SoSyM manuscript No.
(will be inserted by the editor)

Formal Validation of Domain-Specific Languages with Derived Features and Well-Formedness Constraints

Oszkár Semeráth · Ágnes Barta · Ákos Horváth · Zoltán Szatmári · Dániel Varró

Received: date / Accepted: date

Abstract Despite the wide range of existing tool support, constructing a design environment for a complex domain-specific language (DSL) is still a tedious task as the large number of derived features and well-formedness constraints complementing the domain metamodel necessitate special handling. Such derived features and constraints are frequently defined by declarative techniques (such graph patterns or OCL invariants).

However, for complex domains, derived features and constraints can easily be formalized incorrectly resulting in inconsistent, incomplete or ambiguous DSL specifications. To detect such issues, we propose an automated mapping of EMF metamodels enriched with derived features and well-formedness constraints captured as graph queries in EMF-IncQuery or (a subset of) OCL invariants into an effectively propositional fragment of first-order logic which can be efficiently analyzed by back-end reasoners.

On the conceptual level, the main added value of our encoding is (1) to transform graph patterns of the EMF-IncQuery framework into FOL and (2) to introduce approximations for complex language features (e.g. transitive closure or multiplicities) which are not expressible in FOL. On the practical level, we identify and address relevant challenges and scenarios for systematically validating DSL specifications. Our approach is supported by a tool and it will be illustrated on analyzing a DSL in the avionics domain. We also present initial performance experiments for the validation using Z3 and Alloy as back-end reasoners.

Oszkár Semeráth · Ágnes Barta · Ákos Horváth · Zoltán Szatmári · Dániel Varró
Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
E-mail: {semerath,varro}@mit.bme.hu

Keywords language validation · derived features · partial snapshots · model queries · SMT-solvers

Acknowledgements This work was motivated by the FP7 ARTEMIS CONCERTO (ART-2012-333053) project and partially supported by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP (Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems) project.

1 Introduction

1.1 Towards validation of domain-specific languages

The design of integrated development environments for complex domain-specific languages (DSL) is still a challenging task nowadays. Advanced environments such as Xtext, the Graphical Modeling Framework (GMF), Graphiti or Sirius built on top of model management frameworks such as Eclipse Modeling Framework (EMF) significantly improve productivity by automating the production of rich editor features (e.g. syntax highlighting, auto-completion, etc.) to enhance modeling for domain experts. Furthermore, there is efficient tool support for validating well-formedness constraints and design rules over large model instances of the DSL using tools like Eclipse OCL [58] or EMF-IncQuery [10]. As a result, Eclipse-based IDEs are widely used in the industry in various domains including business modeling, avionics or automotive.

However, in case of complex, standardized industrial domains (like ARINC 653 [5] for avionics or AUTOSAR [6] in automotive), the sheer complexity of the DSL is a major challenge itself. (1) First, there are hundreds of well-formedness constraints and design rules defined by those standards, and due to the lack of validation, there is no guarantee for their consistency or unambiguity. (2) Moreover, domain metamodels are frequently extended by derived features, which serve as automatically calculated shortcuts for accessing or navigating models in a more straightforward way. In many practical cases, these features are not defined by the underlying standards but introduced during the construction of the DSL environment for efficiency reasons. Anyhow, the specification of derived features can also be inconsistent, ambiguous or incomplete. In general, the mathematical precise validation of DSL specifications themselves have been attempted by only few approaches so far [31], and even these approaches lack a systematic validation process.

As model-driven tools are frequently used in critical systems design to detect conceptual flaws of the system model early in the development process to decrease verification and validation (V&V) costs, those tools should be validated with the same level of scrutiny as the underlying system as part of a software tool qualification process in order to provide trust in their output. Therefore software tool qualification raises several challenges for building trusted DSL tools for a specific domain.

The main objective of this paper is to propose an *automated validation framework* to formally check the specification of DSLs. For that purpose, we *formalize* DSL modeling artifacts (including metamodels and instance models, constraints and derived feature definitions) by first-order logic (FOL) formulae. Then we carry out a wide range of validation tasks by automated theorem proving based on this formalization to show *consistency*, *ambiguity* and *completeness*, *subsumption* or *equivalence* of a DSL. To decrease the development time and cost of DSL tools, we aim to detect design flaws in the early phase of DSL development by highlighting validation problems to the developer directly in the DSL tool itself by back-annotating analysis results. As a side effect, our validation framework can also be used for *generating prototypical well-formed instance models* for a DSL, which can be used for synthesizing test cases, for instance.

Language level validation is a very challenging task because the analysis has to cover an infinite range of possible design models, which necessitates symbolic approaches. The language elements are representable by sets and relations which makes first order logic suitable to formalize them. Different constraint languages attached to the modelling languages are semantically close to first order logic, extending them with additional elements, like transitive closure. However, reasoning over a language level problem is undecidable in general.

Fortunately logic solvers have become more and more powerful.

- SAT-solvers specialized for graph problems are capable of checking large range of models in order to generate counterexamples of bounded size if the validated property is not satisfied by the target DSL, but they cannot prove the correctness.
- On the other hand, SMT-solvers are able to efficiently handle complex logic problems with unlimited domain by solving them with a combination of background theories. An advanced SMT-solver contains decision procedures for the most common logic fragments, like the effectively propositional [44].

Abstraction is a key element of automatically solving logic problems. First, constraint languages use higher order language elements like transitive closure which cannot be explicitly represented in first order logic. Additionally, a more generic problem can be solved efficiently if it is represented in the target scope of the backend solver. Thus, with suitable approximations, language properties cannot be directly constructed in the target logic formalism.

In the paper, we make the following contributions:

- We propose an approach for the validation of DSLs which covers the handling of metamodels, well-formedness constraints and derived features captured by graph queries or (a subset of) OCL invariants.
- For this purpose, we define transformation of metamodels, models and constraints into FOL formulae. Our aim is to derive effectively propositional formulae wherever possible, which is an efficiently analyzable fragment of FOL. We also propose powerful approximation techniques to handle complex language constructs which cannot be represented in FOL.
- To refine the context of DSL validation, we introduce and map partial snapshots which serve as prototypical initial instance models required to be included when constructing a consistency proof for a DSL. Furthermore, analysis results are also retrieved in the form of partial snapshots.
- In order to systematically carry out the validation process for a DSL, we propose a validation workflow, which consequently investigates each language feature to check consistency, completeness and unambiguity (for derived features) and subsumption or equivalence (for well-formedness constraints).
- We provide prototype tool support which takes EMF metamodels with derived features, instance models, constraints captured by graph patterns of the EMF-IncQuery framework or in OCL as input, and carries out DSL validation using back-end reasoners. Validation results are back-annotated to the source

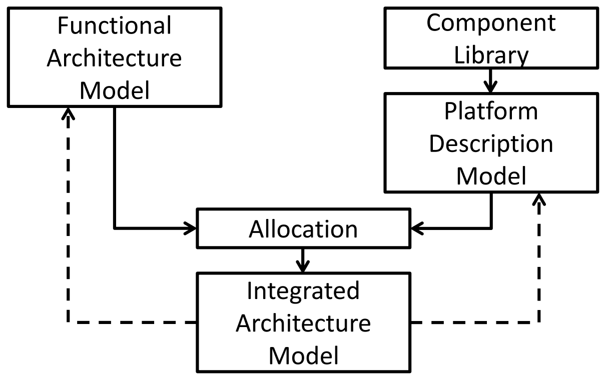

Functional Architecture Model
Component Library
Platform Description Model
Allocation
Integrated Architecture Model

Fig. 1: High-level overview of the Trans-IMA project

DSL specification and to the initial partial model therefore language engineers may inspect those results as regular instance models.

– We carry out an initial performance evaluation on various DSL validation tasks using a motivating example from the avionics domain using the powerful Z3 SMT solver built on high-level decision procedures and Alloy (based on a SAT-solver) as automated back-end reasoning tools.

This paper extends our earlier work in [51] by (i) generalizing partial snapshots, (ii) identifying new validation tasks, (iii) formalizing the DSL specification (metamodels, derived features, well-formedness constraints in graph patterns and OCL) and the validation tasks (iv) providing detailed specification and examples for the DSL-to-FOL transformations and the approximations, and (v) extending the performance evaluation of the validation approach by using both Z3 and Alloy.

Our tool has been successfully applied in case studies of two ongoing projects: the Trans-IMA project [29] serves as a motivating scenario for the current paper taken from the avionics domains, while the R3COP ARTEMIS project [2] used our tool for test case generation for autonomous and cooperative robot systems (which will be documented in an upcoming paper).

1.2 Motivating Scenario

Trans-IMA aims at defining a model-driven approach for the synthesis of complex, integrated Matlab Simulink models capable of simulating the software and hardware architecture of an airplane. The project aimed to (i) define a model-driven development process for allocating software functions captured as Simulink models [39] over different hardware architectures and (ii) develop domain-specific languages and tools for supporting the definition of the allocation process.

The high-level overview of the Trans-IMA challenge is illustrated in Figure 1. In model-driven development of avionics systems, the *functional architecture* and the *platform description* of the system are often developed separately to increase reusability. The former defines the services performed by the system and links between functions to indicate dependencies and communication, while the latter describes platform-specific hardware and software components and their interactions.

1. Functional Architecture Models can be imported from industrial language and tools such as AADL [49] or Matlab Simulink [39] to capture the functional description of different systems.
2. Then the system architect specifies the Platform Description Model from the elements of the Component Library defining the available hardware elements.
3. In the next step the system architect allocates all the functions from the Functional Architecture Model. The allocation itself includes two major parts: (i) the mapping of functions defined in the FAM to their underlying execution elements within the PDM and (ii) the automated discovery of available communication paths for the various information links defined between the allocated FAM elements.
4. Finally, when the allocation is complete and fulfills all safety and design requirements the Integrated Architecture Model is automatically synthesized to enable simulation in Matlab Simulink.

This development environment is built upon eight large metamodels, where complex EMF-IncQuery patterns are extensively used for capturing constraints and derived features. The DSL contains 118 classes, 90 attributes and 170 references, where 56 features were marked as derived (about 20% of total) and each was specified by a corresponding model query. The design rules are defined by 31 well-formedness constraints. The development of other popular standardised industrial DSLs have similar challenges: the AADL standard [49] provides DSLs used mainly in development of avionics systems with 259 classes, 75 implemented well-formedness constraints, and more than the 15% of attributes and references are derived. The AUTOSAR standard [6] specifying DSLs to automotive systems contains more than 1000 classes and about 500 design rules. The definition of such large DSLs is a very challenging task not only due to their size (and thus their sheer complexity) but also to precisely understand the interactions between the additional design rules. Declaring large number of derived features are also challenging with respect to the safety specific well-formedness constraints.

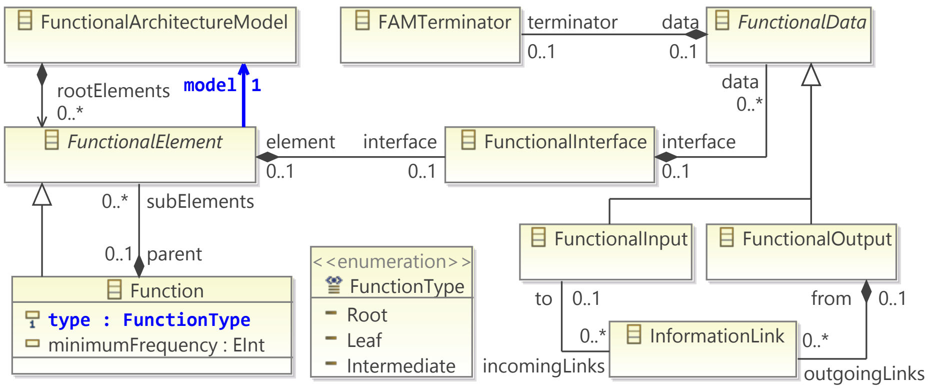


Fig. 2: Metamodel of the Functional Architecture

1.3 Structure of the Paper

The rest of the paper is structured as follows. First we present an overview of defining DSLs and queries in Section 2. Then we present the formalism of partial snapshots in Section 3, which are generalizations of instance models. An overview of the DSL validation approach is provided in Section 4, and the DSL validation workflow is detailed on our avionics case study in Section 5. We intentionally postpone the theoretical details of transforming various DSL constructs to first-order logic to enhance readability of the initial sections of the paper. First the technicalities of transformation is shown in Section 6, afterwards the mapping of the metamodel and partial snapshots are presented in Section 7 and followed by the mapping of constraints (both in graph patterns and in OCL) in Section 8. Section 9 demonstrates the practical feasibility of our approach by presenting prototype tool support and initial experimental evaluation. Related work is discussed in Section 10 and Section 11 concludes our paper.

2 Domain Specific Languages

Complex domain-specific languages (DSL) necessitate a combination of different specification techniques. The abstract syntax of the DSL is usually captured by a metamodel. To create an advanced modeling environment, the metamodel can be augmented with *well-formedness constraints* (or *design rules*), which capture additional restrictions any well-formed instance model needs to respect. Such constraints can be defined by model queries or as OCL invariants. Furthermore, the metamodel can also be enhanced with *derived features*, i.e. attributes and relations calculated from core model elements during model use. In order to illustrate our DSL validation techniques, we use an avionics DSL defined over EMF metamodels and using the language of the EMF-IncQuery [11] framework or OCL to define constraints and derived features over EMF metamodels.

2.1 Metamodeling

Metamodels define the main concepts, relations and attributes of the target domain to specify the basic structure of the models. In this paper, the Eclipse Modeling Framework (EMF) [55] is used for domain modeling.

A simplified metamodel for functional architecture is shown in Figure 2. The FunctionalArchitectureModel element represents the root of a model, which contains each Function (subtype of the FunctionalElement). Functions have a minimumFrequency attribute, a type attribute and multiple FunctionalInterfaces, where each functional data is either a FunctionalOutput (for invoking other functions) or a FunctionalInput (for accepting invocations). An output can be connected to an input through an InformationLink. Finally, if an input or output is not connected to an other Function then it must be terminated in a FAMTerminator.

– **Classes (CLS)**: In this formalism the concepts are represented by *EClasses* (which are simply referred to as classes) that can be instantiated to *EObjects* (or objects). The metamodel can specify finite types with predefined set of $\{\mathsf{l1}, \ldots, \mathsf{ln}\}$ literals by *EEnums*. For both classes and enums, if an e is an instance of a T type it is denoted as $\mathsf{T}(\mathsf{e})$.
– **Generalization (GEN)** can be specified between two classes to express that a more specific (child) class has every structural feature of the more general (parent) class. From a typing perspective, for all metamodel classes *abst*, *spec* with $\mathsf{super}(abst, spec)$, we have $\forall \mathsf{e} : \mathsf{spec}(\mathsf{e}) \Rightarrow \mathsf{abst}(\mathsf{e})$.

– **Abstract (ABS)**: If a class is defined as *abstract*, it is disallowed to have direct instances, i.e. for all abstract classes abs and an instance o with $\mathsf{isAbstract}(abs) \wedge \mathsf{abs}(\mathsf{o})$, there exists a non-abstract subclass cls of abs with o with $\neg\mathsf{isAbstract}(cls) \wedge \mathsf{cls}(\mathsf{o})$
– **References (REF)** *EReferences* defined between classes capture the relations of the domain. When two o and t objects are in a relation r, an *EReference* (or an *EAttribute*) is instantiated leading from o to t denoted as $\mathsf{r}(\mathsf{o},\mathsf{t})$.
– **Attributes (ATT)** *EAttributes* enrich the expressiveness of classes with values of predefined primitive types like integers, strings, etc. If an object o stores a value v as attribute A it is denoted as $A(o, v)$.
– **Type compliance (TC)** Type compliance requires that for any relation $\mathsf{r}(\mathsf{o},\mathsf{t})$, its source and target objects o and t need to have compliant types.
– **Multiplicity (MUL)** The multiplicity of structural features can be limited with upper and lower bound in lower..upper form, i.e. for any relation r leading from o, we have $\mathsf{lower} \leq |\{\mathsf{t} : \mathsf{r}(\mathsf{o},\mathsf{t})\}| \leq \mathsf{upper}$.
– **Containment (CON)** EMF instance models are arranged into a strict containment hierarchy, which is a directed tree along relations marked in the metamodel as containment. Our approach supports models with a single root object. Even though the EMF framework supports resources with multiple model roots, it is not supported by many modelling tools (including its' own tree editor). Additionally, allowing multiple roots lets the solvers generate meaningless models with single isolated objects.
In a containment hierarchy, any non-root model element has exactly one parent as container. Formally, $\forall \mathsf{t} \exists ! \mathsf{o} : \mathsf{nonRoot}(\mathsf{t}) \Rightarrow \mathsf{parent}(\mathsf{t},\mathsf{o})$.
– **Inverse (INV)** Two parallel but opposite directional *inverse* references can be defined as inverses of each other to specify that they always occur in pairs. This means that for all pairs of references forw and back that $\mathsf{inv}(\mathsf{forw},\mathsf{back})$ we have $\forall \mathsf{o},\mathsf{t} : \mathsf{forw}(\mathsf{o},\mathsf{t}) \Leftrightarrow \mathsf{back}(\mathsf{t},\mathsf{o})$.

A model M is a valid instance of a metamodel META (denoted with $\mathsf{M} \models \mathsf{META}$) if all the corresponding constraints above are satisfied, i.e.

$$\mathsf{M} \models \mathsf{CLS,GEN,ABS,REF,ATT,TC,MUL,CON,INV}.$$

In this paper EClass, EEnum, ELiteral, EReference, EAttribute is also used as the set of all classes, enums ... in the metamodel META. EObject notes the set of all objects in an instance model M.

```
   ⟨pattern⟩ → ⟨annotation⟩pattern⟨name⟩(⟨params⟩)
               ⟨bodies⟩
    ⟨params⟩ → ⟨param⟩ | ⟨param⟩,⟨params⟩
     ⟨param⟩ → ⟨var⟩ | ⟨var⟩:⟨EClassifier⟩
    ⟨bodies⟩ → {⟨conlist⟩} | {⟨conlist⟩} or ⟨bodies⟩
   ⟨conlist⟩ → ⟨constraint⟩; | ⟨constraint⟩; ⟨constlist⟩
⟨constraint⟩ → ⟨classifier⟩ | ⟨path⟩ | ⟨equality⟩ |
               ⟨call⟩ | ⟨check⟩
⟨classifier⟩ → ⟨EClass⟩(⟨var⟩)
      ⟨path⟩ → ⟨EClass⟩.⟨featlist⟩(⟨var⟩,⟨var⟩)
  ⟨featlist⟩ → ⟨EAttribute⟩|⟨EReference⟩|
               ⟨EReference⟩.⟨featlist⟩
  ⟨equality⟩ → ⟨var⟩==⟨var⟩|⟨var⟩!=⟨var⟩
      ⟨call⟩ → find ⟨name⟩(⟨binding⟩) |
               neg find ⟨name⟩(⟨binding⟩) |
               find ⟨name⟩+(⟨binding⟩)
   ⟨binding⟩ → ⟨var⟩ | ⟨var⟩,⟨binding⟩
     ⟨check⟩ → check(⟨boolexp⟩)
   ⟨boolexp⟩ → ⟨boolexp⟩&&⟨boolexp⟩|⟨boolexp⟩||⟨boolexp⟩|
               !⟨boolexp⟩|⟨numexp⟩==⟨numexp⟩|⟨var⟩
    ⟨numexp⟩ → ⟨numexp⟩+⟨numexp⟩|⟨numexp⟩-⟨numexp⟩|
               ⟨numexp⟩*⟨numexp⟩|⟨numexp⟩/⟨numexp⟩|
               ⟨var⟩
⟨annotation⟩ → @Constraint|@QueryBasedFeature|ε
```

Fig. 3: A grammar of graph patterns

2.2 Model Queries

Model queries can be frequently captured by graph patterns (GP) [57,11], which are an expressive formalism used for various purposes in model-driven development alternatively for standard OCL constraints [41]. A graph pattern is a graph-like structure representing a condition (or constraint) matched against a typically large instance model.

A model query $\mathsf{q}(\mathsf{p}_1, \ldots, \mathsf{p}_n) = \mathsf{body}$ is defined by a name q and symbolic parameters $\mathsf{p}_1, \ldots, \mathsf{p}_n$, and conditions (or constraints) over the parameters (captured by body). A match m of $\mathsf{q}(\mathsf{p}_1, \ldots, \mathsf{p}_n) = \mathsf{body}$ over model M maps each symbolic parameter p_i to a model element (object, enum literal or primitive) from the target model M, which satisfies the conditions of body: $\forall \mathsf{m} : \mathsf{M} \models \mathsf{body}(\mathsf{m}(\mathsf{params}))$. The task of the query evaluation on a model is to produce each match that satisfies this condition.

EMF-IncQuery offers a textual language describing graph patterns as a set of constraints. Figure 3 summarizes the grammar of the language, the complete query language is described in [11], while relevant language features will be introduced on demand in several examples below adapted from [51].

A graph pattern is identified with a *unique name* and specified with a *parameter list* and some *bodies*. The parameters refer to objects, enum literals or primitive types where the type of a parameter can be explicitly defined. The bodies specify constraints over the parameters. A pattern may have multiple bodies with *constraints*, and may introduce additional local variables beside the parameters. If a variable is used only once it is specified as an anonymous variable with '_' as the first character in its name. A pattern with multiple bodies means a disjunction (`or`), thus a valid *match* necessitates that all the constraints are satisfied by a mapping of those variables for at least one body.

The following types of constraints are supported:

- **Classifier constraint:** checks if a variable is an instance of an EClass.
- **Path constraint:** requires a specific reference, an attribute, or a path of reference and attribute sequence between two variables.
- **Equality constraint:** specifies that two variables have to be mapped to the same model element.
- **Pattern call constraint:** enables the composition of multiple patterns. The positive pattern call refers to another pattern and specifies that the called pattern must be satisfied in the context of the actual parameters. Additionally, a pattern may be composed negatively (`neg` keyword), which means that the target negative pattern is disallowed to have a valid match along the actual parameters. Finally, it is possible to compute the transitive closure of a two-parameter pattern by the + symbol.
- **Check constraint:** evaluates a specific attribute expression on the variables of the pattern and accept matches only if the result of attribute condition is true. In this paper, the basic arithmetic and logic operators are covered.

It is possible to mark the patterns as an ill-formedness pattern with `@Constraint`, or make them define the values of derived features with the `@QueryBasedFeature` annotation.

By default, the result of a model query expressed as a graph pattern is the set of *all* matches with different values for the pattern parameter variables. However, by *binding* parameter variables to specific model elements or attribute values it is possible to filter the returned values. This allows the use of the same pattern for getting all possible matches and for checking whether a selected match is present in the result set.

2.3 Derived Features

Derived features (DF) are frequent extensions of metamodels to improve navigation by path compression or compute derived attributes. The value of a DF can be computed from other parts of the model as defined by a model query [46,41]. Such queries $\mathsf{df}(\mathsf{o},\mathsf{v})$ have two parameters: (i) for derived references o represents the source and v the target object of the reference while (ii) for derived attributes o represents the container object and v is the computed value the attribute.

A derived feature df defines the values of the selected feature in the following way: $\forall \mathsf{o},\mathsf{v} \in \mathsf{EObject} : \mathsf{feature}(\mathsf{o},\mathsf{v}) \Leftrightarrow \mathsf{df}(\mathsf{o},\mathsf{v})$. This has to be satisfied for each derived feature in the DSL which is defined by the DF rule, therefore the definition of a valid model is specified as: $\mathsf{M} \models \mathsf{META} \wedge \mathsf{DF}$. Model query frameworks like EMF-IncQuery automatically recalculate the value of the derived features in the instance model to satisfy DF [46].

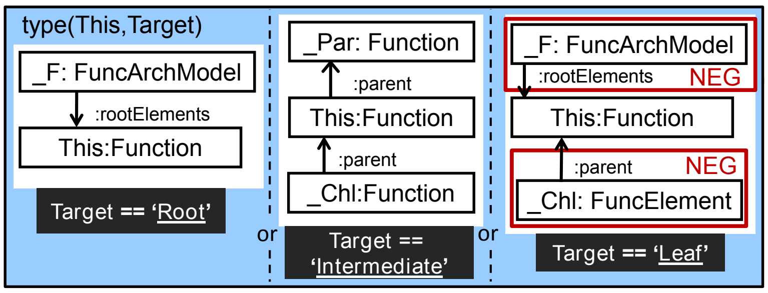
Our sample DSL contains two derived features highlighted in blue in Figure 2: a type which defines the value of an enum attribute and a model which points to the container FAM models.

Sample derived attribute The derived attribute type of Function is defined to take a value from the enumeration literals: Leaf, Root, Intermediate. The pattern defining the type attribute is illustrated on the right side of Figure 4. We use both a custom graphical and the textual EMF-IncQuery notation [11] to illustrate the queries defined for these derived features. In the graphical notation each rectangle is a variable with a declared type, e.g. the variable _Par is a Function, while arrows represent references of the given EReference between the variables, e.g. the function This has the function _Par as its parent. Negative application conditions (NACs) are illustrated as red rectangles. The OR pattern bodies represent that the matches of the query is the union of the matches of its or bodies.

Based on these definitions the type query has three OR pattern bodies each defining the value for the corresponding enum literal of the type attribute:

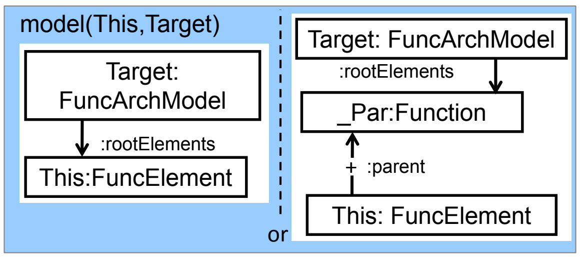
- Root if the container object is directly under the FunctionalArchitectureModel connected by rootElements.
- Leaf if the container object does not have a child along the subElements EReference and it is not a root element (as defined by the corresponding negative application conditions NEG).
- Intermediate if the container object has both parent and child functions.

Sample derived reference FunctionalElements are also augmented with a derived reference model (highlighted in blue in Figure 2) which represents a reference to the container FunctionalArchitctureModel object from any FunctionalElement within the containment hierarchy. The definition of the corresponding graph pattern



```
@QueryBasedFeature pattern type(
    This:Function , Target:FunctionType)=
{     // -- Root body --
  find rootElement(_Model,This);
  Target==FunctionType::Root;
} or {// -- Leaf body --
  neg find parent(_Child,This);
  neg find rootElement(_Model,This);
  Target==FunctionType::Leaf;
} or {// -- Intermediate body --
  find parent(This, _Par);
  find parent(_Child, This);
  Target==FunctionType::Intermediate; }
```

Fig. 4: Definition of derived attribute type



```
@QueryBasedFeature pattern model(
    This : FunctionalElement,
    Target : FunctionalArchitectureModel) =
{
  find parent+(This, Parent);
  find rootElements(Target, Parent);
} or {
  find rootElements(Target, This);
}
```

Fig. 5: Definition of derived reference model

is visible in Figure 5 which calculates the transitive closure of the parent reference between elements This and _Par as denoted by an arrow with a + symbol.

2.4 Well-formedness Constraints

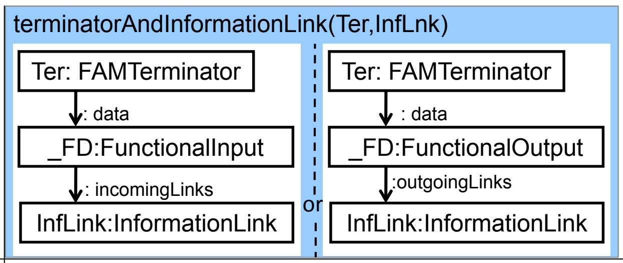
Structural well-formedness (WF) constraints (aka design rules or consistency rules) complement metamodels with additional restrictions that have to be satisfied by a valid instance model (in our case, functional architecture model). Such constraints can also be defined by query languages such as graph patterns or OCL invariants, in fact, our validation approach supports both of these formalisms. In many practical cases, well-formedness constraints are defined by queries which capture ill-formed model structures that are disallowed to have a match in a valid model. In the presence of a set WF of well-formedness constraints, a model M is called valid if $\mathsf{M} \models \mathsf{META} \wedge \mathsf{DF} \wedge \mathsf{WF}$.

In our running example, a WF constraint captures that a FunctionalData object with a FAMterminator cannot be connected to an InformationLink. It is specified by the terminatorAndInformationLink query (see Figure 6) that has two OR pattern bodies, one for the FunctionalInputs and one for the FunctionalOutputs with their corresponding incomingLinks and outgoingLinks, respectively.

The same constraint is also captured in OCL, see bottom part of Figure 6 for a comparison. Note that graph patterns are normally ill-formedness constraints to capture erroneous situations while OCL invariants capture the valid case, and violations are identified by their context. Another WF constraint specifies that the frequency of a subfunction has to be equal to (or exactly twice or four times as much as) the frequency of its parent function in order to enable communication. This WF constraint is specified in Figure 7.

3 Partial Snapshots

For a DSL validation scenario, we argue in this paper that traditional instance models are not sufficiently flexible to serve as direct inputs and outputs of the validation process. For instance, an abstract class is disallowed to have instances in a regular domain model, thus the model editor would prohibit the construction of such an instance. However, allowing the use of an instance of an abstract superclass can succinctly abbreviate many counterexamples retrieved by a solver. For example, retrieving a single instance of the abstract class FunctionalData represents every possible concrete subclasses (e.g. FunctionalInput or FunctionalOutput). Similarly, a small model fragment which highlights a validation problem may still violate other constraints thus it is not a valid instance model. Therefore, in the paper, we use a more permissive instance level formalism called *partial snapshots (PS)* as an additional input or output of DSL validation where certain language constraints are relaxed. During a typical validation run, such a PS eIQ



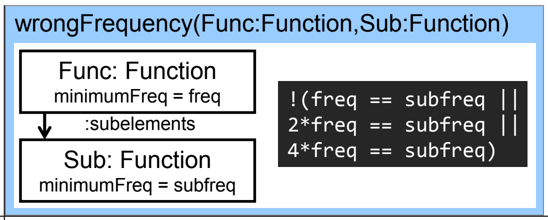
```
@Constraint pattern
  terminatorAndInformation(
    Ter : FAMTerminator,
    InfLink : InformationLink) =
{
FAMTerminator.data.incomingLinks(Ter,InfLink);
} or {
FAMTerminator.data.outgoingLinks(Ter,InfLink);
}
```

OCL

```
context InformationLink inv terminatorNoLink: (self.to <> null implies self.to.terminator = null) and
                                              (self.from <> null implies self.from.terminator = null)
```

Fig. 6: Definition of the WF constraints terminatorAndInformationLink and terminatorNoLink

eIQ



```
!(freq == subfreq ||
2*freq == subfreq ||
4*freq == subfreq)
```

```
@Constraint
pattern wrongFrequency(Func:Function,Sub:Function){
  Function.subElements(Func,Sub);
  Function.minimumFrequency(Func,freq);
  Function.minimumFrequency(Sub,subfreq);
  check(!(freq == subfreq ||
        2*freq == subfreq ||
        4*freq == subfreq));
}
```

OCL

```
context Function inv RightFrequency:Function.allInstances()->forAll(par,child|(child.parent=par)
  implies ((child.minimumFrequency = par.minimumFrequency) or
           (child.minimumFrequency = par.minimumFrequency * 2) or
           (child.minimumFrequency = par.minimumFrequency * 4)))
```

Fig. 7: Constraint for the frequency of the subfunctions

will be extended with further information to obtain a compliant instance model (called *completed model*).

While the underlying formalism of PSs is similar to existing approaches [31,35,50,52], we use them in a novel way to assist the DSL validation process, especially, for constructing validation proofs under specific assumptions. In our workflow, a PS can be generalised from a regular (fully specified) instance model by relaxing specific properties identified by the DSL developer, which can guide efficient validation in practically relevant cases. This allows the DSL developer to derive a PS from existing models developed in a native editor, or iteratively reuse the result of a previous validation run.

Definition 1 (Partial snapshot) A partial snapshot PS is an instance of a metamodel META when only the following (typing-specific) constraints are satisfied: CLS, GEN, REF, TC.

We briefly describe below which constraints are relaxed wrt. the definition of regular instance models.

1. Undefined attributes: In a normal EMF instance object each attribute has a value (or a preset default value), while a PS may contain attributes with undefined values.
2. Abstract objects: Partial snapshots allow to instantiate abstract EClasses. Such instances are handled similarly to regular objects thus they can have attributes and references. When completing a PS into a valid instance model, the type of an non-concrete object has to be refined in to a concrete subtype (including the additional attributes and references of the concrete type).
3. Unconnected partitions: While EMF requires that an instance model is arranged in a strict containment hierarchy, PSs allow to define instance models that consist of unconnected and consist of multiple model fragments. Such model fragment will be completed during a validation run by linking the partitions to a well-formed hierarchical containment tree.
4. Missing / extra edges: PSs may contain references without their inverse relation counterparts. Similarly, PSs may also violate multiplicity constraints.
5. Removed objects: Objects might be removed from a PS in order to exclude unimportant elements from a partial snapshot. In other words a PS defines only an initial model which can be extended with additional objects, therefore defining a minimally required structure. For example, if only the architecture of the functions is relevant in an instance model the communication channels can be removed. The analysis generates models which architecture contains the initial architecture of the PS.

Example 1 Figure 8 shows four PSs generalized from instance models by removing certain model elements.

– architecture: This PS defines a core structure of an IMA architecture prescribed to contain two of each

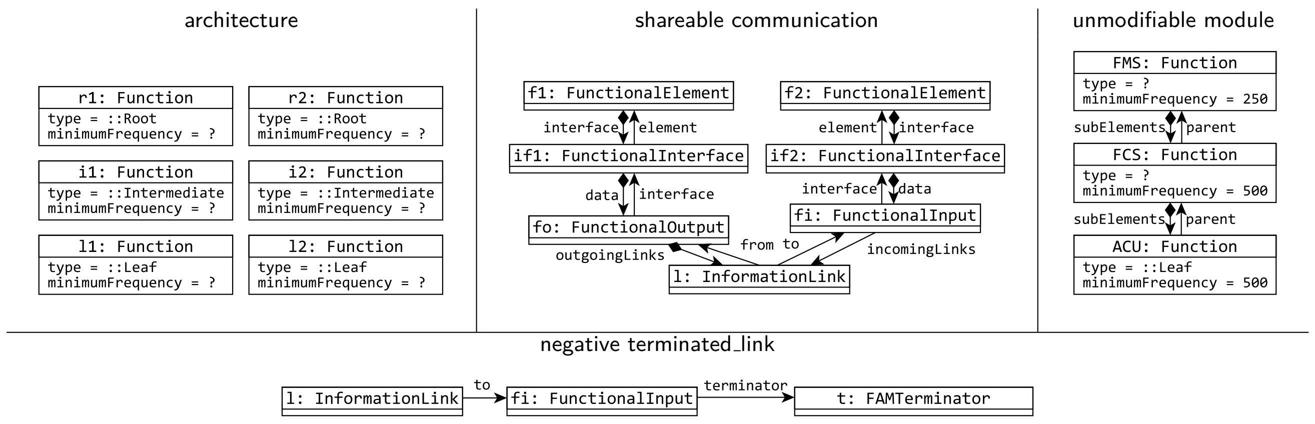


Fig. 8: Partial snapshots with semantic modifiers

root, intermediate and leaf elements, but it does not define their exact structure. As type attribute is a derived feature, the instance models that contain this PS must be arranged in an architecture which evaluates to the correct literals.

– communication: This PS contains a communication link from an input FunctionalElement to an output FunctionalElement via FunctionalInterfaces.
– module: This PS defines a module with three functions (FMS: Flight Management System, FCS: Flight Control System, ACU: Avionics Control Unit) arranged into a tree hierarchy via subelement and parent edges.
– terminated_link: This example shows a FunctionalInput extracted from an invalid model as it contains both a FAMTerminator and an InformationLink.

Multiple PSs will be passed as an input parameter to the validation process (see later in Section 4), and the solver will try to construct a valid instance model which satisfies each of them. However, there are multiple semantic modifiers for combining these PSs into a single valid completed model, which are discussed below (default modifier values are underlined).

1. <u>Positive</u> / Negative: A positive PS is an incomplete model that every output model has to contain as a submodel. Formally, a model M satisfies a positive snapshot POS if there is a match m along which POS can be matched as a query: $\mathsf{M} \models \mathsf{POS}(m)$ for some m.
   A negative PS has the opposite effect: if a model contains it as a submodel then it is invalid. Thus a model M satisfies a negative snapshot NEG if there is no match m along which NEG can be matched as a query: $\mathsf{M} \not\models \mathsf{NEG}(m)$ for all m.
2. <u>Injective</u> / Shareable: In case of an injective PS, the objects of the snapshot have to be mapped to different instance objects in the output model. Formally, $\mathsf{M} \models \mathsf{INJ}(m)$ when m is a query match with $\forall o_1, o_2 \in \mathsf{INJ} : \mathsf{INJ}(m/o_1) = \mathsf{INJ}(m/o_2) \implies o_1 = o_2$ (where $\mathsf{INJ}(m/o_i)$ denote the match of query element o_i in M).
   In a shareable PS, multiple PS elements can be mapped into the same model object if it satisfies the local conditions prescribed by each mapped PS element. Note, however, that each PS is evaluated independently from each other, thus different injective PSs may share objects in the output model.
3. <u>Modifiable</u> / Unmodifiable: A modifiable PS means that the reasoning process can add extra objects and links or fill empty (undefined) attributes in the output model to satisfy the PS.
   An unmodifiable PS means that the reasoning process can only change the context of the match of the PS (by adding/removing objects, links and setting attributes) but not in the match itself. For instance, if two objects in the PS are not linked by a certain relation, they need to remain unlinked in the result model. However, embedding the PS into a context with newly created incoming or outgoing relations (where their source or target model element is not in the match) is still allowed.

It is worth emphasizing that these semantic modifiers of PSs are independent from each other, i.e. they can be used in any combination for defining PSs for a DSL context. The grammar of Figure 9 summarizes the context specification by PSs (psspec).

Example 2 Figure 8 contains several semantic modifiers for the partial snapshots:

$$\begin{aligned}
\langle \mathit{psspec}\rangle &\rightarrow \langle \mathit{pslist}\rangle \\
\langle \mathit{pslist}\rangle &\rightarrow \langle \mathit{psitem}\rangle ; | \langle \mathit{pslist}\rangle | \langle \mathit{psitem}\rangle \\
\langle \mathit{psitem}\rangle &\rightarrow \mathsf{ps}\langle \mathit{modifier}\rangle\langle \mathit{psname}\rangle\{\langle \mathit{pscontent}\rangle\} \\
\langle \mathit{pscontent}\rangle &\rightarrow \langle \mathit{pscon}\rangle\langle \mathit{pscontent}\rangle | \langle \mathit{pscon}\rangle \\
\langle \mathit{modifier}\rangle &\rightarrow \langle \mathit{posneg}\rangle\langle \mathit{inject}\rangle\langle \mathit{unmod}\rangle \\
\langle \mathit{posneg}\rangle &\rightarrow \mathsf{positive}|\mathsf{negative}|\epsilon \\
\langle \mathit{inject}\rangle &\rightarrow \mathsf{injective}|\mathsf{shareable}|\epsilon \\
\langle \mathit{unmod}\rangle &\rightarrow \mathsf{modifiable}|\mathsf{unmodifiable}|\epsilon \\
\langle \mathit{pscon}\rangle &\rightarrow \mathsf{cls(o)}|\mathsf{rel(s,t)}|\mathsf{att(o,v)}
\end{aligned}$$

Fig. 9: A grammar of partial snapshots

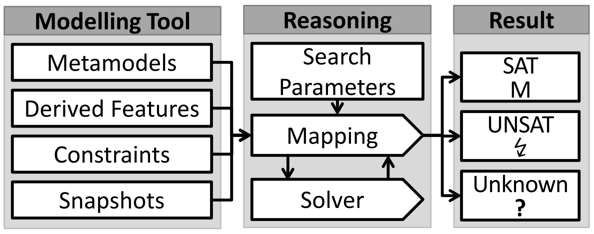


Fig. 10: Functional overview of the approach

– architecture: This PS can be embedded into an output model in accordance with modifiers positive, injective and modifiable, which is the default semantics: arbitrary extensions are allowed, but a TransIMA model needs to contain at least six Function objects (two of each type).
– communication: This PS is matched as positive, shareable and modifiable. This means that functional elements f1 and f2 can be matched to the same object if there is an information link with the same source and target functional element in the model.
– module: This PS has the semantics positive, injective and unmodifiable, thus no new relations can be added between FMS, FCS and ACU. Furthermore, the values of corresponding attributes (type and minimumFrequency) are not allowed to be altered. On the other hand, a new Function can still be added to the output model and linked to the matches of the PS elements by new edges.
– terminated_link: This PS should be treated as negative, injective and modifiable. A successful match of this PS invalidates the output model.

Metamodel elements, derived features, well-formedness constraints and partial snapshots of a DSL specification will be called *language properties* in the paper.

4 Overview of the Approach

This section provides a high-level, functional overview of our DSL validation approach using an SMT-solver. It gives the precise definition of the validation challenges for DSLs and describes how these challenges can be addressed by appropriate configuration of the solver.

4.1 Functional Overview of the Approach

Our approach aims to analyze the DSL specification of *modeling tools* by mapping them into first order logic (FOL) formulae that can be processed by advanced *reasoning applications* such as *SMT solvers* or *SAT solvers* (see Figure 10). The outcome of a reasoning problem is either *Satisfiable* or *Unsatisfiable*. If the problem is satisfiable, the solver constructs an output (or completed) model (which is interpreted as *Witness* or *Counterexample* depending on the validation task), while an unsatisfiable result means a *Contradiction*. Because certain validation tasks are undecidable in FOL it is also possible that validation terminates with an *Unknown* answer or a timeout. Each possible outcome will be illustrated on our case study.

The results of the reasoning need to be traced back and interpreted in modeling terms as attributes of the DSLs. Linking the independent reasoning tool to the modeling tool allows the DSL developer to make mathematically precise deductions over the developed languages and models including different validation techniques and example generations.

The validations are initiated and executed in well-defined *context*, which is treated as a set of axioms for the validation run. This context can be customized during DSL validation by selecting (or de-selecting) certain DSL artifacts from the following list. As a result, the output model M retrieved during DSL validation needs to respect the context.

– **Metamodels:** The set of domain classes allowed to be instantiated for constructing models can be restricted by explicitly selecting classes and structural features. By default, each class from each metamodel is used in the analysis. Then M has to satisfy the constraints of these this (possibly partial) metamodel: $\mathsf{M} \models \mathsf{META}$.
– **Derived Features:** The values of the derived features have to be correctly evaluated with respect to their definition yielding unique and complete results (denoted as $\mathsf{M} \models \mathsf{DF}$).
– **Constraints:** The output model has to satisfy the selected well-formedness constraints: $\mathsf{M} \models \mathsf{WF}$, thus certain constraints can be relaxed or strengthened for a reasoning process.
– **Partial Snapshots:** The output model M has to combine partial snapshots according to their semantic parameters discussed in Section 3, denoted as $\mathsf{M} \models \mathsf{PS}$. Partial snapshots act as explicit assump-

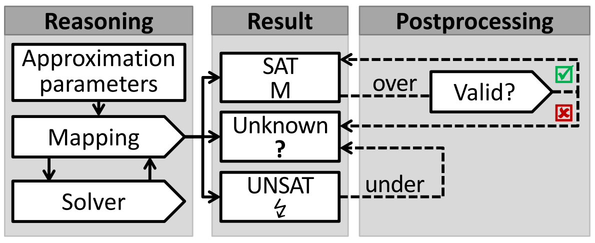


Fig. 11: Reasoning with approximated constraints

tions (or proof obligations) in a validation scenario, so by default, no PS is passed to the solver.

- **Search Parameters:** Additionally, the user may define some reasoning-specific input parameters:
  - **Size:** The number of objects used in the construction of an output model can be restricted by a positive integer (defined by $|\mathsf{M}| \leq \mathsf{size}$). By default, $\mathsf{size} = *$, which means that the analysis covers all each possible model regardless of its size.
  - **Approximation level:** Some DSL property (such as the acyclicity of the containment hierarchy) cannot be represented in FOL. The method is customizable with the level of approximation (see Section 6.2), which allows to set the limit of approximation level. Higher approximation level will reduce the possibility of false positives.

The constraints serving as the context of DSL validation are summarized as $\mathsf{DSL} = \mathsf{META} \wedge \mathsf{DF} \wedge \mathsf{WF} \wedge \mathsf{PS}$, and it defines a possibly infinite set of $\mathsf{Models} = \{\mathsf{M} : \mathsf{M} \models \mathsf{DSL}, |\mathsf{M}| \leq \mathsf{size}\}$. If each parameter is set to the default value, the analysis covers the full range of valid instance models (thus the full language is analyzed).

The constraints in DSL may contain expressions which cannot be processed or effectively handled by the underlying solver thus approximation techniques have to be applied. The use of approximations is an integral part of the proposed approach, which is handled in validation process as Figure 11 illustrates it. Based on the *Approximation Parameters*, under- and overapproximations are applied on the logic problems during the transformation to create stronger or weaker conditions by adding, removing or modifying formulae. The modified problem is expected to be solved more efficiently by the target logic reasoner, and the result can be *Satisfiable* with a model (which is only a *Candidate Model* of the original problem), *Unsatisfiable* or *Unknown*. Because of the approximations, an output of this modified problem needs some additional analysis:

- If an **underapproximated** problem is **satisfiable**, then the original problem is **satisfiable** too, and the candidate model (output of the modified problem) is acceptable for the original problem.
- If an **overapproximated** problem is **unsatisfiable**, then the original problem is **unsatisfiable** too.
- But if an **underapproximated** problem is **unsatisfiable**, then it is uncertain if the original problem has contradictions so the result is **unknown**.
- If an **overapproximated** problem is **satisfiable**, then the candidate model might be a *false positive* which does not satisfy the original problem, so additional validation with the original constraints is necessary. Structural correctness, OCL and EMF-IncQuery constraints can be easily checked on a candidate instance model by dedicated language level validation tools (like EMF-IncQueryor OCL interpreters). If the validation is successful, then the candidate model is valid in the context of the original problem, otherwise the process fails with **unknown**.
- If the theorem prover provides a model, which is formally correct, but does not occur in real scenarios, then it is a **spurious counterexample**. To handle those irrelevant cases, the counterexample is turned into a partial snapshot supplied to the solver in consecutive validation runs.

By using abstraction in the validation process complex language elements can handled in the validation even if they cannot directly handled by the solver.

4.2 Validation tasks

Figure 12 shows a more detailed overview of the different DSL validation tasks, their respective input parameters (upper part) and the possible validation outputs (lower part) of our framework.

The input parameters allow to define the DSL validation context (as discussed above), and the selected elements of the DSL context are mapped to FOL in accordance with further reasoning-specific search parameters. We distinguish between five DSL validation tasks (consistency, subsumption, equivalence, completeness and ambiguity checks), which are detailed below. In each case, the validation run terminates with constructing an output model, or revealing a contradiction. The result of the model generation is interpreted for each validation task, the valid and invalid outcomes are highlighted in Figure 12. If an output model is retrieved it is presented as a witness model (for consistency and subsumption check), or as a counterexample (e.g. a proof of ambiguity or incompleteness).

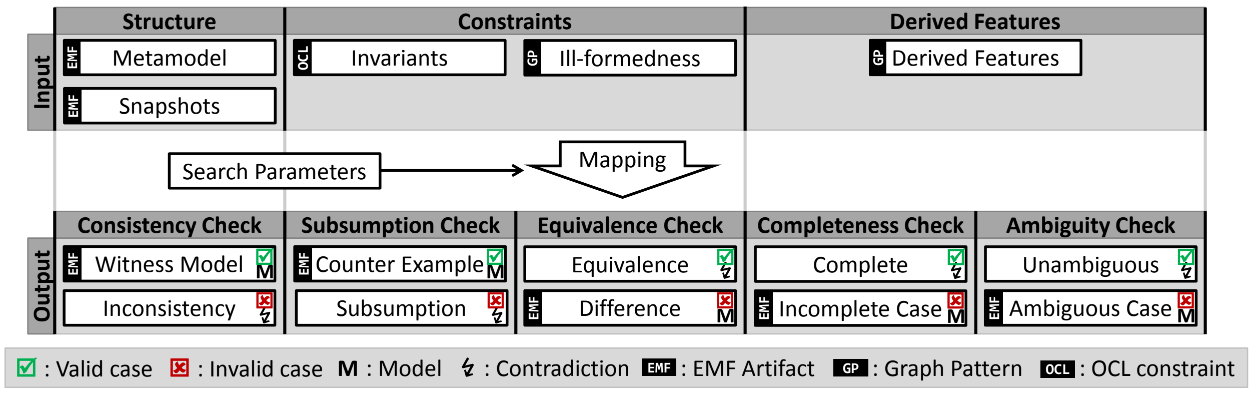


Fig. 12: Overview of validation tasks

4.2.1 Consistency check

Consistency is a property of the whole DSL which means that there is no any contradiction in its specification, i.e. there is at least one valid instance model. A consistency check either reveals the conflicting elements (constraints, derived features) of a DSL or proves that the DSL is consistent. Because any statement can be proven from a contradicting set of axioms, an inconsistency invalidates the result of any further DSL checks (like completeness, ambiguity and subsumption and equivalence checks). A consistency check aims to prevent such a situation.

Definition 2 (Consistency) A DSL context is **consistent** if it has a valid instance model, i.e. $\mathsf{Models} \neq \emptyset$. A DSL context is **inconsistent** if it is not consistent i.e. $\mathsf{Models} = \emptyset$ (where $\mathsf{Models} = \{\mathsf{M} : \mathsf{M} \models \mathsf{DSL}\}$.

Note that there are no further restrictions on the size of M to serve as a proof of consistency, i.e. M might be as simple as a single object. In fact, many SAT-solvers would retrieve such a model by default. Therefore, for practical DSL validation, stricter consistency requirements are necessitated, such as every class and reference in the metamodel have to be instantiated at least once in an output model.

Such consistency criteria can be encoded and checked in our framework using partial snapshots. The use of partial snapshots and flexible model size limit (and further search parameters) make the generation of output model highly customizable. The underlying solver is called with the metamodels, the derived features, the constraints and partial snapshots as input. The output model is interpreted as a witness model of consistency, while a contradiction is a proof of inconsistency. In practice, consistency checks are typically used to (1) identify contradictions in a DSL specification, (2) check if each language element can be instantiated or (3) generate instance models for a given context (e.g. relevant contexts for test cases).

4.2.2 Subsumption check

A complex DSL may contain a large number of independent language properties (well-formedness constraints, derived features, partial snapshots). With a growing number of language properties (i.e. DSL context), a redundant property is difficult to be identified merely by human inspection. While consistency checks reveal a contradicting language specification, it would be advantageous to reveal if a new constraint really imposes further restrictions on valid instances, or it is already covered by the existing DSL specification. Subsumption checks of a language property aims to detect this latter case. A subsumed constraint does not express any additional restriction over the DSL therefore it can be removed without any further consequences.

Definition 3 (Subsumption) A property P is **subsumed** by a DSL context if $\mathsf{DSL} \models \mathsf{P}$. A P property is **not subsumed** by a DSL context if $\mathsf{DSL} \not\models \mathsf{P}$. A P property is **independent** from a DSL context if $\mathsf{DSL} \not\models \mathsf{P} \wedge \mathsf{DSL} \not\models \neg\mathsf{P}$.

Informally, property P is subsumed by a DSL specification when every model that satisfies the DSL specification will also satisfy this property P (formally, $\forall \mathsf{M} \in \mathsf{Models} : \mathsf{M} \models \mathsf{DSL} \Rightarrow \mathsf{M} \models \mathsf{P}$). If property P is not subsumed then there is a valid instance model that satisfies the DSL specification but not the property itself (formally, $\exists \mathsf{M} \in \mathsf{Models} : \mathsf{M} \models \mathsf{DSL} \wedge \mathsf{M} \not\models \mathsf{P}$). Finally, the property is independent from a DSL context if it is consistent with it but not subsumed by it.

Given the DSL context as a set of axioms, we carry out traditional theorem proving to decide if property P is derivable from the set of axioms. For this purpose, we aim to prove that adding $\neg\mathsf{P}$ as an axiom to

DSL makes the new specification $\{\mathsf{DSL}, \neg\mathsf{P}\}$ inconsistent. Therefore, an output model retrieved by the solver is interpreted as a *counterexample* to testify that property P is not subsumed. In order to show the independence of a property P, the axiom set $\{\mathsf{DSL}, \mathsf{P}\}$ is aimed to be refuted as well, and we require both validation runs to retrieve an output model.

4.2.3 Equivalence check

Language properties can be defined in multiple ways, potentially using different languages and formalisms. A well-formedness constraint or a derived feature can be captured as a graph pattern, as an OCL invariant or as positive or negative partial snapshots. In practical scenarios, a DSL specification in fact uses a mixture of such techniques. Moreover, automated transformations are defined to convert OCL constraints into graph patterns and vice versa. Equivalence checks aim to prove the correctness of conversions between the language properties in a DSL.

Definition 4 (Equivalence) Properties A and B are equivalent in a context DSL if $\mathsf{DSL} \models \mathsf{A} \Leftrightarrow \mathsf{DSL} \models B$.

According to the definition, two validation runs are initiated for checking the consistency of sets $\{\mathsf{DSL}, \neg\mathsf{A}\}$ and $\{\mathsf{DSL}, \neg\mathsf{B}\}$, respectively. The result of the validation is the proof of the equivalence, or an example to highlight the semantic difference between the two property, which presents an example where one property is satisfied, but the other is not.

4.2.4 Completeness and Ambiguity Check of DFs

Derived features specified by model queries (defined by graph patterns or OCL constraints) are integral parts of a DSL specification. However, the definition of such DFs is error prone as the corresponding query has to yield well-defined result in all situations. In the paper, we aim to check the completeness and unambiguity of derived features.

Completeness of a derived feature means that the DF satisfies the lower multiplicity constraint of the target structural feature. For example, in case of a reference with 1..? multiplicity, completeness is achieved if the DF evaluates to at least one value for every occurrence of the derived feature. If there is a model where no values can be assigned to an occurrence of the derived feature it is incomplete.

Let $\mathbf{MULT}_{\mathsf{SF}}^{min}$ denote the lower multiplicity and $\mathbf{MULT}_{\mathsf{SF}}^{max}$ denote the upper multiplicity of the structural feature SF by appropriate constraints. Furthermore, let $\mathsf{DSL} \setminus \mathsf{C}$ denote a DSL context where the language constraint C is removed. Completeness is then defined as follows:

Definition 5 (Completeness of Derived Features) A derived feature DF is **complete** in a DSL context if $\mathsf{DSL} \setminus \mathbf{MULT}_{\mathsf{DF}}^{min} \models \mathbf{MULT}_{\mathsf{DF}}^{min}$. Otherwise, DF is **incomplete**.

Ambiguity allows the upper limit of the multiplicity to be exceeded. For example, a DF complies with the ?..1 multiplicity if it evaluates to at most one value for every occurrence of the DF. An output model where multiple values can be assigned to some occurrence of the DF means that the DF is ambiguous.

Definition 6 (Unambiguity of Derived Features) A derived feature DF is **unambiguous** in a DSL context if $\mathsf{DSL} \setminus \mathbf{MULT}_{\mathsf{DF}}^{max} \models \mathbf{MULT}_{\mathsf{DF}}^{max}$. Otherwise DF is **ambiguous**.

In accordance with the definitions above, the corresponding multiplicity constraints are extracted from the DSL context and their negation is added back and checked for contradiction. The result of the validation task could be either the proof of completeness / unambiguity of the checked DF with respect to the validation context, or a counterexample that, although satisfies the specification of the DF, violates its multiplicity.

5 A Case Study on DSL Validation

5.1 Overview of DSL Validation Workflow

Complex DSL specifications may contain multiple inconsistencies, and the erroneous language properties are difficult to identify and localize. To assist the developers finding such inconsistencies, we propose an iterative workflow that defines the practical order of addressing and completing the different validation steps. By following this workflow, our framework will reveal the design flaws one by one so with the help of the counterexamples the root cause of the error can be better detected. This iterative approach can be applied in any stage of DSL development (including also on incomplete language specifications) thus specification errors can be detected in an early phase of DSL design. Additionally, the workflow can guide the developer through a complete language validation process. The validation workflow is illustrated in Figure 13.

1 First, a metamodel is added to the validation context process and checked for consistency.
2 Derived features are iteratively added (extending it with one new DF at a time)

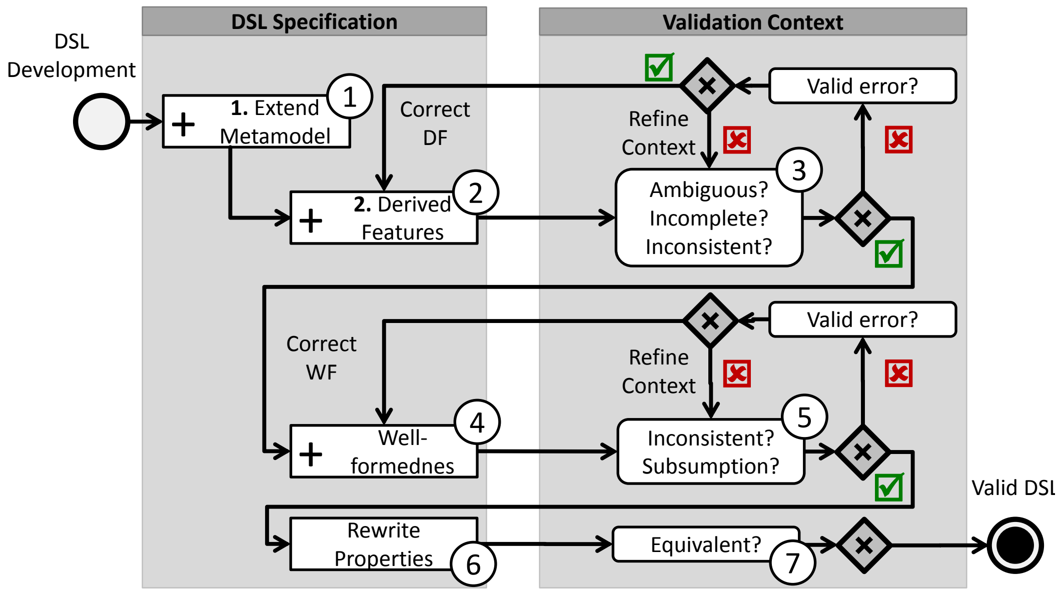


Fig. 13: DSL validation workflow

3 The unambiguity and completeness of the DF as well as the consistency of the entire validation context are automatically checked.
4 After checking all DFs, validation continues with the WF constraints, which are added to the validation context iteratively (one-by-one).
5 Our framework inspects whether the current WF constraint causes inconsistency or it is already subsumed by the current validation context.
6 If the validation of the DSL succeeds with all DF and WF constraints included in the context, then the DSL is valid under the assumptions imposed by the search parameters and the partial snapshots. The DSL validation process succeeds.
7 Partial snapshots retrieved in the validation context can be turned into WF constraints of the DSL. In such a case, our framework formally proves that the WF constraint and the PS in the context are formally equivalent, therefore, the corresponding WF is not required to be re-validated.
ER If the validation fails at any step, the language engineer has to correct the DSL artifact based on the counterexample, and continue the validation from the modified element. In case of *false positives* (which are detected by checking the result model whether it truly satisfies the constraints), the parametrization of the search needs to be fine-tuned. If the result is an error but the framework provide a *spurious counterexample*, then the validation context should be extended by the missing constraints.

Starting the validation of derived features prior to WF constraints is based on the observation that each DF eliminates a large set of trivial, non-conforming instance models (which are not valid instances of the DSL). Moreover, adding a single constraint at a time to the validation problem helps identify the location of the problem, because the solver provides only very restricted traceability information. This eases the refinement in case of an erroneous DF or WF is added in the actual step based on the proof provided by the solver.

The rest of this section demonstrates how this DSL validation workflow can be applied on the running example from the avionics domain.

5.2 Derived Type Validation

For illustration purposes, we artificially inject two conceptual flaws into the query defining the derived feature type in the IMA example (depicted in Figure 14, and see Figure 4 for its correct original definition):

1 The pattern body representing the intermediate case has been removed, which makes the DF incomplete.
2 The constraint defines that the leaf elements cannot be connected by rootElements reference is also removed. This will lead to an ambiguity as the body representing the leaf case becomes more permissive.

The validation process is presented in Figure 15.

– First (**Step 1**) we add the type DF to the DSL context and its consistency is successfully validated.

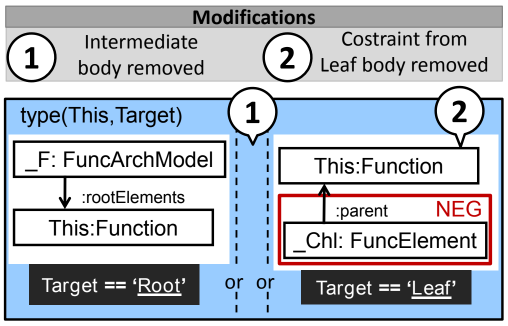
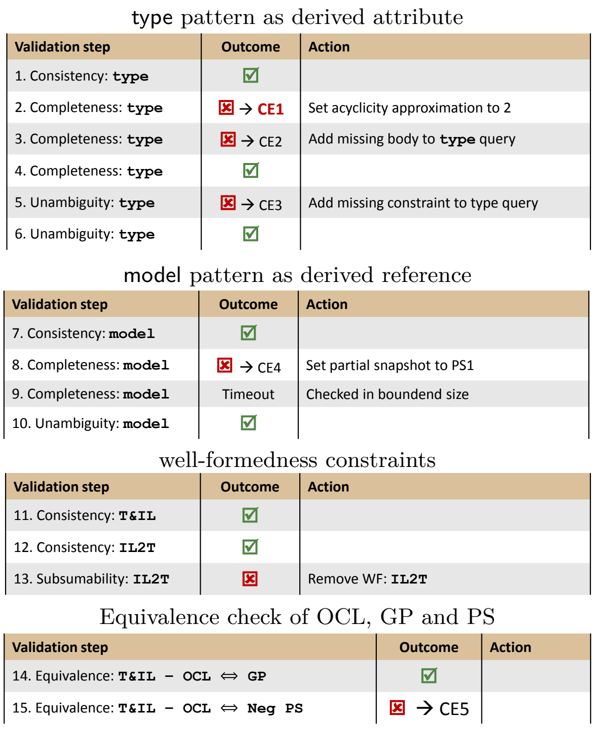


Fig. 14: Modifications on the type pattern



type pattern as derived attribute

Validation step	Outcome	Action
1. Consistency: **type**	☑	
2. Completeness: **type**	☒ → CE1	Set acyclicity approximation to 2
3. Completeness: **type**	☒ → CE2	Add missing body to **type** query
4. Completeness: **type**	☑	
5. Unambiguity: **type**	☒ → CE3	Add missing constraint to type query
6. Unambiguity: **type**	☑	

model pattern as derived reference

Validation step	Outcome	Action
7. Consistency: **model**	☑	
8. Completeness: **model**	☒ → CE4	Set partial snapshot to PS1
9. Completeness: **model**	Timeout	Checked in boundend size
10. Unambiguity: **model**	☑	

well-formedness constraints

Validation step	Outcome	Action
11. Consistency: **T&IL**	☑	
12. Consistency: **IL2T**	☑	
13. Subsumability: **IL2T**	☒	Remove WF: **IL2T**

Equivalence check of OCL, GP and PS

Validation step	Outcome	Action
14. Equivalence: **T&IL – OCL ⇔ GP**	☑	
15. Equivalence: **T&IL – OCL ⇔ Neg PS**	☒ → CE5	

Fig. 15: Example DSL validation run

– Then (**Step 2**), the completeness of the derived feature type is checked resulting in a failure illustrated by the Counterexample 1 showing three functions without type to form a cycle in the containment hierarchy. This counter example is visualized in Figure 16 where the invalid elements are highlighted in red, and the containment references are represented with black diamonds. Note that almost all properties of the instance model are correct, only the containment hierarchy is violated (along the n1-n3-n4 cycle). This is a false positive case since the acyclicity of the containment hierarchy can only be approximated in first order logic. In our framework, this problem can be easily solved by simply raising the level of approximation for transitive acyclicity .
– In **Step 3**, our tool shows a real counterexample (middle part of Figure 16) where the intermediate function n4 does not have type attribute. This is fixed by adding (back) the second pattern body with the Intermediate definition to the type pattern.
– After correcting it, the validation is successfully executed in **Step 4**.
– Then the ambiguity of attribute type is checked (**Step 5**), which fails again with a single function that is both a Leaf and a Root. This counterexample is also visible in the right part of Figure 16.
– This issue is fixed by adding the missing NAC condition on the rootElements to the third pattern body of type in **Step 6**.

5.3 Derived Reference Validation

Now the validation process (see Figure 15) of DF model is presented that defines a reference to the container FunctionalArchitctureModel from a FunctionalElement.

– **Step 7** adds the model DF to the specification, and the consistency check is executed successfully.
– Then in **Step 8**, completeness validation fails as pointed out in Counterexample 4 in Figure 17 since a model with a single Function element does not have anything to refer to with the model link. This result represents a spurious counterexample, because Functions are only used in the context of a FunctionalArchitectureModel. For this purpose, a partial snapshot is defined with a FunctionalArchitectureModel object to prune the search space and avoid such counterexamples (Figure 17).
– However, its revalidation (**Step 9**) ends in a Timeout (more than 2 minutes) and thus this feature can only be validated on a concrete bounded domain of a maximum of 5 model objects.
– Finally in **Step 10**, the unambiguity of the model DF is validated without a problem.

5.4 Validation of Well-Formedness Contraints

To demonstrate the subsumption check, another WF constraint is added to the DSL specification expressed by the InformationAndTermintator query (Figure 18, bottom part), which prohibits that an InformationLink is connected to a FAMTerminator. This constraint only differs from the first body of the original WF constraint (see Figure 6) in that it uses the inverse edges, thus it is redundant.

The validation process of WF constraints is illustrated in Figure 15.

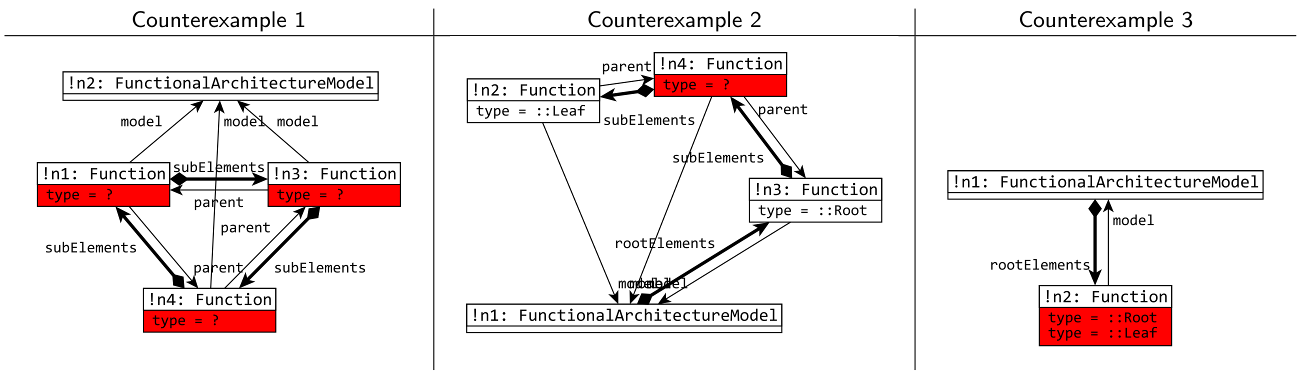


Fig. 16: Counterexamples of the type validation

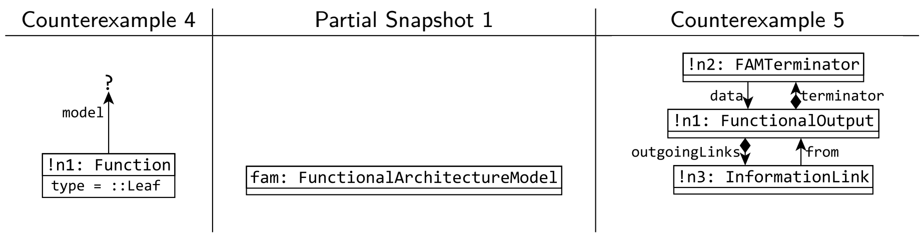
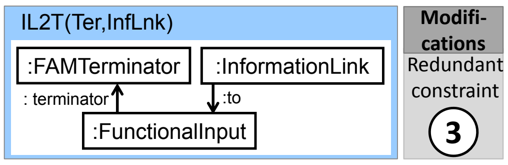


Fig. 17: Counterexample and partial snapshot for validating model DF, and counterexample 5 for failed equivalence check



```
@Constraint pattern InformationAndTermintator(
 T : FAMTerminator, I : InformationLink) =
{ InformationLink.to.terminator(I, T); }
```

Fig. 18: Definition of the redundant InformationAndTermintator pattern

– At first, the consistency validation of the WF constraint terminatorAndInformationLink (**Step 11**) is executed with a success.
– Then the redundant InformationAndTermintator is added which remains consistent (**Step 12**).
– Finally, the last constraint is checked for subsumption (**Step 13**) and found positive. Thus it is already covered by the DSL specification, therefore it can be deleted from the set of WF constraints.

5.5 Equivalence Check

We selected two use cases to demonstrate the equivalence check in DSL validation in Figure 15.

– First (**Step 14**), we used our framework to show the equivalence of the WF constraint defined by the terminatorAndInformation graph pattern and terminatorNoLink OCL invariant (see Figure 6). There was no valid instance model found, that violates only one of these two constraints, so the equivalence of the two different representations is successfully proved, thus they can replace each other in the DSL.
– Then in **Step 15**, we tried to check the equivalence of the terminatorNoLink (see Figure 6) OCL invariant and terminated_link negative PS (see Figure 8). Our framework returned with the Counterexample 5 (see in Figure 17), which highlights the semantic difference between the two constraints. The PS cannot be matched on this counterexample and since it is a negative PS, the constraint is not violated. On the other hand, the second part of the OCL constraint is violated, because there is a FAMTerminator connected to an InformationLink through a FunctionalOutput. This counterexample proves the inequality of this DSL property.

5.6 Model Generation for Partial Snapshots

Our framework can also be used for generating instance models satisfying a DSL specification under certain assumptions (partial snapshots). This is, in fact, very useful in test generation [40] or quick fix generation purposes. Below we only briefly demonstrate how to derive a valid model of minimal size (see Figure 19) which contains all PSs from Figure 8.

– Partial snapshot architecture is satisfied in the output model by objects o2, o3, o5, o6, o7, o8.
– Partial snapshot shareable communication is satisfied along objects o1, o4, o8, o9, o10, (and their corresponding links). Note that Function o8 and FunctionalInterface o4 is shared for the source and target end of the InformationLink.
– Partial snapshot unmodifiable module can be transformed to objects o5, o7, o8.

Our example demonstrates that a single object in the output model can satisfy multiple roles in different PSs. Furthermore, the same object can be shared when matching shareable PSs.

6 Transforming DSLs to FOL Formulae

We now discuss the details of transforming DSL artifacts to first order logic formulae to be processed by SMT solvers. Due to its excessive length, the details of the transformation is split into three sections, and we first present some theoretical foundations and the detailed mapping of metamodels and partial snapshots in Section 7 followed by transformation of the constraint languages into FOL in Section 8.

6.1 Foundations of the Transformation

The transformation takes a DSL context as input to create a set of axioms called DSL_F as output (where F denotes that this is a formalised description), which is satisfiable if and only if the original DSL context was consistent. If the DSL_F is satisfiable then by definition there is an interpretation M_F that satisfies DSL_F. Additionally, we back-annotate the logic structures M_F derived as a result of the validation process to an actual instance model (or partial snapshot) of the DSL, formally:

if $\mathit{forward}(\mathsf{DSL}) = DSL_F$ and $\mathit{back}(M_F) = \mathsf{M}$ then

$$\begin{array}{ll} 1. & M_F \models DSL_F \Leftrightarrow \mathsf{M} \models \mathsf{DSL} \\ 2. & DSL_F \text{ unsatisfiable} \Leftrightarrow \mathsf{DSL} \text{ inconsistent} \end{array} \quad (1)$$

Function *forward* consists of different transformation steps each of which takes certain DSL artifacts and yields a set of corresponding logic axioms:

– The metamodel transformation creates the formulae $META_F$ from the metamodel that maps the structural features of the DSL (detailed in Section 7.1).
– The instance model transformation derives the formulae PS_F from the partial snapshots (discussed in Section 7.2).
– The pattern transformation creates a set of formulae GP_F from the definitions of the graph patterns and links them to their corresponding well-formedness constraints WF_F or derived feature formulae DF_F (explained in Section 8.1).
– Finally, OCL transformation creates formulae OCL_F for OCL constraints (Section 8.2).

So the transformation of DSL into FOL is partitioned in the following way:

$$DSL_F = META_F \wedge WF_F \wedge GP_F \wedge DF_F \wedge PS_F \wedge OCL_F \quad (2)$$

The Search Parameters can further customize the transformation process, their effect will be detailed in each transformation step. Our tool can execute different reasoning tasks on DSL_F. The Validation task transformation prepares DSL_F to create the actual input for the validation run of the reasoning tool based on the method described in Section 4.2. The reasoning tool then carries out the satisfiability check on the input formulae to decide $DSL_F \models P$ and interprets the result with respect to the actual validation task.

$$\begin{array}{c} DSL_F \cup \{\neg P\} \text{is unsatisfiable} \Rightarrow DSL_F \models P \\ \exists M[DSL_F \cup \{\neg P\} \models M] \Rightarrow DSL_F \not\models P, \\ \text{and } M \text{ is a counterexample} \end{array} \quad (3)$$

If the reasoning tool finds the axiom system satisfiable an example interpretation will be created that explicitly defines a (symbolic) value for every uninterpreted features of the axiom system (e.g. how many objects are there in the model, which ones are linked with a reference or what are the matches of the graph patterns). By querying the metamodel specific attributes of this logic model an EMF instance model will be created for back-annotation purposes.

6.2 Approximation techniques

SMT solvers can use a combination of multiple background theories, therefore they can effectively reason over a certain set of logic problems [26]. Our choice of background theory is effectively propositional logic (EPR)[44] as its provides logical formulae that can cover

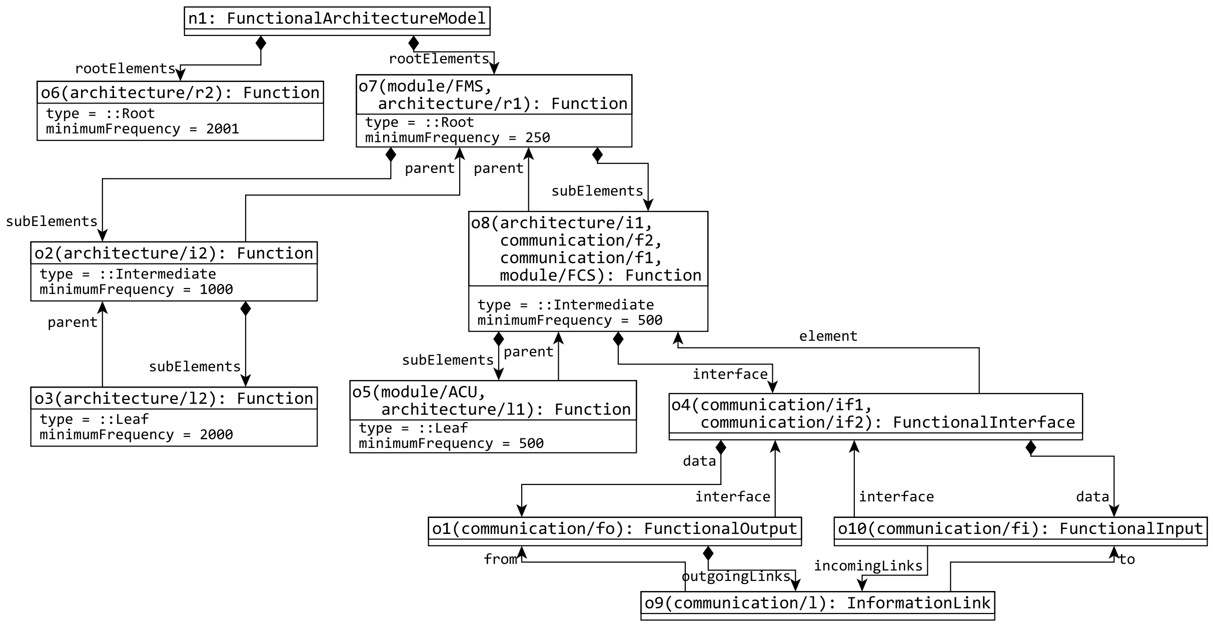

n1: FunctionalArchitectureModel
rootElements
rootElements
o6(architecture/r2): Function
type = ::Root
minimumFrequency = 2001
o7(module/FMS, architecture/r1): Function
type = ::Root
minimumFrequency = 250
parent
parent
subElements
subElements
o2(architecture/i2): Function
type = ::Intermediate
minimumFrequency = 1000
o8(architecture/i1, communication/f2, communication/f1, module/FCS): Function
type = ::Intermediate
minimumFrequency = 500
parent
subElements
element
subElements
parent
interface
o3(architecture/l2): Function
type = ::Leaf
minimumFrequency = 2000
o5(module/ACU, architecture/l1): Function
type = ::Leaf
minimumFrequency = 500
o4(communication/if1, communication/if2): FunctionalInterface
data
interface
interface
data
o1(communication/fo): FunctionalOutput
o10(communication/fi): FunctionalInput
from
outgoingLinks
incomingLinks
to
o9(communication/l): InformationLink

Fig. 19: Screenshot of a valid model satisfying all partial snapshots of Figure 8

the large set of DSL language features yet provide efficient reasoning capabilities.

Definition 7 (Effectively propositional logic) Effectively propositional logic is a fragment of first order logic with formulas of the form

$$\exists e_1, \ldots, e_n \forall a_1, \ldots, a_m : \varphi(e_1, \ldots, e_n, a_1, \ldots, a_m)$$

where φ is a quantifier free, and atomic subformulas range over uninterpreted relations.

However, graph patterns in EMF-IncQuery or the OCL language are more expressive than FOL. Therefore, certain constraints such as recursive patterns, transitive closures, set cardinalities and check expressions cannot be compiled into FOL. The expressiveness of the different constraint languages and logic fragments can be summarized as follows: EPR < FOL < GP, OCL.

To represent problems in the designated logic fragment, we define approximation techniques for predicates:

Definition 8 (Approximations of Predicates) Predicate P^U **underapproximates** (similarly P^O **overapproximates**) constraint P if it satisfies the following implications for every evaluation: $P^U \Rightarrow P, P \Rightarrow P^O$

As a trivial example, constant *true* predicate is always a good overapproximation, and $false$ underapproximates every predicate. A formula also approximates itself. So the strategy of our mapping is to express most of formulae in the target designated logic fragment language, and approximate the inexpressible features.

An axiom system (set of axioms) can be also approximated if every axiom is approximated in it. If a property P is implied by an underapproximation of a DSL context, then P is derivable from original DSL context as well. Similarly, if P is not derivable from an overapproximation of a DSL context, then it is not derivable from the original DSL context.

$$\begin{aligned} &DSL_F^U \models P \Rightarrow DSL_F \models P \\ &DSL_F^O \not\models P \Rightarrow DSL_F \not\models P \end{aligned} \tag{4}$$

This allows the validation properties of the DSL_F by proving the same properties on its under- or overapproximations.

$$\begin{array}{rcl} DSL_F^U \text{ satisfiable} & \Rightarrow & DSL_F \text{ satisfiable} \\ DSL_F^O \text{ unsatisfiable} & \Rightarrow & DSL_F \text{ unsatisfiable} \end{array} \tag{5}$$

Example 3 Using approximations allows to carry out DSL validation by using a more restrictive logic fragment which allows more efficient reasoning. For example, let us provide an approximation of the containment tree hierarchy, which needs to satisfy the following properties (as described in details in Section 7.1):

– Every object (other than the root element) is contained by another. (This is expressible in FOL but not in EPR.)

– The containment hierarchy is acyclic. (This is not expressible in FOL.)

To express containment hierarchy in FOL, the second rule can be overapproximated, e.g. as follows:

– The containment graph is free from cycles with length of at most 5. (This is FOL and also EPR.)

To express the containment hierarchy in EPR the first rule has to be omitted. By doing this, the reasoning tasks can be efficiently executed on a problem in EPR, and if the reasoning tool finds the DSL with more general containment rules unsatisfiable then the original problem has to be unsatisfiable too. The negative side effect is that the tool may provide false positives as well.

Further approximations can be introduced for numeric types to restrict their range to a specific interval. For instance, if we can derive (e.g. from domain-specific knowledge) that a value of an integer attribute is between 0 and 100 then this may help the reasoner to carry out analysis more efficiently. Similar approximations can be used for handling multiplicities.

7 Transforming Metamodels and Partial Snapshots

7.1 Metamodel Transformation

We first discuss the transformation of metamodels into FOL. Table 1 summarises the transformed features of the metamodel. It also presents which property is expressible in FOL or EPR.

Features of the metamodel	**SMT**		**SAT**
Unlimited # of EObjects	E	+	X
EClasses	E	+	E
Class hierarchy	E	+	E
EEnums	E	+	E
EReferences	E	+	E
EAttributes	E	+	E
Numbers	E	−	A
Multiplicity upper bound	E	+	E
Multiplicity lower bound	E	−	E
Inverse edges	E	+	E
Containment hierarchy	A	−	E

E: Expressible A: Approximable X: Inexpressible
+: in EPR −: not in EPR

Table 1: Expressing metamodel features in FOL

7.1.1 Objects

The graph representation of EMF models is transformed to unary predicates (for nodes), binary predicates (for relations) and functions (for regular attributes). EObjects are uniformly mapped to a dedicated type *Object*. If the number of objects (of a specific type) is bounded then the type is defined with a fix range of values, such as $Object = \{o_1, o_2, o_3\}$ for instance, where literals o_1, o_2 and o_3 are representing the objects.

7.1.2 Classes

Every class in a metamodel is transformed to unary characteristic predicate: if an object is an instance of a class then the predicate evaluates to true, otherwise it is false.

EMF	C ∈ EClass
FOL	$type^{\mathsf{C}} : Object \to \{true, false\}$

For example, class Function is transformed to $type^{\mathsf{Function}} : Object \to \{true, false\}$.

7.1.3 Type hierarchy

In many cases, an object is an instance of multiple classes due to the generalization relation between the classes, and the existence of abstract classes which do not have direct instances. A simple way to represent the type hierarchy is using a table where the columns represent the possible classes and the rows the concrete (non-abstract) classes. A cell represents a literal whether the type in the row is compatible with the type in the column.

$$\forall o \in Object : \bigvee_{\substack{\mathsf{A} \in \mathsf{EClass} \\ \neg \mathsf{isAbstract(A)}}} \bigwedge_{\mathsf{B} \in \mathsf{EClass}} type^{\mathsf{cls}}(o) \Leftrightarrow \mathsf{super}(A, B)$$

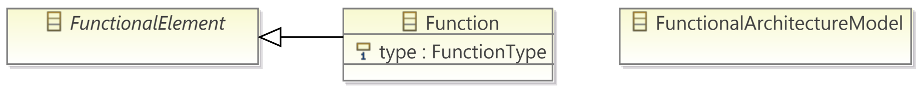
An extract of the transformation of class hierarchy is shown in Figure 20:

7.1.4 References

The references of the metamodels define the directed edges between the instance objects. For EMF models, we allow directed loops but disallow edges of the same type between the same objects. In such a case, edges can be treated as relations (in the mathematical sense).

EMF	R ∈ EReference
FOL	$link^{\mathsf{R}} : Object \times Object \to \{true, false\}$

EMF



FOL

$\forall o \in \mathit{Object}:$

		FunctionalElement		Function		FAM	
Function	(	$\mathit{type}^{\mathsf{FunctionalElement}}(o)$	$\wedge$	$\mathit{type}^{\mathsf{Function}}(o)$	$\wedge$	$\neg\mathit{type}^{\mathsf{Function}}(o)$	$)\vee$
FAM	(	$\neg\mathit{type}^{\mathsf{FunctionalElement}}(o)$	$\wedge$	$\neg\mathit{type}^{\mathsf{Function}}(o)$	$\wedge$	$\mathit{type}^{\mathsf{FAM}}(o)$	)

Fig. 20: Mapping type hierarchy

The definition of the subelements reference on is translated to the following relation in Figure 21. In order to ensure the type compliance, type information needs to be attached to the two relation ends (see line **TC**).

EMF	R $\in$ EReference between SRC and TRG EClasses
FOL	$\forall \mathit{src}, \mathit{trg} \in \mathit{Object}: \mathit{link}^{\mathsf{R}}(\mathit{src}, \mathit{trg}) \Rightarrow (\mathit{type}^{\mathsf{SRC}}(\mathit{src}) \wedge \mathit{type}^{\mathsf{TRG}}(\mathit{trg}))$

7.1.5 Multiplicity

By default, references with 0..* multiplicity are modeled with relations. However, with explicit multiplicity restrictions, further assertions are required. With an n..m multiplicity, the lower bound n means that every object is in relation with n different one, which is checked using an existential quantifier (if n > 0).

EMF	R $\in$ EReference with n..? multiplicity, n > 0 between SRC and TRG
FOL	$\forall s \in \mathit{Object}: \mathit{type}^{\mathsf{SRC}}(s) \Rightarrow (\exists t_1, \ldots, t_n \in \mathit{Object}: \mathit{distinct}(t_1, \ldots, t_n) \wedge \mathit{link}^{\mathsf{R}}(s, t_1) \wedge \ldots \wedge \mathit{link}^{\mathsf{R}}(s, t_n))$

The upper bound m means that there are no more than m different target elements being in relation with the object, which is prescribed using a universal quantifier.

EMF	R $\in$ EReference with ?..m multiplicity, m $\neq$ * between SRC and TRG
FOL	$\forall s \in \mathit{Object} \neg\exists t_1, \ldots t_m, t_{m+1} \in \mathit{Object}: \mathit{distinct}(t_1, \ldots, t_m, t_{m+1}) \wedge \mathit{link}^{\mathsf{R}}(s, t_1) \wedge \ldots \wedge \mathit{link}^{\mathsf{R}}(s, t_m) \wedge \mathit{link}^{\mathsf{R}}(s, t_{m+1})$

For example, the transformation of model reference of the FunctionalElement class visible in Figure 21, creates the following constraint to restrict the upper bound multiplicity to 1 (see line $\mathbf{MULT}^{max}$).

7.1.6 Inverse edges

Inverse edges in metamodels express in that if there is a relation R from the object s to the target t then there has to be an inverse relation I from t to o. The inverse relationship between parent and subElements references is illustrated in Figure 21 at line **INV**.

EMF	R, I $\in$ EReference and inv(R, I)
FOL	$\forall \mathit{src}, \mathit{trg} \in \mathit{Object}: \mathit{link}^{\mathsf{R}}(\mathit{src}, \mathit{trg}) \Leftrightarrow \mathit{link}^{\mathsf{I}}(\mathit{trg}, \mathit{src})$

7.1.7 Containment

The objects of an EMF model are arranged in a directed tree hierarchy along the containment edges. This relationship is formalized by multiple formulae.

1. First the containment relation is defined as the union of the containment-edge relations:
$$\mathit{contains}: \mathit{Object} \times \mathit{Object} \to \{\mathit{true}, \mathit{false}\}$$
$$\forall p, c \in \mathit{Object}: \mathit{contains}(p, c) \Leftrightarrow \bigvee_{\substack{\mathsf{R} \in \mathsf{EReference} \\ \mathsf{R}\text{ containment}}} \mathit{link}^{\mathsf{R}}(p, c)$$
2. The top of the containment hierarchy is called root of the model, which is declared as an uninterpreted constant. The root object does not have a parent.
$\mathit{root}: \emptyset \to \mathit{Object}$
$\forall \mathit{parent} \in \mathit{Object}: \neg\mathit{contains}(\mathit{parent}, \mathit{root})$
3. Every other object has at least one parent:
$\forall o \in \mathit{Object}: (o = \mathit{root}) \vee \exists p \in \mathit{Object}: (p \neq o) \wedge (\mathit{contains}(p, o))$
4. Every object has at most one parent:
$\forall c, p_1, p_2 \in \mathit{Object}: (\mathit{contains}(p_1, c) \wedge \mathit{contains}(p_2, c)) \Rightarrow (p_1 = p_2)$

The tree hierarchy also requires acyclicity which means that any object is unreachable from itself along a path of containment edges. Acyclicity of the containment hierarchy is inexpressible in FOL language thus approximation is needed. For example, a statement like "the containment graph is free from cycles of length 3"

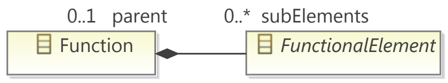
EMF		
FOL		$link^{\mathsf{subelements}} : Object \times Object \to \{true, false\}$
	TC:	$\forall src, trg \in Object : (link^{\mathsf{subelements}}(src, trg) \Rightarrow (type^{\mathsf{Function}}(src) \land type^{\mathsf{FunctionalElement}}(trg))$
	MULT^{max}**:**	$\forall s \in Object \neg \exists t_1, t_2 \in Object : t_1 \neq t_2 \land link^{\mathsf{parent}}(s, t_1) \land link^{\mathsf{parent}}(s, t_2)$
	INV:	$\forall src, trg \in Object : (link^{\mathsf{subElements}}(o, t) \Leftrightarrow link^{\mathsf{parent}}(t, o))$

Fig. 21: Mapping references

overapproximates the acyclicity requirement. Increasing the size of the forbidden cycle converges to the acyclicity, and we can deal with any kind of containment inconsistency using an appropriate approximation level up to a given depth. The following assertion forbids cycles of length 3.

$$\forall c_1, c_2, c_3 \in Object : \neg(\quad contains(c_1, c_2) \land contains(c_2, c_3) \land contains(c_3, c_1))$$

7.1.8 Enums

Enum types of the metamodel are transformed to dedicated sets with the elements of the mapped literals. Figure 22 shows the example of FunctionType.

EMF	$\mathsf{E} \in \mathsf{EEnum}$ and $\mathsf{E} = \{\mathsf{L1}, \ldots, \mathsf{Ln}\}$
FOL	$enum^{\mathsf{E}} = \{lit^{\mathsf{L1}}, \ldots, lit^{\mathsf{Ln}}\}$

7.1.9 Attributes

The attributes of a metamodel are the properties of the classes with built-in type. Our transformation approach currently handles predefined primitive types such as Boolean, integer and real attributes (which usually have an appropriate type in back-end solvers), and an enum type constructed of user-defined literals. The function *range* translates an EMF classifier to the corresponding mathematical set.

EMF	EObject	$\mathsf{E} \in \mathsf{EEnum}$	EBoolean	EInt	EDouble
FOL	*Object*	$enum^{\mathsf{E}}$	$\{true, false\}$	$\mathbb{Z}$	$\mathbb{R}$

Attributes are transformed in the same way as the relations, but the second parameter (i.e. the range) of the parameter defines the value of the attribute. For example, two attributes are defined in Figure 22.

EMF	$\mathsf{A} \in \mathsf{EAttribute}$ and T is the type
FOL	$link^{\mathsf{A}} : Object \times range(\mathsf{T}) \to \{true, false\}$

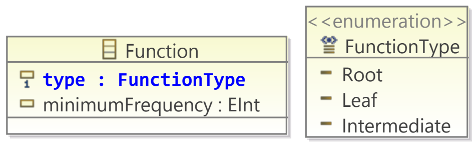
EMF	
FOL	$enum^{\mathsf{FunctionType}} = \{lit^{\mathsf{Root}}, lit^{\mathsf{Intermediate}}, lit^{\mathsf{Leaf}}\}$ $type : Object \times enum^{\mathsf{FunctionType}} \to \{true, false\}$ $minFreq : Object \times \mathbb{Z} \to \{true, false\}$

Fig. 22: Transformation of enum types and attributes

7.2 Transformation of Partial Snapshots

Mapping of partial snapshots to FOL is mainly driven by how to transform instance models to corresponding formulae which are included in the set of axioms for a DSL context. In general, the metamodel and the constraints of the language define universally quantified formulae over all model elements. Partial snapshots enable to efficiently configure the validation process using large existentially quantified properties. Furthermore, the transformation can be customized in accordance with semantic modifiers. Table 2 summarizes the transformation steps.

Features of the PS	**SMT**		**SAT**
Instance Objects	E	+	E
Abstract or Concrete Types	E	+	E
Filled References	E	+	E
Filled Attributes	E	+	E
Configuration of the PS	**SMT**		**SAT**
Positive / Negative	E	+	E
Injective / Shareable	E	+	E
Modifiable / Unmodifiable	E	+	E

E: Expressible A: Approximable X: Inexpressible
+: in EPR −: not in EPR

Table 2: Mapping partial snaptshots to FOL

7.2.1 Instance Models

The basic approach of transforming partial snapshots into FOL is to create a statement to express that the output model needs to contain the partial snapshot. Therefore PS objects are transformed into existentially quantified *Object* variables, and the structure of the PS is defined defined over those variables. When every feature of the PS is transformed, the generated statement is added to the set of axioms derived for a DSL context to express the occurrence or the absence of the PS structure. The partial snapshots are transformed independently to FOL, each of them has to be satisfied separately, and traceability information needs to be produced for each of them.

PS	$\mathsf{M} \models \mathsf{Meta}, \lvert\mathsf{M}\rvert = n, \mathsf{M} = \{pscon_1, \ldots, pscon_m\}$
FOL	$\exists o_1, \ldots, o_n \in Object : distinct(o_1, \ldots, o_n) \wedge \bigwedge_{1 \le i \le m} pscon_i(o_1, \ldots, o_n)$

The result of the transformation of a PS with two objects and a link between them in accordance with the Positive, Injective, Unmodifiable semantics is illustrated in Figure 23, and discussed in details in the sequel.

The type of the object must also be specified in the statement of the partial snapshot. Note that abstract classes can be instantiated in a PS, and thus they are transformed to a structural constraint. For example, if the type of o_1 is FunctionalElement it is transformed to the constraint labeled as **Types** in Figure 23. We explicitly enumerate all classes which are types of o_1 and the negations of all non-types of o_1 in the derived predicate.

PS	$\mathsf{C}(o), \mathsf{C} \in \mathsf{EClass}$
FOL	$type^{\mathsf{C}}(o)$

The references between the objects can be defined by stating that the pair of the source and the target object is in a given relation. The mapping of reference subElements is depicted in Figure 23 (see label **References**), which states that o_1 is a sub-element of o_2 while o_2 is not a sub-element of o_1. Note that the latter restriction is due to the unmodified semantic modifier of the partial snapshot (and not the composition semantics of subElements reference). Attributes in PSs are defined similarly, see the corresponding **Attributes** line in Figure 23 .

PS	$\mathsf{R}(s,t), \mathsf{R} \in \mathsf{EReference}$	$\mathsf{A}(o,v), \mathsf{A} \in \mathsf{EAttribute}$
FOL	$link^{\mathsf{R}}(s,t)$	$link^{\mathsf{A}}(o,v)$

7.2.2 Semantic modifiers

In the previous section, we defined the mapping of partial snapshots according to the Positive, Injective, Unmodifiable semantics, which defines the most concrete (restrictive) specification of a PS. Other semantic modifiers can be handled by simply relaxing (removing) certain constraints from this complete set derived for the Positive, Injective, Unmodifiable case.

This section shows different configurations of partial snapshots with respect to a Positive, Injective, Unmodifiable one. Figure 24 shows how omitting specific constraint from the statement changes the semantics of the PS.

- Shareable: If the PS is configured to be shareable, the **Distinct** constraint have to be omitted from the formula, so multiple variables can be bound to the same object variable of the PS.
- Modifiable: If the PS is unmodifiable then the absence of a reference or attribute is captured as a negated predicate. By omitting the negative predicates from **References** and **Attributes**, it is allowed to add new references to the model. By omitting the negative type predicates from the **Types** constraint, the type of an object can be refined, i.e. an instance object with a T type can be mapped to an object with a subtype of T. As a result, even abstract objects can be used in a PS, and they will be matched to a concrete one.
- Negative Depending on that the PS is configured as positive or negative, the assertion of the generated statement or its negation is inserted to the axioms.

8 Transforming Constraints to First Order Logic

8.1 Transforming Graph Patterns to FOL

This subsection describes how EMF-IncQuery patterns can be transformed to first order logic formulae. Table 3 shows an overview on which feature can be translated to FOL and EPR when using them as wellformedness constraints or as derived features.

8.1.1 Structure of the Patterns

A graph pattern consists of a list of symbolic parameters as header and a content that specifies logical conditions over the parameters. The parameter list of the pattern pattern is a fix sized vector of variables denoted as $\mathsf{params} = (p_1, \ldots p_n)$. A match is a complete function

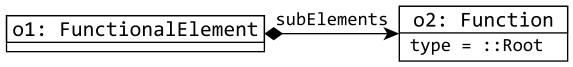
PS		
FOL	**Containment:**	$\exists o_1, o_2 \in Object : [\ldots]$
	Distinct:	$o_1 \neq o_2$
	Types:	$type^{\mathsf{FunctionalElement}}(o_1) \wedge \neg type^{\mathsf{Function}}(o_1) \wedge type^{\mathsf{FunctionalElement}}(o_2) \wedge type^{\mathsf{Function}}(o_2)$
	References:	$link^{\mathsf{subElements}}(o_1, o_2) \wedge \neg link^{\mathsf{subElements}}(o_2, o_1)$
	Attributes:	$link^{\mathsf{type}}(o_2, Root) \wedge \neg link^{\mathsf{type}}(o_2, Intermediate) \wedge \neg link^{\mathsf{type}}(o_2, Leaf)$

Fig. 23: Example: transformation an unmodifiable partial snapshot

	Removed Constraint	$\rightarrow$	Partial Snapshot
Distinct:	$o_1 \neq o_2$	$\rightarrow$	shareable containment
Types:	$\neg type^{\mathsf{Function}}(o_1)$	$\rightarrow$	refinable types
References:	$\neg link^{\mathsf{subElements}}(o_2, o_1)$	$\rightarrow$	additional edges
Attributes:	$\neg link^{\mathsf{type}}(o_2, lit^{\mathsf{Root}})$	$\rightarrow$	additional attributes

Fig. 24: Effects of removing constraints from Partial Snapshots

	SMT				SAT
Features of model query	**DF**		**WF**		
Classifier constraint	E	+	E	+	E
EReference constraint	E	−	E	+	E
Acyclic pattern call	E	−	E	+	E
Negative pattern call	E	−	E	−	E
Transitive closure	A	−	A	+	E
Arbitrary call graph	A	−	A	−	X
Aggregate (eg. Count, Sum)	X		X		E
Check (for algebra)	X		X		A

E: Expressible A: Approximable X: Inexpressible +: in EPR −: not in EPR

Table 3: Expressing Ecore and EMF-IncQuery language features in Z3

of the parameters to the elements of the model.

$$\mathsf{m} : \mathsf{params} \rightarrow \mathsf{EObject} \cup \bigcup_{\mathsf{E} \in \mathsf{EEnum}} \mathsf{E} \cup \bigcup_{\mathsf{P}\ \mathrm{primitive}} \mathsf{P} \qquad (6)$$

In order to represent queries in the axiom system, their match set of each pattern is transformed to an uninterpreted predicate over the corresponding types of model elements defined by the *range* function. This predicate to true for a specific assignment of parameters params/m exactly when m constitutes a match i.e. satisfies the pattern constraints.

eIQ	`pattern` q $(p_1 :\ T_1, \ldots, p_n :\ T_n)$ $\langle constraint \rangle$
FOL	$pattern^{\mathsf{q}} : range(T_1) \times \ldots \times range(T_n) \rightarrow \{true, false\}$ $\forall p_1 \in range(T_1), \ldots, p_n \in range(T_n) :$ $pattern^{\mathsf{q}}(p_1, \ldots, p_n) \Leftrightarrow constraint(p_1, \ldots, p_n)$

For example, let us take a two parameter pattern called type with the parameter list This: Function and Target: FunctionType. The matches of this pattern are defined by the predicate in **Declaration** column of Figure 25.

In EMF-IncQuery the constraint of a pattern is specified by pattern bodies. An individual object vector is a member of the match set if and only if the vector satisfies the condition defined by one of the pattern bodies, which are defined the disjunction conditions in the pattern body. The type pattern specifies three bodies which is illustrated at the in **Bodies** column of Figure 25.

eIQ	`pattern` q $(p_1, \ldots, p_n)$ $\{b_1\}$`or`$\ldots$`or`$\{b_m\}$
FOL	$pattern^{\mathsf{q}}(p_1, \ldots, p_n) \Leftrightarrow \bigvee_{1 \leq i \leq m} b_i(p_1, \ldots, p_n)$

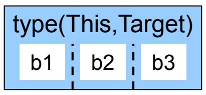
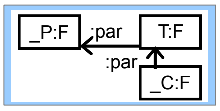
8.1.2 Transformation of the pattern condition

The pattern body condition is defined by the constraints of the body, where the condition is the conjunction of the constraints. A pattern body may introduce additional existentially quantified local variables. A variable might be introduced for a single use in a constraint, in this case the variable is specified as anonymous with '_' as the first character in its name. For example, `Function.parent(_C,T)` specifies that there have to be an incoming parent reference to the object T from an irrelevant object. By default, anonymous variables are also transformed to a existentially quantified local variables.

eIQ	$\{c_1(pars, vars);\ \ \ldots\ \ c_n(pars, vars);\}$
FOL	$body(p_1, \ldots, p_n) \Leftrightarrow \exists vars \bigwedge_{1 \leq i \leq n} c_i(pars, vars)$

For example, the intermediate body of the type pattern contains two path and three classifier constraints which introduce two local variables to the five constrains in the **Constraints** column of Figure 25. In

	Declaration	Bodies	Constraints

eIQ	`pattern type(`
	` This:Function,`
	` Target:FunctionType)`
	`{`
	` ...`
	`}`
		
		`{` b_1 `}or{` b_2 `}or{` b_3 `}`
			
			c_1:`Function.parent(T,_P)`
			c_2:`Function.parent(_C,T)`
			c_3:`Function(T)`
			c_4:`Function(_C)`
			c_5:`Function(_P)`

FOL	$pattern^{\mathsf{type}} : Object \times enum^{\mathsf{FunctionType}} \to \{true, false\}$	$pattern^{\mathsf{type}}(v_1, v_2) \Leftrightarrow b_1 \vee b_2 \vee b_3$	$b_2(v_1, v_2) \Leftrightarrow \exists C, P \in Object : c_1 \wedge c_2 \wedge \ldots \wedge c_5$

Fig. 25: Example pattern structure transformation

summary the pattern condition is structured as follows: $pattern(pars) = \bigvee_{body} \exists vars \bigwedge_{const} const(pars, vars)$.

The following subsection defines the transformation for each supported type of constraints.

8.1.3 Constraint Mapping

Classifier constraints define the type of the objects that are bound to a variable. A graph constraint can be easily compiled to a type predicate as follows:

eIQ	$\langle\mathsf{class}\rangle$ `(`v`);`
FOL	$type^{\mathsf{class}}(v)$

Path constraints define that there is a path consisting of a sequence of references of corresponding types that leads from a source object to a target object (identified by pattern variables). In the most simple case, a path constraint consists of navigating along a single reference, which is transformed in the following way:

eIQ	$\langle\mathsf{class}\rangle$`.`$\langle\mathsf{feature}\rangle$`(`$src$`,` trg`);`
FOL	$link^{\mathsf{feature}}(src, trg)$

For complex paths, we introduce implicit object variables as the inner nodes of the path, thus the expression can be compiled into a conjunction of simple reference predicates form the first variable through the inner ones to the last.

eIQ	$\langle\mathsf{class}\rangle$`.`$\langle\mathsf{feature}_1\rangle \ldots \langle\mathsf{feature}_n\rangle$`(`$src$`,` trg`);`
FOL	$\exists o_1, \ldots o_{n-1} \in Object : link^{\mathsf{feature}_1}(src, o_1) \wedge$ $\bigwedge_{2 \leq i \leq n-2} link^{\mathsf{feature}_i}(o_{i-1}, o_i) \wedge link^{\mathsf{feature}_n}(o_{n-1}, trg)$

For example, the simple path expression constraint FunctionalElement.parent(_ChI,This) defines that there is a path that starts from _ChI, ends in the This object in Figure 26;

Equality and non-equality of two individuals can be simply defined as FOL equality:

eIQ	a `==` b`;`	a `!=` b`;`
FOL	$a = b$	$a \neq b$

Pattern call constraints enable the creation by calling elementary ones. The following list provides the transformation rules for each kind of transformation, and an example result of a negative pattern call is presented in the **Pattern call** column of Figure 26.

– A positive call defines that the substituted parameters have to create a match of the referred pattern.

eIQ	`find` $\langle\mathsf{called}\rangle$ `(`v_1`,...,`v_n`);`
FOL	$pattern^{\mathsf{called}}(v_1, \ldots, v_n)$

– Negative calls may introduce new (in this case universally quantified) variables. A negative pattern call defines that the target pattern should not have a match for the substituted old variables with for any possible substitution of the new parameters.

eIQ	`neg find` $\langle\mathsf{called}\rangle$ `(`$v_1 \in T_1$`,...,`$v_n \in T_n$`);`
FOL	$\forall \underbrace{\ldots, v_i \in T_i, \ldots}_{\substack{1 \leq i \leq n, \\ \text{new variable } v_i}} : \neg pattern^{\mathsf{called}}(v_1, \ldots, v_n)$

– Transitive closure is an advanced language element of the EMF-IncQuery pattern language. The transitive closure of a two-parameter GP matches on the pair (e_1, e_n) if there is a sequence $e_1, e_2, \ldots e_n$ of model elements where the pattern matches every pair (e_i, e_{i+1}).

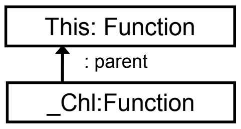
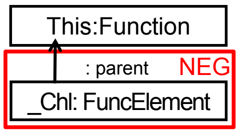
	Classifier	Path	Equivalence	Pattern call
eIQ	This:Function `Function(This)`	 `Function.parent(_Chl,This)`	Target==‘Intermediate’ `Target==::Intermediate`	 `neg find parent(_Child,This)`
FOL	$type^{\text{Function}}(This)$	$link^{\text{parent}}(\_Chl, This)$	$Target = lit^{\text{Intermediate}}$	$\forall This : \neg pattern^{\text{parent}}(Child, This)$

Fig. 26: Example constraints

Transitive pattern:

```
pattern parentOf(This,P) {
  Function.parent(This,P);
}
```

Transitive approximations:

2:Path from *This* to *P*

$pattern^{\text{parentOf+}}_{O=2}(This, P) \Rightarrow link^{\text{parent}}(This, P) \vee$
$\exists m_1 : link^{\text{parent}}(This, m_1) \wedge pattern^{\text{parentOf+}}_{O=1}(m_1, P, This)$

1:Path from *This* to *P* without d_1

$pattern^{\text{parentOf+}}_{O=1}(This, P, d_1) \Rightarrow link^{\text{parent}}(This, P) \vee$
$\exists m_2 : m_2 \neq d_1 \wedge$
$link^{\text{parent}}(This, m_2) \wedge pattern^{\text{parentOf+}}_{O=0}(m_2, P, d_1, This)$

0:Path from *This* to *P* without d_1 and d_2

$pattern^{\text{parentOf+}}_{O=0}(This, P, d_1, d_2) \Rightarrow link^{\text{parent}}(This, P) \vee$
$\exists m_3 : m_3 \neq d_1 \wedge m_3 \neq d_2 \wedge$
$link^{\text{parent}}(This, m_3) \wedge \textbf{true}$

Overapproximated pattern specification:

$pattern^{\text{parentOf+}}(This, P) \Rightarrow pattern^{\text{parentOf+}}_{O=2}(This, P)$

Table 4: Overapproximation of the transitive closure.

eIQ	`find` $\langle\text{called}\rangle$`+(`$v_s$`, `$v_t$`);`
FOL	$\exists o_1, \ldots o_{n-1} \in Object : pattern^{\text{called}}(v_s, o_1) \wedge$ $\bigwedge_{2 \leq i \leq n-2} pattern^{\text{called}}(o_{i-1}, o_i) \wedge$ $pattern^{\text{called}}(o_{n-1}, v_t)$

Transitive closure approximation The transitive closure of a pattern can only be approximated in FOL. The essence of this approximation is to generate a sequence of predicates p_i by unrolling its definition so that each predicate p_i checks for matches of length i. At depth i, a predicate checks if there is a match exactly at length i or recursively checks for a match at depth $i+1$ by using predicate p_{i+1}. At maximal depth n, the predicate is overapproximated by *true* or underapproximated by *false*. A more detailed handling of transitive closure approximation and other recursive pattern calls is available in [43].

For example, let us define an overapproximation for length 2 of the transitie call of the parentOf pattern by unrolling the transitive closure of the parent EReference in Table 4.

Check expression By the use of check constraint it is available to call imperative (Java-like) xBase expressions to be evaluated on the variables of the pattern. A check constraints specifies that the result of the evaluation have to be true for each valid match. This paper discusses the translation of basic arithmetic and logic operators, which are simply translated to the corresponding logic expression.

eIQ	`a+b`	`a-b`	`a*b`	`a/b`	`a=b`	`a&&b`	`a||b`	`!a`
FOL	$a+b$	$a-b$	$a \cdot b$	a/b	$a=b$	$a \wedge b$	$a \vee b$	$\neg a$

8.1.4 Patterns for advanced DSL constructs

Graph patterns are used in different ways to specify restrictions on the structure of the DSL by well-formedness (or ill- formedness) constraints or defining derived features.

– Regular *well-formedness constraints* are treated as they are, while *ill-formedness constraints* are defined as a statement that the model is free from matches of this pattern. For example, in case of the pattern terminatorAndInformation the corresponding predicate looks like this:

eIQ	`@Constraint`
	`pattern q(`$v_1 \in T_1, \ldots, v_n \in T_n$`)`

FOL	$\forall v_1 \in T_1, \ldots, v_n \in T_n : \neg pattern^{\text{q}}(v_1, \ldots, v_n)$

– Predicates for *derived features* state that features evaluate to the value exactly when the specifying pattern has a match on the given object and the value. The predicate for DF type looks like this:

eIQ	`@QueryBasedFeature`
	`pattern` $\langle feature\rangle$`(`$src, trg \in T$`)`

FOL	$\forall src \in Object, trg \in range(T) :$ $pattern^{\text{feature}}(src, trg) \Leftrightarrow link^{\text{feature}}(src, trg)$

8.2 Transforming OCL Invariants to FOL

OCL constraints (invariants) are widely used means to express *well-formedness rules* of DSLs which has to be satisfied by all valid instance models. Here we present a transformation from a subset of OCL invariants to FOL. As a result, DSL validation tasks (e.g. consistency check, subsumption check) can be executed even when certain WF constraints are defined by graph patterns while others are captured by OCL invariants. This is a very practical setup to allow DSL engineers to mix specification languages.

While there are existing OCL-to-FOL transformations (e.g. [21, 23]) which cover a larger portion of the OCL language (and its semantic cornercases), our technique allows to detect inconsistencies between WF constraints with different representations (OCL vs GP) and allows to reason about subsumption or equivalence of such constraints. Furthermore, the combined use of our rich partial snapshot language and OCL invariants also allows to gain additional insight into DSL specifications. Finally, we also introduce approximations for certain OCL language elements.

8.2.1 An overview of OCL transformation

The syntax of an OCL invariant expression is presented in the template below, where context specifies the environment (e.g. class) on which the constraint is interpreted, the name of the constraint can be given after the inv keyword and finally the expression is specified.

context ⟨*classifier*⟩ **inv** [⟨*name*⟩] : ⟨*expression*⟩

Our transformation takes an OCL invariant as input and synthesizes FOL formulae as output. Certain OCL language elements are too expressive to be represented in FOL, but they can be approximated by appropriate FOL formulae. Each supported language element is presented in this subsection.

The OCL standard defines a four valued logic (with true, false, null and undefined values), while we mostly restrict our mapping to a two-valued logic and the null is also supported as input of the comparison operators.

The structure of an OCL invariant is similar to its FOL counterpart, so the mapping algorithm transforms the elements explicitly, one-by-one to FOL formulae.

The mapping algorithm traverses the *abstract syntax tree* (AST) of the OCL invariant recursively and applies the corresponding rules to each subexpression element. However, there are some special cases which require pre- or post-processing the result, e.g. for comparison functions, null references and collection operators. The mapping also handles a restricted set of higher-order structures (like sets) which structures are unfolded and represented by FOL predicates.

8.2.2 Basic expressions

The mapping rules of the basic expressions (primitive-type, simple arithmetic and bool expression and variable) of OCL are depicted in Table 5. In OCL, we cover the primitive types Integer, Boolean and Real, while the String types is not yet supported. The logic and arithmetic operators are directly transformed to FOL, since they have their equivalent counterpart in the FOL if they are interpreted on Integers, Reals and Booleans.

OCL	`var`	a+b	a-b	a*b	a/b	a=b
FOL	var	$a+b$	$a-b$	$a\cdot b$	a/b	$a=b$

OCL	a **and** b	a **or** b	a **implies** b	**not** a
FOL	$a \wedge b$	$a \vee b$	$a \Rightarrow b$	$\neg a$

Table 5: Mapping of basic OCL expressions

The OCL function **oclIsKindOf**(Class) is translated to a type predicate of Class, which is satisfied if and only if the object has the same type as the argument of the function.

OCL	var.**oclIsKindOf**(class)
FOL	$type^{\text{class}}(var)$

Objects, as complex structures require special transformation rules. *Single-valued attributes* of objects are translated to functions.

OCL	var_1.⟨attribute⟩ = var_2
FOL	$link^{\text{attribute}}(var_1, var_2)$

Navigation along *references with at most one multiplicity* is compiled into a two-parameter predicate and an existentially quantified formula to avoid dealing with undefined values.

OCL	var_1.⟨referred⟩ = var_2
FOL	$\exists var_2\ link^{\text{referred}}(var_1, var_2)$

The *equal* operator of OCL (=) is transformed similarly to other mathematical operators unless objects need to be compared. The transformation of such equality expressions are divided into two cases: (1) if a variable is placed on the right hand side, then an existential quantifier is used to avoid undefined values, (2) if the comparison is to value null, then the expression is transformed using a negated existential qualifier in FOL.

OCL	$var_1.\langle\mathsf{referred}\rangle = var_2$
FOL	$\exists var_2\ link^{\mathsf{referred}}(var_1, var_2)$
OCL	$var_1.\langle\mathsf{referred}\rangle$=`null`
FOL	$\neg\exists var_2\ link^{\mathsf{referred}}(var_1, var_2)$

The transformation of the *not equal* operator of OCL (<>) uses the dual counterparts of equal operation by adding a negation to the corresponding FOL expressions.

8.2.3 Collections

OCL collections are special sets in the mathematical aspect obtained mostly by specific language constructs (e.g. `allInstances` or navigations along references). In our approach, every collection c is represented by a characteristic predicate $P^{\mathsf{c}}(x)$ captured in FOL, where the predicate evaluates to *true* on a model element only if it is the member of the collection: $P^{\mathsf{c}}(x) \Leftrightarrow \mathsf{x} \in \mathsf{c}$. If the collection can be implied by the context, the index of the collection is omitted: $P(x)$.

The OCL operation `allInstances` refers to all instances of a certain type, that way, the corresponding FOL formula collects the objects with the referred type. We also can refer to the set of elements with a certain type as defined by the context of the invariant using `self` keyword, which is handled exactly as the `allInstances` construct.

OCL	`context` ⟨class⟩ `inv: ... self ...`
OCL	⟨class⟩`.allInstances()`
FOL	$P(x) \equiv type^{\mathsf{class}}(x)$

A set of instances can be also referred by a reference with more than one multiplicity. The predicate of the reference and the predicate of the variable is included in the FOL expression during the mapping.

OCL	$var.\langle\mathsf{referred}\rangle$
FOL	$P(x) \equiv link^{\mathsf{referred}}(var, x)$

Navigation along references (see next example) with more than one multiplicity is also alloved in OCL (first line), but it is a shorthand of the `collect` operator and is transformed to this operator directly by the OCL parser (second line). The equivalent FOL expression contains the predicates of the references included in the path and a temporary variables with universal quantifier for the intermediate points. The predicates of the intermediate- and endpoints are not needed, since the predicates of the references include them.

OCL	$var.\langle\mathsf{referred}_1\rangle\dots\langle\mathsf{referred}_n\rangle$
FOL	$P(x) \equiv \exists v_1,\dots,v_{n-1} : link^{\mathsf{referred}_1}(var, v_1)\wedge$ $\bigwedge_{2\le i\le n-2} link^{\mathsf{referred}_i}(o_{i-1}, o_i)\wedge$ $link^{\mathsf{referred}_n}(v_{n-1}, x)$

The `collect` operator derives a collection from another by applying an $exp(v)$ OCL expression on the elements of the collection v, which is defined by other mapping rules.

OCL	`c -> collect(`$v \mid exp(v)$`)`
FOL	$P(x) \equiv \exists v : P^{\mathsf{c}}(v) \wedge exp(v) = x$

The `closure` operator derives a collection using an $exp(v)$ expression by iteratively applying on the elements of the collection until it reaches fix point. The transitive closure in ocl approximated similarly as in the case of graph patterns. The following transformation presents the overapproximation of the `closure` operator by 3 steps:

OCL	`c -> closure(`$v \mid exp(v)$`)`
FOL	$P_{O=3}(x) \equiv P^{\mathsf{c}}(x) \vee \exists v : P^{\mathsf{c}}(v)\wedge$ $((x = exp(v) \wedge distinct(v, x))\vee$ $(x = exp^2(v) \wedge distinct(v, exp(v), x))\vee$ $(x = exp^3(v) \wedge distinct(v, exp(v), exp^2(v), x))\vee$ $(true \wedge distinct(v, exp(v), exp^2(v), exp^3(v), x)))$

The `select` and `reject` operators are used to define a special subset of the collection by an expression $exp(v)$. The `select` constructs a condition, which elements should be included, while the `reject` defines the elements that should be excluded from the collection. The transformation of these operators appends the transformation of the expression to end of the predicate of the collection.

OCL	`c->select(`$v \mid exp(v)$`)`	`c->reject(`$v \mid exp(v)$`)`
FOL	$P(x) \equiv P^{\mathsf{c}}(x) \wedge exp(x)$	$P(x) \equiv P^{\mathsf{c}}(x) \wedge \neg exp(x)$

8.2.4 Collection operators

We now overview the transformation of different OCL operators which are applicable to collections.

The OCL operation `includes` is evaluated to *true* if the collection contains at least one element satisfying the argument expression of the function. This function is translated to a predicate which checks containment. The `excludes` OCL function is the dual of `includes`, so it is transformed to the negated FOL expression of `includes`. This expression is satisfied if the elements of the collection do not satisfy the condition.

OCL	c `->` `includes`(v)	c `->` `excludes`(v)
FOL	$P^{\mathsf{c}}(v)$	$\neg P^{\mathsf{c}}(v)$

OCL operation `forAll` is an iterator over a collection to state that certain conditions hold for each member of the collection. We restrict our transformation to set semantics, and then the FOL equivalent is the universal quantifier. The OCL operation `exists` implements an iterator and the FOL equivalent is the existential quantifier.

OCL	c`->forAll`$(v \mid exp(v))$	c`->exists`$(v \mid exp(v))$
FOL	$\forall v : P^{\mathsf{c}}(v) \Rightarrow exp(v)$	$\exists v : P^{\mathsf{c}}(v) \wedge exp(v)$

The OCL function `notEmpty` is applied on collections and is satisfied if the collection is not empty. This operation is transformed to an existentially quantified predicate which means that there is at least one object in the given set or collection. The OCL function `isEmpty` handled with an additional negation.

OCL	c `->` `notEmpty()`	c `->` `isEmpty()`
FOL	$\exists v : P^{\mathsf{c}}(v)$	$\neg\exists v : P^{\mathsf{c}}(v)$

The OCL function `size` returns the size of the collection. In FOL the size of a collection cannot be formulated in general, but if the complarison of the size of the collection to an integer is translated similarly as in case of handling multiplicities in metamodels (see Section 7.1.5).

We implemented transformations using approximation for every comparison operators. The transformation of the less than (=<) and greater than (=>) operators provides the basis for the mapping of the equality and inequality operators.

The translation of the greater than (=>) operator is carried out by introducing temporary variables (as much as needed), adding the predicates of the reference for each variable and finally declare pair-wise inequality.

OCL	c `->` `size()>=`n , $n \in \mathbb{Z}^+$
FOL	$\exists v_1, \ldots, v_n : distinct(v_1, \ldots, v_n) \wedge \bigwedge_{1 \leq i \leq n} P^{\mathsf{c}}(v_i)$

The translation of the less than (=<) operator is similar, but the equality is declared at least for one pair of the temporary variables.

OCL	c `->` `size()<=`n , $n \in \mathbb{Z}^+$
FOL	$\neg\exists v_1, \ldots, v_n : distinct(v_1, \ldots, v_n) \wedge \bigwedge_{1 \leq i \leq n} P^{\mathsf{c}}(v_i)$

Finally, the equality operator is divided into two parts before the mapping:

OCL	c `->` `size()=`n , $n \in \mathbb{Z}^+$
OCL	c `->` `size()<=`n `and` c `->` `size()>=`n

The handling of equality and inequality operators has specific rules if two collections are passed as input. Two collections are equal if and only if neither of them contains an element which is not in the other set.

OCL	c `=` d
FOL	$\neg\exists v : (P^{\mathsf{c}}(v) \wedge \neg P^{\mathsf{d}}(v)) \vee (\neg P^{\mathsf{c}}(v) \wedge P^{\mathsf{d}}(v))$

Passing two collections as input, the inequality operator evaluates to *true* if there exists at least one element which is not contained by both of them.

OCL	c `<>` d
FOL	$\exists v : (P^{\mathsf{c}}(v) \wedge \neg P^{\mathsf{d}}(v)) \vee (\neg P^{\mathsf{c}}(v) \wedge P^{\mathsf{d}}(v))$

8.2.5 Restrictions and Expressiveness

The expressiveness of OCL is higher than first-order logic, so some language constructs are obviously not covered by our transformation. Due to the undecidable nature of the full OCL language, it cannot be expected to come up with an automated unsatisfiability checker for all the OCL expressions. Still, we believe that covering a subclass of OCL expressions and support language-level validation on them is a practical solution.

Features of the OCL	**SMT**		**SAT**
Logic operators	E	+	E
Arithmetic operators	E	−	A
oclIsTypeOf	E	+	E
Attributes	E	+	E
References	E	+	E
Collections (Sets)	E	+	E
Collections (Bag, Sequence)	X		X
allInstances, self	E	+	E
Iterator expressions (e.g. exists, forAll)	E	−	E
notEmpty	E	−	E
isEmpty	E	+	E
Transitive closure	A	+	E
min, max	X		E
pre, post	X		X

E: Expressible A: Approximable X: Inexpressible
+: in EPR −: not in EPR

Table 6: Expressing OCL features in FOL

In our approach, the OCL constructs like OrderSet, Bag and Sequence and operations like max() and min() are not handled. We only focus on OCL invariants and do not support general OCL queries or operation constraints captured by pre- and postconditions. The list of the supported language elements is overviewed in Table 6

9 Tool Support and Experimental Evaluation

9.1 Prototype Tool

Our DSL validation tool is fully integrated to the Eclipse (Modeling) framework, and can be used immediately on the developed DSL at design time. An important advantage of our validation tool is that it can be operated by a DSL developer. Since it can be configured from the DSL front-end and all validation results are back-annotated to the DSL itself, it does not require any theorem proving knowledge from DSL experts.

An architectural overview of the tool is presented in Figure 27, which divides the core reasoning process into four consecutive components:

1. Input Parametrization: The tool collects the input DSL artifacts (see Figure 12) and the reasoner customizations according to a configuration file to select the validation task and refine the DSL context.
2. Modular Transformation: The different DSL artifacts are forwarded to the appropriate transformation module, which transforms the validation task to an abstract consistency checking problem. The output of the validation, i.e. the generated set of formulae is passed to the next component.
3. Theorem Prover Integration: The axiom system in FOL is transformed to the concrete syntax of the target reasoner, (i.e. the SMT2 input language of the Z3 SMT solver [24], or the Alloy language of the Alloy Analyzer [30] in our case). The framework calls the reasoners with the specified configuration and compiles a logic model from the result of the reasoning and passes it to the final component.
4. Post-processing: In this phase, a completed PS is built up from the results of different queries over the logic model and interpreted in the context of the validation task. At the end of the validation process, the result PS can be presented to the user in multiple ways, e.g. as an object diagram-like graph, or as a standard instance model with the original concrete syntax of the language.

We support translation of FOL axioms to both Z3 and Alloy (with sat4j) tools; a brief overview of these transformation is provided Figure 28 in the Appendix.

- An SMT-solver like Z3 can address the full range of DSL validation problems. It constructs both models and proofs, and an infinite number of cases can be symbolically checked. It handles numbers and mathematical theories efficiently, but it is not particularly good at handling objects.
- Alloy can only construct (finite) models as output. It can generate models with many objects, but the handling of integers (and other core datatypes) is questionable. However, since bounded, finite domains are considered by the underlying SAT-solver, Alloy is unable to prove if a set of axioms is unsatisfiable (i.e. the absence of an output model is not a proof).

The output partial snapshot represents a witness model or a counterexample depending on the validation problem. An extensible set of presentation techniques have been implemented to ease the interpretation of the result of the validation on the DSL level.

- Different visualization techniques are applied to show an object diagram-like representation of partial snapshots. At first, a graph format is created for every instance model, which is editable with the yFiles yEd [60] tool. Additionally, a graph is presented in the editor using the Zest [56] plugin of Eclipse.
- If the result model is a valid instance model then the partial snapshot is transformed to a standard instance model of the domain specific language. As a result, the DSL expert can observe the instance model in its native editor. Moreover, it is also useful when the validation tool is used in a tool chain.
- Finally, a report is generated to summarize the results of multiple validation tasks executed in a batch including the statistics of the reasoning process.

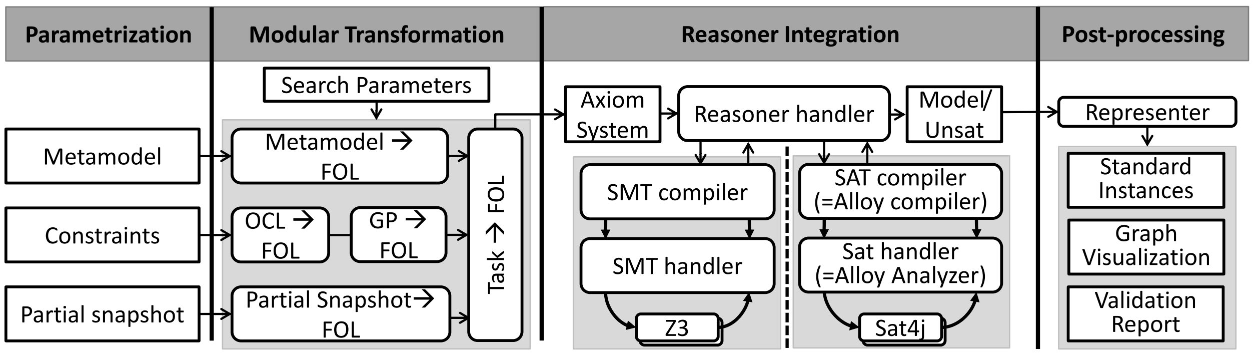


Fig. 27: Architecture of the prototype tool

9.2 Runtime Measurements

9.2.1 Measurement Environment

In order to assess the performance of our approach, we carried out initial experimental evaluation. The execution of our validation framework consists of four phases:

1. the *transformation of the validation task* (DSL2FOL), which is proportional to the size of models and the number of constraints;
2. the *execution of the reasoner tool* (FOL2FOL), which can be complex and time consuming
3. the *resolution of the output model* (FOL2PS), which is proportional to the size of the output FOL model
4. the *visualization of the output model* (PS2DSL), which is proportional to the size of output partial snapshot

We concluded that the runtime of Steps 1, 3 and 4 is predictable and negligible compared to the execution of the reasoning step, which is, in fact, difficult to predict. Therefore, below we restrict our runtime measurements to assess the performance of the reasoning step for various validation tasks on the avionics DSL presented in the paper (see Table 7 for an overview of properties used for our experiments).

Abbr.	Property	Defined in
Freq-GP	wrongFrequency	Figure 7
Freq-OCL	RightFrequency	Figure 7
T&I-GP	terminatorAndInformation	Figure 6
T&I-OCL	terminatorNoLink	Figure 6
Half-GP	InformationandTermintator	Figure 18
Half-PS	negative terminated_link	Figure 8

Table 7: Properties in experimental validation

The tasks serving as test cases are marked with the expected output, which can be positive or negative. The analysis can prove the selected property, check the absence of violations within a bounded context, or result in a timeout. The marking used in this section are illustrated in Table 8.

[✓]	Positive result is expected
[×]	Negative result is expected
[?]	Checked in a bounded context, but not proven
-	Timeout

Table 8: Measurement outcomes

Each validation task was executed on the DSL presented in this paper three times for both the Z3 SMT solver and the Alloy Analyzer (with SAT4j-solver), then the median of the execution times was calculated. The measures are executed with a 5 minute timeout on an average personal computer[1]. Execution times are presented in seconds.

9.2.2 Evaluation of Language-level Validation

Consistency Analysis First, consistency analysis of the full DSL (without the use of partial snapshots) is executed for arbitrary model size ($|M| = *$), then it is repeated for exactly 10 and 100 number of objects. The results are presented in Table 9. As the results show, consistency analysis is easily solved with each reasoning approach for small models. The Alloy Analyzer is unable to solve large models (with $\approx$ 100 elements), while the SMT-solver easily solves consistency check for larger size by generating highly symmetric models.

These measurements indicate that consistency analysis on the language level (i.e. without PSs) is an easy DSL validation task as the output models retrieved as consistency proofs are trivial in most practical cases (e.g. consisting of a single model element).

[1] CPU: Intel Core-i5-m310M, MEM: 16GB, OS: Windows 8.1 Pro, Reasoners: Alloy Analyzer 4.2 and Z3 4.3.0.

	$\|M\| =$		
	* [✓]	10 [✓]	100 [✓]
Z3, int=$\mathbb{Z}$	0.02	1.85	0.1
Alloy, int=undef	0.06	0.26	-

Table 9: Consistency check measurements

Derived Features Next, we checked completeness and unambiguity for derived features type and model, and the measurement results are summarized in Table 10. The complete validation problem was given to the SMT solver, while we restricted the analysis to at most 10 model elements and a [-64; 63] interval for integers in case of Alloy. This is an underapproximation due to the limited expressive power of SAT problems, therefore the UNSAT result from the Alloy SAT-solver can not be used as a proof.

Since completeness and unambiguity still leads to a real theorem proving problem, the SMT solver excels in these cases (both for proving correctness without assumptions and retrieving counterexamples). Furthermore, the lack of counter-examples in case of SAT solvers is still not a general proof due to the bounded context.

Subsumption and equivalence checks We also carried out subsumption checks to decide if a certain constraint (captured in GP, OCL or PS formalism in the different cases) is already covered by the DSL specification. Measurement results are summarized in Table 11. The SMT solver is particularly good for proving subsumption (3rd case) but it also has predictable runtime for the negative cases (1st and 2nd). It is interesting to note that FOL formulae derived from OCL constraints required more time, which might indicate some inefficiencies in our OCL transformation.

We also aimed to prove equivalence for certain constraints captured in different formalisms (graph patterns vs. OCL invariants; partial snapshots vs. graph patterns or OCL invariants). Measurement results are listed in Table 13. In this validation setup, Alloy also performed well - but the SMT solver still had a predictable runtime.

Our experiments to carry out various language-level validation tasks clearly indicates that the Z3 SMT solver outperforms the SAT-based Alloy reasoner tools, which coincides with our a priori expectations.

9.3 Model Generation Evaluation

In DSL validation properties are decided by detecting contradicting requirements or providing small examples. Valid instance models of increasing size are generated to measure its efficiency.

By customizing the validation context, our approach generates various instance models with designated properties. To avoid empty and symmetric models, the generation processes are executed with three PSs as input: architecture, shareable communication and unmodifiable module which are defined in Figure 8, so the results are similar to the model in Figure 19. Models are generated with 10 to 20 objects, where the smallest model (with 10 objects) is too small to contain each PS, but with 11 or more objects the problem is satisfiable. The results are presented in Table 13.

In the first series of measurements (with arbitrary integers), the Z3 solver generated models up to 16 elements, with increasing execution times, while the Alloy Analyzer was unable to initialize. The reason of the Alloy failure is that the unmodifiable module contains large integers (around 500), which is difficult for SAT-solvers.

In the second series of measurements, the minimalFrequency values are reduced to 2 and 4 (from 250 and 500), and the range of the integers is reduced to the $[-64; 63]$ interval for both solvers. With this integer range, the two solvers produced models after about the same runtime, but the Alloy Analyzer ran out of memory for models with over 14 elements. Note that using a integer limit decreased the efficiency of model generation for the Z3 solver.

When the interval of the integers is further reduced, the Alloy Analyzer clearly outperforms the Z3 solver. A third series of measurement were executed, where the integers are removed from the DSL, only objects and enums are present. In this case the Alloy Analyzer has close to zero runtime, and even models with 80 objects can be generated within 145 seconds. The Z3 solver also generates models without integers with higher efficiency.

Our measurements indicate that SMT-solvers are strong in proving language-level properties and handling integer attributes, while SAT-solvers can generate larger instance models as witness or counter-example. Part of our future work will be directed to combine the strengths of the two approaches.

10 Related Work

In Model-Driven Engineering language analysis and validation has become a very hot topic, especially in the safety critical design and development domain (e.g., DO-178C [48] for the civil avionics domain) that accepts formal verification as certification artifacts. In the current section we provide an insight to similar approaches in a broader research scope.

	type		type w. error		model	
	Comp [✓]	Unamb [✓]	Comp [×]	Unamb [×]	Comp [×]	Unamb [✓]
Z3, Complete	0.27	0.03	0.70	0.08	0.02	0.01
Alloy, $\|M\| \leq 10$, int=$[-64;63]$	20.65 [?]	15.23 [?]	24.75	23.54	29.06	30.13 [?]

Table 10: Derived feature validation measurements

	DSL $\models$ Freq		DSL $\models$ T&I		DSL $\models$ Half	
	GP [×]	OCL [×]	GP [×]	OCL [×]	GP [✓]	PS [✓]
Z3, Complete	0.46	1.50	0.85	1.50	0.01	0.02
Alloy, $\|M\| \leq 10$, int=$[-64;63]$	0.27	0.18	33.26	32.85	22.58 [?]	24.16 [?]

Table 11: Subsumption check measurements of DSL constraints

	Freq [✓] GP ⇔ OCL	T&I [✓] GP ⇔ OCL	Half [✓] GP ⇔ PS	T&I - OCL ⇔ Half - PS [×]
Z3, Complete	0.04	0.03	0.02	1.14
Alloy, $\|M\| \leq 10$, int=$[-64;63]$	-	1.40 [?]	1.11 [?]	0.40

Table 12: Equivalence check measurements of DSL constraints

Metamodeling framework validation Formula[31] is a tool for validating DSLs, which takes a metamodel, a partial instance model and a set of constraints and rewriting rules as input, and it aims to extend the partial instance model so that the dedicated state can be reached from it by applying the rewriting rules. As a technological difference, our tool is compliant with standard Eclipse based technologies, while Formula uses its own modeling language. Most validation tasks identified in the paper are not yet supported in Formula, which is specialized in reachability and consistency checks. The Formula tool also uses the Z3 SMT-solver as underlying engine.

Clafer [7] is a lightweight structural modeling language used for feature modeling with minimalistic syntax and rich semantics equivalent to first-order relational logic. The specification language supports structural modeling, constraints (well-formedness constraints are written in their own language, which is said to be equivalent to FOL) and also partial configurations. Partial configurations are like partial snapshots in our approach: instance models with undefined attributes and features that can be the basis of model completion. DSL specification given in Clafer are validated using the Clafer Tools [4] that supports various tasks for domain engineering, like consistency checking and instance model generation based on backend reasoners like Alloy or Choco [1,37]. The provided solution for model completion (ClaferIG - Instance Generator) for structural requirements and another solution for model optimization (ClaferMOO - Multi-Objective Optimizer) [42] for attributed models to find a set of Pareto-optimal model instances based on given a set of optimization objectives.

The main difference between the Clafer and our approach is that we support EMF as our metamodeling language compared to the Clafer specification language, which is only supported by their own framework. However, one interesting feature of the Clafer tooling is that it uses two different tools for the structural and attribute rule validation therefore it might scale better in case of complex DSLs and thus is one of our future goal to adapt such approach.

Validation of OCL enriched metamodels There are several approaches and tools aiming to validate models enriched with OCL constraints [27] relying upon different logic formalisms such as constraint logic programming [17,18,13], SAT-based model finders (like Alloy) [53,3,15,36,54], first-order logic [8], constructive query containment [45], higher-order logic [12,28], or rewriting logics [20]. Some of these approaches (like e.g. [18, 15,36,53]) offer bounded validation (where the search space needs to be restricted explicitly) in order to execute the validation and thus results can only be considered within the given scope, others (like [12,8]) allow unbounded verification (which normally results in increased level of interaction and decidability issues).

One of the most relevant mapping from a subset of OCL into first order logic (OCL2FOL) is presented in [21], that proposes an approach using theorem provers and SMT solvers to automatically check the unsatisfiability of non-trivial sets of OCL constraints without generating the SMT code. In [23] the authors present

	$\|M\| =$										
	10 [×]	11 [✓]	12 [✓]	13 [✓]	14 [✓]	15 [✓]	16 [✓]	17 [✓]	18 [✓]	19 [✓]	20 [✓]
Z3, int=$\mathbb{Z}$	8.29	11.16	22.13	26.31	194.93	236.15	363.39	-	-	-	-
Alloy, int≈500	-	-	-	-	-	-	-	-	-	-	-
Z3, int=$[-64; 63]$	24.31	41.81	31.03	83.24	125.32	235.29	416.71	357.33	-	-	-
Alloy, int=$[-64; 63]$	29.85	33.46	38.00	68.54	-	-	-	-	-	-	-
Z3, int=$\emptyset$	20.80	8.81	11.87	18.2	23.43	38.24	41.71	51.02	57.01	77.29	94.91
Alloy, int=$\emptyset$	0.33	0.22	0.24	0.21	0.20	0.20	0.21	0.46	0.32	0.31	0.32

Table 13: Model generation with increasing size

the extension of their previous mapping, called OCL2FOL$^+$, which deals with a four-valued logic defined in the OCL standard that is not yet supported by our approach. These works support consistency checking between a set of OCL invariants, while our approach aims to deal with the whole specification and is able to detect inconsistencies between the different DSL artefacts.

In [22] the transformation is done in a reverse direction, but includes similar approach for mapping the Alloy language elements to UML and OCL. The main difference between our work and this solution is that the engineer should use Alloy to formalize the model and do the V&V tasks, while we allow the usage of pure EMF and OCL. As a key difference is that our work covers (1) multiple inheritance in metamodels and (2) handling of float arithmetics while the rest of OCL coverage is similar. Their work can better exploit some higher order features in Alloy to capture OCL constructs like size(), min() - where our approach can only provide an approximation.

In [34] a mapping from UML and OCL to Relational Logic Formulas is presented. As a difference our paper covers (1) multiple inheritance in metamodels and (2) the new transitive closure construct in OCL by approximation (3) handling of float/double arithmetics. On the other hand, their approach covers equality between strings and other collections like Bags. Some technical details of their mapping relies upon advanced language features available in Alloy, which we do not use as being independent of target back-end solvers.

The [19] presents a mapping from UML models enriched with OCL formulas to CSP, but the main goal is the consistency check and provides formal verification for the models. Additionally their solution provides similar consistency checks and formal verifications (like subsumption, equivalence and advanced consistency checks). As a difference, our approach handles multiple inheritance, and approximate transitive closure, but they support higher order OCL constructs like size(), min() or max().

Additionally, we proposed a translation [9] of a subset of OCL to graph patterns to provide a effective model validation on the instance level as opposed to the current work, which aims DSL specification validation. There are also mappings from programming languages extended with OCL constraints to reasoners such as Testera [33] (from Java to Alloy) and Pex [47] (from C# to Z3).

Analysis of model and graph transformations SMT solvers have also been used to verify declarative ATL transformations [14] allowing the use of an efficiently analyzable fragment of OCL [21]. The main advantage of using SMT solvers is that it is refutationally complete for quantified formulas of uninterpreted and almost uninterpreted functions and efficiently solvable for a rich subset of logic. Our approach uses SMT-solvers both in a constructive way to find counterexamples (model finding) as well as for proving theorems. In case of using approximations for rich query features, our approach converges to bounded verification techniques.

Graph constraints captured as a subset of graph transformation rules are used in [59] as means to formalize a restricted class of OCL constraints in order to find valid model instances by graph grammars. An inverse approach is taken in [16] to formalize graph transformation rules by OCL constraints as an intermediate language and carry out verification of transformations in UML-to-CSP tool. These approaches mainly focus on mapping core graph transformation semantics, but does not cover many rich query features of the EMF-IncQuery language (such as transitive closure and recursive pattern calls). Many ideas are shared with approaches aiming to verify model transformations [16,38,14], as they built upon the semantics of source and target languages to prove or refute properties of the model transformation. However, the validation tasks identified in the paper are different from the verification challenges of model transformations.

Model extensions using partial models The idea of using *partial models*, which are extended to valid models during verification also appears in [52,31,35]. These initial hints are provided manually to the verification process, while in our approach, these models are assembled from a previous (failed) verification run by adding partial snapshots of the spurious counterexamples or increase the level of approximation. [36] presents an approach for the completion of partial snapshots where OCL constraints have to be satisfied. Instead of creating new PS notation the structure is defined by a concrete model and the relaxed properties are specified by dedicated invariants and queries.

Partial models also share certain similarity with uncertain models, which offer a rich specification language [25] amenable to analysis by the Alloy Analyzer [50]. Uncertain models provide a richer language for partial snapshots for a different purpose: to document semantic variation points generically for instance models. Different potential system models are then synthesized by Alloy accordingly as design decisions. However, their formalism does not support the instantiation of abstract classes, while semantic modifiers are defined individually for model elements (and not for entire snapshots).

However, any approximations are only used in [32] to propose a type system and type inference algorithm for assigning semantic types to constraint variables to detect specification errors in declarative languages with constraints. The PSs are constructed from fully specified instance models in a similar way. However, we additionally propose semantic modifiers which simplifies the specification of complex partial snapshots. On the technological level, our approach handles standard EMF models.

11 Conclusions and Future Work

In this paper, we presented a validation technique for domain-specific language specifications by a transforming to a first-order logic formulae. The main added value of our approach is to cover rich DSL constructs such as derived features and well-formedness constraints captured in declarative languages such as graph patterns and OCL invariants. We identified several validation tasks (such as consistency, completeness, unambiguity, subsumption and equivalence checks) which are relevant in a DSL validation context. We also proposed a workflow to systematically address these validation tasks for a DSL. We also enhanced this context with partial snapshots to capture further (instance-level) assumptions on valid models. Our mapping tries to transform language features into a decidable fragment of first-order logic (called effectively propositional logic), and to handle language features which cannot be represented in FOL, we proposed powerful approximations.

Our approach is supported by a prototype tool integrated into Eclipse, which takes EMF metamodels, instance models, EMF-IncQuery graph patterns and OCL constraints as input to carry out DSL validation. When an output model is derived as a witness or counterexample, this model is back-annotated to the DSL tool itself so that language engineers could observe the source of the problem in their own language. We provided initial experimental evaluation by using two back-end reasoners, namely, Alloy and the Z3 SMT solver for DSL validation purposes. Our prototype tool was successfully used in two major projects, for validating a DSL from the avionics domain, and for test generation purposes for autonomous robots.

As future work, we primarily aim at investigating if back-end reasoners can be combined for the model generation purposes. Our experiments also highlighted that Alloy is strong at generating the structure of a model while SMT solvers excel at completing attributes for different primitive types. Their combined use could significantly enhance the validation process. For test generation purposes, one frequently needs to synthesize test cases where the values lie on some boundaries. Adapting our approach for such a case is part of our future plans. Finally, we plan to investigate the applicability of the approach in the context of the new DO-178C certification standard [48] for civil avionics software development that accepts formal validation as certification artifacts to obtain certification credits for graph patterns or OCL invariants, for instance.

References

1. Choco. `http://www.emn.fr/z-info/choco-solverp`
2. R3-cop (resilient reasoning robotic co-operative systems). ARTEMIS project n° 100233, `http://http://www.r3-cop.eu/`
3. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Softw. Syst. Model. **9**(1), 69–86 (2010)
4. Antkiewicz, M., Bak, K., Murashkin, A., Olaechea, R., Liang, J., Czarnecki, K.: Clafer tools for product line engineering. In: SPLC. Tokyo, Japan (2013)
5. ARINC - Aeronautical Radio, Incorporated: A653 - Avionics Application Software Standard Interface. `http://www.aviation-ia.com/standards`
6. AUTOSAR Consortium: The AUTOSAR Standard. (2013). `http://www.autosar.org/`
7. Bak, K., Czarnecki, K., Wasowski, A.: Feature and metamodels in clafer: Mixed, specialized, and coupled. In: 3rd International Conference on Software Language Engineering. Eindhoven, The Netherlands (2010). DOI 10.1007/978-3-642-19440-5_7
8. Beckert, B., Keller, U., Schmitt, P.H.: Translating the Object Constraint Language into first-order predicate logic. In: Proc of the VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark (2002)
9. Bergmann, G.: Translating OCL to Graph Patterns. In: ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, MODELS 2014. Springer, Springer, Valencia, Spain (2014). Accepted.

10. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF Models. In: MODELS'10, *LNCS*, vol. 6395. Springer (2010)
11. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A graph query language for emf models. In: J. Cabot, E. Visser (eds.) Fourth International Conference on Theory and Practice of Model Transformations, *LNCS*, vol. 6707, pp. 167–182. Springer (2011)
12. Brucker, A.D., Wolff, B.: The HOL-OCL tool (2007). `http://www.brucker.ch/`
13. Büttner, F., Cabot, J.: Lightweight string reasoning for OCL. In: A. Vallecillo, J.P. Tolvanen, E. Kindler, H. Störrle, D.S. Kolovos (eds.) Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Lyngby, Denmark, July 2-5, 2012. Proceedings, *LNCS*, vol. 7349, pp. 244–258. Springer (2012)
14. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: Proc. of the 15th Int. Conf. on MODELS, *LNCS*, vol. 7590 (2012)
15. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: 14th International Conference on Formal Engineering Methods,ICFEM'12, pp. 198–213. LNCS 7635, Springer (2012)
16. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: A UML/OCL framework for the analysis of graph transformation rules. Softw. Syst. Model. **9**(3), 335–357 (2010)
17. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), pp. 547–548. ACM, New York, NY, USA (2007). DOI 10.1145/1321631.1321737. URL `http://doi.acm.org/10.1145/1321631.1321737`
18. Cabot, J., Clariso, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on, pp. 73–80 (2008). DOI 10.1109/ICSTW.2008.54
19. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software **93**, 1–23 (2014)
20. Clavel, M., Egea, M.: The ITP/OCL tool (2008). `http://maude.sip.ucm.es/itp/ocl/`
21. Clavel, M., Egea, M., de Dios, M.A.G.: Checking unsatisfiability for OCL constraints. ECEASST **24** (2009)
22. Cunha, A., Garis, A., Riesco, D.: Translating between alloy specifications and uml class diagrams annotated with ocl. Software & Systems Modeling pp. 1–21 (2013)
23. Dania, C., Clavel, M.: OCL2FOL+: Coping with undefinedness. In: J. Cabot, M. Gogolla, I. Ráth, E.D. Willink (eds.) OCL@MoDELS, *CEUR Workshop Proceedings*, vol. 1092, pp. 53–62. CEUR-WS.org (2013). URL `http://dblp.uni-trier.de/db/conf/models/ocl2013.html#DaniaC13`
24. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'08/ETAPS'08, pp. 337–340. Springer-Verlag (2008)
25. Famelis, M., Salay, R., Chechik, M.: Partial models: Towards modeling and reasoning with uncertainty. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 573–583. IEEE Press, Piscataway, NJ, USA (2012). URL `http://dl.acm.org/citation.cfm?id=2337223.2337290`
26. Ge, Y., Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: A. Bouajjani, O. Maler (eds.) Computer Aided Verification, *LNCS*, vol. 5643, pp. 306–320. Springer Berlin Heidelberg (2009). DOI 10.1007/978-3-642-02658-4_25. URL `http://dx.doi.org/10.1007/978-3-642-02658-4_25`
27. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Softw. Syst. Model. **4**(4), 386–398 (2005)
28. Grönniger, H., Ringert, J.O., Rumpe, B.: System model-based definition of modeling language semantics. In: Formal Techniques for Distributed Systems, *LNCS*, vol. 5522, pp. 152–166. Springer (2009)
29. Horváth, Á., Hegedüs, Á., Búr, M., Varró, D., Starr, R.R., Mirachi, S.: Hardware-software allocation specification of ima systems for early simulation. In: Digital Avionics Systems Conference (DASC). IEEE, IEEE, Colorado Springs, Colorado, US (2014)
30. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002). DOI http://doi.acm.org/10.1145/505145.505149
31. Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Reasoning about metamodeling with formal specifications and automatic proofs. In: Proc. of the 14th Int. Conf. on MODELS, *LNCS*, vol. 6981, pp. 653–667 (2011)
32. Jackson, E.K., Schulte, W., Bjørner, N.: Detecting specification errors in declarative languages with constraints. In: Proc. of the 15th Int. Conf. on MODELS, *LNCS*, vol. 7590, pp. 399–414 (2012)
33. Khurshid, S., Marinov, D.: Testera: Specification-based testing of java programs using sat. Automated Software Engineering **11**(4), 403–434 (2004). DOI 10.1023/B:AUSE.0000038938.10589.b9. URL `http://dx.doi.org/10.1023/B%3AAUSE.0000038938.10589.b9`
34. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back, *Lecture Notes in Computer Science*, vol. 7590. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-33666-9_27. URL `http://dx.doi.org/10.1007/978-3-642-33666-9_27`
35. Kuhlmann, M., Gogolla, M.: Strengthening SAT-based validation of UML/OCL models by representing collections as relations. In: European Conf. on Modelling Foundations and Applications, *LNCS*, vol. 7349, pp. 32–48 (2012)
36. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into use. In: TOOLS'11 - Objects, Models, Components and Patterns, *LNCS*, vol. 6705, pp. 290–306 (2011)
37. Liang, J.: Solving clafer models with choco (GSDLab-TR 2012-12-30) (2012)
38. Lucio, L., Barroca, B., Amaral, V.: A technique for automatic validation of model transformations. In: Proc. of the 13th Int. Conf. on MODELS, *LNCS*, vol. 6394, pp. 136–150 (2010)
39. Mathworks: Matlab Simulink - Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/`
40. Micskei, Z., Szatmári, Z., Oláh, J., Majzik, I.: A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In: G. Jezic, M. Kusek, N.T. Nguyen, R. Howlett, L. Jain (eds.) Agent and Multi-Agent Systems. Technologies and Applications, *LNCS*, vol. 7327, pp. 504–513. Springer Berlin / Heidelberg (2012). DOI 10.1007/978-3-642-30947-2_55
41. The Object Management Group: Object Constraint Language, v2.0 (2006). `http://www.omg.org/spec/OCL/2.0/`
42. Olaechea, R., Stewart, S., Czarnecki, K., Rayside, D.: Modeling and multi-objective optimization of quality attributes

in variability-rich software. In: International Workshop on Non- functional System Properties in Domain Specific Modeling Languages. Innsbruck, Austria (2012)

43. Oszkár Semeráth: Validation of Domain Specific Languages (2013). Technical Report, `https://incquery.net/publications/dslvalid`
44. Piskac, R., de Moura, L., Bjorner, N.: Deciding effectively propositional logic with equality (2008). Microsoft Research, MSR-TR-2008-181 Technical Report
45. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. Data Knowl. Eng. **73**, 1–22 (2012)
46. Ráth, I., Hegedüs, A., Varró, D.: Derived features for EMF by integrating advanced model queries. In: A. Vallecillo, J.P. Tolvanen, E. Kindler, H. Störrle, D. Kolovos (eds.) Modelling Foundations and Applications, *LNCS*, vol. 7349, pp. 102–117. Springer Berlin / Heidelberg (2012). DOI 10.1007/978-3-642-31491-9_10
47. Microsoft Research: Pex. `http://research.microsoft.com/projects/pex/`
48. of RTCA, S.C.: DO-178C, software considerations in airborne systems and equipment certification (2011)
49. SAE - Radio Technical Commission for Aeronautic: Architecture Analysis & Design Language (AADL) v2, AS-5506A, SAE International, 2009
50. Salay, R., Famelis, M., Chechik, M.: Language independent refinement using partial modeling. In: J. de Lara, A. Zisman (eds.) Fundamental Approaches to Software Engineering, *Lecture Notes in Computer Science*, vol. 7212, pp. 224–239. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-28872-2_16. URL `http://dx.doi.org/10.1007/978-3-642-28872-2_16`
51. Semeráth, O., Horváth, Á., Varró, D.: Validation of derived features and well-formedness constraints in dsls - by mapping graph queries to an smt-solver. In: MODELS - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings, pp. 538–554 (2013)
52. Sen, S., Mottu, J.M., Tisi, M., Cabot, J.: Using models of partial knowledge to test model transformations. In: 5th Int. Conf. on Theory and Practice of Model Transformations, *LNCS*, vol. 7307, pp. 24–39 (2012)
53. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, pp. 1–10. ACM (2009)
54. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using boolean satisfiability. In: Design, Automation and Test in Europe, (DATE'10), pp. 1341–1344. IEEE (2010)
55. The Eclipse Project: Eclipse Modeling Framework. `http://www.eclipse.org/emf`
56. The Eclipse Project: Zest. `http://www.eclipse.org/gef/zest/`
57. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming **68**(3), 214–234 (2007)
58. Willink, E.D.: An extensible OCL virtual machine and code generator. In: Proc. of the 12th Workshop on OCL and Textual Modelling, pp. 13–18. ACM (2012)
59. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. ENTCS **211**(0), 159 – 170 (2008). DOI 10.1016/j.entcs.2008.04.038. Proc. of the 5th Int. Workshop on Graph Transformation and Visual Modeling Techniques
60. yEd Graph Editor: yED. `http://www.yworks.com/en/products_yed_about.html`

A Appendix: Syntax of the Concrete Solvers

Types

Expression	Mathematical	SMT2 standard	Alloy
Boolean $\rightarrow$	$\{true, false\}$	`Bool`	`one sig True, False extends Bool {}`
Integer $\rightarrow$	$\mathbb{Z}$	`Int`	`Int`
Real $\rightarrow$	$\mathbb{R}$	`Real`	–
Set $\rightarrow$	S	`(declare-sort S)`	`sig S {}`
interpreted $\mid$	$S = \{e_1, \ldots, e_n\}$	`(declare-datatypes () ((S e!1 ... e!n)))`	`enum S {e1, ... , en}`

Symbolic Value Declaration and Definition

Expression	Mathematical	SMT2 standard	
Function $\rightarrow$	$f : type_1 \times \ldots \times type_n \rightarrow type$	`(declare-fun f (type!1 ... type!n) type)`	as a relation
interpreted $\mid$	$f(x_1, \ldots, x_n) = y$	`(define-fun f (type!1 ... type!n) type y)`	`fun f[x1: Type1, ... , xn: Typen] : one type {y}`
Constant $\rightarrow$	$c : \emptyset \rightarrow type$	`(declare-fun c () type)`	`one sig Constants{c:one type}`
Relation $\rightarrow$	$R : type_1 \times \ldots \times type_n$ $\rightarrow \{true, false\}$	`(declare-fun R (type!1 ... type!n) Bool)`	binary: `R: set type2`, defined in `sig type1 {...}` non binary: `sig R {v1: one type1, ..., vn one typen }`

Terms and Formulae

Expression	Mathematical	SMT2 standard	
Formula $\rightarrow$	$relation(term_1, \ldots, term_n)$	`(relation term!1 ... term!n)`	binary: `term2 in type2.R` non binary: `some r:R{term1=r.v1 and...and termn=r.vn}`
$\mid$	$\neg formula$	`(not formula)`	`not formula`
$\mid$	$formula_1 \wedge formula_2$	`(and formula!1 formula!2)`	`formula1 and formula2`
$\mid$	$formula_1 \vee formula_2$	`(or formula!1 formula!2)`	`formula1 or formula2`
$\mid$	$formula_1 \Rightarrow formula_2$	`(=> formula!1 formula!2)`	`formula1 implies formula2`
$\mid$	$term_1 = term_2$	`(= term!1 term!2)`	`term1 = term2`
$\mid$	$term_1 \neq term_2$	`(distinct term!1 term!2)`	`not(term1 = term2)`
$\mid$	$\exists \quad var_1 \in type_1, \ldots,$ $var_n \in type_n : formula$	`(exists ((var!1 type!1) ... (var!n type!n)) formula)`	`some var1: type1,..., varn: typen {formula}`
$\mid$	$\forall \quad var_1 \in type_1, \ldots,$ $var_n \in type_n : formula$	`(forall ((var!1 type!1) ... (var!n type!n)) formula)`	`all var1: type1,..., varn: typen {formula}`
Term $\rightarrow$	$function(term_1, \ldots, term_n)$	`(function term!1 ... term!n)`	`function[term1, ..., termn]`
$\mid$	(Variable) var_1	`var!1`	`var1`
$\mid$	(Individual) e_1	`e!1`	`e1`

Fig. 28: Mathematical, SMT2 Standard and Alloy syntax of first order logic