Towards User-Friendly and Efficient Analysis with Alloy

Xiaoliang Wang, Adrian Rutle and Yngve Lamo
Bergen University College,
P.O. Box 7030, N-5020, Bergen, Norway
Email:xwa,aru,yla@hib.no

***Abstract*—In model-driven engineering, structural models represent software at the early phases of software development. They are assumed to generate the models in subsequent phases which finally result in software. Thus, it is important to make sure these models are correct w.r.t. different concerns, e.g., consistency, or lack of redundant constraints. In this paper, we present a bounded verification approach using Alloy and integrate it into a graphical modelling tool. The graphical models and the properties to be verified are automatically transformed to Alloy specifications, which are examined by the Alloy Analyzer to verify whether the models satisfy the properties. The verification results are presented as feedbacks in the modelling tool. In this way, a model designer can verify models without knowing the underlying verification techniques and receive user-friendly feedbacks. A challenge in the verification approach is scalability. To tackle this, we present a technique for splitting models into submodels according to their constraints and the properties to be verified. A submodel is left-total if each of its instances can be extended to an instance of the whole model by adding elements typed by the elements that are not in the submodel. The verification of a model is then reduced to the verification of its left-total submodels. We will demonstrate the approach by a running example and we present an experimental result to show that the splitting technique may alleviate the scalability problem.**

I. Introduction

Model-driven engineering (MDE) is a branch of software engineering in which models are the first class entities [1]. In the initial phases of software development, structural models (static models in [2]) specify the structural information of a problem domain. They identify the artifacts and their relationships in the domain. The structure of these models can be represented as graphs; nodes represent the artifacts while edges represent relationships. In addition, the requirements of the domain can be expressed as constraints of these models in different formalisms, e.g., the Object Constraint Language (OCL) [3]. An instance of a structural model is a graph which is well-typed by the underlying graph of the model, and, in addition, satisfies all the constraints of the model.

Usually, structural models are specified using a modelling language, e.g., UML plus OCL, within a modelling tool. However, since manual work may cause errors, these models should be verified to ensure correctness. In addition, in MDE, these models are assumed to generate the models in subsequent phases which then generate software in the final phases by model transformations. Thus, the verification of these models can avoid propagating the errors into the software. Moreover, it is obvious that finding design mistakes as early as in the modelling phase helps to build better software at a lower cost.

Different kinds of correctness properties on structural models are studied in MDE [4]. For instance, *consistency* requests that a model has at least one instance; *lack of redundant constraints* requests that, given a model, there exists no constraint C_1 that can be derived from another constraint C_2, i.e., there exists at least one instance of the model which satisfies C_1 but not C_2. These properties can be categorised into *validity*, i.e., whether all the instances of a model satisfy a property, and *satisfiability*, i.e., whether there exists an instance which satisfies a property [2].

Several approaches and tools have been presented to verify these properties on structural models [2]. Generally, they translate a structural model and the properties to be verified into a specification in some formalism, e.g., Constraint Satisfaction Problem (CSP) [4], [5], Relational Logic [6], [7], etc. Then the specification is analysed by theorem provers or constraint solvers to answer whether the model satisfies the properties. However, since these approaches are not integrated into modelling tools, the model designers have to switch from a modelling tool to a verification tool to use the approaches; they also need background in verification. Moreover, most of the approaches present instances when the properties are satisfied, but give no feedback when the properties are violated. This is not convenient for model designing.

In this paper, we present a bounded verification approach of structural models using Alloy [8] and integrate it into a modelling tool DPF Model Editor [9]. The procedure of the approach is illustrated in Fig. 1. It translates a structural model specified in Diagram Predicate Framework (DPF) [10] and

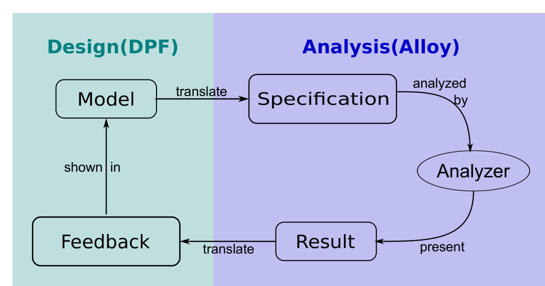


Fig. 1: Workflow for analysing structural models using Alloy

a property to be verified into an Alloy specification. Then, the specification is examined by the Alloy Analyzer to check if the model satisfies the desired property. If the property is satisfied (violated), an instance (counterexample) of the model is generated and displayed in the DPF Model Editor. Otherwise, it means that there are some problems with the model. The problematic part of the model will be highlighted to help the model designer to identify the problem. In this way, the model designer can verify the model under design and receive user-friendly feedback which he can understand, without knowing the underlying verification technique.

The approach is bounded, using a similar strategy as used by other approaches [4], [5], [6], [7] to verify properties specified in undecidable formalisms, e.g, First Order Logic (FOL). It means that the approach finds instances or counterexamples which satisfies or violates properties within a bounded search space. The space is determined by a *scope*, i.e., a user-defined number which restricts the number of instances of each model element in a model. In this way, the undecidability of the underlying formalism is handled. This approach has scalability problems since the search space grows exponentially along with the scope; the verification of large models with a large search space may take long time or become intractable.

To solve this issue, we introduce a technique for splitting models into submodels based on the *factors* of the constraints, i.e., the model elements which are affected by the constraints. We will look for submodels which are *left-total*, i.e. the submodels of which the instances can be extended to be instances of the model. We outline an approach to check whether a submodel is left-total based on *forbidden patterns* of the constraints. That is, these submodels do not contain any match of patterns which are forbidden by any constraints of the model. Then the validation of a model can be reduced to the validation of its left-total submodels.

In order to demonstrate the contributions of this paper, we use a civil status model which modifies the traditional civil status model in [11] originally specified in UML and OCL. For the splitting technique, we run an experiment to compare the performance before and after splitting, the result shows that it can alleviate the scalability problem.

In Section II we present background information, and in Section III the integrated bounded verification approach is illustrated. Then in Sections IV, we present the techniques for splitting models into submodels. Afterwards, we compare our approach with other studies in Section V. Finally, in Section VI, we summarise and discuss some future work.

II. Background

We will start by presenting our running example which describes persons, their relations and features: gender and civil status. In addition, according to the reality, several properties are required in this model, two of which are as following.

p1) each person has exactly one gender

p2) no person can have both a wife and a husband

Based on these requirements, we can define a model in DPF (see Fig. 2) using the DPF Model Editor (The complete example is available at http://dpf.hib.no/downloads/civil.zip).

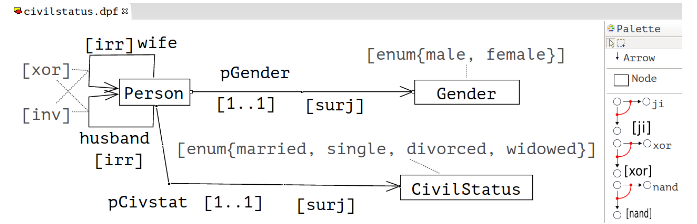


Fig. 2: Civil status model in DPF

DPF provides a formalisation of (meta)modelling based on category theory [12]. This is different from UML-based modelling in which diagrammatic languages are used to specify model structure and the text-based OCL to specify constraints. Moreover, DPF workbench enables development of domain specific languages in a hierarchy where the domain under study can be specified at several abstraction levels [9]. A model in DPF is represented by a diagrammatic specification $\mathfrak{S} = (S, C^{\mathfrak{S}})$ consisting of an underlying graph structure S and a set of constraints $C^{\mathfrak{S}}$. The structure S is a directed graph containing nodes and arrows that represent the artifacts and their relationships in a problem domain. As shown in Fig. 2, the domain artefacts are specified as the nodes **Person**, **Gender**, and **CivilStatus**. The relations between the artefacts are specified as the edges **wife**, **husband**, **pGender** and **pCivstat**.

TABLE I: A sample signature Σ

p	$\alpha^{\Sigma}(p)$	Proposed Visualization	Semantic Interpretation
enum({literals})	1	[enum{literals}] X	$\forall x \in X : x \in \{literals\}$
multi(n,m)	$1 \xrightarrow{f} 2$	X —f [n..m]→ Y	$\forall x \in X : n \leq \lvert f(x) \rvert \leq m \wedge 0 \leq n \leq m \wedge m \geq 1$
surjective	$1 \xrightarrow{f} 2$	X —f [surj]→ Y	$\forall y \in Y, \exists x \in X : f(x) = y$
irreflexive	$1 \circlearrowright f$	X [irr]	$\forall x \in X : f(x) \neq x$
inverse	$1 \underset{g}{\overset{f}{\rightleftarrows}} 2$	X f [inv] g Y	$\forall x \in X$, $\forall y \in Y : y \in f(x)$ iff $x \in g(y)$
xor	$1 \xrightarrow{f} 2$, $1 \xrightarrow{g} 3$	X —f→ Y, X —g→ Z [xor]	$\forall x \in X : (\exists y \in Y : y \in f(x)) \Leftrightarrow (\nexists z \in Z : z \in g(x))$
not-and	$1 \xrightarrow{f} 2$, $1 \xrightarrow{g} 3$	X —f→ Y, X —g→ Z [nand]	$\forall x \in X : \neg(\exists y \in Y : y \in f(x) \wedge \exists z \in Z : z \in g(x))$

Models in DPF also contain constraints which specify required properties. For instance, the property p1 is specified as the constraint [1..1] on the edge **pGender** (ac_1); p2 is specified as the constraint [xor] on the edges **wife** and **husband** (ac_2). These two constraints are *atomic constraints* (AC) which are formulated based on predicates from the signature shown in

TABLE I. The atomic constraints ac_1 and ac_2 are formulated based on `multi(n,m)` and `xor`, respectively. Each predicate has a name p, an arity $\alpha^{\Sigma}(p)$, a parameter list (optional), e.g. `n, m` in the `multi` predicate, and a semantic interpretation which can be specified in different ways. Currently, the semantic interpretation of predicates can be specified in Java, OCL and Alloy. Each AC (p, δ) on a structure S is given by a graph morphism $\delta : \alpha^{\Sigma}(p) \rightarrow S$. A graph I well-typed by S satisfies (p, δ) if the I^* in Fig. 3 satisfies the semantics of p where the diagram is a pullback. The DPF Model Editor has several pre-defined predicates including the ones in TABLE I. But users also can design their own predicates in the Signature Editor [13].

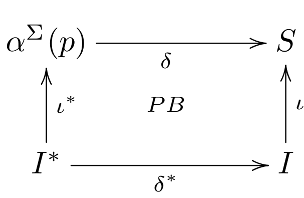


Fig. 3: Semantics of AC

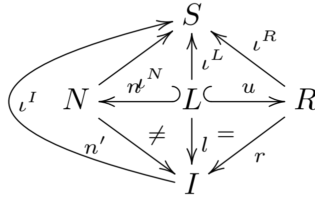


Fig. 4: Semantics of GC

In addition to ACs, graph constraints (GC) may be used to define dependencies among constraints and/or the structures of a model [10]. For example, we further specify that "if a person is married and has no wife, the person should have a husband" as the GC *MarriedWithoutWife* (gc_1) shown in Table II. Each GC $N \xleftarrow{n} L \xrightarrow{u} R$ consists of three graphs: left L, right R and negative application condition N which are typed by the underlying graph S (denoted by $L{:}S$, $R{:}S$ and $N{:}S$ in the table); and two injective graph homomorphisms n and u (see [14], [10]). In Table II, the morphisms are identified by the names, e.g., L embeds **p1** in R and N. The other GCs express properties which are explained by their names. Note that in DPF, GCs are generalised such that L and R can be specifications instead of graphs, see [10]. However, we only consider classical GCs here and leave the general case to future work. A graph I well-typed by S satisfies a GC when for every match l of L in I (i.e. a graph morphism $l : L \rightarrow I$), if there is no match n of N in I where $l = n; n'$, then there must exist a match of R in I where $l = u; r$, outlined in Fig. 4. We require also that a match is compatible with typing. For example, a match of L in a graph I typed by S is a graph morphism $l : L \rightarrow I$ such that $l; \iota^I = \iota^L$.

A model in DPF defines its instances as graphs well-typed by the structure and satisfying all the constraints. We use $I \vDash M$ to denote that the graph I is an instance of the model M. Recall that the model properties to be verified can be categorised into validity and satisfiability; these can be formally expressed as the right two formulae, where M is a structural model and e is a logical expression, respectively.

$$\exists I \vDash M \mid I \vDash e \quad (1)$$

$$\forall I \vDash M \mid I \vDash e \quad (2)$$

TABLE II: GCs applied to instances of the civil status model

$N{:}S$	$L{:}S$	$R{:}S$
MarriedWithoutWife		
p1:Person —w:wife→ p3:Person p1:Person —s:pCivstat→ married:CivilStatus	p1:Person —s:pCivstat→ married:CivilStatus	p1:Person —h:husband→ p2:Person p1:Person —s:pCivstat→ married:CivilStatus
MarriedWithoutHusband		
p1:Person —h:husband→ p3:Person p1:Person —s:pCivstat→ married:CivilStatus	p1:Person —s:pCivstat→ married:CivilStatus	p1:Person —w:wife→ p2:Person p1:Person —s:pCivstat→ married:CivilStatus
HasWifeIsMarried		
	p1:Person —w:wife→ p2:Person	p1:Person —w:wife→ p2:Person p1:Person —s:pCivstat→ married:CivilStatus
HasHusbandIsMarried		
	p1:Person —h:husband→ p2:Person	p1:Person —h:husband→ p2:Person p1:Person —s:pCivstat→ married:CivilStatus

III. Bounded Verification Approach

In this section, we present a bounded verification approach using Alloy. Firstly, we will give a brief introduction to Alloy.

A. Alloy

Alloy consists of a structural modelling language and a constraint solver, the Alloy Analyzer, which are used to specify and analyse specifications. A specification in Alloy consists of *signature*s, which represent artifacts of a domain; each signature may have *field*s which represent relationships; properties about artifacts and relationships are specified as *fact*s in relational logic. An instance of an Alloy specification is a set of *atom*s, the primitive entities in Alloy; each atom belongs to a signature (in modelling terms we say that the atom is typed by the signature); each field of the signature is instantiated by a relation among those atoms. In addition, the atoms and the relations of an instance satisfies all the facts in the specification. Alloy supports also inheritance between signatures. Thus, an atom can be typed by more than one signature if there is inheritance among those signatures.

Specification analysis is performed by the Alloy Analyzer. Given a specification and an expression, the analyzer can examine satisfiability by finding an instance which satisfies the expression. It can also examine validity by using refutation, i.e., validity is falsified if there is an instance which violates the expression, called counterexample. The analysis is bounded: it uses a user-defined *scope* which assigns a number to every top level signature (i.e. signatures which do not inherit from others) that defines the size of the search space. For each instance in the space, the size of each signature s, i.e., the number of atoms typed by s, cannot be larger than $scope(s)$. Thus, the termination of the analysis is guaranteed on the expense of being complete, i.e., the analysis results are valid only if an instance or a counterexample is found within the search space. Another limitation of the Alloy Analyzer, as for other solvers, is scalability. Even a small scope may lead to a large search

TABLE III: Correspondence between DPF Models and Alloy Specifications

DPF	Alloy
X:Node	`sig NX{}`

X:Node with [enum$\{l_1,\ldots,l_n\}$]	`abstract sig NX`
	`lone sig` $Nl_1,\ldots,Nl_n$
	`extends NX{}`

ST:Edge:S:Node $\rightarrow$**T:Node**	`sig EST{` `src:one NS, trg:one NT}`
Constraint	`fact`

space [15]. With a large scope, the analysis will become slow, or even intractable. However, according to the *small scope hypothesis*, most bugs have small counterexamples, thus many applications have used Alloy successfully to examine complex structures [15].

B. Transformation from DPF to Alloy

DPF represents models diagrammatically while Alloy represents models textually. Despite of this difference, there is a similarity between the two approaches. To describe artifacts and relationships in a domain, DPF uses nodes and edges while Alloy uses signatures and fields. From the similarity, we can derive a mapping from DPF models to Alloy specifications shown in Table III. We assume that nodes, edges and GCs in DPF are uniquely named. Moreover, we do not allow a predicate to be applied to the same structure more than once. For example, two surjective predicates on the same arrow is not allowed. That is, for any two ACs (p,δ_1) and (p,δ_2), $\delta_1(\alpha(p)) \neq \delta_2(\alpha(p))$.

```
1  //Signatures of nodes
2  sig NPerson{}
3  abstract sig NGender{}
4  abstract sig NCivilStatus{}
5  lone sig Nmale, Nfemale extends NGender{}
6  lone sig Nmarried, Nsingle, Ndivorced, Nwidowed
       extends NCivilStatus{}
7
8  //Signatures of edges
9  sig Ehusband{src:one NPerson, trg:one NPerson}
10 sig Ewife{src:one NPerson, trg:one NPerson}
11 sig EpGender{src:one NPerson, trg:one NGender}
12 sig EpCivstat{src:one NPerson, trg:one NCivilStatus}
```

Listing 1: Civil status model in Alloy

According to TABLE III, nodes without enumeration predicates are translated into signatures without fields while edges are translated into signatures with two fields `src` and `trg` which encode the source and target nodes. The names for the derived signatures are equal to the names for corresponding nodes (edges) prefixed with `N` (`E`). The prefixes distinguish signatures for nodes and edges. This makes it easier to translate the analysis result from Alloy to DPF, as shown in the sequel. For example, the structure of the civil status model is translated into signatures shown in Listing 1. The node **Person** is translated into `sig NPerson{}`; the edge **wife** is translated into `sig Ewife{src:one NPerson, trg:one NPerson}`. It should be mentioned that the approaches [4], [5], [6], [7] handled edges (association in their works) differently. Associations are encoded as relations between two classes. With their approaches, it is not possible to represent parallel links between two instances. But DPF is a framework to design not only models but also metamodels. Therefore, it is necessary to consider a more general case where parallel edges may exist between two nodes. Nodes with enumeration predicates are handled differently because Alloy does not have a primitive enumeration type. We simulate enumeration by inheritance and restricting the size of a signature in an instance. For example, the two signatures `Nmale` and `Nfemale` extend `NGender`; the keyword `abstract` restricts that, in an instance, no atom is typed by `NGender` except the atoms typed by `male` or `female`; the keyword `lone` restricts the number of instances of `Nmale`, `Nfemale` to at most one. In this way, the signature `NGender` defines an enumeration type with `Nmale` and `Nfemale` as its literals. Similarly, `CivilStatus` defines an enumeration type.

Fig. 5: Specification of the predicate `multi` in the Signature Editor

Constraints (ACs and GCs) from the DPF models are encoded as uniquely named facts in Alloy. ACs are formulated based on predicates; the DPF Workbench enables the designer to specify the semantics of these predicates using the Alloy specification language (see Fig. 5). For example, the semantics of the predicate multi(min,max) can be specified as the expression in the Validator field in Fig. 5. The variables between `$$` refer to 1) elements in the arity, e.g., `X`, `XY`, and 2) parameters of the predicate, e.g., `pmin`, `pmax` (prefixed with `p`). Since each AC (p,δ) is given by a graph morphism $\delta:\alpha^{\Sigma}(p)\rightarrow S$, it is translated into a named fact by replacing the variables referring to elements in $\alpha^{\Sigma}(p)$ with their corresponding signature names and the parameter variables with their values; the fact name is p and the names of arrows

(nodes if no arrows in the predicate) connected with "_". For instance, the AC in Fig. 2 is formulated on the edge **pGender** based on [1..1]; its graph morphism is $\delta = \{X \mapsto$**Person**, $Y \mapsto$**Gender**, $XY \mapsto$**pGender**$\}$; the values for the parameters are $min = max = 1$. The derived fact, on Line 1 in Listing 2, is named as `multi_EpGender` and is instantiated by replacing variables in `$$`, e.g. `$X$` with `NPerson` and `$pmin$` with `1`.

```
1 fact multi_EpGender{all n:NPerson| let count = #{e:
      EpGender|e.src=n}|count>=1 and count <=1}
2 fact MarriedWithoutWife{
3 all p1:NPerson|((some g:EpCivstat|g.src=p1) and
4 not (some w:Ewife|w.src=p1))
5 implies (some h:Ehusband|h.src=p1) }
```

Listing 2: Atomic constraints and GCs in Alloy

For the GCs, a named fact can be derived according to their semantics as follows:

1 fact gc{
2 all $e_1^L{:}T_1^L,\ldots, e_l^L{:}T_l^L$ | $L(e_1^L, \ldots e_l^L)$ and
3 not (some $e_1^{N/L}{:}T_1^{N/L},\ldots, e_n^{N/L}{:}T_n^{N/L}$ | $N(e_1^L, \ldots e_l^L, e_1^{N/L}, \ldots e_n^{N/L})$)
4 implies (some $e_1^{R/L}{:}T_1^{R/L},\ldots, e_r^{R/L}{:}T_r^{R/L}$ | $R(e_1^L, \ldots e_l^L, e_1^{R/L}, \ldots e_r^{R/L})$)}

Line 2 encodes the match of L (l in Fig. 4); Line 3 encodes that for each match l, there is no match of N (n') where $n; n' = l$; Line 4 encodes that for each match l, there is match of R (r') where $r; r' = l$. For example, gc_1 is translated into the Alloy fact as shown on Line 2 - 5 in Listing 2.

C. Alloy Analysis and Presentation of Feedback

Now, we analyse the Alloy specification with the Alloy Analyzer. We will show how the verification of satisfiability and validity properties is integrated into DPF Model Editor by analysis of model consistency and detecting redundant constraints. Moreover, we translate the analysis result and present it in the DPF model editor.

1) Analysis of Model Consistency: The DPF workbench is extended to support consistency checking of DPF models. The designer can choose the context menu `Validate Model Consistency` and the model will be automatically translated into a corresponding Alloy specification as described in Section III-B. The $run\{\}$ command is used to search for an instance of the Alloy specification. Recall that the Alloy Analyzer needs a user-defined scope. In the tool, users can specify the scope by using a configuration in Eclipse. Here we use the scope 3 which is the default scope of Alloy. In the future, we will study how to automatically derive a suitable scope from models.

If the Alloy Analyzer finds an instance of a specification, it visualizes the instance as a graph, e.g., one instance of the civil status model is shown in Fig. 6. The instance contains atoms and relations, e.g., `NPerson={(NPerson0),(NPerson1)},Ehusband={(Ehusband)},Ehusband.trg={(Ehusband1,NPerson0)}` etc. The atoms typed by signatures are visualised as yellow boxes while the relations which instantiate fields of these signatures as arrows between those boxes. In order to make it easier for the designer to interpret the instance, we translate it back to a DPF instance.

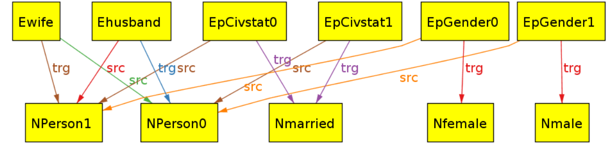


Fig. 6: Instance in Alloy

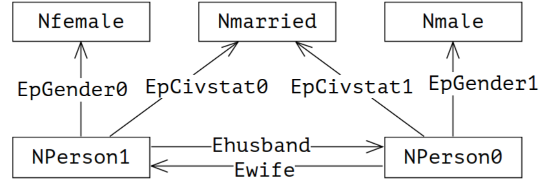


Fig. 7: Instance in DPF

Using Table III, we can derive a mapping from an instance of an Alloy specification (representing a DPF model) to an instance of the DPF model. An atom that is typed by a node signature `NX` (e.g. `NPerson`) is translated back to a node in a DPF instance which is typed by its corresponding node **X** (e.g. **Person**) in the DPF model. In the same way, an atom that is typed by an edge signature (e.g. `EpGender`) is translated back to an edge typed by its corresponding edge in the DPF model (e.g. **pGender**) in the DPF model. Moreover, for each edge in the DPF instance, its source and target nodes are set according to the relations `src` and `trg` in the Alloy instance. If an atom is typed by both `NX` and `NY` where `NX` inherits from `NY`, the atom is translated to a node typed by the node **Y** in the DPF model. According to these rules, the instance in Fig. 6 can be translated into a DPF instance as shown in Fig. 7.

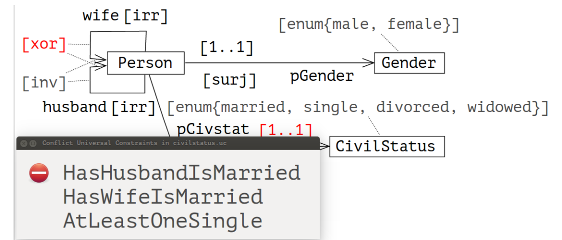


Fig. 8: Highlight the inconsistency cause in DPF model Editor

If the Alloy Analyzer cannot find an instance, it may be that the model is inconsistent. For instance, assume that the designer adds another constraint which ensures that at least one single person exists. The constraint is specified as the GC $singlePerson$; where $N:S$ and $L:S$ are empty while the $R:S$ is [p1:Person] —s:pCivstat→ [single:CivilStatus]. After analysing the model again, the Alloy Analyser finds no instance; instead, it gives information about the part of the specification that cause the inconsistency. For the Alloy Analyzer to give such information, it is required to select a SAT solver with Unsat Core. The information contains the locations of the expressions which contradict each other. We can use the locations to identify the facts in the Alloy specification that contains such

expressions. Recall that the constraints in DPF is translated into uniquely named facts in Alloy. From the name of the facts, we can obtain the corresponding constraints in DPF and highlight them in the DPF Workbench. In this example, the facts *xor_Ewife_Ehusband*, *multi_Ecivstat*, *HasWifeIsMarried*, *HasHusbandIsMarried* and *singlePerson* in the Alloy specification are identified as the cause of the inconsistency. By the name of the facts, we find that the corresponding ACs and GCs in DPF. For instance, the name *xor_Ewife_Ehusband* indicates that the fact is derived from the AC on the edges **wife** and **husband** which is formulated based on the predicate `xor` (see Fig. 8). In the same way, we can find other constraints in DPF, e.g., the multiplicity constraint [1..1] on the edge **pCivstat** (highlighted in red), and the GCs *hasWifeIsMarried*, *hasHusbandIsMarried* and *singlePerson* (shown in message box) are the inconsistency causes. After checking the five constraints, the designer can easily find the problem, e.g, in the civil status model, the atomic constraint [xor] should be replaced with a [nand] constraint which specifies that a person cannot have both a **wife** and a **husband**. After this correction, the model is verified consistent.

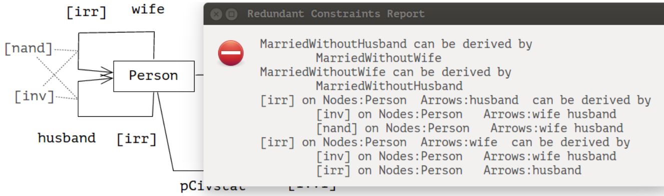


Fig. 9: Redundant constraints feedback in DPF Workbench

2) Detecting Redundant Constraints: Searching for redundant constraints is performed similarly as model consistency analysis. When the designer chooses the context menu `Redundancy Check`, the DPF model is translated to an Alloy specification. Then we use the command `check{c_i }` to check whether the constraint c_i is redundant. Given a model $(S, C^{\mathfrak{S}} = \{c_1, \ldots, c_n\})$, a constraint $c_{n+1} \notin C^{\mathfrak{S}}$ is redundant if every graph that is typed by S and satisfies $\{c_1, \ldots, c_n\}$ also satisfies c_{n+1}. In Alloy, the check can be preformed by a *check*$\{c_{n+1}\}$ command which tries to find a counterexample satisfying $c_1 \wedge \ldots \wedge c_n$ but not c_{n+1}. If such a counterexample is found, it means that the constraint is not redundant. Otherwise, the constraints which can imply c_{n+1} will be reported. The technique to find which constraints that can imply c_{n+1} is the same as to find which constraints make a model inconsistent. For the civil status model, four constraints are found redundant as shown in Fig. 9. The designer may choose to keep or delete these. Notice that the two graph constraints *marriedWithoutHusband* and *marriedWithoutWife* are derived by each other. Only one of the constraints can be deleted otherwise the model will lose information.

In the end, we should emphasise that the approach to analyse models using Alloy is incomplete. If the analyzer cannot find an instance or a counterexample in a certain scope, we can only assure that the model is not consistent or a constraint is not redundant with the given scope.

IV. Model Splitting

In this section, we present a model splitting technique which split a model into submodels. We will show that the verification of a model can be reduced into the verification of a submodel if the submodel is left-total.

A. Left-total submodel

Given a constraint c on a structure S, c can only affect a part S' of S, where $S' \subseteq S$ and c is either an AC or a GC. Due to the pullback construction in Fig. 3 and the semantics of GCs in Fig. 4, this means that for every graph I well-typed by S, we only need to inspect the elements which are typed by S' to decide whether I satisfies c. For example, the AC [surj] on the edge **pGender** affects only [Person] $\xrightarrow{\text{pGender}}$ [Gender] (S_1). Given a graph, e.g., [p:Person] → [s:Civilstatus], only the subgraph typed by S_1, i.e., [p:Person], affects whether the graph satisfies the constraint. We will define this restriction formally as follows.

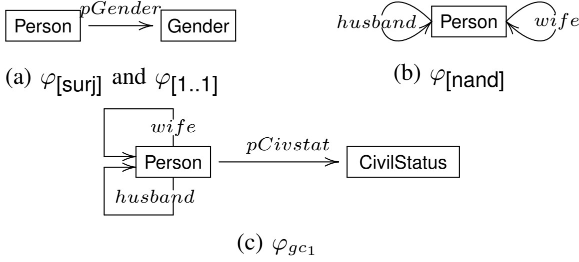
Definition 1 (Restriction)**.** *Given a subgraph S' of a graph S, the restriction of a graph I well-typed by S on S' is the largest subgraph of S which is well-typed by S'. The restriction is denoted as $I{\downarrow_{s'}}$, where s' is the inclusion $s' : S' \to S$. When the inclusion can be identified from the context, we will use $I{\downarrow_{S'}}$ to denote the restriction. Formally, it is the pullback of the diagram $S' \xrightarrow{s'} S \xleftarrow{\iota} I$, where ι is the typing of I, as shown in Fig. 10.*

$$
\begin{array}{ccccc}
S'' & \xhookrightarrow{s''} & S' & \xhookrightarrow{s'} & S \\
\uparrow{\scriptstyle \iota''} & & \uparrow{\scriptstyle \iota'} & & \uparrow{\scriptstyle \iota} \\
I{\downarrow_{(s'';s')}} & \hookrightarrow & I{\downarrow_{s'}} & \hookrightarrow & I
\end{array}
$$

Fig. 10: Restriction

Since pullback are compositional, restrictions are also compositional along inclusions. That is, given a subgraph S'' of S', the restriction of I on S'' equals to the restriction of I on S' then further on S'', i.e., $I{\downarrow_{s'';s'}} = (I{\downarrow_{s'}}){\downarrow_{s''}}$. Given a structure S and two of its subgraphs S_1 and S_2 with the overlapping $S_1 \cap S_2$, for every graph I well-typed by S, $(I{\downarrow_{S_1}}){\downarrow_{S_1 \cap S_2}} = (I{\downarrow_{S_2}}){\downarrow_{S_1 \cap S_2}}$.

Definition 2 (Factor)**.** *Given a constraint c on a structure S, the factor φ_c of the constraint is a subgraph of S which contains all the types referred to in the definition of the constraint. Given an AC $c = (p, \delta)$, its factor φ_c is $\delta(\alpha(p))$; Given a GC gc, its factor φ_{gc} is $\iota^N(N) \cup \iota^L(L) \cup \iota^R(R)$.*



(a) $\varphi_{\text{[surj]}}$ and $\varphi_{\text{[1..1]}}$ (b) $\varphi_{\text{[nand]}}$

(c) φ_{gc_1}

Fig. 11: Factors

Corollary 1. *Given a constraint c on a structure S, its factor φ_c affects whether a graph satisfies the constraint. It means that for each graph I well-typed by S, I satisfies c if and only if $I{\downarrow_{\varphi_c}}$ satisfies c.*

For example, the factors of the four constraints in the civil status model ([surj] and [1..1] on the edge **pGender**, [nand] between the edges **wife** and **husband**, and gc_1) , are shown in Fig. 11. Consider the [surj] on the edge **pGender**, the constraint can be expressed as the FOL expression `∀g:Gender|∃pg: pGender|pg.trg=g`. The types referred to in the expression are the node **Gender** and the edge **pGender**. The factor $\varphi_{\text{[surj]}}$ contains these two types, however, since the factor is a graph, $\varphi_{\text{[surj]}}$ also contains the node **Person**.

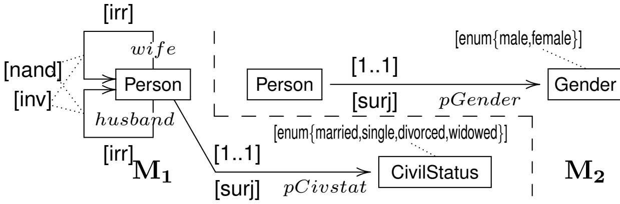
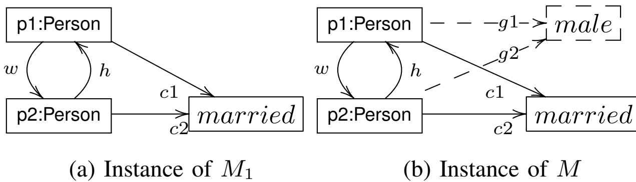


Fig. 12: Submodels

Definition 3 (Splitting)**.** *Given a model $M = (S, C^{\mathfrak{S}})$, a splitting of M, denoted as $M = M_1 \oplus M_2$, are two submodels $M_1 = (S_1, C^{\mathfrak{S}_1})$ and $M_2 = (S_2, C^{\mathfrak{S}_2})$, where $S_1 \cup S_2 = S$. In addition, for every constraint c of each submodel M_i, its factor φ_c is a subgraph of the structure S_i. If the factor of a constraint is the subgraph of both structures, the constraint belongs to both submodels.*

If the factor of one constraint is a subgraph of the factor of another constraint, the two constraints belong to the same submodels, e.g, [surj] and [1..1] on **pGender**. In Fig. 12 the civil status model is split into two submodels according to the factors of its constraints. Notice that, submodels are also models, thus, these can be further split into submodels.



(a) Instance of M_1 (b) Instance of M

Fig. 13: Factors

The instances of submodels may not be instances of the whole model. For example, an instance of the submodel M_1 shown in Fig. 13a is not an instance of M, since it violates [1..1] on **pGender**. But it can become an instance of M by adding elements typed by the types in M_2, e.g., an edge **g1** to **p1**, depicted in dash lines in Fig. 13b. Some submodels are special in that, every instance of these submodels can become an instance of the whole model by adding elements typed by the types in other submodels. Inspired by the term left-total relations, we call these submodels as left-total submodels.

Definition 4 (Left-total model)**.** *Given a submodel $M_1 = (S_1, C^{\mathfrak{S}_1})$ of model M, M_1 is left-total, if for every instance I_1 of the submodel M_1, there exists an instance I of the model M where the restriction of I on S_1 equals to I_1. Formally, it means that $\forall I_1 {\vDash} M_1 \exists I {\vDash} M \mid I{\downarrow_{S_1}} = I_1$.*

A left-total submodel can be used to reduce the verification of the whole model if the factor of the property to be verified is a subgraph of the underlying structure of the submodel, as proved by the following theorem.

Theorem 2. *Verification of a property p on M can be reduced to the verifications of p on M_1 if M_1 is a left-total submodel of M and the factor of p is a subgraph of S_1. Formally, it can be expressed as the following two formulae.*

$$\forall I {\vDash} M \mid I {\vDash} p \quad \Leftrightarrow \quad \forall I_1 {\vDash} M_1 \mid I_1 {\vDash} p \tag{3}$$

$$\exists I {\vDash} M \mid I {\vDash} p \quad \Leftrightarrow \quad \exists I_1 {\vDash} M_1 \mid I_1 {\vDash} p \tag{4}$$

Proof. The proof of formula 4 follows from formula 3.
Proof of formula 3.
$\Rightarrow$ Since M_1 is left-total, for each instance I_1 of M_1, there exists an instance I of M such that $I{\downarrow_{S_1}} = I_1$. Because I satisfies p and $I{\downarrow_{\varphi_p}} = (I{\downarrow_{S_1}}){\downarrow_{\varphi_p}} = I_1{\downarrow_{\varphi_p}}$, I_1 also satisfies p.
$\Leftarrow$ For each instance I of M, I satisfies p because $I{\downarrow_{\varphi_p}} = (I{\downarrow_{S_1}}){\downarrow_{\varphi_p}}$ and $I{\downarrow_{S_1}}$ satisfies p. □

The theorem implies that a verification problem can be reduced when there is a left-total submodel containing the factor of the property to be verified. For example, Given a model $(S, C^{\mathfrak{S}} = \{c_1, \ldots, c_n\})$, to check if a constraint c_{n+1} is redundant, we need to check the formula $\forall I {\vDash} M \mid I {\vDash} c_{n+1}$. We will not verify this on the whole model. Instead, we can verify this on the left-total submodel where the factor of c_{n+1} is a subgraph of the structure of the submodel.

But some properties can be verified on a left-total submodel even if theirs factor is not a subgraph of the structure of the submodel. For instance, consistency is such a special property where the factor of the property is the whole structure. However, given a left-total submodel M_1, M is consistent if and only if M_1 is consistent. If M_1 has an instance, according to Definition 4, M also has an instance. Conversely, if M has an instance, M_1 also has an instance. Hence, the verification of consistency can be performed on a left-total submodel.

B. How to find left-total submodels?

Given a model, we can split the models into submodels based on their factors. Now, the problem is how to find left-total submodels. We find that given two submodels M_1 and

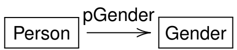
M_2, the key point to find whether M_1 is left-total, is to check every graph well-typed by the overlapping structure of the two submodels. According to Definition 4, M_1 is left-total if and only if every instance I_1 of M_1 can become an instance I of M such that $I{\downarrow_{S_1}} = I_1$. According to the restriction definition, if $I_1{\downarrow_{S_1 \cap S_2}}$ is an instance of M_2, I_1 satisfies all the constraints of M_2. Thus, I_1 can become an instance of M. Otherwise, if $I_1{\downarrow_{S_1 \cap S_2}}$ is not an instance of M_2, we should find an instance I_2 of M_2 where $I_2{\downarrow_{S_1 \cap S_2}} = I_1{\downarrow_{S_1 \cap S_2}}$. Notice that if a graph well-typed by $S_1 \cap S_2$ is not contained by any instance of M_1, we will not bother to consider it. In the following theorem, we will prove the criteria formally.

Theorem 3. *Given a splitting $M = M_1 \oplus M_2$, M_1 is left-total, if and only if, for each graph well-typed by the overlapping of the underlying structures, denoted as $S_1 \cap S_2$, if $G \not\models M_2$ and there is an instance I_1 of M_1 where $I_1{\downarrow_{S_1 \cap S_2}} = G$, then there is an instance I_2 of M_2 where $I_2{\downarrow_{S_1 \cap S_2}} = G$. Formally, it can be expressed as:*

$$\forall G : S_1 \cap S_2 \quad | \quad (G \not\models M_2 \wedge \exists I_1 \models M_1 \mid I_1{\downarrow_{S_1 \cap S_2}} = G) \Rightarrow \exists I_2 \models M_2 \mid I_2{\downarrow_{S_1 \cap S_2}} = G \tag{5}$$

Proof. $\Leftarrow$ For every instance I_1 of M_1, if $I_1 \models M$ then the condition for left total is satisfied. Otherwise, $I_1 \not\models M$. Let $G = I_1{\downarrow_{S_1 \cap S_2}}$. According to the condition, there exists an instance I_2 of M_2 where $I_2{\downarrow_{S_1 \cap S_2}} = G$. Let $I = I_1 \cup I_2$, since $I{\downarrow_{S_1}} = I_1 \models M_1$ and $I{\downarrow_{S_2}} = I_2 \models M_2$, I is an instance of M. Thus, the submodel M_1 is left-total.
$\Rightarrow$ Assume that the formula is false. It means that there is an instance I_1 of M_1 where $G = I_1{\downarrow_{S_1 \cap S_2}}$ is not an instance of M_2. In addition, there is no instance I_2 of M_2 where $G = I_2{\downarrow_{S_1 \cap S_2}}$. Since M_1 is left-total, there exists an instance of M where $I{\downarrow_{S_1}} = I_1$. It also means there is an instance $I' = I{\downarrow_{S_2}}$ of M_2 where $G = I'{\downarrow_{S_1 \cap S_2}}$, which causes contradiction. ☐

To check the criteria, we now consider a more general problem: *given a model M with structure S, whether there exists a **forbidden graph**, i.e. a graph well-typed by S which is not an instance of M and not a subgraph of any instance of M*. To answer this problem, we introduce the notion of *forbidden pattern* and show that whether a submodel M_1 is left-total is depending on the forbidden patterns of M_1 and M_2.

Given a model M with structure S, if a graph is a forbidden graph for c, it means that the graph will never become an instance of M whenever adding elements typed by S. For example, for the multiplicity constraint [1..1] on the edge 

, a forbidden graph is a person having two genders. Constraints may have no forbidden graph. e.g., [surj] on the edge **pGender**. Since a forbidden graph will never be valid although we add correctly typed elements to it, each constraint has either no forbidden graph or infinitely many. In other words, forbidden graphs can be generalised as some patterns. If one of the patterns is present in a graph, the graph is a forbidden graph.

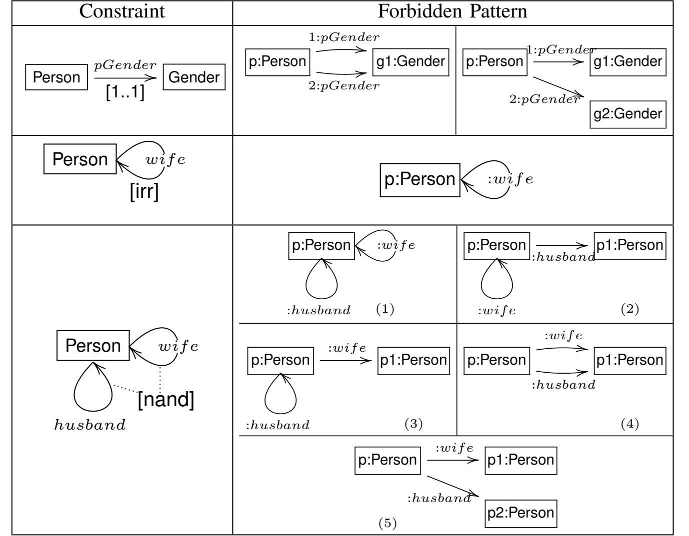


TABLE IV: Forbidden Patterns examples

Definition 5 (Forbidden Pattern)**.** *Each constraint c on a structure S has a set of forbidden patterns where*

1) *each forbidden pattern FP is a forbidden graph FG*
2) *each FG of c has an injective match from one of its forbidden patterns to FG*
3) *for each FP, no other forbidden pattern includes FP*

Forbidden pattern represents forbidden graphs; each forbidden pattern represents a set of forbidden graphs. For example, for the multiplicity constraint [1..1] on the edge **pGender**, the graph that a person has two genders is its only forbidden pattern. It represents all the forbidden graphs where there is a person having more than one gender. The forbidden patterns of the constraints [1..1], [irr] and [nand] are shown in the TABLE IV.

Given a model M with structure S, the forbidden patterns of M is the union of the forbidden patterns of all the constraints on the model. Given a subgraph S' of S, there are forbidden graphs well-typed by S' if and only if there exists one forbidden pattern well-typed by S'. It means that the type graph of one forbidden pattern is a subgraph of S'.

Based on forbidden patterns, the criteria in Theorem 3 can be reduced into several conditions, which are stated in the following theorem.

Theorem 4. *Given a splitting $M = M_1 \oplus M_2$, the submodel M_1 is left-total if and only if one of the following conditions are satisfied:*

1) *the overlapping of the underlying structures is empty, i.e., $S_1 \cap S_2 = \emptyset$*
2) *every graph well-typed by $S_1 \cap S_2$ is an instance of M_2*
3) *there are no instances of M_1*
4) *If M_1 and M_2 have instances, every forbidden pattern of M_2 well-typed by $S_1 \cap S_2$ is also a forbidden pattern of M_1*

According to Theorem 3, the first three conditions are obviously correct. In the following, we just prove the last condition.

Proof. Given a forbidden pattern FP of M_2 which is well-typed by $S_1 \cap S_2$, there is a graph G well-typed by $S_1 \cap S_2$ which is not an instance of M_2. If FP is also a forbidden pattern of M_1, there is no instance I_1 of M_1 where $I_1{\downarrow_{S_1 \cap S_2}} = G$, therefore, the formula 5 is true. Otherwise, there is no instance I_2 of M_2 where $I_2{\downarrow_{S_1 \cap S_2}} = G$, therefore, the formula 5 is false. □

The conditions 1 and 4 can be checked based on the definition of constraints or properties; while the other two conditions can be checked by using of the Alloy Analyzer. It should be noted that, the last condition implies also that, if no forbidden pattern of M_2 is well-typed by $S_1 \cap S_2$, M_1 is a left-total submodel. This can be used to evaluate whether a submodel is left-total according only to the forbidden patterns of M_2. For verification purpose, we can use conditions 1 and 4 to check if M_1 is left-total. In the example, the M_2 has only one forbidden pattern which is not well-typed by **Person**. Thus, M_1 is a left-total submodel.

C. Experimental Results

TABLE V: Experiment Result For Consistent Model

S	M			M'			M_1		
	T	V	T+V	T	V	T+V	T	V	T+V
5	0.097	0.060	0.157	0.762	1.736	2.498	0.168	0.058	0.226
6	0.101	0.043	0.144	0.605	1.685	2.290	0.143	0.104	0.247
7	0.114	0.093	0.207	0.806	1.709	2.515	0.226	0.093	0.319
8	0.122	0.240	0.362	1.220	3.718	4.938	0.203	0.108	0.311
9	0.158	0.328	0.486	1.393	4.046	5.439	0.263	0.066	0.329
10	0.138	0.353	0.491	1.977	9.701	11.678	0.345	0.131	0.476
11	0.203	0.070	0.273	1.683	17.991	19.674	0.300	0.057	0.357
12	0.234	0.085	0.319	2.377	23.906	26.283	0.571	0.320	0.891
13	0.440	1.224	1.664	2.381	34.818	37.199	0.628	0.303	0.931
14	0.344	0.447	0.791	2.595	87.201	89.796	0.626	1.144	1.770
15	0.389	0.688	1.077	3.244	33.779	37.023	0.653	2.981	3.634
16	0.454	0.087	0.541	3.601	121.566	125.167	0.692	0.210	0.902
17	0.529	0.180	0.709	4.892	134.505	139.397	0.742	0.244	0.986
18	0.664	0.267	0.931	5.370	51.201	56.571	0.826	0.139	0.965
19	0.688	0.107	0.795	5.814	366.220	372.034	0.927	0.318	1.245
20	0.888	0.193	1.081	6.480	30.418	36.898	1.375	0.499	1.874

After we find the left-total submodels, we can verify properties on these submodels. For example, when we check the consistency of the civil status model using the Alloy Analyzer, the verification can be performed on the submodel M_1. The experimental results are shown in Table V. It shows the verification performances for each scope (S, from 5 to 20) in seconds: translation time from Alloy to SAT (T), verification time (T) and the total time (V+T), of three models M, M' and M_1. M is the whole model in Fig. 12. M_1 is the left-total submodel by splitting M; From the result of M and M_1, we can see that there is no big improvement after splitting. The reason is that M is almost unchanged compared to M_1; only one edge is different. In order to show the improvement by splitting, we use M' as an artificial model, by adding Person $\xrightarrow{\text{pGenderi}}$ Genderi for $i = 1..100$, to the M. The performance result shows big improvement especially in a large scope; e.g. at scope 20 the time cost is reduced from 36.898s to 1.874s.

The experiment result in TABLE V is for a consistent model. For this model, the Alloy Analyzer will stop when it finds an instance. This is different from the verification of inconsistent models, where the Alloy Analyzer will walk through the whole search space. In order to demonstrate the splitting technique for the two different situations, we run an experiment on the civil model in Fig 2, i.e., the inconsistent version of the model before replacing the constraint [xor] with [nand]. The experimental results are shown in Table VI. M is the inconsistent civil status model; M_1 is the left-total model by splitting M; Again, since M_1 is only slightly different from M, there is no big improvement after splitting the model. In the same way as for the consistent model, we create an artificial model M. The performance result shows a better improvement, e.g., at scope 20 the time cost is reduced from 211.730s to 19.842s.

The two experiment results show that when applying the splitting technique to a model, if the derived left-total model is much smaller than the original model, the performance can be greatly improved, especially at larger scopes.

TABLE VI: Experiment Result For Inconsistent Model

S	M			M'			M_1		
	T	V	T+V	T	V	T+V	T	V	T+V
5	0.037	0.042	0.079	0.458	1.528	1.986	0.089	0.057	0.146
6	0.077	0.097	0.174	0.946	0.682	1.628	0.121	0.156	0.277
7	0.103	0.160	0.263	0.769	1.009	1.778	0.203	0.104	0.307
8	0.088	0.333	0.421	0.862	2.116	2.978	0.199	0.270	0.469
9	0.115	0.317	0.432	1.119	10.970	12.089	0.435	0.605	1.040
10	0.139	0.526	0.665	1.465	23.245	24.710	0.425	0.697	1.122
11	0.139	0.802	0.941	1.606	5.737	7.343	0.231	0.910	1.141
12	0.151	1.069	1.220	1.761	9.427	11.188	0.246	1.290	1.536
13	0.285	1.839	2.124	2.329	9.284	11.613	0.596	1.718	2.314
14	0.316	3.717	4.033	2.678	40.081	42.759	0.355	2.898	3.253
15	0.289	3.922	4.211	2.972	23.798	26.770	0.313	4.745	5.058
16	0.354	5.256	5.610	3.549	22.583	26.132	0.343	5.868	6.211
17	0.447	7.645	8.092	3.963	104.809	108.772	0.429	8.033	8.462
18	0.545	11.691	12.236	4.882	119.565	124.447	0.560	10.995	11.555
19	0.622	12.968	13.590	5.323	266.333	271.656	0.629	15.891	16.520
20	0.704	19.576	20.280	5.839	205.891	211.730	0.703	19.139	19.842

V. Related Work

There exists many approaches which work on verification of structural models in MDE [2]. Generally, these approaches translate a structural model and the properties to be verified into a specification in some formalisms. Then the specification is analysed by an existing tool to answer whether the model satisfies the properties. For example, the authors in [16], [17], [4] translated UML or EMF models into Constraint Logic Programming (CSP) and use the constraint solvers, ECL^{i}PS^e; the works in [18], [19], [20], [21] translated UML models into Alloy specifications, and use the Alloy Analyzer; Ali et al. [22] and Brucker et al. [23] translated UML models into HOL and use the theorem solver Isabelle. Our approach is similar to the ones using Alloy. But we use a different formalization for the structure which can handle general cases; moreover, we do not provide a transformation of FOL expressions into Alloy facts; instead, we provide an interface with which users can specify their constraints in Alloy directly. Most of these

works implemented their approaches but did not integrate the implementation into a modelling tool, except [16] which is integrated into EMF model editor in Eclipse. While our work integrates the verification approach into our modelling tool and thus the model designer avoids switching between modelling tools and verification tools, in the other approaches the model designer specifies models in the modelling tool and uses verification tools to verify the models; afterwards, she has to interpret the verification results back to the corresponding artifacts in the modelling tool. Our approach also hides the underlying verification techniques and provides feedback to the model designer no matter what verification results is given.

Many verification approaches have scalability problems [2]. Our work provides a technique to reduce the verification of a model into the verification of its left-total submodels. The technique splits models into submodels according to the factors of constraints; then the left-total submodels are identified based on the forbidden patterns of the constraints. Shaikh et al. [24] has present a similar splitting algorithm for verification of UML class diagrams with invariants in OCL. This algorithm splits models based on dependencies between classes which are derived from invariants. According to the dependencies, submodels which are trivially satisfiable can be omitted. These submodels are mainly the ones which contain only multiplicity constraints and have no other constraints. In other words, their algorithm splits models mainly based on the multiplicity constraints. In contrast, our technique consider more general constraints. Moreover, we present the technique formally and give a formal prove for it. This can also be used to formally prove Shaikh's algorithm.

VI. Conclusion

We have presented a bounded verification approach of structural models using Alloy. The approach is integrated into our modelling tool. The model designers can verify models under construction with the approach. They can receive user-friendly feedback as to the verification results of some properties, e.g., consistency and lack of redundant constraints without knowing the underlying verification technique. The approach is general in the sense that as long as the language used to specify the constraints can be translated to Alloy, the properties can be verified using the DPF Workbench. We also provide a technique to tackle the scalability problem of the verification approach. According to the factors and the forbidden patterns of the constraints, models can be split into some left-total submodels. The verification of the models can be reduced to the verification of those left-total submodels. We illustrated the work by checking the consistency of a sample model. The experimental results show that the splitting technique alleviates the scalability problem when the left-total submodel is much smaller than whole model.

Currently, to verify models, users must specify a scope for verification. We will study how to derive a suitable scope from constraints. In addition, in this paper, we construct the forbidden patterns manually, but in future work we will study if and how these patterns can be derived automatically.

References

[1] D. C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[2] C. A. González and J. Cabot, "Formal verification of static software models in MDE: A systematic review," *Information & Software Technology*, vol. 56, no. 8, pp. 821–838, 2014.

[3] Object Management Group, *Object Constraint Language Specification*, February 2010, http://www.omg.org/spec/OCL/2.2/.

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

[5] C. A. González, F. Büttner, R. Clarisó, and J. Cabot, "EMFtoCSP: a tool for the lightweight verification of EMF models," in *Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012*, S. Gnesi, S. Gruner, N. Plat, and B. Rumpe, Eds. IEEE, 2012, pp. 44–50.

[6] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to Alloy," *Software and Systems Modeling*, 2009.

[7] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *Software and System Modeling*, vol. 4, no. 4, pp. 386–398, 2005.

[8] Alloy, *Project Web Site*, http://alloy.mit.edu/community/.

[9] Y. Lamo, X. Wang, F. Mantz, Øyvind. Bech, A. Sandven, and A. Rutle, "DPF Workbench: a multi-level language workbench for MDE," in *Proceedings of the Estonian Academy of Sciences*, 2013, pp. 3–15.

[10] A. Rutle, "Diagram Predicate Framework: A Formal Approach to MDE," Ph.D. dissertation, University of Bergen, 2010.

[11] M. Gogolla, F. Büttner, and J. Cabot, "Initiating a benchmark for UML and OCL analysis tools," in *7th TAP*, 2013, pp. 115–132.

[12] M. Barr and C. Wells, *Category Theory for Computing Science (2nd)*. Prentice-Hall, Inc., 1995.

[13] Y. Lamo, X. Wang, F. Mantz, W. MacCaull, and A. Rutle, "DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment," in *Computer and Information Science*, ser. Studies in Computational Intelligence, 2012, vol. 429, pp. 37–52.

[14] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*, ser. Springer, 2006.

[15] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[16] F. Büttner and J. Cabot, "Lightweight string reasoning for OCL," in *8th ECMFA Proceedings*, 2012, pp. 244–258.

[17] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming," in *22nd IEEE/ACM ASE*, 2007, pp. 547–548.

[18] S. M. A. Shah, K. Anastasakis, and B. Bordbar, "From UML to Alloy and Back Again," in *MoDELS Workshops*, ser. LNCS, vol. 6002. Springer, 2009, pp. 158–171.

[19] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to Alloy," *Software and System Modeling*, vol. 9, no. 1, pp. 69–86, 2010.

[20] B. F. B. Braga, J. P. A. Almeida, G. Guizzardi, and A. B. Benevides, "Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method," *ISSE*, vol. 6, no. 1-2, pp. 55–63, 2010.

[21] F. Mostefaoui and J. Vachon, "Verification of Aspect-UML models using alloy," in *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*. ACM Press, 2007, pp. 41–48.

[22] T. Ali, M. Nauman, and M. Alam, "An Accessible Formal Specification of the UML and OCL Meta-Model in Isabelle/HOL," in *Multitopic Conference, 2007. INMIC 2007. IEEE International*, Dec 2007, pp. 1–6.

[23] A. D. Brucker and B. Wolff, "HOL-OCL: A formal proof environment for UML/OCL," in *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008 Proceedings*, ser. Lecture Notes in Computer Science, J. L. Fiadeiro and P. Inverardi, Eds., vol. 4961. Springer, 2008, pp. 97–100.

[24] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon, "Verification-driven slicing of UML/OCL models," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 185–194.