. Thus, all relations on Rules could be moved to Constraints without changing the concrete syntax. We justify the decision of Rules with the language's expandability. More complex constraints might combine multiple rules using propositional logic in one constraint in future versions of the DSL. This also applies to the Statement. In the current version of the DSL, the only possible statement is `NEVER FLOWS`. However, future versions might support more modalities and flow relations. In order to support this evolution, we kept both Rules and Statements as part of the abstract syntax. This does not influence the concrete syntax and is thus not considered to be problematic to architects who write constraints using the DSL.

In summary, multiple non-minimal elements can be identified in both abstract and concrete syntax. These elements are motivated by the language's evolution and readability. It's possible to remove these elements from the abstract syntax and shorten the concrete syntax. However, we don't consider the removal of 4 elements in the abstract syntax and 2 keywords to be significantly enough to the space efficiency in order to justify worse maintainability and worse understandability.

Question Q9. We discuss the verbosity of large constraints and both the vertical composition (adding more elements to an constraint) and the horizontal composition (adding more constraints). First, we define an synthetic constraint which uses most of the provided elements of our DSL. Using all elements (e.g. all provided operations) is not considered to represent a realistic scenario. Thus, we selected one element of each type, e.g. one ClassSelector, one CharacteristicSelector and so on. This enables us to discuss the vertical composition.

```
1  constraint LargeConstraint {
2    data.attribute.type.A & data.class.SampleClass & data.attribute.type.$X
3    NEVER FLOWS node.name."SampleNode" & node.property.type.$Y{}
4    WHERE !elementOf(X,Y)
5  }
```

Listing 9.4: Concrete syntax of a large constraint

Listing 9.4 shows the concrete syntax of our exemplary large constraint. We use an AttributeSelector, a CharacteristicClassSelector and a CharacteristicVariable in line 2. Line 3 defines a NodeIdentitySelector and uses a CharacteristicSetVariable. Both variables are used inside the Condition together with BooleanOperations in line 4. We consider this constraint to be large but still representative for complex analysis goals.

First, we discuss the readability. The different parts of the constraint are divided using the upper-case keywords `NEVER FLOWS` and `WHERE`. Multiple Selectors are divided using the `&` operator. This shall improve the distinction as well as implicate that multiple Selectors evaluated together. The readability benefits from the orthogonality discussed in question **Q5**. Operations in the Condition part of the DSL can be nested or combined with operators known from propositional logic. As discuss previously, no final statement is possible on the language's readability without the execution of user studies. However, because of the orthogonality discussion, it can be assumed that constraints of different sizes remain readable. This indicates good vertical composition.

Second, we consider the horizontal composition of constraints. More complex analysis goals require the definition of multiple constraints. An example is the analysis goal of the *UMLsec Secure Links* scenario presented in Subsection 9.2.1. We show the constraints using our concrete syntax in Listing 9.5. This scenario uses two CharacteristicClasses together with three constraints. All constraints are fairly simple and only consist of one DataSelector and one DestinationSelector.

```
1   class AllSensitivities {
2     Sensitivity.[High,Integrity,Secrecy]
3   }
4
5   class AllLinks {
6     Link.[Encrypted,Internet,LAN]
7   }
8
9   constraint InsiderAttacker {
10    data.class.AllSensitivities NEVER FLOWS node.class.AllLinks
11  }
12
13  constraint DefaultAttackerInternet {
14    data.class.AllSensitivities NEVER FLOWS node.property.Link.Internet
15  }
16
17  constraint DefaultAttackerEncrypted {
18    data.attribute.Sensitivity.High NEVER FLOWS node.property.Link.Encrypted
19  }
```

Listing 9.5: Concrete syntax of the UMLsec Secure Links constraints

Individual classes and constraints are independent by design. CharacteristicClasses can only be referenced using CharacteristicClassSelectors. There is no possibility to reference constraints or to combine them. The orthogonality is preserved. Thus, we propose a good horizontal composition.

Question Q10. We compare the DSL's space efficiency to the space efficiency of constraints formulated in the underlying formalism. Therefore, we gathered several constraints and use change scenarios already discussed in question **Q3**. We use the abstract syntax of the DSL and the Prolog meta-model [40]. Although we don't discuss the concrete syntax in this question, the evaluation of number of elements using abstract

Scenario / Constraint	DSL	Prolog
Minimal constraint from **Q8**	9	101
Geolocation Constraints	9	102
Geolocation Constraints with Classes	10	116
Geolocation Constraints with Identity	12	119
TravelPlanner	18	118
ContactSMSManger	18	118
DistanceTracker	18	118
UMLsec Secure Links	28	205
Large constraint from **Q9**	29	151

Table 9.6.: Number of model elements of scenarios' constraints

syntaxes can be still used as indicator. For the discussion of space efficiency of the concrete syntax, please see questions **Q8** and **Q9**. We collected multiple scenarios and constraints. We formulated the constraints using our DSL and using native Prolog code together with the *Constraint Query API*.

Table 9.6 shows the number of elements of the analyzed constraints. The number of elements used for constraints formulated with our DSL is significantly lower than the number of elements required to formulate the same constraints in Prolog. The range is between 10 time less code (e.g. for the minimal constraint) and 5 times less code (e.g. for the large constraint).

Taking into account which DSL elements are used to formulate the constraints yields additional results. For instance, the gap between both approaches grows when multiple constraints are defined. We analyzed the *UMLsec Secure Links* constraints in question **Q9** when discussing horizontal composition. This is the only evaluated scenario with three independent constraints. Because each constraint requires a separate clause in Prolog, the number of model elements grows.

The opposite can be observed regarding the vertical composition with the synthetic, large constraint. Multiple combined selectors require many DSL elements. Prolog benefits from the reuse of predicates and thus scales better. This indicates a better horizontal composition than vertical composition. However, the results provide not enough data for a final judgement.

In order to gain additional insights whether or not the usage of selected DSL elements influences the ratio between the number of model elements, we analyze selected change scenarios from question **Q3**. We measure the change in the number of model elements for change scenarios **S11** to **S15** which add different elements to an existing constraint.

The result is shown in Table 9.7. As shown above, the DSL requires significantly less elements than constraints formulated using Prolog. The ratio between both approaches remains nearly the same for the most of the change scenarios. The only outlier is the addition of a Condition with one Operation in change scenario **S15**. Due to the existence of many useful predicates in the `list`-library of Prolog [44], Conditions can be expressed very space efficiently in Prolog. Additionally, our DSL uses CharacteristicReferences in order to decouple Selectors and Conditions which adds extra elements. This is also indicated by the

#	**Change Scenario**	DSL	Prolog
11	Literal added	0	8
12	Data selector added	2	8
13	Destination selector added	2	8
14	Class and selector added	2	10
15	Condition added	4	4

Table 9.7.: Change in number of elements in change scenarios

scenarios which use Conditions in Table 9.6, namely *TravelPlanner*, *ContactSMSManger* and *DistanceTracker*. Here, the number of elements when using the DSL is higher compared to the *Geolocation Constraints* examples while the number of Prolog elements remain consistent. We discussed this desgin decision together with the language's orthogonality in question **Q5**.

In summary, the space efficiency of constraints defined in our DSL is better compared to constraints formulated with Prolog code using the *Constraint Query API*. The efficiency of different concepts of the DSL differs: Some elements like Selectors are very space efficient while other elements like Operations can be reproduced in Prolog with similar space efficiency. The overall ratio is between 5 to 10 times less elements when using the DSL.

9.3.4. G4 - Equivalence of Analysis

The fourth goal evaluates the equivalence of the analysis using the DSL compared to the underlying formalism. In question **Q11**, we evaluate the correctness of the mapping between the DSL and Prolog code which uses the *Constraint Query API*. Afterwards, we discuss concrete analysis results from constraints formulated both using the DSL and the underlying formalism in question **Q12**.

Questions Q11. We perform a correctness proof to validate the preservation of semantics of our DSL mapping. This approach is motivated by the domain of compiler construction [30, 45, 11]. Due to the effort of such formal approaches, we only consider the core part of our DSL which consists of Constraints, Rules, DataSelectors, DestinationSelectors and CharacteristicClasses. Additionally, we only consider well-formed constraints formulated in our DSL because the DSL mapping is only defined for well-formed constraints. The abstract syntax shows clear restrictions whether or not constraints are well-formed, e.g. by defined multiplicity. The highest degree of freedom is achieved using the Condition part where arbitrary operations can be nested and combined using propositional logic. Thus, we append a short discussion of the well-formedness of Conditions using the lambda calculus in Section A.1.

In the following, we show excerpts of the formalization of our DSL and the DSL mapping prior to performing the structural induction. The complete definitions can be found in Section A.2. According to the structure of this proof presented in Subsection 9.2.5, this discussion is structured as follows: First, we formalize the source context C, the source language L, the target context K and the target language T using universal algebra. Then,

we formalize the transformation between the languages γ and the source semantics θ and target semantics ψ. Based on this information, we conduct the structural induction which consists of a proposition, a basis and the induction step.

We start by defining the source context C. This contains concepts which we use from *Data-Centric Palladio*: Characteristics, literals and elements of the architectural model. We define the domains by enumerating all concepts in a structure Υ_C as proposed by Jackson et al. [16]. Using universal algebra, this can be denoted as follows:

$$\Upsilon_C = \begin{cases} \mathsf{CharacteristicType(C)}: & C \textit{ is the name of a characteristic type} \\ \mathsf{CharacteristicLiteral(L,C)}: & L \textit{ is the name of a literal of Characteristic type } C \\ \mathsf{ArchitecturalElement(N,L)}: & N \textit{ is the name of an architectural elem. with literal } L \end{cases}$$

A realization $r \in R_{\Upsilon_C}$ of this context can represent a concrete architectural element. For instance, an online shop deployed inside the EU (see Section 4.1 for the complete example) can be denoted as follows:

$$\mathsf{ArchitecturalElement}(\textit{OnlineShop}, \mathsf{CharacteristicLiteral}(\textit{EU}, \mathsf{CharacteristicType}(\textit{Location})))$$

This enables us to define arbitrary architectural elements with characteristics and literals as it's done by the architectural model of *Data-Centric Palladio*. We use this structure Υ_C of the source context domain to formalize the source language L which represents our DSL. We present an excerpt of the structure Υ_L:

$$\Upsilon_L = \begin{cases} \mathsf{CharacteristicTypeSelectorSingle(C,L)}: & C \textit{ is a characteristic type, } L \textit{ one of its literals} \\ \mathsf{AttributeSelector(C)}: & C \textit{ is a CharacteristicTypeSelector}^* \\ \mathsf{Rule(D,Z)}: & D \textit{ is the rule's DataSelector}^*, Z \textit{ its DestinationSelector}^* \\ \mathsf{Constraint(N,R)}: & N \textit{ is the name of a constraint, } R \textit{ is its rule} \end{cases}$$

We marked several elements with a star, namely DataSelector, DestinationSelector and CharacteristicTypeSelector. As implied by the meta-model, these can be replaced by their sub types using polymorphisms. For instance, every AttributeSelector is also a DataSelector. In order to correctly formalize this relation, multiple entries of a structure Υ can be denoted. To enhance the readability, we choose to highlight these elements and refer to our abstract syntax for more information on inheritance relations. For the complete formalization of our core DSL, please refer to Section A.2.

Similar to the formalization of the source context C and the source language L, we formalize excerpts from the target context K and the target language T. The target context K represents both Prolog predicates and concepts of the *Operation Model*. We denote an excerpt from the structure Υ_K:

$$\Upsilon_K = \begin{cases} \text{ValueSetType(T)}: & \textit{T is the name of a ValueSetType} \\ \text{AttributeType(A,T)}: & \textit{A is the name of an attribute type, T its ValueSetType} \\ \text{Term(V)}: & \textit{V is a Prolog term} \\ \text{Clause(H,B)}: & \textit{H is the clauses head, B its body; both are terms} \\ \text{Unification(L,R)}: & \textit{Unification of L and R; both are terms} \\ \text{NotProvable(P)}: & \textit{Invert the statement of P; P is a term} \\ \text{And(L,R)}: & \textit{Combine the statements of L and R; both are terms} \end{cases}$$

ValueSetTypes and AttributeTypes are elements from the *Operation model* and represent characteristics [25]. The other entires are syntactical elements of Prolog or common predicates. The shown structure Υ_K is not meant to be comprehensive; it only shows how Prolog elements can be formalized using universal algebra.

Accordingly, we present an excerpt of the structure Υ_T which consists of elements of the *Constraint Query API*. We discussed this API in Section 6.2. Two exemplary elements are:

$$\Upsilon_T = \begin{cases} \text{StackValid(S)}: & \textit{S is the name of a stack which represents a valid call sequence} \\ \text{CallArgument(S,P,A,V)}: & \textit{S is a call stack, P a parameter name, A an} \\ & \qquad \textit{AttributeType, V a value of this AttributeType's ValueSetType} \end{cases}$$

Based on these four domains, Υ_C, Υ_L, Υ_K and Υ_T the transformations π and γ can be defined. The transformation π maps the source context C to the target context K and is out out of scope of this work. It has already been informally defined by Kunz [25]. We only consider the transformation γ which maps from the source language L to the target language T. This transformation is defined as: $\gamma = \langle \Upsilon_L, \Upsilon_T, \tau \rangle$. In order to discuss the mapping, we give examples for the transformation rules τ:

$$\tau = \begin{cases} \text{AttributeSelector}(\text{CharacteristicTypeSelectorSingle}(C, L)) \\ \qquad \longrightarrow \text{CallArgument}(\text{Term}(S), \text{Term}(P), C, L) \\ \text{Constraint}(N, \text{Rule}(D, Z)) \longrightarrow \text{Clause}(\text{Term}(N, O, S, P), \\ \qquad \text{And}(\text{Unification}(S, \text{List}(O)), \text{And}(\text{StackValid}(S), \text{And}(D, Z)))) \end{cases}$$

The first rule maps an AttributeSelector to a term using the CallArgument predicate. The CharacteristicTypeSelector is also specified, because a well-formed AttributeSelector always contains one. The result uses the CallArgument predicate with two terms S and P and the mapped Characteristic C and L. This mapping is part of π and thus not considered here. Please note that we denote the mapped elements C and L using the same symbols as their original based on the work of Jackson et al. [16]. Whenever a symbol is reused, its mapping is realized by another transformation rule which is part of the language transformation γ or the context transformation π.

The second rule transforms constraints. The result is a Prolog clause which consists of the creation of a list variable S using the Unification, its validity check using the StackValid

predicate and the combination of the DataSelector D and the DestinationSelector Z. The mapping of these elements is realized using other rules of τ. Although this mapping result uses universal algebra, its structure resembles concrete Prolog code as discussed in Listing 8.4.

The last step prior to the structural induction is the definition of the source semantics θ and target semantics ψ. We show an excerpt from the mapping val_θ which maps domain elements to the boolean values *true* or *false*:

$$val_\theta(\mathsf{AttributeSelector}(\mathsf{CharacteristicTypeSelectorSingle}(C, L))) = \begin{cases} \textit{true}, & \textit{if an ArchitecturalElement with a data flow with literal L of} \\ & \textit{characteristic type C is selected} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

$$val_\theta(\mathsf{Constraint}(N, \mathsf{Rule}(D, Z))) = \begin{cases} \textit{true}, & \textit{if a selection of both D and Z is true for any valid data flow} \\ & \textit{between any ArchitecturalElements} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

The definition of semantics θ of the source language L is also based on the source context C, e.g. by referring to ArchitecturalElements or Characteristics. Additionally, we show an excerpt from the mapping val_ψ which defines the target semantics ψ. These semantics are based on semantics defined in natural language by Kunz [25]:

$$val_\psi(\mathsf{StackValid}(S)) = \begin{cases} \textit{true}, & \textit{if the list S represents a correct call sequence} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

$$val_\psi(\mathsf{CallArgument}(S, P, A, V)) = \begin{cases} \textit{true}, & \textit{if the Value V of the Attribute A of the parameter P is present} \\ & \textit{given the call stack S. The operation which owns the} \\ & \textit{Parameter P is defined by the stack top of} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

Based on the formalization discussed above, we conduct the structural induction. We defined the proposition of this induction in Subsection 9.2.5: For every possible model realization r, its meanings defined by the source semantics θ shall imply the meanings of the transformation result of the realization. Formally:

$$\forall r \in R_{\Upsilon_{DSL}} : val_\theta(r) \implies val_\psi([\![r]\!]^\Upsilon)$$

In the induction basis, we discuss the implication of meanings of the atomic cases discussed above. Based on the well-formedness, every element in a constraint can rely on

the context which is implied by the surrounding constraint. In the context of the target language T, we assume the existence of a stack S with an operation O on top of which represents the currently selected element. This operation has a parameter P. O is the transformation result of the ArchitecturalElement which is derived using the mapping of contexts π. We show the induction basis for two examples. First, the transformation of AttributeSelectors to terms using the CallArgument predicate:

Source syntax: AttributeSelector(CharacteristicTypeSelectorSingle(C, L))
Target syntax: CallArgument(Term(S), Term(P), C, L)
Source semantics: *true, if an ArchitecturalElement with a data flow with literal L of characteristic type C is selected*
Target semantics: *true, if the Value L of the Attribute C of the parameter P is present given the call stack S. The operation which owns the Parameter P is defined by the stack top of*

The *data flow with literal L of characteristic type C* is represented by the parameter P of operation O in call stack S. This is ensured by the context and its transformation π. The selection of such an ArchitecturalElement implies its existence on top of the call stack. The parameter P has an attribute C with value L because its mapped from the original ArchitecturalElement and its characteristic. Thus, the implication holds.

The second example is the transformation of complete constraints:

Source syntax: Constraint(N, Rule(D, Z))
Target syntax: Clause(Term(N, O, S, P),
And(Unification(S, List(O)), And(StackValid(S), And(D, Z))))
Source semantics: *true, if a selection of both D and Z is true for any valid data flow between any ArchitecturalElements*
Target semantics: *true, if at least one call stack S satisfies both D and Z. The operation O is hereby the head of the call stack S.*

Any valid data flow is represented by the call stack S in combination with the StackValid predicate. This call stack provides the operation O and parameter P which is the element under consideration by both selectors. Both selectors D and Z in the source languages imply the transformed selectors in the target languages. This is ensured by other parts of the induction basis as disussed above for AttributeSelectors. In summary, the selected element of the target language is the transformed element of the source language. This is ensured by the context transformation π and the transformed selectors meanings are implied by the originating selectors. Thus, the implication holds.

In the induction step, we show that the proposition holds for more complex examples which rely on the induction basis. This includes the discussion of CharacteristicTypeSelectors with multiple literals, the use of more than one DataSelector or DestinationSelector and

Constraint	Equivalent	Missing	Additional
Geolocation without Classes	1	0	0
Geolocation with Classes	1	0	0
Geolocation with Encryption	1	0	0
TravelPlanner	0	0	3
TravelPlanner (Inv.)	6	0	0
UMLsec Secure Links	10	0	0

Table 9.8.: Number of found violations of different constraints

the transformation of DataSelectors to terms using different predicates than CallArgument, e.g. ReturnValue or PreCallState. For the sake of brevity, we only present an informal discussion.

CharacteristicTypeSelectors with more than one literal are mapped by γ to multiple terms with the same predicate, e.g. to multiple CallArguments which are than combined using Prolog's Or predicate. The source semantics θ evaluate to *true* if at least one of the literals is present. The target semantics ψ of Prolog's Or predicate evaluate to *true* if at least one of the selection predicates evaluates to true. This implication holds. The discussion of multiple DataSelectors and DestinationSelectors is similar. The only difference is that all selectors must evaluate to *true*. Thus, γ uses Prolog's And predicate. The discussion of different data flow related predicates like ReturnValue is based on the definition of data flows: In order to consider each type of flow such as ingoing and outgoing data of an operation, all predicates are generated implicitly. The use of call states extends mapped constraints by another variable ST but does not change alter the underlying meanings.

In summary, we formalized parts of our language using universal algebra and conducted a correctness proof using structural induction. The result shows that the semantics are preserved by the transformation which is performed by the DSL mapping. In this section, we did only explain few examples for the sake of brevity. A more comprehensive excerpt can be found in Section A.2.

Question Q12. This question discusses the equivalence of analysis results. We gathered formulated constraints which use the *Constraints Query API* from other work [25, 31] and reformulated them using our DSL. This enables us to evaluate whether or not the analysis results are identical. We consider an analysis result identical if the same constraint violations are yield by the Prolog solving process. We hereby ignore the ordering of results.

We included three different constraints from the *Geolocation Constraints* scenario. We also included the constraint from the *TravelPlanner* case study. Here, we encountered an additional inverted constraint which tests for permitted accesses. Last, we included the *UMLsec Secure Links* analysis which originates from the additional constraints presented in Subsection 9.2.1. Table 9.8 shows the results of this evaluation. Most of the resulting constraint violations are found by both the original Prolog constraints and the reformulated constraints. Using our DSL, the analysis did not miss any violations. However, the analysis of the *TravelPlanner* case study yields three additional violations with the use of our DSL. We investigated the reason of this problem and identified that the DSL mapping does not consider two predicates of the *Constraint Query API* in this context. The use of the

`operationParameterType` and the `dataTypeAttribute` predicate would have avoided the *false-positive* violations. By altering the DSL mapping, this problem can be solved in future versions without changing the DSL's syntax.

Based on the metrics of binary classifications [36], we calculate the precision and recall:

$$Precision = \frac{19}{19 + 3} = 0.8\overline{63} \qquad Recall = \frac{19}{19 + 0} = 1$$

We consider a 100% recall which maintains a precision of approximately 90% as good result. The remaining *false-positive* results originate from the generalization step prior to the development of the DSL and thus didn't become visible throughout the correctness proof which only analyzes the realized DSL mapping. By fixing this problem, we would approach the equivalence of analysis results. However, strictly speaking this assumption is only valid for the selected constraints and the analyzed part of our DSL.

9.3.5. Summary

We close the discussion of the results with a brief summary of our findings. We evaluated the DSL's expressiveness (**G1**), usability (**G2**) and space efficiency (**G3**). Additionally, we evaluated the equivalence of the analysis (**G4**) of our DSL compared to the underlying formalism.

Goal **G1** showed overall good expressiveness. In questions **Q1**, we evaluated whether collected scenarios can be expressed using our DSL. 6 out of 8 scenarios or 75% are expressible. Additionally, we analyzed the coverage of the underlying formalism namely the *Constraint Query API*. Approximately 80% of this API can be expressed either explicitly or implicitly. This result is supported by the evaluation of met requirements which we collected in Chapter 6: 11 of 13 requirements have been realized throughout this thesis. So sum up, the combination of this results indicates good expressiveness. However, there is still room for a more detailed evaluation, e.g. by collecting more real-world scenarios.

With goal **G2**, we evaluated the languages usability. First, we detected a small reduction in change effort compared to the underlying formalism (**Q3**). We analyzed whether different language concepts are used which is true for most of them. This yields a good degree of conciseness (**Q4**). We evaluated the languages orthogonality (**Q5**). Due to the structure of constraints which splits up the goal formulation in mostly independent parts, we achieved lower coupling and high orthogonality. In question **Q6**, we discussed the uniqueness of language features. One threat to the uniqueness is the concept of `CharacteristicClasses` which can be justified as trade-off decision between expressiveness and usability. Last, we discussed which knowledge is required in order to use the DSL compared to Prolog code (**Q7**). Besides common knowledge on data flow modeling, the DSL does not require any knowledge about the transformation or the underlying formalism. Moreover, the knowledge required to use our DSL is considered to be less diverse.

Goal **G3** evaluates the language's space efficiency. We considered small analyses be defining a minimal constraint in question **Q8**. We identified multiple elements of the abstract syntax which could be reduced but support the language's evolution. Additionally, they don't violate the space efficiency of the concrete syntax. We evaluated the horizontal and vertical composition of larger constraints in question **Q9**. The overall level of verbosity

of our DSL is acceptable. Last, we compared the space efficiency of our DSL with the underlying formalism (**Q10**). The space efficiency of our DSL is better than using the underlying formalism. We measured ratios between 5 and 10 times more required metamodel elements when using the underlying formalism.

Last, we evaluated the equivalence of the analysis with goal **G4**. In order to validate the transformation, we conducted a correctness proof motivated by the domain of compiler construction (**Q11**). We formalized the core of our DSL and performed a structural induction in order to show the correctness of our DSL mapping. Last, we evaluated concrete results of constraints formulated in the underlying formalism compared to reformulated constraints in the DSL (**Q12**). We compared the analysis results and mostly found equivalent analysis results. This yields 100% recall while maintaining 90% correctness due to several *false-positive* violations.

9.4. Threats to Validity

We discuss the threats to the validity of our evaluation. The validity indicates the "trustworthiness of the results, to what extent the results are true and not biased by the researchers' subjective point of view" [39]. We use the scheme proposed by Runeson et al. [39] and discuss internal, external and construct validity as well as reliability separately.

Threats to the internal validity state whether the evaluation result only depends on the investigated factor without interferences of a third factor. Overlooking or ignoring additional influences to the result violate the strength of the conclusion. For every unconsidered confounding variable, competing hypotheses arise. In order to evaluate the language's expressiveness, we formulated existing analysis goals using our DSL (**Q1**). We were not able to formulate 25% of the constraints. This might depend on the language's expressiveness or the author's formulation abilities which represents a threat to the internal validity. However, since the author created the DSL in the first place the probability of a lack of formulation abilities is negligible.

A similar threat occurs whenever we use handwritten Prolog code. This code is used to compare the DSL's abstraction (**Q3**) or it's space efficiency (**Q10**) compared to underlying formalism. The Prolog code is optimized in order to create a fair comparison. Nevertheless, the author's experience of using Prolog is limited. An more experienced Prolog developer might have written better code. However, especially the results of question **Q10** are evident which avoids this threat.

Due to the importance of the equivalence of results, we conducted a correctness proof of our mapping (**Q11**). Prior to this proof, the author had near to no experience in correctness proofs. We tried to minimize this threat by discussing our approach with more experienced researches. However, there is still a risk of faults inside the formalization and structural induction.

Threats to the external validity impact the generalization of our evaluation results. In order to evaluate these threats, we discuss whether or not selected samples are representative for the population. The first threat occurs in question **Q3**. Here, we discussed the abstraction of our DSL using a set of 15 change scenarios. This selection is not based on a common approach to gather change scenarios of existing models and thus at risk to

not being comprehensive enough. The authors were not able to find a common basis for these scenarios in other work and thus selected a to their knowledge comprehensive set of change scenarios. This discussion also applies to our selection of knowledge domains in question **Q7**. There is no comprehensive model available to identify these domains. However, this selection is based on the foundations of this work and considers the surrounding process of *Data-Centric Palladio* and its required knowledge which minimizes the risk of missing concepts. Additionally, the discussion does not quantify knowledge and only shows a trend in the required knowledge. Thus, the results are less vulnerable to single missing domains.

The biggest threat to external validity is the selection of scenarios and constraints which were used to gather requirements as well as to evaluate the DSL. It cannot be assumed that our selection of 8 scenarios is representative for the domain of data-flow modeling. Strictly speaking, our results are only valid for the selected scenarios. In order to cope with this uncertainty, we tried to generalize the DSL's concepts whenever possible. For instance, we included 11 operations which can be used in the `Condition` part of our DSL although the constraints under consideration only required a subset of them. We discussed the comprehensibility of this selection in question **Q2**. Still, this threat affects the DSL and all questions which depend on the selection of constraints, e.g. the discussion of expressiveness (**Q1**) and conciseness (**Q4**) as well as the equivalence of analysis results (**Q12**).

Threats to the construct validity state whether our selection of questions and measurements contribute to the overarching goals. To minimize this threat, we included the results of other work on the development of DSLs [1, 20, 28, 33, 46] into our process. The usage of a *Goal-Question-Metric* plan does also mitigate this threat. We consider our plan to be sufficient to evaluate our approach. Still, some attributes cannot be evaluated without considering the user's behavior. Due to the high effort, we were not able to perform user studies in the scope of this thesis.

Reliability deals with the repeatability of the evaluation and asks "to what extent the data and the analysis are dependent on the specific researchers" [39]. In order to aid the repeatability, we defined the evaluation design in Section 9.2 prior to the discussion of the results. Additionally, we publish our prototypical implementation along with models and constraints created throughout this thesis [14] to allow others to reproduce the evaluation.

9.5. Assumptions and Limitations

In this section, we discuss assumptions we made during the design and realization or our DSL. This includes analyzing the limitations of the current version of our approach.

First, we assume our selection of scenarios and constraints to be sufficiently representative for the domain of data flow modeling. We already discussed this concern together with the external validity in Section 9.4. This also influences the design of our DSL. We tried to generalize our approach but still depend on the scenarios we found during the initial research.

Another assumption affects the analysis of data flows using the *Constraint Query API* of *Data-Centric Palladio*. The *Operation Model* uses call arguments, return values and call

states to represent data flowing in or out of operations. In order to cover all possible flows, our DSL Mapping combines all predicates: `CallArgument`, `ReturnValue`, `PreCallState` and `PostCallState`. Here, two problems occur. In theory, it's possible to identify a constraint violation in ingoing data as call argument as well as in outgoing data as return value. This would yield two constraint violations which belong to the same problem. Furthermore, it depends on the architectural model whether these different types are used. For instance, if a transformed architecture does not use call arguments and return values at all, the analysis becomes unnecessary and might even yield execution errors if these predicates are not generated at all. This could be solved by coupling the transformation of the DSL and the architectural model. For the initial design, we ignored this detail and always considered all four data flow related predicates as discussed above.

The first limitation is related to *Palladio*. Using NodeIdentitySelectors, architects can select elements from the architectural model by specifying the assembly context, a component and a *SEFF*. However, this integration is not complete. For instance, nested assembly contexts are not supported.

The second limitation occurs from the strcuture of constraints in our DSL. We defined a fixed structure of the selection of data and data flow destinations. Constraints which do not match to this structure cannot be expressed. This is related to the third limitation: Because we did not meet requirement **R9**, our DSL is not capable of considering multiple data flows in one constraint.

9.6. Data Availability

The data of this thesis is publicly available [14]. This includes the prototypical implementation of our DSL, its mapping and the result mapping along with models and constraints collected and created during this thesis.

10. Conclusion

To conclude this thesis, we summarize our approach and findings in Section 10.1. We give an outlook on future work in Section 10.2 and express our acknowledgements in Section 10.3.

10.1. Summary

The goal of this thesis was to provide an approach to formulate analysis goals for data-driven design time analyses. In order to abstract from the underlying formalism, we proposed the use of a *domain-specific language (DSL)* which uses the terminology known from the domain of data flow modeling. We provided a mapping between the DSL and the formalism which is used for the analysis. In order to ease the interpretation of analysis results, we defined a mapping of results back into the architectural domain. Our research questions targeted the collection of architecture-level constructs to define data flow constraints and the mapping of constraints and results between the architecture and the formalism.

Our DSL is based on the collection of data flow scenarios and constraints from other work. We derived requirements from real-world use cases and combined this information with analysis capabilities of the underlying formalism which is given by *Data-Centric Palladio*. We generalized the analysis goals of multiple scenarios and defined the syntax of our DSL. Using our DSL, architects are able to describe data flow constraints by specifying data and data flow destinations using characteristics. The expressiveness of the selection is enhanced by the definition of conditions on variable characteristics as well as the definition of characteristic patterns using characteristic classes.

We defined a mapping from our DSL to the underlying formalism. This transformation generates executable Prolog code which uses the Prolog API provided by *Data-Centric Palladio*. Together with the existing transformation of architectural models to Prolog code, our mapped constraints can be evaluated by the Prolog environment. Afterwards, our result mapping transforms identified constraint violations back into the architectural domain and presents them to the architect.

Our approach enables architects to specify data flow constraints while modeling the architecture without the need of switching the abstraction level. The transformation to and from the underlying formalism is fully automated. Thus, architects are not required to know how to use the underlying formalism; only basic knowledge on data flow modeling, *Data-Centric Palladio* and our DSL is required.

We evaluated our approach regarding the DSL's expressiveness, usability and space efficiency. In order to evaluate the expressiveness, we collected scenarios and case studies from other work and reformulated the constraints. 75% of all considered constraints can be

expressed using our DSL which matches the coverage of the API of *Data-Centric Palladio* which is 80%. Regarding the DSL's usability, we considered the abstraction from the underlying formalism, the language's conciseness, orthogonality and the uniqueness of features. Overall, the results look promising: our DSL requires less effort to define and alter data flow constraints than the underlying formalism. All concerns regarding other language quality attributes originate from trade-off decisions regarding its expressiveness. The evaluation of the DSL's space efficiency indicates that the DSL requires between 5 to 10 times less code than similar constraints defined using the underlying formalism.

Additionally, we evaluated the equivalence of the analysis using our DSL to define constraints compared to the underlying formalism. We performed a correctness proof of the mapping of the core DSL. We also evaluated whether the analysis results of constraints defined in our DSL match the results of already formulated constraints in Prolog. The results show a recall of 100% while maintaining 90% precision.

This thesis aimed to improve the understanding of the definition of analysis goals for design time analysis which bridge the gap between the architectural domain and the underlying formalism. We developed and realized a DSL and its mapping which improves the architects capabilities by removing accidental complexity. We also discussed requirements which have to be considered in order to enable the mapping of analysis results.

10.2. Future Work

The approach presented in this thesis is ready to be used in its current form. However, we discovered several parts which can be enhanced and are thus subject to future work.

First, we identified several smaller possible improvements of the current version of the DSL. Currently, constraints always consist of exactly one rule which restricts the flow of data to one destination. Here, supporting multiple rules which could be combined using propositional logic would enhance the architects capabilities. Another placeholder element of our DSL is the statement which currently always restricts data to *never flow*. Here, more modalities would be possible, e.g. requiring data to *always flow* or *flow at least once*. This discussion is based on modal logic and the underlying formalism is already capable of analyzing constraints of such nature.

Moreover, the DSL could be extended to support types of constraints which don't follow the pattern of selecting data which flows to a selected destination. An example is the consideration of multiple data flows which e.g. are not allowed to be joined in a selected node. This requirement is motivated by a real-world scenario analyzed in this thesis. Another type of constraint is the specification of data without considering its destination. For instance, requiring certain characteristics to never appear together in a datum during the analysis.

In the current version of the DSL, architects can choose the target model type. Currently supported are constraints based on *Data-Centric Palladio* or its underlying *Operation Model*. However, the genericness of the DSL might suffice to support more than these two data flow models. Further research is required to collect more data flow modeling environments and generalize their analysis goal formulation.

Last, the quality of the existing DSL could be enhanced by executing a user study. This would not only help to improve on the concrete syntax and usability of the language but also yield empirical results whether the usage of an abstraction approach like a DSL improves the architects capabilities and performance while defining data flow constraints.

10.3. Acknowledgements

I want to thank my advisors Stephan Seifermann and Frederik Reiche for their support which greatly improved the quality of this thesis. I also want to thank Dr.-Ing. Thomas Kühn for his advice on the realization of formal correctness proofs. Last, I want to thank all participants of the breakout group "Evaluation of DSLs" during the research retreat in march 2020 for their great input.

Bibliography

[1] Marcel van Amstel, Mark van den Brand, and Luc Engelen. "An exercise in iterative domain-specific language design". In: *Proceedings of the joint ERCIM workshop on software evolution (EVOL) and international workshop on principles of software evolution (IWPSE)*. ACM. 2010, pp. 48–57.

[2] Victor R Basili and David M Weiss. "A methodology for collecting valid software engineering data". In: *IEEE Transactions on software engineering* 6 (1984), pp. 728–738.

[3] Max Bramer. *Logic programming with Prolog*. Springer, 2005.

[4] F Brooks and HJ Kugler. *No silver bullet*. 1987.

[5] Theodor G Bucher. *Einführung in die angewandte Logik*. Vol. 2231. Walter de Gruyter, 1998.

[6] Jordi Cabot, Robert Clarisó, and Daniel Riera. "On the verification of UML/OCL class diagrams using constraint programming". In: *Journal of Systems and Software* 93 (2014), pp. 1–23.

[7] Victor R Basili Gianluigi Caldiera and H Dieter Rombach. "The goal question metric approach". In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[8] Luca Cardelli. "Type systems". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 263–264.

[9] Tom DeMarco. "Structured Analysis and System Specifications, rentice-Hall". In: (1979).

[10] Hans M Dietz. *Mathematik für Wirtschaftswissenschaftler*. Springer, 2019.

[11] Peter Dybjer. "Using domain algebras to prove the correctness of a compiler". In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 1985, pp. 98–108.

[12] Eclipse Foundation. *Xtext - Language Engineering Made Easy!* 2020. URL: `https://www.eclipse.org/Xtext/` (visited on 07/27/2020).

[13] Sebastian Hahner. "Confidentiality Formalisms for Design Time Security Analyses". 2019.

[14] Sebastian Hahner. *Domain-specific Language for Data-driven Design Time Analyses and Result Mappings for Logic Programs - Data Set*. Aug. 2020. DOI: `10.5281/zenodo.3973100`. URL: `https://doi.org/10.5281/zenodo.3973100`.

[15] Xudong He et al. "Formally analyzing software architectural specifications using SAM". In: *Journal of Systems and Software* 71.1-2 (2004), pp. 11–29.

[16] Ethan Jackson and Janos Sztipanovits. "Formalizing the structural semantics of domain-specific modeling languages". In: *Software & Systems Modeling* 8.4 (2009), pp. 451–478.

[17] Jan Juerjens. "Principles for secure systems design". PhD thesis. University of Oxford, 2002.

[18] Jan Jürjens and Pasha Shabalin. "Automated verification of UMLsec models for security requirements". In: *International Conference on the Unified Modeling Language*. Springer. 2004, pp. 365–379.

[19] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[20] Gabor Karsai et al. "Design guidelines for domain specific languages". In: *arXiv preprint arXiv:1409.2378* (2014).

[21] Kuzman Katkalov. "Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme". doctoralthesis. Universität Augsburg, 2017.

[22] Kuzman Katkalov. *Modeling the Travel Planner Application with IFlow*. 2013. URL: `https://kiv.isse.de/projects/iflow/TravelPlannerSite/index.html` (visited on 06/20/2020).

[23] Kuzman Katkalov et al. "Model-driven development of information flow-secure systems with IFlow". In: *2013 International Conference on Social Computing*. IEEE. 2013, pp. 51–56.

[24] Heiko Koziolek and Ralf Reussner. "A model transformation from the palladio component model to layered queueing networks". In: *SPEC International Performance Evaluation Workshop*. Springer. 2008, pp. 58–78.

[25] Jonas Kunz. "Efficient Data Flow Constraint Analysis". MA thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2018.

[26] Philipp Meier. "Automated Transformation of Palladio Component Models to Queueing Petri Nets". FZI Prize "Best Diploma Thesis". MA thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2010.

[27] Philipp Meier, Samuel Kounev, and Heiko Koziolek. "Automated transformation of component-based software architecture models to queueing petri nets". In: *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE. 2011, pp. 339–348.

[28] Marjan Mernik, Jan Heering, and Anthony M Sloane. "When and how to develop domain-specific languages". In: *ACM computing surveys (CSUR)* 37.4 (2005), pp. 316–344.

[29] Parastoo Mohagheghi and Øystein Haugen. "Evaluating domain-specific modelling solutions". In: *International Conference on Conceptual Modeling*. Springer. 2010, pp. 212–221.

[30] F Lockwood Morris. "Advice on structuring compilers and proving them correct". In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1973, pp. 144–152.

[31] Philip Mueller. “Abbildung von UMLsec-Vertraulichkeitsanalysen auf Data-Centric Palladio”. MA thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2019.

[32] Anantha Narayanan and Gabor Karsai. “Towards verifying model transformations”. In: *Electronic Notes in Theoretical Computer Science* 211 (2008), pp. 191–200.

[33] Richard F. Paige, Jonathan S. Ostroff, and Phillip J Brooke. “Principles for modeling language design”. In: *Information and Software Technology* 42.10 (2000), pp. 665–675.

[34] Sven Peldszus et al. “Secure Data-Flow Compliance Checks between Models and Code based on Automated Mappings”. In: Oct. 2019.

[35] Benjamin C Pierce. *Type Systems and Programming Languages*. 2002.

[36] David Martin Powers. “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation”. In: (2011).

[37] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press, 2016. ISBN: 026203476X, 9780262034760.

[38] Arnaud Roques. *Drawing UML with PlantUML*. 2019. URL: `http://plantuml.com/en/guide` (visited on 11/18/2019).

[39] Per Runeson and Martin Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical software engineering* 14.2 (2009), p. 131.

[40] Stephan Seifermann. *Palladio Supporting Prolog*. 2020. URL: `https://github.com/Trust40-Project/Palladio-Supporting-Prolog` (visited on 07/03/2020).

[41] Stephan Seifermann, Robert Heinrich, and Ralf H. Reussner. “Data-Driven Software Architecture for Analyzing Confidentiality”. In: *IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019*. IEEE, 2019, pp. 1–10. DOI: `10.1109/ICSA.2019.00009`. URL: `https://doi.org/10.1109/ICSA.2019.00009`.

[42] Mark Strembeck and Uwe Zdun. “An approach for the systematic development of domain-specific languages”. In: *Software: Practice and Experience* 39.15 (2009), pp. 1253–1292.

[43] SWI-Prolog. *SWI-Prolog Manual: Glossary*. 2020. URL: `https://www.swi-prolog.org/pldoc/man?section=glossary` (visited on 06/16/2020).

[44] SWI-Prolog. *SWI-Prolog Manual: library(lists)*. 2020. URL: `https://www.swi-prolog.org/pldoc/man?section=lists` (visited on 06/01/2020).

[45] James W Thatcher, Eric G Wagner, and Jesse B Wright. “More on advice on structuring compilers and proving them correct”. In: *Theoretical Computer Science* 15.3 (1981), pp. 223–249.

[46] Arie Van Deursen, Paul Klint, and Joost Visser. “Domain-specific languages: An annotated bibliography”. In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36.

[47] Philipp Weimann. “Automated Cloud-to-Cloud Migration of Distributed So ware Systems for Privacy Compliance”. MA thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2017.

[48] William Wernick. “Complete sets of logical functions”. In: *Transactions of the American Mathematical Society* 51.1 (1942), pp. 117–132.

[49] Huiqun Yu et al. “Formal Software Architecture Design of Secure Distributed Systems.” In: Jan. 2003, pp. 450–457.

A. Appendix

In the appendix, we present additional information which completes this thesis. In Section A.1, we discuss the well-formedness of Conditions, a part of constraints in our DSL. We show a more comprehensive formalization of the parts of the correctness proof in Section A.2. In Section A.3, we show complete examples of generated Prolog using our DSL mapping. Last, we present the complete meta-model in Section A.4.

A.1. Well-formedness of Conditions

The well-formedness of constraints formulated in our DSL is ensured by the abstract syntax to a large extent. The highest degree of freedom exists in the Condition part, where architects can freely nest and combine operations on characteristic variables (see Section 7.4 for all details). We chose a strictly functional approach without side effects or states. In order to discuss the well-formedness of Conditions, we formulate these operations using the lambda calculus. This discussion is part of the appendix because its not considered to match the evaluation but instead provides a different view on the definition of Conditions.

Listing A.1 shows several examples of Conditions. For instance the test for emptiness and the intersection of two set variables in line 3 or the comparison of variables in line 4. Line 5 shows how these operations can be nested in order to formulate more complex Conditions.

```
1   // A, B, C, D are characteristic variables
2   // X, Y, Z are characteristic set variables
3   isEmpty(intersection(X,Y))
4   A == B & C == D
5   elementOf(A, union(Y,intersection(X,Z)))
```

Listing A.1: Examples for conditions which consist of multiple operations

Operations can be expressed using the simply typed lambda calculus and our three types: *Characteristic Variable* (short: Var), *Characteristic Set Variable* (short: Set) and boolean variables (short: *Bool*). Translated operations can be denoted as follows:

$$
\begin{aligned}
\texttt{isEmpty} &: \mathit{Set} \rightarrow \mathit{Bool} \\
\texttt{elementOf} &: \mathit{Var} \rightarrow \mathit{Set} \rightarrow \mathit{Bool} \\
\texttt{intersection} &: \mathit{Set} \rightarrow \mathit{Set} \rightarrow \mathit{Set} \\
\texttt{union} &: \mathit{Set} \rightarrow \mathit{Set} \rightarrow \mathit{Set} \\
\texttt{equals} &: \mathit{Var} \rightarrow \mathit{Var} \rightarrow \mathit{Bool} \\
\texttt{and} &: \mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool}
\end{aligned}
$$

We use this formalization to show the relation between strictly functional Conditions and the type system of the simply typed lambda calculus. Cardelli uses a formal language to define type systems [8]. A type system is a set of *judgements* which represent assertions. A judgement can be valid or invalid. The typing environment holding free variables is denoted by Γ.

Our abstract syntax implies that the the most-outer operation is always a BooleanOperation. This assertion *"The type of a well-typed condition C is always bool"* can be formalized as follows:

$$\Gamma \vdash C : \mathit{Bool}$$

Based on the concept of judgements, the type system of the simply typed lambda calculus can be defined with the following rules [35].

$$\text{T-Var:} \quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \tag{A.1}$$

$$\text{T-Abs:} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : (A \rightarrow B)} \tag{A.2}$$

$$\text{T-App:} \quad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \tag{A.3}$$

The first rule *(A.1)* handles free variables in the typing environment Γ and is used to extract a variable x of type A which can then be used in a term. The second rule *(A.2)* gives a term a function type $A \rightarrow B$ if its body is typed correctly. The third rule *(A.3)* applies a function M to an argument N.

Based on these foundations, we can verify terms using the formalized operations and the simply typed lambda calculus type system. The typing environment Γ contains all free variables and all translated operations as shown above. This can be denoted as follows:

$$
\begin{aligned}
\mathit{free} &: \text{All free variables defined prior to the condition} \\
\mathit{operations} &: \{\texttt{isEmpty}, \texttt{elementOf}, \texttt{intersection}, \dots\}
\end{aligned}
$$

$$(\mathit{free} \cup \mathit{operations}) \in \Gamma$$

To demonstrate this, we show the type derivation of two terms which use the `isEmpty` operation. First, we use a characteristic set variable *S*. This term is typed correctly. Second, we use a characteristic variable *V* which is not typed correctly because a single variable can not be empty. Thus, `isEmpty` is only defined on set variables. This can also be seen in the formalization of the operations shown above which defines `isEmpty` as lambda expression from *Set* to *Bool*.

The derivation tree for the correctly typed term `isEmpty`(*S*) with *S* being a characteristic set variable looks as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{isEmpty} : Set \to Bool \in \Gamma}{\Gamma, x : Set \vdash \texttt{isEmpty} : Set \to Bool}\,\text{T-Var} \quad \dfrac{x : Set \in \Gamma}{\Gamma, x : Set \vdash x : Set}\,\text{T-Var}}{\Gamma, x : Set \vdash \texttt{isEmpty}(x) : Bool}\,\text{T-App}}{\Gamma \vdash \lambda x : Set\,.\,\texttt{isEmpty}(x) : Set \to Bool}\,\text{T-Abs} \quad \dfrac{S : Set \in \Gamma}{\Gamma \vdash S : Set}\,\text{T-Var}}{\Gamma \vdash (\lambda x : Set\,.\,\texttt{isEmpty}(x))\,S : Bool}\,\text{T-App}$$

Starting at the bottom of the derivation, we first split the term `isEmpty(S)` into its operation `isEmpty(x)` and the variable `S` on which its applied using the T-App rule. This variable is a free variable and thus part of the typing environment Γ which is verified using the T-Var rule. The `isEmpty(x)` operation is further split into its argument and the atomic operation using the T-Abs and T-App rules. Last, the existence of the operation in the typing environment Γ is verified using the T-Var rule. In this derivation, every judgement succeeds until no further rules can be applied. Thus, the term is typed correctly.

Next, we consider the derivation tree of an incorrectly typed term `isEmpty(V)` with *V* being a characteristic variable:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : Set \vdash \texttt{isEmpty}(x) : Bool}}{\Gamma \vdash \lambda x : Set\,.\,\texttt{isEmpty}(x) : Set \to Bool}\,\text{T-Abs} \quad \dfrac{V : Set \neq V : Var \in \Gamma \;\lightning}{\Gamma \vdash V : Set}\,\text{T-Var}}{\Gamma \vdash (\lambda x : Set\,.\,\texttt{isEmpty}(x))\,V : Bool}\,\text{T-App}$$

The structure of this tree is similar to the example discussed before. However, after the application of the T-App rule, a variable of type *Set* is expected because the operation `isEmpty(x)` is defined with the function type $Set \to Bool$. Thus, the T-Var rule tries to find a variable *V* of type *Set* in the typing environment Γ which fails because *V* has the type *Var*. Thus, not every applicable judgement succeeds and the a type error occurs.

These two simple examples show how the simply typed lambda calculus can be used to express and verify Conditions. This approach is applicable to terms of arbitrary size and nesting based on the three judgements presented above. Thus, we consider Conditions which follow our abstract syntax to be type-safe.

A.2. Correctness Evaluation

As part of goal **G4** which considers the equivalence of the analysis, we outlined a correctness proof with question **Q11** in Subsection 9.2.5. In the results discussion in Subsection 9.3.4, we presented only short excerpts of the formalization for the sake of brevity. In this section, we show more comprehensive examples. We present concepts of the source context C, the source language L, the target context K and the target language T. Then, we show the formalization of the transformation γ as well as the semantics of L and T. Last, we present more examples of the induction basis as part of the structural induction. Please note, that the formalization shown in this Section is still not exhaustive and has only been performed for the subset required for the structural induction.

$$\Upsilon_C = \begin{cases} \text{CharacteristicType(C)}: & C \textit{ is the name of a characteristic type} \\ \text{CharacteristicLiteral(L,C)}: & L \textit{ is the name of a literal of Characteristic type } C \\ \text{ArchitecturalElement(N,L)}: & N \textit{ is the name of an architectural elem. with literal } L \end{cases}$$

$$\Upsilon_L = \begin{cases} \text{CharacteristicTypeSelectorSingle(C,L)}: & C \textit{ is a characteristic type, } L \textit{ one of its literals} \\ \text{CharacteristicTypeSelectorNegated(C,L)}: & C \textit{ is a characteristic type, } L \textit{ one of its literals} \\ \text{CharacteristicTypeSelectorMulti(C,L1,L2)}: & C \textit{ is a characteristic type, L1 and L2 its literals} \\ \text{CharacteristicClass(N,C)}: & N \textit{ is the name of a class, } L \textit{ a list of CharacteristicTypeSelectors}^* \\ \text{AttributeSelector(C)}: & C \textit{ is a CharacteristicTypeSelector}^* \\ \text{PropertySelector(C)}: & C \textit{ is a CharacteristicTypeSelector}^* \\ \text{AttributeClassSelector(C,N)}: & C \textit{ is a characteristic type, } N \textit{ is a characteristic class} \\ \text{PropertyClassSelector(C,N)}: & C \textit{ is a characteristic type, } N \textit{ is a characteristic class} \\ \text{NodeIdentitySelector(N)}: & N \textit{ is the name of an architectural element} \\ \text{Rule(D,Z)}: & D \textit{ is the rule's DataSelector}^*, Z \textit{ its DestinationSelector}^* \\ \text{Constraint(N,R)}: & N \textit{ is the name of a constraint, } R \textit{ is its rule} \end{cases}$$

$$\Upsilon_K = \begin{cases} \text{ValueSetType(T)}: & T \textit{ is the name of a ValueSetType} \\ \text{AttributeType(A,T)}: & A \textit{ is the name of an attribute type, } T \textit{ its ValueSetType} \\ \text{PropertyType(P,T)}: & P \textit{ is the name of a property type, } T \textit{ its ValueSetType} \\ \text{Operation(O)}: & O \textit{ is the name of an operation} \\ \text{Term(V,A)}: & V \textit{ is a Prolog term, } A \textit{ a list of arguments (terms)} \\ \text{Fact(H)}: & H \textit{ is the facts head (term)} \\ \text{Clause(H,B)}: & H \textit{ is the clauses head, } B \textit{ its body; both are terms} \\ \text{Unification(L,R)}: & \textit{Unification of } L \textit{ and } R\textit{; both are terms} \\ \text{NotProvable(P)}: & \textit{Invert the statement of } P\textit{; } P \textit{ is a term} \\ \text{And(L,R)}: & \textit{Combine the statements of } L \textit{ and } R\textit{; both are terms} \\ \text{Or(L,R)}: & \textit{Combine the statements of } L \textit{ and } R\textit{; both are terms} \end{cases}$$

$$\Upsilon_T = \begin{cases} \text{ValueSetMember(T,V)}: & T \text{ is a ValueSetType, } V \text{ a value} \\ \text{StackValid(S)}: & S \text{ is the name of stack which represents a valid call sequence} \\ \text{CallArgument(S,P,A,V)}: & S \text{ is a call stack, } P \text{ a parameter name,} \\ & \quad A \text{ an AttributeType, } V \text{ a value of this AttributeType's ValueSetType} \\ \text{ReturnValue(S,R,A,V)}: & S \text{ is a call stack, } r \text{ a return value name,} \\ & \quad A \text{ an AttributeType, } V \text{ a value of this AttributeType's ValueSetType} \\ \text{PreCallState(S,O,T,A,V)}: & S \text{ is a call stack, } O \text{ an operation, } T \text{ a state variable} \\ & \quad A \text{ an AttributeType, } V \text{ a value of this AttributeType's ValueSetType} \\ \text{PostCallState(S,O,T,A,V)}: & S \text{ is a call stack, } O \text{ an operation, } T \text{ a state variable} \\ & \quad A \text{ an AttributeType, } V \text{ a value of this AttributeType's ValueSetType} \\ \text{OperationProperty(O,P,V)}: & O \text{ is an operation, } P \text{ an PropertyType} \\ & \quad V \text{ a value of this PropertyType's ValueSetType} \end{cases}$$

$$\tau = \begin{cases} \text{NodeIdentitySelector(N)} \\ \quad \longrightarrow \text{Unification(Term(O),N)} \\ \text{CharacteristicClass(N,CharacteristicTypeSelectorSingle(C,L))} \\ \quad \longrightarrow \text{Fact(Term(X),L),Clause(Term(N),And(ValueSetMember(C,L),Term(X,L)))} \\ \text{CharacteristicClass(N,CharacteristicTypeSelectorNegated(C,L))} \\ \quad \longrightarrow \text{Fact(Term(X),L),Clause(Term(N),And(ValueSetMember(C,L),} \\ \quad\quad \text{NotProvable(Term(X,L))))} \\ \text{CharacteristicClass(N,CharacteristicTypeSelectorMulti(C,L1,L2))} \\ \quad \longrightarrow \text{Fact(Term(X),L1),Fact(Term(Y),L2),Clause(Term(N),And(ValueSetMember(C,L),} \\ \quad\quad \text{Or(Term(X,L1),Term(Y,L2))))} \\ \text{PropertySelector(CharacteristicTypeSelectorSingle(C,L))} \\ \quad \longrightarrow \text{OperationProperty(Term(O),C,L)} \\ \text{PropertySelector(CharacteristicTypeSelectorNegated(C,L))} \\ \quad \longrightarrow \text{NotProvable(OperationProperty(Term(OP),C,L))} \\ \text{PropertySelector(CharacteristicTypeSelectorMulti(C,L1,L2))} \\ \quad \longrightarrow \text{Or(OperationProperty(Term(O),C,L1),OperationProperty(Term(O),C,L2))} \\ \text{AttributeSelector(CharacteristicTypeSelectorSingle}(C, L)) \\ \quad \longrightarrow \text{CallArgument(Term}(S), \text{Term}(P), C, L) \\ \text{AttributeSelector(CharacteristicTypeSelectorNegated(C,L))} \\ \quad \longrightarrow \text{NotProvable(CallArgument(Term(S),Term(P),C,L))} \\ \text{AttributeSelector(CharacteristicTypeSelectorMulti(C,L1,L2))} \\ \quad \longrightarrow \text{Or(CallArgument(Term(S),Term(P),C,L1),CallArgument(Term(S),Term(P),C,L2))} \end{cases}$$

$$\tau = \begin{cases} \text{PropertyClassSelector(C,N)} \\ \quad \rightarrow \text{And(OperationProperty(Term(OP),C,Term(V)),Term(N,V))} \\ \text{AttributeClassSelector(C,N)} \\ \quad \rightarrow \text{And(CallArgument(Term(S),Term(P),C,Term(V)),Term(N,V))} \\ \text{Constraint}(N, \text{Rule}(D, Z)) \\ \quad \rightarrow \text{Clause(Term}(N, O, S, P), \\ \quad\quad \text{And(Unification}(S, \text{List}(O)), \text{And(StackValid}(S), \text{And}(D, Z)))) \end{cases}$$

$$val_\theta(\text{NodeIdentitySelector(N)} = \begin{cases} true, \text{ if an architectural element with name } N \text{ is selected} \\ false, \text{ otherwise} \end{cases}$$

$$val_\theta(\text{CharacteristicClass(N,CharacteristicTypeSelectorSingle(C,L))} = \begin{cases} true, \text{ if used to select a literal } L \text{ of characteristic type } C \\ false, \text{ otherwise} \end{cases}$$

$$val_\theta(\text{CharacteristicClass(N,CharacteristicTypeSelectorNegated(C,L))} = \begin{cases} true, \text{ if used to select any literal except } L \text{ of} \\ \quad \text{characteristic type } C \\ false, \text{ otherwise} \end{cases}$$

$$val_\theta(\text{CharacteristicClass(N,CharacteristicTypeSelectorMulti(C,L1,L2))} = \begin{cases} true, \text{ if used to select a literal } L1 \text{ or } L2 \text{ of} \\ \quad \text{characteristic type } C \\ false, \text{ otherwise} \end{cases}$$

$$val_\theta(\text{PropertySelector(CharacteristicTypeSelectorSingle(C,L))} = \begin{cases} true, \text{ if an ArchitecturalElement with literal } L \text{ of} \\ \quad \text{characteristic type } C \text{ is selected} \\ false, \text{ otherwise} \end{cases}$$

$$val_\theta(\text{PropertySelector(CharacteristicTypeSelectorNegated(C,L))} = \begin{cases} true, \text{ true, if an ArchitecturalElement without literal } L \\ \quad \text{of characteristic type } C \text{ is selected} \\ false, \text{ otherwise} \end{cases}$$

$val_\theta(\text{PropertySelector(CharacteristicTypeSelectorMulti(C,L1,L2))} =$

$$\begin{cases} \textit{true}, & \textit{if an ArchitecturalElement with either literal} \\ & \textit{L1 or L2 of characteristic type C is selected} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

$val_\theta(\text{AttributeSelector(CharacteristicTypeSelectorSingle}(C, L))) =$

$$\begin{cases} \textit{true}, & \textit{if an ArchitecturalElement with a data flow with literal L} \\ & \textit{of characteristic type C is selected} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

$val_\theta(\text{AttributeSelector(CharacteristicTypeSelectorNegated(C,L))} =$

$$\begin{cases} \textit{true}, & \textit{if an ArchitecturalElement without a data flow} \\ & \textit{with literal L of characteristic type C is selected} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

$val_\theta(\text{AttributeSelector(CharacteristicTypeSelectorMulti(C,L1,L2))} =$

$$\begin{cases} \textit{true}, & \textit{if an ArchitecturalElement with a data flow with literal} \\ & \textit{L1 or L2 of characteristic type C is selected} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

$val_\theta(\text{PropertyClassSelector(C,N)} =$

$$\begin{cases} \textit{true}, & \textit{if an ArchitecturalElement with characteristic type C is} \\ & \textit{selected for which the evaluation of class N returns true} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

$val_\theta(\text{AttributeClassSelector(C,N)} =$

$$\begin{cases} \textit{true}, & \textit{if an ArchitecturalElement with a data flow of characteristic type C} \\ & \textit{is selected for which the evaluation of class N returns true} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

$val_\theta(\text{Constraint}(N, \text{Rule}(D, Z))) =$

$$\begin{cases} \textit{true}, & \textit{if a selection of both D and Z is true for any valid} \\ & \textit{data flow between any ArchitecturalElements} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

val_ψ(ValueSetMember(T,V) =

$$\begin{cases} \textit{true}, & \textit{if V is the name of a Value which belongs to the} \\ & \textit{ValueSetType with the name T} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

val_ψ(StackValid(S) =

$$\begin{cases} \textit{true}, & \textit{if the list S represents a correct call sequence} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

val_ψ(CallArgument(S,P,A,V) =

$$\begin{cases} \textit{true}, & \textit{if the Value V of the Attribute A of the parameter P} \\ & \textit{is present given the call stack S. The operation which owns the} \\ & \textit{Parameter P is defined by the stack top of} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

val_ψ(ReturnValue(S,R,A,V) =

$$\begin{cases} \textit{true}, & \textit{if the Value V of the Attribute A of the return value R} \\ & \textit{is present given the call stack S. The operation which owns the} \\ & \textit{Return Value R is defined by the stack top of S} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

val_ψ(PreCallState(S,O,T,A,V) =

$$\begin{cases} \textit{true}, & \textit{if the Value V of the Attribute A of the state variable T} \\ & \textit{owned by the operation O is present before the call to the} \\ & \textit{operation on top of the call stack S is executed} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

val_ψ(PostCallState(S,O,T,A,V) =

$$\begin{cases} \textit{true}, & \textit{if the Value V of the Attribute A of the state variable T} \\ & \textit{owned by the operation O is present after the call to the} \\ & \textit{operation on top of the call stack S is executed} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

val_ψ(OperationProperty(O,P,V) =

$$\begin{cases} \textit{true}, & \textit{if the property P of Operation O has the} \\ & \textit{Value V as present value} \\ \textit{false}, & \textit{otherwise} \end{cases}$$

Source syntax: NodeIdentitySelector(N)
Target syntax: Unification(Term(O),N)
Source semantics: *true, if an architectural element with name N is selected*
Target semantics: *true, if N can be unified with O which is the case if O has the name N*
Encoding: *Valid, because O is the the transformed architectural element currently selected*

Source syntax: PropertySelector(CharacteristicTypeSelectorSingle(C,L))
Target syntax: OperationProperty(Term(O),C,L)
Source semantics: *true, if an ArchitecturalElement with literal L of characteristic type C is selected*
Target semantics: *true, if the property C of Operation O has the Value L as present value*
Encoding: *Valid because O is the the transformed architectural element currently selected*

Source syntax: AttributeSelector(CharacteristicTypeSelectorSingle(C, L))
Target syntax: CallArgument(Term(S), Term(P), C, L)
Source semantics: *true, if an ArchitecturalElement with a data flow with literal L of characteristic type C is selected*
Target semantics: *true, if the Value L of the Attribute C of the parameter P is present given the call stack S. The operation which owns the Parameter P is defined by the stack top of*
Encoding: *Valid because the data flow with literal L of characteristic type C is represented by the parameter P in call stack S. The selected architectural element is represented by the operation on the top of the stack S. Thus, the implication holds.*

Source syntax: AttributeSelector(CharacteristicTypeSelectorNegated(C,L))
Target syntax: NotProvable(CallArgument(Term(S),Term(P),C,L))
Source semantics: *true, if an ArchitecturalElement without a data flow with literal L of characteristic type C is selected*
Target semantics: *false, if the Value L of the Attribute C of the parameter P is present given the call stack S. The operation which owns the Parameter P is defined by the stack top of*
Encoding: *Valid, because the CallArgument query acts as described above. But only if no parameter P with the given requirements exists, the term cannot be proven. In this case, the NotProvable predicate returns true as is implicated by the negated selector.*

Source syntax: AttributeSelector(CharacteristicTypeSelectorMulti(C,L1,L2))
Target syntax: Or(CallArgument(Term(S),Term(P),C,L1),CallArgument(Term(S),Term(P),C,L2))
Source semantics: *true, if an ArchitecturalElement with a data flow with literal L1 or L2 of characteristic type C is selected*
Target semantics: *true, if the values L1 or L2 of the Attribute C of the parameter P are present given the call stack S. The operation which owns the Parameter P is defined by the stack top of*
Encoding: *Either L1 or L2 have to present in order to select the architectural element (or its mapped operation on top of call stack S). This is true because either the first or second CallArgument query have to be proven in order to prove the Or predicate. Thus, the implication holds.*

Source syntax: CharacteristicClass(N,CharacteristicTypeSelectorSingle(C,L))
Target syntax: Fact(Term(X),L),Clause(Term(N),And(ValueSetMember(C,L),Term(X,L)))
Source semantics: *true, if used to select a literal L of characteristic type C*
Target semantics: *true, if L is the name of a Value which belongs to the ValueSetType with the name C and L is used in fact X*
Encoding: *Valid because literal L belongs to characteristic type c and thus also to the generated ValueSetType. The fact X can be proven for the literal L. Thus, the conjunction is true and the implications holds.*

Source syntax: Constraint(N, Rule(D, Z))
Target syntax: Clause(Term(N, O, S, P),
And(Unification(S, List(O)), And(StackValid(S), And(D, Z))))
Source semantics: *true, if a selection of both D and Z is true for any valid data flow between any ArchitecturalElements*
Target semantics: *true, if at least one call stack S satisfies both D and Z. The operation O is hereby the head of the call stack S.*
Encoding: *Any valid data flow is represented by any valid call stack in the Operation Model. If a selection of both D and Z is valid for at least one data flow, both D and Z can be proven for this state of the call stack S and the operation O on the top of the stack. Thus, the implication holds. The concrete selection depends on the choice of selectors as shown above.*

A.3. Mapped Prolog Code Examples

In Section 8.6, we discussed the transformation of two exemplary scenarios: The *Geolocation Constraints* scenario and the *Access Control* scenario. For the sake of brevity, we only presented excerpts from the generated Prolog code using the DSL mapping. In this Section, we show the complete generated code for both examples.

```
1  class UnsafeLocations {
2    location.!EU
3  }
4
5  constraint NoFlowToUnsafeLocations {
6    data.attribute.privacy.personal NEVER FLOWS node.class.UnsafeLocations
7  }
8
9  constraint AccessRightsViolation {
10   data.attribute.accessRights.$rights{}
11     NEVER FLOWS node.property.roles.$roles{}
12     WHERE isEmpty(intersection(rights,roles))
13 }
```

Listing A.2: Shortened concrete syntax of constraints of both exemplary scenarios

Listing A.2 repeats the constraints of both scenarios. The first constraint in line 5 to 7 and the CharacteristicClass in line 1 to 3 belong to the *Geolocation Constraints* scenario. The second constraint in line 9 to 13 belongs to *Access Control* scenario. We show the generated code of the *Geolocation Constraints* scenario in Listing A.3 and the code of the *Access Control* scenario in Listing A.4.

```
1  characteristicsClass_UnsafeLocations_location_0_NEG('EU').
2  characteristicsClass_UnsafeLocations(ClassVar_location) :-
3    valueSetMember('location', ClassVar_location),
4    \+ characteristicsClass_UnsafeLocations_location_0_NEG(ClassVar_location).
5
6  constraint_NoFlowToUnsafeLocations(ConstraintName, QueryType, OP, S, P, ClassVar_location)
       :-
7    ConstraintName = 'NoFlowToUnsafeLocations',
8    (constraint_NoFlowToUnsafeLocations_CallArgument(QueryType, OP, S, P, ClassVar_location);
9    constraint_NoFlowToUnsafeLocations_ReturnValue(QueryType, OP, S, P, ClassVar_location);
10   constraint_NoFlowToUnsafeLocations_PreCallState(QueryType, OP, S, ST, ClassVar_location);
11   constraint_NoFlowToUnsafeLocations_PostCallState(QueryType, OP, S, ST,
         ClassVar_location)).
12
13 constraint_NoFlowToUnsafeLocations_CallArgument(QueryType, OP, S, P, ClassVar_location) :-
14   QueryType = 'CallArgument',
15   S = [OP | _],
16   stackValid(S),
17   operationParameter(OP, P),
18   callArgument(S, P, 'privacy', 'personal'),
19   operationProperty(OP, 'location', ClassVar_location),
20   characteristicsClass_UnsafeLocations(ClassVar_location).
21
22 constraint_NoFlowToUnsafeLocations_ReturnValue(QueryType, OP, S, P, ClassVar_location) :-
23  QueryType = 'ReturnValue',
24  S = [OP | _],
25  stackValid(S),
26  operationReturnValue(OP, P),
27  returnValue(S, P, 'privacy', 'personal'),
28  operationProperty(OP, 'location', ClassVar_location),
29  characteristicsClass_UnsafeLocations(ClassVar_location).
30
31 constraint_NoFlowToUnsafeLocations_PreCallState(QueryType, OP, S, ST, ClassVar_location) :-
32  QueryType = 'PreCallState',
33  S = [OP | _],
34  stackValid(S),
35  operationState(OP, ST),
36  preCallState(S, OP, ST, 'privacy', 'personal'),
37  operationProperty(OP, 'location', ClassVar_location),
38  characteristicsClass_UnsafeLocations(ClassVar_location).
39
40 constraint_NoFlowToUnsafeLocations_PostCallState(QueryType, OP, S, ST, ClassVar_location) :-
41  QueryType = 'PostCallState',
42  S = [OP | _],
43  stackValid(S),
44  operationState(OP, ST),
45  postCallState(S, OP, ST, 'privacy', 'personal'),
46  operationProperty(OP, 'location', ClassVar_location),
47  characteristicsClass_UnsafeLocations(ClassVar_location).
```

Listing A.3: Complete transformation result of the Geolocation Constraints scenario

```
1  constraint_AccessRightsViolation(ConstraintName, QueryType, OP, S, P, ST, VarSet_rights,
       VarSet_roles) :-
2    ConstraintName = 'AccessRightsViolation',
3    (constraint_AccessRightsViolation_CallArgument(QueryType, OP, S, P, VarSet_rights, VarSet_roles);
4    constraint_AccessRightsViolation_ReturnValue(QueryType, OP, S, P, VarSet_rights, VarSet_roles);
5    constraint_AccessRightsViolation_PreCallState(QueryType, OP, S, ST, VarSet_rights, VarSet_roles);
6    constraint_AccessRightsViolation_PostCallState(QueryType, OP, S, ST, VarSet_rights, VarSet_roles)).
7
8  constraint_AccessRightsViolation_CallArgument(QueryType, OP, S, P, VarSet_rights, VarSet_roles) :-
9    QueryType = 'CallArgument',
10   S = [OP | _],
11   stackValid(S),
12   operationParameter(OP, P),
13   findall(IteratorTemplate, callArgument(S, P, 'authorizedRoles', IteratorTemplate), VarSet_rights),
14   findall(IteratorTemplate, operationProperty(OP, 'accessRoles', IteratorTemplate), VarSet_roles),
15   intersection(VarSet_rights, VarSet_roles, Temp_0), length(Temp_0, 0).
16
17 constraint_AccessRightsViolation_ReturnValue(QueryType, OP, S, P, VarSet_rights, VarSet_roles) :-
18   QueryType = 'ReturnValue',
19   S = [OP | _],
20   stackValid(S),
21   operationReturnValue(OP, P),
22   findall(IteratorTemplate, returnValue(S, P, 'authorizedRoles', IteratorTemplate), VarSet_rights),
23   findall(IteratorTemplate, operationProperty(OP, 'accessRoles', IteratorTemplate), VarSet_roles),
24   intersection(VarSet_rights, VarSet_roles, Temp_0), length(Temp_0, 0).
25
26 constraint_AccessRightsViolation_PreCallState(QueryType, OP, S, ST, VarSet_rights, VarSet_roles) :-
27   QueryType = 'PreCallState',
28   S = [OP | _],
29   stackValid(S),
30   operationState(OP, ST),
31   findall(IteratorTemplate, preCallState(S, OP, ST, 'authorizedRoles', IteratorTemplate),
       VarSet_rights),
32   findall(IteratorTemplate, operationProperty(OP, 'accessRoles', IteratorTemplate), VarSet_roles),
33   intersection(VarSet_rights, VarSet_roles, Temp_0), length(Temp_0, 0).
34
35 constraint_AccessRightsViolation_PostCallState(QueryType, OP, S, ST, VarSet_rights, VarSet_roles) :-
36   QueryType = 'PostCallState',
37   S = [OP | _],
38   stackValid(S),
39   operationState(OP, ST),
40   findall(IteratorTemplate, postCallState(S, OP, ST, 'authorizedRoles', IteratorTemplate),
       VarSet_rights),
41   findall(IteratorTemplate, operationProperty(OP, 'accessRoles', IteratorTemplate), VarSet_roles),
42   intersection(VarSet_rights, VarSet_roles, Temp_0), length(Temp_0, 0).
```

Listing A.4: Complete transformation result of the Access Control scenario

A.4. Meta-Model

We present the complete abstract syntax of our meta-model in Figure A.1. In order to enhance the readability, we omit attributes and the relations between Operations and References. For more information, please see Chapter 7.

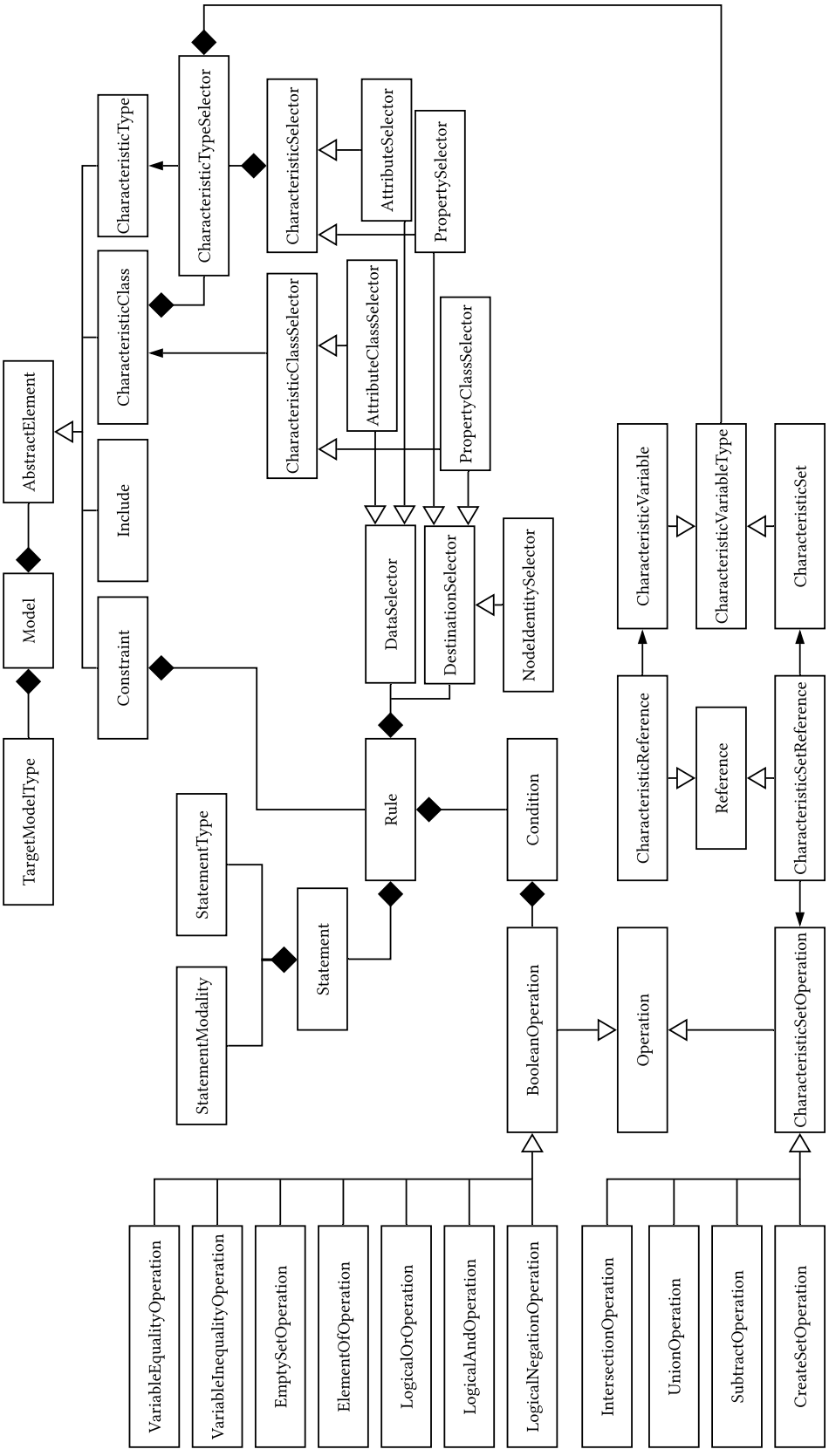

Figure A.1.: Complete abstract syntax of the meta-model of our DSL