A Formal Approach to Data Validation Constraints in MDE

Alessandro Rossini, Khalid A. Mughal, Uwe Wolter

Department of Informatics
University of Bergen
Bergen, Norway

Adrian Rutle, Yngve Lamo

Department of Computer Engineering
Bergen University College
Bergen, Norway

Abstract

Software security encompasses the measures taken to ensure confidentiality, integrity and availability in software systems. In present-day software development, security is often an afterthought rather than part of the software development life-cycle. In order to reveal potential security flaws before a software system is actually implemented, security aspects should be taken into account starting from the early phases of the development. With model-driven engineering (MDE) gaining momentum in both academia and industry, an interesting challenge is the specification of security constraints within software models. In this paper we focus on data validation – the process of ensuring that a system operates on correct and meaningful data – in the context of MDE. Our contribution is a formal approach to the specification of data validation constraints which involve multiple structural properties. In addition, constraints specified at model level are mapped to Java annotations which are then transformed to executable tests by an existing data validation framework.

Keywords: data validation; model-driven engineering; category theory; Diagram Predicate Framework; SHIP Validator

1 Introduction

Software systems are nowadays widespread in all walks of society. Violating the confidentiality, integrity and availability of these systems can therefore lead to a negative impact on the economy and health. Software security aims at ensuring that these properties are not compromised. In present-day software development, security is often neglected because of lack of skills and budget, and time-to-market constraints. Typically, security concerns are considered far too late when the system is already nearing deployment. This is clearly insufficient since security aspects should be taken into account starting from the early phases of the development [8,11] in order to reveal potential security flaws before a software system is actually implemented.

Model-driven engineering (MDE) is a branch of software engineering which aims at improving productivity, quality, and cost-effectiveness of software by shifting the paradigm from code-centric to model-centric. MDE promotes models and modelling languages as the main artefacts of the development process and model transformation as the primary technique to generate (parts of) software systems out of models. Models enable developers to reason at a higher level of abstraction while model transformation alleviates developers from repetitive and error-prone tasks such as coding.

In this regard, an interesting challenge is the specification of security constraints within models. In this paper we focus on data validation – the process of ensuring that a system operates on correct and meaningful data – in the context of MDE. The lack of proper data validation is listed as the most prevalent cause of software vulnerabilities by the OWASP [14].

In the state-of-the-art of MDE, models are typically specified by means of modelling languages such as the Unified Modeling Language (UML) [13]. These modelling languages are diagrammatic and allow for the specification of constraints on single structural properties, e.g., a data validation constraint on a single input field. However, the specification of complex constraints on multiple structural properties, e.g., data validation constraint on multiple input fields, requires textual constraint languages such as the Object Constraint Language (OCL) [12].

It is the authors' experience that a completely diagrammatic approach to the specification of data validation constraints in MDE would be desirable [17]. The contribution of this paper is a formal approach to the specification of data validation constraints which can involve multiple, interdependent structural properties. The underpinning of the proposed approach is the Diagram Predicate Framework (DPF) [15,16,17,18] which provides a formalisation of (meta)modelling and model transformation based category theory [1] and graph transformation [5]. The paper also shows how data validation constraints specified at model level are mapped to Java annotations. These annotations are in turn transformed to executable tests at run-time by the SHIP Validator [7,10], a Java based framework which enables the validation of multiple interdependent properties of Java objects.

The remainder of the paper is structured as follows. Section 2 presents DPF. Section 3 introduces the formal approach to data validation by means of a running example. In Section 4, the current research in security within MDE is summarised. Finally, in Section 5, some concluding remarks and ideas for future work are outlined.

2 Diagram Predicate Framework

Before introducing DPF, the terminology adopted in this paper is clarified. The term *model* has different meanings in different contexts. In software engineering, a model denotes "an abstraction of a (real or language-based) system allowing predictions or inferences to be made" [9]. Models in software engineering are typically diagrammatic.

The term *diagram* has also different meanings in different contexts. In software engineering, a diagram denotes a structure which is based on graphs, i.e., a collection of nodes together with a collection of arrows between nodes. Graphs are a well-known and well-understood means to represent structural and behavioural properties of a software system [5], e.g., Entity-Relationship (ER) diagrams and UML diagrams [13].

Since graph-based structures are often visualised in a natural way, the terms *diagrammatic* and *visual* and are often treated as synonyms. In this paper, however, visualisation and diagrammatic syntax are clearly distinguished; i.e., this work focuses on syntax and semantics of diagrammatic models independent of their visualisation.

In DPF, a model is represented by a *specification* $\mathfrak{S}$. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$ consists of an *underlying graph* S together with a set of *atomic constraints* $C^{\mathfrak{S}}$ which are specified by a *signature* Σ. A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of *predicates* $\pi \in \Pi^{\Sigma}$, each having an arity (or shape graph) $\alpha^{\Sigma}(\pi)$, a proposed visualisation and a semantic interpretation. An atomic constraint (π, δ) consists of a predicate $\pi \in \Pi^{\Sigma}$ together with a graph homomorphism $\delta : \alpha^{\Sigma}(\pi) \to S$ from the arity of the predicate to the underlying graph of the specification.

Definition 2.1 [Signature] A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of predicate symbols Π^{Σ} and a map α^{Σ} which assigns a graph to each predicate symbol $\pi \in \Pi^{\Sigma}$. $\alpha^{\Sigma}(\pi)$ is called the *arity* of the predicate symbol π.

Definition 2.2 [Atomic constraint] Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, an atomic constraint (π, δ) on a graph S consists of a predicate symbol $\pi \in \Pi^{\Sigma}$ and a graph homomorphism $\delta : \alpha^{\Sigma}(\pi) \to S$.

Definition 2.3 [Specification] Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$ consists of a graph S and a set $C^{\mathfrak{S}}$ of atomic constraints (π, δ) on S with $\pi \in \Pi^{\Sigma}$.

The semantics of nodes and arrows of the underlying graph of a specification has to be chosen in a way which is appropriate for the corresponding modelling environment [17]. In object-oriented structural modelling, each object may be related to a set of other objects. Hence, it is appropriate to interpret nodes as sets and arrows $\mathsf{X} \xrightarrow{\mathsf{f}} \mathsf{Y}$ as multi-valued functions $f : X \to \wp(Y)$. The powerset $\wp(Y)$ of Y

is the set of all subsets of Y, i.e., $\wp(Y) = \{A \mid A \subseteq Y\}$. Moreover, the composition of two multi-valued functions $f : X \to \wp(Y)$, $g : Y \to \wp(Z)$ is defined by $(f;g)(x) := \bigcup\{g(y) \mid y \in f(x)\}$.

Example 2.4 [Signature and specification] Let us consider an information system for the management of students and universities. The information system has the following requirements:

(i) A student studies at *one to four* universities.

(ii) A university educates *none to many* students.

Table 1 shows a signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$ which is suitable for object-oriented structural modelling.

Table 1
A signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$

π	$\alpha^\Sigma(\pi)$	**Proposed vis.**	**Semantic interpretation**
`[mult(`m, n`)]`	$1 \xrightarrow{a} 2$	$\boxed{\text{X}} \xrightarrow[\text{[m..n]}]{\text{f}} \boxed{\text{Y}}$	$\forall x \in X : m \leq \lvert f(x)\rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[injective]`	$1 \xrightarrow{a} 2$	$\boxed{\text{X}} \xrightarrow[\text{[inj]}]{\text{f}} \boxed{\text{Y}}$	$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[surjective]`	$1 \xrightarrow{a} 2$	$\boxed{\text{X}} \xrightarrow[\text{[surj]}]{\text{f}} \boxed{\text{Y}}$	$\forall y \in Y\ \exists x \in X : y \in f(x)$
`[inverse]`	$\begin{tikzcd} 1 \arrow[r, bend left, "a"] & 2 \arrow[l, bend left, "b"] \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{X}} \arrow[r, bend left, "\text{f}"] \arrow[r, phantom, "\text{[inv]}"] & \boxed{\text{Y}} \arrow[l, bend left, "\text{g}"] \end{tikzcd}$	$\forall x \in X$, $\forall y \in Y : y \in f(x)$ iff $x \in g(y)$

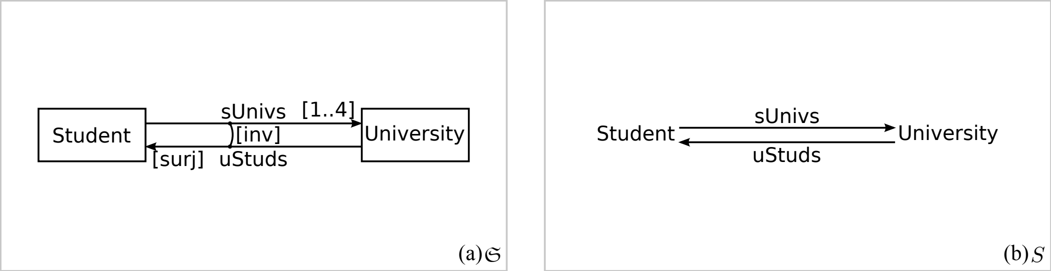


Figure 1. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ and its underlying graph S

Fig. 1(a) shows a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ which is compliant with the requirements above. Fig. 1(b) shows the underlying graph S of $\mathfrak{S}$, i.e., the graph of $\mathfrak{S}$ without any atomic constraints.

In $\mathfrak{S}$, the nodes Student and University are interpreted as sets $Student$ and $University$, and the arrows sUnivs and uStuds are interpreted as multi-valued functions $sUnivs : Student \to \wp(University)$ and $uStuds : University \to \wp(Student)$, respectively.

Based on the requirement i, the function $sUnivs$ has cardinality between one and four. This is enforced by the atomic constraint $(\texttt{[mult(}1,4\texttt{)]},\delta_1)$ on the arrow sUnivs. Moreover, the function $uStuds$ is *surjective*. This is enforced by the atomic constraint $(\texttt{[surjective]},\delta_3)$ on the arrow uStuds. Finally, the functions $sUnivs$ and $uStuds$ are *inverse* of each other, i.e., $\forall s \in Student$ and $\forall u \in University : s \in uStuds(u)$ iff $u \in sUnivs(s)$. This is enforced by the atomic constraint $(\texttt{[inverse]},\delta_2)$ on sUnivs and uStuds. The graph homomorphisms δ_1, δ_2 and δ_3 are defined as follows (see Table 2):

$$\begin{array}{llll}
\delta_1(1) = \text{Student}, & \delta_1(2) = \text{University}, & \delta_1(a) = \text{sUnivs} & \\
\delta_2(1) = \text{Student}, & \delta_2(2) = \text{University}, & \delta_2(a) = \text{sUnivs}, & \delta_2(b) = \text{uStuds} \\
\delta_3(1) = \text{University}, & \delta_3(2) = \text{Student}, & \delta_3(a) = \text{uStuds} &
\end{array}$$

Table 2
The atomic constraints $(\pi,\delta) \in C^{\mathfrak{S}}$ and their graph homomorphisms

(π,δ)	$\alpha^{\Sigma}(\pi)$	$\delta(\alpha^{\Sigma}(\pi))$
$(\texttt{[mult(}1,4\texttt{)]},\delta_1)$	$1 \xrightarrow{a} 2$	$\text{Student} \xrightarrow{\text{sUnivs}} \text{University}$
$(\texttt{[inverse]},\delta_2)$	$1 \underset{b}{\overset{a}{\rightleftarrows}} 2$	$\text{Student} \underset{\text{uStuds}}{\overset{\text{sUnivs}}{\rightleftarrows}} \text{University}$
$(\texttt{[surjective]},\delta_3)$	$1 \xrightarrow{a} 2$	$\text{University} \xrightarrow{\text{uStuds}} \text{Student}$

Remark 2.5 [Predicate symbols] Some of the predicate symbols in Σ (see Table 1) refer to single predicates, e.g., [surjective], while some others refer to a family of predicates, e.g., $\texttt{[mult(}m,n\texttt{)]}$. In the case of $\texttt{[mult(}m,n\texttt{)]}$, the predicate is parametrised by the (non-negative) integers m and n, which represent the lower and upper bounds, respectively, of the cardinality of the function which is constrained by this predicate.

The semantics of predicates of the signature Σ (see Table 1) is described using the mathematical language of set theory. In an implementation, the semantics of a predicate is typically given by the code of a corresponding validator where both the mathematical and the validator semantics should coincide. However, it is not necessary to choose between the above mentioned possibilities; it is sufficient to know that any of these possibilities defines valid instances of predicates.

Definition 2.6 [Semantics of predicates] Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, a semantic interpretation $[\![..]\!]^{\Sigma}$ of Σ consists of a mapping that assigns to each predicate symbol $\pi \in \Pi^{\Sigma}$ a set $[\![\pi]\!]^{\Sigma}$ of graph homomorphisms $\iota : O \to \alpha^{\Sigma}(\pi)$, called valid instances of π, where O may vary over all graphs. $[\![\pi]\!]^{\Sigma}$ is assumed to be closed under isomorphisms.

The semantics of a specification is defined in the so-called *fibred* way [4,20]; i.e., the semantics of a specification is given by the set of its instances. An instance

(I, ι) of a specification $\mathfrak{S}$ consists of a graph I together with a graph homomorphism $\iota : I \to S$ which satisfies the set of atomic constraints $C^{\mathfrak{S}}$.

To check that an atomic constraint is satisfied in a given instance of a specification $\mathfrak{S}$, it is enough to inspect only the part of $\mathfrak{S}$ which is affected by the atomic constraint. This kind of restriction to a subpart is obtained by the pullback construction [1], which can be regarded as a generalisation of the inverse image construction.

Definition 2.7 [Instance of specification] Given a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$, an instance (I, ι) of $\mathfrak{S}$ consists of a graph I and a graph homomorphism $\iota : I \to S$ such that for each atomic constraint $(\pi, \delta) \in C^{\mathfrak{S}}$ we have $\iota^* \in [\![\pi]\!]^{\Sigma}$, where the graph homomorphism $\iota^* : O^* \to \alpha^{\Sigma}(\pi)$ is given by the following pullback:

$$\begin{tikzcd}
\alpha^{\Sigma}(\pi) \arrow[r, "\delta"] & S \\
O^* \arrow[u, "\iota^*"] \arrow[r, "\delta^*"'] \arrow[ur, phantom, "P.B."] & I \arrow[u, "\iota"']
\end{tikzcd}$$

3 Data Validation

A running example based on [7,10] is adopted to show how the formal approach to (meta)modelling can be applied to the problem of data validation. Note that the example is kept intentionally simple, retaining only the details which are relevant for the discussion.

Example 3.1 [International money transfers] Let us consider international money transfers. *IBAN* (International Bank Account Number) is the standard for identifying bank accounts internationally. Some countries have not adopted this standard and, for money transfer to these countries, a special *clearing code* is needed in combination with the plain *account number*. *BIC* (Bank Identifier Code) is the standard for identifying banks globally. Therefore, a form for international money transfers should contain (at least) the input fields bic, iban, account and clearingCode. Moreover, supposing that the currency is Euro, the form should also contain the input fields amountEuros and amountCents. In addition, the transfer system should satisfy the following requirements:

(i) The BIC code of the beneficiary's bank is required.

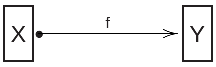
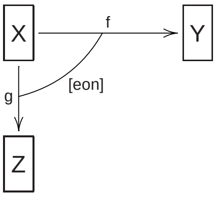
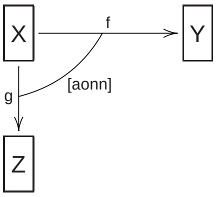
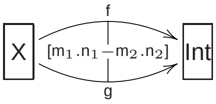
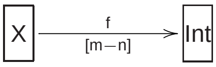
(ii) Either the IBAN or both clearing code and account number are required.

(iii) The amount to transfer must be between 0.01 and 100000.00 Euros.

Table 3 shows a signature $\Phi = (\Pi^{\Phi}, \alpha^{\Phi})$ which contains predicates used to specify data validation constraints.

Note that in the semantic interpretation of the `[cross-range]` predicate we denote lexicographical order by $\leq$.

Table 3
The data validation signature Φ

π	$\alpha^{\Phi}(\pi)$	**Proposed vis.**	**Semantic interpretation**
`[required]`	$\begin{tikzcd} 1 \arrow[r, "a"] & 2 \end{tikzcd}$		$\forall x \in X : f(x)$ defined
`[exactly-one-null]`	$\begin{tikzcd} 1 \arrow[r, "a"] \arrow[d, "b"'] & 2 \\ 3 & \end{tikzcd}$		$\forall x \in X : (f(x)$ defined and $g(x)$ undefined) or $(f(x)$ undefined and $g(x)$ defined)
`[all-or-none-null]`	$\begin{tikzcd} 1 \arrow[r, "a"] \arrow[d, "b"'] & 2 \\ 3 & \end{tikzcd}$		$\forall x \in X : (f(x)$ defined and $g(x)$ defined) or $(f(x)$ undefined and $g(x)$ undefined)
`[cross-range-((`m_1, n_1`),(`m_2, n_2`))]`	$\begin{tikzcd} 1 \arrow[r, bend left, "a"] \arrow[r, bend right, "b"'] & 2 \end{tikzcd}$		$\forall x \in X : (m_1, n_1) \leq (f(x), g(x)) \leq (m_2, n_2)$
`[range(`m, n`)]`	$\begin{tikzcd} 1 \arrow[r, "a"] & 2 \end{tikzcd}$		$\forall x \in X : m \leq f(x) \leq n$

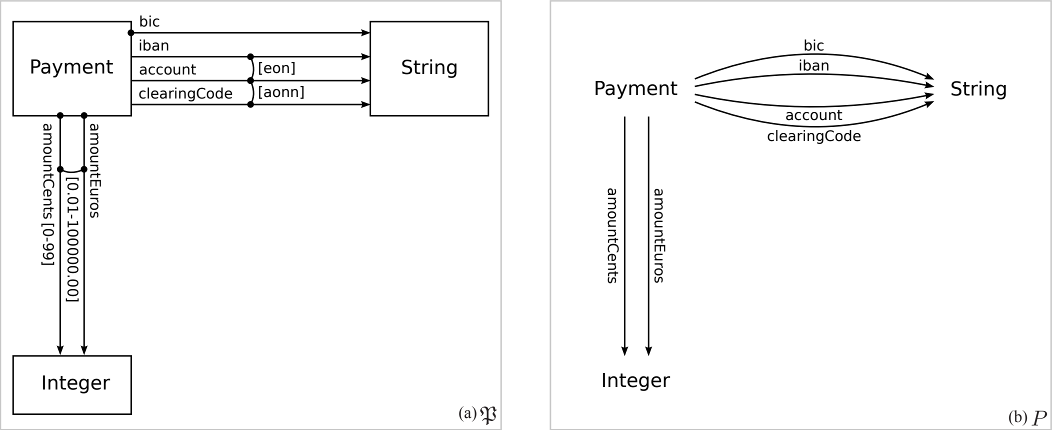


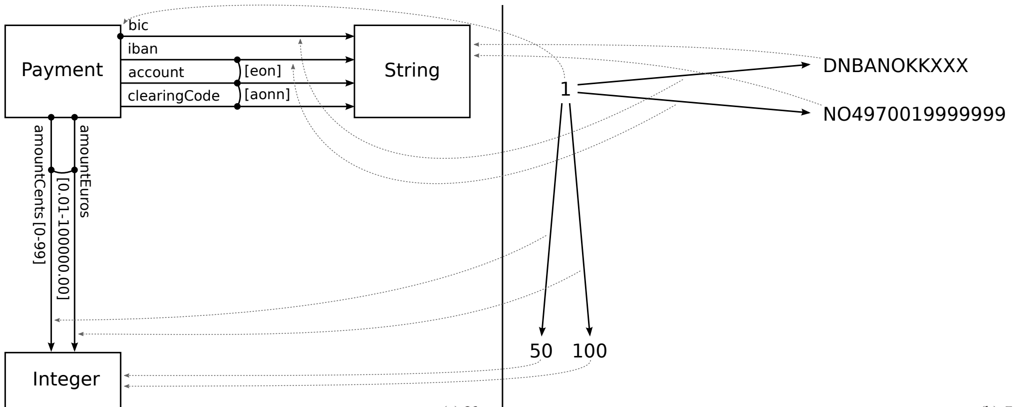
Figure 2. The specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$ and its underlying graph P

Fig. 2(a) shows a specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$ which is compliant with the requirements above. The form is represented by the node Payment while the input fields are represented by the arrows bic, iban, account, clearingCode, amountEuros and amountCents. Fig. 2(b) presents the underlying graph P of $\mathfrak{P}$, i.e., the graph of $\mathfrak{P}$ without any atomic constraints.

In $\mathfrak{P}$, the requirement i is enforced by the atomic constraint (`[required]`, δ_1) on the arrow bic, i.e., $\delta_1 : (1 \xrightarrow{a} 2) \mapsto (\text{Payment} \xrightarrow{\text{bic}} \text{String})$. This atomic constraint ensures that the user provides a value in the input field bic. Moreover, the requirement ii is enforced in $\mathfrak{P}$ by two atomic constraints: (`[exactly-one-null]`,

δ_2) on the arrows iban and account together with ($\texttt{[all-or-none-null]}, \delta_3$) on the arrows account and clearingCode. These atomic constraints ensure that a user provides values in either the input field iban or both the input fields account and clearingCode. Furthermore, the requirement iii is enforced in $\mathfrak{P}$ by the atomic constraint ($\texttt{[cross-range(}(0,1),(100000,0)\texttt{)]}, \delta_4$) on the arrows amountEuros and amountCents. This atomic constraint ensures that the user provides values in the input fields amountEuros and amountCents which sum up to a value within the range 0.01 to 100000.00. In addition, the atomic constraint ($\texttt{[range(}0,99\texttt{)]}, \delta_5$) on the arrow amountCents ensures that a user provides a value in the input field amountCents within the range 0 to 99.

Fig. 3(b) shows a valid instance I of the specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$. Fig. 3 also shows the mappings of the graph homomorphism $\iota : I \to P$ as dashed, grey arrows.



(a) $\mathfrak{P}$ (b) I

Figure 3. The specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$ and a possible instance I

As mentioned, in an implementation, the semantics of a predicate is typically given by the code of a corresponding validator where both the mathematical and the validator semantics should coincide. In this paper, we have chosen to base the implementation of each predicate on the SHIP Validator [7,10]. The XMI serialisation (see Listing 1) of the specification $\mathfrak{P} = (P, C^{\mathfrak{P}} : \Phi)$ specifying the form in Example 3.1 can be transformed to a Java class (see Listing 2) tagged by Java annotations compatible with the SHIP Validator. For each atomic constraint $(\pi, \delta) \in C^{\mathfrak{P}}$ a corresponding Java annotation is attached to the getter methods of the Java class. Note that an atomic constraint on a single arrow, e.g., ($\texttt{[required]}, \delta_1$) on the arrow bic, translates to a single Java annotation, e.g., `@Required` on the method `getBic()`. Likewise, an atomic constraint on multiple arrows, e.g., ($\texttt{[exactly-one-null]}, \delta_2$) on the arrows iban and account, translates to multiple Java annotations, e.g., `@ExactlyOneNull` on the methods `getIban()` and `getAccount()`. The interested reader can download a proof-of-concept implementation of a code generator from [2].

Listing 1: XMI serialisation of the specification $\mathfrak{P} = (P, C^{\mathfrak{P}}\colon \Phi)$

```
<?xml version="1.0" encoding="ASCII"?>
<no.hib.dpf.metamodel:Specification
xmlns:no.hib.dpf.metamodel="http://no.hib.dpf.metamodel"
id="9090a2ec-0e36-4fcc-8f04-3a0226f0a938" name="P">

 <node id="525d2a64-66e1-42f8-aec9-9f186379a77b" name="Payment"/>
 <node id="d3ae4964-d091-41d7-9127-09856b3ce316" name="String"/>
 <node id="0cac0671-a7e0-4d99-8216-14d24f186375" name="Integer"/>

 <arrow id="b5a45cda-3ee0-42a0-a568-81f9e92d7e25" name="bic" source="//@node.0"
     target="//@node.1"/>
 <arrow id="ad030229-b66c-40b5-8f7f-59f1a25e24a8" name="iban" source="//@node.0"
     target="//@node.1"/>
 <arrow id="1d54b8c6-a51b-4858-ade9-0a66522b80eb" name="account" source="//@node
     .0" target="//@node.1"/>
 <arrow id="2c4b8f89-dc27-44e6-bdb4-a0e298c26f85" name="clearingCode" source="//
     @node.0" target="//@node.1"/>
 <arrow id="07a4001b-4c8e-461f-a845-4ac985b0c36d" name="amountEuros" source="//
     @node.0" target="//@node.2"/>
 <arrow id="7559cb35-863a-49dd-a2b3-3e9e893c1356" name="amountCents" source="//
     @node.0" target="//@node.2"/>

 <constraints id="33003eb9-d287-4bd8-9a28-ccf6d3ea9ee0" type="[required]">
   <arrow source="//@arrow.0" />
 </constraints>

 <constraints id="33003eb6-7987-4558-ba28-aaf693349ee0" type="[not-required]">
   <arrow source="//@arrow.1" />
   <arrow source="//@arrow.2" />
   <arrow source="//@arrow.3" />
   <arrow source="//@arrow.4" />
   <arrow source="//@arrow.5" />
 </constraints>

 <constraints id="e0661dc3-0620-44e6-af54-07bf14875c16" type="[exactly-one-null]">
   <arrow source="//@arrow.1" />
   <arrow source="//@arrow.2" />
 </constraints>

 <constraints id="1160e483-b701-4c23-9641-7e73909de528" type="[all-or-none-null]">
   <arrow source="//@arrow.2" />
   <arrow source="//@arrow.3" />
 </constraints>

 <constraints id="e1f2bab1-b58c-4273-97bb-d0cdd14abe45" type="[cross-range]">
   <param name="m1" value="0" />
   <param name="n1" value="01" />
   <param name="m2" value="10000" />
   <param name="n2" value="00" />
   <arrow source="//@arrow.4" />
   <arrow source="//@arrow.5" />
 </constraints>

 <constraints id="9132c6e8-7af9-4fc6-8b67-afac0471b13b" type="[range]">
   <param name="min" value="0" />
   <param name="max" value="99" />
   <arrow source="//@arrow.5" />
 </constraints>

</no.hib.dpf.metamodel:Specification>
```

Listing 2: Java class generated by transformation

```java
public class Payment {

  private String bic;
  private String iban;
  private String account;
  private String clearingCode;

  private int amountEuros;
  private int amountCents;

  @Required
  public String getBic() {
    return bic;
  }

  @ExactlyOneNull
  @NotRequired
  public String getIban() {
    return iban;
  }

  @ExactlyOneNull
  @AllOrNoneNull
  @NotRequired
  public String getAccount() {
    return account;
  }

  @AllOrNoneNull
  @NotRequired
  public String getClearingCode() {
    return clearingCode;
  }

  @IntRange(min=0,max=100000)
  @CrossRange
  public int getAmountEuros(){
    return this.amountEuros;
  }

  @IntRange(min=0,max=99)
  @CrossRange
  public int getAmountCents(){
    return this.amountCents;
  }

}
```

These Java annotations are in turn transformed into executable tests by the SHIP Validator. The interested reader can consult [7,10] for details about the implementation and execution of these tests. Note that the idea of using annotations to hide the actual validation code and, at the same time, tag the properties to be tested, allow the constraints to be easily integrated into existing code. Besides, the validation aspects of the system remain well separated from the application aspects. This separation of concerns facilitates the transformation of the diagrammatic constraints into actual existing working code.

4 Related Work

In [6], an approach to integrate input validation constraints into UML diagrams using OCL is presented. In particular, this approach targets four different UML diagrams, i.e., use case diagram, class diagram, sequence diagram and activity diagram. This solution enables the specification of input validation constraints on behavioural models while our approach targets structural models only. However, it adopts a textual constraint language such as OCL while our approach is completely diagrammatic.

In [8], the author illustrates an approach to enrich UML models with security requirements such as secrecy, integrity and authenticity. The approach exploits UML extension mechanisms such as keywords, tags and constraints. In particular, keywords are used together with tags to specify security requirements on the system, while constraints give criteria to determine if these requirements are satisfied by the UML model. However, keywords and tags can be attached only to single model elements, thus these mechanism are not sufficient to express data validation constraints involving multiple structural properties at the model level. On the contrary, data validation constraints involving multiple structural properties can be expressed in our approach in a diagrammatic fashion.

5 Conclusion and Future Work

In this paper, we have illustrated some of the key aspects of data validation in MDE. We have adopted DPF to define an approach to the specification of data validation constraints in models. Moreover, we have shown how these constraints can be mapped to Java annotations which are transformed to executable tests. The diagrammatic and formal nature of the proposed approach constitutes the main contribution and novelty of this work.

In a future work, we will introduce a reasoning system for the analysis of predicate dependencies and a logic for this analysis. This extension will enable users of the proposed approach to detect possible inconsistencies between data validation constraints. Moreover, we will integrate the code generator, which transforms the constraints at model level to Java annotations, in the DPF Editor [3], a diagrammatic (meta)modelling tool based on DPF and Eclipse Modeling Framework (EMF) [19].

Acknowledgement

The authors would like to thank Øyvind Bech and Dag Viggo Lokøen for the proof-of-concept implementation of the code generator, and Federico Mancini for the support with the SHIP Validator.

References

[1] Barr, M. and C. Wells, “Category Theory for Computing Science (2^{nd} Edition),” Prentice Hall, 1995.

[2] Bech, Ø. and D. V. Lokøen, “DPF to SHIP Validator Proof-of-Concept Transformation Engine,” http://dpf.hib.no/code/transformation/dpf_to_shipvalidator.py.

[3] Bergen University College and University of Bergen, “Diagram Predicate Framework Web Site,” http://dpf.hib.no/.

[4] Diskin, Z. and U. Wolter, *A Diagrammatic Logic for Object-Oriented Visual Modeling*, in: *Proceedings of ACCAT 2007: 2^{nd} Workshop on Applied and Computational Category Theory*, Electronic Notes in Theoretical Computer Science **203/6** (2008), pp. 19–41.

[5] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer, “Fundamentals of Algebraic Graph Transformation,” Springer, 2006.

[6] Hayati, P., N. Jafari, S. M. Rezaei, S. Sarencheh and V. Potdar, *Modeling Input Validation in UML*, in: *Proceedings of ASWEC 2008: 19^{th} Australian Software Engineering Conference* (2008), pp. 663–672.

[7] Hovland, D., F. Mancini and K. Mughal, *The SHIP Validator: An Annotation-Based Content-Validation Framework for Java Applications*, Technical Report 389, Department of Informatics, University of Bergen, Norway (2009).

[8] Jürjens, J., “Secure Systems Development with UML,” Springer, 2005.

[9] Kühne, T., *Matters of (meta-)modeling*, Software and Systems Modeling **5** (2006), pp. 369–385.

[10] Mancini, F., D. Hovland and K. Mughal, *Investigating the Limitations of Java Annotations for Input Validation*, in: *Proceedings of ARES 2010: 5^{th} International Conference on Availability, Reliability and Security* (2010).

[11] McGraw, G., “Software Security: Building Security in,” Addison-Wesley Professional, 2006.

[12] Object Management Group, “Object Constraint Language Specification,” (2010), http://www.omg.org/spec/OCL/2.2/.

[13] Object Management Group, “Unified Modeling Language Specification,” (2010), http://www.omg.org/spec/UML/2.3/.

[14] OWASP, “Top Ten Project,” http://www.owasp.org.

[15] Rossini, A., A. Rutle, Y. Lamo and U. Wolter, *A formalisation of the copy-modify-merge approach to version control in MDE*, Journal of Logic and Algebraic Programming **79** (2010), pp. 636–658.

[16] Rutle, A., “Diagram Predicate Framework: A Formal Approach to MDE,” Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2010).

[17] Rutle, A., A. Rossini, Y. Lamo and U. Wolter, *A Diagrammatic Formalisation of MOF-Based Modelling Languages*, in: M. Oriol, B. Meyer, W. Aalst, J. Mylopoulos, M. Rosemann, M. Shaw and C. Szyperski, editors, *Proceedings of TOOLS 2009: 47^{th} International Conference on Objects, Components, Models and Patterns*, Lecture Notes in Business Information Processing **33** (2009), pp. 37–56.

[18] Rutle, A., A. Rossini, Y. Lamo and U. Wolter, *A formal approach to the specification and transformation of constraints in MDE*, Journal of Logic and Algebraic Programming (To appear).

[19] Steinberg, D., F. Budinsky, M. Paternostro and E. Merks, “EMF: Eclipse Modeling Framework 2.0 (2^{nd} Edition),” Addison-Wesley Professional, 2008.

[20] Wolter, U. and Z. Diskin, *From Indexed to Fibred Semantics – The Generalized Sketch File*, Technical Report 361, Department of Informatics, University of Bergen, Norway (2007).