Karlsruhe Institute of Technology

Software Design and Quality

Efficient Data Flow Constraint Analysis

Master's Thesis of

Jonas Kunz

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

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor:	Dr. rer. nat. Robert Heinrich
Second advisor:	M.Sc. Stephan Seifermann

7. May 2018 – 6. November 2018

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, 06.11.18

. . . .

(Jonas Kunz)

Abstract

With recent trends such as cloud computing and micro services modern software systems become more and more decentralized. As a result more and more data processed by the systems flows across public networks and environments hosted by third parties. With recent legal regulations, such as the *General Data Protection Regulations* of the EU, it becomes even more important for software developers to ensure that all data flows of their software adhere to legal constraints. While several model-based approaches have been proposed for modeling data flows and related constraints on architecture level their automated analysis capabilities are limited. In many cases no automated analysis is available or an analysis has to be implemented on a per-scenario basis.

We therefore propose a novel meta-model for modeling data flows of software systems. Alongside we provide a translation transforming model instances to programs based on the logic programming language Prolog. This combination allows to easily define automated analysis of software systems regarding data flow constraint violations. For the design and implementation we ensure that our approach is efficient regarding the scalability. For this purpose we introduce several techniques for optimizing Prolog programs which are not only limited to our approach.

In addition we provide an extensive evaluation of our approach. Hereby we investigate the accuracy, the scalability and the genericness of our approach. We show that our approach is able to accurately analyse various types of scenarios while maintaining a good scalability. We show that our proposed Prolog optimizations are effective as they potentially reduce the run time from exponential to constant scaling in certain scenarios.

Zusammenfassung

Aktuelle Entwicklungen in der Software-Technik zeigen einen Trend zur Dezentralisierung von Software-Systemen. Mit dem Einsatz von Techniken wie Cloud-Computing oder Micro-Services fließen immer mehr Daten über öffentliche Netzwerke oder über die Infrastruktur von Drittanbietern. Im Gegensatz dazu führen aktuelle gesetzliche Änderungen wie die *Datenschutz-Grundverordnung* dazu, dass es für Software-Entwickler immer wichtiger wird sicherzustellen, dass die Datenflüsse ihrer Software gesetzliche Beschränkungen einhalten. Um dies trotz der stetig wachsenden Komplexität von Software-Systemen zu ermöglichen wurden verschiedene modellbasierte Ansätze auf Architekturebene vorgeschlagen. Ein Nachteil der meisten Ansätze ist jedoch, dass sie oftmals keine voll automatisierte Analyse bezüglich der Verletzung von Datenflussbeschränkungen ermöglichen. Oft sind keine automatisierten Analysen möglich oder Analysen müssen individuell für jedes Szenario entwickelt werden.

Aus diesem Grund schlagen wir ein neues Metamodell zur Beschreibung der Datenflüssen von Softwaresystemen vor. Dieses Metamodell ist so entworfen, dass eine automatisierte übersetzung von Instanzen in ein Programm der logischen Programmiersprache Prolog ermöglicht wird. Dieses Programm erlaubt dann die einfache Formulierung von Regeln zur automatisierten Prüfung der Einhaltung von Datenflussbeschränkungen. Ein wichtiger Aspekt für den Entwurf und die Implementierung unseres Ansatzes ist die Skalierbarkeit: Ziel ist es, sicherzustellen dass unser Ansatz effizient einsetzbar ist. Hierbei haben wir insbesondere Techniken zur Optimierung von Prolog Programmen entwickelt, deren Einsatzmöglichkeiten nicht nur auf unseren Ansatz beschränkt sind.

Desweiteren haben wir eine umfangreiche Evaluation unseres Ansatzes durchgeführt. Hierbei haben wir die Genauigkeit, Skalierbarkeit sowie die Generizität unseres Ansatzes untersucht. Wir haben gezeigt, dass unser Ansatz für mehrere Arten von Szenarien genau arbeitet und dabei eine gute Skalierbarkeit aufweist. Es hat sich herausgestellt, dass unsere vorgestellten Optimierungen in manchen Fällen sogar zu einer Reduktion von exponentieller zu konstanter Laufzeit führen können.

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1
- 1.1. Motivation 1
- 1.2. Contribution 2
- 1.3. Overview 2

2. Fundamentals 3
- 2.1. Software Modeling 3
- 2.2. Prolog 3
  - 2.2.1. Basic Concepts 4
  - 2.2.2. Cut Predicate 5
  - 2.2.3. Negation 6
  - 2.2.4. Predicate Indexing 7

3. Data Flow Scenarios 9
- 3.1. Access Control 9
- 3.2. Geolocation Privacy Restrictions 12

4. Related Work 17
- 4.1. Data Flow Analysis 17
- 4.2. Prolog Performance 18

5. Data Flow System Model 19
- 5.1. Conceptual Overview 19
- 5.2. Data Flows without Control Flow 22
- 5.3. Type Definitions 23
- 5.4. Operations 25
- 5.5. Propositional Logic Terms 27
- 5.6. Variable Assignments 30
- 5.7. Example Instances 33
  - 5.7.1. Access Control Scenario 34
  - 5.7.2. Geolocation Constraints Scenario 36

6. Translation to Prolog 41
- 6.1. Constraint Query API 41
  - 6.1.1. Type Information 41

6.1.2. Operations, SystemUsages and Calls 43
6.2. Constraint Formulation Examples 46
6.2.1. Access Control Scenario 46
6.2.2. Geolocation Constraints Scenario 48
6.3. Deriving the Prolog Program 50
6.3.1. Program Header Definition 50
6.3.2. Definition of Operations and SystemUsages 54

7. Prolog Optimizations **59**
7.1. First Argument Indexing 59
7.2. Efficient Logical Negations 61
7.3. Cut-Based Assignments 64

8. Evaluation **69**
8.1. Goals and Questions 69
8.2. Evaluation Design 71
8.2.1. Randomized Call Graph Scaling 72
8.2.2. G1 - Accuracy 73
8.2.3. G2 - Scalability 77
8.2.4. G3 - Genericness 82
8.2.5. Prolog Implementations 83
8.2.6. Experiment System Specification 84
8.3. Results 84
8.3.1. G1 - Accuracy 84
8.3.2. G2 - Scalability 85
8.3.3. G3 - Genericness 100
8.3.4. Summary 102
8.4. Threats to Validity 103
8.5. Assumptions and Limitations 105

9. Conclusion **107**
9.1. Summary 107
9.2. Future Work 108

Bibliography **109**

A. Appendix **113**

List of Figures

3.1. UML sequence diagram of the slightly adapted booking process of the Travel Planner example. . . . 10
3.2. UML sequence diagram of the interaction of a user with the example online shop system. . . . 14

5.1. Conceptual illustration of an instance of our meta model . . . 20
5.2. Example of an decryption operation . . . 20
5.3. Minimal example of a data joining scenario where a data flow without control flow is required. . . . 22
5.4. Functionally equivalent scenario to the scenario in Figure 5.3 with all data flows bound to the control flow. . . . 23
5.5. Excerpt of the UML class diagram of the system element and its contents 24
5.6. Excerpt of the UML class diagram of the operation class and its related elements . . . 25
5.7. Excerpt of the UML class diagram of the basic propositional logic terms. 27
5.8. Excerpt of the UML class diagram of the reference types for logic terms. 28
5.9. Excerpt of the UML class diagram of the state reference types for logic terms. 29
5.10. Excerpt of the UML class diagram of the *VariableAssignment* type. . . . 31
5.11. Full system model of the excerpt from the TravelPlanner case study. . . . 35
5.12. Full system model of the online shop example for geolocation constraints. 38

6.1. Illustration of the resulting program structure of our proposed Prolog translator. . . . 51

8.1. Basic structural change actions for call graph scaling. . . . 72
8.2. System model of the TravelPlanner case study with violations injected. The changed parts in comparison to Figure 5.11 are highlighted in red. The assignments of the call chain of *TravelPlanner-getFlightOffers* are unchanged and have been omitted in the figure to preserve clarity. . . . 74
8.3. Model used for performance testing of the indexing optimization. The number of paths to the process operation scales linearly with n . . . 80
8.4. Model used for performance testing of the negation optimization. The number of paths to the process operation scales exponentially with n . . 81
8.5. Translation time measured for the scaling with the number of operations (**RQ-2.1**). . . . 85
8.6. Load time of ECLiPSe measured for the scaling with the number of operations (**RQ-2.1**). . . . 86
8.7. Analysis time of JIProlog measured for the scaling with the number of operations (**RQ-2.1**). . . . 87

8.8. Analysis time of ECLiPSe measured for the scaling with the number of operations (**RQ-2.1**). . . . 87
8.9. Load time of ECLiPSe measured for the scaling with the complexity of the call graph (**RQ-2.2**). . . . 88
8.10. Analysis time of JIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**). . . . 88
8.11. Comparison of the analysis times of JIProlog with and without the indexing optimization for **RQ-2.1** and **RQ-2.2**. . . . 89
8.12. Translation time measured for the scaling with the number of parameters and return values (**RQ-2.3**). . . . 90
8.13. Load time of ECLiPSe measured for the scaling with the number of parameters and return values (**RQ-2.3**). . . . 90
8.14. Analysis time of ECLiPSe measured for the scaling with the number of parameters and return values (**RQ-2.3**). . . . 91
8.15. Analysis time of SWIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**). . . . 91
8.16. Translation time measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). . . . 92
8.17. Load time of ECLiPSe measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). . . . 93
8.18. Load time of JIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). . . . 93
8.19. Analysis time of SWIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). . . . 94
8.20. Analysis time of ECLiPSe measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). . . . 94
8.21. Translation time measured for the scaling with the number of properties (**RQ-2.5**). . . . 95
8.22. Load time of ECLiPSe measured for the scaling with the number of properties (**RQ-2.5**). . . . 96
8.23. Analysis time of SWIProlog measured for the scaling with the number of properties (**RQ-2.5**). . . . 96
8.24. Analysis time of ECLiPSe measured for the scaling with the number of properties (**RQ-2.5**). . . . 97
8.25. Analysis time of ECLiPSe measured for the first argument indexing optimization performance experiment (**RQ-2.6**). . . . 97
8.26. Analysis time of SWIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**). . . . 98
8.27. Analysis time of SWIProlog measured for the negation optimization experiment (**RQ-2.7**). . . . 99

A.1. Load time of JIProlog measured for the scaling with the number of operations (**RQ-2.1**). . . . 113
A.2. Load time of SWIProlog measured for the scaling with the number of operations (**RQ-2.1**). . . . 113

A.3. Analysis time of SWIProlog measured for the scaling with the number of operations (**RQ-2.1**). 114
A.4. Total time of ECLiPSe measured for the scaling with the number of operations (**RQ-2.1**). 114
A.5. Total time of JIProlog measured for the scaling with the number of operations (**RQ-2.1**). 114
A.6. Total time of SWIProlog measured for the scaling with the number of operations (**RQ-2.1**). 115
A.7. Translation time measured for the scaling with the complexity of the call graph (**RQ-2.2**). 115
A.8. Load time of JIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**). 115
A.9. Load time of SWIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**). 116
A.10. Analysis time of ECLiPSe measured for the scaling with the complexity of the call graph (**RQ-2.2**). 116
A.11. Analysis time of SWIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**). 116
A.12. Total time of ECLiPSe measured for the scaling with the complexity of the call graph (**RQ-2.2**). 117
A.13. Total time of JIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**). 117
A.14. Total time of SWIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**). 117
A.15. Load time of JIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**). 118
A.16. Load time of SWIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**). 118
A.17. Analysis time of JIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**). 118
A.18. Total time of ECLiPSe measured for the scaling with the number of parameters and return values (**RQ-2.3**). 119
A.19. Total time of JIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**). 119
A.20. Total time of SWIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**). 119
A.21. Load time of SWIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). 120
A.22. Analysis time of JIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). 120
A.23. Total time of ECLiPSe measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). 120
A.24. Total time of JIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). 121

A.25. Total time of SWIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**). 121
A.26. Load time of JIProlog measured for the scaling with the number of properties (**RQ-2.5**). 121
A.27. Load time of SWIProlog measured for the scaling with the number of properties (**RQ-2.5**). 122
A.28. Analysis time of JIProlog measured for the scaling with the number of properties (**RQ-2.5**). 122
A.29. Total time of ECLiPSe measured for the scaling with the number of properties (**RQ-2.5**). 122
A.30. Total time of JIProlog measured for the scaling with the number of properties (**RQ-2.5**). 123
A.31. Total time of SWIProlog measured for the scaling with the number of properties (**RQ-2.5**). 123
A.32. Translation time measured for the first argument indexing optimization performance experiment (**RQ-2.6**). 123
A.33. Load time of ECLiPSe measured for the first argument indexing optimization performance experiment (**RQ-2.6**). 124
A.34. Load time of JIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**). 124
A.35. Load time of SWIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**). 124
A.36. Analysis time of JIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**). 125
A.37. Total time of ECLiPSe measured for the first argument indexing optimization performance experiment (**RQ-2.6**). 125
A.38. Total time of JIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**). 125
A.39. Total time of SWIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**). 126
A.40. Translation time measured for the negation optimization experiment (**RQ-2.7**). 126
A.41. Load time of ECLiPSe measured for the negation optimization experiment (**RQ-2.7**). 126
A.42. Load time of JIProlog measured for the negation optimization experiment (**RQ-2.7**). 127
A.43. Load time of SWIProlog measured for the negation optimization experiment (**RQ-2.7**). 127
A.44. Analysis time of ECLiPSe measured for the negation optimization experiment (**RQ-2.7**). 127
A.45. Analysis time of JIProlog measured for the negation optimization experiment (**RQ-2.7**). 128
A.46. Total time of ECLiPSe measured for the negation optimization experiment (**RQ-2.7**). 128

A.47. Total time of JIProlog measured for the negation optimization experiment (**RQ-2.7**). 128
A.48. Total time of SWIProlog measured for the negation optimization experiment (**RQ-2.7**). 129

List of Tables

3.1. Roles of the involved Actors and Systems of the Travel Planner example. 11
3.2. Involved Data with its access control restrictions of the Travel Planner example. 11

5.1. Notation for logic terms. The values in square brackets are replaced with their real values when used. 30
5.2. Typing restrictions induced by reference logic terms based on wildcards. 33
5.3. DataTypes of the model instance for the TravelPlanner example. 34
5.4. DataTypes of the model instance for the online shop example. 37

1. Introduction

In this chapter we provide a short introduction to the field covered by this masters thesis. First we motivate the problem this work attempts to solve. Afterwards we outline our approach for solving it. Finally, an overview of the contents of this work is given in the last section.

1.1. Motivation

Developing and maintaining modern software systems is becoming more and more difficult due to their increased complexity. In addition to classic functional and quality requirements, such as performance, the number of data flow and security related requirements steadily grows. Decentralized applications commonly make use of modern technologies and paradigms, such as distributed micro services or cloud computing. Due to the decentralization of the software resulting from these approaches, the data flow via public networks increases inherently. This however conflicts with the increasing privacy and security related requirements posed to software systems. With recent legal regulations, such as the *General Data Protection Regulations* of the European Union [8] which came into force in mid 2018 this is becoming even more important. Companies not conforming to the required privacy and security standards risk being fined high charges.

In order to aid software developers with meeting the imposed data flow requirements several approaches have been proposed. Many of such approaches try to uncover potential security issues by performing automated analyses through the inspection of the source code. This way, formal verification tools can help ensuring security requirements such as non-inference [28]. However, such source code based approaches often operate on a very fine-granular level which can impose a high manual overhead for certain security analyses.

In contrast model based approaches on architecture level try to minimize the manual efforts required by performing analyses on a higher abstraction level. In addition such approaches often can already be applied at design time and therefore can uncover issues prior to the implementation. Many such approaches enhance existing models used during development and maintenance with data flow and security related information. An example for such an approach is UMLSec [16] which provides an extension to the UML meta model [20]. As these UML models are still code-centric, an extension of the Palladio Component Model [27] has been proposed for modeling these concerns on architecture level. Both approaches provide the means to specify data flow requirements and constraints in an intuitive manner for developers. This however comes with the downside that the corresponding model elements contain much scenario specific semantic information. For

this reason, an automated evaluation of data flow constraints is only partially possible or requires per-usecase specialized analysis tools.

1.2. Contribution

The goal of this thesis is to ease the evaluation of data flow constraints based on models at architecture level. While there are several approaches for enhancing models with information about data flows, their automatic analysis capabilities are currently limited. In this thesis we propose a model which allows an easy formulation of automated data flow constraint analyses based on the logic programming language Prolog [3]. This model is designed to have a concretely specified semantic which is easily understandable by machines. In addition we provide a translator for transforming model instances to Prolog programs. We chose logic programming for performing the analysis as it removes the burden of having to implement an analysis algorithm. Instead logic programming allows us to focus solely on the analysis logic in this thesis. As logic programming language we decided to use Prolog as it is very wide spread and offers a wide range of features. In our work we design a Prolog API for the easy formulation of constraints. This API is exposed by programs generated through our approach. Through this API the resulting Prolog representation of the system allows an automated ad-hoc analysis using queries based on Prolog terms. As a result of this analysis, our approach will either verify that all constraints are held or it will provide sample call traces which lead to constraint violations.

The meta model we propose is not designed for allowing an intuitive manual definition of the system. Instead we designed it so that in future work models of existing approaches, such as the data flow extension of Palladio [27], can be translated into our proposed model. The process of this translation is expected to remove implicit semantic information by translating it into explicit requirements understandable by a machine.

For the design of our approach a central aim is to maximize the performance of both the translation to a logical program as well as the evaluation of the resulting program. Hereby, we propose and analyse Prolog optimization patterns for an efficient mapping of concepts such as parameter passing and stack management to logical programs. We provide an extensive evaluation of the performance of the overall approach as well as of the effectiveness of the proposed optimizations.

1.3. Overview

In Chapter 2 we start with an introduction to the concepts required for the understanding of this thesis. Afterwards in Chapter 3 we define the two main data flow constraint analysis scenarios on which we focus for the design of our approach. Next in Chapter 4 we outline related work. We introduce our proposed meta model in Chapter 5 and the corresponding translation to Prolog code in Chapter 6. In Chapter 7 we show our proposed performance optimization for Prolog. Afterwards we experimentally evaluate our approach in Chapter 8. Finally in Chapter 9 we summarize the results of our work and outline potential future work.

2. Fundamentals

In this chapter we briefly introduce the concepts required for the understanding of this thesis. In Section 2.1 we give a short introduction to the field of software modeling. Afterwards in Section 2.2 we outline the key concepts and properties of the logic programming language Prolog.

2.1. Software Modeling

In software engineering, models are often used as an abstraction of software. Hereby, the elements of which such model instances are built are itself defined in models, which are called meta models. An example for a set of software meta models is the *Unified Modeling Language* (UML) [20]. While models are often used for the design, documentation and communication during the development of software systems, they also enable analysis in order to predict properties of the system, such as the performance or confidentiality.

Data flow models are commonly used to formalize how information flows through a software system. In the proposed thesis we will use the terminology and model proposed by Seifermann et al. [27]. In this terminology, a data flow typically consists of a *source*, a set of *processing operations* and a *sink*. Sources produce information, for instance by Users entering personal information. This information flows through the system where it is possibly accessed and modified, which is modeled by the processing operations. Finally, the information leaves the system, for example in form of a web page or as record in a database. This is represented in the model by sinks. Note that the set of sources and sinks does not have to be disjoint, for example a database can serve as a sink as well as a source.

In previous work [27] an extension for the Palladio Component Model [23] has been proposed for enriching PCM models with data flow information with low manual effort. The core idea is to enrich component methods with specialized data-flow-SEFFs. These SEFFs describe which information is transferred via method parameters and return values as well as how the information is modified. Therefore, the complete flow of information with sources, sinks and processing operations can be derived by combining the data-flow-SEFFs with the assembly model of the modeled system.

2.2. Prolog

Our proposed data flow analysis tool performs its analysis based on the logic programming language Prolog [3]. For this reason we first introduce the basic concepts of this language in Section 2.2.1. In the section afterwards we explain more advanced language and implementation details. These are required for understanding our proposed performance optimizations of our approach introduced in Chapter 7.

2.2.1. Basic Concepts

Logic programming is an alternative programming paradigm to the classic procedural programming. In classic procedural programming languages, a program consists of a set of statements, which are executed one after another. There the developer influences the execution order with control structures, such as loops or if clauses. The processed information typically is stored in variables.

Logic programming is fundamentally different. Instead of specifying statements in their execution order, a logic program consists of a list of *facts* and *rules*. Based on only these data sets, a solver for the logical program can decide whether a specified *goal* can be proven or not. Note that in contrast to structural programming, the solver is capable of doing so without the need of the program to specify an algorithm on *how* the solver finds the proof. Details on the differentiation between structural and logic programming can be found in the work of Kowalski [19]. An example for facts in *Prolog* is given below:

```
owner(hans,bmw).
owner(lisa,ford).
```

This examples states that *hans* owns a *bmw* and that *lisa* owns a *ford*. Hereby *hans*, *bmw*, *lisa* and *ford* are called *Atoms*. Atoms are the objects your program specifies logical rules for. Facts can be seen as special rules, which specify that a given fact is true.

Based on a fact base, now it is possible to specify rules, such as the following:

```
ownsCar(X):-owner(X,_).
```

This rule reads as “given that *X* owns the car _, then *X* in general owns a car”. Hereby, *X* and _ are *variables*. The underscore _ is an anonymous variable. With the facts and rules specified, the logic program can now already be used for deduction. System prompts can be used to query the fact base. In this work we use the syntax of SWI-Prolog [29] where prompts are started with the symbols “-?”. For example, executing the following system prompts will yield the results shown beneath them:

```
?-owner(lisa,ford).
true.
?-owner(lisa,bmw).
false.
?-ownsCar(lisa).
true.
```

The first prompt is true, as there is a fact exactly specifying that *lisa* owns a *ford*. The second one is false, as there is no fact or rule, with which it can be proven that *lisa* owns a *bmw*. The third examples evaluates to true, as the Prolog implementation is capable of deducing *ownsCar* based on its facts. By setting *X* to *lisa* and *Y* to *ford*, *ownsCar(lisa)* can be deduced form the fact *owner(lisa,ford)*.

This deduction is performed using an algorithm similar to a depth-first search. A critical step in this algorithm is the *unification* of terms. The goal of unification is to find variable bindings to make two terms identical. For example, consider the two terms **owner(lisa,X)** and **owner(Y,ford)**. By binding *X=ford* and *Y=lisa*, these terms are made identical. An in

detail explanation of the algorithm to find such unifying variable bindings can be found in the book of Max Bramer [3, p. 31–35].

In order to prove goals such as the ones shown in the listing above, the Prolog interpreter has to systematically explore the rule and fact database. Consider for example that we want to prove the goal **owner(X,bmw)**. In other words, we want to find all persons who own a BMW. The interpreter now looks through all facts and rules in the order they are stated in the program, and tests whether these can be unified with the goal. If this succeeds for a fact, such as **owner(hans,bmw)**, then a solution has been found.

If however the unification succeeds for a *rule*, the Prolog interpreter has to step into the right side of the rule and has to prove its truthness. The proving of this subgoal works exactly the same way as for the initial goal: Again, we scan the rule and fact database for unifiable definitions. If the proof of this subgoal does not succeed, the interpreter has to perform *backtracking*: In order to continue the search of unifiable rules and fact for the initial goal, all variable bindings performed during the proofing of the subgoal have to be undone.

For this case we consider the goal **ownsCar(P)**. In our database, only one definition unifying with this goal exists: **ownsCar(X):-owner(X,_)**. This results in the variable binding **P=X**. Next the interpreter has to find solutions for the subgoal **owner(X,_)**. This yields the two solutions **P=X=hans;_=bmw** and **P=X=lisa;_=ford**. After these two successes, the subgoal fails. This means, that backtracking is performed and the variable binding **P=X** is undone. Afterwards, the interpreter continues to scan for terms unifying with **ownsCar(P)**, which also fails. This means, that all solutions have been found. Again, a more detailed explanation of this algorithm can be found in the book of Max Bramer [3, p. 39–50].

A noteworthy property of Prolog is that it is very simple to write correct logical programs as the language specifies the implementation on how to find proofs. However, it can become quite difficult to write *efficient* programs due to the nature of the depth-first search used by the solver as introduced above. For example, Prolog specifies that the solver examines the rules in the order they appear in the program. Therefore, a logical program with a large number of rules where the rule leading to a successful proof comes early can be expected to terminate much faster than a program where the rule comes last. In addition, solver dependent factors can influence the performance: For example, a solver might decide to index the rules based on the first argument in case it is an atom. This mechanism is explained in more detail in Section 2.2.4. Due to this fact, the order of the arguments can have a noticeable impact on the performance.

2.2.2. Cut Predicate

The built-in cut predicate of Prolog is a feature which allows to prevent backtracking. It is best explained by an example:

```
predA(a).
predA(X):-!,fail.
predA(b).
```

```
predB(X):- predA(X).
predB(c).
```

In this example, the cut predicate written as exclamation mark appears in combination with the fail predicate, which is a common combination. The fail predicate simply causes the current goal to fail when reached.

The cut predicate prevents the interpreter from backtracking for the current goal when it is encountered. Consider that we ask the interpret to prove **predA(X)**. The interpreter finds a first solution with **X=a**. Afterwards, he examines the rule containing the cut predicate. The cut predicate succeeds to be proven by definition, therefore the interpreter reaches the fail predicate. As the fail predicate always fails, the proof of the goal using this rule does not succeed. Without the cut predicate, backtracking would happen and the interpreter would find the next solution with **X=b**. However, the cut operator prevent the backtracking from happening, therefore no additional solutions are found.

The cut operator only prevents backtracking for the current goal, this is not the case for any parent goals. Consider we ask the interpreter to prove **predB(X)** instead. This immediately leads to **predA(X)**, which yields only **X=a** and not **X=b** as result due to prevented backtracking. However, as the cut appeared while proving the goal **predA(X)**, it does not prevent the backtracking for the parent goal **predB(X)**. For this reason **X=c** is found as additional solution in this example.

Further examples and explanations of the cut predicate can be found in Chapter 7 of Max Bramers book [3, p. 99-108].

2.2.3. Negation

The logical negation is a commonly required operator when specifying formal constraints. Prolog does come with the negation operator \+, which however does have a different semantic: The interpreter succeeds to prove a negated term **\+ myPredicate(...)**, if the term is *not provable*. This means, that the negation is true, if the goal **myPredicate(...)** cannot be proven as true. For example, given the fact and rule database from Section 2.2.1, we can execute the following queries:

```
?-\+ owner(lisa,bmw).
true.
?-\+ owner(lisa,ford).
false.
```

So far, the behaviour looks the same as for the mathematical logical negation. Where it differs is when unbound variables are present in the negated term:

```
?- \+owner(X,bmw).
false.
```

Based on the logic semantic of a negation, we would expect the query to return us all people who do not own a bmw. This means we would have expected the query to succeed yielding the binding **X=lisa**. What prolog does during the evaluation of this query is that it tries to prove the term **owner(X,bmw)**. This does succeed for the binding **X=hans**.

Therefore, because the Prolog negation is defined based on the not-provable semantic, the negation returns *fail* as result as the term could be proven.

It however is possible to mimic the behaviour of a logical negation using the not-provable negation of Prolog. This can be done by simply binding all unbound variables prior to executing the negation. In our example we first need to define a predicate in order to make it possible for Prolog to know which atoms are persons:

```
person(lisa).
person(hans).
```

This predicate can be employed as *generator*, meaning that it can be used to instantiate a variable to all possible person atoms. Now we can formulate the logical negation as follows, which yields the expected result:

```
?- person(X), \+owner(X,bmw).
X=lisa
```

Note that this approach can be very expensive performance wise: The Prolog interpreter has to try to prove the negated term for every variable combination which are instantiated using the generators. Depending on the value space of the variables, this can easily lead to a very long runtime.

2.2.4. Predicate Indexing

In Section 2.2.1 we briefly described the basic algorithm for proofing goals in Prolog. We explained that it is required to scan the database for rules and facts which unify with the goal. This can naively be implemented using a linear search where we simply try out every definition. This approach does however not scale well when the size of the program increases. In order to optimize this search for unifiable rules and facts *indexing* mechanism are required which allow a quick lookup of candidates.

We consider again the example predicate **ownsCar(P,C)** from Section 2.2.1, but we assume that the program contains much more facts for persons and their cars:

```
ownsCar(lisa,ford).
ownsCar(lisa,vw).
ownsCar(fritz,audi).
...
ownsCar(hans,bmw).
ownsCar(hans,opel).
```

Assume that we want to lookup the cars owned by hans: **ownsCar(hans,X)**. The more facts we have about people and their cars, the longer the linear search approach takes for finding the solutions for this goal.

An observation that has been made early, is that often rules and facts are defined with their first argument being an atom and not a variable. Similarly, goals as **ownsCar(hans,X)** also often have an atom as first argument specified. Based on this observation *first argument indexing* has been proposed early [32] and can be found in almost every modern Prolog interpreter.

The idea of first argument indexing is simple: The database of facts and rules is realized as a hashtable. Hereby, the combination of the predicate name, its arity and the atom used as first argument is used as lookup key for the hashtable. In the following example we solely focus on the atom argument as hash key as we only use one predicate:

```
%---- Hashbucket for 'lisa'
ownsCar(lisa,ford).
ownsCar(lisa,vw).
%---- Hashbucket for 'fritz'
ownsCar(fritz,audi).
...
%---- Hashbucket for 'hans'
ownsCar(hans,bmw).
ownsCar(hans,opel).
```

When asking the Prolog interpreter to proof the goal **ownsCar(hans,X)**, he can simply inspect the rules in the hashbucket for *hans* and omit inspecting all other rules. This leads to a significant performance improvement.

However, this indexing approach quickly reaches its limits. For example, queries where the first argument is a variable, such as **ownsCar(P,bmw)** cannot make use of this index. In this case linear search has to be used as fallback again. Another example where first argument indexing provides no benefit is if there are very many rules with the same atom as first argument, as this leads to big hashbuckets.

For this reason, more sophisticated indexing approaches have been proposed [7, 30]. In contrast to first argument indexing, these approaches greatly vary from Prolog implementation to implementation.

3. Data Flow Scenarios

The goal of this thesis is to design and implement a tool for the automated detection of violations of data flow constraints. As the range of such scenarios is very wide, we focus on two scenario classes for the design of our approach: Access Control and Geolocation Privacy Restrictions. In this section we provide a general introduction to these two scenario classes.

3.1. Access Control

A traditional problem in software security is the management of the access rights of different parties to resources in a multi-user environment. One of the most known examples for an access control systems is the Unix file permission management. Other variations of the access control systems also exist, for example the Bell-LaPadula Model [2] which tries to prevent the flow of flow of information classified as “high” to information classified as “low”.

In our approach we define the considered Access Control problem as follows: Every party which accesses restricted data performs its access with one or more *Roles*. In our case, this means that every software component of the system-under-investigation has such a role assigned. In turn every datum is annotated with a set of *Authorized Roles*. In this basic form, only components whose role is present in the set of authorized roles are allowed to access this datum. The set of authorized roles is not static. As the datum flows through the system, it may be altered. For example, access rights may be granted or revoked as the datum is processed by different operations. This model can be further refined by specifying the type of access rights. For example, common types of access rights are *read* and *write* access. In this case, there is a separate set of authorized roles for each type of rights attached to each datum.

As simple example for a system with access control restrictions is the TravelPlanner case study[1] of the IFlow project [18]. The TravelPlanner application in the case study is modeled as a mobile application which allows the user to plan their trip, including the direct booking of flights. The data flow of this booking transaction illustrates how the authorized roles of a datum can change during processing.

[1] `https://swt.informatik.uni-augsburg.de/swt/projects/iflow/TravelPlannerSite/index.html`

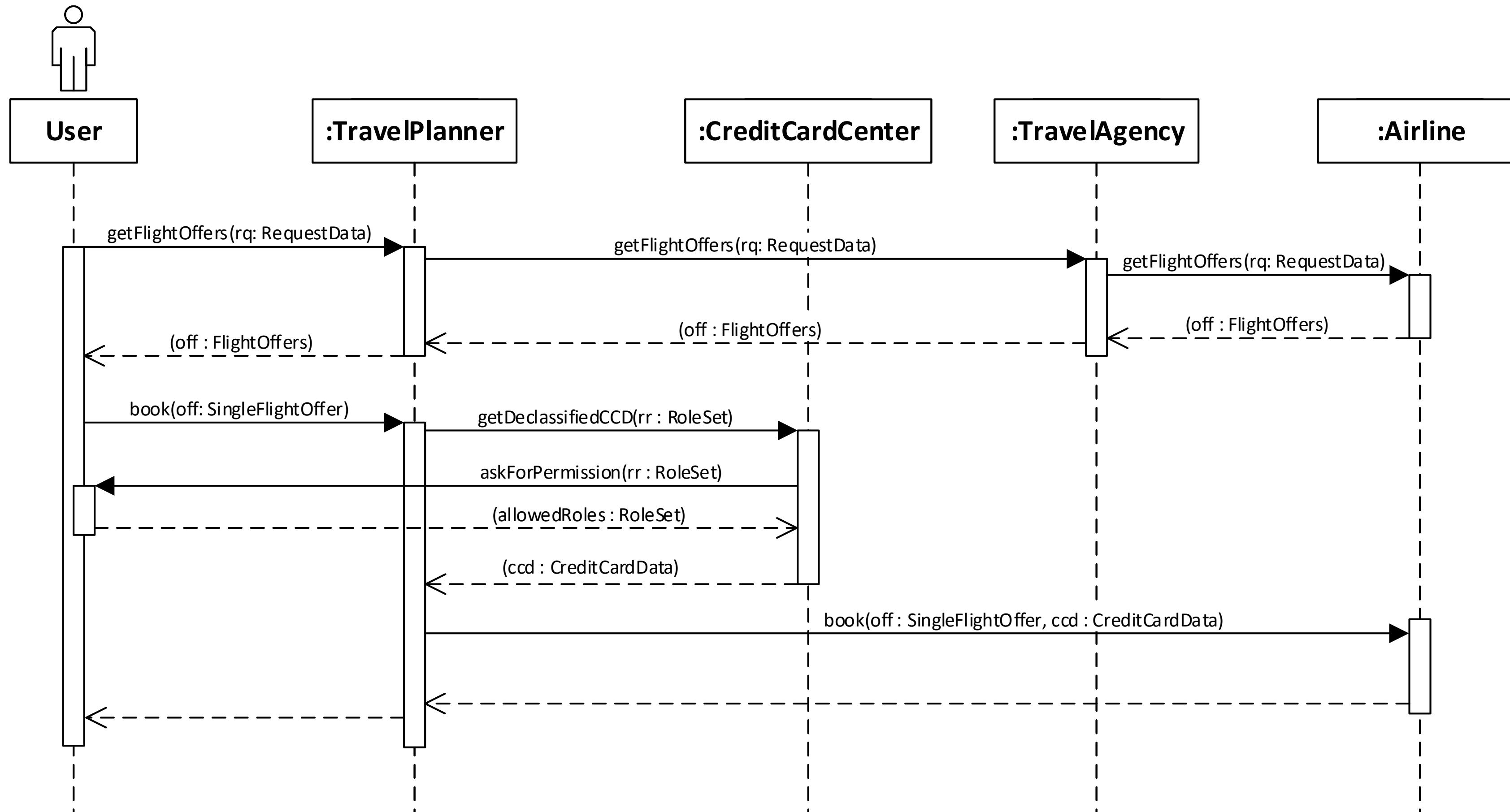

Figure 3.1.: UML sequence diagram of the slightly adapted booking process of the Travel Planner example.

Actor / System	Role
User	*User*
:TravelPlanner	*User*
:CreditCardCenter	*User*
:TravelAgency	*TravelAgency*
:Airline	*Airline*

Table 3.1.: Roles of the involved Actors and Systems of the Travel Planner example.

Datum	Authorized Roles
rq:RequestData	*User*, *TravelAgency*, *Airline*
off:FlightOffers	*User*, *TravelAgency*, *Airline*
off:SingleFlightOffer	*User*, *Airline*
rr:RoleSet	no access control
allowed:RoleSet	no access control
ccd:CreditCardData	default: *User* after declassification: *User*, *Airline*

Table 3.2.: Involved Data with its access control restrictions of the Travel Planner example.

In the scenario, the user has in addition to the TravelPlanner application the CreditCard-Center application installed which is responsible for managing the access to the credit card data of the user. Initially, the credit card data of the user only has the authorized role *User*, implying that no component is allowed to access it. However, for the booking of the flight, the credit card data must be sent to the airlines booking application. Therefore, with confirmation of the user the *Airline* role is added to the set of authorized roles of the credit card details datum. Afterwards, this datum then can be transferred to the airlines booking application without violating access rights.

Our slightly adapted version of this booking transaction is illustrated in Figure 3.1. The roles involved in this example are *User*, *TravelAgency* and *Airline*. Hereby it is noteworthy that the *TravelPlanner* as well as the *CreditCardCenter* application have the *User* roles as they are installed on the users smartphone. The mapping of the involved parties to their roles is shown in Table 3.1. In addition, the access control restrictions for all of the involved data is shown in Table 3.2.

The flight booking transaction starts with a simple query of the user for all matching flight offers. For this purpose he passes his preferences via the *(rq : ReqeustData)* object to the TravelPlanner application. The request is then forwarded to the TravelAgency, which contacts the Airline(s) to find matching offers. These offers are represented by the *(off : FlightOffers)* object. This object is returned as response to the user along the same call chain backwards. In this part of the transaction, no access control restrictions are present: Both the request as well as the response are public data, as all involved roles have access rights as shown in Table 3.2.

This is not the case for the second part of the booking transaction: As the user proceeds to select and book his flight, access control restrictions are involved. After the user has

received all possible flight offers, he selects the one he would like to book in the *(off : SingleFlightOffer)* object. This object may only be accessed under the *User* or the *Airline* role, as the Airline needs to know which flight is booked. However, the TravelAgency is not allowed to learn which flight the user selected and is therefore not authorized.

The booking starts with the user sending his selected flight to the TravelPlanner application. The TravelPlanenr applications then forwards the selection combined with the credit card data of the user to the Airline. The credit card data is represented by the *(ccd : CreditCCardData)* object. However, the TravelPlanner application does not have access to the credit card data, as it is managed by the CreditCardCenter application. In addition, it underlies very strict access control restrictions: The credit card data normally may only be accessed by the user as shown in Table 3.2.

For the booking to proceed, the credit card data has to be declassified first to allow it to be accessed under the *Airline* role. For this purpose the TravelPlanner sends a request for the declassified credit card data to the CreditCardCenter application. As payload the request contains a role request *(rr : RoleSet)*. This RoleSet describes which roles request access to the credit card data. The CreditCardCenter application in turn asks the user for permission for declassifying the data for these roles. The user then responds with *(allowed : RoleSet)* which contains the roles for which the user has granted access. Then the CreditCardCenter application returns the credit card data as *(ccd : CreditCardData)* to the TravelPlanner. This credit card data has then been declassified for the roles which where allowed by the user as shown in Table 3.2. So in this case, the credit card data would be authorized for the roles *User* and *Airline*. As last step the TravelPlanner now sends the credit card data as well as the selected flight to the Airline to book the flight.

3.2. Geolocation Privacy Restrictions

The definition of the Geolocation Privacy Restrictions scenario type is based on the scenario presented by Seifermann et al.[27] as well as the work of Weimann [33].

This scenario class is based on the observation, that often legal aspects imply geolocation restrictions on data flows. For example with the *General Data Protection Regulations* certain personal data is only allowed to be processed or stored within the European Union or certain countries with equivalent privacy regulations. So in general, this scenario class restricts data flows based on *(a)* the type of the data being transmitted and *(b)* the geolocation where the data is being processed or stored.

For the classification of the data based on its type, a sensible abstraction is to define confidentiality levels for the data. We use the definition of these levels as well as the general privacy definition from the work of Weimann [33]:

- **Type-0: Personal Information**: Data which relates directly or indirectly to personal information.

- **Type 1: Personally Identifiable Information**: This data does not directly contain personal information. However, when combined with other Type-0 or Type-1 data, personal information can be completely or partially reconstructed.

- **Type-2: Anonymous Data**: Data which even when anaylzed combination with any other data cannot provide any personal information.

For the classification of geolocations, the two categories *safe* and *unsafe* are used. Hereby, data of any of the types presented above is allowed to be processed or stored in *safe* geolocations. In *unsafe* locations, it is never permitted to process or store *Type-0* data while *Type-2* data does not underlie any restrictions.

A special case is the *Type-1* data: When deployed individually in an *unsafe* geolocation, no privacy violatiosn are induced according to the definition of *Type-1* data. However, when *Type-1* data of different sources are processed or stored together in the same location, it is possible that *Type-0* data can be derived. This problem is known as *joining data streams* [33]. In conclusion two flows of *Type-1* data are only allowed to be deployed together in an unsafe location, if their information origins from the same *Type-1* data source. This restriction implies that both data flows effectively contain the same information, which prevents teh derivation of *Type-0* data.

In our work we use an online shop scenario as running example for geolocation data flow restrictions. This scenario is based on an example given in Section 3.1 of the work of Weimann[33]. Figure 3.2 illustrates the interaction of a user with the example online shop system. The privacy level of the data passed between the user and the components is indicated by its text color.

The user in the example first visits the page of a product and afterwards performs a checkout of his shopping cart. For these transactions he interacts with the *ShopServer* component. He begins by requesting the details page for a product. The *ShopServer* generates the product page which includes the recommendations in the form “user who have bought …also bought … ” which are common for online shops. In addition, the *ShopServer* generates transaction logs for monitoring purposes. The log data is written into a log database illustrated by the *LogDB* component. The transaction log entry is classified as *Type-1* data.

Afterwards, the user performs a checkout of his cart. For this purpose he transmits his shopping cart as well as his customer information, such as his address, to the *ShopServer*. Both these data are *Type-0* data. His customer information is then stored in the *UserDB* component. The shopping cart provided by the user is not directly stored, but instead is transformed into an *AnonymizedOrder*. This anonymized order does not contain any reference to the user, it only consists of a timestamp as well as the bought products. It is used to update the recommendation model managed by the *RecommendationSystem* component.

However, the *AnonymizedOrder* is not fully anonymous, it is classified as *Type-1* data instead. When combined with the *TransactionLog* generated during the checkout transaction, personal information can be derived: The *TransactionLog* also contains timestamps. This allows a correlation with the timestamp of the *AnonymizedOrder*, which in turn can lead to the order being relatable to a shop user.

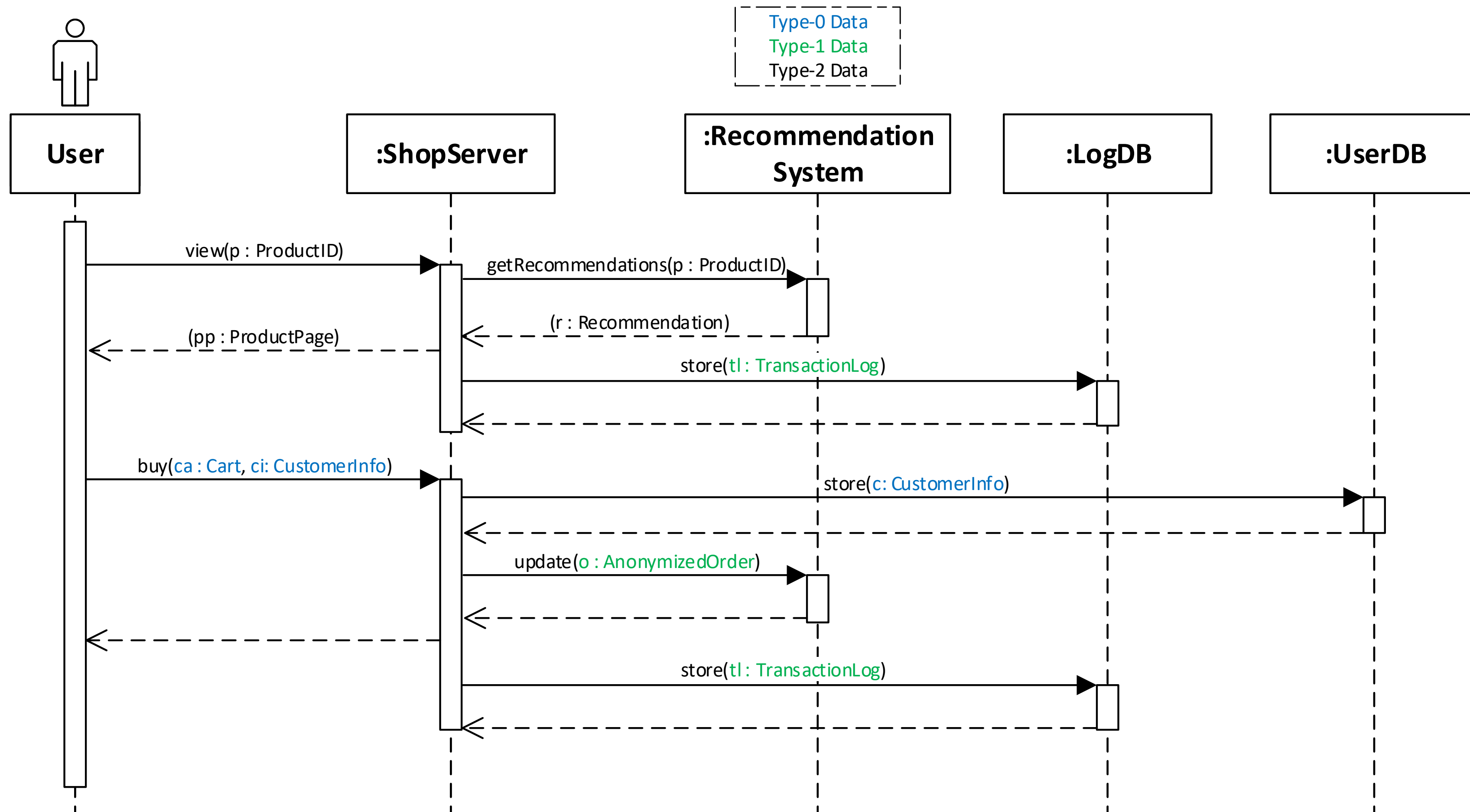

Figure 3.2.: UML sequence diagram of the interaction of a user with the example online shop system.

We chose this example as it illustrates the problem of *joining data streams*: The *RecommendationSystem* as well as the *LogDB* do not have access to *Type-0* data. Therefore, they can be deployed in *unsafe* locations individually. However, they are note allowed to be deployed in the same *unsafe* location, because then their data can be correlated to derive personal information.

The presented example is a very basic instantiation of this scenario. In this case, constraint violations can be trivially detected by inspecting the edges of the data flows in combination with the flowing data types. However, the complexity increases when taking additional factors into account: For example, in the work of Seifermann et al.[27] an exception is made for the restriction of the flow of private data: In their scenario personal data is also allowed to flow across the borders if it is encrypted. However, whether the data is encrypted depends on the processing performed in the application server before transmitting the data. In this case the analysis becomes non-trivial.

4. Related Work

In this chapter we give an overview of existing work related to our approach. First, in Section 4.1 we show the differences to other approaches in the field of data flow analysis. As the scalability is an important aspect for our approach we outline other work related to the performance of Prolog in Section 4.2.

4.1. Data Flow Analysis

While the UML meta model [20] is widely used in the field of software engineering, in its core it does not include means for modeling data flow security properties of software systems. For this reason, UMLSec [16] has been proposed as an extension for UML to fulfill this purpose. In UMLSec, security is modeled using UML Stereotypes and UML Tags paired with constraints. Using these means, the properties of the system as well as adversaries can be modeled.

In addition an automated analysis for such models has been proposed [17]. However, this approach requires the definition of a specialized constraint analysis on a per-scenario basis. A list of available implementations for specific checks can be found online for the UMLSec analysis tool *CARiSMA*[1]. In contrast to this, our approach aims to provide a more generic analysis platform where writing a full-fledged analysis for each scenario is not necessary.

Other approaches, such as JOANA [12] analyse data flows on code level. JOANA analyses Java bytecode to ensure non-inference. Non-inference is a data-flow requirement, that no information about data classified as secret may leak into data classified as not-secret for example through computations. In contrast to this, our proposed approach operates on architecture level. As a result our approach requires manual modeling, it however allows the analysis of additional data flow constraints different from non-inference.

Another data flow modeling language that has been proposed is EDDY [4]. EDDY has been designed for managing privacy and security requirements of multi-tier applications. The goal is to ensure that formulated requirements are met across all tiers, where the different tiers are owned by different companies or parties in general. In their work, facebook is used as running example in the role of a platform provider for facebook games by third-party companies. The formulation of such constraints is performed via description logic. The main difference to our proposed approach is that EDDY tries to check the compatibility of services based on the requirements defined in their contracts. Our approach in contrast formalizes the system structure and performs the data flow analysis there.

[1] `https://rgse.uni-koblenz.de/carisma/checks.shtml`

An approach which operates on both runtime and model level is iObserve [13]. In general, iObserve aims to support the software evolution process of cloud applications. This is done by continuously monitoring the target application and updating corresponding models based on the monitoring data. In further work iObserve was extended to detect data flow violations based on the deployment geolocation [33]. The main difference to our approach is that the work of Weimann does not analyse data flows, but instead uses the structural features of components to find deployment constraint violations.

Many approaches for analysing data flows focus on mobile applications. TaintDroid [11] has been proposed for tracking the flow of privacy-sensitive data within android applications. The idea of this approach is to detect the flow sinks of privacy-sensitive data, such as the geolocation or data from the contact list. Therefore, the approach is similar to ours as it allows to analyse a specialized constraint: The detection of privacy relevant data leaving the application. Our approach however aims to be more generic, so that different types of constraints can also be evaluated.

A similar approach for non-mobile systems is Privacy Oracle [15]. However, in contrast to TaintDroid this approach does not modify the code. Privacy Oracle treats the monitored system as a black-box and only monitors the network traffic of it. In this work the authors propose a testing technique for correlating perturbations in the input data of the system with perturbations in the output of the system for detecting privacy leaks. This allows for example to detect the unencrypted transmission of data classified as sensitive. Again however, this approach focuses on specialized set of constraints and scenarios, whereas our approach aims to be more generic.

4.2. Prolog Performance

We decided to use the logic programming language Prolog for the implementation of the analysis for our approach. As a central aspect of our work is the scalability, it is therefore necessary to investigate which factors influence the performance of Prolog. In Chapter 7 we introduce antipatterns we discovered when generating Prolog code and how they can be tackled.

Unfortunately we found only few scientific sources which examine the performance influence factors of Prolog. Most of the work we found focuses on how to implement a fast Prolog interpreter and not how to write fast Prolog programs. An example for such a guideline on how to implement an efficient Prolog interpreter is the work of Van Roy et al. [31]. Even though the paper is very old it can be used for deducing how to efficiently write Prolog code. For example it is shown how first argument predicate indexing is implemented, which in turn allows to deduce how Prolog code can be written to make use of this mechanism. In addition, scientific resources are also available for advanced mechanism such as the indexing approach of the ECLiPSe interpreter [26].

A good non-scientific guideline on writing efficient Prolog code is provided on the homepage of Markus Triska [10]. There general rules are given to improve performance, for example on how to structure predicates to make use of the implemented indexing mechanisms.

5. Data Flow System Model

In Chapter 4 we outlined the limitations of existing data flow constraint analysis approaches: in most cases the models are designed for an intuitive high-level representation of data flow constraints instead of for an efficient automated analysis. Therefore, often specialized per-scenario analysis tools or manual analyses based on domain knowledge are required for analysing the data flows of systems regarding constraints. For this reason, we define our own data flow system model in this chapter, which supports automated analysis. In contrast to existing models, our data flow system model is designed to be translatable to an efficient Prolog program. At the same time, we aim to keep our model generic enough so that a wide range of scenarios can be represented using it.

Our model is not designed to be manually defined by developers. Instead, it is designed in a fashion that it can easily be derived form existing data flow modeling approaches, which in turn can be manually defined in an intuitive way. We expect several benefits from our approach: Firstly, our model enables a consistent way of defining the input data for data flow analyses. This consistency in turn eases the definition of different analyses. Secondly, this reduces the coupling between the system definition and the analysis.

However, for this idea to be applicable, it is required that our analysis runs reasonably efficient. This is ensured through performance optimization we present in Chapter 7 and evaluated in Chapter 8.

We start by giving a conceptual overview of our model in Section 5.1 and Section 5.2. Afterwards in Section 5.3 to Section 5.6 we give a detailed definition of it. Finally, in Section 5.7 we show how the example scenarios presented in Chapter 3 can be represented using our meta model.

5.1. Conceptual Overview

We can identify the following two core requirements for our system model: It *(a)* has to be capable of expressing most of the common data flow analysis scenarios and *(b)* must be translatable into an efficient Prolog program.

Figure 5.1 shows a simplified illustration of an instance of our system meta model. The basic idea is to model data flows very similarly to the typical structure of software: data flows through the system in the form of parameters for operation calls and in the form of their return values. Similarly to the Data Centric Palladio approach, the data is hereby defined by its meta-attributes, not the concrete data values. The available attributes for each type of data are defined in data types. Note that the parameters and operations in our model do not necessarily correspond to variables or methods in the modeled application. Our meta model defines a data flow, whereas the variables and methods of applications are usually defined based on the control flow.

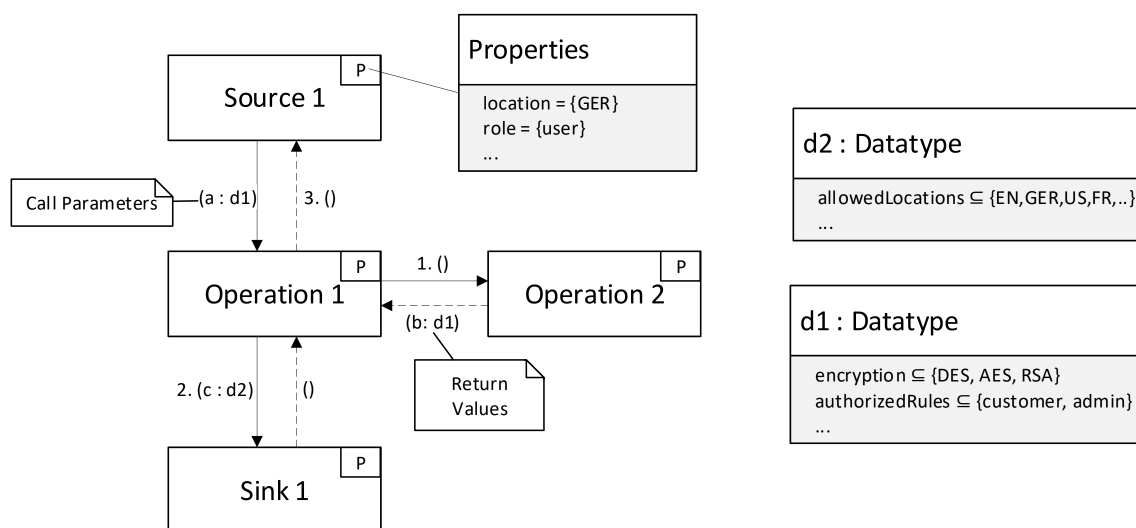

Figure 5.1.: Conceptual illustration of an instance of our meta model

A data type consists of a set of attributes, which represent meta-attributes of the data. Examples for such meta-attributes are the encryption status or the set of roles which are authorized for access. Every attribute has a set of possible values assigned.

An instance of a data type defines which subset of the values of each attribute of the corresponding data type is present. This is explained in detail in Section 5.3. For example, the operation *Source 1* calls *Operation 1* with the parameter a of type *d1*. The data type *d1* consist of the two attributes *encryption* and *authorizedRoles*. These two attributes each define their set of possible values. As the parameter a is an instance of *d1*, it defines which of the possible values are present for each attribute. For example, a could be defined with a.encryption = {} and a.authorizedRoles = {customer, admin}. This instantiation would define a as unencrypted data which is allowed to be accessed by the customer as well as the admin role. In other words each data instance *dat* defines boolean variables in the form of $dat.attribute.value = true/false$ for each of the attributes and the values of its data type. This set of boolean expressions allows for a simple transformation into a logical program, which is presented in Chapter 6.

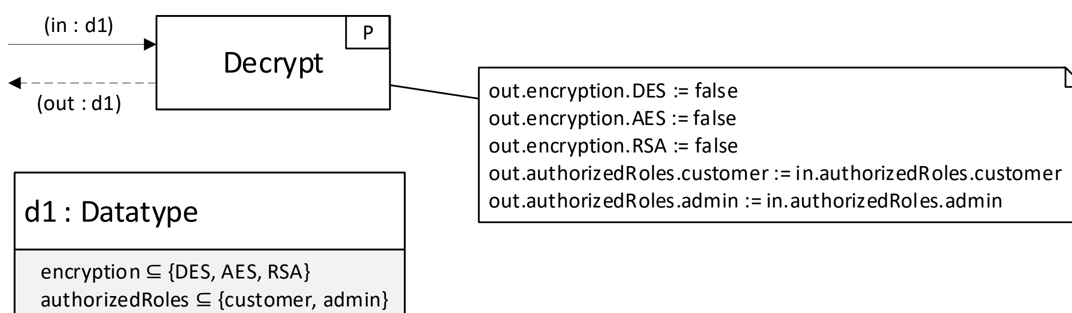

Figure 5.2.: Example of an decryption operation

In our model operations are actions which transform data. Hereby, transformation means that the values of attributes are possibly changed or newly defined. It is even possible to generate data of a different type from the input data. Each operation has a signature, which contains a named and typed list of parameters. The types hereby are data types which have to be defined as explained previously. In addition, each operation also has a well defined named and typed list of return values. The modification of the data an operation performs is formalized using propositional logic. An example for such a definition of a modification is given in Figure 5.2. In this example an operation performing a decryption is shown. For each attribute and value of the return value *out*, a boolean logic formula defines its truthness. This formula is allowed to depend on the values of input parameters to the operation. While in the example only simple assignments are performed, the formulas are allowed to be of any complexity by employing logical operators, such as *or*, *and* or *not*. In addition to depending on input parameters of the current operation, the action may also depend on *Properties* of the operation as well as the return values of previously issued calls to other operations. An additional variable type such terms can depend on are state variables which we introduce in Section 5.2. The idea is that the logical program for evaluating constraints starts with the definition of the return value. The interpreter then traces through the dependencies of the return value definition to prove the constraints.

For example, consider that an operation calls the decryption operation shown in Figure 5.2. When querying the return value of this call, the Prolog interpreter looks at the definition of the return value of the decryption operation. Because this return value depends on the call parameters, the interpreter then looks at the definition of the corresponding call parameters. As these again may depend on other variables, the Prolog interpreter transitively follows the definitions through the system.

Note that this explicitly does not require a trial of all possible call sequences up to their root. Instead, a trial is only required until all dependencies are resolved. For example, consider we wanted to prove that a datum is not encrypted. If we trace backwards through the system and find an "decrypt" operation on our way we do not need to trace further. The reason is that the decrypt operation defines the encryption attribute without any dependencies.

As illustrated by the numbering on the edges of *Operation 1* in Figure 5.1, outgoing calls must have a fixed order. *Properties* are sets of values defined per operation. These are used to model properties of operations which are independent of their input parameters. An example where properties are useful is the modeling of the geolocation of the server which performs this operation. Another example is the modeling of the role under which the operation accesses its data in case of an access control scenario. Note properties are defined as constant and cannot be modified during the execution of the operation.

Operations are allowed to call other operations, as done by *Operation 1* in Figure 5.1. For such a call to take place, the required call parameters of the called operation have to be specified. This is done exactly as for the specification of the return values: for each call, a set of propositional logic formulas defines the truthness of each attribute and value pair for each call parameter. These formulas again are allowed to depend on several variables: the properties of any operation, the parameters with which the current operation was called as well as the return values of other called operations. However, as these calls are

performed in their specified order, the parameter definition of a call may only depend on the return values of previously issued calls. They are not allowed to depend on the return values of calls which are performed after the current one.

5.2. Data Flows without Control Flow

So far, operations have been modeled as stateless: Their computation only depends on input parameters and the return values of called operations. Across different calls of the same operation, no data is shared so far. In addition according to our current definition, the data flow is bound to the control flow. This means, that it is not possible to pass data around without explicitly calling the receiving operation. However, in data flow modeling this kind of flow is not uncommon as illustrated by the example in Figure 5.3.

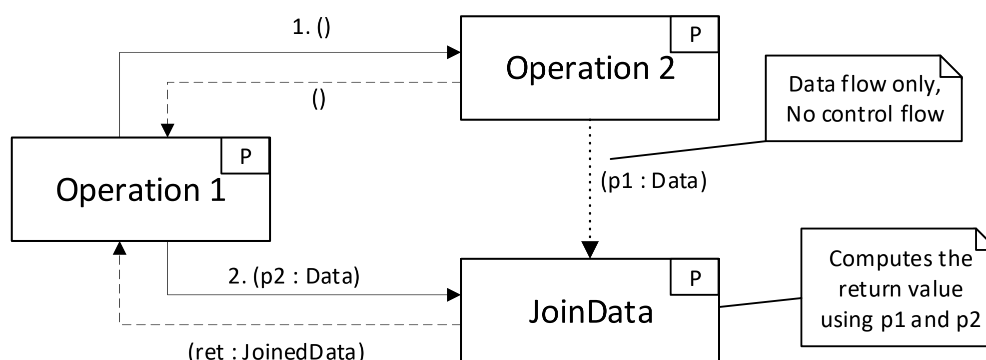

Figure 5.3.: Minimal example of a data joining scenario where a data flow without control flow is required.

In this example the operation *JoinData* joins the data from two different sources. This example can not be modeled yet using the features of our approach we explained so far: we only have defined data flows bound to control flows. However, we can transform this scenario so that it represents the same functionality while being representable with our meta model. This is illustrated in Figure 5.4.

As shown in Figure 5.4, the problematic data flow has been removed. It was replaced with a return value of *Operation2* and an additional parameter of *JoinData*. Instead of directly passing the data to *JoinData*, we attach it to the control flow until it reaches its destination.

This example shows how data flows which are not bound to control flow can be erased in general: For each such data flow, an implicit return value and call parameter is added to each operation. These parameters and return values then represent the datum. However, we decided for usability reasons to include a mechanism into our meta-model for passing data around without being bound to the control flow. Our Prolog translator presented in Chapter 6 performs the task of transforming these flows to implicit parameters and return values.

The mechanism we introduced for such special data flows are *state variables*. Just like operation parameters and return values, they have a type and belong to a certain operation.

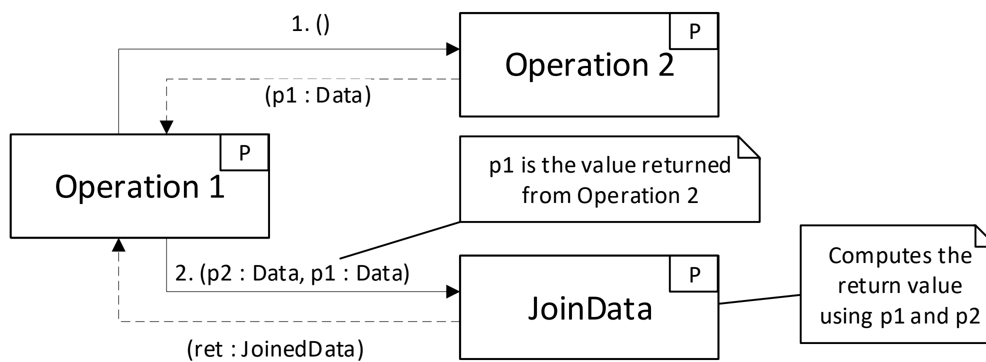

Figure 5.4.: Functionally equivalent scenario to the scenario in Figure 5.3 with all data flows bound to the control flow.

However, in contrast they are globally accessible: They can be written and read at any time by any operation, not only by the operation which owns them. A write on a state variable remains persistent until it is overridden.

With state variables we can now trivially model the example shown in Figure 5.3: *JoinData* defines a state variable for *p1*. *Operation 2* now writes to this state variable when it is called. As the variable is persistent, it can be read and processed by *JoinData* as soon as this operation is executed.

5.3. Type Definitions

In our meta model, the root element *System* acts as a container for all first class entities. Figure 5.5 is an excerpt from the meta model which shows the *System* definition including its contents. Such a system defines which *Operations* it contains as well as the usage of system entry points through *SystemUsage* elements. In addition, it contains typing related elements: *DataTypes*, *Attributes*, *Properties* and their common type *ValueSetType*.

We decided to introduce *System* as a container for first class entities for technical reasons. Apart from this, *System* elements are not relevant for our approach.

As explained in the previous section *DataTypes* are used to model information about a datum on meta level. *Variables* represent operation parameters, return values and state variables. To specify their type they reference a *DataType*. This is explained later in more detail in Section 5.4. As it is common for variables to have the same type, we decided to model *DataType* as first class entities.

DataTypes are defined just as a named set of *Attributes*. *Attributes* hereby model the previously explained meta attributes of data which flows through the system. Examples for such meta attributes are the authorized roles in an access control scenario which are allowed to access the data or the encryption of the data. The encryption attribute example also shows why it is beneficial to model *Attributes* as first class entities instead of being owned by a single *DataType*: Often, *DataTypes* have attributes such as encryption in common. For example, for the geolocation restrictions scenario the customer data as well as the order information can be modeled as different data types. However, depending

Figure 5.5.: Excerpt of the UML class diagram of the system element and its contents

on the context it can make sense for both to specify the encryption. Therefore, it is useful to share an encryption *Attribute* between these to avoid redundancies in the model and later in the generated code.

Properties allow to specify *constant* meta information about operations. From a technical perspective it would have been possible to avoid the definition of separate *Properties*. Instead it would have been possible to reuse the *Attributes* of a system. However, we decided to keep them separated due to the semantic difference: While *Attributes* of data can change as the data flows through the system and is processed by *Operations*, *Properties* are constants attached to *Operations*. Now it can be questioned why *Properties* where introduced in the first place if they are constants. As we expect that instances of our model are generated automatically and not altered by developers, constants could just be directly inlined where they are used. However while we expect that the model is generated automatically, we do not expect the formulation of constraint queries to be done fully automated: For example, all *Operations* could have a *Property* deploymentLocation attached. If this property was not present and instead inlined everywhere it is used, it would be hard to formulate a query such as “For all operations which are not deployed in the EU the input parameters must be encrypted”. When modeling *Properties* explicitly, a simple Prolog generator-predicate can be used for a readable definition of this query.

Without the definition of Properties one would have to manually enumerate all operations which are not deployed in the EU.

The final element of the typing system in our meta model are *ValueSetTypes* which contain a set of *Values*. A *ValueSetType* specifies the type of an *Attribute* or *Property*. The contained *Values* define all possible values of which a certain subset is present for the instances. This is best explained by a simple example: Let's say we have the *Property* "deploymentLocation" and the *Attribute* "origin", which specifies from which country a data originates. One can observe that both share the same *ValueSetType*, which could be named "geolocation". This *ValueSetType* now defines which values are possible geolocations. Depending on the required granularity and countries, the *Values* could be "US", "EU" and "Asia". This example also shows why we decided to model *ValueSetTypes* as first class entities: they are commonly shared between multiple *Attribute* and *Properties*.

In addition, Figure 5.5 shows that *Operations* and *SystemUsages* share a common super type: *Caller*. This super type was extracted due to the fact that both have the ability to call *Operations*. This is explained in more detail in the next section.

5.4. Operations

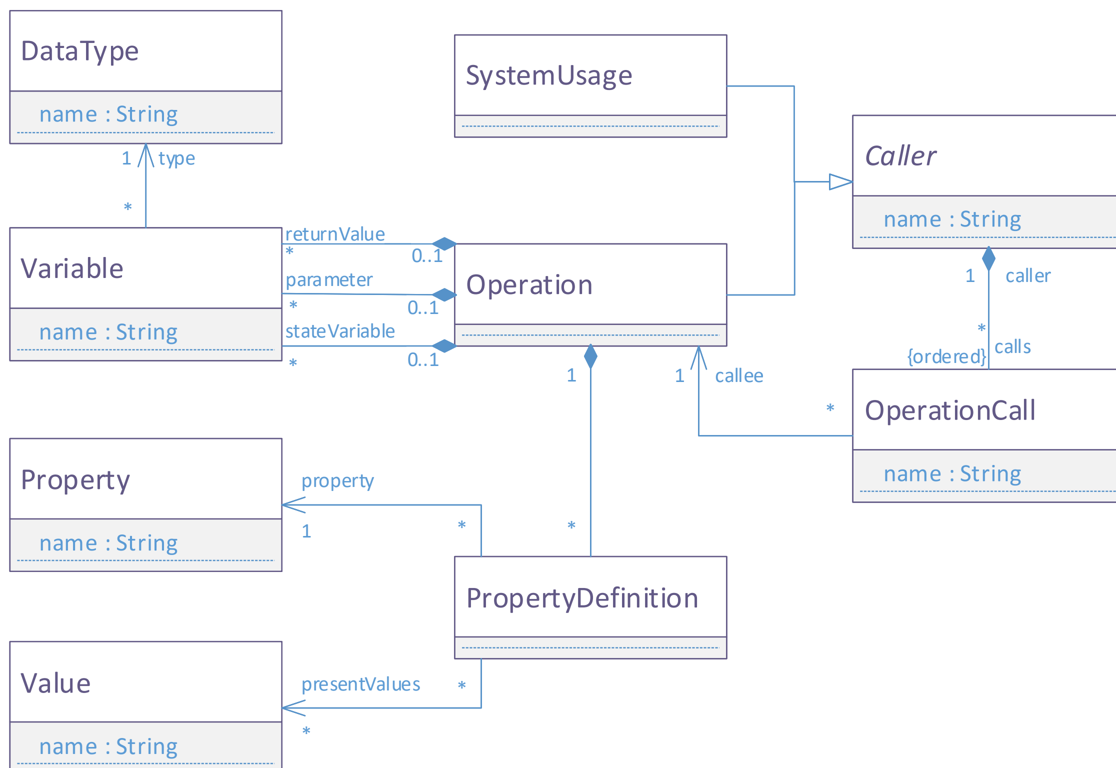

Figure 5.6.: Excerpt of the UML class diagram of the operation class and its related elements

An *Operation* is closely related to functions or methods in traditional programming languages: It is a formal specification on how output data is generated based on input values.

In this section we introduce the structural elements required for defining operations. The elements required for a definition of *what* an *Operation* does are introduced in Section 5.6. An overview of the structural elements required for the *Operation* definition are shown in Figure 5.6

An *Operation* has a signature which defines the required input data and specifies the format of the output data. However, an *Operation* has to be uniquely identifiable in the system by just its *name*. In contrast to many programming languages the signature is not used for this purpose. This requirement does however not restrict the applicability of our approach: As our model is designed as an intermediate model, composite identifiers in the source model can usually be automatically concatenated to a single identifier string for our model.

In our model the signature is represented by the contained *parameter*, *returnValue* and *stateVarible Variables*. The order of the *Variables* hereby does not matter: they are identified based on their name and the context in which they are referenced. The context is hereby defined by a) which *Operation* is referenced and b) whether a return value or a parameter is referenced. A *Variable* is basically a referenceable instance of a *DataType*.

As outlined in the previous section, meta information about *Operations* can be defined using *Properties*. For this purpose, each *Operation* contains a set of *PropertyDefinitions*. A *PropertyDefinition* is associated with the *Property* which is being defined as well as with the *Values* which are present. Hereby, it is required that these *Values* are owned by the *ValueSetType* which is the type of the defined *Property*. Every *Value* of this type which is not present in the *presentValues* association is implicitly set to false for this *Property*. For each *Property*, an *Operation* may contain only one *PropertyDefinition* at maximum.

As previously explained *Operations* and *SystemUsages* share the common super type *Caller*. This type was introduced due to the fact that both have the ability to call *Operations*. In the model a call to an *Operation* is represented by an *OperationCall* instance, which is owned by the *Caller*. An *OperationCall* has to reference which *Operation* is being called, which is done through the *callee* association. Note that the list of *OperationCalls* is ordered and that each *OperationCall* has a *name* which is unique in the list. The list order semantically represents the order in which the calls occur. This is important because for *OperationCalls* which appear later in the list it is possible to reference the return values of previously issued *OperationCalls*.

OperationCalls are named to make it possible to represent call sequences without ambiguities. An *OperationCall* cannot uniquely be identified by the callee and caller, as it is possible for an *Operation* to call another *Operation* more than one time. While in theory it would be sufficient to just use the index of each *OperationCall* within its containing *Caller*, we decided to introduce the *name* field instead. The reasoning behind this decision is that call sequences will often be shown to the user of our constraint analysis tool when constraint violations are found. In such a case, call sequences like

$$(processData \overset{storeUserData}{\rightarrow} storeInDB)$$

are more speaking than:

$$(processData \overset{2}{\rightarrow} storeInDB)$$

Each *OperationCall* also needs to provide values for the parameters of the called *Operation*. An explanation on how this is done is given in Section 5.6.

SystemUsages are used for modeling the entry points which can also act as data sources. They define a sequence of *OperationCalls* which are issued and may have interdependencies in their parameter assignments. Basically *SystemUsages* are *Operations* which cannot be called and which do not have parameters, return values or state variables.

A noteworthy limitation of our approach is that recursion is not supported: This means, that the directed graph formed by *Operations* through their *OperationCalls* has to be acyclic. The reason for this limitation is that as previously explained the Prolog interpreter tries to find proofs by tracing backwards through the dependencies of the variables. However, if the call graph contains cycles, there are cyclic dependencies. This would result in an infinite recursion of the Prolog interpreter.

5.5. Propositional Logic Terms

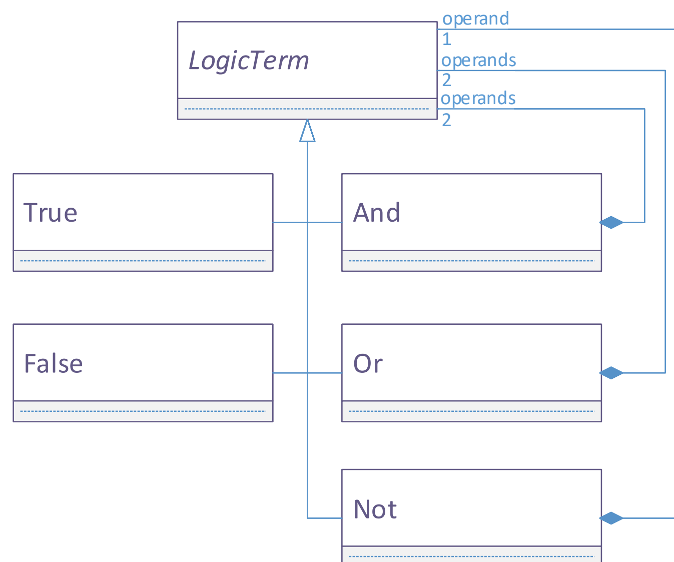

Figure 5.7.: Excerpt of the UML class diagram of the basic propositional logic terms.

In our approach the combination of a variable within its calling context with an attribute of it and one of its possible values can be seen as a boolean variable. Therefore, we employ propositional logic terms for formulating the value of assignments. The operators we use are functionally complete, therefore any truth table can be represented.

In the model we express propositional logic in a tree structure, as shown in Figure 5.7. For this reason, every logic term inherits from the abstract super class *LogicTerm*. A logic term can either be atomic or compound depending on whether it is a root or a leaf in the tree. For the representation of the constants *true* and *false*, the atomic terms *True* and *False* were introduced. As compound terms we added *Not* as logical negation, *And* as conjunction and *Or* as disjunction.

We chose these three compound operators for several reasons: First, this operator system is the typical logical operator system which is also implemented in classic programming languages, such as Java. This allows terms to be easily defined by developers. While we do not expect that it is necessary for developers to alter the logic terms used in variable assignments, logic terms will also be used for formulating the constraints. The latter part can be expected to be done at least partly manually.

A second reason for this choice of operators is that it potentially reduces the amount of negations: Whereas for example the operator systems $\{\wedge, \neg\}$ or $\{\vee, \neg\}$ are functionally complete, they typically require more negations to represent the same formula as when using the system $\{\wedge, \vee, \neg\}$. Negations however can be expensive in terms of performance in Prolog if no special optimization steps are taken. This is explained in Section 2.2.3.

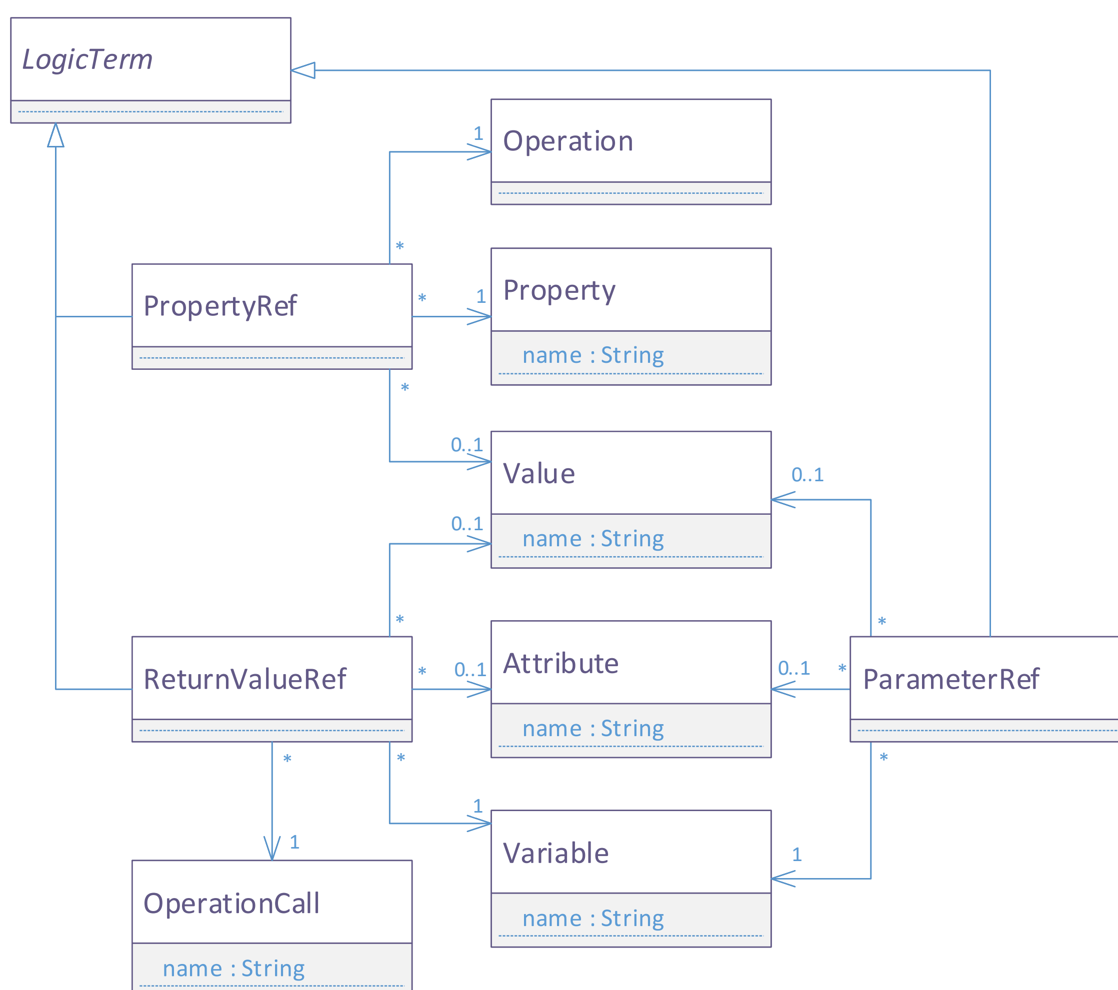

Figure 5.8.: Excerpt of the UML class diagram of the reference types for logic terms.

In addition to basic atomic and compound terms Figure 5.8 shows the second type of atomic terms available: references to parameters, return values and properties. References to return values and call parameters are required as these variables are non constant and context dependent. Property references could in theory just be replaced by their value

with *True* or *False* terms. However, as previously explained it is useful to have the ability to reference properties when formulating constraint queries, which is potentially done manually.

PropertyRefs are simple to define: They just require an association to the *Operation* they refer to, the *Property* to read and the corresponding *Value* of which the truthness shall be queried. It is required, that the *Operation* has a *PropertyDefinition* for this *Property* and that the *Value* given belongs to the type of the *Property*. Note that the reference to the *Value* is optional. If no *Value* is defined, a wildcard is assumed for the value, which means that the actual *Value* read gets chosen based on the context. For example, a wildcard can be used to initialize all values of a variable to false with a single assignment. The wildcarding mechanics are explained in detail in the next section.

A *ParameterRef* is used to reference call arguments of an *Operation*. The concrete value of the parameter depends on from where the operation was called. A *ParameterRef* is defined by a reference to the *Variable*, which must be a call parameter, and the *Attribute* and *Value* to query. The *Operation* which is referenced is implicitly referenced as it contains the *Variable* as parameter. For type safety, it is required that the *Attribute* is part of the *DataType* of the referenced *Variable*. In addition, the referenced *Value* must belong to the type of the *Attribute*.

Return values are referenced very similarly to call arguments through *ReturnValueRefs*. They also have references to a *Variable*, *Attribute* and *Value*. While for the *Attribute* and *Value* the same restrictions apply as when used in a *ParameterRef*, the *Variable* must reference a return value *Variable* instead of a parameter. In addition, the *OperationCall* of which the return value shall be queried has to be referenced. The *callee* of this call has to be the *Operation* which owns the referenced *Variable*.

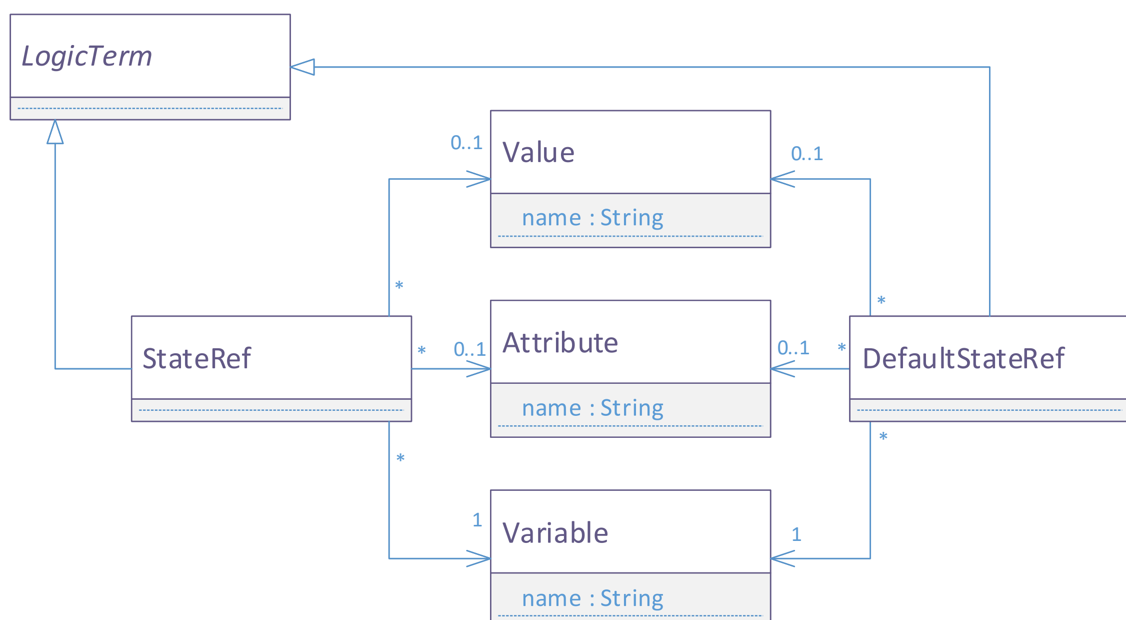

Figure 5.9.: Excerpt of the UML class diagram of the state reference types for logic terms.

Logic Term	**Notation**
And	[firstOperand] ∧ [secondOperand]
Or	[firstOperand] ∨ [secondOperand]
Not	¬[operand]
True	$true$
False	$false$
PropertyRef	pr([operation].[property].[value])
ParameterRef	pa([variable].[attribute].[value])
ReturnValueRef	rv([call].[variable].[attribute].[value])
StateRef	st([operation].[variable].[attribute].[value])
DefaultStateRef	dst([operation].[variable].[attribute].[value])

Table 5.1.: Notation for logic terms. The values in square brackets are replaced with their real values when used.

As shown in Figure 5.9 references to state variables using *StateRef* or *DefaultStateRef* are highly similar to *ParameterRefs*. They also require a reference to a *Variable*, which in this case has to be a state variable. In addition, an *Attribute* and *Value* can be specified, for which the same restrictions apply as for *ParameterRefs* or *ReturnValueRefs*. In contrast to these there is no restrictions on which *Variable* can be referenced as long as it is a state variable. As previously mentioned, state variables of any operation can be read and written at any time.

The difference between *StateRef* and *DefaultStateRef* is that *StateRef* refers to the *current* value of the given variable. This means that the value is returned, to which the variable has previously been set. In contrast, *DefaultStateRef* refers to the *default* value of the variable. The default value is the value a state variable has when it has not been written yet. This default value is defined by the operation which owns the variable, which is shown in the next section.

Note that for all kinds of variable references the reference to both the *Attribute* and the *Value* are optional. Again, if no value for these is given, a wildcard is used instead when generating the Prolog code.

Instead of using UML instance graphs for Logic terms as concrete syntax, we introduce the notation shown in Table 5.1. This notation is the same as the classic notation for propositional logic terms with the addition of a special syntax for property and variable references.

5.6. Variable Assignments

In the previous section we explained how logic terms are represented in our meta model. In this section we introduce how they are used for the specification of the behaviour of *Operations*. For this purpose we employ logic terms to provide definitions for return values, call parameters and state variables.

Figure 5.10 shows the *VariableAssignment* which is used for this task. A *VariableAssignment* can either be owned by an *Operation* or by an *OperationCall*. If it is owned by an

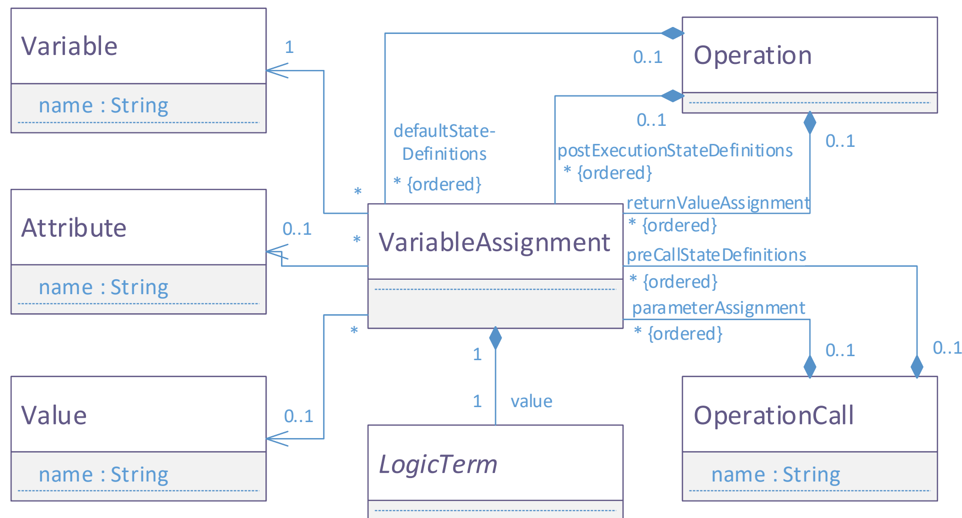

Figure 5.10.: Excerpt of the UML class diagram of the *VariableAssignment* type.

Operation, it is used to define a return value of this operation or a state variable. For state variables *Operations* own two types of assignments. First there are *defaultStateDefinitions* which define the default value of state variables which are owned by this *Operation*. This means that the value defined by these assignments is returned if the corresponding state variable is read before it was written. The second type of state assignments of *Operations* are *postExecutionStateDefinitions*. These assignments define the changes to make to state variables before the *Operation* returns.

In case that a *VariableAssignment* is contained within an *OperationCall*, it is either used to define the value of the arguments passed to the callee for this call or to define state changes prior to performing the call. Both *postExecutionStateDefinitions* and *preCallStateDefinitions* are optional: If none are defined, the value of the state variables are not altered.

In all cases state variables are assigned last. This means that when performing a call, the call arguments are defined first and state variable changes are executed directly afterwards. Similarly, when an *Operation* returns it first defines its return values and changes state variables afterwards. This order ensures that the definitions of return values and call parameters can refer to state variables before they are overridden.

The left-hand side of the assignment is defined by a reference to a *Variable*, *Attribute* and a *Value*. Depending on the container of the *VariableAssignment*, the *Variable* must either be a return value, a call parameter or a state variable. The *Attribute* must be part of the type of the *Variable* and the *Value* must belong to the type of the *Attribute*. The right-hand side of the assignment, the value, is defined by a *LogicTerm*. This *LogicTerm* can be a compound term of any complexity as introduced in the previous section.

For the remainder of this work, we use the following notation for representing assignments:

$$[\text{variable}].[\text{attribute}].[\text{value}] := [\text{logicterm}]$$

The logic term on the right side hereby is noted as described by Table 5.1.

Again, the *Attribute* and *Value* references are optional. If they are not present, they are interpreted as wildcards. We note wildcards using a single asterisk "*" in our notation. We explain the semantic of a wildcards in the various positions on the left-hand side of an assignment in the following using examples:

- input.authorizedRoles.* := $false$
  For the variable *input* set **all** values of the attribute to *false*. This is useful for example for initializing all values of a single attribute before selectively overwriting them.

- output.*.* := $false$
  For the variable *output* set **all** values of **all** attributes to *false*. This is useful for initializing all values of all attribute before selectively overwriting them.

- output.*.EU := $true$
  For the variable *output* set the value *EU* of **all** attributes which have the *ValueSetType* of *EU* to *true*. This is mainly useful for copying certain values from other variables.

To summarize, wildcards can be used to write a single assignment instead of multiple ones. For example if a wildcard is used in place of the *Value*, one could just duplicate the assignment for each *Value* manually. There are several benefits of such a reduction of the number of assignments even though assignments are not planned to be generated or maintained by developers: Firstly, the total size of model instances is reduced. Secondly, we implement an optimization which allows the translation of wildcard rules to single Prolog rules, which therefore potentially speeds up the analysis.

Whereas in the examples above we only used constant logic terms on the right-hand side of the assignments, logic terms of any complexity can be used. The wildcarding mechanism is especially powerful when using variable references with wildcards. Consider the following example:

$$\text{output.*.*} := pv(\text{input.\{A\}.\{V\}})$$

In this example the *Attribute* and the *Value* on the left hand side are wildcards. In addition, the logic term used on the right hand side of the assignment is *ParameterRef* of which the *Attribute* and the *Value* are also wildcards. However, wildcards on the left hand side of an assignment have a different semantic than wildcards on the right hand side: wildcards on the left hand side **quantify** for which *Attributes* and *Values* the assignment should be used. Wildcards on the right hand side **refer** to the corresponding bound values instead. This is comparable to the semantic of the classical mathematical quantors: wildcards on the left side act as universal quantors ($\forall x \in \ldots$), wildcards on the right side refer to the quantified variable (x).

Because of this difference of the semantic we note wildcards on the right-hand side using {A} or {V} depending on whether they refer to the currently bound *Attribute* or the currently bound *Value*. In order to further clarify this, the example above could be unrolled

as follows:

$$\text{output.authorizedRoles.User} := pv(\text{input.authorizedRoles.User})$$
$$\text{output.authorizedRoles.Airline} := pv(\text{input.authorizedRoles.Airline})$$
$$\text{output.authorizedRoles.TravelAgency} := pv(\text{input.authorizedRoles.TravelAgency})$$
$$\text{output.encryption.AES} := pv(\text{input.encryption.AES})$$
$$\text{output.encryption.DES} := pv(\text{input.encryption.DES})$$
$$\ldots$$

Due to the wildcarding mechanism it is possible that multiple rules match for the assignment of certain *Attribute* and *Value* combinations. For this reason, the assignment associations are ordered as visible in Figure 5.10. Assignments appearing later in these overwrite previous matching rules.

As illustrated by the initial examples in this section, the position of wildcards on the left side of assignments has an impact on which *Attribute* and *Value* combinations are possible in an assignment. This is due to the typing system we introduced in Section 5.3. In addition, wildcards in the *LogicTerm* used as value for the assignment also restrict the set of possible combinations. These restrictions are listed in Table 5.2.

Logic Term	Induced Restrictions
$pr([operation].[property].\{V\}$ $pa([variable].[attribute].\{V\}$ $rv([call].[variable].[attribute].\{V\}$ $st([operation].[variable].[attribute].\{V\}$ $dst([operation].[variable].[attribute].\{V\}$	{A} must have the same *ValueSetType* as [property].
$pa([variable].\{A\}.\{V\}$ $rv([call].[variable].\{A\}.\{V\}$ $st([operation].[variable].\{A\}.\{V\}$ $dst([operation].[variable].\{A\}.\{V\}$	{A} must be an *Attribute* of the *DataType* of [variable].
$pa([variable].\{A\}.[value]$ $rv([call].[variable].\{A\}.[value]$ $st([operation].[variable].\{A\}.[value]$ $dst([operation].[variable].\{A\}.[value]$	{A} must have the *ValueSetType* containing [value].

Table 5.2.: Typing restrictions induced by reference logic terms based on wildcards.

5.7. Example Instances

In this section we illustrate how the meta model defined in the previous sections can be used for modeling data flow constraint scenarios. For this purpose we provide two example model instances, one for each of the scenarios introduced in Chapter 3. In Section 5.7.1 we model the TravelPlanner example as access control scenario. Afterwards in Section 5.7.2 we show how our example shop system with its geolocation constraints can be represented using our model.

DataType	Attributes
RequestData	authorizedRoles
FlightOffsers	authorizedRoles
SingleFlightOffer	authorizedRoles
CreditCardData	authorizedRoles
RoleSet	roles

Table 5.3.: DataTypes of the model instance for the TravelPlanner example.

5.7.1. Access Control Scenario

In Section 3.1 we introduced the TravelPlanner example as an instance of an access control scenario. In this section we show how the TravelPlanner system can be accurately represented using our proposed meta model. In addition, we give an informal explanation on how the corresponding access control constraint can be formulated in relation to this model.

5.7.1.1. Model Definition

The model instance for the TravelPlanner example is shown in Figure 5.11. The model elements are visually represented in the same way as in Section 5.1. Operations are shown as white boxes with solid arrows for their out-going calls and dashed arrows for return values. The properties of each operation are shown as annotation. The corresponding call parameters and return values are annotated to each arrow. In addition, each call has a number assigned to specify the order in case of multiple outgoing calls.

The used data types are not defined in Figure 5.11, they are listed in Table 5.3 instead. Every data type underlying access control restrictions has the attribute *authorizedRoles*. The attribute *authorizedRoles* has the *ValueSetType Roles*, which contains the three values *User*, *TravelAgency* and *Airline* representing the corresponding roles from the TravelPlanner example.

During the process of the declassification of the credit card data, the RoleSet type is required for defining for which roles a declassification is requested and in turn granted. For this purpose, the data type *RoleSet* contains the attribute *roles*, which again has the type *Roles*. Data of the type *RoleSet* however does not underlie access control restrictions, therefore the attribute *authorizedRoles* is not present.

In addition to the data being classified with *authorizedRoles*, the operations have to define under which role they access the data. This is done using the *roles* property illustrated by the annotations of the operations in Figure 5.11.

The *VariableAssignments* for call parameters and return values are shown as annotations for each call and return. Hereby, we use the notation for the assignments which we introduced in the previous sections.

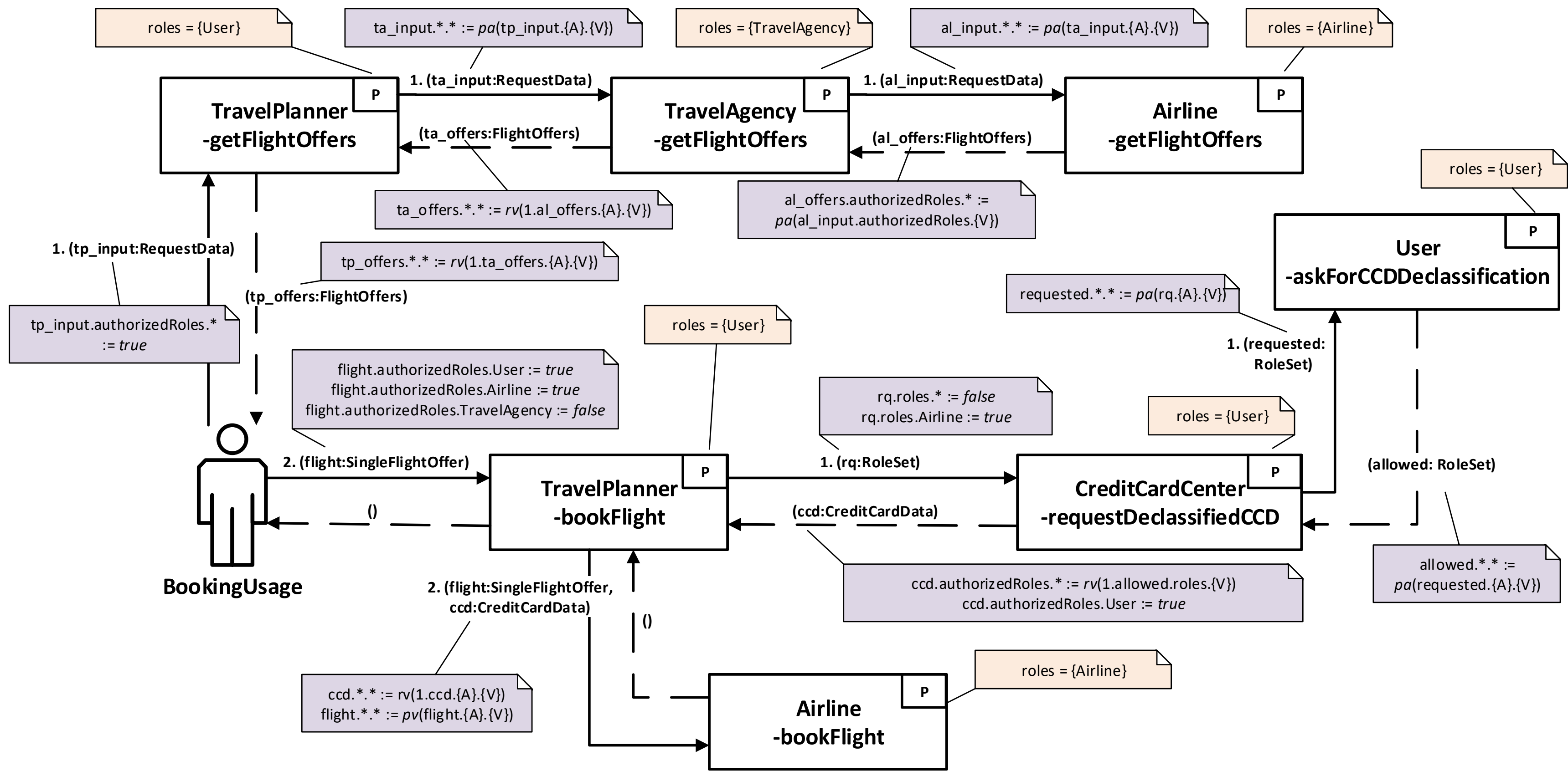

Figure 5.11.: Full system model of the excerpt from the TravelPlanner case study.

The definition of the model for the TravelPlanner example is straightforward: As the user in the example initiates the booking transaction, he is mapped to the BookingUsage pictogram shown in Figure 5.11 as the person pictogram. In the example scenario, the user first requests the set of possible flights from the Airline indirectly via the TravelAgency and the TravelPlanner application. This call chain is directly mapped to corresponding operation calls in our model.

The same applies to the process of booking a selected flight. Hereby, the *BookingUsage* initially defines that the selected flight offer may only be accessed by the *Airline* and *User* roles using the *authorizedRoles* attribute. This offer then gets passed to the TravelPlanner, which in turn requests the declassified credit card data from the CreditCardCenter application. Hereby, the operation *CreditCardCenter-requestDeclassifiedCCD* is generic: A declassification can be requested for any set of roles. In this example, it is declassified for the role *Airline*.

The interaction where the user is asked whether he grants access to the data for the requested roles is represented by the operation *User-askForCCDDeclassification*. This interaction is realised through an operation instead of a system usage because usages are only entry points. In this case however, the user is not the entry point but is called instead.

The user then responds with a set of roles for which he granted access to the credit card data. The CreditCardCenter afterwards sends back the correctly declassified data to the TravelPlanner application, which in turn books the flight by invoking *Airline-bookFlight*.

5.7.1.2. Constraint Definition

In the previous section we showed how to map the TravelPlanner example to an instance of our meta model. However, this model does not contain any semantic information on how to check for access control violations. This semantic is added by formulating the constraint as logic program which refers to the model elements.

In this case we define the *authorizedRoles* attribute as marker for data which underlies access control restrictions. It contains the roles which are authorized, whereas each operation specifies the *roles* property to define under which roles it accesses the data. Therefore, we can define the access control constraint as follows:

> *For each operation **o** within any call sequence, for each parameter **p** of **o** with the attribute authorizedRoles, a role **r** must exist so that both **p.authorizedRoles.r** and **o.roles.r** are true.*

More informally speaking, whenever a parameter is passed to an operation, one of the operations roles must be authorized for the access to this parameter value.

5.7.2. Geolocation Constraints Scenario

In this section we model the online shop example we introduced in Section 3.2 as an example for geolocation based data flow constraints. Like in the previous section, we first show how to model the occurring data flows using our meta model. Afterwards, we explain how the constraints for the different privacy levels of the data can be formulated.

DataType	Attributes
ProductId	level
Recommendation	level
ProductPage	level
TransactionLog	level
Cart	level
CustomerInfo	level
AnonymizedOrder	level

Table 5.4.: DataTypes of the model instance for the online shop example.

5.7.2.1. Model Definition

The model for the example online shop system is shown in Figure 5.12. Hereby, we use the same visual representation as in Figure 5.11: The operations are illustrated using white boxes, operation calls are represented using solid arrows. The operation calls hereby are annotated with call parameters and return values. The definition of call parameters, return values and operation properties are represented using annotations.

Like in the previous section, the used types are enumerated separately in Table 5.4. All used types consist of the same single attribute: *level*. It defines the privacy level of the data. Therefore, its *ValueSetType* defines the three *Values type0*, *type1* and *type2*. These correspond exactly to the privacy level we introduced in Section 3.2.

For this definition the design the question may arise why different data types were used when they all consist of the same single attribute. The reason is that we decided to use the data type to differentiate between the contained information of variables: For detecting the problem of *joining data streams*, it is required to differentiate between data flows which have the same information source and data streams which have different sources. This is required because only Type 1 data from different sources imposes a constraint violation when deployed at the same unsafe location. We decided to define that data has the same information source when it has the same data type in our model. This decision was made to keep the model easily understandable, as it is used for illustration purposes. An alternative representation would be to add an additional attribute to the data which specifies the information source.

The geolocation of the operations is represented using the *location* property. In our example we used the three regions *EU*, *US* and *Asia*. If required, this can be modeled on a more fine granular level, e.g. for representing individual data centers within the regions.

Our model does not include a classification of which regions are considered *unsafe* and which are considered *safe*. We decided to leave this classification open for the analysis, because this way different analysis can be performed based on the viewpoint: For example when analysing under US legal restrictions different countries may be considered safe than when checking EU legal constraints.

The actual operations and operation calls are directly derived from the UML sequence diagram of the online shop presented in Figure 3.2. Hereby, also the *level* of the data is assigned according to the level specified in Figure 3.2.

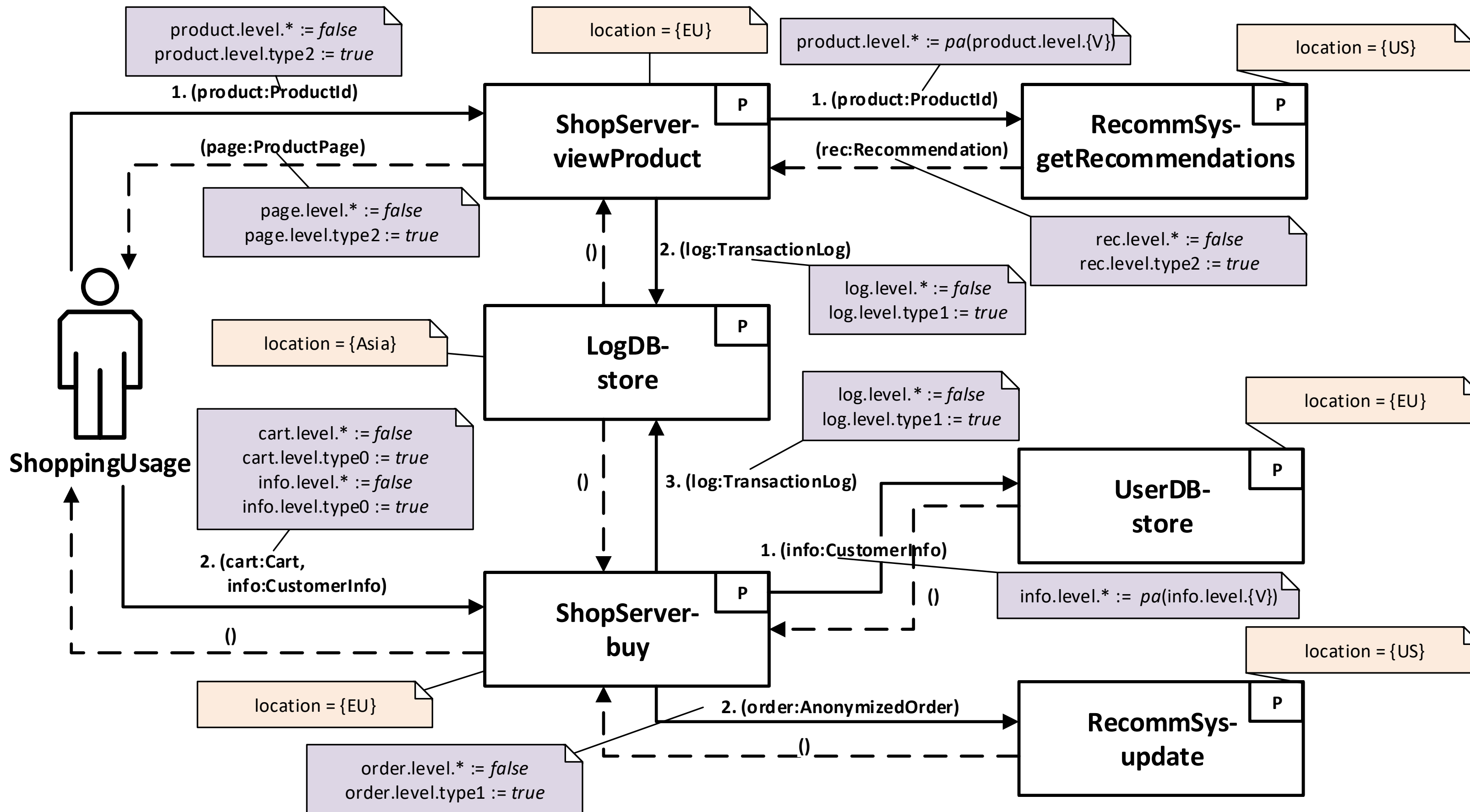

Figure 5.12.: Full system model of the online shop example for geolocation constraints.

5.7.2.2. Constraint Definition

In this section we show how the geolocation constraints can be formulated as constraints for our model of the online shop example. We apply the geolocation constraints we explained in general in Section 3.2 onto our meta model definition.

The constraint for Type-0 data is straight forward: No data of *Type-0* may be processed or stored in an *unsafe* location. Applied onto our model, this imposes the following constraint:

> *For each operation **o** within any call sequence, for each parameter **p** of **o** where the **level** is **type0**, the deployment specified by the **location** property of **o** must be a safe location.*

For Type-1 data, the constraint is more complex due to the problem of *joining data streams*. However, as we defined for our model that data has the same source exactly when its data type is the same, it also can be expressed:

> *For each operation **o1** within any call sequence, for each parameter **p1** of **o1** where the **level** is **type1**, no other operation **o2** meeting all the following restrictions may exist:*
>
> - ***o2** is deployed in the same location as o1, which is defined via the **location** property of both.*
> - ***o2** has a parameter **p2** which has the **level type1**, which however has a different datatype than **p1**.*

This constraint ensures, that for no location it is possible that Type-1 data of a different type (and therefore of a different source) flows there.

6. Translation to Prolog

In this section we show how instances of our data flow system meta model can be automatically translated to Prolog programs. These programs can be queried in order to detect constraint violations in the modeled systems.

We first introduce our Prolog API for formulating data flow constraint queries in Section 6.1. Afterwards, in Section 6.2 we illustrate the usage of the API based on examples. Finally, in Section 6.3 we show how the translation can be realized.

6.1. Constraint Query API

In this section we introduce our Prolog API for formulating data flow constraint queries. The goal is that after a system has been modeled using our meta model presented in the previous section, it can be automatically translated to a Prolog program. Using the API we present in this section, the system can then be queried for constraint violations.

6.1.1. Type Information

The meta model of our approach includes a type system. This type system is useful for formulating queries for which one would normally use the universal or the existential quantors. For example, queries starting like *"For all operations with property XYZ, … "* can be formulated based on the predicates for the type system. The predicates exposed by our API for this purpose are as follows:

isProperty(P)
true when **P** is the name of a *Property*

isDataType(D)
true when **D** is the name of a *DataType*

isAttribute(A)
true when **A** is the name of an *Attribute*

isOperation(OP)
true when **OP** is the name of a *Operation*

isSystemUsage(SU)
true when **SU** is the name of a *SystemUsage*

The names of the elements are defined by the *name* attribute used in the meta model. They are used as Prolog atoms. The predicates presented above can be used for both testing

or as generators. To use them as generators, the predicate just has to be called with an uninstantiated variable. For example, when calling *"isOperation(X)"*, the Prolog interpreter will instantiate X with all available operations.

Both *Attributes* and *Properties* have a *ValueSetType*. The members of a *ValueSetType* can be queried using the following predicate:

valueSetMember(T,V)
true when **V** is the name of a *Value* which belongs to the *ValueSetType* with the name **T**.

The relation between *ValueSetTypes* and *Attributes* and *Properties* is represented using the following predicates:

attributeType(A,T)
true when **A** is the name of a *Attribute* and **T** is the name of its *ValueSetType*.

propertyType(P,T)
true when **P** is the name of a *Property* and **T** is the name of its *ValueSetType*.

As *DataTypes* consist of a set of *Attributes*, their relationship can be queried using the following predicate:

dataTypeAttribute(D,A)
true when **D** is the name of a *DataType* and **A** is the name of one of its *Attributes*.

The last part of type system query API is about the parameters, return values and state variables of *Operations*. Every *Operation* has named *Variables*, of which each one has a type. This can be queried using the predicates shown below:

operationParameter(OP,P)
true when **OP** is the name of a *Operation* and **P** is the name of one of its parameters.

operationParameterType(OP,P,T)
true when **OP** is the name of a *Operation* and **P** is the name of one of its parameters. In addition **T** has to be the name of the *DataType* of **P**.

operationReturnValue(OP,R)
true when **OP** is the name of a *Operation* and **R** is the name of one of its return values.

operationReturnValueType(OP,R,T)
true when **OP** is the name of a *Operation* and **R** is the name of one of its return values. In addition **T** has to be the name of the *DataType* of **R**.

operationState(OP,ST)
true when **OP** is the name of a *Operation* and **ST** is the name of one of its state variables.

operationStateType(OP,ST,T)
true when **OP** is the name of a *Operation* and **ST** is the name of one of its state variables. In addition **T** has to be the name of the *DataType* of **ST**.

6.1.2. Operations, SystemUsages and Calls

For operations we showed in the previous section how their call and return signature as well as their state variables can be queried. In this section we introduce how the actual values of these variables can be referenced for formulating constraints. However, for this purpose we first have to introduce the concept of call stacks in Prolog. Consider the following example call sequence:

$$Usage \stackrel{call1}{\rightarrow} Operation1 \stackrel{call2}{\rightarrow} Operation2$$

In this example sequence, the invocation chain starts with the *SystemUsage Usage*. *Usage* then calls *Operation1*. This call is named *call1*. *Operation1* then invokes *Operation2* with a call named *call2*.

The values of the variables for these calls, namely parameters, return values or state variables as explained in Section 5.2 are defined via *VariableAssignments* as introduced in Section 5.6. These assignments allow variables to depend on parameters, on the return values of previously invoked *Operations* or on arbitrary state variables. In tun, these values can have dependencies on other calls. This implies that the actual value of these parameters can be dependent on the entire call history.

For this reason, we have to provide the call history as call sequence (also referred to as call stack) when querying such variable values. We decided to use Prolog lists for this purpose. The Prolog syntax hereby is optimized for adding or removing the head of such lists via the “|” operator [3]. This implies that Prolog lists should have similar timing characteristics as Single-Linked lists in most applications. This makes them ideal for representing stacks, where the list head is used as stack top.

As we identify model elements from Chapter 5 by using their names as atoms, we can therefore represent the call sequence shown above using a Prolog list as follows:

```
['Operation2','call2','Operation1','call1','Usage']
```

This list represents the call sequence leading to *Operation2* from *Usage*. Note that we use single quotes to prevent Prolog from interpreting names starting with upper case letters as variables instead of atoms. At first sight the list might seem reversed, as it starts with the top of the stack. However, this is due to the Prolog notation, where the leftmost element is the head of the list. In addition it is noteworthy that both the names of the operations as well as the names of the operations are listed by themselves and not in combination, for example as tuples. This design choice was made to make the handling of call sequences easier with the Prolog list concatenation operator.

If we wanted to refer to the call of *Operation1*, we could use the following list instead:

```
['Operation1','call1','Usage']
```

The power of this representation of call stacks becomes visible in combination with the Prolog “|” concatenation operator. Given for example a call stack list as variable *S*. The only thing we know about *S* is that *Operation1* is at the stack top. This means that *S* represents an arbitrary call sequence leading to *Operation1*. In Prolog notation, this means that the following unification holds:

```
S=['Operation1'|_]
```

What if we wanted to refer to the call to *Operation2* given the previous invocation sequence *S*? This is for example useful when defining the call arguments for *Operation2* if they are context dependent. We can now simply formulate this in Prolog using list concatenation:

```
['Operation2','call2','Operation1'|S]
```

This new list refers to the call to *Operation2* from *Operation1* given the previous call history *S*.

With the concept of representing call stacks using lists we now have a mechanism to unambigously identify variables given their call context. This allows us to define the following predicates for querying the values of parameters and return values of our API:

callArgument(S,P,A,V)
true when the *Value V* of the *Attribute A* 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

returnValue(S,R,A,V)
true when the *Value V* of the *Attribute A* of the return value *R* is present given the call stack *S*. The operation which owns the *Return Value R* is defined by the stack top of *S*.

An intuitive interpretation of these predicates can be done by viewing the combination *variable.attribute.value* as boolean variable. This view was already introduced in Chapter 5. For each *Operation* we defined assignments for this variable. The value of **callArgument** or **returnValue** is now exactly true when the assignment of the corresponding variable leads to a propositional logic formula which evaluates to true. hereby the full formula is derived by transitively replacing dependencies to other variables with the help of the given call sequence.

Both predicates can be called with any of their arguments bound, partially bound or unbound. Consider that *Operation2* has a parameter named *file* with the *Attribute encryption* of which a possible *Value* is *AES*. If we want to know all call sequences for which the file is encrypted, we can use the following query:

```
?-      S=['Operation2'|_],
        callArgument(S,'file','encryption','AES').
```

In this example we partially bind the stack list, as we define its head but not the tail. The Prolog interpreter will then provide all instantiations for *S*, where the file is encrypted with AES. A special ability of the Prolog solver is that in many cases it does not need to fully instantiate the stack up to the SystemUsage where the call sequence begins. Instead, the interpreter might just return the following solution:

```
S=['Operation2', 'call2', 'Operation1'|_].
```

This solution reads as that for any call to *Operation2* from *Operation1* the file is encrypted. It hereby does not matter from where *Operation1* was called.

However, often it can be useful to enumerate all possible call stacks fully instantiated. For this purpose the following predicate was introduced:

stackValid(S)
true when the list *S* represents a correct call sequence starting at a *SystemUsage*.

This predicate can be used for testing or as a generator for call stacks. Extending the example above to provide a fully instantiated stack works as follows:

```
?- S=['Operation2'|_],
   callArgument(S,'file','encryption','AES'), stackValid(S).

S=['Operation2', 'call2', 'Operation1', 'call1', 'Usage'].
```

In this example we only have one call path leading to *Operation2* via *Operation1*, therefore only one solution is returned. However, if more paths existed, the Prolog interpreter would list them all.

So far, we presented the predicates for querying call arguments and return values. In addition we need to define predicates for accessing state variables. As explained in Section 5.2, state variables are handled as implicit parameters and return values which are passed with every call. However, in order to differentiate between them and normal call parameters and return values, we decided on introducing separate predicates for them:

preCallState(S,OP,ST,A,V)
true when the *Value V* of the *Attribute A* of the state variable *ST* owned by the operation *OP* is present before the call to the operation on top of the call stack *S* is executed. The operation *OP* hereby is completely independent of the stack *S*.

postCallState(S,OP,ST,A,V)
true when the *Value V* of the *Attribute A* of the state variable *ST* owned by the operation *OP* is present after the call to the operation on top of the call stack *S* is executed. The operation *OP* hereby is completely independent of the stack *S*.

Effectively, these predicates are realized in exactly the same manner as **callArgument** and **returnValue**. **preCallState** corresponds to **callArgument** and **postCallState** corresponds to **returnValue**. The main difference is that in order to unambiguously identify state variables, their containing operation is required as additional parameter. This is not required for return values and call parameters as their containing operation is defined by the call stack top.

Another unique property of state variables is the fact that they have a default value. This default value can be queried using the following predicate:

defaultState(OP,ST,A,V)
true when the *Value V* of the *Attribute A* of the state variable *ST* owned by the operation *OP* is present by default.

Note that **defaultState** is a constant value and therefore does not require a call stack as parameter.

In addition to these value queries, our API also allows to simply query whether a direct call from one operation to another one exists:

> **operationCall(SRC,DEST,CALL)**
> *true* when the caller *SRC* performs a call to the *Operation DEST* with the name *CALL*.

Note that in this predicate, the source *SRC* is a *Caller*. This means that it can be either an *Operation* or a *SystemUsage*.

The last uncovered aspect is the querying of operation property values. We define the following two predicates for this purpose:

> **hasProperty(OP,PR)**
> *true* when the *Operation OP* contains a *PropertyDefinition* for the *Property PR*.
>
> **operationProperty(OP,PR,V)**
> *true* when the *PropertyDefinition* of *Operation OP* for the *Property PR* has the *Value V* as present value.

6.2. Constraint Formulation Examples

In this section we illustrate based on examples how the introduced API can be used to formulate queries for data flow constraint violations. We show how the informal constraint definitions given in Section 5.7 can be formulated in Prolog for the presented examples.

A common approach hereby is to query for constraint violations instead of proofing that the constraint holds. In the case that violations are present, asking Prolog to proof that the constraint holds would just yield *fail* as result. Querying for violation instead gives us an exact context, e.g. a call sequence *when* the constraint is violated. If no violations are present, the query for violations will just yield *fail* as result.

6.2.1. Access Control Scenario

In this section we show how the model presented in Section 5.7.1 can be validated using our API. In Section 5.7.1.2 we already gave the following informal definition for the access control constraint:

> *For each operation **o** within any call sequence, for each parameter **p** of **o** with the attribute authorizedRoles, a role **r** must exist so that both **p.authorizedRoles.r** and **o.roles.r** are true.*

Note that in this definition we solely focus on call parameters which are sufficient for our example model instance. When access control is required on return values or state variables, the constraint query can be formulated similarly. In addition we assume that only data with the attribute **authorizedRoles** and operations with the property **roles** underlie access control. This means that for other data and operations by default no checks

are performed. As previously stated we however do not query that the constraint holds, but instead we query for violations. This means we query for the negation of the statement above, which looks 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.*

We can now directly translate this definition to the following Prolog query:

```
1 ?- isOperation(OP), hasProperty(OP, 'accessRoles'),
2     S=[OP|_],
3     operationParameterType(OP,P,PT),
4     dataTypeAttribute(PT, 'authorizedRoles'),
5     accessRoles(OP,R),
6     isNoRoleAuthorized(R,S,P).
```

At the end, the query uses the helper predicates **accessRoles(OP,R)** and **isNoRoleAuthorized(R,S,P)** which we will define later. We will now go over each line step by step and show how they map to the informal definition of the query from above.

- In line 1 we use **isOperation(OP)** as generator: Prolog will cycle through every operation in our system and instantiate *OP* with it. Afterwards we filter for operations which underlie access control restriction by ensuring that the *accessRoles* property is present using **hasProperty(OP,'accessRoles')**.

- In line 2 we define that we are interested in any call stack *S*, which leads to the current Operation **OP**.

- In line 3 and 4 we iterate over all call parameters of the operation *OP*. First in line 3 we use **operationParameterType(OP,P,PT)** as generator: We cycle through every parameter of **OP**, where the name of the parameter is stored in **P** and its data type is stored in **PT**. As we are only interested in parameters which underlie access control restrictions, we have to filter for parameters whose type has the *authorizedRoles* attribute. This filtering is performed using the **dataTypeAttribute(PT,'authorizedRoles')** term.

- In line 5 we query all roles under which **OP** accesses the data and store it in the list **R**.

- In line 6 we check that none of the roles in **R** is authorized for an access to **P**.

The **accessRoles(OP,R)** predicate called in line 5 is defined as follows:

```
accessRoles(OP,R) :-
 findall(X, operationProperty(OP, 'accessRoles',X),R)
```

The goal of **accessRoles(OP,R)** is that given the operation **OP**, **R** is the list of all roles under which **OP** accesses its data. These roles are defined by the *accessRoles* property. Therefore, we use the built-in predicate **findall**, which returns all solutions for a given query as a list. As query for **findall** we use our API predicate **operationProperty(OP,'accessRoles',X)**.

Finally, in line 6 the predicate **isNoRoleAuthorized(R,S,P)** is called, which we define as follows:

```
1 isNoRoleAuthorized([], _S, _P).
2 isNoRoleAuthorized([Role | R], S, P) :-
3   stackValid(S), \+ callArgument(S,P,'authorizedRoles',Role),
4   isNoRoleAuthorized(R, Stack, P).
```

Given a list of roles **R**, a call stack **S** and a operation parameter **P**, it is defined to return *true* if none of the roles in **R** are authorized to access **P**. Hereby, this check is performed for the operation which is on the top of the call stack **S**. Note that is explicitly **not** required for **S** to be fully instantiated: only the call stack top has to be bound. If the predicate is successful in finding a violation, **S** will be instantiated further to contain the call sequence leading to the violation.

In the definition of **isNoRoleAuthorized(R,S,P)** the base case is defined in line 1: If the list of roles is empty, none of the roles can be authorized. In line 2 we recursively iterate over the list of roles, ensuring that none is authorized. This means that we have to ensure that for a given **Role**, the *authorizedRoles* attribute of the parameter is false. Therefore we need a logical negation, which is performed in line 3. As explained in Section 2.2.3, the Prolog negation is only equivalent to a logical negation if all variables of the negated term are bound. This is already the case for all variables except for the call stack. For this reason **S** is bound using **stackValid(S)** as generator prior to the negation.

Note that this variable binding before the negation can be potentially expensive performance wise. For this reason we introduce in Chapter 7 an optimized way of expressing logical negations.

6.2.2. Geolocation Constraints Scenario

In this section we show how the constraints for our geolocation restrictions example presented in Section 5.7.2 can be expressed in Prolog. For the example we derived the following two constraints. First, we presented one for Type-0 data:

> *For each operation **o** within any call sequence, for each parameter **p** of **o** where the **level** is **type0**, the deployment specified by the **location** property of **o** must be a safe location.*

The second constraint was defined for Type-1 data to prevent the problem of *joining data streams*:

> *For each operation **o1** within any call sequence, for each parameter **p1** of **o1** where the **level** is **type1**, no other operation **o2** meeting all the following restrictions may exist:*

- ***o2*** *is deployed in the same location as o1, which is defined via the* ***location*** *property of both.*
- ***o2*** *has a parameter* ***p2*** *which has the* ***level type1****, which however has a different datatype than* ***p1****.*

Again, data which has no privacy type assigned is ignored in our analysis. If this behaviour is not desired it is possible for example to define an additional constraint which forces data to have type defined. As done previously, we have to negate these constraints in order to find violations. For the Type-0 constraint, this leads to the following informal query:

> *Does an operation* ***o*** *exist within any call sequence, where for any parameter* ***p*** *of* ***o*** *the* ***level*** *is* ***type0*** *and the deployment specified by the* ***location*** *property of* ***o*** *specifies an unsafe location.*

This definition can now be easily translated to Prolog based on our API:

```
1 type0Violation(OP,P,L) :-
2  S=[OP|_], callArgument(S,P,'level','Type-0'),
3  operationProperty(OP,'location',L), isNotSafe(L).
```

In line 2 we define that we are looking for a call sequence with an operation which has a Type-0 parameter. Afterwards in line 3, we ensure that the location specified by the *location* property is an unsafe location. However, for this to work, we need to specify which locations we consider as safe and which we consider as unsafe. This is done using the following statements:

```
1 isSafe('EU').
2 isNotSafe(X) :- valueSetMember('Locations',X), \+ isSafe(X).
```

For our example, we only consider the EU as safe. We define this using the fact in line 1. Our definition for the unsafe location is performed in line 2: We define every location which is not specified as safe to be unsafe.

The second constraint is more complex because we have to find a combination of call sequences which leads to a joining data stream. Negating the definition of the constraint leads to the following informal definition for finding violations:

> *Does a combination two operations* ***o1*** *and* ***o2*** *exist meeting all the following restrictions:*
>
> - ***o2*** *is deployed in the same location as o1, which is defined via the* ***location*** *property of both.*
> - ***o1*** *has a parameter* ***p1*** *which has the* ***level type1***
> - ***o2*** *has a parameter* ***p2*** *which has the* ***level type1****, which however has a different datatype than* ***p1****.*

It is translated to Prolog as shown below:

```
1 type1Violation(S1,P1,S2,P2,L) :-
2  S1=[OP1|_], S2=[OP2|_],
3  operationProperty(OP1, 'location',L), isNotSafe(L),
4  callArgument(S1,P1, 'level', 'Type-1'),
5  operationProperty(OP2, 'location',L),
6  callArgument(S2,P2, 'level', 'Type-1'),
7  operationParameterType(OP1,P1,T1),
8  operationParameterType(OP2,P2,T2),
9  \+ T1=T2.
```

In line 2 we define that we are looking for two call stacks with the operations *OP1* and *OP2* on top. In line 3 we define that *OP1* has to be deployed in an unsafe location *L* and in line 4 we define that *OP1* has to have a parameter *P1* of Type-1. In line 5 and 6 we define the same requirements for *OP2*: It has to be deployed at the same location and also needs to have a Type-1 parameter *P2*. In the lines 7 to 9 we define the final requirement: The problem of *joining data streams* only occurs when *P1* and *P2* originate from different data sources. We defined that the source is equal when both parameters have the same *Datatype*. Therefore we specify in our rule that the types of both parameters have to be different.

6.3. Deriving the Prolog Program

In this section we introduce how a Prolog program can be automatically generated from a system specified using our meta model from Chapter 5. This program is designed to conform to the API from Section 6.1.

An overview of the structure of the resulting logical program is given in Figure 6.1. The first part of the program is the header section: It starts with a preamble, where the predicates which are independent of the modeled system are defined. An example for such a predicate is **stackValid(S)**. Afterwards, the required types of the system, namely *ValueSetTypes*, *Properties*, *Attributes* and *DataTypes* are defined as facts. The process of deriving the header section is explained in Section 6.3.1.

In the following two sections of the program, a block of predicates is generated for each *Operation* and *SystemUsage*. These blocks are especially responsible for defining the values of parameters, return values and state variables. The generation of these predicates is explained in detail in Section 6.3.2.

6.3.1. Program Header Definition

The header section of the generated program contains model-independent predicate definitions as well as the definition of the types specified in the model. For each of the subsections of the header presented in Figure 6.1, we explain how they are derived in the following sections.

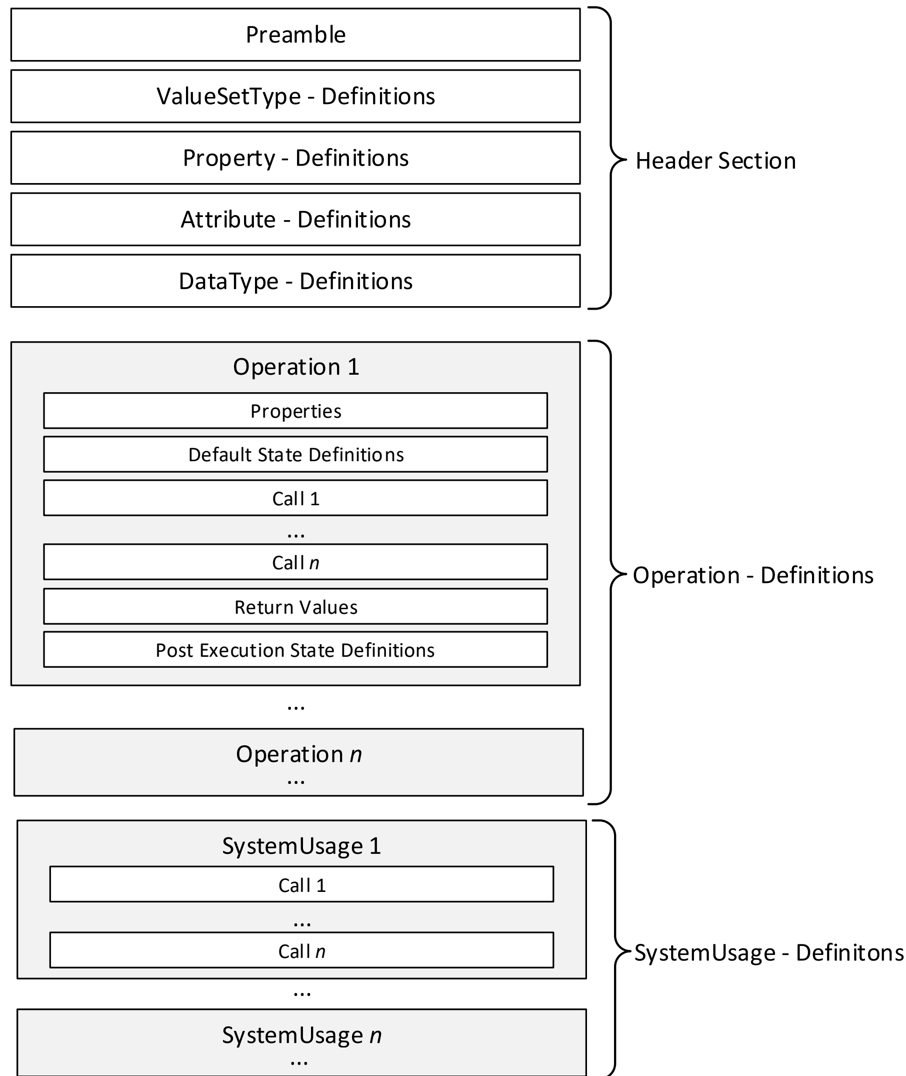

Figure 6.1.: Illustration of the resulting program structure of our proposed Prolog translator.

6.3.1.1. Preamble

The header of the generated Prolog program starts with a preamble which is independent of the generated model. The idea is that several API predicates defined in Section 6.1 can be derived from existing definitions of other predicates. An example is the **isAttribute(A)** predicate: Every *Attribute* has exactly one *ValueSetType* assigned as type. This type can be queried using **attributeType(A,T)**. As we know that each attribute has *exactly* one type, we can therefore define **isAttribute(A)** in the preamble as follows:

```
isAttribute(A) :- attributeType(A,_).
```

The definition of **attributeType(A,T)** is generated for each *Attribute* in the *Attribute-Definitions* section, which is explained in Section 6.3.1.3.

The benefit of this definition of **isAttribute(A)** in the preamble is that it reduces the number of rules required and therefore the total program size: Instead of having one **isAttribute(A)** definition as fact for each attribute, only one in the preamble is required.

The same idea is used for the definition of **isProperty(P)**, **operationParameter(OP,P)**, **operationReturnValue(OP,R)** and **operationState(OP,ST)**:

```
isProperty(P) :- propertyType(P,_).
operationParameter(OP,P) :-
 operationParameterType(OP,P,_).
operationReturnValue(OP,R) :-
 operationReturnValueType(OP,R,_).
operationState(OP,ST) :-
 operationStateType(OP,ST,_).
```

The **stackValid(S)** predicate is also defined in the preamble. We define a stack as valid if it meets the following requirements:

- The top of the stack must be an *Operation*.
- The bottom of the stack must be a *SystemUsage*.
- Between every called *Operation* and its *Caller* there must be the name of the call on the stack.

Based on these requirements we can easily define **stackValid(S)** as follows:

```
stackValid([OP,CALL,SU]) :- isSystemUsage(SU),
    isOperation(OP), operationCall(SU,OP,CALL).
stackValid([DEST,CALL,SOURCE | S]) :-
    stackValid([SOURCE | S]), operationCall(SOURCE,DEST,CALL)
```

The implementation of the predicates used in this definition is explained in Section 6.3.2.

Finally, we define the following predicates as facades in the preamble:

- **callArgument(S,P,A,V)**
- **returnValue(S,R,A,V)**
- **preCallState(S,OP,ST,A,V)**
- **postCallState(S,OP,ST,A,V)**
- **defaultState(OP,ST,A,V)**

In our implementation of these predicates in Section 6.3.2 we rely on the fact that all variables except for the call stack **S** are bound. However, to keep our API as flexible as possible, we do not want to pose this restriction to the user of the API. The solution is to use corresponding generators for the definition in the preamble:

```
callArgument([OP|S],P,A,V) :-
  isOperation(OP), operationParameterType(OP,P,T),
  dataTypeAttribute(T,A), attributeType(A,VT),
  valueSetMember(VT,V),
  callArgumentImpl([OP|S],P,A,V).
returnValue([OP|S],R,A,V) :-
  isOperation(OP), operationReturnValueType(OP,R,T),
  dataTypeAttribute(T,A), attributeType(A,VT),
  valueSetMember(VT,V),
  returnValueImpl([OP|S],R,A,V).
  ...
```

The definition of the corresponding predicates for state variables was omitted here, as it works in exactly the same manner. Note that in the shown examples we call **callArgumentImpl** and **returnValueImpl** after the generators. These predicates are the hidden implementations we define in Section 6.3.2. Again, the corresponding definitions for state variables also delegate to **-Impl** variants of their predicates.

6.3.1.2. ValueSetType Definitions

ValueSetTypes are exposed through our API only via the **valueSetMember(T,V)** predicate. The generation of this predicate is trivial: For each *ValueSetType* in the model, one **valueSetMember(T,V)** fact is generated for each of its *Values*.

For example, the generated code for the *ValueSetType role* of the TravelPlanner example model from Section 5.7.1 looks as follows:

```
valueSetMember('role','User').
valueSetMember('role','Airline').
valueSetMember('role','TravelAgency').
```

6.3.1.3. Property and Attribute Definitions

Properties are exposed via the **isProperty(P)** and **propertyType(P,T)** predicates. **isProperty(P)** is already defined in the preamble. Therefore, we just add one **propertyType(P,T)** fact for each *Property* of the model. For the TravelPlanner example this results in the following code:

```
propertyType('accessRoles','role').
```

The approach for *Attributes* with the **isAttribute(A)** and **attributeType(A,T)** predicates is exactly the same. For this section, the code for the TravelPlanner example is the following:

```
attributeType('authorizedRoles','role').
attributeType('containedRoles','role').
```

6.3.1.4. DataType Definitions

For *DataTypes* our API defines the predicates **isDataType(D)** and **dataTypeAttribute(D,A)**. Again, these can be generated as facts by iterating over the *DataTypes* of the model. For each *DataType* one **isDataType(D)** fact is generated. In addition, for each *Attribute* of the *DataType*, one **dataTypeAttribute(D,A)** fact is added to the program.

The corresponding code for the TravelPlanner example looks as follows:

```
isDataType('RequestData').
dataTypeAttribute('RequestData','authorizedRoles').
isDataType('FlightOffers').
dataTypeAttribute('FlightOffers','authorizedRoles').
isDataType('SingleFlightOffer').
dataTypeAttribute('SingleFlightOffer','authorizedRoles').
isDataType('RoleSet').
dataTypeAttribute('RoleSet','containedRoles').
isDataType('CreditCardData').
dataTypeAttribute('CreditCardData','authorizedRoles').
```

6.3.2. Definition of Operations and SystemUsages

In this section we show how the meta information about *Operations* and *SystemUsages* as well as their execution semantic is translated to Prolog. The execution semantic is hereby specified via operation calls as well as the definition of return values and state variables as explained in Section 5.6.

The translation of *SystemUsages* is very similar to the translation of *Operations*. As already hinted by Figure 6.1, the only difference is that *SystemUsages* do not define return values or state variables and do not have *Properties*. For this reason we only explain the translation of *Operations* and point out when the translation of *SystemUsages* behaves differently.

As shown in Figure 6.1, each *Operation* of the model is translated as an independent block of Prolog statements. This block consists of subblocks for the defined *Properties*, the return values, state variables as well as one subblock for each outgoing call.

The definition of an *Operation* starts with a fact for the **isOperation(OP)** predicate. Correspondingly, the definition of a *SystemUsage* begins with a **isSystemUsage(SU)** fact.

The defined *Properties* are accessible through our API via **hasProperty(OP,PR)** and **operationProperty(OP,PR,V)**. The code generation hereby works similar as for the type definitions shown in the previous section: Firstly, for each *PropertyDefinition* of an *Operation* a **hasProperty(OP,PR)** fact is generated. Afterwards, for each of the present *Values* of the corresponding *Property*, a **operationProperty(OP,PR,V)** fact is added. For the *Airline-bookFlight* operation from the TravelPlanner example, the resulting code therefore looks as follows:

```
hasProperty('Airline-bookFlight', 'accessRoles').
operationProperty('Airline-bookFlight',
                  'accessRoles', 'Airline').
```

After the definition of the *Properties* every operation defines the default values for its state variables. The translation of these assignments works exactly the same as for other assignment based predicates, such as **callArgument**. We will illustrate the approach for translating this type of predicates based on the **callArgument** predicate later in this section. The reason is that the term used in assignments cannot make use of all types of *LogicTerms* while assignments for call arguments can. Default state definitions are not allowed to reference variables as their value depends on the execution stack.

For each outgoing call of the operation a separate block of code is generated. To allow the querying the topology of the system, the **operationCall(SRC,DEST,CALL)** was introduced. Again, one fact per call is added for the definition of this predicate. In case of the call from *TravelPlanner-bookFlight* to *Airline-bookFlight*, this looks as shown below:

```
operationCall('TravelPlanner-bookFlight',
              'Airline-bookFlight','2').
```

Note that the name of the call is *2*, as we used in Figure 5.11 numbers instead of names for the calls.

For each call, it is required that the values of the parameters are specified. This is done by adding rules for the **callArgumentImpl(S,P,A,V)** predicate, which we introduced in Section 6.3.1.1. In our model, the definition of the parameters is performed via the *VariableAssignments* in the *OperationCall*. Given this list of assignments, we generate the rules as follows: For each parameter of the callee, we add one rule for each combination of its *Attributes* with its *Values*.

As example we again consider the call to *Airline-bookFlight* from *TravelPlanner-bookFlight*. The left sides of the rule definitions then look as follows:

```
callArgumentImpl(
  ['Airline-bookFlight','2','TravelPlanner-bookFlight' | S],
  'flight', 'authorizedRoles', 'User') :- ...
callArgumentImpl(
  ['Airline-bookFlight','2','TravelPlanner-bookFlight' | S],
  'flight', 'authorizedRoles', 'TravelAgency') :- ...
callArgumentImpl(
  ['Airline-bookFlight','2','TravelPlanner-bookFlight' | S],
  'flight', 'authorizedRoles', 'Airline') :- ...
callArgumentImpl(
  ['Airline-bookFlight','2','TravelPlanner-bookFlight' | S],
  'ccd', 'authorizedRoles', 'User') :- ...
callArgumentImpl(
  ['Airline-bookFlight','2','TravelPlanner-bookFlight' | S],
  'ccd', 'authorizedRoles', 'TravelAgency') :- ...
callArgumentImpl(
  ['Airline-bookFlight','2','TravelPlanner-bookFlight' | S],
  'ccd', 'authorizedRoles', 'Airline') :- ...
```

We will explain later in this section what the right side of the rule definitions looks like. In the example above, we have the two parameters *flight* and *ccd*, which both only have *authorizedRoles* as single *Attribute*. As *authorizedRoles* has three possible *Values*, this yields six rules in total.

Note how we defined the call stack in the rules: The **specification** of the parameters, namely its propositional logic formula, solely depends on the call itself. It does not depend on the previous call history. The actual **value** however may depend on the previous call history, for example as it can reference the return value of other calls.

Through our rules we define the values using their propositional logic formula. For this reason, we define the call stack in the rule definitions with our call on top and allow any valid call history to be stored in the variable *S*.

To summarize, we now have one rule definition for each parameter, *Attribute* and *Value* combination per call. On the right side of these definitions we want a Prolog representation for the propositional logic formula of the corresponding tuple.

The propositional logic formulas are defined via the list of *VariableAssignments*. For each (*parameter*, *Attribute*, *Value*)-tuple, we select the last matching assignment in the list. We select the last one in order to implement the override-semantic we specified for assignments in Section 5.4. Hereby, we have to consider the wildcarding mechanism we introduced in Section 5.6. An *VariableAssignment* matches for the tuple, if its wildcards can be replaced with the given *Attribute* and *Value* without violating the typing requirements.

After this selection process, we have exactly one *LogicTerm* for each (*parameter*, *Attribute*, *Value*)-tuple. This term now has to be translated to Prolog to form the right side of the rule definitions shown above. As *LogicTerms* form a tree, we translate them to nested Prolog statements.

First, we explain how the atomic terms are translated. The most basic atomic terms we have are *True* and *False*. These are simply translated to the Prolog statements *true* and *fail*. For example, the value of (*rq*,*roles*,*Airline*) of the call to *CreditCardCenter-requestDeclassifiedCCD* from *TravelPlanner-bookFlight* was defined using the following assignment:

rq.roles.Airline := true

This assignment translates to the following Prolog rule definition:

```
callArgumentImpl(
  ['CreditCardCenter-requestDeclassifiedCCD','1' ,
  'TravelPlanner-bookFlight' | S],
  'rq', 'roles', 'Airline') :- true.
```

The translation of references to *Properties* is also simple: We can directly call the **operationProperty(OP,P,V)** predicate we specified earlier in this section.

The last types of atomic terms are references to variables. Parameter and return value references are translated to calls to **callArgumentImpl**, **returnValueImpl**. *DefaultStateRefs* are translated to calls to **defaultStateImpl** and *StateRefs* to **pre/postCallStateImpl**. Whether a *StateRef* is translated to **preCallStateImpl** or **postCallStateImpl** depends on the context of the current *LogicTerm*. As explained in Section 5.2 state variables are realized by passing each of them with operations calls and returns. If now the current

state for the current *LogicTerm* originates from an executed call, **postCallStateImpl** is used. If the current state was passed to the current operation **preCallStateImpl** is used instead.

All variable references except for *DefaultStateRefs* require a call stack as context information. This call stack is derived using the call stack variable defined on the left side of the assignment predicate definitions. For example, consider we are defining the values for the call of *Airline-bookFlight* from *TravelPlanner-bookFlight.* As shown above, we therefore have the following list as call stack on the left hand side of the rule definition:

```
['Airline-bookFlight','2','TravelPlanner-bookFlight' | S]
```

Based on this, the stack representing the call to *'TravelPlanner-bookFlight'* within the same history can be defined as follows:

```
['TravelPlanner-bookFlight' | S]
```

This means, that we can simply reference the call values of *'TravelPlanner-bookFlight'* using the following statement:

```
callArgumentImpl(['TravelPlanner-bookFlight' | S], ...)
```

The referencing of return values works similarly: if we want to reference the return value of the call to *'CreditCardCenter-requestDeclassifiedCCD'*, we can use the following statement:

```
returnValueImpl(['CreditCardCenter-requestDeclassifiedCCD',
        '1','TravelPlanner-bookFlight' | S], ...)
```

Note how we reused in both cases the variable *S*. When looking for proofs, Prolog will systematically instantiate *S* by tracing through our definitions for **callArgumentImpl** and **returnValueImpl**. The definition of the stack works the same way for *StateRefs*. Hereby, the stack is used which references the previous value of the corresponding state variable.

For the compound *LogicTerms* the definition is much simpler. For *And* and *Or* Prolog already comes with the built-in operators **(,)** and **(;)** [3]. For the logical negation *Not* we use the Prolog negation by failure **\+**.

As we explained in Section 2.2.3 the Prolog negation only works as a logical negation if all variables are fully instantiated. We however often use the call stack *S* as only partially instantiated variable. For this reason, the negation of any {Term} is translated as follows:

```
(stackValid(S), \+ {Term})
```

We use **stackValid(S)** as a generator prior to performing the negation. This ensures that *S* is instantiated.

While this approach works, it can come at a severe performance cost: Even though a proof potentially could be found by inspecting only a few operations at the top of the stack, this negation results in every stack being examined fully up to the root. For this reason, we introduce an optimization in Chapter 7 which eliminates the need for using **stackValid(S)** as generator prior to negations.

As the definition of the **returnValueImpl** predicate works exactly the same as for the **callArgumentImpl**, we omit a detailed explanation for it here. At the end of every *Operation*, a set of **returnValueImpl** rules are generated based on the specified *Variable-Assignments* in the same manner as the call arguments for outgoing calls are defined.

For state variables, we have three sets of assignments in total: The definition of the default state, the definition of pre-call state changes and the definition of post execution state changes. The definition of assignments is done using sets of **defaultStateImpl** predicate. Again, the generation works the same way as for **callArgumentImpl** except that no variable for the call stack is used. The default state is context independent. As explained in Section 5.2 state variables are realized by implicitly translating them as parameters and return values. Therefore, the translation of pre-call state changes works the same way as the translation of call arguments. Correspondingly the translation of post execution state changes is realized the same way as the translation of return values. The main difference is that each of these assignments have implicit *VariableAssignments* for copying the previous state values. For each state variable *var* of the corresponding owning operation *op* in the system the following assignment is added to the pre-call and post execution state changes:

$$\text{op.var.*.*} := st(\text{op.var.\{A\}.\{V\}})$$

The only exceptions hereby are the pre-call state changes of the first call of every *SystemUsage*. There no previous state exists. Therefore the default state is used:

$$\text{op.var.*.*} := dst(\text{op.var.\{A\}.\{V\}})$$

This ensures that state variables are always transferred with each call, even if no explicit definitions are given.

7. Prolog Optimizations

In the previous section we introduced our API for formulating constraints and showed how instances of our meta model can be translated to Prolog. Despite the fact that the presented translation should work correctly, the analysis can perform badly regarding the runtime in certain scenarios.

For this reason we introduce several performance optimizations for our approach in this chapter. Even though the optimizations are presented for our approach, their application is not limited to it: The ideas on which the optimizations are based on can be used in other Prolog based approaches as well.

First, in Section 7.1 we show how to enable Prolog to perform first-argument indexing on lists in order to allow more efficient call stack based lookups. Afterwards, in Section 7.2 we tackle the problem of logical negations requiring the exploration of the entire variable space. Finally, in Section 7.3 we remove the need for unrolling *VariableAssignments* for each *Attribute-Value* combination.

We explain analytically in this chapter why we expect performance gains from these optimizations. These hypothesis are experimentally validated in Chapter 8.

7.1. First Argument Indexing

In Section 2.2.4 we explained how Prolog can perform first argument indexing on rules: If the first argument of a specified rule is an atom the interpreter can use hashing to minimize the rule lookup time.

Fortunately, this is already the case for most of our generated Prolog rules: The generated static type information for *DataTypes*, *Attributes* and *Values* all are based on facts which have atoms as first argument. The same is the case for *Properties* as well as the structural information on *Operations*, such as the call parameters and return values.

However, we discovered that the indexing currently does not work for predicate lookups which depend on the call stack. This means, that most Prolog Interpreters perform a linear search for a lookup of all of the following predicates:

- **returnValue(S,R,A,V)**
- **callArgument(S,P,A,V)**
- **preCallState(S,OP,VAR,A,V)**
- **postCallState(S,OP,VAR,A,V)**

For all of these predicates, the call stack represented as Prolog list is used as first argument. However, in all of our rule definitions the head of these lists, which represent the

top of the call stack, is an atom. For example, in the definitions of **returnValue(S,R,A,V)** generated by our translator the head of **S** is always instantiated to the *Operation* whose return values are defined. However, we discovered that many Prolog interpreters are unable to detect this pattern. Therefore, even when **returnValue** is used in a goal with the top of the stack initialized, a linear search still is performed for finding matching rules. This is especially performance critical as the number of definitions of the predicates listed above scales linearly with the number of operations.

In the following we present our approach for solving this issue using the **returnValue** predicate as example. However, the found solution can be applied in the same manner to all the predicates listed above.

One solution for enabling first argument-indexing on **returnValue(S,R,A,V)** is to simply change the parameter order: We can put the return value **R**, the *Attribute* **A** or the *Value* **V** to the front of the predicate. This does enable the interpreter to use first argument indexing because in all rule definitions generated by our translators these values are instantiated with atoms.

This solution, however, does come with a downside: Both the *Attribute* and the *Value* are commonly reused: As a result it can be expected, that in a large system very many **returnValue** definitions occur with exactly the same values for the *Attribute* and the *Value*. If these are used as keys for the first argument indexing, this leads to very large hash buckets. Such large hash buckets reduce the gain of the indexing, because then a long running linear search has to be performed within the corresponding hash buckets. The same is the case for the return value **R**: Variable names are often reused in different operations as they commonly document the semantic of the stored values. Again this leads to large hash buckets when used as first argument.

For this reason we assumed that it would be best to index by the top of the call stack **S**. For the **returnValue** predicate this would mean that each operations return value definitions are put together in unique hash buckets. This is a desirable result as especially in combination with the optimization for shorter assignments we present in Section 7.3 we expect the number of such rule definitions per operation to be relatively low.

As previously stated many Prolog interpreters are unable to index by the head of the list used as first argument. The idea to circumvent this is that we simply add a redundant parameter to our rule definition which enables indexing. Consider the following example rule definition generated by our Prolog translator:

```
returnValueImpl(
  ['op1'|S], 'myRV', 'myAtt', 'myVal']) :- ...
```

The idea for our optimization is simple: we just copy the top of the call stack and put it to the front as a new parameter. This means, that all rules now look as follows

```
returnValueIndexed(
  'op1',['op1'|S], 'myRV', 'myAtt', 'myVal']) :- ...
```

The problem we face with this code generation is that we only defined **returnValueIndexed** rules. In our generated code we however reference return values using the **returnValueImpl** predicate. For this reason, we have to add the following indexing resolution rule to the program preamble:

```
returnValueImpl([T|S],R,A,V) :-
 returnValueIndexed(T,[T|S],R,A,V).
```

When the Prolog interpreter now has to prove a **returnValueImpl** goal, it directly jumps to this indexing resolution rule as it is the only definition for **returnValueImpl**. This rule extracts the top of the call stack and then calls **returnValueIndexed**. On **returnValueIndexed** then the first argument indexing takes place. A benefit of this index resolution rule is that all other aspects of the code generation can be left unchanged. Especially our API facade presented in Section 6.1 is not touched. In our provided translator implementation, we implemented this optimization for all of the predicates listed in the beginning of this section.

7.2. Efficient Logical Negations

We explained in Section 2.2.3 that Prolog does not support a logical negation. However, we showed how a logical negation can be implemented based on Prologs not-provable negation operator. For our realization of logic terms we therefore defined the logical not as follows in Section 6.3.2:

```
(stackValid(S), \+ {Term})
```

As we already hinted, this implementation for logical negations has a very bad worst-case performance: It is possible that an exponential amount of different call sequences have to be tried when using this approach, even though a proof could normally be found in constant time. Consider for example that we have the following definition in our code:

```
returnValueImpl(['op1'|S],'rv','att','val']) :- true.
```

We now want to find all call stacks to *op1*, where the *Value val* of the *Attribute att* of the return value *rv* is *false*. Obviously, no such call stack exists because of the definition above. However, when formulated as logical term query the proof can still be very time consuming:

```
?- (stackValid(['op1'|S]),
 \+returnValueImpl(['op1'|S],'rv','att','val'])).
```

Even though **returnValueImpl([op1|S],rv,att,val])** can never be *false*, the Prolog interpreter will still try to prove it for every call stack with *op1* on top.

Our solution for this problem is to implement a custom logical negation operator *lnot*. This means our query from above looks like this instead:

```
?- lnot(returnValueImpl(['op1'|S],'rv','att','val'])).
```

Note that there is no more a **stackValid** statement used as generator. Instead, our operator *lnot* by definition instantiates the callstack *S* to all values, for which the negated term yields *false*.

The idea is that we provide the interpreter a way of proofing that a statement is *false* in the same manner as it currently proves that a statement is *true*. For example, for each definition of *returnValueImpl* such as

```
returnValueImpl(['op1'|S],'rv','att','val']) :- {term}.
```

we add a dual definiton to the code:

```
not_returnValueImpl(['op1'|S],'rv','att','val']) :-
  lnot({term}).
```

.

Note that the actual value of *{term}* is the same in both definitions. However, for *not_-returnValueImpl* it is negated. This means that a query of *not_returnValueImpl* succeeds exactly when a query of the corresponding *returnValueImpl* would fail. Given these dual definitions, we can now add the following rule to the preamble for resolving *lnot* predicates:

```
lnot(returnValueImpl(S,R,A,V)) :-
  not_returnValueImpl(S,R,A,V).
```

When the prolog interpreter now encounters a *lnot* predicate during its proof, it searches for a rule to resolve it. Above, we provided a rule for resolving the atomic *ReturnValueRef* logical term. For the negation to be fully resolvable for complex terms, we have to provide such resolution rules for all logical terms we introduced in Section 5.6. For the atomic terms *True* and *False* we simply add the following two rules to the preamble:

```
lnot(true) :- fail.
lnot(fail) :- true.
```

For all other atomic terms such as *ParameterRef* or *PropertyRef* the approach is the same as as for *ReturnValueRef*: This means for example that for every *callArgument* definition, we add a dual *not_callArgument* definition alongside with a resolution rule in the preamble.

With these rules we are now able to succesfully perform logical negations using *lnot* on all atomic terms. However, the compound terms *And*, *Or* and *Not* are not resolvable yet. To resolve these we use the following basic logical identities:

$$\neg(A \land B) \leftrightarrow (\neg A \lor \neg B)$$
$$\neg(A \lor B) \leftrightarrow (\neg A \land \neg B)$$
$$\neg\neg A \leftrightarrow A$$

By repeatedly applying these rules on any complex logical terms, the negations are pushed towards the atomic terms. When representing logical terms in tree form this means that the negation are pushed down towards the leafs. If we apply these rules repeatedly until none is applicable, negations only occur on atoms in the result term.

Fortunately, it is trivial to translate these rules to Prolog:

```
lnot((A,B)) :- (lnot(A);lnot(B)).
lnot((A;B)) :- (lnot(A),lnot(B)).
lnot(lnot(A)) :- A.
```

After adding these rules to the preamble, Prolog now automatically performs the pushdown of logical negations when proving negated compound terms. After this pushdown has been performed, negations only occur on atomic terms. However, for atomics terms the *lnot* predicate is already well defined based on our definitions from above.

To further illustrate this mechanism, the following listing shows how Prolog applies the rules for performing the pushdown of negations:

```
lnot((callArgumentImpl(..);lnot(returnValueImpl(..)))
%Applying lnot((A;B)):-(lnot(A),lnot(B)).
(lnot(callArgumentImpl(..)),lnot(lnot(returnValueImpl(..))))
%Applying lnot(lnot(A)):-A.
(lnot(callArgumentImpl(..)),returnValueImpl(..))
%finished! now only atomic terms are negated
```

With these changes we now have enabled the Prolog interpreter to proof negated terms exactly in the same manner as it proves non-negated terms. This means we erased the performance hit of negation at the cost of having one additional dual definition for each *VariableAssignment* and a few more rules in our program preamble. As a result, we expect an increased load time of the program by a factor of two. This is however little cost compared to that we now prevent the interpreter from running for exponential time for even simple proofs containing negations.

Note that our introduced *lnot()* predicate does not act as a universal replacement for the Prolog negation-by-failure operator “\+”. The difference is that our negation is limited to be used to negate combinations of the following predicates:

- The Prolog conjunction with two arguments **(,)**
- The Prolog disjunction with two arguments **(;)**
- The **lnot** predicate, meaning that chains of negations can be resolved
- The Prolog **true** and **fail** predicates
- The **callArgument** and **callArgumentImpl** predicates
- The **returnValue** and **returnValueImpl** predicates
- The **preCallState** and **preCallStateImpl** predicates
- The **postCallState** and **postCallStateImpl** predicates
- The **operationProperty** predicate

This set of negatable predicates is sufficient to allow the negation of all terms which can be modeled using *LogicTerms*. A minor downside of our negation approach is that when adding new types of terms we also have to add rules for resolving the negation of them. For example consider that we want to use logical implications for defining some constraints:

```
implies(A,B) :- (lnot(A);B).
```

When the interpreter now encounters negated implications of the form *lnot(implies(...))*, it will fail as no rule for resolving negations of the *implies* predicate. Therefore, we have to add a resolution rule for the negation:

```
lnot(implies(A,B)) :- lnot((lnot(A);B)).
```

7.3. Cut-Based Assignments

In Section 5.6 we introduced *VariableAssignments*. They are used for the definition of all types of variables, such as parameters, return values or state variables. As for a datatype the number of attribute-value combinations can easily become very high, we introduced wildcards: with wildcards, we can specify the value of multiple attribute-value combinations using a single assignment. When used correctly this implies that the number of required assignments for a single variable is typically much lower than the number of combinations of all attributes and their values of the variables datatype.

However, during the translation to Prolog shown in Section 6.3.2 this benefit is lost. The translator looks for the matching assignment for every attribute-value combination and generates one Prolog rule for each. This means that the number of generated rules can easily become very high, leading to a long loading time of the program. For example consider that we have a parameter **data** which has **origin** as only attribute. **origin** hereby is meant to specify the country where the data was acquired. Therefore it has a *ValueSetType* which contains one value for each country. If we now want to specify that data originates from Germany we can use the following two assignments:

$$\text{data.origin.*} := false$$
$$\text{data.origin.germany} := true$$

However, during the code translation the Prolog translator generates one statement for every existing country:

```
callArgument(..., 'data', 'origin', 'germany') :- true.
callArgument(..., 'data', 'origin', 'france') :- fail.
callArgument(..., 'data', 'origin', 'spain') :- fail.
callArgument(..., 'data', 'origin', 'austria') :- fail.
callArgument(..., 'data', 'origin', 'switzerland') :- fail.
...
callArgument(..., 'data', 'origin', 'belgium') :- fail.
callArgument(..., 'data', 'origin', 'italy') :- fail.
```

In the following we introduce a more complex realization of *VariableAssignments* in Prolog, which in contrast generates only one statement per assignment.

As first observation we make use of the fact that every Prolog interpreter is guaranteed to examine rules in the order they appear in the Program. For a list of *VariableAssignments* we want the last matching to be used. Therefore as initial idea we could translate the assignments from above simply by using Prolog variables and reversing the order of the assignments:

```
callArgument(...,'data','origin','germany') :- true.
callArgument(...,'data','origin',_V) :- fail.
```

In the second rule we simply translated the wildcard of the first assignment as an anonymous variable. While this approach works for this example, it can easily break as shown by the next example assignments:

$$\text{data.authorizedRoles.*} := true$$
$$\text{data.authorizedRoles.nsa} := fail$$

This translates to the following rules according to our new approach:

```
callArgument(...,'data','authorizedRoles','nsa') :- fail.
callArgument(...,'data','authorizedRoles',_V) :- true.
```

These rules do not have the desired behaviour: When asking the Prolog interpreter for the value of **callArgument(. . . ,'data','authorizedRoles','nsa')** it yields *true* instead of the expected *false*. This happens because of backtracking: The interpreter encounters the first rule which specifies the value for *nsa*. However, because this value is *fail*, the interpreter scans for the next matching rule, which in turn yields *true*. In terms of *VariableAssignments* this is equivalent to scanning for matching assignments until we find one for which result is *true*.

To summarize, we need a mechanism of stopping the interpreter from backtracking as soon as he found a rule (representing a single assignment) that matches for the query. This is exactly what the cut predicate introduced in Section 2.2.2 does. As soon as it is reached, no other rules are examined even if the current rule yields fail. This means that we now translate our assignments from above as follows:

```
callArgument(...,'data','authorizedRoles','nsa') :- !,fail.
callArgument(...,'data','authorizedRoles',_V) :- !,true.
```

This approach now provides the correct results. When querying for the value *nsa*, the first rule is used. There, the cut is reached meaning that no other rule will be examined. As the rule fails, the result for *nsa* is *fail*. For all other roles, the first rule does not match on the left side. Therefore, the second rule is used, which also yields the correct result *true*.

However, this approach is still not fully correct. In Table 5.2 we listed combinations of wildcards which induce typing restrictions. An example for such a typing restriction is the following assignment:

$$\text{retVal.*.*} := pa(\text{param.\{A\}.germany})$$

In this example the *Value* **germany** is part of the *ValueSetType* **geolocation**. This assignment therefore has to be applied only for attributes which also have the *ValueSet-Type* **geolocation**, because otherwise typing errors would be induced. With our current approach we would translate this assignment to the following Prolog rule:

```
returnValue(..., 'retVal',A,V) :-
  !,callArgument(..., 'param',A, 'germany').
```

The problem we observe now is that the left side of the rule matches for every *Attribute*, even for ones which do not have the **geolocation** type. As result of this typing error, the **callArgument(...)** on the right side will always yield *fail* as result.

The solution for this problem is to perform type checks before the cut is reached. If these type checks fail, the Prolog interpreter will continue to scan for the next matching rule. As shown in Table 5.2, we only have two kinds of type restrictions in fact: either an *Attribute* has to be part of a certain *DataType* or it is required to have a certain *ValueSetType*. In both cases, the corresponding *DataType* or *ValueSetType* is known at translation time due to static typing. Therefore we can simply use the **dataTypeAttribute(D,A)** and **attributeType(A,T)** predicates we introduced in Section 6.1.1 to perform the type checks. Now, our example translates fully correctly as follows:

```
returnValue(..., 'retVal',A,V) :-
  attributeType(A, 'geolocation'),
  !,callArgument(..., 'param',A, 'germany').
```

While the left side of the rule still matches for every attribute, the predicate **attributeType(A,'geolocation')** fails for every attribute which does not have the correct type. As this check is performed before the cut is reached, the interpreter continues to scan for the next matching assignment.

The final problem that needs to be resolved is the fact that different sets of assignments can interfere with each other. Consider for example that we have two calls to the same operation:

```
%Call to op1 from src1
callArgument([op1,c1,src1|_],'arg',A,V) :- !,true.
%Call to op1 from src2
callArgument([op1,c1,src2|_],'arg',A,V) :- !,true.
```

We now want to find all call stacks for which the value *germany* of the attribute *geolocation* is true. Therefore, we execute the following query:

```
?- callArgument(S,'arg','geolocation','germany').
```

The interpreter now finds **S=[op1,c1,src1,...]** as the only solution. This is not correct because **S=[op1,c1,src2,...]** is a valid solution too. This happens because the cut from the definition of the call arguments from **src1** prevents the interpreter from scanning for additional solutions. This problem can be solved by moving each group of assignments to a separate predicate which is called as sub goal:

```
%Call to op1 from src1
callArgument([op1,c1,src1|_],'arg',A,V) :-
  assignments_1('arg',A,V).
assignments_1('arg',A,V) :- !,true.
%Call to op1 from src2
callArgument([op1,c1,src2|_],'arg',A,V) :-
  assignments_2('arg',A,V).
assignments_2('arg',A,V) :- !,true.
```

As we generate a separate subgoal **assignments_*x*** for each group of assignments, the effects of cuts are guaranteed to be local for each assignment set. The cut now prevents backtracking within the **assignments_*x*** predicate, but not for the **callArgument(...)** query. Executing the query from above now correctly gives both solutions.

To summarize this optimization, the following steps have to be taken to generate correct rules for a set of *VariableAssignments* for a single variable:

- For each set of assignments a unique predicate **assignments_*x*** is generated and used for the definitions. The predicate for which the assignments are performed (e.g. **callArgument** or **returnValue**) call this predicate as subgoal.
- The order of the assignments has to be reversed. This ensures that the assignment with the highest priority appears first in the code.
- If the *Attribute* or the *Value* of the assignment are wildcards, Prolog Variables are used in their place. If these wildcards are not referenced on the right side of the assignment, anonymous variables are used.
- The pattern **{type restrictions},!,{logic term}** is used for the right side of each generated rule:
  - The type restrictions are generated using **dataTypeAttribute(D,A)** and **attributeType(A,T)** according to Table 5.2.
  - The cut is used to prevent the interpreter from backtracking when a matching assignment was found.
  - The logic term generation is left unchanged. The only difference is that the introduced Prolog variables are used at places where wildcards are referenced.

With this approach we now generate exactly one Prolog rule per *VariableAssignment* instead of requiring one for each attribute-value combination.

8. Evaluation

In this chapter we evaluate our proposed approach. Firstly, we introduce the goals of the evaluation and the resulting research questions in Section 8.1. Afterwards in Section 8.2 we explain the evaluation design we use for answering the research questions. The results are then given in Section 8.3. In Section 8.4 we show the threats to the validity of our evaluation. Finally, in Section 8.5 we discuss the assumptions and limitations of our approach.

8.1. Goals and Questions

The general goal of the evaluation of this thesis is to examine how well the combination of our proposed meta-model (Chapter 5) and the Prolog Translator (Chapter 6) perform regarding accuracy, scalability and genericness. To quantify this, we apply the Goal-Question-Metric approach [1]. In this section we introduced the corresponding goals alongside with their research questions. The metrics are introduced separately for each goal in Section 8.2.

In order to derive the goals it is required that we first specify our expectations regarding our approach. In terms of functionality, our approach is primarily designed for detecting data flow constraint violations for two main scenarios. The two main scenarios are access control and geolocation based flow restrictions as presented in Chapter 3. However, while our approach was designed with these scenarios in mind, we still aimed to keep it as generic as possible for being applicable in additional scenarios.

A central additional requirement we impose on our approach is a good performance and scalability. In order for our approach to be as applicable as possible, it is required that it provides analysis results in a timely manner. In order to achieve this, we proposed several performance optimizations in Chapter 7. Therefore another central aspect of the evaluation is to examine the effectiveness of the proposed optimizations.

Based on these requirements regarding our approach, we can identify the following three evaluation goals:

G1 Evaluate whether our approach enables the accurate analysis of data flow constraint violations in access control and geolocation based restrictions scenarios.

G2 Examine how well the translation and analysis scale with the size of the model as well as the effectiveness of the proposed optimizations.

G3 Examine how well our approach is suited for the analysis of data flow constraints other than access control and geolocation based privacy restrictions.

With **G1** we evaluate the functional accuracy of our approach. As it is hard to prove that a software system is fully correct, we focus on the accurate handling of our two main scenarios. In addition we need to ensure that our proposed performance optimizations do not influence the accuracy of our approach. This leads to the following research questions which allow an experimental evaluation:

RQ-1.1 Does our approach accurately detect access control violations for the TravelPlanner example?

RQ-1.2 Does our approach accurately detect data flow violations for the shop example?

RQ-1.3 Does any of the performance optimizations influence the accuracy of the analysis?

In these research question we assume that our approach functions accurately when it finds all constraint violations without returning false positives. In order to experimentally evaluate these questions we add specific violations to our example models before analysing them. Details can be found in Section 8.2.2.

The scalability analysis of our approach **G2** is performed regarding two aspects. Firstly we examine how well the Prolog Translator as well as the Prolog analysis scale with the size of the model. Secondly, we evaluate the effectiveness of our proposed performance optimizations.

For the first aspect we identified the following research questions:

RQ-2.1 How well does our approach scale with an increasing number of operations at constant callgraph complexity?

RQ-2.2 How well does our approach scale with an increasing callgraph complexity?

RQ-2.3 How well does our approach scale with an increasing number of call parameters and return values per operation?

RQ-2.4 How well does our approach scale with an increasing number of attribute-value-combinations?

RQ-2.5 How well does our approach scale with an increasing number of properties and property definitions?

We identified these research questions based on in which dimensions instances of our meta-model are explicitly scalable given a base model. These dimensions can be scaled automatically without changing the semantic of the model instance. Other dimensions, such as the number of variable assignments are scaled indirectly through these dimension. Even though variable assignments are a central aspect of our approach, they cannot be scaled automatically without altering the semantic of the model. The scaling of the callgraph complexity for **RQ-2.2** means that we scale the length of the paths the Prolog interpreter has to instantiate to find variable values. For each research question we perform an analysis with multiple combinations of optimization configurations. For **RQ-2.3** the question may arise how practically relevant it is to scale the number of parameters and

return value per operation. Usually, operations are comparable to functions in the modeled system which on average have a low, constant number of parameters. However, in our approach we introduced state variables. As explained in Section 5.2 state variables are effectively realized using the same mechanisms as parameters and return values. Therefore we assume that we can quantify the impact of state variables by scaling the number of parameters and return values for each operation.

The second aspect of **G2** is the evaluation of the effectiveness of our optimizations. As we introduced three different optimizations, this leads to the following reasearch questions:

RQ-2.6 How does the first argument indexing optimization affect the performance of our approach?

RQ-2.7 How does the logical negation optimization affect the performance of our approach?

RQ-2.8 How does the cut-based assignments optimization affect the performance of our approach?

To answer these questions, we use the experiments from **RQ-2.1** to **RQ-2.5** as indicators for the average case performance. However, we provided a theoretical motivation of the performance problem that is tackled for each optimization in Chapter 7. In order to evaluate the hypothesis of these performance problems, we also measure specialized model instances where we provoke the expected performance problems. These can be seen as best-case for the effectiveness of the proposed optimizations.

For the last goal **G3** an experimental evaluation is difficult to achieve. We therefore decided to evaluate this goal argumentation based. For this purpose we performed additional research in order to find additional data flow constraint scenarios. This lead to the following research questions, where each refers to a found scenario class:

RQ-3.1 Is our meta-model capable of expressing information flow constraint scenarios?

RQ-3.2 Is our meta-model capable of expressing data secrecy and integrity constraint scenarios?

RQ-3.3 Is our meta-model capable of expressing data lifecycle constraint scenarios?

Details on how we identified these scenario classes can be found in Section 8.2.4. In order to answer these questions, we discuss which features of our meta-model can be used for modeling the specific characteristics of each scenario class.

8.2. Evaluation Design

In this section we explain how we plan on answering the research questions we introduced in Section 8.1. The experiments are defined for each goal separately in Section 8.2.2 to Section 8.2.4. Prior to this, we introduce the randomized call graph scaling technique in Section 8.2.1 as it is used for the experiments. Finally, we explain our choice of Prolog implementations in Section 8.2.5 and list the specification of our experiment system in Section 8.2.6

8.2.1. Randomized Call Graph Scaling

In order to maximize the informative value of **RQ-1.1** and **RQ-1.2** it is required that we execute our approach on many different model instances for each scenario. However, we decided that manually defining semantically equivalent models for our base models introduced in Section 5.7 is too time consuming for this thesis. Instead, we designed a method for automatically generating models with a randomized topology given a base model: the randomized call graph scaling.

The goal of this method is to alter the call graph of the given model without influencing the expected analysis results. This means that we change the calls between operations. However, to preserve the semantic of the model it is required that the data flows remain the same: If data flows from a certain operation to another one in the original model, the same flow has to be present in the derived scaled model. However, the flow may happen across other operations which do not alter the data.

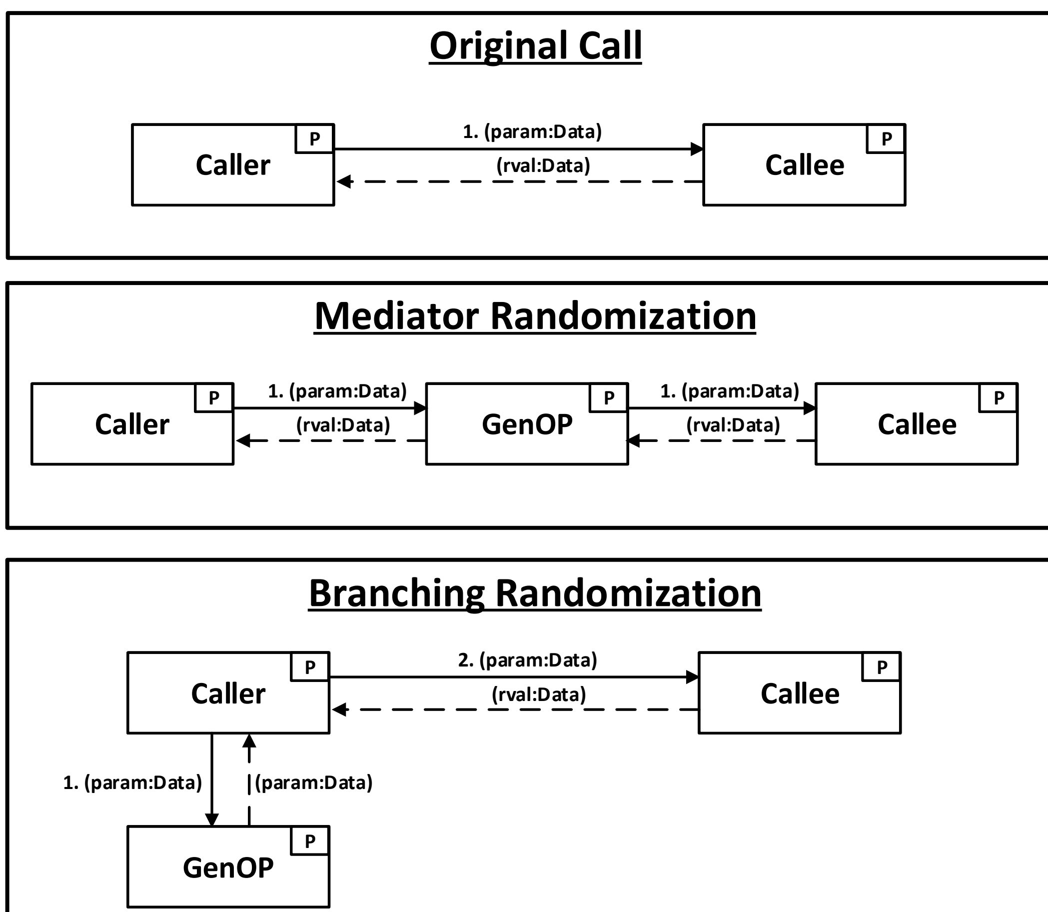

Figure 8.1.: Basic structural change actions for call graph scaling.

To achieve this, we first define two basic structural change actions which can be applied to any model instances. We define these actions so that the data flow is preserved while

the call graph topology is changed. By repeatedly applying these change actions we can then derive highly randomized model instances.

The two change actions are illustrated in Figure 8.1. Firstly, we define the *Mediator Randomization*. For this change we chose a random call to a random operation. Given this call, we insert a generated operation, which is named *GenOp* in Figure 8.1. The original call is then replaced with a call to and a call from *GenOp*. The original caller calls the generated operation and the original callee is the only operation called from *GenOp*. The key point is the fact that *GenOp* copies the parameter and return value signature of the callee. This makes it possible that *GenOp* simply passes the data through: *GenOp* defines the call arguments for the outgoing call by simply copying its input parameters. Similarly, the return value is defined by copying the values returned by the callee. This way, the original data flow is preserved.

The insertion of mediator operations alone is not sufficient for a good randomization of the call graph topology: Edges are simply replaced by chains of operations, however no new branching occurs. For this reason we define an additional structural change action, the *Branching Randomization*. For the first part it is similar to the *Mediator Randomization*: We generate a new operation which we name *GenOp* in the example. Instead of the original callee this operation receives the call parameters from the caller. In contrast to the *Mediator Randomization GenOp* does not call the callee. Instead, *GenOp* simply returns its input parameters unchanged. These returned values are then used by the caller to call the original callee. Again, the original data flow is preserved. The difference is that we introduced a branch: The single original call was replaced by two outgoing calls.

The randomized call graph scaling now works as follows: We select a random call in the model. Then we either apply the *Mediator Randomization* or the *Branching Randomization*, either one is chosen equally likely. Afterwards we chose another random call and repeat the process. The number of repetitions defines the size of the randomized model: If we apply 100 change actions, the resulting model has 100 more operations than the original model. As each individual change action preserves the data flow, the data flow is also preserved in the resulting model.

8.2.2. G1 - Accuracy

In order to answer **RQ-1.1** and **RQ-1.2** we perform an experimental evaluation where we apply our approach in order to detect data flow constraint violations. Hereby, we use the models we presented in Section 5.7 in combination with the presented analysis from Section 6.2. However, these models do not contain any constraint violations. For this reason it is required that we inject violations in order to evaluate our analysis. We decided on injecting two different violations for each model.

Figure 8.2.: System model of the TravelPlanner case study with violations injected. The changed parts in comparison to Figure 5.11 are highlighted in red. The assignments of the call chain of *TravelPlanner-getFlightOffers* are unchanged and have been omitted in the figure to preserve clarity.

For our shop scenario (Figure 5.12) which is an example for geolocation based restrictions the injection of violations is simple. We just need to alter the deployment of some operations which is specified via properties. In order to keep the violations representative it is required that we inject one violation for each privacy type we presented in Section 3.2. Therefore we choose the following violations:

1. The location property of *UserDB-store* is changed from *EU* to *US*. This operation receives *CustomerInformation*, which is personal information. As we defined that only the EU is a safe location, we therefore induce a type-0 violation as described in Section 3.2.

2. The location property of *RecommSys-getRecommendations* and *RecommSys-update* is changed from *US* to *Asia*. This is a redeployment from one unsafe location to another unsafe location. However, *LogDB-store* is already deployed in *Asia*. *RecommSys-getRecommendations* and *LogDB-store* both have type-1 data as input, but their inputs have a different sources. We therefore injected a type-1 violation due to joining data streams.
   Note that in our analysis this root violation manifests as two violations in the Prolog analysis results: There are two different call stacks which call *LogDB-store*: one via *ShopServer-viewProduct* and one via *ShopServer-buy*. Both call stacks lead in combination with a call stack to *RecommSys-update* type-1 violations in our specified analysis.

For the TravelPlanner example the choice of model changes is simple. The model contains two types of data which underlie access control: The selected flight as well as the credit card data. Therefore we decided on injecting one unauthorized access for each of these data. As the model changes are more complex, we give an illustration of the differences to the original model (Figure 5.11) in Figure 8.2. The two induced violations are as follows:

1. The user does not declassify the *CCD* for the *Airline* role. This is realized by changing the return value assignment of *User-askForCCDDeclassification*. However, the *CCD* is still passed to *Airline-bookFlight* via *TravelPlanner-bookFlight*. As the Airline is not allowed to access the *CCD* this is an access control violation.

2. The new operation *TravelAgency-notify* is added. This operation is called to let the TravelAgency know which flight was chosen by the user, for example in order to improve the marketing. *TravelAgency-notify* is called by *TravelPlanner-bookFlight* with the flight offer which was chosen by the user. However, the TravelAgency role is not allowed to access the flight offer. Therefore an access control constraint violation is induced.

Based on these violations we define four equivalence classes of model instances per scenario: One for each of the two violations, one without any violations and one with both violations injected. So far, we only have one model instance per class: For every class we have the corresponding base model defined in Section 5.7 with the violations of

the equivalence class injected. In order to improve the informative value we decided on automatically generating more models for each class given the existing instance. For this purpose we use the randomized call graph scaling we introduced in Section 8.2.1. The scaling does not influence the semantic of the model, as original data flows are preserved. However, we randomize the topological structure of the call graph of the models. Therefore this evaluation approach allows us to make statements about the accuracy of our approach independent of the topological call graph structure.

To minimize the probability that we just generated model instances for which our approach works by accident, we decided on generating 100 models for each equivalence class. These are generated using the randomized call graph scaling given the base models from Section 5.7 with the violations for the equivalence class injected. The randomized call graph scaling is executed with 100 iterations, which means that 100 operations are injected into each model. These numbers were chosen to keep the run time of an experiment low. With 100 model instances per class all experiments execute in a total time of about two hours on our experiment system.

It is noteworthy that these equivalence classes where chosen due to the structure of our used models. Our proposed equivalence classes focus on the analysis of call arguments according to the definition of the analyses in Section 6.2. This implies that additional equivalence classes can be constructed based on analysis definitions which examine return values and state variables. However, in our model instances these analysis are not suited as we model data flows only through call arguments. An exception is hereby that call arguments depend on return values. However, as the underlying conceptual mechanism of call arguments, return values and state variables are highly similar we would not expect to see different evaluation results for these additional classes. Therefore we decided on only evaluating the four classes proposed above for practical reasons due to the limited time for this thesis.

To answer **RQ-1.1** and **RQ-1.2** we now execute our Prolog Translator and the Prolog analysis with all chosen interpreters. The choice of interpreters is explained in Section 8.2.5. As result the interpreters return the locations of all found violations, e.g. the call stacks in combination with parameter names. We compare these detected violations with the violations that were injected into the model and apply the following classification for each violation:

- A violation that has been injected into the model and is found by the Prolog interpreter is a **true positive** (*tp*)
- A violation that is found by the Prolog interpreter but that was not injected into the model is a **false positive** (*fp*)
- A violation that is not found by the Prolog interpreter but that was injected into the model is a **false negative** (*fn*)
- A violation that that neither has been injected into the model nor is found by the Prolog interpreter is a **true negative** (*tn*)

Given this classification, we can now use precision and recall as accuracy metrics for each equivalence class. Precision and recall are commonly used metrics for classification problems. They are defined as follows[22]:

$$precision = \frac{tp}{tp + fp} \qquad recall = \frac{tp}{tp + fn}$$

We also employ the F-measure which is defined as the harmonic mean of precision and recall [25]:

$$F - measure = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

Note that these metrics cannot be used for the equivalence classes where we did not inject any violations. The reason is that the number of true positives for these classes is zero. Therefore we use the absolute number of false positives as metric for these two classes.

In chapter 7 we defined three performance optimizations for our translator which could potentially affect the accuracy. Therefore to answer **RQ-1.3** we execute our analysis pipeline of translator and Prolog interpreter twice for each model of each class for each Prolog implementation: once with all optimization disabled and once with all enabled.

For the logical negation optimization (Section 7.2) to be used it is required that negations in the analysis are altered to use the **lnot** predicate. For the shop example scenario this is not a problem as no negations are used in the query. However, the analysis code of the TravelPlanner example presented in Section 6.2.1 needs to be adapted. The **isNoRoleAuthorized** predicate contains a negation and therefore has to be changed:

```
1 isNoRoleAuthorized([], _S, _P).
2 isNoRoleAuthorized([Role | R], S, P) :-
3  (current_predicate(lnot/1) ->
4   lnot(callArgument(Stack,P,'authorizedRoles', Role));
5   (stackValid(S),\+ callArgument(S,P,'authorizedRoles',Role))
6  ),
7  isNoRoleAuthorized(R, Stack, P).
```

The term **current_predicate(lnot/1)** is exactly true when the **lnot** predicate is defined. Therefore we can use this term to switch between the optimized and the unoptimized negation based on if the corresponding optimization is enabled.

In total we execute 4800 experiments: we have two scenarios with four equivalence classes for each where each class contains 100 different models. For each model the analysis is executed for each of the three chosen Prolog implementations twice: once with all optimizations disabled and once with the optimizations enabled.

8.2.3. G2 - Scalability

A central aspect of the evaluation is the analysis of the scalability of our approach for **G2**. Hereby we quantify the impact of the size of the model on the runtime of our approach as well as the effectiveness of the implemented performance optimization.

The measurement approach is the same for all research questions of **G2**. We measure the time required given an in-memory model instance until all analysis results are available. This means that we execute the pipeline of our Prolog Translator implementation followed by the Prolog interpreter loading the resulting program and afterwards executing the analysis. We measure the timing of each stage, which therefore yields the following metrics:

- **Translation Time**: The execution duration for our Prolog Translator implementation. The input model is assumed to be already present in-memory. This time includes the IO time required for writing the result program to the disk.
- **Load Time**: The time required for the Prolog interpreter to load the program from the disk into its database. Depending on the Prolog interpreter this is the runtime of the **consult/1**, **load_files/1** or **compile/1** predicate.
- **Analysis Time**: The time required for executing the analysis query after the program has already been loaded.
- **Total Time**: The time required given an in-memory model instance until the analysis results are available. It is the sum of the translation time, the load time and the analysis time.

We use a Java implementation for automatically executing the experiments. Hereby, the Prolog interpreters are accessed through their offered Java interfaces. For measuring the timings we use the *System.nanoTime()* method.

With **RQ-2.1** to **RQ-2.5** we evaluate how our approach scales with the input model size. For each of these research questions we generate scaled instances of a base model and execute our pipeline while measuring the timings as explained above. We decided on using the TravelPlanner model without any constraint violations as base model for these experiments. We chose the TravelPlanner model over the shop example model because of its higher analysis complexity: In the TravelPlanner example the variable definitions induce longer dependency chains. In other words we expect that the Prolog interpreter is required to instantiate longer call sequence prior to finding values for variables. We therefore assumed that the TravelPlanner model is better suited for a performance analysis. In addition we assume that the worst case performance can be observed when no violations are present in the model. The reason is that then the analysis effectively proofs for the entire model that no violation exists. If violations are present in contrast the Prolog interpreter would halt as soon as a violation is found.

Depending on the research question, we scale the base model in a certain dimension by a parameter *n*. The scaling per research question is performed as follows:

RQ-2.1 We copy the entire call graph of operations and system usages *n*-times including their contained features such as variables, assignments and operation calls. In the result model we therefore have *n* disjoint copies of the TravelPlanner scenario which however do share their data types. This copying ensures that the average call graph depth for each operation stays constant with *n* while the total number of operations to analyse increases linearly.

RQ-2.2 We scale the model by using the randomized call graph scaling presented in Section 8.2.1 with *n* iterations. This implies that exactly *n* operations are added to the model. Note that these added operations are excluded from the analysis: They do not define the *roles* property and are therefore ignored. However, they do increase the length of the call sequences which need to be instantiated for the analysis of the existing operations of the TravelPlanner model. To keep the results comparable across the used Prolog implementations, we use the same randomized model instance as input for each interpreter.

RQ-2.3 In order to scale the number of parameters and return values while keeping their usage realistic we decided to copy each operations parameters and return values *n* times. Hereby not only the variables are copied but also their assignments. As a result we get *n* sets of variables, where the assignments within each set only refer to variables within the same set.

RQ-2.4 For scaling the number of attribute-value combinations we found that two approaches for scaling the model size are feasible. As first approach it is possible to add a single new *ValueSetType* with *n* values. In addition one new attribute is generated and this attribute is added to every data type. Based on our scenario examples we would expect to see a high but constant number of values per *ValueSetType* in a real world scenario. For this reason we use the following scaling approach: We generate *n* new *ValueSetTypes* with 100 values each. For each generated type, a new attribute is generated and added to every existing datatype.

RQ-2.5 For scaling the number of properties we chose a similar approach as for the scaling of the number of attribute-value combinations. Again we generate *n* new *ValueSetTypes* with 100 values each. For each type we then define a new property. A new property definition is added for each generated property to each operation. Hereby, the present values are chosen randomly, where each value has a 50% chance of being present. So on average 50 values are defined for each property definition.

In addition to measuring the scaling of our approach with the model size we also want to quantify the impact of the performance optimizations we introduced in Chapter 7. We therefore execute all experiments on each Prolog implementation with all optimizations enabled as well as with all optimizations disabled. To ensure that the negation optimization is properly used when enabled we use the adapted analysis query we presented in Section 8.2.2. This way we can measure the impact of all optimizations combined on the runtime. However, as we also find it desirable to see the impact of each optimization individually, we decided on performing three additional experiments per model where each of the optimizations is enabled individually.

As we are measuring timing information which potentially has a high variance we decided on running each experiment ten times. We can then use the mean measured time in combination with the observed standard deviation to asses the results.

With the experiments for **RQ-2.1** to **RQ-2.5** we measure the scalability of our approach with the size of the input model. Hereby, we also measure the impact of our performance optimizations on a model which we consider to be a representative use case. However,

our performance optimizations target specific scenarios which introduce performance problems. In Chapter 7 we theoretically explained why we expect the optimizations to improve the run time of our approach in these scenarios. In order to evaluate these theoretical assumptions and therefore to give a more detailed answer for **RQ-2.6**, **RQ-2.7** and **RQ-2.8** we perform additional experiments on minimal models where we inject the performance problems. These experiments allow us the evaluate if our optimizations are effective in their expected best-case in contrast to the average case which we evaluate with the experiments for **RQ-2.1** to **RQ-2.5**.

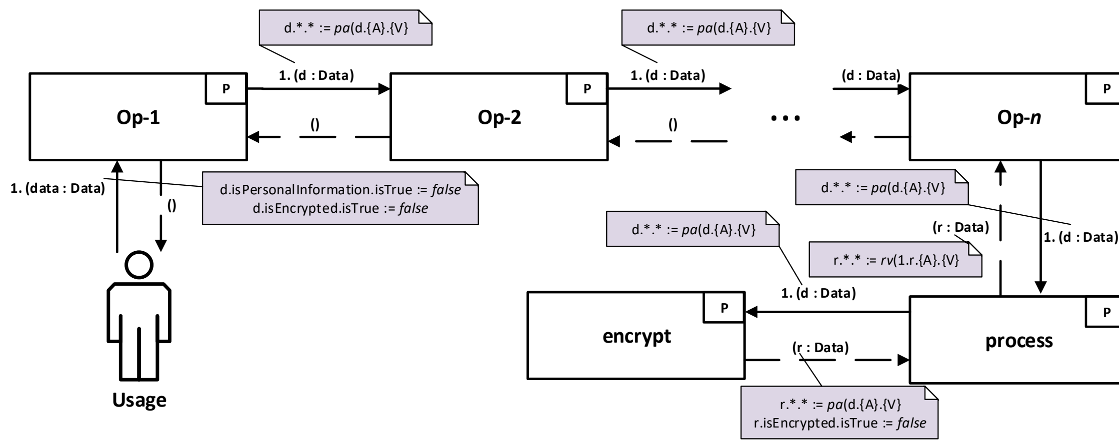

Figure 8.3.: Model used for performance testing of the indexing optimization. The number of paths to the process operation scales linearly with n

The minimal model we use for the performance measurement of the indexing optimization (Section 7.1) is shown in Figure 8.3. The only used *DataType* in the model is called *Data*. It contains two attributes: *isPersonalInformation* and *isEncrypted*. Both are boolean variables and therefore use a *ValueSetType* with only the single *Value isTrue*. The main operation of the model is *process*. This operation takes a parameter of the type *Data*, encrypts it and then returns the encrypted datum. The *process* operation is called through a chain of operations which can be linearly scaled by the scaling parameter n. This chain is started by the system usage, which initializes the datum to be unpersonal, unencrypted information. The chain of operations does nothing but assign the datum through to the *process* operation.

The analysis query we now use for this model for answering **RQ-2.6** is the following:

```
1 linearDependency(S) :-
2   S=['process'|_],
3   returnValue(S,output,'isPersonalInformation','isTrue').
```

This query tries to look for a call path where personal information is passed to *process*. Such a path does not exist. However to detext this the Prolog interpreter has to trace back along the chain of operations to the system usage. This means that about $O(n)$ predicate lookups of **callArgument** have to be performed. As the number of **callArgument** definitions also grows linear with the number of operations, we expect each lookup to

take $O(n)$ time when the first argument indexing optimization is not enabled. If it is enabled, we expect to see a lookup time per operation of $O(1)$ instead. This means that with the optimization enabled we expect a total analysis time in $O(n)$ whereas without the optimization we expect it to be in $O(n^2)$.

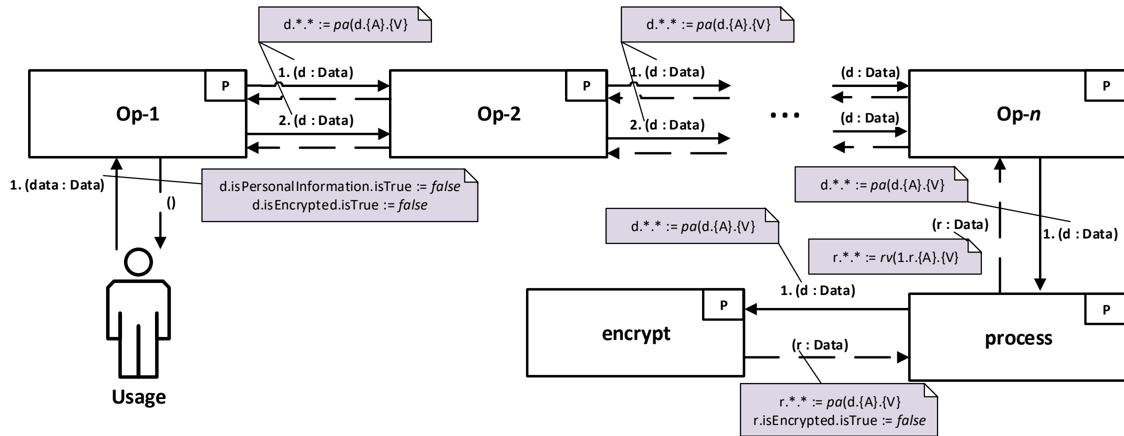

Figure 8.4.: Model used for performance testing of the negation optimization. The number of paths to the process operation scales exponentially with n

For answering **RQ-2.7** we use a very similar model which is shown in Figure 8.4. The model is exactly the same except for one difference: the operations in the call chain to *process* call their successor twice. The datum that is passed is still left unchanged. This small change now causes the number of call paths to *process* to grow exponentially. For measuring the impact of the negation optimization (Section 7.2) we use the following analysis query:

```
1 constantDependency(S) :-
2  S=['doSomething'|_],
3   (current_predicate(lnot/1) ->
4    lnot(returnValue(S,output,'encrypted','isTrue'));
5    (stackValid(S),
6     \+returnValue(S,output,'encrypted','isTrue'))).
```

With this query we look for a call stack where *process* returns an unecrypted datum. This is of course impossible as *process* always encrypts datum. However, due to the way the logical negation is implemented if the negation optimization is disabled, the interpreter has to inspect all paths to *process* to proof this. In this case we therefore expect to see an analysis time in $O(2^n)$. When the negation optimization is enabled, we instead expect that the interpreter only needs to trace back one call from *process* to *encrypt* to find the proof. Therefore, with the negation optimization enabled we expect to see an analysis time in $O(1)$.

In contrast to the other two optimizations, the cut-based assignments optimization (Section 7.3) we evaluate in **RQ-2.8** aims to improve the load time instead of the analysis time. For the best case of this optimization no additional experiments are required, as

the model scaling for **RQ-2.4** already covers the best case. We expect the optimization to perform best when the number of attribute-value combinations grows while the number of assignments remains constant. This is exactly what we do in the model scaling for **RQ-2.4**. Therefore we can use these experiments to answer **RQ-2.8**.

8.2.4. G3 - Genericness

For the goal **G3** we want to analyse our approach for further applications. Our approach was designed with two scenarios in mind: the access control scneario and the geolocation based restrictions scenario which we both explained in Chapter 3. However, we tried to keep our approach as generic as possible in order to also be applicable for other scenarios. For this reason we decided to answer **G3** by finding additional types of scenarios which can be modeled and analysed using our approach.

For finding additional scenario types we performed a literature research. We performed a keyword based search with keywords which are related to data flows and their typical potential risks. The keywords we chose are “data flow”, “security” and “privacy”. We performed our search on Google Scholar with an disjunctive linkage of these keywords on the 3rd of September. As the focus of the thesis and the evaluation lies on the scalability and not the genericness, we decided to stop when we find three additional scenario types. For each found scenario type we analyse the applicability of our approach via the corresponding research questions **RQ-3.1**, **RQ-3.2** and **RQ-3.3**. In this section we explain the key characteristics we extracted from the found scenario types. The research questions are answered in Section 8.3.3 where we analyse how these key characteristics can be modeled using the features of our approach.

RQ-3.1 The first scenario type we discovered is information flow control. It is closely related to the access control scenario we presented in Section 3.1. Two traditional models of this type are the Bell-La-Padula model [2] as well as the Brewer-Nash model [5]. Both are also access control models: They restrict the access to data based on the role of the accessing entity as well as the configured accessed rights of the accessed data. In its original form the Bell-La-Padula model hereby manages the access to files, it however can be used for any type of data other than files. In addition to restricting the read access to data, both models try to prevent unwanted information leakage. In the Bell-La-Padula model every datum has a classification level assigned, for example *public*, *secret* and *topsecret*. The model now prevents information leakage by restricting write access based on previously read data. For example, if a *secret* datum was read by a user, the user is not allowed to write to *public* data anymore: the user could potentially leak *secret* information. The Brewer-Nash model has a similar mechanism, it however does not classify the data but instead directly models conflicts using a relation.

RQ-3.2 The second scenario type focuses on the concept of secrecy and integrity of data. Both have been identified as crucial properties of data flows in certain scenarios [21] and are also covered by UMLSec [16]. The concept of secrecy ensures that no or only limited information can be extracted from data if one does not posses a key

to it. In practice, secrecy is realized using encryption schemes. The goal of integrity is to ensure that data is not modified or corrupted without being detected when transmitted from a sender to a receiver. This is most commonly realized using digital signatures, which also ensure authenticity: When data is received alongside with a signature, the receiver can check if the datum was sent by a certain sender. However, other mechanism are also usable for achieving secrecy and integrity, such as for example using a physically secure link which cannot be accessed by an adversary. The goal of a data flow analysis therefore would be to ensure that sensitive data is securely transmitted from its sender to its receiver. Hereby, it is required that for every hop the data takes one of the mechanism is active.

RQ-3.3 The third scenario type is about restrictions regarding the life cycle of managed data. This is especially considered important for data containing personal information [21, 6]. Hereby we focus on a crucial part of the lifecycle, the *deletion* of the data. Depending on the type of data a person gives to the software system, different rules may apply on how long the data is allowed to be stored. Data may for example be persisted for the duration of a session, for a certain time span or may not be persisted at all and only processed. Such rules can already be illustrated based on a simple online shop: If we consider the checkout process in a shop, it is common that multiple steps are performed during the checkout: the user enters his address and afterwards the payment method. This is often realized through separate forms which are sent to the server of the shop. Initially, the sent address may only be persisted for the duration of the user session: The user has not completed the checkout process yet and therefore the shop is not allowed to persists personal information. However, when the user submits the checkout, the data needs to be persisted for a longer time: It is required for example to handle warranty cases.

8.2.5. Prolog Implementations

The Prolog code which is generated by our Prolog Translator as explained in Chapter 6 only uses ISO Prolog features. For this reason, theoretically any ISO compilant Prolog implementation can be used. We selected the following three implementations:

- SWI-Prolog (64 bit, Version 7.6.4) [29]
- ECLiPSe (Version 7.0) [9]
- JIProlog (Version 4.1.6.1) [14]

Firstly, this choice was made for practical reasons: To answer our research questions it is required that a very high number of experiments with varying inputs is executed. Therefore, an experiment automation is required, which includes the execution of the used Prolog interpreter. All of the implementations listed above offer a Java interface through which the interpreter can be controlled and queries can be executed. These interfaces therefore allow an automated experiment execution.

Secondly, SWI-Prolog and ECLiPSe were chosen as they both implement more sophisticated predicate indexing algorithms than only first argument indexing [30, 26]. Therefore

they are especially interesting for the scalability evaluation performed for **G2**. In contrast, JIProlog was chosen with the practical applicability of our approach in mind. JIProlog is a cross-platform Java-only implementation of Prolog. Whereas the other two implementation require an installation to the system prior to usage, JIProlog is available as a single, standalone jar file which does not require an installation.

8.2.6. Experiment System Specification

All experiments are executed on a Lenovo Thinkpad P51. The relevant hardware specification of it are as follows:

- CPU: Intel Core i7-7820HQ CPU (2.9 Ghz, 4 Cores, 8 Logical Cores)
- RAM: 32 GB
- Disk drive: Samsung MZVLW1T0HMLH 1Tb solid state disk

The system uses Windows 10 (64-bit) as operation system. All experiments are executed with power plugged in to prevent CPU throttling. The installed Java Runtime Environment is OpenJDK 1.8.0.161 (64-bit). For all experiments the JVM is started with 4gb starting and maximum heap memory.

8.3. Results

In this section we present the results of our evaluation. In Section 8.3.1 to Section 8.3.3 we answer the research question for **G1**, **G2** and **G3**. Afterwards, we summarize and discuss the results in Section 8.3.4.

8.3.1. G1 - Accuracy

As result for the accuracy experiments as presented in Section 8.2.2 we can state that our approach correctly detects all covered violations for both the access control and the geolocation restrictions scenario with our used models. For all equivalence classes where we injected at least one violation we observe a precision and recall of 100%. Therefore the F-Score also is 100%. This value was observed with all optimizations enabled as well as with all disabled. We can therefore state that for these classes of models our approach finds exactly the present constraint violations without returning false positives independent of the callgraph topology.

For the model classes where we did not inject any violations we observed zero false positives. Again this was observed with all optimizations enabled as well as with all disabled. Therefore we can also state that for this classes of models our approach works accurately. In addition we found that the performance optimizations do not have an impact on the analysis results in the investigated cases.

8.3.2. G2 - Scalability

For the analysis of the scalability (**G2**) we performed 245 measurement series. As an in-depth analysis of the results of each measurement would be out of scope of this thesis we use the following approach for answering the research questions: In this section we only show a selection of measurement results. In addition we provide the plots of all other experiments in Appendix A. Our selection is made based on similarities across the measurements. When we observe the same scaling behaviour across multiple interpreters, only the plot for one is shown. The representativeness of the selected measurements can be traced using the plots given in the appendix. In addition the raw measurement data of all experiments as well as all plots have been archived online[1].

All measurements results are presented as line plots of the average measured time. In addition, vertical error bars are used to show the observed standard deviation. Therefore the error bars can be used to analyse the stability of the observed timings. In order to allow statements about the scalability we also show the Pearson correlation coefficient r for each series in the legend of each plot. The closer the value of the correlation coefficient is to one, the more likely is a linear scaling behaviour of the series. The correlation coefficient is computed based on the mean values given in the plots. As our observed measurement data only has a low variance we require a correlation coefficient greater or equal to 0.995 to assume a linear scaling.

8.3.2.1. Scalability with the number of operations (RQ-2.1)

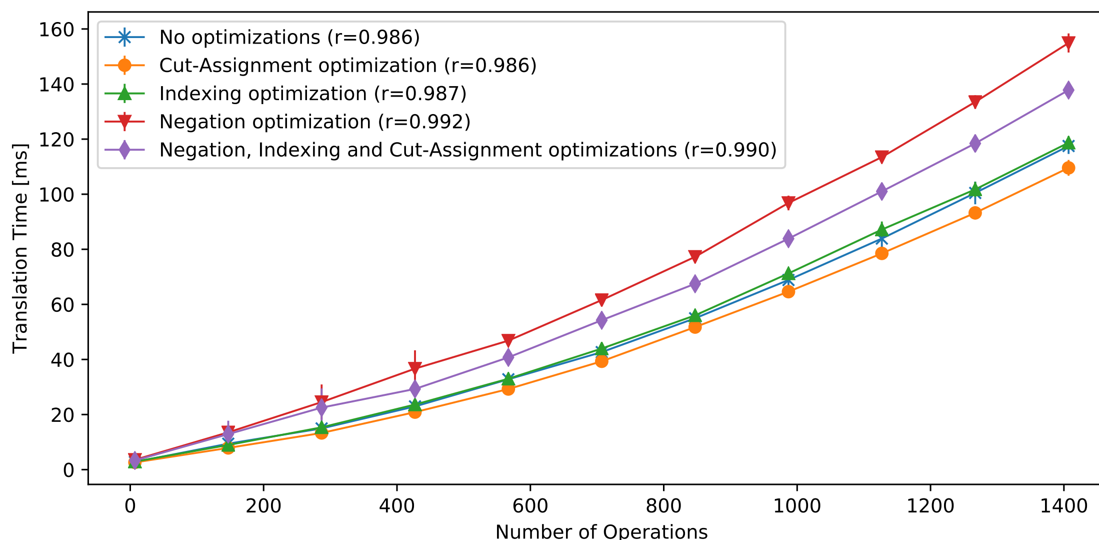

Figure 8.5.: Translation time measured for the scaling with the number of operations (**RQ-2.1**).

For **RQ-2.1** the observed translation time is shown in Figure 8.5. The plot indicates a quadratic scaling with a low quadratic component for all optimization configurations.

[1] `https://doi.org/10.5281/zenodo.1477608`

Hereby, the timing is slightly worse when the negation optimization is enabled. This can be easily explained due to the fact that more predicates have to be generated in total in this case.

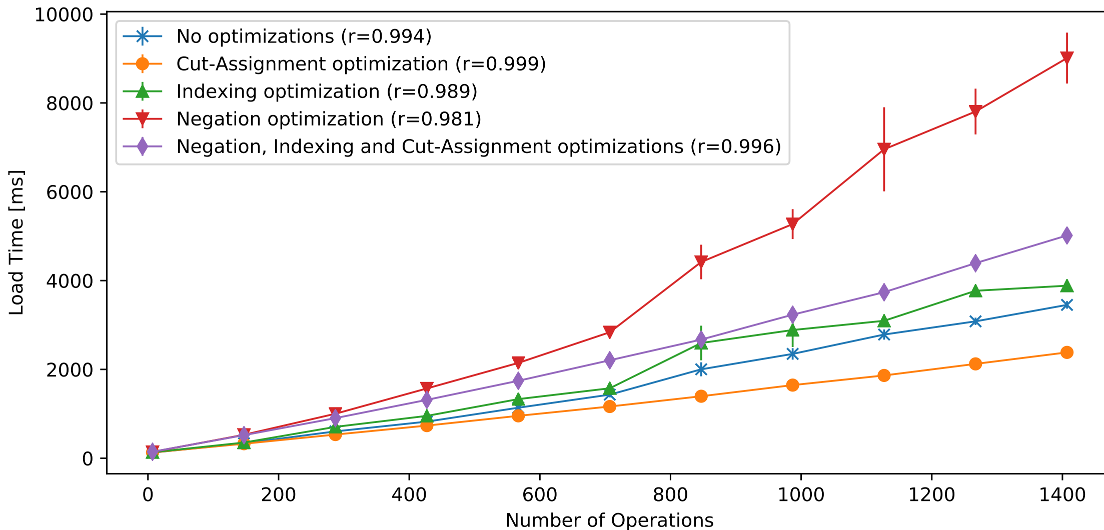

Figure 8.6.: Load time of ECLiPSe measured for the scaling with the number of operations (**RQ-2.1**).

For JIProlog and SWIProlog the measured load times grow linearly for all optimization configurations. For ECLiPSe the measured values are shown in Figure 8.6. Here, for the most configurations also a linear growth can be observed as shown by the correlation coefficient value. The slightly lower correlation coefficient value when only the indexing optimization is enabled can be explained by the variance of the measured timings. However, when only the negation optimization is enabled, the scaling seems to be super linear for the observed model sizes. This is only the case for ECLiPSe, the other two interpreters have a strictly linear growth in load time. All interpreters share the fact that the two best performing optimization configurations are the ones where the assignments optimization is enabled: This can be easily explained by the fact that this optimizations reduces the total number of rules in the program. The absolute impact of the optimizations is dependent on the interpreter.

We also observed that the analysis time for **RQ-2.1** is highly dependent on the chosen implementation. The analysis time of JIProlog is shown in Figure 8.7 and of ECLiPSe in Figure 8.8. The observed analysis times of SWIProlog are highly similar to the ones of ECLiPSe. In all cases we can see the positive impact of the first-argument indexing optimization on the analysis time: If this optimization is disabled, a quadratic scaling of the performance can be observed for all interpreters. With the optimization enabled, our approach scales much better.

For JIProlog, the scaling is still quadratic for the observed model sizes but with a much lower constant factor. For ECLiPSe the observed analysis time in contrast is near constant when the indexing optimization is enabled. An interesting fact that can be observed is that when only the negation optimization is enabled, the analysis time is worse than

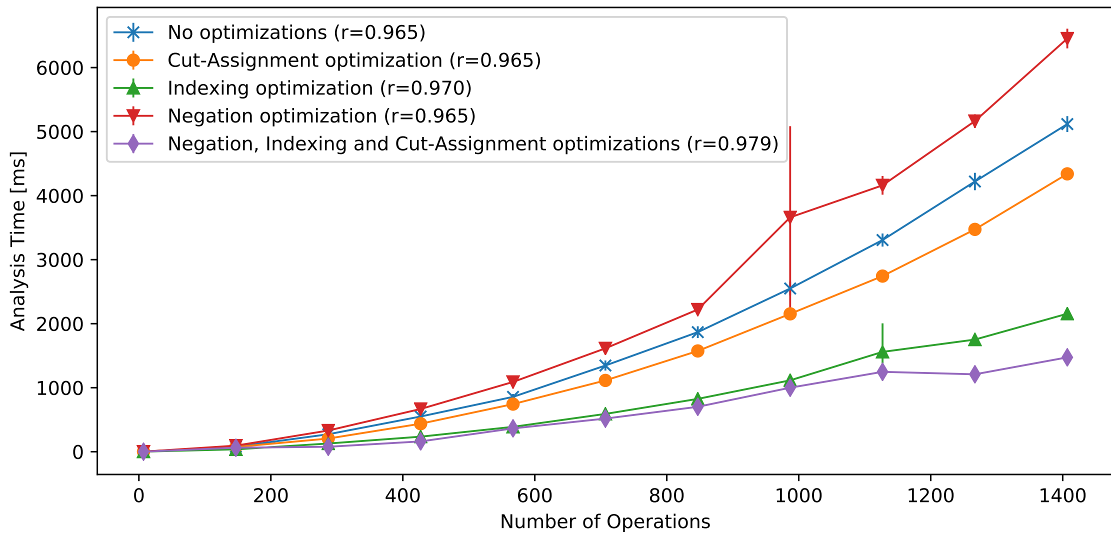

Figure 8.7.: Analysis time of JIProlog measured for the scaling with the number of operations (**RQ-2.1**).

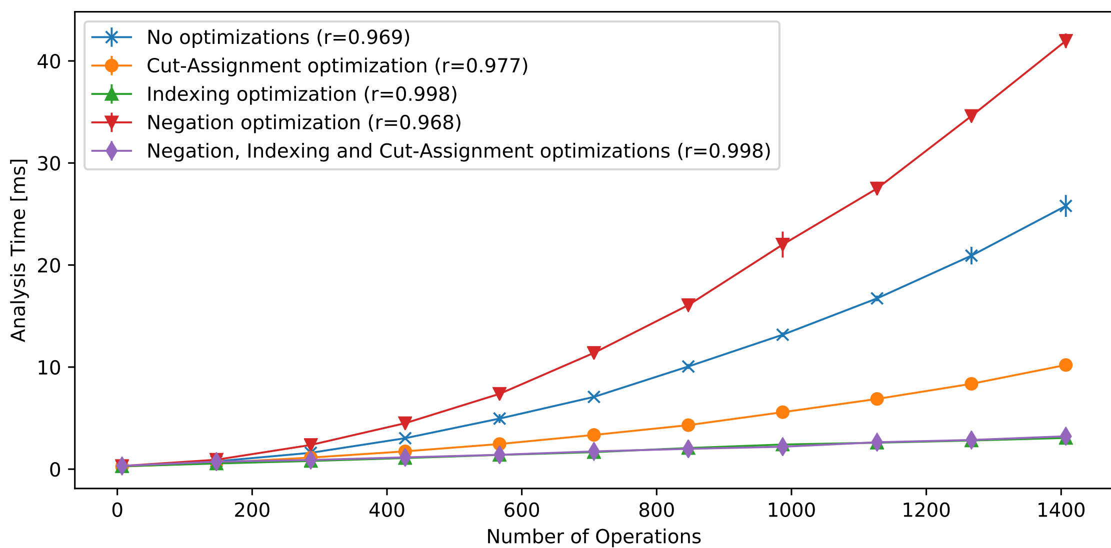

Figure 8.8.: Analysis time of ECLiPSe measured for the scaling with the number of operations (**RQ-2.1**).

without any optimizations. This can be explained by the fact that in our scenario not many different call graph paths exists, which makes the naive negation perform decently. We assume that the performance hit of the negation optimization can be explained by the overall larger fact base the interpreter has to maintain. Also it is noteworthy that the analysis time of JIProlog is much higher than the analysis time of SWIProlog and ECLiPSe. It is greater by a factor of 50 to 100. This shows that JIProlog is not as much optimized as the other two interpreters. In addition we can often observe measurement points with a

very high standard deviation and a slight higher average time for JIProlog. We assume that this is caused by garbage collection pauses as JIProlog is a pure Java implementation.

8.3.2.2. Scalability with the call graph complexity (RQ-2.2)

For **RQ-2.2** we observed highly similar results to the experiments of **RQ-2.1**. For this reason we focus on the differences in this section.

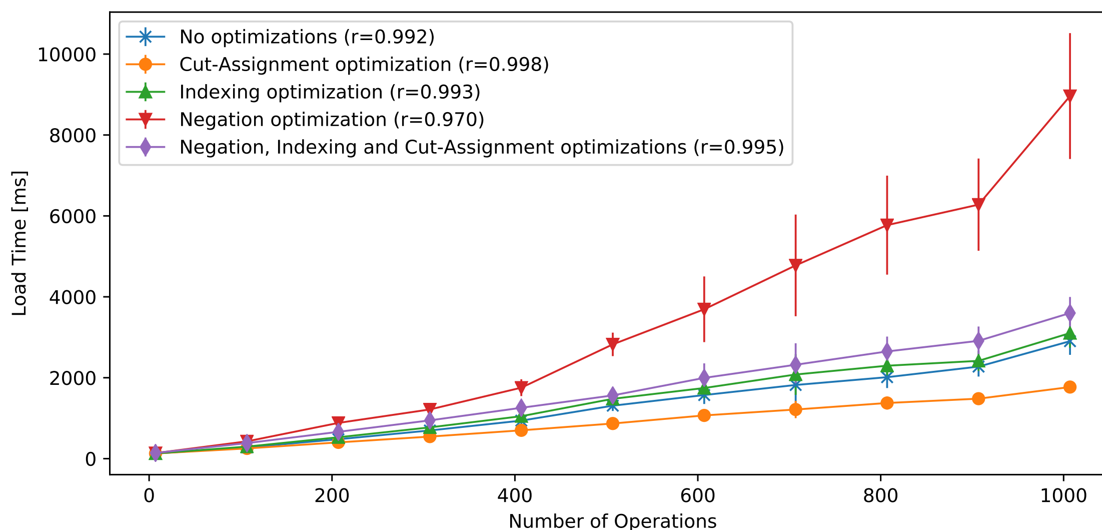

Figure 8.9.: Load time of ECLiPSe measured for the scaling with the complexity of the call graph (**RQ-2.2**).

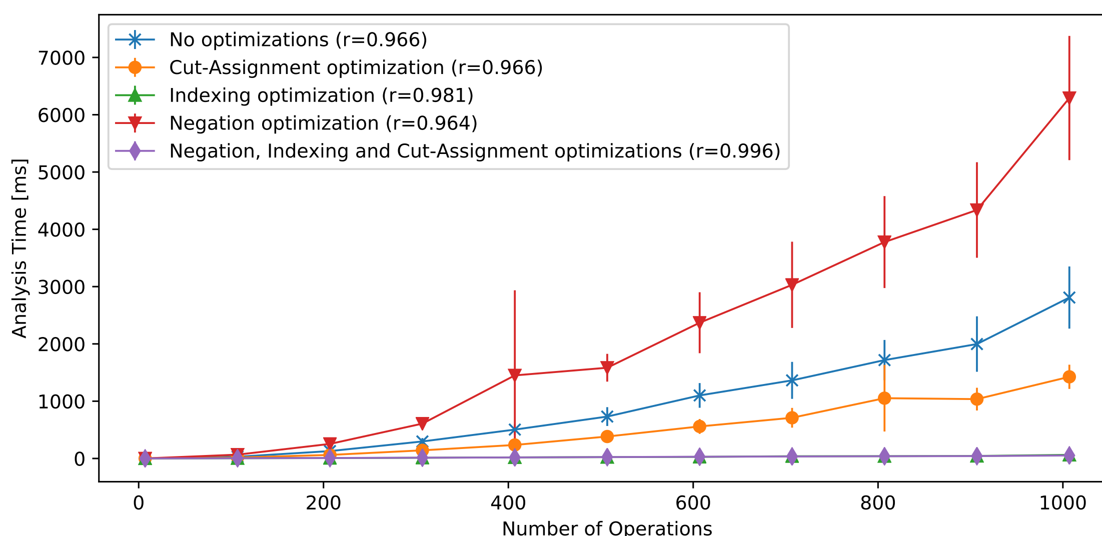

Figure 8.10.: Analysis time of JIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**).

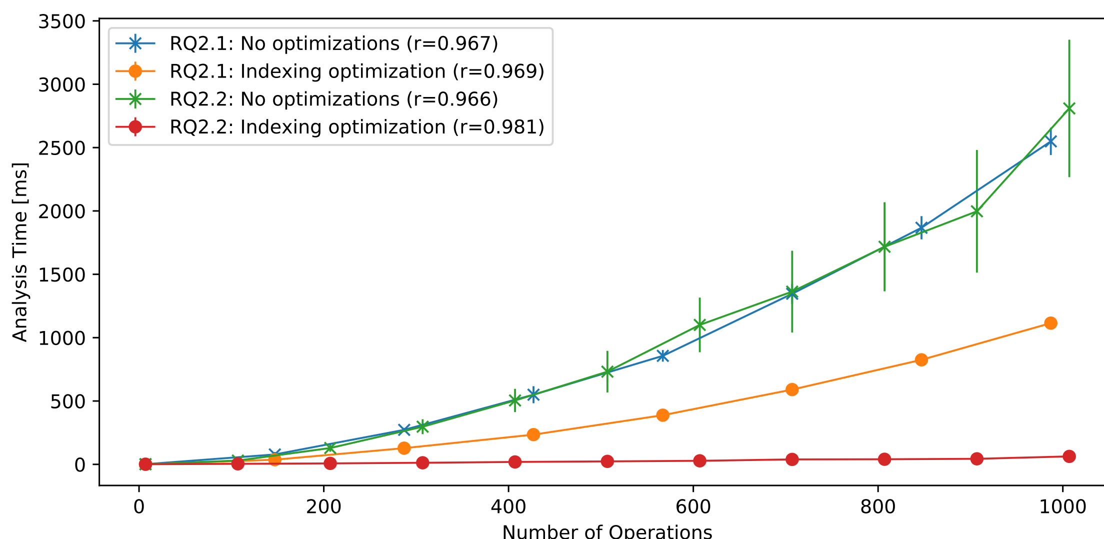

Figure 8.11.: Comparison of the analysis times of JIProlog with and without the indexing optimization for **RQ-2.1** and **RQ-2.2**.

Again, the observed translation time seems to scale quadratic with a low quadratic component as for **RQ-2.1**. The load time is shown exemplary for ECLiPSe in Figure 8.9. When comparing it to the load time for **RQ-2.1** shown in Figure 8.6 the results are almost the same. This is also the case for all other interpreters. To summarize, we observe a linear to quadratic scalability for ECliPSe and a linear scalability for JIProlog and SWIProlog.

The observed analysis time also shows the same behaviour as for **RQ-2.1** with two differences. Firstly as shown in Figure 8.11 the effectiveness of the indexing optimization has improved for JIProlog. When it is enabled, now JIProlog also shows a low linear or near constant scaling of the analysis time. Secondly, we observe a much higher standard deviation for each model size for each interpreter. This can be explained due to the randomness of the models: As we applied the randomized call graph scaling, the actual length of call sequences the interpreter has to inspect is also random. This therefore induces variance to the observed timings.

8.3.2.3. Scalability with the number of parameters and return values (RQ-2.3)

When only the number of parameters and return values is scaled we can now observe a linear translation time as shown in Figure 8.12.

This is different to the quadratic translation time observed for **RQ-2.1** and **RQ-2.2** and therefore indicates that there is possibly optimization potential in our translator implementation. According to our definition of the translation process in Chapter 6 it should be possible to implement the translation in linear time.

In contrast the load times show no different behaviour than in the experiments of **RQ-2.1** and **RQ-2.2**. Again a linear growth of the load time can be seen in the plots of all interpreters with the exception of ECLiPSe: for ECLiPSe the load time grows linearly for

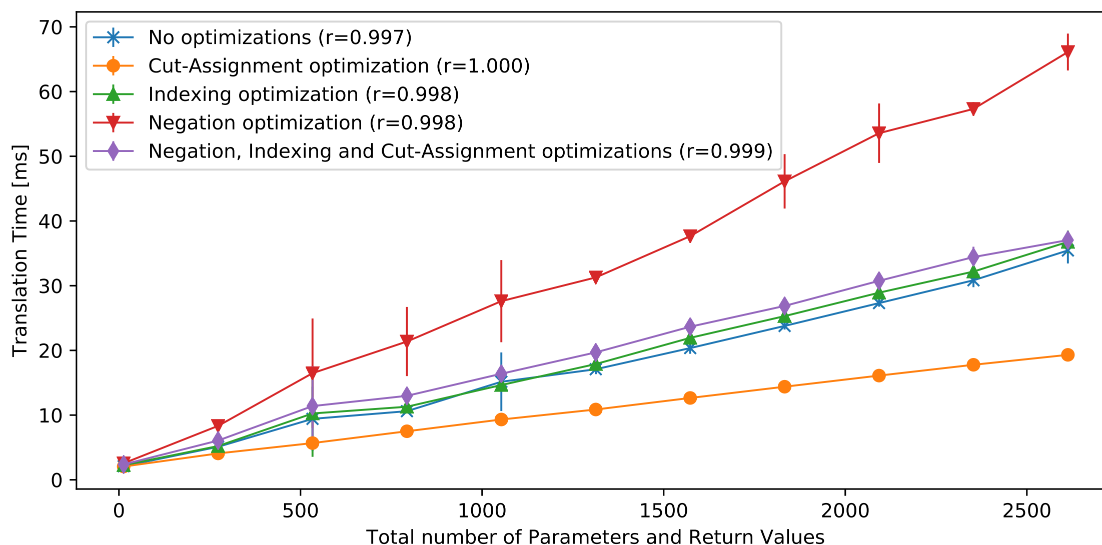

Figure 8.12.: Translation time measured for the scaling with the number of parameters and return values (**RQ-2.3**).

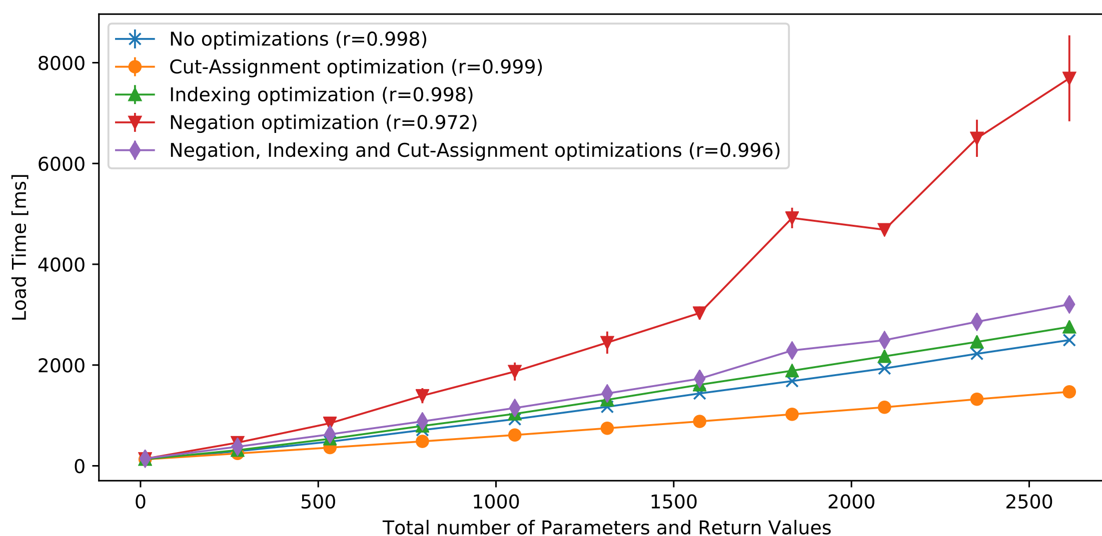

Figure 8.13.: Load time of ECLiPSe measured for the scaling with the number of parameters and return values (**RQ-2.3**).

all configurations except when only the negation optimization is enabled. In this case a super linear load time is observed. The load time of ECLiPSe is illustrated in Figure 8.13.

This is not the case for the observed analysis times: Whereas for all interpreters and all optimization configurations the analysis time scales either linear or quadratic, the impact of the performance optimizations has changed. For ECLiPSe the analysis times are shown in Figure 8.14. The figure shows that the timings scale linearly for all optimization

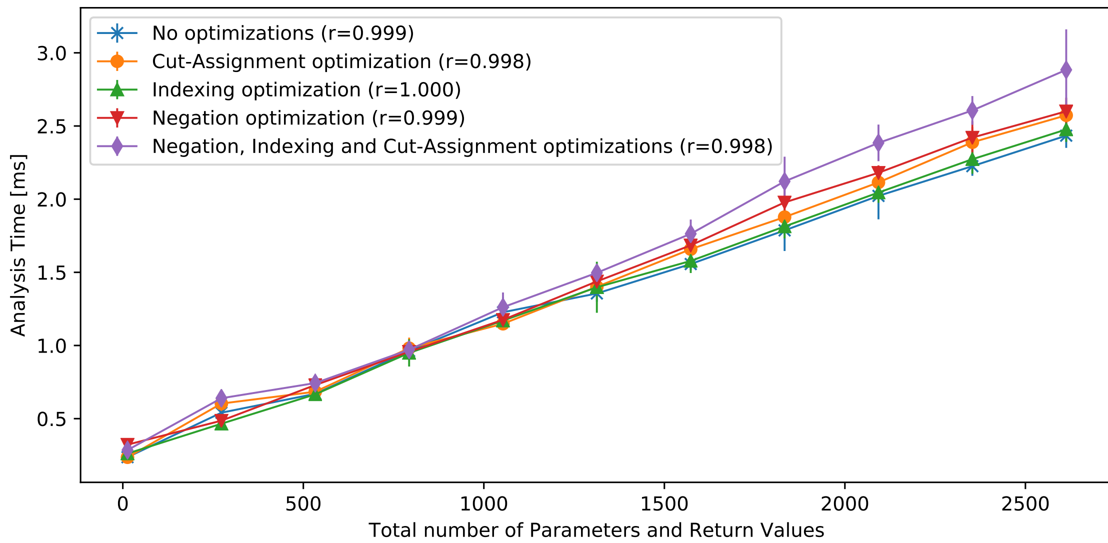

Figure 8.14.: Analysis time of ECLiPSe measured for the scaling with the number of parameters and return values (**RQ-2.3**).

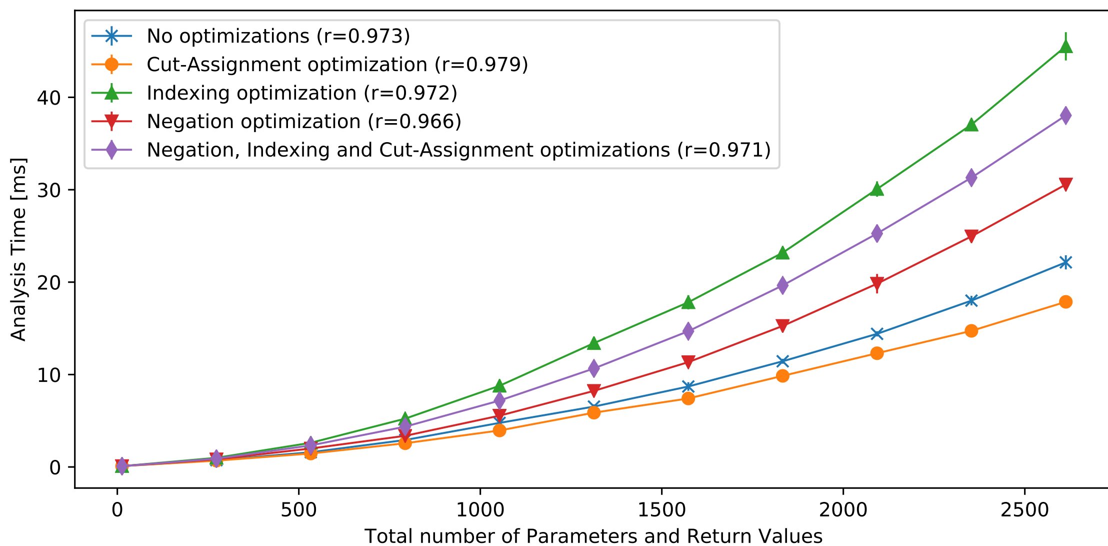

Figure 8.15.: Analysis time of SWIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**).

configurations. In addition the difference of the timings between the optimizations is very low.

For SWIProlog the results are shown in Figure 8.15. The analysis time of SWIProlog and JIProlog show almost the same behaviour, with the exception that again JIProlog is slower by a big constant factor. The observed scalability is different than the one observed for ECLiPSe: all timings scale quadratically. In addition now the chosen optimizations have a big impact. In these experiments, configurations where the indexing optimization

is enabled perform the worst. This is an unexpected behaviour as we assumed that the first-argument indexing optimization has little to no overhead. A possible explanation here is that in these experiments it is more beneficial to index by the parameter name than by the operation. Our optimization possibly misleads the SWIProlog just-in-time indexing to index the operation first. The best performance is observed when only the assignments optimization is enabled. For JIProlog this is the other way around. There the indexing optimization outperforms the assignments optimization.

8.3.2.4. Scalability with the number of attribute-value combinations (RQ-2.4)

When the number of attribute-values combinations is scaled we expect to see the best performance by the assignments optimization as the number of assignments remains constant.

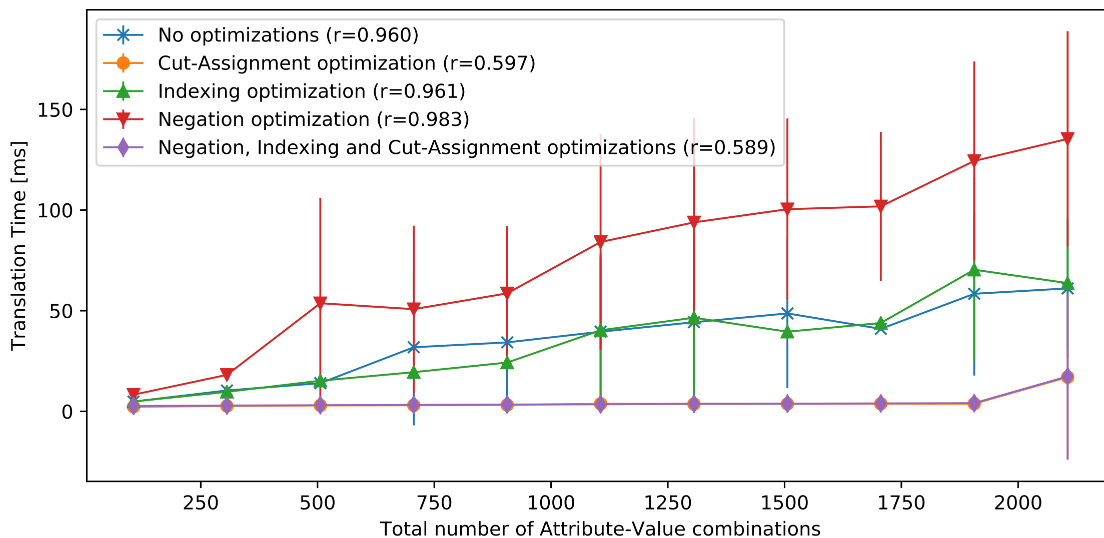

Figure 8.16.: Translation time measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

The positive effect can already be observed for the translation times as shown in Figure 8.16. When the assignments optimization is enabled, an overall linear growth of the time can be seen for the observed model sizes. However, when the optimization is enabled the load time remains nearly constant. As previously mentioned this can be explained due to the fact that with the optimization enabled one rule is generated per assignment and not per attribute-value combination.

The smaller number of rules in total also greatly impacts the load time. The load times of ECLiPSe are shown in Figure 8.17 and the one of JIProlog in Figure 8.18. The timings of SWIProlog are exactly the same as for JIProlog except that they are faster by a factor of about 10. For all interpreters we can see a near constant load time when the assignments optimization is enabled. When it is disabled, we observe a linear growth. The worst performance can be again observed when only the negation optimization is enabled: For SWIProlog and JIPRolog the load times grows linearly but with a much greater constant

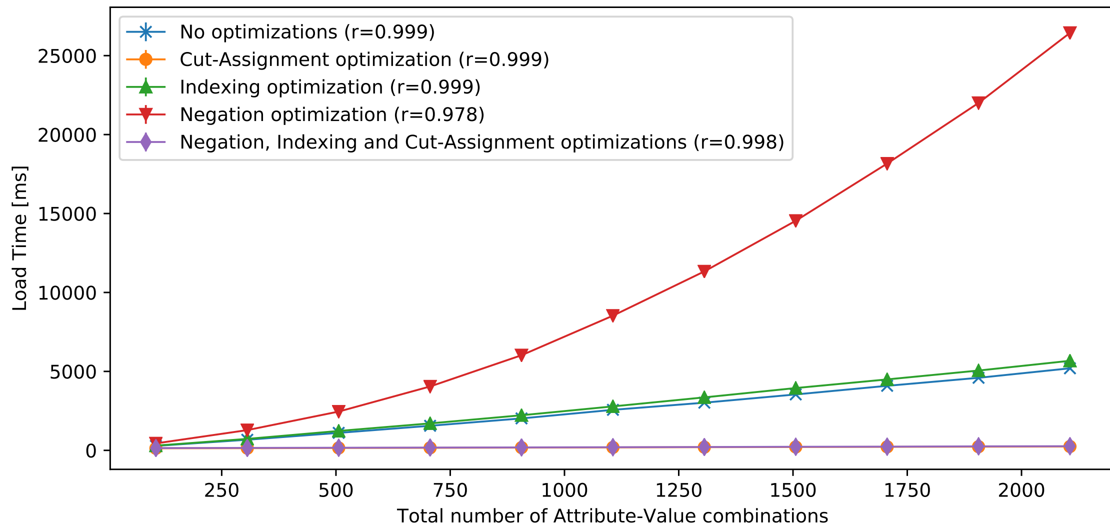

Figure 8.17.: Load time of ECLiPSe measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

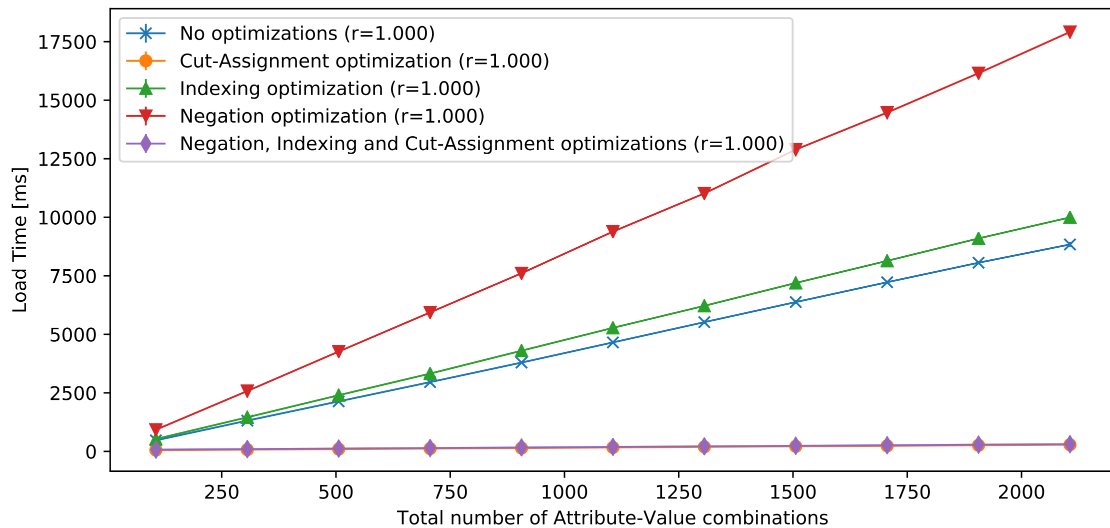

Figure 8.18.: Load time of JIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

factor. For ECLiPSe the load time even scales quadratic in this case. Even though the assignments optimization was primarily designed for reducing the translation and load time, we can also observe a positive effect on the analysis time.

For SWIProlog the analysis timings are shown in Figure 8.19. Again the scaling behaviour of the timings is similar to the one of JIProlog, except that this time JIProlog is slower by factor of 50 to 100. Even though the overall timings are relatively low, a linear growth can be observed for all cases except when the assignments optimization is enabled. In this case the timing remains constantly near zero. We expect that this is due to the fact

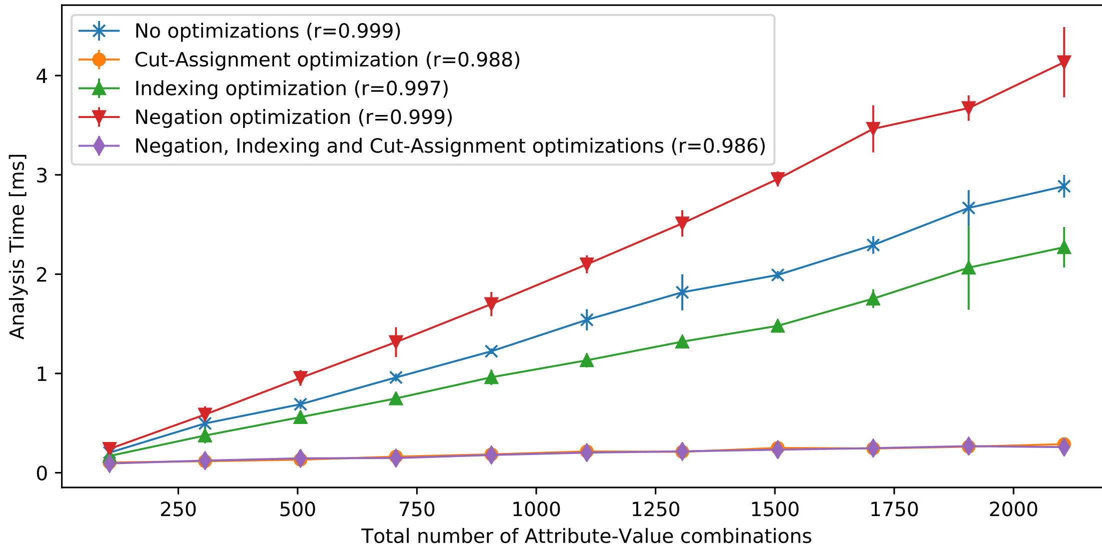

Figure 8.19.: Analysis time of SWIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

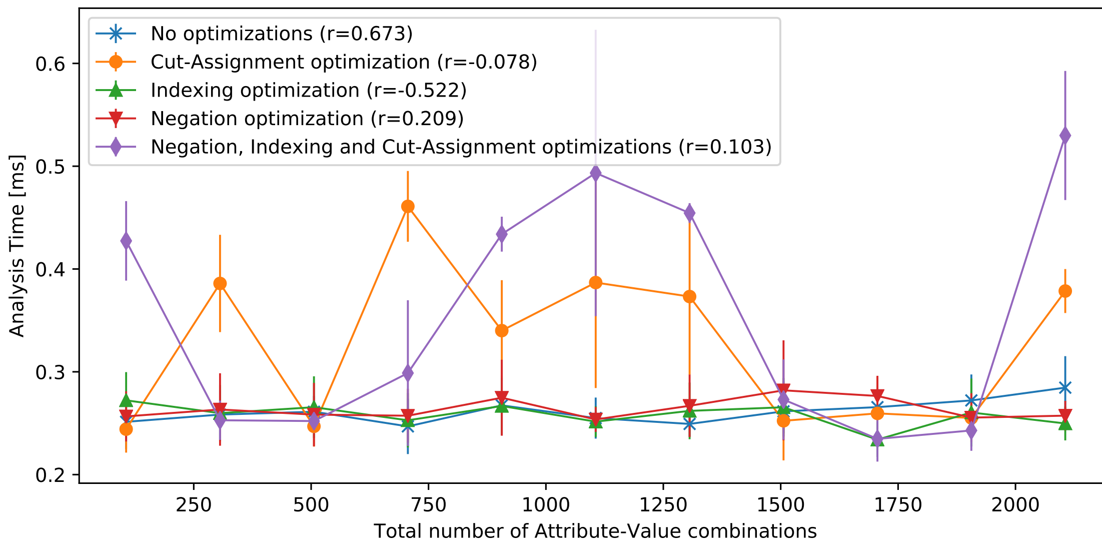

Figure 8.20.: Analysis time of ECLiPSe measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

that SWIProlog performs just-in-time indexing, which therefore is impacted by the total number of rules. For JIProlog the linear growth can be explained by the fact that JIProlog is possibly unable to index by attributes and values at all. This would also explain the much higher constant factor of the timings.

For ECLiPSe the analysis times are shown in Figure 8.20. Here the opposite effect can be observed: When the assignments optimization is disabled, the analysis time remains constant at a near zero level. When it is enabled, the observed values vary greatly with

no observable trend. However, the overall values still remain at a near zero level. To find the root cause of this behaviour we expect that it is required to analyse implementation details of ECLiPSe. As this would be out of scope for this thesis we therefore cannot give an answer for the reason of this behaviour here.

8.3.2.5. Scalability with the number of property-value combinations (RQ-2.5)

For properties we did not implement a dedicated optimization as we expect them to not have a great impact on the performance. This hypothesis is evaluated by the experiments for **RQ-2.5**.

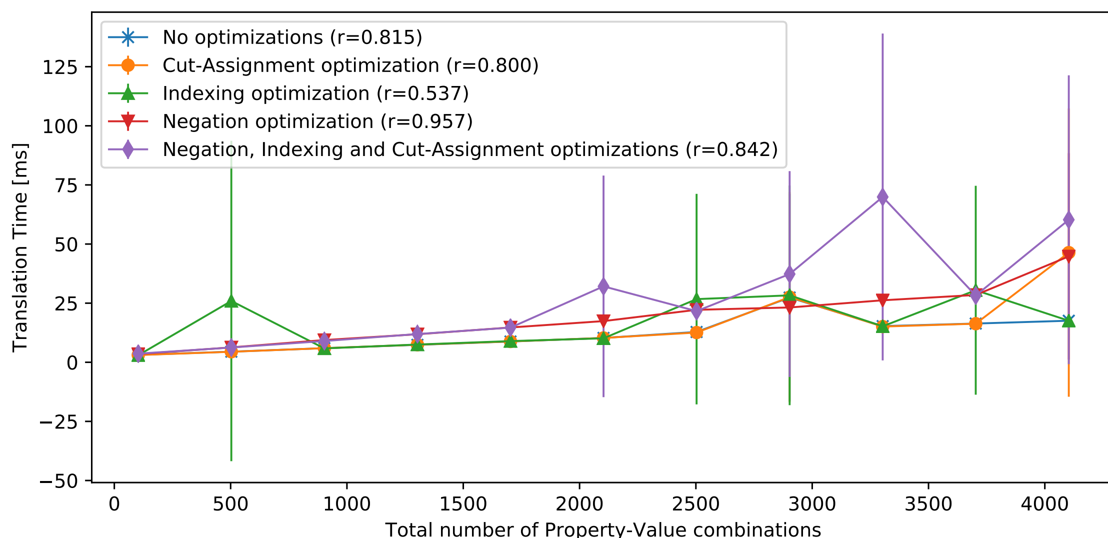

Figure 8.21.: Translation time measured for the scaling with the number of properties (**RQ-2.5**).

The translation time for **RQ-2.5** is shown in Figure 8.21. As expected, all optimization configurations perform nearly identically. With the exception of a few outliers an overall moderate linear growth can be observed.

The measured load times are also as expected. For all interpreters and all optimizations the load time grows linearly with the number of properties. The results are shown exemplary for ECLiPSe in Figure 8.22. The linear growth can be explained by the linearly increasing number of **operationProperty** definitions. Hereby, the constant factor of the growth is greater when the negation optimization is enabled. This is due to the fact in addition the negated **operationProperty** definitions are present in the program.

The results for the analysis times are similar to the ones for the load times. For SWIProlog the timings are presented in Figure 8.23 and for ECLiPSe in Figure 8.24. Again, the results of JIProlog are the same as the ones of SWIProlog with the exception of one outlier. We assume that this outlier is caused by a garbage collection.

For ECLiPSe the analysis times are nearly constant independently of the enabled optimizations. For SWIProlog and JIProlog the analysis time grows linearly. Hereby, the times are slightly worse when the negation optimization is enabled. For SWIProlog we

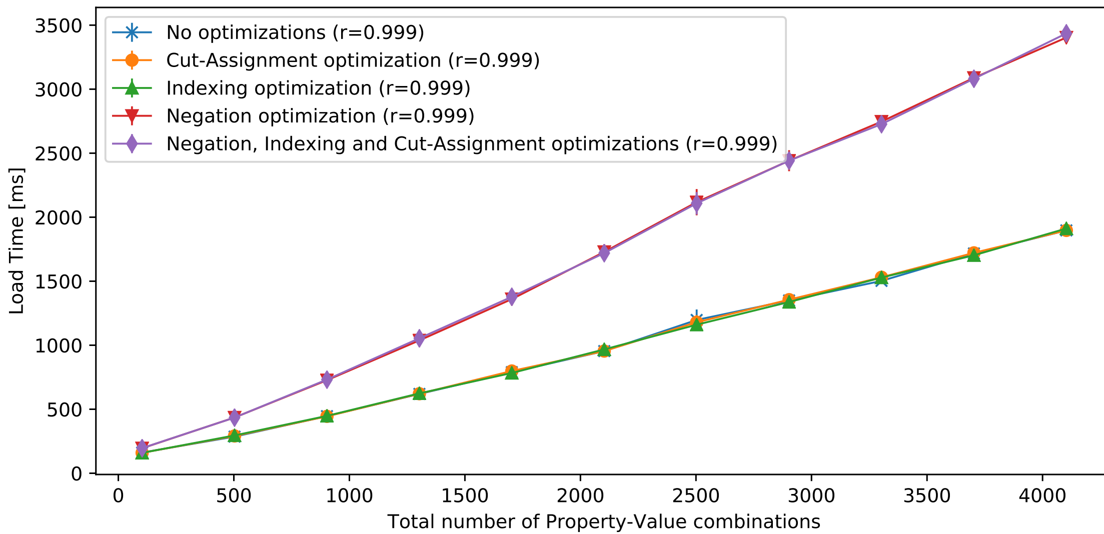

Figure 8.22.: Load time of ECLiPSe measured for the scaling with the number of properties (**RQ-2.5**).

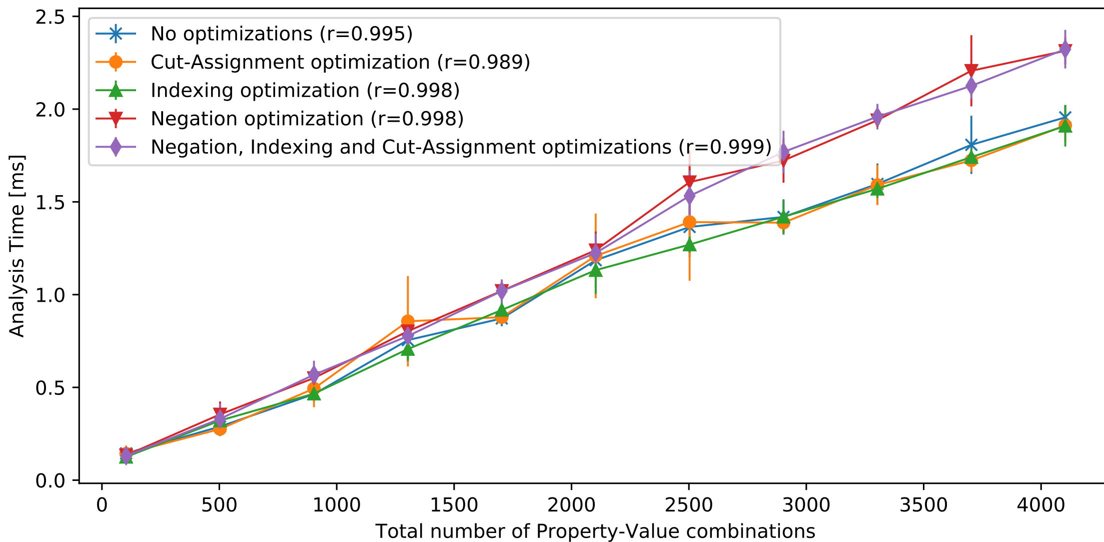

Figure 8.23.: Analysis time of SWIProlog measured for the scaling with the number of properties (**RQ-2.5**).

can again assume that the growth is caused by the just-in-time indexing performed by the interpreter.

8.3.2.6. Effectiveness of the indexing optimization (RQ-2.6)

The results of **RQ-2.1** and **RQ-2.2** already indicated the positive effect of the indexing optimization on the analysis time. With the experiment design for **RQ-2.6** we aimed to show the best case of the first-argument indexing by using a specialized model.

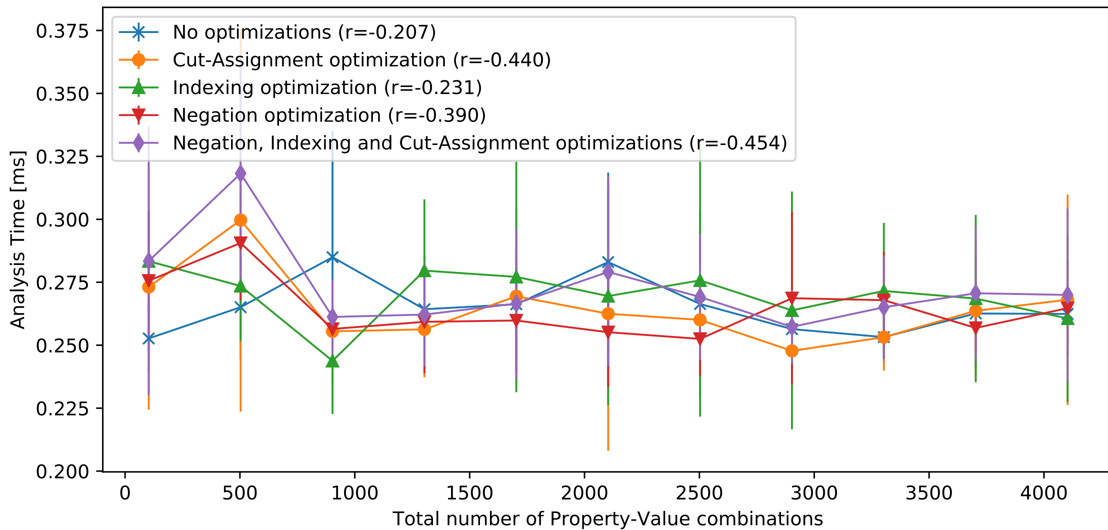

Figure 8.24.: Analysis time of ECLiPSe measured for the scaling with the number of properties (**RQ-2.5**).

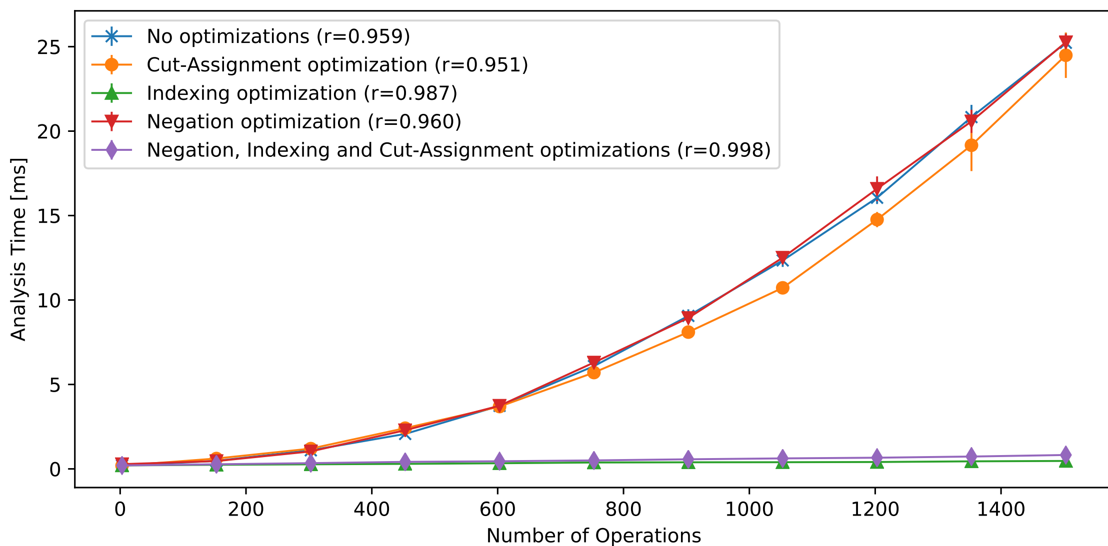

Figure 8.25.: Analysis time of ECLiPSe measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

As expected, the translation time and the load times show no difference to the observation we made in the experiments for **RQ-2.1** and **RQ-2.2**. For this reason, we solely focus on the analysis time in this section. The analysis times of ECLiPSe are shown in Figure 8.25 and of SWIProlog in Figure 8.26. The plot for JIProlog has been omitted as the growth of the analysis time are the same as for ECLiPSe.

For all three interpreters we can observe a quadratic growth of the analysis time when the indexing optimization is not enabled. When it is enabled, we can see a very low linear, near constant growth for JIProlog and ECLiPSe at least for the observed model sizes. For

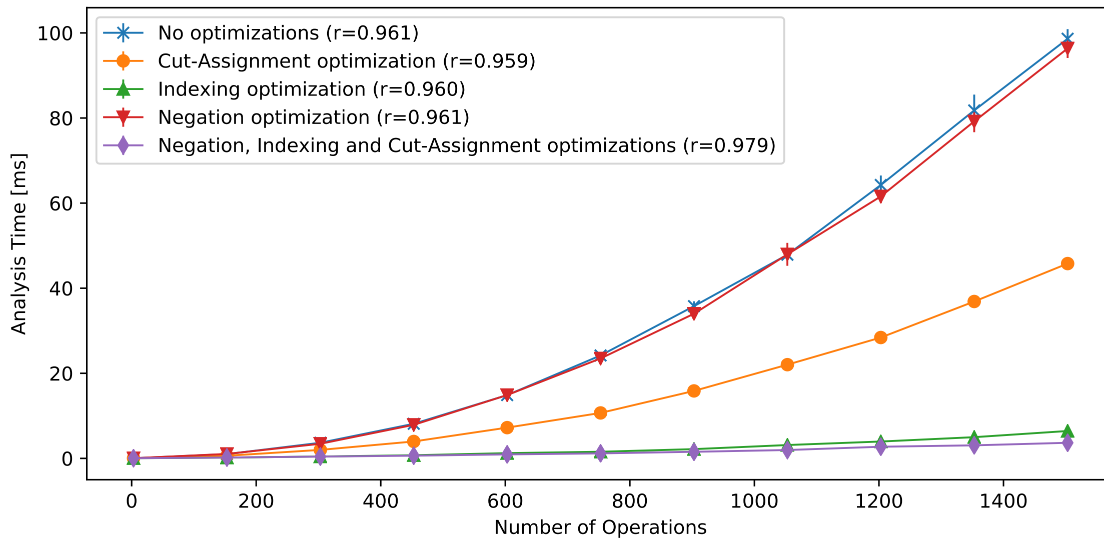

Figure 8.26.: Analysis time of SWIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

SWIProlog the growth seems to be quadratic, but with a very low quadratic factor. These are exactly the results we expect to see based on our hypothesis that the interpreters are unable to use first-argument indexing on the **callArgument** and **returnValue** predicates when the indexing optimization is disabled. Hereby, a linear scaling is possible because with perfect indexing the lookup of a single **callArgument** or **returnValue** is algorithmically possible in $O(1)$. As the proof for the model requires a linear number of such lookups, a total analysis time in $O(n)$ therefore is possible. For SWIProlog we can explain the still quadratic growth again by just-in-time indexing: As the indexing possibly requires $O(n)$ time, the overall scaling can be quadratic. However, as shown by our measurement the analysis time is still greatly reduced in comparison to when the indexing optimization is disabled.

An unexpected observation is the fact that the assignments optimization also has a positive impact on the analysis time. Even though the used datatype in the model has only two attribute-value combinations, the assignments optimization improves the performance for SWIProlog. This means that the assignments optimization only reduces the total number of rules by a factor of about two. However, this reduction of the size of the fact base already seems to have an positive impact on the performance. For ECLiPSe and JIProlog only a minor improvement can be seen.

8.3.2.7. Effectiveness of the negation optimization (RQ-2.7)

For the negation optimization the experiments have only shown a negative impact of this optimization so far. In almost all experiments, the translation and load time has increased by a factor slightly less than two. The analysis time has mostly been unaffected.

The reason for this bad performance of the negation optimization is that it is designed for a scenario which is not present in the TravelPlanner example. For the negation optimization

to become effective it is required that proofs for negated terms can be found faster than it is possible to try out every existing call sequence. For this reason we designed the still practically relevant model for the experiment of **RQ-2.7** as explained in Section 8.3.2. This model illustrates-the best case where the number of call sequences grows whereas the actual complexity of the proof remains constant.

Just like the experiments for **RQ-2.6** the translation time and the load time can be observed to grow linearly. Hereby, the timings are slower by factor of up to two when the negation optimization is enabled.

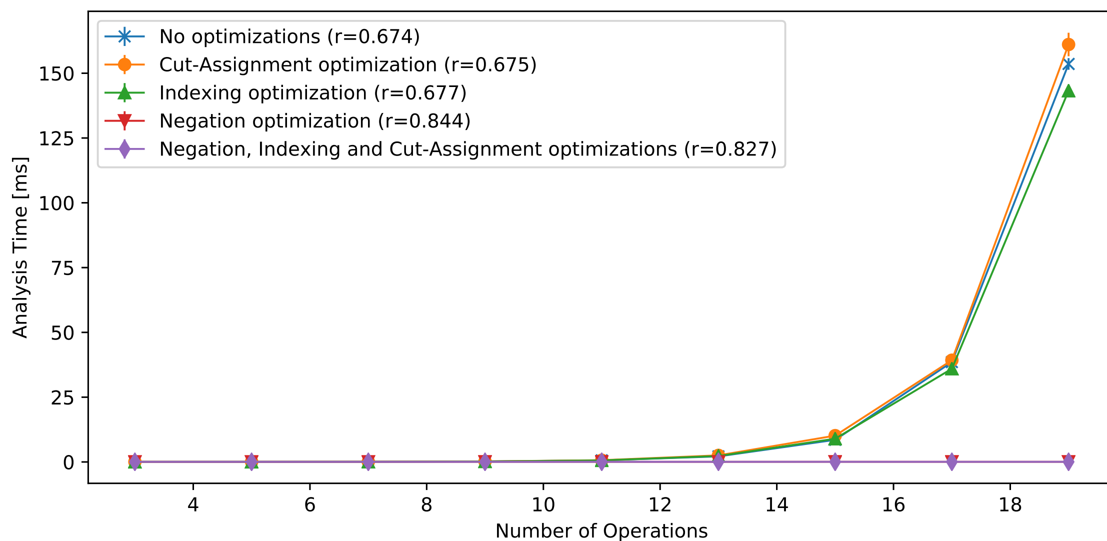

Figure 8.27.: Analysis time of SWIProlog measured for the negation optimization experiment (**RQ-2.7**).

The effectiveness of the negation optimization is illustrated by the analysis time shown in Figure 8.27. In the figure the timings are shown for SWIProlog, however the observed timings are the same for ECLiPSe and JIProlog with different constant factors. When the negation optimization is enabled, the analysis time remains constant at a near zero level. However, when it is disabled the time grows exponentially.

This therefore verifies our hypothesis from Section 8.3.2. Without the optimization, the interpreter has to try every call path in order to proof a negation. With the optimizations enabled, negations can be used freely with only a constant impact on the translation and load time of the resulting program. In this case, the analysis time does not get negatively affected by the use of negations.

8.3.2.8. Effectiveness of the cut-based assignments optimization (RQ-2.8)

In order to answer **RQ-2.8** we use the results of the experiments for **RQ-2.4**. The cut-based assignments optimization targets scenarios where the number of attribute-value combinations is higher than the number of assignments. In these cases, we expect to see an improved load and translation time due to the decreased number of rule definitions.

In the experiments for **RQ-2.4** we clearly observed these benefits. As shown in Figure 8.16 the assignments optimization causes the translation time to stay nearly constant. The same results can be observed regarding the load time: As shown in Figure 8.17 and Figure 8.18 the load time also remains constant when the assignments optimization is enabled.

An unexpected benefit of the assignments optimization is that we observed an improved analysis time for SWIProlog and JIProlog. We expected that the total number of rule definitions would not affect the analysis time due to the presence of first-argument indexing. However, for both SWIProlog and JIProlog the optimization caused the analysis time to stay constant whereas without the optimization it grows linear. For SWIProlog this is shown in Figure 8.19.

In the experiments for **RQ-2.4** we observed the best case for the assignments optimization. However, the results for other experiments, for example for **RQ-2.1** showed that even in other scenarios this optimization provides performance gains. As shown in Figure 8.6 the best load time was observed when only the assignments optimization is enabled. In contrast to the results for **RQ-2.4** the overall scalability is not affected, instead the timings are just improved by a constant factor.

In general it is possible to deduce from the experiment results that the assignments optimization is an optimization with no downsides performance wise. We observed no metrics where enabling the optimization caused a decreased performance.

8.3.3. G3 - Genericness

In Section 8.2.4 we identified three additional scenario types for which our approach should be applicable. For each scenario type we identified key characteristics which should be covered by our meta model and analysis. In this section we therefore answer **RQ-3.1**, **RQ-3.2** and **RQ-3.3** by explaining how these key characteristics can be modeled and analysed using our approach.

8.3.3.1. Information Flow Constraints (RQ-2.8)

To answer RQ-2.8 we exemplary show how the key features of the Bell-La-Padula model can be modeled and analysed using our approach. First of all, the Bell-La-Padula model provides access control similar to our scenario presented in Section 3.1. A core difference is that the access control does not work role based but based on classification levels. For example the ordered set of classification levels could be public, secret and topsecret. These could be realized as a *ValueSetType*. This means that a datum which is for example classified as secret may only be accessed by a user who has the access level secret *or* topsecret. This implies that a comparison operator needs to be implemented for the classification levels. As the number of levels is finite, this comparison can be realized as a boolean function and therefore using our *LogicTerms*.

The second part of the model is fundamentally different from the access control model we presented in Section 3.1: the information flow control. This means that when a user has read highly classified data, he is not allowed to write to lower classified data anymore. In order to implement this, we first need a way of explicitly specifying when data is

read and written. For this purpose two operations can be defined: *read(data: Data)* and *write(data : Data)*. Whenever now any operation semantically reads or writes a datum, the corresponding operation calls these operations. The *Data* data type hereby only contains the classification level of the datum. The model needs the ability to remember the highest classification level of the data a user previously read. When assuming that we are in a single user scenario, this can be done using a state variable. The read operation defines a state variable *userState : State*. Hereby *State* has only a single attribute which stores the highest classification level of previously read data. The read operation now has to define post execution state changes where userState is updated. This update is again just a boolean function which can be realized via our *LogicTerms*.

With this infrastructure, the formulation of the analysis is straightforward: We look for a call sequence to *write(data : Data)*, where the given datum is lower classified than the level stored in the userState variable. For the comparison we can reuse the same boolean function as for the access control.

In this model we assumed a single-user scenario. If multiple users are present, the user state cannot be realized as a state variable but instead has to be a parameter of the read and write operations.

8.3.3.2. Data Flow Secrecy and Integrity (RQ-3.2)

To answer RQ-3.2 we show how to ensure the secrecy and integrity of flowing data. Hereby the goal is to ensure that these properties are present for each flow. The goal is not to show that a given encryption or signature algorithm is secure. We model three approaches for achieving these properties on a high level: encryption, digital signatures and physically secure links.

The modeling of encryption and digital signatures is trivial. In the most simple case we can just add the boolean attributes *isEncrypted* and *isSigned* to every datum. If these attributes are set to *true*, the datum is assumed to be encrypted and signed correspondingly. This simple modeling can also be expanded if multiple parties and therefore public-key cryptography is involved. In this case a *ValueSetType* is required for modeling the identity of the parties. The parties can be identified based on their public keys. Therefore the *ValueSetType* is defined as *partyIds* = $\{pk_1, pk_2, \ldots, pk_i\}$. We can now use this type as type for the attributes *isEncrypted* and *isSigned*. The meaning for *isEncrypted* is now changed that the datum has been encrypted with the selected public keys. Similarly the value of *isSigned* now states with which public keys the signature can be verified. To model the identity of operations a property can be added which lists the public keys to which the operation knows the private keys.

The modeling of a physically secure link is more complex. The link of data flow is implicitly represented in our model by the *OperationCalls*. However, in our model it is not possible to add properties to *OperationCalls* which makes it impossible to distinguish between secure and unsecure links. This issue can be overcome by modeling the links as operations instead. Similar to the inserted mediator operation used in Section 8.2.1 an intermediate operation is placed in every call. These operations represent links and therefore just pass their call arguments through. In order to distinguish between links and

normal operations in the analysis a property is required to identify these link operations. In addition a property is required for specifying whether the link is secure or unsecure.

With this modeling the analysis to ensure secrecy and integrity is now trivial. For every hop a datum takes it either has to be encrypted or the hop must be across a secure link for secrecy to be achieved. Therefore (a) the *isEncrypted* attribute of the datum has to be set accordingly or (b) either the source or the receiver of the flow must be a link operation which is secure. For integrity the analysis looks the same: (a) the *isSigned* attribute of the datum has to be set accordingly or (b) either the source or the receiver of the flow must be a link operation which is secure.

8.3.3.3. Data Life Cycle Management (RQ-3.3)

For answering RQ-3.3 we focus exemplary on one critical part of the life cycle of data: the deletion.

First of all it is required to model the allowed lifetime for the data. This can be done by adding an attribute to the data whose *ValueSetType* specifies the lifetime. For example possible values for the *lifetime* attribute could be *processingOnly*, *sessionPersistence* and *longTermPersistence*. The classes required have to be identified based on the concrete system which is being modeled.

The second part that has to be modeled is how the data is used persistence wise. Therefore we add a property named *persistenceType* to every operation. As *ValueSetType* we use the same type as for the *lifetime* attribute. However, the semantic is a little different: The property now states how *every* incoming datum is persisted: does the operation only process the data and not persist it at all or is it a database persisting the data for a long time? Note that we require that this persistence level is the same for every incoming datum. This strict requirement is needed for the analysis. How can operations be modeled which have multiple persistence levels? For example what about a servlet which (a) only processes some data and (b) stores some data for the time of the session? Such operations have to be split into multiple operations. In our example the operation can be split in a *Servlet* operation and a *ServletStorage* operation. The *Servlet* operation has *processingOnly* as *persistenceType* whereas *ServletStorage* has *sessionPersistence*. *Servlet* now receives all the data but only delegates the data to be persisted to *ServletStorage*.

The strict requirement that *persistenceType* specifies the persistence for every incoming datum makes it possible to formulate the analysis. We can now simply compare the *lifetime* attribute of all parameters with the *persistenceType* property for every operation. This comparison can be written as a propositional logic formula as the number of combinations is finite.

8.3.4. Summary

With our evaluation goals **G1**, **G2** and **G3** we aimed to show that our approach is accurate, scalable and also applicable to additional scenarios.

For **G1** the evaluation showed that our approach accurately detects constraint violations for our access control and geolocation restrictions scenario presented in Chapter 3. Hereby in all tried models all violations have been found without any false positives. As we

employed the call graph randomization technique (Section 8.2.1) for generating the model instances we can state that the findings are independent of the topological structure of the model.

We analysed scalability of our approach (**G2**) regarding two main aspects: (a) the scalability with the size of the input system model and (b) the impact of our implemented performance optimizations. The results showed that at worst our approach scales quadratically with the size of the input model. However, in most cases with our optimizations enabled the performance scaled either linearly or only with a very low quadratic component. We also observed that the first argument indexing and the cut based assignments optimizations never reduced the performance by a significant factor and can therefore be always enabled. The logical negation optimization comes at the cost of a higher constant factor regarding the load time of the model, it however potentially reduces the analysis time from exponential to constant scaling.

As a side effect we discovered that the absolute run time highly depends on the chosen Prolog implementation. In most experiments ECLiPSe is the fastest at analysing but has a longer load time. SWIProlog is slightly slower performing the analysis but has a much lower loading time. For JIProlog the loading times are similar, however the analysis times we observed usually are slower by a factor of about 50 to 100.

To answer **G3** we selected three additional scenario types and evaluated how well they can be modeled with our approach. The scenarios we selected are information flow control, secrecy and integrity enforcement and data lifecycle management. For all three we showed that it is possible to model scenarios in these domains using the features our model and analysis API offer. However, the modeling often required special tweaks, such as the modeling of physically secure links using additional operations.

8.4. Threats to Validity

In this section we briefly discuss the threats to the validity of our evaluation. Hereby we apply the classification scheme for threats to validity proposed by Runeson et al. [24, Section 5.4].

Threats to internal validity are threats where factors the researcher is unaware of possibly influence the results. If such threats are present it is questionable that the experiments actually can be used for answering the research questions. For our evaluation of the accuracy (**G1**) we only used two base models for the evaluation: one for the access control scenario and one for the geolocation based restrictions scenario. Therefore a potential internal threat is that not all features of our approach are covered by these models. As we already mentioned in Section 8.2.2 our analyses only examine call arguments, which can depend on return values in our used models. State models have not been used as they were added later to our approach. However, as explained in Section 5.2, state variables are effectively realized using the same techniques as parameters and return values. In addition even though our analyses do not directly reference return values, the models have call arguments which depend on return values. For this reason return values are also covered by our evaluation. Because our evaluation shows that parameters and return values work accurately, we assume that with a high probability there are no conceptual issues with

state variables either. However, we cannot exclude implementation errors at this point. For this reason it would be beneficial to evaluate return values and state variables through additional experiments in future work.

For the scalability evaluation (**G2**) we considered three influence factors in our experiment design: The size of the model in multiple dimensions, the enabled performance optimizations as well as the used Prolog implementation. Hereby, a minor threat we see is the possibility that we missed out an important dimension of the model. However, as we chose the dimensions to scale the model instances based on the structure of the meta model as well as the translation process, we classify this as a minor threat.

An additional threat to internal validity can be identified regarding the evaluation of the genericness (**G3**) of our approach. To evaluate the genericness, we identified additional scenario types and explained how their key characteristics can be modeled using the features of our approach. A possible threat is the fact that we might have missed some important features of these scenario types. However, as the features we identified are sufficient for building functional models, we therefore assume that we covered at least the most important ones. Regarding that the focus of our work lies on the scalability of our approach, we therefore consider this a minor threat.

Threats to the external validity of our evaluation impact how well our findings can be generalized. For the experiments regarding accuracy a threat is induced by the fact that we investigated only two specific model instances. However, both models have been designed to cover the important aspects of the scenarios they target. In addition, we employed the call graph randomization technique (Section 8.2.1) during our analysis. Whilst this technique does not fully randomize the model, at least the topological structure of the callgraph is randomized.

For the analysis of the scalability there is always the threat that the chosen input sizes are to small to uncover the actual scalability. As we executed very many experiments, we had to keep the per experiment run time low for practical reasons. However, we can assume that the size of the model instances we analysed is big enough in comparison to the model size we would expect in a real world usage of our approach at least for small systems. The models we used as input for the scalability experiments are up to 200 times the size of the base TravelPlanner model presented in Section 5.7.1. Still when all optimizations are enabled we observed very low run times in many experiments. In these cases more measurements with larger models as future work are required to fully answer the question of the scalability.

The external validity of the genericness evaluation is threatened by the fact that we did not explicitly build example model instances, but instead only showed how key concepts of the scenario types can be realized. Therefore no experimental proof is given that this realization works. Nevertheless, we showed the functional correctness of most of the features our approach provides experimentally through (**G1**). As the realization of the key concepts of the scenario types we presented using these features is straight forward, we can assume that this threat is minor.

8.5. Assumptions and Limitations

During the design and implementation of our approach we made several assumptions and design decisions which in turn induce limitations. In this section we briefly discuss these assumptions and limitations.

One central assumption we made during the design of our meta model is that we assume that all variables of the system can be expressed as boolean values via *ValueSetTypes*. This assumption was made as these types of variables are sufficient for our two main scenarios, access control and geolocation based restrictions. Therefore this allowed us to provide a working implementation of our meta-model and the Prolog translator despite the limited time available for the thesis. Note that this limitation is not induced by Prolog: as Prolog is a constraint programming language it has built-in support for numeric values and constraints. Therefore a future extension of our approach to support numeric variables should be possible. In addition in theory our approach as-is is capable of representing numeric values at a fixed range. Numeric values can be represented in their binary form using multiple boolean variables. The arithmetic in turn has to be expressed via boolean equations. However, in practice this approach can be expected to be too cumbersome and slow to be useable.

The second central assumption we made during our meta model specification is the fact that we assume that operations are essentially stateless. While we introduced state variables for modeling state within the execution of a single *SystemUsage* state variables are not persisted across the execution of *SystemUsage*. This means that in our model no long-term persistence outlasting a single usage execution exists. However, we expect that this assumption does not severely limit the applicability of our approach. Depending on the scenario, restrictions regarding data persistence can be modeled with other means. An example for such a ways of modeling persistence restrictions is given with **RQ-3.3**.

Another important assumption we made is that we can express every kind of operation using our proposed *LogicTerms*. Due to the fact that we assumed that all in- and output variables of operations are boolean we can show that this assumption does not induce any limitation regarding the expressiveness. Given any set of input variable configuration, it is possible to define the desired output variables based on truth tables. These truth tables can be expressed using boolean formulas, for example by using the disjunctive or the conjunctive normal form. As we have modeled *And*, *Or* and *Not* as *LogicTerms*, we can express these formulas in our model.

A limitation of a more practical nature is the fact that our model is not as expressive as Prolog which is used for the analysis. In certain cases it might not be possible to express additional required information using our meta model, where code in addition to the analysis has to be added manually. An example for such a case is our example shop scenario for geolocation based restrictions. There, the classification whether a geolocation is considered safe or unsafe was manually added using the analysis prolog code. We however assume that this limitation is not severe because (a) every model element can be easily referenced and therefore annotated based on its name and (b) additional code has to be added anyway for the formulation of most analysis.

9. Conclusion

To conclude this thesis, we briefly summarize our approach and the findings of our evaluation in Section 9.1. Finally, we outline possible enhancements we see as future work in Section 9.2.

9.1. Summary

The goal of this thesis was to provide an efficient model based approach for analyzing software systems regarding data flow constraints on architecture level. To achieve this we designed a specialized meta-model for modeling data flows within systems. In addition we defined a Prolog translator which allows the translation of model instances of our meta model to Prolog programs. The resulting Prolog programs expose the data flows and the structure of the modeled system through a specialized API. Based on this API, it is possible to formulate queries for uncovering data flow constraint violations. For the design of our approach we focused on two main application scenario: access control and geolocation based privacy restrictions.

The central idea we based our meta-model on is the modeling of data based on its meta-attributes. Representing such meta-attributes using sets of boolean variables allowed us to model data manipulation using boolean algebra. This approach enables us to define data manipulation through *Operations* in our model with a high degree of flexibility. In addition we allowed *Operations* to interact with each other, allowing a decomposition to closely match the underlying modeled system.

To allow an automated analysis for data flow constraint violations we provided a translator transforming instances of our meta-model to Prolog programs. These programs can be queried for violations using our proposed API. During the design of the translator we introduced new fundamental concepts, for example the representation of system call stack traces using Prolog lists. With the combination of these concepts and our API it is now possible to easily formulate various types of data flow constraint queries.

In addition a central aspect of this thesis was the scalability of our approach. For this reason we designed and evaluated three different optimizations of the generated Prolog code: The efficient usage of first-argument indexing, an implementation of efficient logical negations and a reduction of the number of required rules based on the Prolog cut predicate. The concepts these optimizations are based on are not limited to our work. They are potentially also applicable to other Prolog based approaches.

We evaluated our approach regarding its accuracy, scalability and genericness. For the accuracy we showed that our approach is able to express and analyse our example scenarios for access control and geolocation based restrictions. For the scalability we showed that the translation as well as the analysis scale well with the size of the input.

In addition we showed that our proposed performance optimizations provide significant gains in most of the investigated cases. In some cases we were able to reduce the runtime of the analysis from exponential to constant time. To show the genericness we discussed how additional types of scenarios can be expressed and analysed using our approach.

9.2. Future Work

While our approach is functional as-is, we still discovered several aspects which could be addressed as future work.

Add support for numeric variables Currently our meta-model models all variables as sets of boolean variables. While this variable type has shown to be sufficient for the scenarios we focused on, we assume that it is likely that other types of scenarios require numeric variables. The addition of a numeric type is especially feasible because Prolog has built-in support for numeric values. In order to support numeric types, another type has to be added beside the existing *ValueSetType*. In addition, custom term types have to be added for expressing arithmetic terms and constraints.

Realization of a translation from Data-Centric Palladio to our approach Our approach has been designed for expressing data flow systems so that they can be analysed automatically. For this reason, the manual definition of model instances can easily become cumbersome. Therefore, it is desirable to implement additional model transformations to bridge this gap. For example, Data-Centric Palladio offers an intuitive way for users to define the data flows of a system. However, currently its automated analysis capabilities are limited. Therefore it would be desirable to have a translation to the meta-model of our approach to enable an automated analysis in combination with an easy model definition.

Evaluation of our approach with additional scenarios During the design of our approach we focused on two main scenario types: access control and geolocation based restrictions. However as we tried to keep our approach as generic as possible during the design, we assume that it also can be applied to other scenario types. Supporting this assumption we provided a discussion based evaluation where we showed our approach potentially can be applied to other scenarios. However it would be desirable to have concrete system models and analysis for such additional scenario types to validate the genericness of our approach. In addition our choice of models did not directly cover all features of our approach, such as state variables. Therefore it is desirable to have an additional accuracy evaluation with models which cover these features.

Improve the scalability of the Prolog Translator implementation In the evaluation we found out that in certain scenarios our Prolog Translator scales quadratic (e.g. see Figure 8.5). However, according to our definition of the translation process in Chapter 6 it should be possible to implement the translator with a linear scalability. While the impact is minor due to the low total run time of the translator, this fact still shows that there is optimization potential for the implementation.

Bibliography

[1] 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.

[2] D Elliott Bell and Leonard J LaPadula. *Secure computer systems: Mathematical foundations*. Tech. rep. MITRE CORP BEDFORD MA, 1973.

[3] Max A. Bramer. *Logic Programming with Prolog*. London, 2013. URL: `http://dx.doi.org/10.1007/978-1-4471-5487-7`.

[4] Travis D Breaux, Hanan Hibshi, and Ashwini Rao. "Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements". In: *Requirements Engineering* 19.3 (2014), pp. 281–307.

[5] David FC Brewer and Michael J Nash. "The chinese wall security policy". In: *Security and privacy, 1989. proceedings., 1989 ieee symposium on*. IEEE. 1989, pp. 206–214.

[6] Deyan Chen and Hong Zhao. "Data security and privacy protection issues in cloud computing". In: *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*. Vol. 1. IEEE. 2012, pp. 647–651.

[7] Vítor Santos Costa, Konstantinos Sagonas, and Ricardo Lopes. "Demand-driven indexing of prolog clauses". In: *International Conference on Logic Programming*. Springer. 2007, pp. 395–409.

[8] Council of European Union. *Council regulation (EU) no 679/2016*. `http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679`. 2016.

[9] *ECLiPSe Homepage*. `http://eclipseclp.org/`. Accessed: 2018-09-17.

[10] *Efficiency of Prolog*. `https://www.metalevel.at/prolog/efficiency`. Accessed: 2018-10-11.

[11] William Enck et al. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), p. 5.

[12] Jürgen Graf, Martin Hecker, and Martin Mohr. "Using JOANA for Information Flow Control in Java Programs-A Practical Guide." In: *Software Engineering (Workshops)*. Vol. 215. 2013, pp. 123–138.

[13] Robert Heinrich. "Architectural run-time models for performance and privacy analysis in dynamic cloud applications". In: *ACM SIGMETRICS Performance Evaluation Review* 43.4 (2016), pp. 13–22.

[14] *JIProlog Homepage*. http://www.jiprolog.com/. Accessed: 2018-09-17.

[15] Jaeyeon Jung et al. "Privacy oracle: a system for finding application leaks with black box differential testing". In: *Proceedings of the 15th ACM conference on Computer and communications security*. ACM. 2008, pp. 279–288.

[16] Jan Jürjens. "UMLsec: Extending UML for secure systems development". In: *International Conference on The Unified Modeling Language*. Springer. 2002, pp. 412–425.

[17] 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.

[18] K. Katkalov et al. "Model-Driven Development of Information Flow-Secure Systems with IFlow". In: *2013 International Conference on Social Computing*. 2013, pp. 51–56. DOI: 10.1109/SocialCom.2013.14.

[19] Robert Kowalski. "Algorithm = Logic + Control". In: *Commun. ACM* 22.7 (July 1979), pp. 424–436. ISSN: 0001-0782. DOI: 10.1145/359131.359136. URL: http://doi.acm.org/10.1145/359131.359136.

[20] Object Management Group (OMG). *Unified Modeling Language Specification*. URL: http://www.omg.org/spec/UML/ (visited on 04/19/2018).

[21] Siani Pearson and Azzedine Benameur. "Privacy, security and trust issues arising from cloud computing". In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE. 2010, pp. 693–702.

[22] David Martin Powers. "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation". In: *Journal of Machine Learning Technologies* (2011).

[23] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, 2016. 408 pp. ISBN: 9780262034760. URL: http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

[24] Per Runeson et al. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[25] Yutaka Sasaki et al. "The truth of the F-measure". In: *Teach Tutor mater* 1.5 (2007), pp. 1–5.

[26] JOACHIM SCHIMPF and KISH SHEN. "ECLiPSe From LP to CLP". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), 127156. DOI: 10.1017/S1471068411000469.

[27] S. Seifermann. "Architectural Data Flow Analysis". In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 2016, pp. 270–271. DOI: 10.1109/WICSA.2016.49.

[28] Stephan Seifermann, Kateryna Yurchenko, and Max E Kramer. "Challenges to Trading-Off Performance and Privacy of Component-Based Systems". In: *Softwaretechnik-Trends* 36.4 (2016).

[29] *SWI Prolog Homepage*. `http://www.swi-prolog.org/`. Accessed: 2018-09-17.

[30] *SWI Prolog Just-in-time clause indexing*. `http://www.swi-prolog.org/pldoc/man?section=jitindex`. Accessed: 2018-07-27.

[31] Peter Van Roy, Bart Demoen, and Yves D Willems. “Improving the execution speed of compiled Prolog with modes, clause selection, and determinism”. In: *International Joint Conference on Theory and Practice of Software Development*. Springer. 1987, pp. 111–125.

[32] David HD Warren. “An abstract Prolog instruction set”. In: *Technical note 309* (1983).

[33] Philipp Weimann. “Automated Cloud-to-Cloud Migration of Distributed Soware Systems for Privacy Compliance”. MA thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology, 2017.

A. Appendix

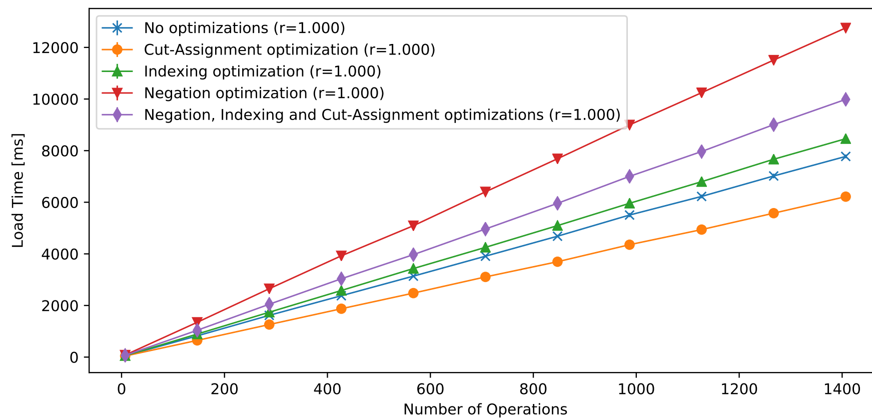

Figure A.1.: Load time of JIProlog measured for the scaling with the number of operations (**RQ-2.1**).

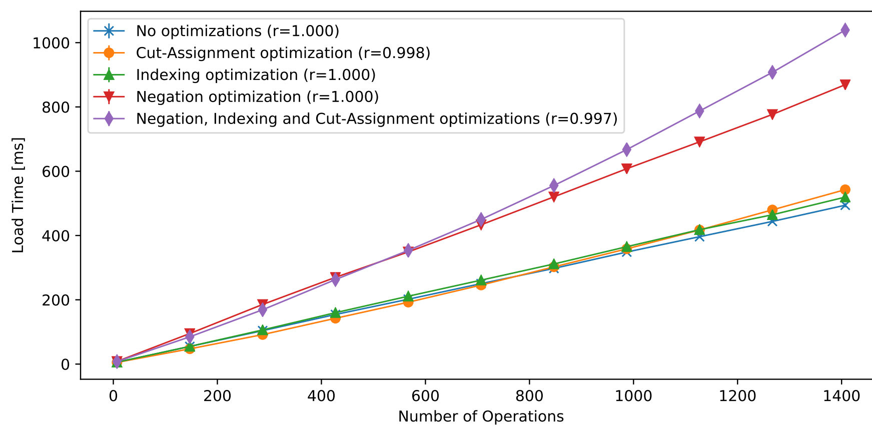

Figure A.2.: Load time of SWIProlog measured for the scaling with the number of operations (**RQ-2.1**).

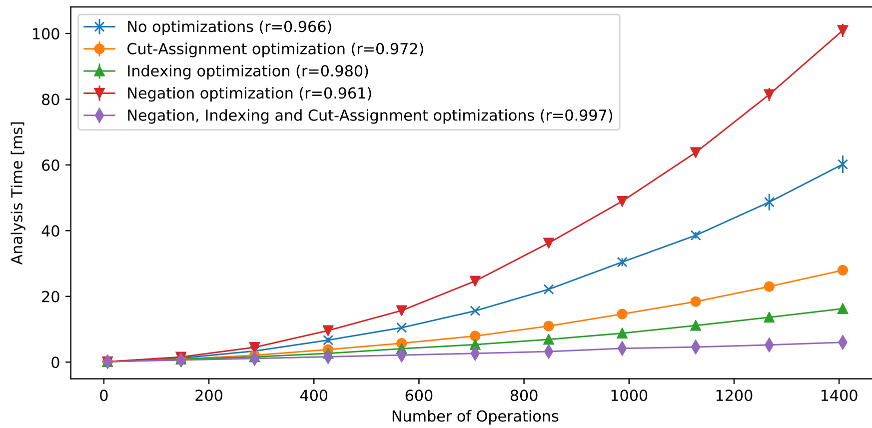

Figure A.3.: Analysis time of SWIProlog measured for the scaling with the number of operations (**RQ-2.1**).

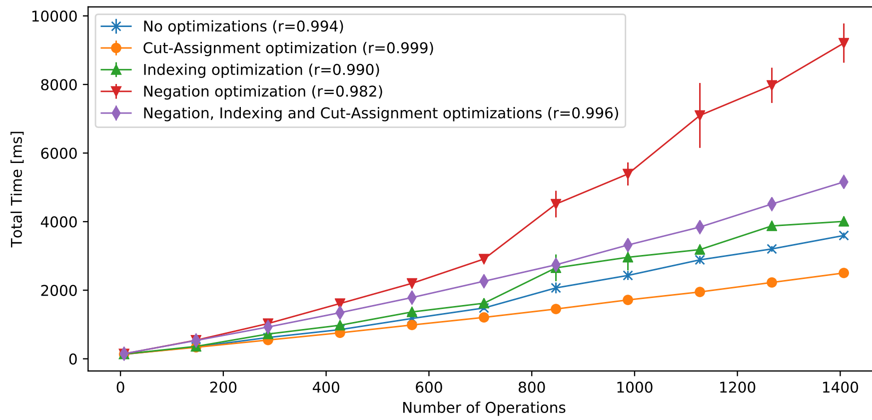

Figure A.4.: Total time of ECLiPSe measured for the scaling with the number of operations (**RQ-2.1**).

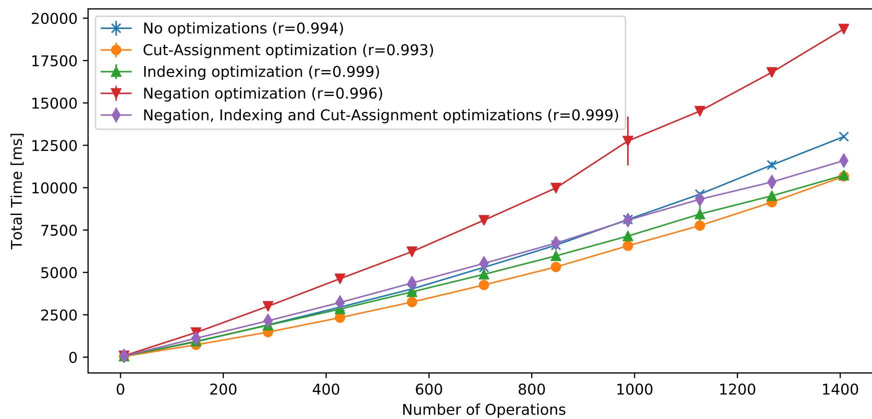

Figure A.5.: Total time of JIProlog measured for the scaling with the number of operations (**RQ-2.1**).

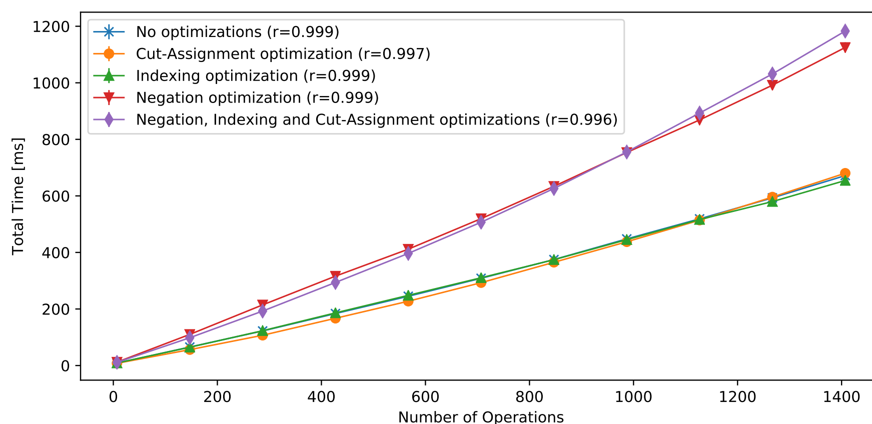

Figure A.6.: Total time of SWIProlog measured for the scaling with the number of operations (**RQ-2.1**).

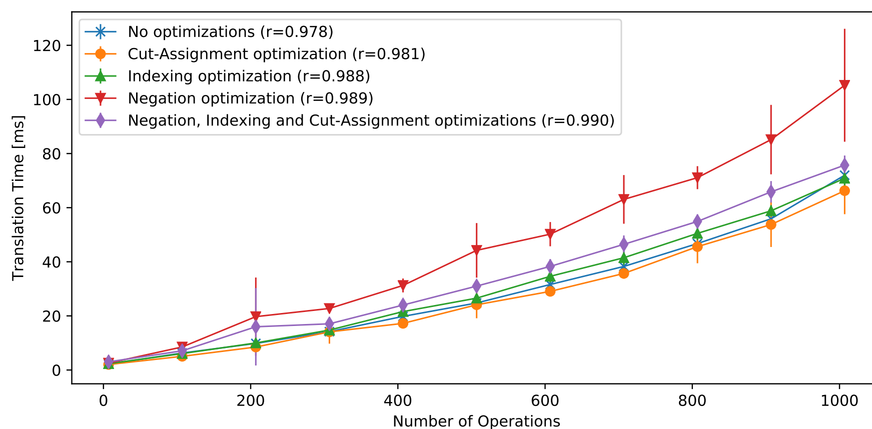

Figure A.7.: Translation time measured for the scaling with the complexity of the call graph (**RQ-2.2**).

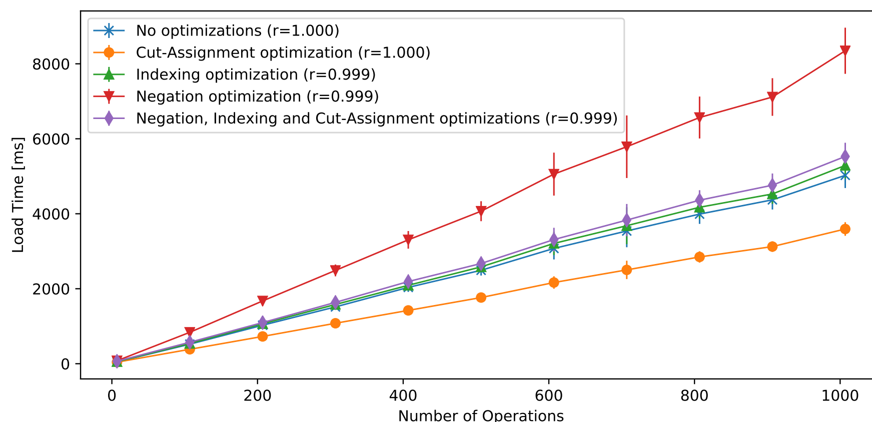

Figure A.8.: Load time of JIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**).

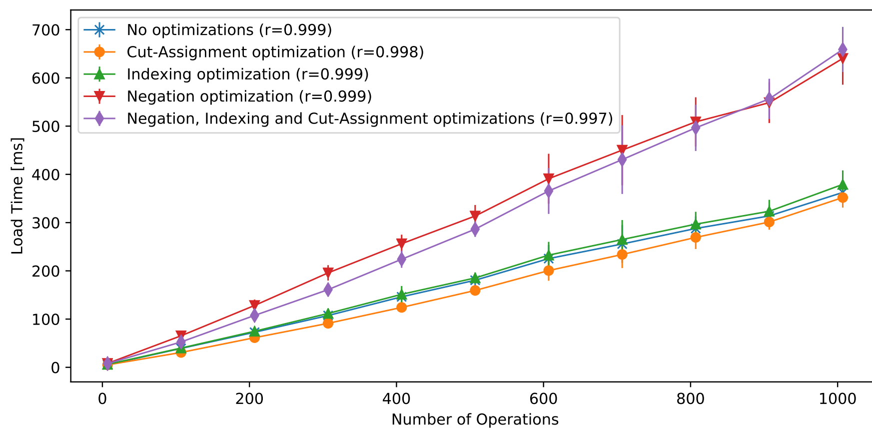

Figure A.9.: Load time of SWIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**).

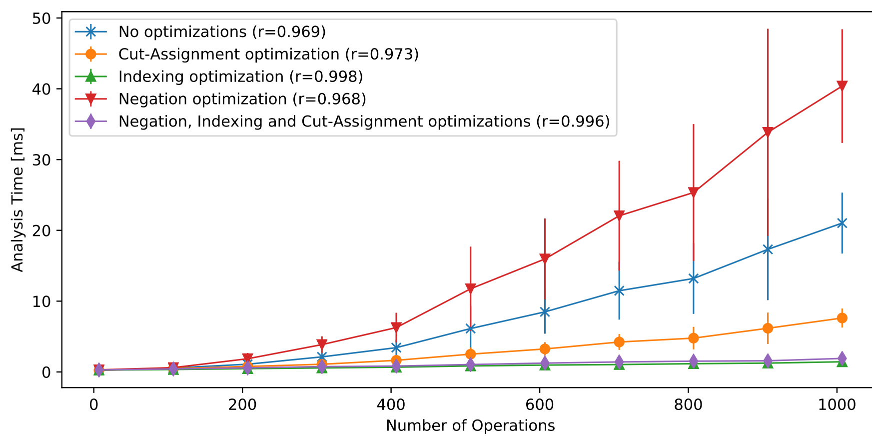

Figure A.10.: Analysis time of ECLiPSe measured for the scaling with the complexity of the call graph (**RQ-2.2**).

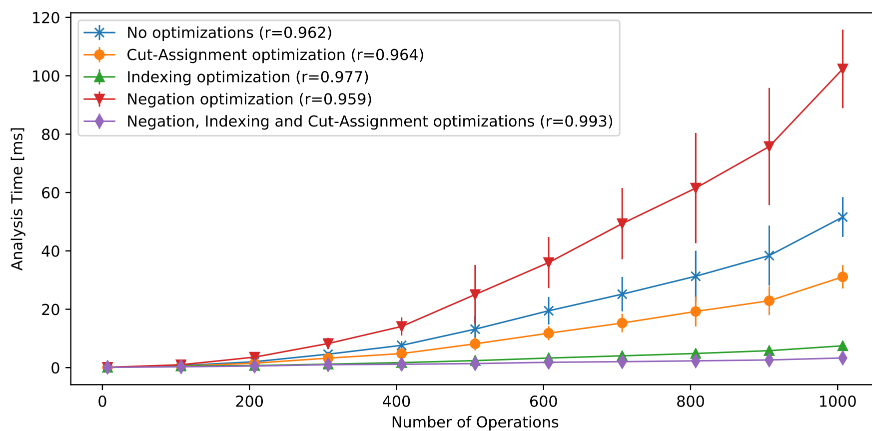

Figure A.11.: Analysis time of SWIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**).

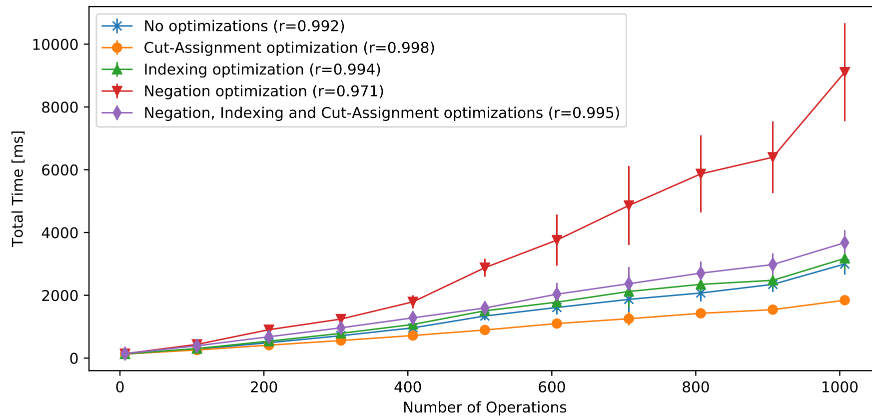

Figure A.12.: Total time of ECLiPSe measured for the scaling with the complexity of the call graph (**RQ-2.2**).

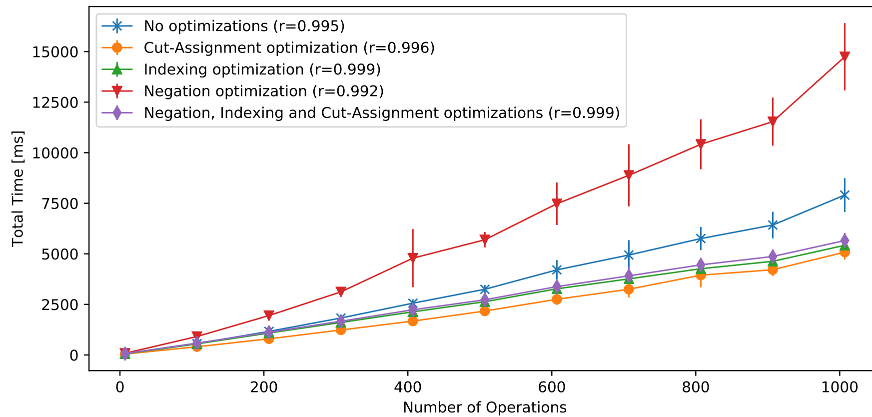

Figure A.13.: Total time of JIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**).

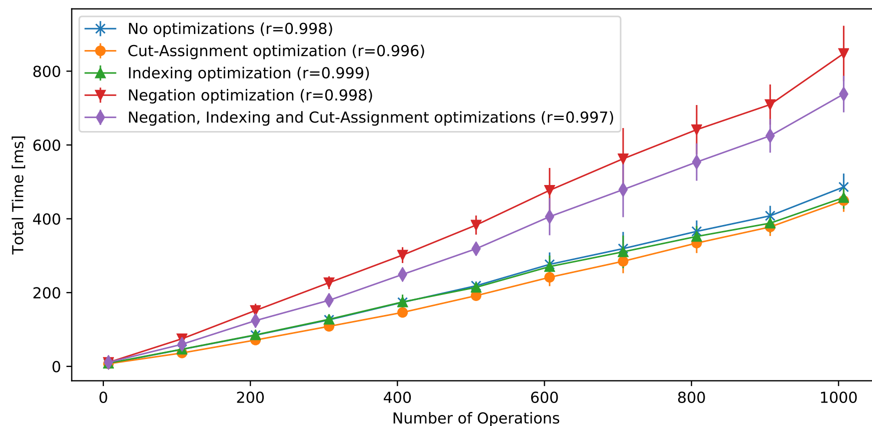

Figure A.14.: Total time of SWIProlog measured for the scaling with the complexity of the call graph (**RQ-2.2**).

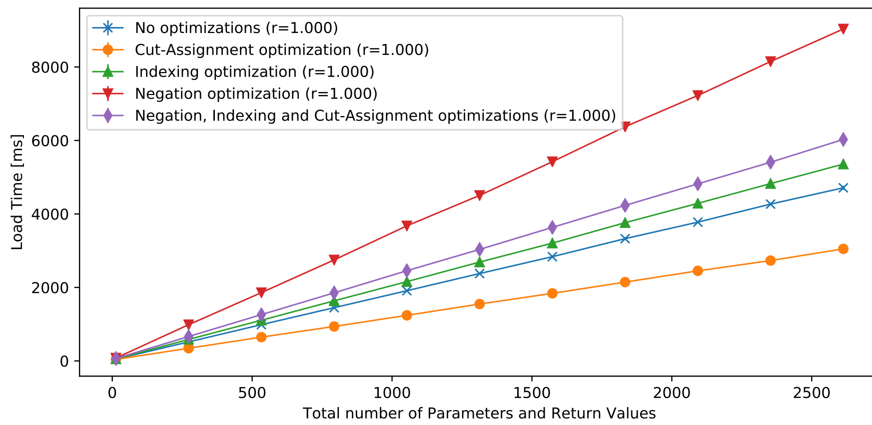

Figure A.15.: Load time of JIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**).

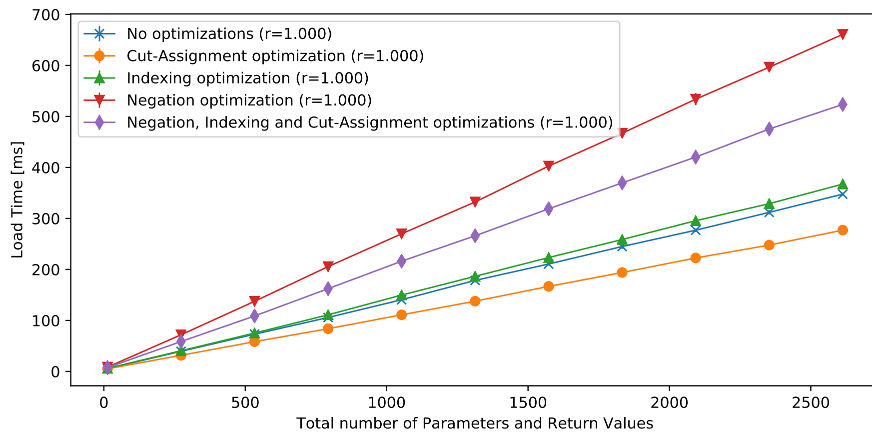

Figure A.16.: Load time of SWIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**).

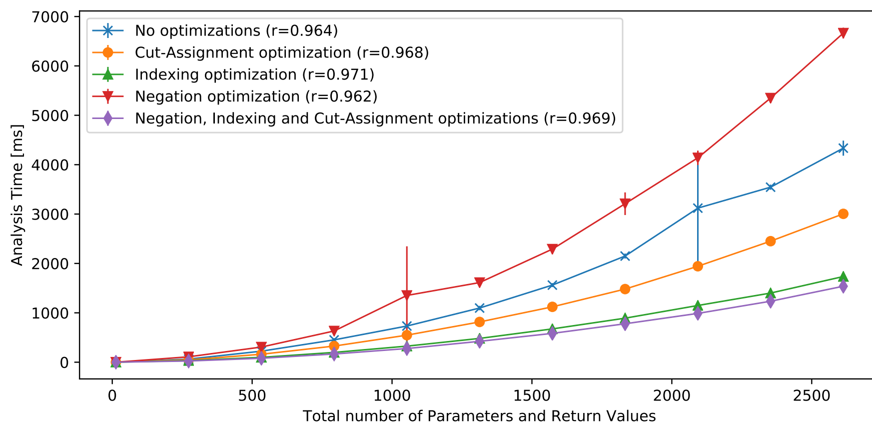

Figure A.17.: Analysis time of JIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**).

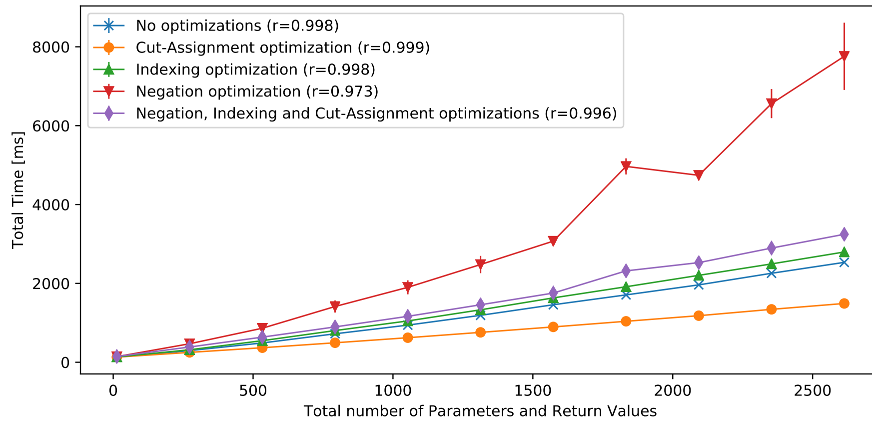

Figure A.18.: Total time of ECLiPSe measured for the scaling with the number of parameters and return values (**RQ-2.3**).

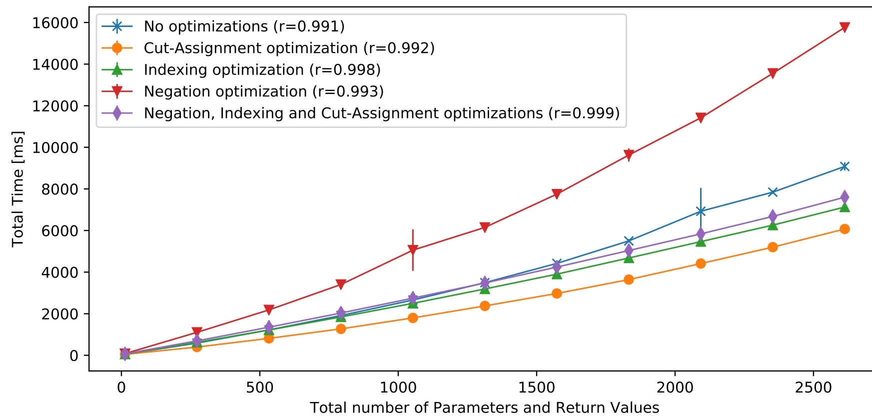

Figure A.19.: Total time of JIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**).

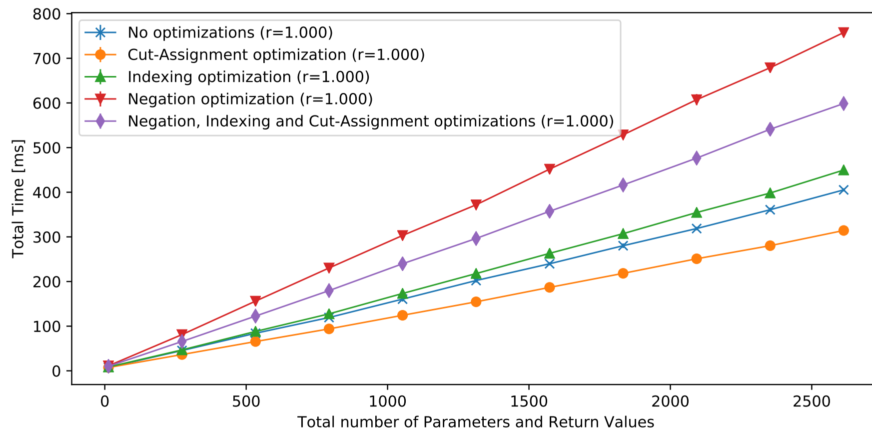

Figure A.20.: Total time of SWIProlog measured for the scaling with the number of parameters and return values (**RQ-2.3**).

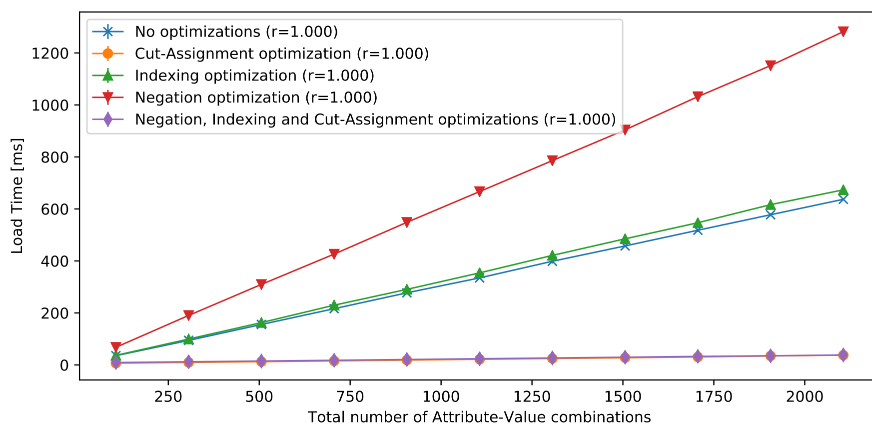

Figure A.21.: Load time of SWIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

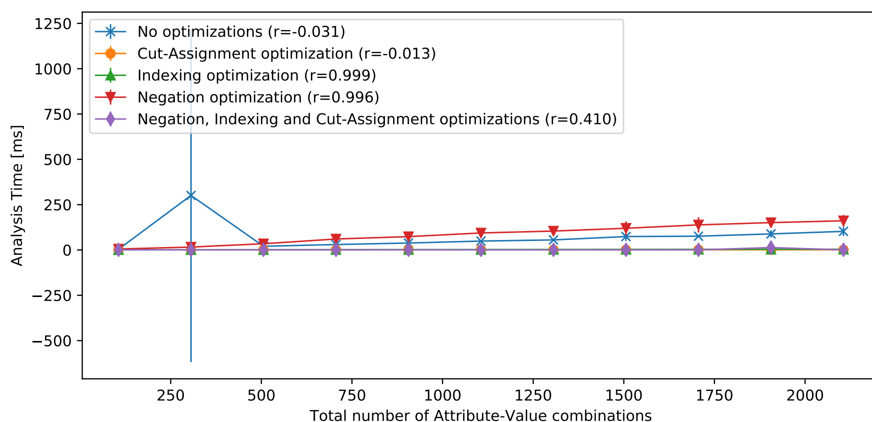

Figure A.22.: Analysis time of JIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

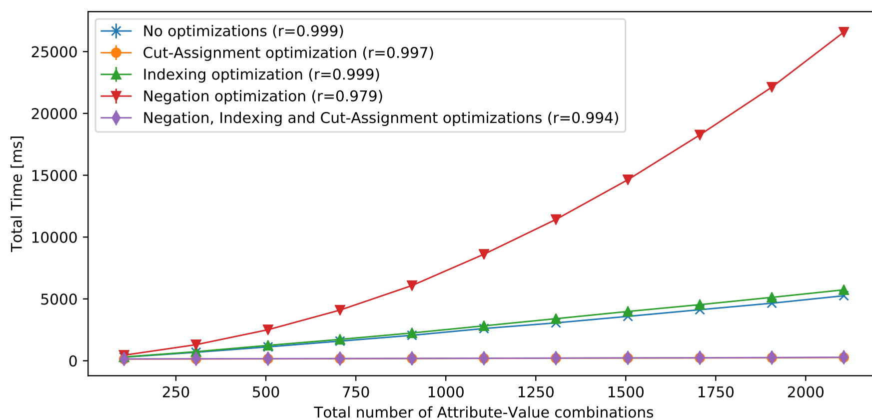

Figure A.23.: Total time of ECLiPSe measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

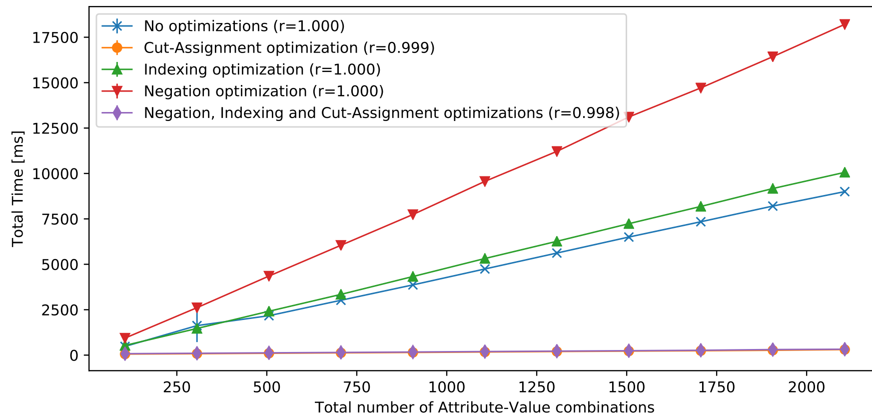

Figure A.24.: Total time of JIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

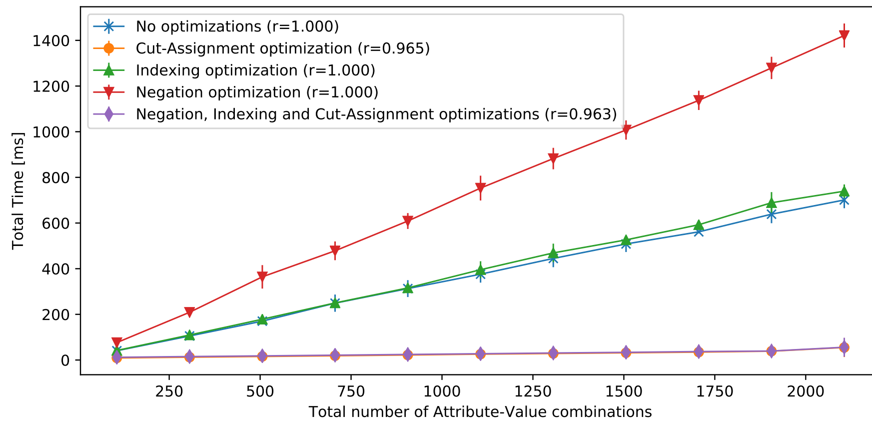

Figure A.25.: Total time of SWIProlog measured for the scaling with the number of attribute-value combinations (**RQ-2.4**).

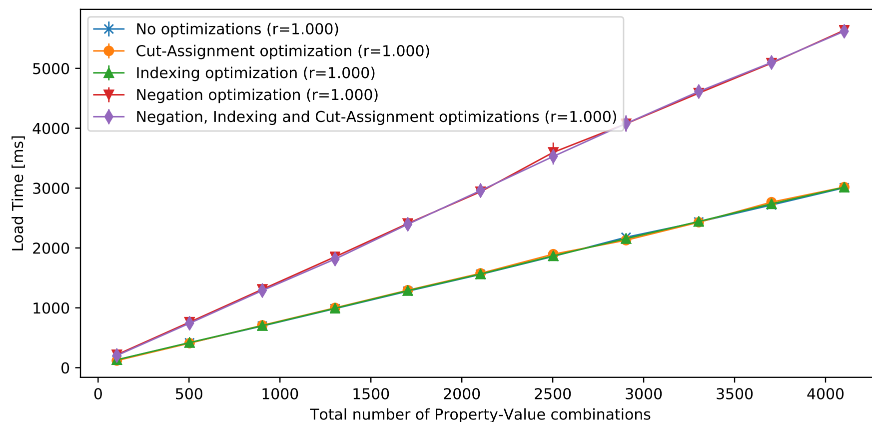

Figure A.26.: Load time of JIProlog measured for the scaling with the number of properties (**RQ-2.5**).

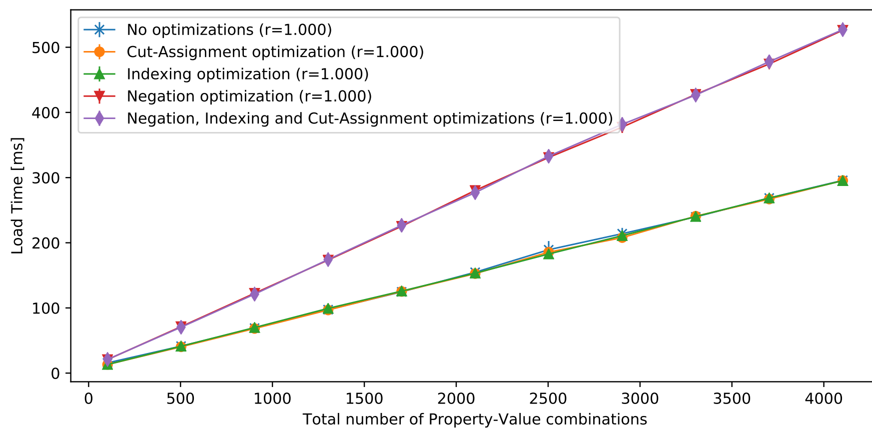

Figure A.27.: Load time of SWIProlog measured for the scaling with the number of properties (**RQ-2.5**).

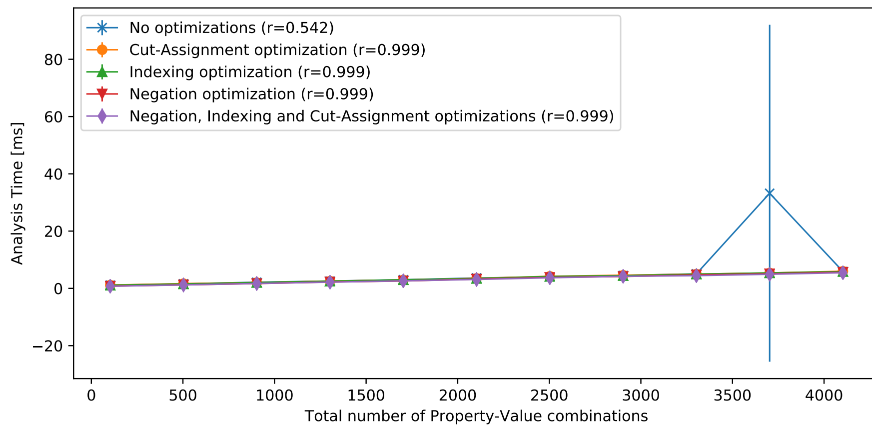

Figure A.28.: Analysis time of JIProlog measured for the scaling with the number of properties (**RQ-2.5**).

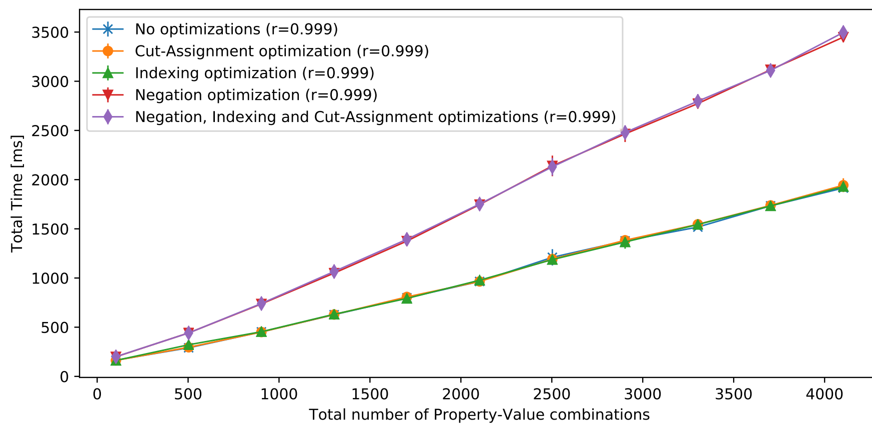

Figure A.29.: Total time of ECLiPSe measured for the scaling with the number of properties (**RQ-2.5**).

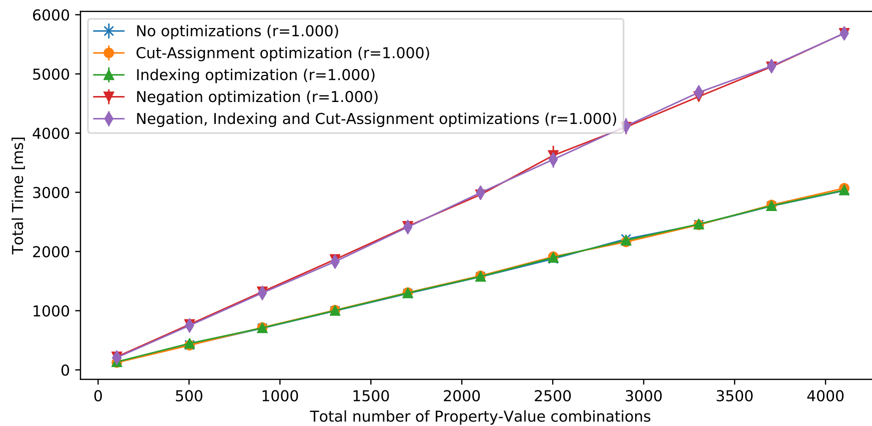

Figure A.30.: Total time of JIProlog measured for the scaling with the number of properties (**RQ-2.5**).

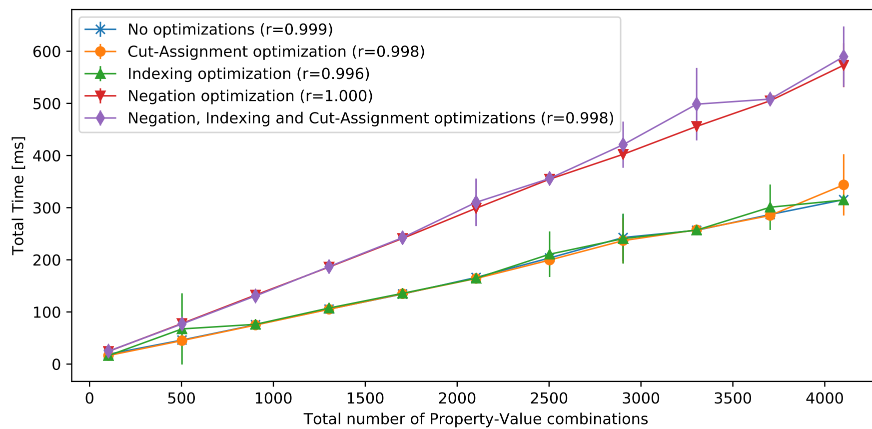

Figure A.31.: Total time of SWIProlog measured for the scaling with the number of properties (**RQ-2.5**).

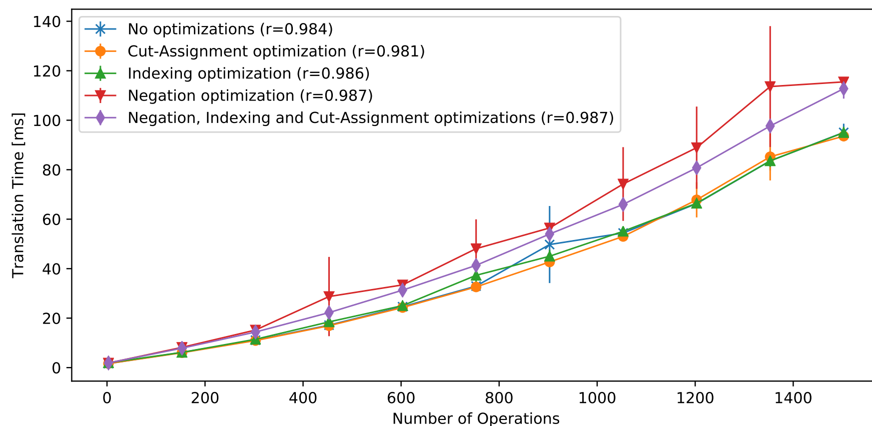

Figure A.32.: Translation time measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

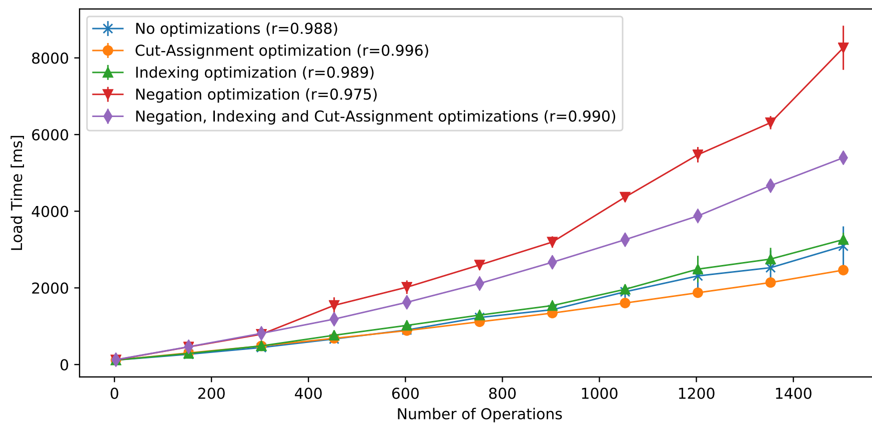

Figure A.33.: Load time of ECLiPSe measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

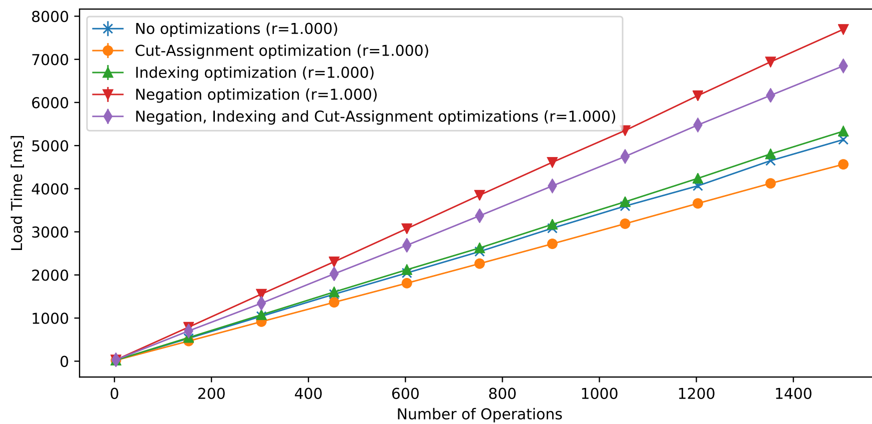

Figure A.34.: Load time of JIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

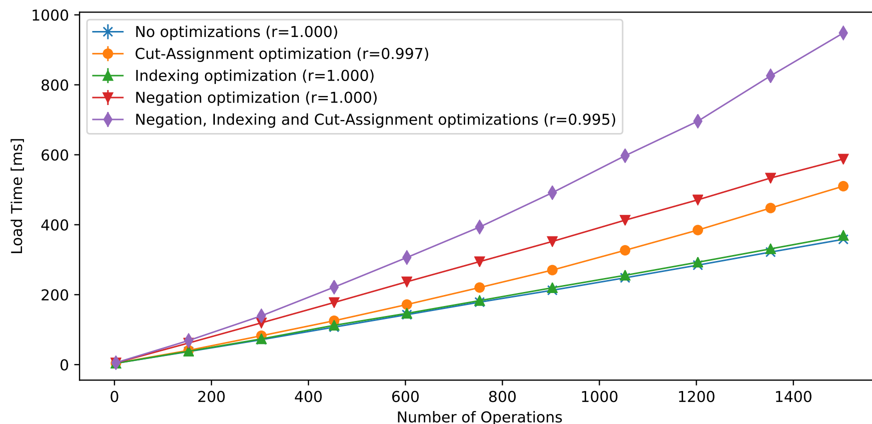

Figure A.35.: Load time of SWIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

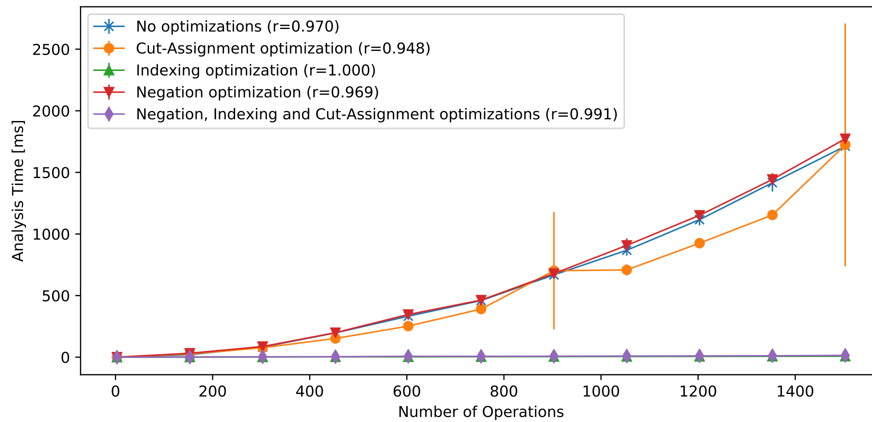

Figure A.36.: Analysis time of JIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

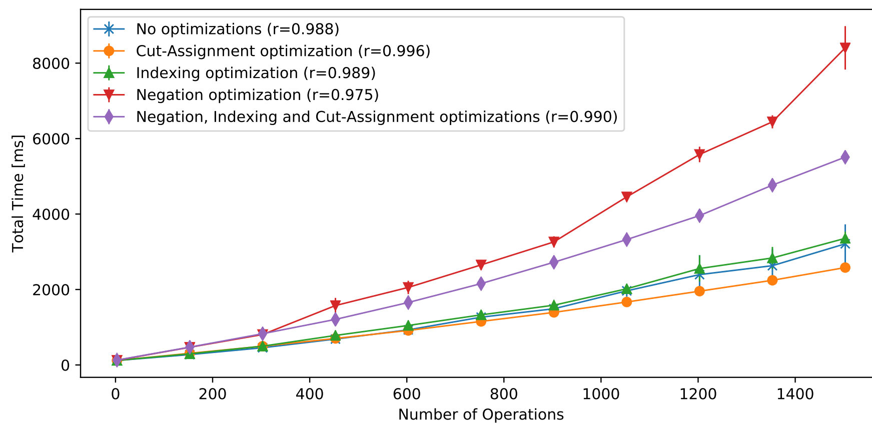

Figure A.37.: Total time of ECLiPSe measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

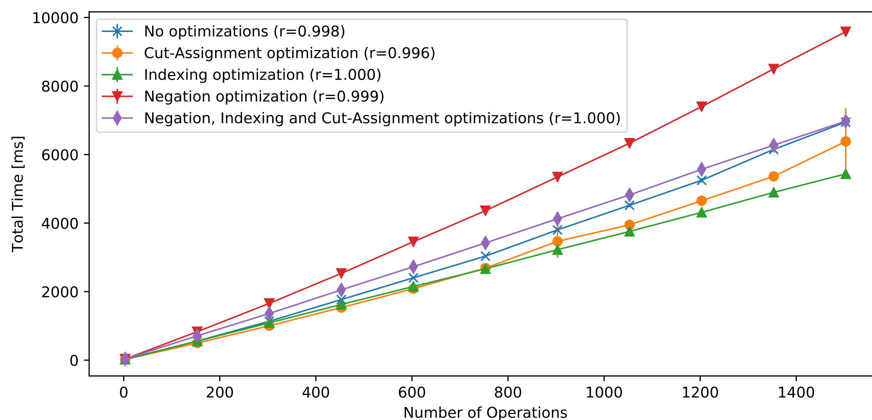

Figure A.38.: Total time of JIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

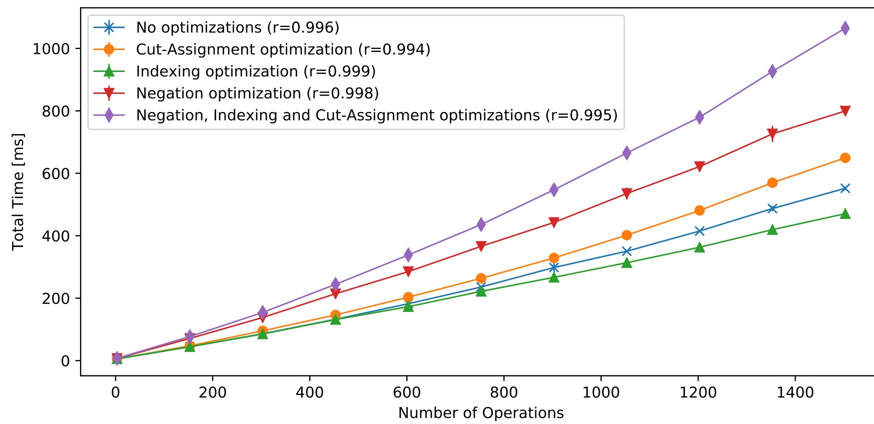

Figure A.39.: Total time of SWIProlog measured for the first argument indexing optimization performance experiment (**RQ-2.6**).

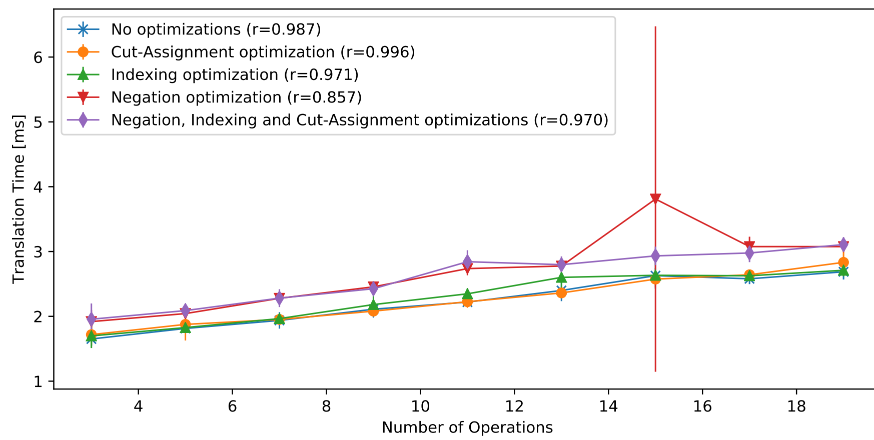

Figure A.40.: Translation time measured for the negation optimization experiment (**RQ-2.7**).

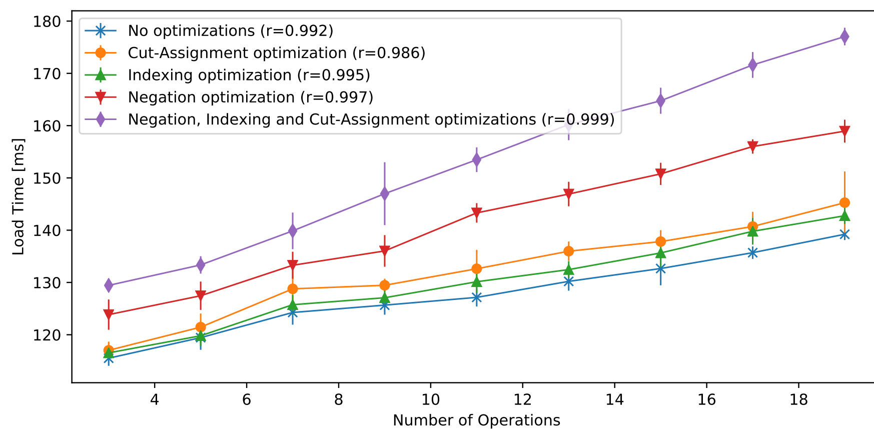

Figure A.41.: Load time of ECLiPSe measured for the negation optimization experiment (**RQ-2.7**).

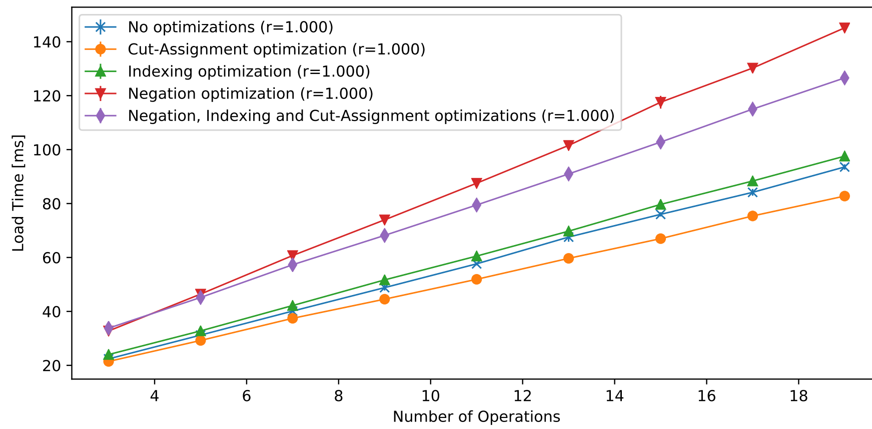

Figure A.42.: Load time of JIProlog measured for the negation optimization experiment (**RQ-2.7**).

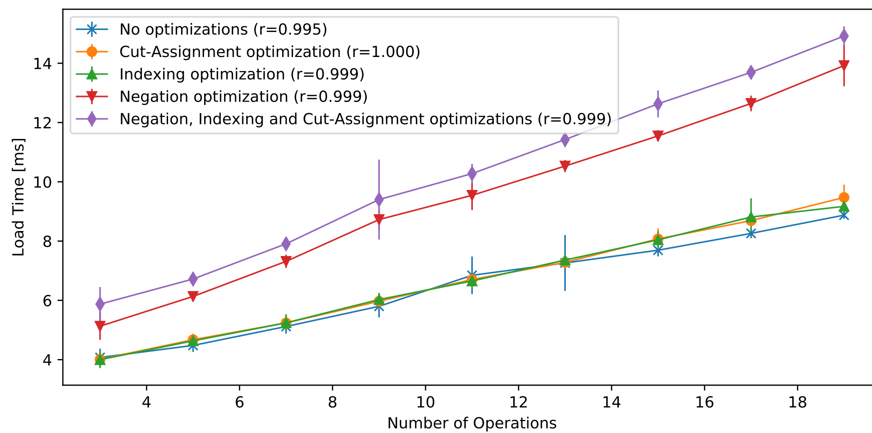

Figure A.43.: Load time of SWIProlog measured for the negation optimization experiment (**RQ-2.7**).

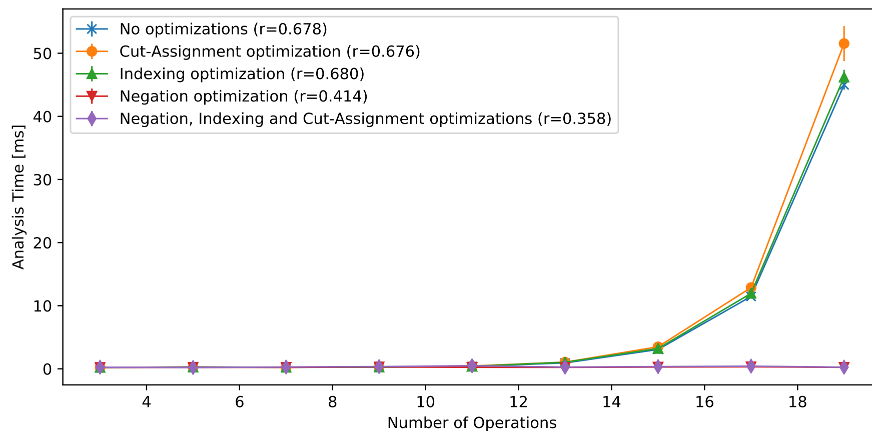

Figure A.44.: Analysis time of ECLiPSe measured for the negation optimization experiment (**RQ-2.7**).

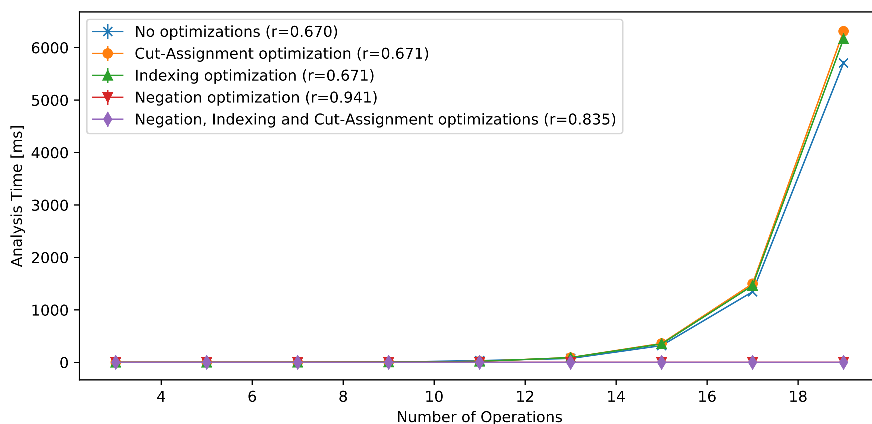

Figure A.45.: Analysis time of JIProlog measured for the negation optimization experiment (**RQ-2.7**).

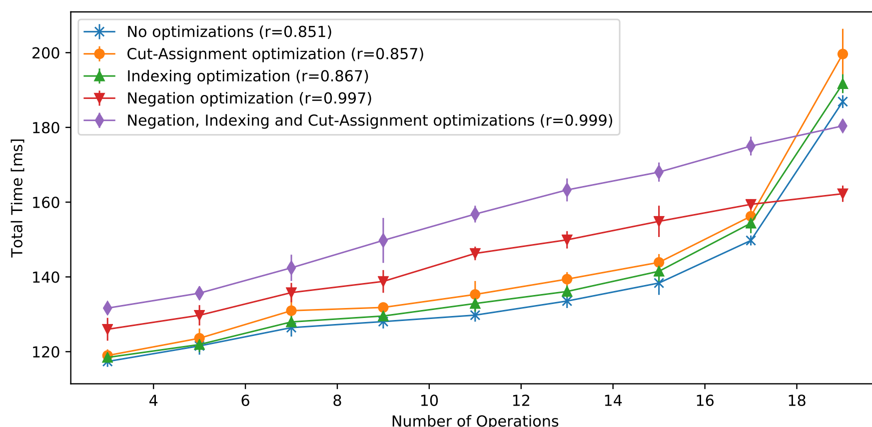

Figure A.46.: Total time of ECLiPSe measured for the negation optimization experiment (**RQ-2.7**).

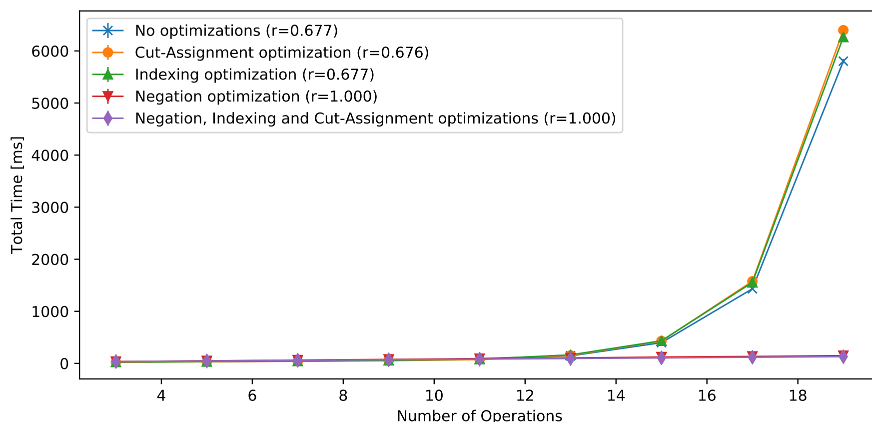

Figure A.47.: Total time of JIProlog measured for the negation optimization experiment (**RQ-2.7**).

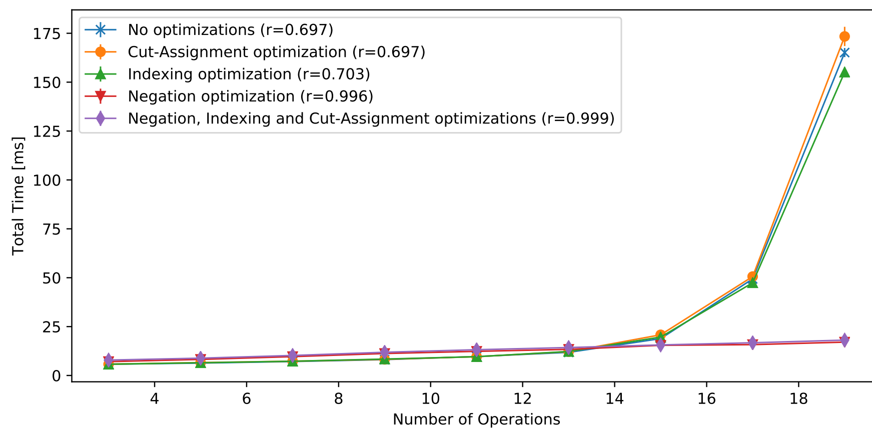

Figure A.48.: Total time of SWIProlog measured for the negation optimization experiment (**RQ-2.7**).