Modeling Data Flow Constraints for Design-Time Confidentiality Analyses

Sebastian Hahner, Stephan Seifermann, Robert Heinrich, Maximilian Walter
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{sebastian.hahner, stephan.seifermann, robert.heinrich, maximilian.walter}@kit.edu

Tomáš Bureš, Petr Hnětynka
Charles University
Prague, Czech Republic
{bures, hnetynka}@d3s.mff.cuni.cz

***Abstract*—With the increase in connectedness and the growing volume of data, ensuring confidentiality becomes increasingly critical. Data-driven analyses try to cope with this complexity by automatically verifying confidentiality at design time. However, confidentiality constraints are manifold. Thus, analyses limit the software architect's possibilities of expression or require them to use the underlying verification formalism directly. We propose a domain-specific language to enable architects to formulate data flow constraints using the terminology and abstraction of the architectural domain. We present a mapping of data flow constraints and results which is compliant to the transformation of the architecture and evaluated based on real-world scenarios.**

I. Introduction

In today's connected world, exchanging data is essential to many business applications. With growing system complexity and size, ensuring confidentiality becomes increasingly challenging [1]. Confidentiality demands that "information is not made available or disclosed to unauthorized individuals, entities, or processes" [2]. Violations cannot only harm user's privacy but also have legal consequences [3].

To cope with confidentiality early in design time, data-driven analyses have been proposed [4], [5]. These approaches are based on Data Flow Diagrams (DFDs) which are commonly used in threat modeling because security "problems tend to follow the data flow, not the control flow" [6]. By modeling data sources, sinks and data processing and verifying constraints at design time, e.g., by transforming the model to logic programs [4], confidentiality issues can be detected and repaired earlier which can reduce cost significantly [7].

Defining confidentiality-oriented data flow constraints is challenging: Too strict constraints may reduce functionality and flexibility and forcing strict non-interference is considered to be insufficient [8]. Thus, constraint definitions need to target the flow of selected classes of confidential data directly which is a more fine-grained approach than using standardized metrics whose expressiveness is limited. Data-Driven Software Architectures (DDSAs) [4] use characteristics to annotate data and data processors on architectural level while constraints are defined by utilizing the underlying formalism. This requires the architect to switch between the abstraction layers of the architecture and analysis. This gap can also be observed in other related work [1], [5], [9], [10] and increases with the analysis variability. It cannot be assumed that software architects know the architectural transformation and the underlying analysis formalism well enough to be able to define fine-grained data flow constraints without training.

To avoid the gap, defining constraints must be possible in the architectural domain. Here, domain-specific languages (DSLs) offer appropriate notations by abstracting from the verification [11]. They require less knowledge and promise faster and less error-prone definitions [12]. Regarding the integration of a DSL into existing analysis approaches, additional challenges occur: The DSL mapping has to be compliant to the existing transformation of the annotated software architecture and to the analysis formalism. References to architectural elements must remain valid throughout the transformation. Moreover, the mapping of analysis results is necessary in order to present results in a meaningful way to the architect.

In this paper, we propose an approach to define data flow constraints for data-driven analyses using the terminology and abstraction level of the architectural domain. We start by introducing a running example for data flow analysis in Section II and discussing the state of the art in Section III. We present the contributions of this paper hereafter.

- **C1** First, we introduce a DSL to express constraints without knowledge of the underlying formalism or the transformation and analysis process. We list requirements and present language constructs for the constraint definition in Section IV. This shall enhance the architects' formulation capabilities compared to annotating UML models [1] or using predefined security checks [5].
- **C2** Second, we present a mapping to the underlying formalism which is compliant to the mapping of the modeled software architecture. We also define a mapping of analysis results back into the architectural domain which lacks in related work [1], [4], [13]. The contribution of this paper in Section V includes the presentation of relations between modeling and analysis which represents a difficult problem due to additional dependencies.

The evaluation of both the DSL and the mapping in Section VI is based on case studies from related work, e.g., the iFlow approach [1]. The results indicate that our approach is expressive enough to model most considered scenarios and the abstraction from the analysis formalism reduces accidental complexity. Section VII concludes this paper.

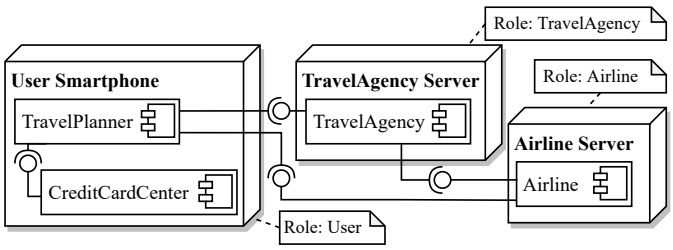

Figure 1. Travel Planner deployment diagram with role annotations

II. Running Example

In this section, we introduce the running example which will be used in the remainder of this paper. The *Travel Planner* scenario is based on a case study [14] by the iFlow approach [1] and models a flight booking system. Figure 1 shows the simplified deployment of the four components: The *TravelPlanner* and *CreditCardCenter* apps, the *TravelAgency* and the *Airline*.

We consider two scenarios: Requesting offers and booking flights. Whenever users request flight offers by using the *TravelPlanner* app, the *TravelAgency* forwards the request to the *Airline* and returns the answer to the users. Users choose a flight and book seats by directly contacting the *Airline* server using the *TravelPlanner* app. In order to complete the booking process, users transmit their credit card details. Since this is considered to be confidential information, users must explicitly allow the transmission using the *CreditCardCenter* app.

In order to ensure the confidentiality, we apply Role Based Access Control (RBAC) [15] which demands that data can only flow to entities whose role is authorized for the access. This can be defined as data flow constraint. We annotate the roles in Figure 1 similarly as in [4]: *User*, *TravelAgency*, *Airline*. Credit card details are only released for the User role. In order to transmit this information to the *Airline* server, the *Airline* role has to be authorized first. A constraint violation occurs if the information is passed without an user permit or if it is passed to the *TravelAgency* which is never authorized.

III. State of the Art

The major objective of our approach is providing software architects with means for defining confidentiality analyses using architectural terminology and executing them in an existing analysis formalism. The work related to our approach falls into the following two categories.

Direct Analyses analyze architecture models directly without an intermediate transformation into an analysis formalism. Such analyses are often defined in terms of model queries [16], [17] but can also be defined by formulas [9], [18] if the underlying model is already given as formal system. All of these approaches do not provide a solution for bridging the gap between the architecture and dedicated analysis models. However, this is what we are interested in.

Indirect Analyses analyze architecture models by first transforming it into another representation to make use of an analysis formalism. Most of these analyses are predefined. There are approaches only supporting one particular type of analysis such as iFlow [1] or the approach of Gerking et al. [19] with respect to information flow. However, predefined analyses do not necessarily mean low expressiveness. UMLSec [13] covers different analyses for confidentiality, integrity, authenticity, freshness or even information flow. All of these approaches do not claim to provide extensible analysis capabilities, so they do not provide a generic solution for bridging the gap between architecture and analysis models. Here, we identify the biggest advantage of our approach. The question, which category is superior is not considered in this paper.

Papers usually do not report on the transformation of the analysis results back into the architecture, so it remains unclear if they provide such a transformation. However, this is also an important issue for architects that we address in our approach.

IV. Modeling Data Flow Constraints

In this section, we introduce a DSL to define data flow constraints in the architectural domain (**C1**). We present language requirements, give an overview over the syntax, and present exemplary usage scenarios.

Based on Feature-Oriented Domain Analysis (FODA) [20], we consider user-centered and technology-centered influences on our DSL. Here, end-user feature requirements form a lower limit of the desired expressiveness which the DSL has to achieve in order to be applicable. The technology, namely data-driven analyses, represents an upper limit of constraint description possibilities. Put simply, constraints which cannot be analyzed shall not be expressible using the DSL.

We extracted data flows and constraints from scenarios described in related work presented in Section III. We identified several similarities regarding end-user requirements (**R1**—**R4**). Data flow constraints shall use the established terminology of DFDs [21] (**R1**). Constraints shall not limit all data flows but select flows by using attributes from the architectural model (**R2**). To aid the usability and understandability, data flow selections shall be reusable, e.g., by combining attributes or defining classes of selections (**R3**). Last, architects shall be able to define variable conditions, e.g., to model access control [22] (**R4**). These requirements also appear in our running example in the context of RBAC. The upper limit of expressiveness of the DSL is restricted by the capabilities of the data flow analysis. For example, DDSAs provide an interface to define constraints against the modeled software architecture [4]. These constraints are based on the annotation of data and data processors using characteristics.

Based on these requirements, we define a DSL to formulate data flow constraints. Figure 2 shows the simplified abstract syntax with exemplary values of our running example. *Constraints* represent the analysis goal, e.g., to enforce RBAC. They contain a *Rule* which defines which data flows shall be restricted. *DataSelectors* and *DestinationSelectors* are used to formulate the selection of data flows, e.g., based on RBAC roles and access rights. Since both represent variable values, the *Condition* describes their relation. If this *Condition* evaluates to true, the *Statement* specifies that the flow is illegal.

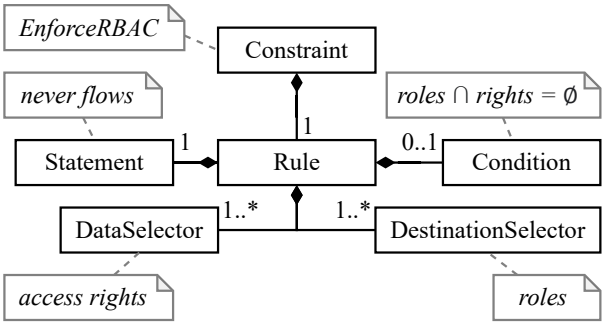

Figure 2. Simplified abstract syntax annotated with exemplary values

Listing 1 shows our exemplary concrete syntax. Both `accessRights` and `userRoles` are specified in the architectural model and referenced as set variables. The *Condition* in line 6 describes the relation between both that would lead to a violation of RBAC. The keywords `NEVER FLOWS` and `WHERE` indicate the restriction of certain data flows. This syntax enables the formulation of constraints close to the natural language. This shall enable architects to define constraints with no or only little training in data flow modeling.

Listing 1
CONCRETE SYNTAX OF AN CONSTRAINT TO THE RUNNING EXAMPLE

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

In the following, we describe the individual language parts in more detail. We touch the intuitive semantics and relate to the requirements listed above.

a) Selection: To focus the scope of single data flow constraints, we added selection capabilities to the DSL based on architectural elements and annotations to the architectural model, e.g., characteristics (**R2**). The selection is mandatory, because if all data flows are disallowed, any functional call would be interpreted as constraint violation. We distinguish between *DataSelectors* which select data by their characteristics and *DestinationSelectors* which select sinks by referencing architectural elements or characteristics (**R1**). Architects can combine, invert, and predefine reusable classes of selections (**R3**). To realize more complex relations between data and flow destination, variables can be used. Instead of static values, these are assigned during the analysis. They contain single values or are declared as set variables. The running example uses set variables to reference roles and access rights in RBAC.

b) Condition: To evaluate variable values, we added *Conditions*. These are mandatory if one or multiple *Selectors* declare variables. Otherwise, defining constraints without precise data flow selection would be possible. *Conditions* represent boolean expressions which state under which conditions data flow are not allowed (**R4**). We provide operators which evaluate variables, e.g., by comparing values, by checking if a value is contained in a set or by creating an intersection or union of set variables as shown in the running example. Additionally, expressions can be nested and combined based on propositional logic.

c) Statement: The *Statement* defines the restriction of the selected data flows. It consists of a modality and a type. Currently, the only supported *Statement* is *never flows* which restricts selected data to never flow to the selected destination (**R1**). If at least one such flow is found, the statement cannot be fulfilled and thus a constraint violation occurs. We defined the *Statement* to be expendable in order to support more modalities and types in the future, e.g., to test if at least one specified data flow exists in the modeled system.

d) Rule: Rules act as container for *Selectors*, *Statements* and *Conditions*. At least one *DataSelector* and one *DestinationSelector* are mandatory but multiple can be combined to more detailed selections. Every *Rule* has one *Statement* and requires a *Condition* if at least one *Selector* declares variable selections. The *Rule* itself is contained in a named *Constraint*. We choose this distinction between *Rule* and *Constraint* to enhance the DSL's extensibility, e.g., by combining the restriction from multiple rules inside one constraint using propositional logic.

In the reminder of this chapter, we show two additional examples to demonstrate the applicability of the DSL in common scenarios. More examples can be found in [23].

Listing 2
CONCRETE SYNTAX OF COMMON USAGE SCENARIOS' CONSTRAINTS

```
1  constraint NonInterference {
2     data.attribute.level.private
3   NEVER FLOWS
4     node.property.level.public
5  }
6
7  constraint ConfidentialDataEncryption {
8     data.class.ConfidentialData &
9     data.attribute.encryption.noEncryption
10  NEVER FLOWS
11    node.property.location.!internal
12 }
13
14 class ConfidentialData {
15    type.[personal,secret]
16 }
```

The first constraint in Listing 2 considers non-interference, i.e., "all information-flows from secret data to public observers" [8]. We denote this using a single *DataSelector* in line 2 and a single *DestinationSelector* in line 4. The second constraint in line 7 is more advanced and forbids confidential data to flow to non-internal system nodes if the data is not encrypted. In line 11, we invert the location `internal` to specify that any other locations are restricted without having to enumerate them which would harm readability and also harm maintainability on system changes. In line 14, we use a class to precisely define that both data characterized as `personal` and `secret` are considered to be confidential. This does not only enhance the understandability but also eases the reuse of the definition of confidential data in additional constraints.

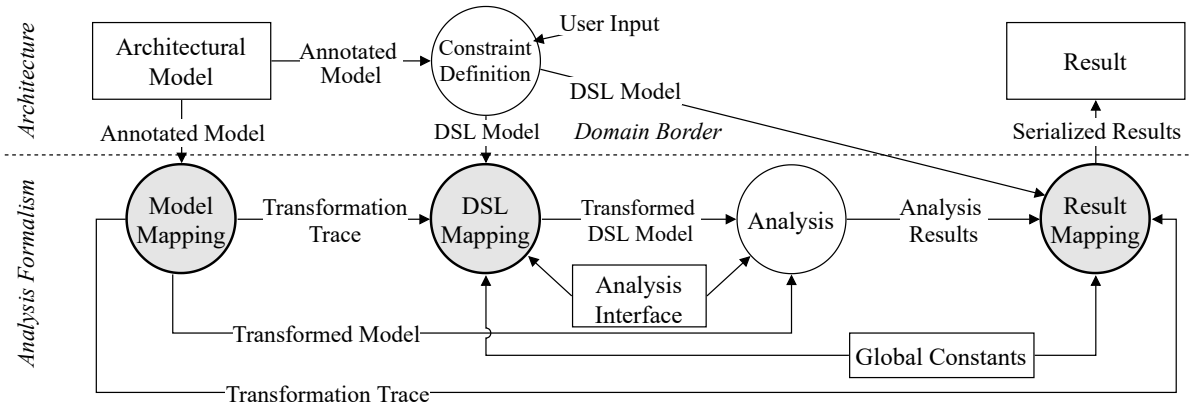

Figure 3. Data flow diagram of the mapping and analysis process

V. Mapping of Architecture and Constraints

In this section, we discuss the mapping of constraints formulated in our DSL to the underlying analysis formalism and the mapping of analysis results back into the architectural domain (**C2**). We show the dependencies between these mappings, the architecture model, and the analysis formalism. Last, we give examples based on our DSL and the DDSA approach [4]. A prototypical implementation can be found in [24].

Figure 3 gives an overview of the data sources and processes in the transformation and analysis using the syntax of DFDs [21]. It consists of the three mappings of model, DSL, and results (highlighted gray), the definition and analysis processes, sources like the *Architectural Model*, and sinks like the *Result*.

Architects perform the *Constraint Definition* based on the annotated *Architectural Model*. Afterwards, both are mapped to the analysis formalism. To maintain valid references from the DSL to the *Transformed Model* of the architecture, the *DSL Mapping* receives the *Transformation Trace* and is able to resolve the mapped relations. The *Transformed Model* and the *Transformed DSL Model* are input to the *Analysis*. The *Analysis Interface* describes how analysis goals are formulated. For example, the *Analysis* of DDSA uses the logic programming language Prolog and offers an API which has to be known to the *DSL Mapping* in order to generate compliant goals.

The output of the *Analysis* represents constraint violations which are mapped back to the architectural domain by the *Result Mapping* to ease the interpretation. Analogous to the *DSL Mapping*, the *Transformation Trace* is used to resolve the originating elements. Another input to the *Result Mapping* are *Global Constants* (i.e., conventions like prefixes) which are also visible to the *DSL Mapping*. They are used to identify the meaning of parts of the *Analysis Results* which is not represented by the formalism, e.g., the meaning of certain variables. This gap exists because formalisms are usually not built for one special analysis but exist for wider application. E.g., Prolog is used for more than data flow analyses. The *DSL Model* is required as input to the *Result Mapping* to match *Analysis Results* with constraint details, e.g., assigning values to specified variables. Last, the *Result* is shown to the architect.

We aim towards an automated approach without user interference in transformation or analysis. As the *Domain Border* implicates, the transformation and analysis are hidden from the architects. They shall only see everything above the border, namely the *Architectural Model*, the *Constraint Definition* and the *Result* which only requires already known terminology.

We exemplarily realized these mappings based on DDSA, which offers annotations to mark data flows and data characteristics in the *Architectural Model*. The *Annotated Model* is mapped to a set of Prolog predicates [4]. We implemented a *DSL Mapping* which maps constraints to Prolog goals. The Prolog engine tries to satisfy these goals. Every solution to the Prolog program represents one constraint violation. We describe the semantics of our DSL based on these mappings.

Listing 3
Simplified Prolog code of the running example constraint

```
1 c_EnforceRBAC(OP, S, P, VS_rights, VS_roles):-
2  S = [OP | _], stackValid(S),
3  findall(I, arg(S, P, 'rights', I), VS_rights),
4  findall(I, op(OP, 'roles', I), VS_roles),
5  intersection(VS_rights, VS_roles, Temp_0),
6  length(Temp_0, 0).
```

Listing 3 shows a simplified Prolog clause which results of the *DSL Mapping* of the constraint in Listing 1. *Selectors* and *Conditions* are mapped to predicates which are combined using conjunction. They use built-in predicates, e.g., `findall` and predicates from the DDSA API, e.g., `stackValid`.

We highlight several parts to illustrate the mappings described above. Line 3 and 4 reference `rights` and `roles` which are annotated characteristics from the architectural model. Here, the *Transformation Trace* is used to resolve the mapped characteristics in the *Transformed Model*. The use of predicates depends on the *Selectors* from the *DSL Model*, e.g., the `findall` predicate is used because `roles` and `rights` are set variables. *Conditions* are mapped from the strictly functional paradigm to the logical paradigm, e.g., by using temporary variables like `Temp_0` and unfolding nested expressions [25]. The snippet also shows the usage of *Global Constants*, e.g., to identify the meaning of variables.

Listing 4
MAPPED ANALYSIS RESULTS OF A CONSTRAINT VIOLATION

```
1  CONSTRAINT
2  Constraint name: "EnforceRBAC"
3  Violations found: 1
4
5  CONSTRAINT VIOLATIONS
6   1. Parameter "CCD" not allowed in "getDetails"
7   - Call Stack: "getDetails", "bookFlight"
8   - Variables:
9      - Variable "accessRights" set to "User"
10     - Variable "userRoles" set to "Airline"
```

Listing 4 shows an excerpt from the mapped and *Serialized Results*. In this violation, credit card details are passed in the flight booking process without prior permission by the user. This is detected due to the RBAC rights and roles mismatch. The *Result Mapping* enables architects to identify the defect with information on location, call stack and declared variables.

VI. EVALUATION

In this section, we present the evaluation of our approach. We define goals, show the evaluation design, and summarize the results. For the sake of brevity, this section does not focus on details. A more in-depth evaluation can be found in [23].

A. Evaluation Goals

We use a *Goal-Question-Metric*-plan [26] to evaluate both our DSL (**C1**) presented in Section IV and the mapping (**C2**) shown in Section V. We define the following goals:

- **G1** Evaluate the expressiveness of the DSL for availability of concepts which allow versatile data flow constraint definitions to aid software architects.
- **G2** Evaluate the usability of the DSL for improvement of architect's analysis productivity by reducing complexity.
- **G3** Evaluate the mapping for preservation of semantics of the DSL compared to the native formalism.

We choose goal **G1** and **G2** because a sufficient expressiveness has to be paired with a decent usability [11], e.g., by hiding complexity, requiring less knowledge or effort to define data flow constraints. Both quality attributes cannot substitute each other. Goal **G3** evaluates the correctness of our mapping to the underlying formalism. To be applicable, data flow constraints which are mapped from our DSL must yield equivalent analysis results. An erroneous transformation would highly decrease the quality of our approach.

B. Evaluation Design

We present questions and metrics together with the evaluation design. Our evaluation is based on data flow constraints gathered from case studies and related work. This includes the *Geolocation* scenarios derived from privacy violations in cloud services [27], the *ContactSMSManager* and *DistanceTracker* case studies from iFlow [28] which were used to evaluate DDSA [4] and the *SecureLinks* constraint from UMLsec [13].

Goal **G1** evaluates the DSL's expressiveness. We ask:

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

The answer to this question is twofold: Regarding the lower limit, we analyze whether scenarios from additional case studies [1], [13] can be expressed (**M1.1**). Additionally, we evaluate whether the underlying data flow analysis formalism is used comprehensively which forms the upper limit (**M1.2**).

Goal **G2** evaluates the DSL's usability. We ask:

- **Q2.1** Does the DSL provide abstraction from the underlying formalism to the architectural domain?
- **Q2.2** Which knowledge is required to define constraints?

To answer question **Q2.1**, we evaluate whether our DSL hides complexity by measuring the required effort to use our DSL compared to the underlying formalism, i.e., native Prolog code (**M2.1**). We build 8 classes of change scenarios and count required atomic steps (e.g., changing an attribute) to perform these scenarios. To answer question **Q2.2**, we collect knowledge areas in the analysis process shown in Figure 3 and discuss the architect's knowledge which is required with and without our approach (**M2.2**). We distinguish between essential and accidental complexity [29].

Goal **G3** evaluates the correctness of our mapping. We ask:

- **Q3.1** Is the mapping correct and thus preserves semantics?
- **Q3.2** Does using the native analysis formalism compared to the mapped DSL yield the same constraint violations?

Question **Q3.1** evaluates the correctness of our mapping and "whether the semantics of the input model were preserved in the output model of a transformation" [30]. We chose to perform a correctness proof (**M3.1**) as advised in the domain of compiler construction [31]–[33]. We aim to show that the meanings of constraints defined in our DSL (the input model) imply the meanings of mapped constraints in the analysis formalism (the output model). We use universal algebra for the formalization as proposed by Jackson et al. [34] and perform a structural induction. This can be denoted as follows:

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

$R_{\Upsilon_{DSL}}$ represents all possible realizations of constraints in the DSL domain Υ_{DSL}. val_{DSL} and val_A yield the meaning of a realization in the DSL or analysis domain. $[\![\,]\!]^{\gamma}$ represents the transformational interpretation, i.e., the *DSL Mapping*. If the implication holds for every expressible constraint r of our DSL, the analysis shall yield results which comply with the defined constraint despite the domain gap.

Complimentary, we answer question **Q3.2** and evaluate the mapping empirically by comparing analysis results with and without using our DSL to define data flow constraints. We collect existing constraints [4] which use the underlying formalism and reformulate them using our DSL. The reformulated and mapped constraints shall yield the same constraint violations as the native constraints using the formalism directly (**M3.2**). We apply the metrics of binary classification [35]. Note, that this is not equivalent to the consideration of question **Q3.1** because the reformulation also depends on the generalization prior to the definition of the DSL. Thus, neither of both question's results imply each other.

C. Evaluation Results

We present and discuss our evaluation results for every question individually. To answer question **Q1**, we expressed all data flow scenarios described above using our DSL (**M1.1**). We were able to formulate data flow constraints using our DSL for all scenarios, except for one *Geolocation* scenario [27]. This scenario handles personally identifiable information which prohibits selected data flow from different location to be joined together. This cannot be expressed using our DSL due to the lack of consideration of interaction of multiple types of data flows in one constraint. We consider this to be a limitation to the current version of the DSL. Complementary, we evaluate whether our DSL uses the analysis formalism comprehensively (**M1.2**). For all elements of the analysis interface, we analyzed whether they can be expressed using our DSL. In total, 19 of 24 Prolog predicates can be expressed. It shall be noted that the interface also contains predicates which are meant for testing purposes which explains the remaining gap of 5 predicates.

To answer question **Q2.1**, we compared the effort of different change scenarios while using our DSL compared to the underlying formalism (**M2.1**). Here, both the DSL and the formalism are nearly on a par with 18 compared to 19 atomic steps. We expected this result because Prolog is a high-level programming language with much higher general expressiveness than our DSL. In some cases, like in the inversion of *Selectors*, the DSL abstracts from multiple interface calls and thus requires less effort. Regarding *Conditions*, writing constraints directly in Prolog requires less effort due to the good support of built-in evaluation predicates.

The abstraction cannot only be measured in terms of required effort but also regarding the required knowledge. To answer question **Q2.2**, we discuss which knowledge is required by architects to use our DSL in comparison to the underlying formalism (**M2.2**). Some knowledge is required despite the approach, namely knowledge about the *Architectural Model* and its annotations. We consider this to belong to essential complexity, since architects need to know the architecture and its data in order to describe data flow constraints. Knowledge about *Constraint Definition* and how to interpret mapped *Results* is only required when using our DSL. In contrast, using the analysis formalism directly requires knowledge about the mapping and nature of the transformed architecture and its elements as well as the analysis interface. Some knowledge is never required, independent of the selected approach, e.g., the inner details of the data flow analysis which are hidden by the analysis interface. Regarding the domain gap between architecture and formalism, we achieved to only require knowledge from the architectural domain. The *DSL Mapping* enables architects to define constraints without knowing analysis details and the *Result Mapping* enables them to read the results without considering the raw analysis results in the formalism abstraction layer. We consider the DSL to be closer to the architectural domain and thus easier to understand by architects. We reduced what we consider to be accidental complexity in the context of data flow constraint definitions.

In order to evaluate the mapping, we performed a correctness proof to answer question **Q3.1**. We formalized the core language presented in Section IV, its semantics, and the mapping (**M3.1**). We performed a structural induction showing that proposition holds for every analyzed constraint variation. For the complete formalization and induction, please see [23].

To answer **Q3.2**, we evaluated the equivalence of analysis results empirically and compared violations resulting from mapped DSL constraints to violations found by using native constraints formulated in the underlying formalism. The results consist of 100 equivalent violations. 0 violations were missing but 9 violations were additionally found for the *ContactSMSManager* scenarios. The results regarding equivalent violations (*true-positive*), missing violations (*false-negative*), and additional violations (*false-positive*) yield a precision of approximately *92%* and a recall of *100%*. The remaining error of additional violations was traced back to the generalization of existing constraints described in Section IV. Here, the lack of precision in the constraint definition results in additional violations. This will be treated in future versions of the DSL.

D. Threats to Validity

We briefly discuss the threats to validity by using the scheme proposed by Runeson and Höst [36]. The most noteworthy threat to *internal validity* arises from the fact that several answers depend on the authors' experience. This includes formulating constraints (**Q1**) and writing Prolog code (**Q2.1**). More experienced developers might have written more optimized code. Regarding *external validity*, our selection of scenarios to evaluate the questions **Q1** and **Q3.2** limits the generalizability. Our DSL expressiveness could be enhanced and validated more comprehensively by considering more scenarios which is a known limitation. To ensure *construct validity*, we included the results of other work on the development of DSLs [11], [12], [37]–[39] and applied a *GQM*-plan.

VII. Conclusion

In this paper, we proposed an approach to express data flow constraints in the architectural domain. We defined a DSL which uses the terminology known from data flow modeling and abstracts from the analysis. We also defined mappings for constraints and analysis results to bridge the gap between the architectural domain and the analysis formalism. We presented an exemplary realization of the approach based on DDSAs. The benefit of our approach is that software architects can conduct confidentiality analyses without leaving the architectural domain which shall enhance their formulation capabilities and yield more understandable results while requiring less effort. Both mapping and analysis are fully automated and hidden from the architects. In future work, we want to consider more scenarios to enhance the DSL's expressiveness.

Acknowledgment

This work is funded by the DFG (German Research Foundation) – project number 432576552, HE8596/1-1 (FluidTrust) and the KASTEL institutional funding and also supported by the Czech Science Foundation project 20-24814J.

References

[1] K. Katkalov, K. Stenzel, M. Borek, and W. Reif, "Model-driven development of information flow-secure systems with IFlow", in *2013 International Conference on Social Computing*, 2013, pp. 51–56.

[2] International Organization for Standardization, "ISO/IEC 27000:2018(e) information technology – security techniques – information security management systems – overview and vocabulary", Standard, 2018.

[3] J. Isaak and M. J. Hanna, "User data privacy: Facebook, cambridge analytica, and privacy protection", *Computer*, vol. 51, no. 8, pp. 56–59, 2018.

[4] S. Seifermann, R. Heinrich, and R. Reussner, "Data-driven software architecture for analyzing confidentiality", in *2019 IEEE International Conference on Software Architecture (ICSA)*, 2019, pp. 1–10.

[5] S. Peldszus, K. Tuma, D. Struber, J. Jurjens, and R. Scandariato, "Secure data-flow compliance checks between models and code based on automated mappings", presented at the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), 2019, pp. 23–33.

[6] A. Shostack, *Threat Modeling: Designing for Security*. John Wiley & Sons, 12, 2014.

[7] B. Boehm and V. Basili, "Software defect reduction top 10 list", *Computer*, vol. 34, no. 1, pp. 135–137, 2001, Conference Name: Computer.

[8] S. Zdancewic, "Challenges for information-flow security", *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, p. 5, 2004.

[9] H. Yu, X. He, S. Gao, and Y. Deng, "Formal software architecture design of secure distributed systems", *SEKE*, p. 9, 2003.

[10] J. Jürjens, "Towards development of secure systems using UMLsec", in *Fundamental Approaches to Software Engineering*, red. by G. Goos, J. Hartmanis, and J. van Leeuwen, Series Title: Lecture Notes in Computer Science, 2001, pp. 187–200.

[11] R. Paige, J. Ostroff, and P. Brooke, "Principles for modeling language design", *Information and Software Technology*, vol. 42, no. 10, pp. 665–675, 2000.

[12] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages", *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[13] J. Jürjens, "UMLsec: Extending UML for secure systems development", in *UML 2002 — The Unified Modeling Language*, red. by G. Goos, J. Hartmanis, and J. van Leeuwen, vol. 2460, Series Title: Lecture Notes in Computer Science, 2002, pp. 412–425.

[14] K. Katkalov, *Modeling the Travel Planner Application with IFlow*, Accessed 2/7/2021, 2013. [Online]. Available: https://kiv.isse.de/projects/iflow/TravelPlannerSite/index.html.

[15] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models", *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[16] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures", in *2013 35th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, 2013, pp. 662–671.

[17] K. Tuma, L. Sion, R. Scandariato, and K. Yskout, "Automating the early detection of security design flaws", p. 11, 2020.

[18] M. Guerriero, "Defining, enforcing and checking privacy policies in data-intensive applications", in *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, 2018, p. 11.

[19] C. Gerking, D. Schubert, and E. Bodden, "Model checking the information flow security of real-time systems", in *Engineering Secure Software and Systems*, 2018, pp. 27–43.

[20] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study", Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[21] T. DeMarco, "Structure analysis and system specification", in *Pioneers and Their Contributions to Software Engineering*, 1979, pp. 255–288.

[22] R. Sandhu and P. Samarati, "Access control: Principle and practice", *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[23] S. Hahner, "Domain-specific language for data-driven design time analyses and result mappings for logic programs", Master's Thesis, Karlsruher Institut für Technologie (KIT), 2020, 138 pp. DOI: 10.5445/IR/1000123271.

[24] ——, *Domain-specific Language for Data-driven Design Time Analyses and Result Mappings for Logic Programs - Data Set*, Accessed 2/7/2021, 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3973100.

[25] J. Cabot, R. Clarisó, and D. Riera, "On the verification of UML/OCL class diagrams using constraint programming", *Journal of Systems and Software*, vol. 93, pp. 1–23, 2014.

[26] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 728–738, 1984.

[27] E. Schmieders, A. Metzger, and K. Pohl, "Runtime model-based privacy checks of big data cloud services", in *Service-Oriented Computing*, 2015, pp. 71–86.

[28] K. Katkalov, "Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme", PhD thesis, University of Augsburg, 2017.

[29] Brooks, "No silver bullet essence and accidents of software engineering", *Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[30] A. Narayanan and G. Karsai, "Towards verifying model transformations", *Electronic Notes in Theoretical Computer Science*, vol. 211, pp. 191–200, 2008.

[31] F. L. Morris, "Advice on structuring compilers and proving them correct", in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*, 1973, pp. 144–152.

[32] J. W. Thatcher and E. G. Wagner, "More on advice on structuring compilers and proving them correct", *Theoretical Computer Science*, vol. 15, no. 3, p. 27, 1981.

[33] P. Dybjer, "Using domain algebras to prove the correctness of a compiler", in *STACS 85*, vol. 182, Series Title: Lecture Notes in Computer Science, 1985, pp. 98–108.

[34] E. Jackson and J. Sztipanovits, "Formalizing the structural semantics of domain-specific modeling languages", *Software & Systems Modeling*, vol. 8, no. 4, pp. 451–478, 2009.

[35] D. M. W. Powers, "Evaluation: From precision, recall and f-measure to ROC, informedness, markedness and correlation", *CoRR*, vol. abs/2010.16061, 2011.

[36] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical software engineering*, vol. 14, no. 2, p. 131, 2009.

[37] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography", *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[38] M. van Amstel, M. van den Brand, and L. Engelen, "An exercise in iterative domain-specific language design?", in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) on - IWPSE-EVOL '10*, 2010, pp. 48–57.

[39] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, "Design guidelines for domain specific languages", p. 7, 2014.