REPORTS
IN
INFORMATICS

ISSN 0333-3590

Model design in DPF with transparent analysis in Alloy

Xiaoliang Wang, Adrian Rutle, Yngve Lamo

REPORT NO 410 March 2015

Department of Informatics
UNIVERSITY OF BERGEN
Bergen, Norway

Model design in DPF with transparent analysis in Alloy

Xiaoliang Wang, Adrian Rutle and Yngve Lamo[*]

Department of Informatics
University of Bergen
N-5020 Bergen
Norway

April 13, 2015

Abstract

Model driven software engineering (MDSE) aims to raise the abstraction level in software engineering by representing software as models. User-friendly diagrammatic modelling languages like UML, EMF, etc, have boosted the acceptance of MDSE. However, it is still an open challenge to ensure that diagrammatic models are correct with respect to certain properties. In this paper we present an approach to automatically verify these models and provide feedback of the analysis in the modelling domain. The approach is based on a bidirectional transformation between models specified in the Diagram Predicate Framework (DPF) and Alloy specifications. The Alloy Analyzer is used to check the consistency of the models by searching for valid instances. The results of the analysis are then presented as user-friendly feedback directly in the DPF model editor. In case the model is inconsistent, its problematic part is highlighted and an error message is displayed to the designer. Furthermore, the approach supports discovery and presentation of redundant constraints assisting the modeller to design more compact models. With this approach the designer can focus more on the modelling activities without bothering about the underlying formal methods used for the analysis. We use a running example to illustrate the approach, which is based on a benchmark of examples presented by Gogolla *et al* in [1].

1 Introduction

Abstraction in software engineering is an endeavour to manage the complexity of software. Model-driven software engineering (MDSE) is a branch of software engineering in which software is developed from models. One of the main ideas of MDSE is that one can reason about models at a higher level of abstraction than the programming code. These models are usually defined in diagrammatic modelling languages and frameworks, such as the UML and the EMF, where models are represented as graphs with nodes representing domain concepts and edges representing relations between these concepts. In addition, properties of the models are typically defined either by built-in constraints, e.g., cardinalities of relations, or in external constraint definition languages, e.g., the Object Constraint Language (OCL).

[*]Email: {`xiaoliang.wang, adrian.rutle, yngve.lamo`}`@hib.no`

Another main idea in MDSE is to use model transformations to automatically generate software from models. Since the correctness of the generated software highly depends on the correctness of the models, it is important to analyse the models and check whether they have the intended meaning. In addition to consistency checking, one may analyse models to detect whether they contain redundant constraints. Thus, one of the main arguments for adoption of MDSE depends on the existence of robust and user-friendly techniques and tools for model analysis [2].

Although there are several tools supporting MDSE activities, they are usually meant for design rather than analysis. Despite the existence of numerous formal analysis tools, using formal methods is not attractive for software engineers due to time-to-market constraints and since special expertise is required to apply such tools [3]. A consequence is that, although it is obvious that finding design mistakes as early as in the modelling phase will help building better software at a lower cost, model analysis is usually underestimated and seen as a time- and money consuming activity.

In this paper, we take a step to bridge the gap between the design and analysis spaces in order to build robust software. To explain the approach, we use a sample structural model which is an extended version of the traditional civil status model originally presented in [1]. The model is specified in the Diagram Predicate Framework (DPF) [4], and implemented in the DPF Workbench [5]. Our approach is to transform these structural models into Alloy specifications, analyse the specifications with the Alloy Analzyer, and then present the results of the analysis directly in the design space (see Fig. 1). In this way the designer can mainly focus on the modelling activities without bothering about the underlying formal methods used for the analysis.

Note that we illustrate our approach by using DPF for modelling and Alloy for analysis. We chose DPF since it has a formal foundation and it provides a fully diagrammatic representation of both model structure and constraints. DPF also provides tool support for creating domain specific languages, model editors, code-generators, etc. Moreover, we chose Alloy since it is a lightweight approach to formal verification that is well suited for automation. It uses first-order relational logic to specify complex structural models; and offers an analysis tool which ensures termination and generate instances or counterexamples for different analysis cases. Furthermore, we believe that the ideas of the paper could be generalised for other modelling and analysis spaces.

The three main contributions of the paper are checking **consistency** of structural models, checking existence of **redundant constraints**, and presenting **user-friendly feedback** to the model designer. Consistency of models are checked by analysing whether there exists at least one instance conforming to the model. The analysis result assist the model designer in two ways. If such an instance exists, an example instance will be presented to the designer for manual inspection so that she can decide whether or not to modify the model. For

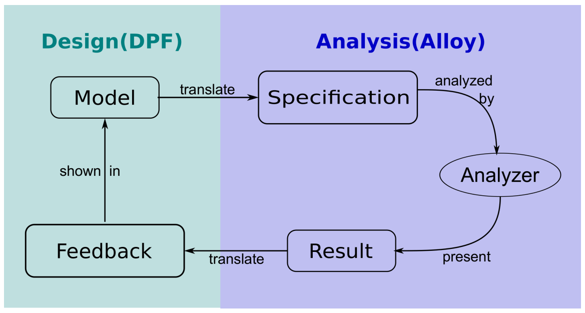


Figure 1: Workflow for analysing structural models using Alloy

example, for a model representing persons and their civil status an instance which shows that ”a person could be both male and female at the same time” would indicate that the model is not restrictive enough, and the designer may decide to add additional constraints to the model. Otherwise, the model may be inconsistent, which means there are constraints causing a conflict. For example, in the above mentioned model of civil status, the constraint which specifies that ”all persons must be married” will conflict with the constraint which specify that ”at least one person is single” since a person cannot be both single and married at the same time. Such conflicts will be highlighted to help the designer to quickly find the problematic part of the model. Furthermore, we can check whether some constraints are redundant in the sense that they could be derived from other constraints in the model. If this is the case, the model designer can choose to delete redundant constraints from the model, making the model more lightweight and easier to comprehend. However, in some cases the model designer may use redundant constraints to emphasize their importance and therefore keep them despite the feedback from the tool.

In Section 2 we outline a brief background of the Diagram Predicate Framework (DPF) and Alloy before we present a bidirectional transformation between DPF models and Alloy specifications. In section 4 we illustrate how this transformation is used for presentation of feedback from the Alloy Analyser in the DPF Workbench. Finally, we present some related work and concluding remarks in Sections 5 and 6, respectively.

2 Background

We start this section with a brief introduction to structural modelling in DPF and the DPF Workbench, then we give a short introduction to Alloy. For details and formal definitions we refer to [4, 5, 6, 7, 8, 9].

2.1 Diagram Predicate Framework

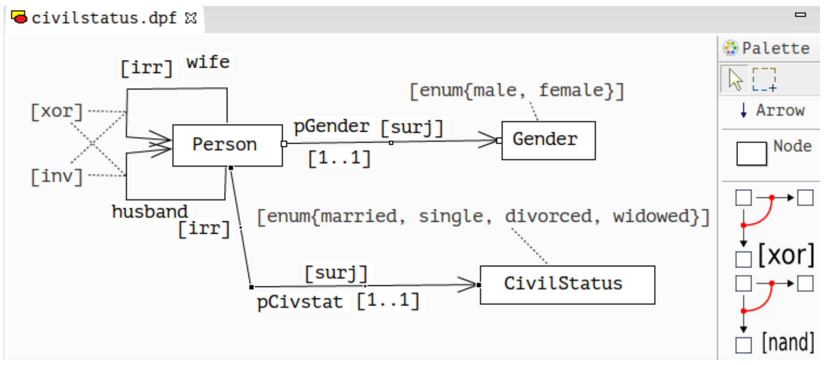


Figure 2: Civil status model in DPF

DPF provides a formalisation of (meta)modelling based on category theory [10]. 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 comes with a workbench which enables development of domain specific languages in a hierarchy where the domain under study can be specified at several abstraction levels [4].

A DPF model is represented by a diagrammatic specification $\mathfrak{S} = (S, C^{\mathfrak{S}})$ consisting of an underlying graph structure S and a set of constraints $C^{\mathfrak{S}}$. S is a directed graph containing nodes and arrows that represent the concepts and their relationships in a problem domain.

Table 1: A sample signature Σ

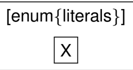
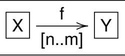
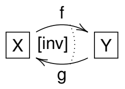
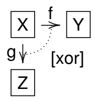
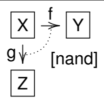
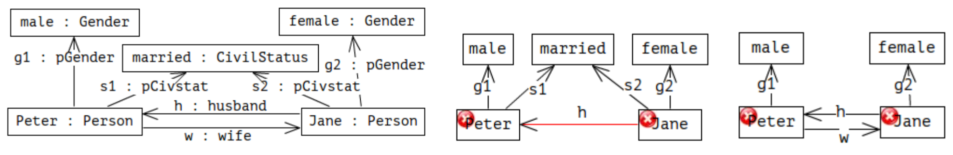
p	$\alpha^{\Sigma}(p)$	Proposed Visualization	Semantic Interpretation
`enum({literals})`	1		$\forall x \in X : x \in \{literals\}$
`multi(min, max)`	$1 \xrightarrow{f} 2$		$\forall x \in X : n \leq \lvert f(x) \rvert \leq m \wedge 0 \leq n \leq m \wedge m \geq 1$
`inverse`	$\begin{tikzcd} 1 \arrow[r, bend left, "f"] & 2 \arrow[l, bend left, "g"] \end{tikzcd}$		$\forall x \in X , \forall y \in Y : y \in f(x)$ iff $x \in g(y)$
`xor`	$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[d, "g"] & 2 \\ 3 \end{tikzcd}$		$\forall x \in X : (\exists y \in Y : y \in f(x)) \Leftrightarrow (\nexists z \in Z : z \in g(x))$
`not-and`	$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[d, "g"] & 2 \\ 3 \end{tikzcd}$		$\forall x \in X : \neg(\exists y \in Y : y \in f(x) \wedge \exists z \in Z : z \in g(x))$

Fig. 2 presents a model which describes the civil status of people in the DPF Model Editor (The complete example can be download at `http://dpf.hib.no/downloads/civil.zip`). The model extends the traditional civil status model in [1] originally specified in UML and OCL. In the figure, the nodes **Person**, **CivilStatus** and **Gender** denote the concepts: person, civil status choices and gender choices, respectively; the edges **husband** and **wife** denote the relationships between persons while the edges **pCivstat** and **pGender** describe the civil status and the gender of persons, respectively. In addition to the structure described by the underlying graph of the model, constraints are used to specify properties of the problem domain. For example, the property $prop_1$ "if person p1 has husband p2, then p2 must have p1 as wife and vice versa" can be specified as a constraint between the relationships husband and wife; the property $prop_2$ "a person can have exactly one civil status" can be specified as a constraint on the relationship civil status.

The semantics of a model is given by its instances. An instance of a model $\mathfrak{S} = (S, C^{\mathfrak{S}})$ is a graph I which is typed by the underlying graph S (denoted by $I : S$) and satisfies each constraint $c \in C^{\mathfrak{S}}$, also written as $I \vDash C^{\mathfrak{S}}$. A graph I typed by S means that there exists a graph homomorphism $\iota : I \to S$. Fig. 3 shows three graphs of which only Fig. 3a is an instance of $\mathfrak{S}$, since Fig. 3b and violate the properties $prop_1$ and $prop_2$, respectively.

In DPF, atomic constraints and graph constraints [11], are used to specify requirements. In the following, we will give a brief introduction of the syntax, semantics and tool support for these two kinds of constraints.

Given a model $\mathfrak{S}$ with structure S, an atomic constraint $c = (p, \delta)$ is specified on a subgraph of S by means of a predicate p from a predefined signature Σ (see Table 1) and a graph homomorphism $\delta : \alpha^{\Sigma}(p) \to S$ from the arity α^{Σ} of p to S. The δ indicates which part of S is affected by p (see Fig. 2). Each predicate in Σ shown in Table 1 has a name p, an ar-



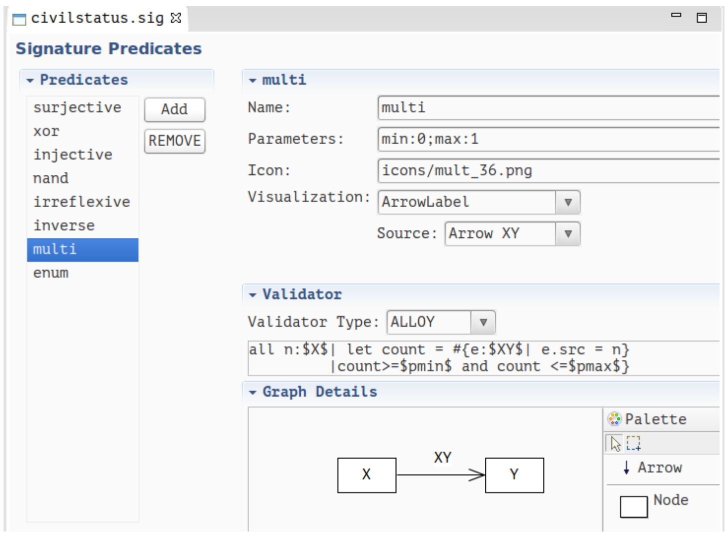
(a) Example Instance (b) Invalid Graph1 (c) Invalid Graph2

Figure 3: Graphs typed by the structure in Fig. 2

ity $\alpha^{\Sigma}(p)$, a proposed visualization and a semantic interpretation. The arity is a graph that specifies on which patterns the atomic constraint can be applied; the visualization proposes a diagrammatic way to represent constraints; finally, the semantic interpretation defines (in set theory) the valid instances of the predicate. A predicate is applicable on a subgraph S' of S if there exists an epimorphism δ from $\alpha^{\Sigma}(p)$ to S'. If we select a subgraph S' the predicates that are applicable on S' will be shown in the palette of the DPF Model Editor in form of icons. The designer can click the icons to formulate atomic constraints. For instance, the properties $prop_1$ and $prop_2$ are formulated as the atomic constraints [xor] and [1..1] on the edges **husband** and **wife**, and the **pCivstat** based on the predicate `xor` and `mult`, respectively. An instance I satisfies an atomic constraint (p, δ) if the graph I^* derived by the pullback in Fig. 4 satisfies the semantic interpretation of the predicate p (see [4]).

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

Figure 4: Semantics of atomic constraints


civilstatus.sig

Signature Predicates

Predicates

surjective
xor
injective
nand
irreflexive
inverse
multi
enum

Add
REMOVE

multi

Name: multi

Parameters: min:0;max:1

Icon: icons/mult_36.png

Visualization: ArrowLabel

Source: Arrow XY

Validator

Validator Type: ALLOY

```
all n:$X$| let count = #{e:$XY$| e.src = n}
          |count>=$pmin$ and count <=$pmax$}
```

Graph Details

Figure 5: Specification of user-defined predicates in the Signature Editor

Note that, in Fig. 2, we use the atomic constraint [enum{literals}] to specify an enumeration data type. The allowed literals are defined within curly brackets, separated by ",", e.g. the literals of **Gender** are $\{male, female\}$. Since a person must have exactly one gender, we restrict the relation **pGender** with the constraint [1..1]. In UML, this kind of relations are specified as attributes of an enumeration type and it is default that each attribute can have exactly one value from the defined literals. In DPF, we specify these relations as edges and define the multiplicity constraint explicitly.

In the DPF workbench [5], several predefined predicates are provided. Some of these are shown in Table 1. In addition, model designers can define their own predicates using the Signature Editor. For example, the predicate `multi` is defined in the Signature Editor as shown in Fig. 5. The arity of the predicate is specified in the Graph Details area. The Parameter is a string to specify the necessary parameters of the predicate. Each parameter has a name and a default value which are separated by ":". Different parameters are separated by ";". For instance, $min : 0; max : 1$ in `multi(min, max)` defines two parameters min and max and their default values 0 and 1, respectively; the Icon is the filename of the image which will be displayed in the palette in the DPF Model Editor (see Fig. 2); in the Visualization field, the designer may specify how to visualise the constraints formulated by the predicate, in the Validator field the designer can specify the semantics of the predicate in Java, OCL or Alloy. For more information about the signature editor, please consult [5].

Table 2: GCs applied to instances of the civil status model

$N:S$	$L:S$	$R:S$
$MarriedWithoutWife$		
$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[r, "\text{w:wife}"] \arrow[d, "\text{s:pCivstat}"] & \boxed{\text{p3:Person}} \\ \boxed{\text{married:CivilStatus}} \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[d, "\text{s:pCivstat}"] \\ \boxed{\text{married:CivilStatus}} \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[r, "\text{h:husband}"] \arrow[d, "\text{s:pCivstat}"] & \boxed{\text{p2:Person}} \\ \boxed{\text{married:CivilStatus}} \end{tikzcd}$
$MarriedWithoutHusband$		
$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[r, "\text{h:husband}"] \arrow[d, "\text{s:pCivstat}"] & \boxed{\text{p3:Person}} \\ \boxed{\text{married:CivilStatus}} \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[d, "\text{s:pCivstat}"] \\ \boxed{\text{married:CivilStatus}} \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[r, "\text{w:wife}"] \arrow[d, "\text{s:pCivstat}"] & \boxed{\text{p2:Person}} \\ \boxed{\text{married:CivilStatus}} \end{tikzcd}$
$HasWifeIsMarried$		
	$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[d, "\text{w:wife}"] \\ \boxed{\text{p2:Person}} \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[r, "\text{w:wife}"] \arrow[d, "\text{s:pCivstat}"] & \boxed{\text{p2:Person}} \\ \boxed{\text{married:CivilStatus}} \end{tikzcd}$
$HasHusbandIsMarried$		
	$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[d, "\text{h:husband}"] \\ \boxed{\text{p2:Person}} \end{tikzcd}$	$\begin{tikzcd} \boxed{\text{p1:Person}} \arrow[r, "\text{h:husband}"] \arrow[d, "\text{s:pCivstat}"] & \boxed{\text{p2:Person}} \\ \boxed{\text{married:CivilStatus}} \end{tikzcd}$

In addition to atomic constraints, Graph constraints (GC) may be used to define dependencies among constraints and/or the structures of a model [4]. For example, we further specify that "if a person is married and has no wife, the person should have a husband". We can specify this as the GC $MarriedWithoutWife$ as shown in Table 2. Each GC $N \overset{n}{\hookleftarrow} L \overset{u}{\hookrightarrow} R$ consists of three graphs: left L, right R and negative application condition N; and two injective graph homomorphisms n and u (see [4, 11]). The components L, R and N are graphs typed by the underlying graph S, denoted by $L:S$, $R:S$ and $N:S$ in the table. Note that in DPF, GCs are generalised such L and R can be specifications instead of graphs, see [4]. However in this paper, we only consider classical GCs and leave the case with universal constraints to future work. In the constraint $MarriedWithoutWife$ in Table 2, **p1**, **p2** and **p3** are typed by **Person**; **married** is typed by **CivilStatus**; **p1** in $L:S$ identifies **p1** in $R:S$ and $N:S$. The other GCs express properties which are explained by their names.

The semantics of GCs is outlined in Fig. 6. An instance I of $\mathfrak{S}$ satisfies a GC when for every match of L in I (i.e. there exists a graph morphism $l : L \to I$), if there is no match of N in I where $l = n; n'$, then there must exist a match of R in I where $l = u; r$. Since we consider graphs which are typed by S, we have to also require that a match is compatible with typing. For example, a match of L in an instance I which is typed by S is a graph homomorphism $l : L \to I$ such that $l; \iota^I = \iota^L$.

$$\begin{tikzcd}
 & S & \\
N \arrow[ur, "\iota^N"] \arrow[dr, "n'"'] & L \arrow[l, hook', "n"'] \arrow[r, hook, "u"] \arrow[u, "\iota^L"] \arrow[d, "l"] \arrow[dl, phantom, "\neq"] \arrow[dr, phantom, "="] & R \arrow[ul, "\iota^R"'] \arrow[dl, "r"] \\
 & I \arrow[uu, bend left=80, "\iota^I"] &
\end{tikzcd}$$

Figure 6: Semantics of GC

In the DPF workbench, we use an editor to specify GCs (see Fig. 7). The UConstraints field lists all the GCs for a model; the other fields specify the name, the Graph Details of a GC and the Type. Since we assume that both n and u are injective, inspired by Henshin [12], the GSs are specified with the following colour coding: red elements belong to N minus L; green elements belong to R minus L; and grey elements belong to L. In other words, N is the sum of grey and red elements; R is the sum of grey and green elements; and L is the grey elements.

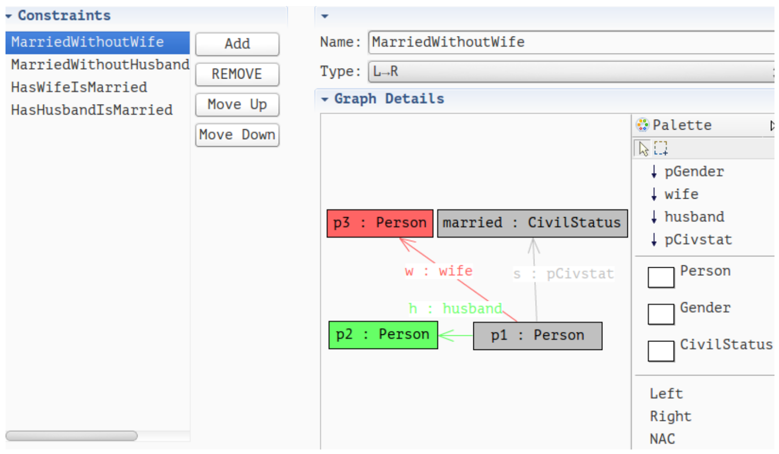


Figure 7: Specification of GCs in DPF Workbench

2.2 Alloy

Alloy consists of a structural modelling language and a tool which are used to specify and analyse models. The modelling language is a declarative textual language, suited for describing complex model structures and constraints based on relational logic. In Alloy, signatures (*sig*) are used to define concepts of a domain, while relationships among those concepts are defined as fields of the signatures. Each signature denotes a set of *atoms* as its instances. Moreover, *atoms* are the primitive entities. A user-defined scope specifies the size of a signature, i.e. the number of its atoms. Fields denote relations between those atoms while the arity of a relation is decided by the field declaration. For example, the traditional marriage can be modelled as the specification in Listing 1. Gender, civil status and person are modelled by the signatures `Gender`, `CivilStatus` and `Person`; the marriage relationship between persons is modelled as the two fields `husband` and `wife`, two binary relations between `Person`; the gender and civil status of a person is modelled as the fields `pGender` and `pCivstat`, respectively. These are binary relations between `Person` and `Gender` and `PersonCivilStatus`. The keyword `lone` restricts that a person has at most one husband or wife; the keyword `one` restricts that a person has exactly one gender and one civil status.

Alloy does not have a primitive enumeration type. We simulate enumeration by inheritance and restricting the size of a signature. For example, the two signatures `male` and `female` extend `Gender`; the keyword `abstract` restricts that `Gender` has no instances except the instances of `male` and `female`; the keyword `lone` restricts that `male`, `female` have at most one instance. In this way, the signature `Gender` defines an enumeration type with `male` and `female` as its literals. Similarly, `CivilStatus` defines an enumeration type.

```
abstract sig Gender{}
lone sig male, female extends Gender{}
abstract sig CivilStatus{}
lone sig married, single, divorced, widowed extends CivilStatus{}
sig Person{
    husband, wife:lone Person,
    pGender:one Gender,
    pCivstat:one CivilStatus}
pred inverse[p:Person]{all p1:Person|p1 in p.wife iff p in p1.husband}
fact inv{all p:Person|inverse[p]}
```

Listing 1: Civil status model in Alloy

Constraints on Alloy specifications are defined as `fact`s. Predicates (`pred`) are analogous to function declarations (with boolean return type) in programming languages. Predicates affect specifications only if they are invoked in a `fact`. For example, we use the predicate `inverse` (Line 9 in Listing 1) to encode that a person has a wife if and only if the person is the husband of the wife. The fact `inv` (Line 10) restricts the specification by invoking the predicate `inverse`.

An instance of an Alloy specification is a set of atoms where each atom belongs to a signature (in modelling terms we say that the atom is typed by the signature), and each field of the signature is instantiated by a relation among those atoms. Note that the top level signatures (i.e. signatures which do not inherit from other signatures) define disjoint sets of atoms. Thus, an atom can only belong to more than one signature if there is an inheritance relation among those signatures. In addition, the atoms and the relations of an instance satisfies all the facts in the specification. A sample instance of the civil status model with one single man may be described as a relation with a set of tuples: `Person={(P1)}, single ={(single)}, Gender={(male)}, CivilStatus={(single)}, male={(male)}, pGender={(P1, male)}, pCivstat={(P1, single)}`. Other signatures and fields are empty. Note that the atoms of signatures are also represented as relations, since atoms are also treated as unary relations in Alloy.

Model analysis is performed by a constraint solver called Alloy Analyzer. It analyses a specification by first translating it into a SAT problem and then solving the problem using some off-the-shelf SAT solvers. The analyzer uses two commands: *run* or *check* to find instances (or counterexamples) satisfying (or violating) some properties of the specification. It tries to find these instances by exhaustively iterating through a search space containing a set of possible instances, each being well-typed by the specification. Usually, there are infinitely many possible instances. To make the analysis feasible, a user-defined *scope* restricts the size of each signature. In this way, the search space is bounded to a finite set restricted by the scope. However, this means that the analysis is incomplete. That is, if an instance or a counterexample can be found within the scope, it shows that the properties are satisfied or violated, respectively. Otherwise, it is uncertain whether some instances or counterexamples satisfy or violate the properties in a larger scope. In addition to incompleteness, scalability is another limitation with the Alloy Analyzer. The user can increase the scope to obtain greater confidence about the analysis result when no instance or counterexample is found. However, the search space grows exponentially along with the scope. Within a larger scope, the analysis will become very slow, or even intractable.

Model analysis in Alloy is different to analysis using other formal verification techniques such as model checking and theorem proving. To explain this difference, we have to make it clear that what we call a model or a specification in DPF and Alloy are called a *formula* in other verification techniques, while instances are called *models*. In model checking, given a formula and a model, the model is checked to show whether it is valid with respect to the formula. Different from model checking, the Alloy Analyzer examines the formula by finding if there is a counter-model within a search space. If such a counterexample is found, it asserts that the formula is invalid. Otherwise, it is still uncertain whether the formula is valid. In this sense, the Alloy Analyzer is more like a model finder rather than a model checker. Similar to the Alloy Analyzer, theorem provers examine whether a given formula is valid. However, theorem provers verify the formula by building a proof rather than finding a counter-model. If such a proof is constructed, the formula is verified valid. Otherwise, it is not clear if the formula is invalid or the formula cannot be verified.

Since the Alloy Analyzer verifies models by searching for counterexamples, verifying that a model is inconsistent can be accomplished quicker than finding a proof. Although the Alloy Analyzer is incomplete and has scalability problems, according to the *small scope hypothesis*, most bugs have small counterexamples. Therefore, it has been used to examine systems that involve complex structured state such as network configuration protocols, scheduling, access control [9].

3 Transformations between DPF and Alloy

In this section, we present a bidirectional transformation between DPF and Alloy which will be used in section 4 for analysis and feedback report.

DPF represents models diagrammatically while Alloy represents models textually. Despite of this difference, there are similarities between the two approaches. For the description of concepts and their relations in a domain, DPF uses nodes and edges while Alloy uses signatures and *field*s. For the specification of constraints, in DPF, atomic constraints are formulated based on predicates while GCs are applied to instances of models. Similarly, constraints (*fact*s) in Alloy are specified based on predicates (*pred*s) or without invoking predicates. These similarities make it easier to translate back and forth between the two approaches, at least when it comes to constraints. Table 3 shows the mappings between DPF and Alloy. 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 subgraph more than once. That is, for any two constraints (p_1, δ_1) and (p_2, δ_2), $p_1 = p_2 \Rightarrow \delta_1(\alpha(p_1)) \neq \delta_2(\alpha(p_2))$.

DPF	Alloy
Node	`sig NNode{}`
Node with [enum$\{l_1, \dots, l_n\}$]	`abstract sig NNode lone sig N`$l_1, \dots,$`N`l_n `extends NNode{}`
Edge:S $\rightarrow$**T**	`sig EEdge{ src:one NS, trg: one NT}`
Constraint	`fact`

Table 3: Correspondence between DPF Models and Alloy Specifications

```
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 2: Civil status model in Alloy

Constraints from the DPF models are also encoded as uniquely named facts in Alloy. Atomic constraints are constructed based on predicates whose semantics are defined in first-order logic (FOL). In DPF workbench, the semantics of predicates is implemented using a validator. In previous versions of the DPF Workbench, the validators were defined as Java methods or as OCL expressions. However, as shown in Fig. 5, the DPF Workbench enables the designer to specify the semantics of predicates in Alloy. For example, the semantics of the predicate xor can be specified in Alloy as Line 2 in Listing 3. In the civil status model in Fig. 2, a constraint [xor] is formulated on the edges **wife** and **husband**. The graph morphism from the arity of the predicate to the constrained subgraph is $\delta = \{X \mapsto$**Person**, $Y \mapsto$**Person**, $Z \mapsto$**Person**, $XY \mapsto$**wife**, $XZ \mapsto$**husband**$\}$. When translating the model to Alloy, the constraint is translated into a named fact where the name is the predicate name (e.g. xor) plus the names of arrows (nodes if no arrows in the predicate) connected with "_", e.g., `xor_wife_husband`; and the content of the fact is obtained from the semantics of the predicate in Alloy by replacing the variables enclosed with $, e.g. `$X$`, with the signature of its value $\delta(X)$, e.g. `NPerson`. As a result, the constraint [xor] on the edges **wife** and **husband** is translated into the fact on Line 4 in Listing 3. GCs are also automatically translated to FOL formulae in Alloy according to their semantics; e.g., the constraint *MarriedWithoutWife* is translated to the fact on Line 6.

The mappings between the underlying structure of DPF models and their counterparts in Alloy are more challenging. The structure of the civil status model in Fig. 2 is translated into Alloy as shown in Listing 2. The nodes, e.g. **Person**, are encoded as signatures in Alloy as in Line. 1 without any field. Edges, e.g. **husband**, are also encoded as signatures with two fields *src*, *trg* representing which nodes the edges connect. Each of these fields is restricted with the keyword *one*, meaning that they have exactly one source and one target. Moreover, signatures representing nodes and edges are prefixed with N and E, respectively. This makes it easier to translate the analysis result from Alloy to DPF, as shown in Section 4. Furthermore, since the Alloy Analyzer use the scope variable to restrict the size of each signature, the encoding enables us to have control over multiplicity on the edges. In Listing 1, the scope is not affecting the edges, hence the number of edges are dependent on the scope of the nodes. For modelling, it is also more general in the sense that the encoding allows parallel relations between two nodes in the instance level.

```
//predicate semantic defined in DPF
all x:$X$|(some xy:$XY$|xy.src=x) iff not (some xz:$XZ$|xz.src=x)
//constraint translated to Alloy
fact xor_wife_husband{
all x:NPerson|(some xy:Ewife|xy.src=x) iff not (some xz:Ehusband|xz.src=x)}
fact MarriedWithoutWife{all p1:NPerson|((some g:EpCivstat|g.src=p1) and not (some w:Ewife|
    w.src=p1)) implies (some h:Ehusband|h.src=p1) }
```

Listing 3: Atomic constraints and GCs in Alloy

Using Table 3, we can derive a mapping from an instance of an Alloy specification, which represents a DPF model, to an instance of the DPF model. In more details, an atom that belongs to a `sig NNode` is translated back to a node in a DPF instance which is typed by the node **Node** in the DPF model. Atoms that belong to edge signatures are translated similarly. Moreover, for each edge in the DPF instance, the source and target nodes of the edge are set according to the relations `src` and `trg`. If an atom belongs to both `sig NNode` and `sig NNode1` where `NNode` inherits from `NNodes1`, the atom is translated to a node typed by the node **Node1** in the DPF model. We can use this mapping to translate the result of the model analysis in Section 4, i.e. Alloy instances to DPF instances.

4 Alloy Analysis and Presentation of Feedback

After translating DPF models into Alloy specifications, we analyse them with the Alloy Analyzer. The Alloy commands *run* and *check* are used for model consistency analysis and to check if a constraint is redundant, respectively. Moreover, we translate the result of the analysis and present it in the DPF model editor.

4.1 Analysis of Model Consistency

The DPF workbench is extended with new tool support that enables 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 3. The $run\{\}$ command is used to search for an instance of the Alloy specification. Since the transformation is bidirectional we can use this Alloy instance to construct an instance of the DPF model. Recall that the Alloy Analyzer need a user-defined scope restricting the size of the search space. Here we set the default scope to 3 which is the default scope in Alloy.

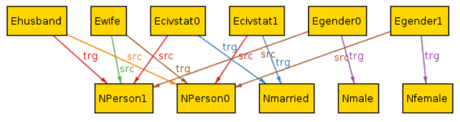


Figure 8: Instance in Alloy

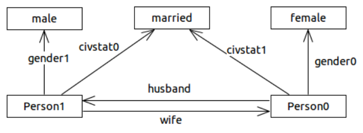
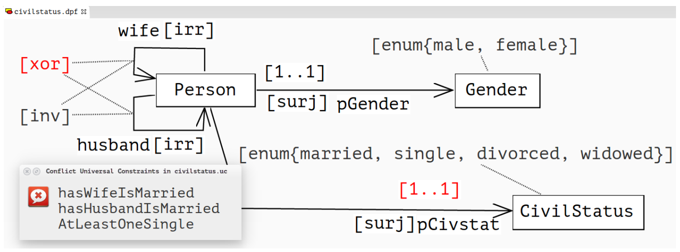


Figure 9: Instance in DPF

If the Alloy analyzer finds a valid instance for a specification, it will present the instance as a graph. For example, the civil status model is verified consistent by Alloy and one of its instances is shown in Fig. 8. As we explained in Section 2.2, an instance in Alloy is

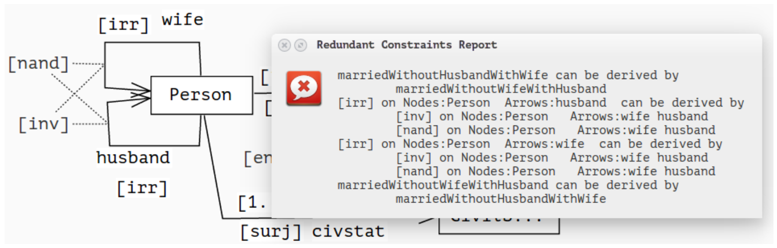


```
hasWifeIsMarried
hasHusbandIsMarried
AtLeastOneSingle
```

Figure 10: Highlighting the inconsistency cause in DPF model Editor

a set of relations where each tuple is an ordered list of atoms. We translate the nodes and edges of DPF models into signatures in Alloy as in section 3, hence the Alloy instances consist of unary relations for node and edge signatures, and binary relations for the `src` and `trg` fields. The instance in Fig. 8 contains the relations, `NPerson={(NPerson0), (NPerson1)}, Ehusband={(Ehusband)}, Ehusband.trg={(Ehusband1,NPerson0)}` etc. The atoms of signatures, i.e., tuples of the unary relations, are visualised as yellow boxes representing nodes and edges in DPF where the tuples for the binary relations are visualised as arrows between those boxes. However, the instance graph given in Alloy is not easily understandable for designers. In order to make it easier for the designer to interpret the instance, we translate it back to a DPF instance. Using the mapping from Alloy instances to DPF instances, the instance in Fig. 8 can be translated into a DPF instance as shown in Fig. 9.

If the analyser cannot find an instance, it may be that the model is inconsistent. Now we assume that the designer adds another constraint to the model, it ensures that at least one single person exists. The constraint is specified with the GC $singlePerson$. The $N{:}S$ and $L{:}S$ side is empty while the $R{:}S$ side is $\boxed{\text{p1:Person}} \xrightarrow{\text{s:civstat}} \boxed{\text{single:CivilStatus}}$. Afterwards, the designer analyses the model again, the civil status model is found inconsistent. In this case, the Alloy Analyser gives information about the part of the specification that cause the inconsistency[1]. The information is given as the locations of the expressions which contradict each other. We can use the locations to find which 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. For example, in the inconsistent model, the locations of some expression contained in the facts $xor\_Ewife\_Ehusband$, $multi\_Ecivstat$, $hasWifeIsMarried$, $hasHusbandIsMarried$ and $singlePerson$ in the Alloy specification are given by the Alloy Analyzer. By the name of the facts, we find that the atomic constraint [xor] on the edges **husband** and **wife**, and 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 (see Fig. 10). 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.



```
marriedWithoutHusbandWithWife can be derived by
        marriedWithoutWifeWithHusband
[irr] on Nodes:Person  Arrows:husband  can be derived by
        [inv] on Nodes:Person   Arrows:wife husband
        [nand] on Nodes:Person   Arrows:wife husband
[irr] on Nodes:Person  Arrows:wife  can be derived by
        [inv] on Nodes:Person   Arrows:wife husband
        [nand] on Nodes:Person   Arrows:wife husband
marriedWithoutWifeWithHusband can be derived by
        marriedWithoutHusbandWithWife
```

Figure 11: Redundant constraints feedback in DPF Workbench

4.2 Detecting Redundant Constraints

Searching for redundant constraints is performed in a similar way as model consistency analysis. When the designer chooses the context menu ”Redundancy Check”, the DPF model is translated to an Alloy specification. The only difference is that we use the command `check{c}` to check whether the constraint c 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 c_{n+1} can be derived from $\{c_1, \ldots, c_n\}$, i.e., $c_1 \wedge \ldots \wedge c_n \Rightarrow c_{n+1}$. That is, whether every valid instance of the model satisfies c_{n+1}. In Alloy, the check can be accomplished 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}. After executing the command, if such a counterexample is found, it means that the constraint is not redundant and the counterexample will be presented to the designer. Otherwise, the constraints which can derive c_{n+1} will be reported. The technique to find which constraints that can derive 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. 11. The designer may choose to keep or delete these.

In the end, we should state again 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 within the given scope.

5 Related Work

There are several approaches to apply Alloy for analysis of software models. In [13], Anastasakis *et al* analysed Alloy specifications that represent UML models. The idea was to use model transformations to translate UML models to Alloy specifications. The structural information and corresponding constraints, expressed in a subset of OCL, are translated to specifications in the Alloy language. This transformation approach is implemented in the tool UML2Alloy. The UML2Alloy tool is also used in [14], where Georg *et al* promotes the use of the Aspect-Oriented Risk-Driven Development (AORDD) methodology for developing secure systems. With the approach, systems specified in UML are translated to Alloy by UML2Alloy before they formally verify some security properties.

In [15], Aspect-Oriented Systems specified as Aspect-UML models are translated to Alloy specification for analysis. Some local and global properties are verified with the approach. However, the translation is not implemented. Moreover, Braga *et al* [16] analyse the models specified in OntoUML by translating them into Alloy and checking them by the Alloy

[1]It is required to select a SAT solver with Unsat Core for the Alloy Analyzer to give such information. Moreover, Alloy also uses two options : Core Size and Core Granularity to control the size and the granularity of the unsat core. Here, we choose the Minisat with Unsat Core and set the two options to Slow and 2 (Highest granularity). We will use the same settings later for searching after redundant constraints.

Analyzer. Similar to our approach, the mentioned approaches perform model analysis by checking properties of Alloy specifications. In contrast to our approach the result of the analysis is only presented in Alloy and not translated back to the modelling tool.

There are also some approaches on translating the result of the Alloy analysis back to the design domain. For example, in [17], the authors build on the approach of [13] and extend it by translating the analysis result back to UML and OCL. A Trace2MT transformation is derived automatically from the UML2Alloy transformation. Basically the authors convert the mapping in the UML2Alloy transformation into another transformation, which converts an Alloy instance to an UML Object Diagram. Similar to our work, Shah *et al* provides an approach to bridge the gap between design and analysis, but the actual implementation is different. In [17], the mapping from UML to Alloy is maintained and used to translate Alloy instances back to UML Object diagram. However, in our approach, because of the similarity between DPF and Alloy, we are able to define the correspondence between DPF elements and Alloy artifacts as bidirectional transformations. We can derive the DPF instance from an Alloy instance without maintaining tracing. Moreover, we can also highlight the problem cause in the DPF model when the analysis fails. This is not covered in [17]. Similarly, Cunha *et al* [18] presented an approach to translate Alloy to UML Class Diagram. Their work provided a transformation from Alloy to OCL. However, they concentrated on the transformation rather than the analysis. Therefore, the presentation of feedback to the modeller is not covered.

6 Conclusion

We have presented an approach to automatically analyse diagrammatic, structural models. The approach transforms graph-based structural models into Alloy specifications and uses the Alloy Analyser to analyse the models within a given scope. The main contributions of the paper are checking consistency of structural models, finding redundant constraints, and presenting user-friendly feedback to the model designer. These feedbacks are presented in the graph-based modelling framework which was used for the design. We analyse both atomic constraints (e.g. cardinality constraints) and graph constraints (e.g. implications), as well as the relation between these constraints (e.g. whether they contradict or induce each other). In future work, we will consider universal constraints; i.e., graph constraints which contain atomic constraints. The approach may be slow for large models or larger scopes. Therefore, we plan to address the performance issues by de-composing models into independent components, analysing the components, then composing them back to obtain the whole, original model. Moreover, we are currently working on an approach to deal with the incompleteness of the verification by deducing a maximal scope which is sufficient to verify given constraints. The presented technique is general in the sense that it can be used to analyse other models without dynamic behaviour. We will study if the technique can be adapted to analysis of models with dynamic behaviour in the future.

References

[1] Martin Gogolla, Fabian Büttner, and Jordi Cabot. Initiating a benchmark for UML and OCL analysis tools. In *Tests and Proofs - 7th International Conference, TAP, Budapest, Hungary*, pages 115–132, 2013.

[2] Bran Selic. What will it take? a view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

[3] Jörg Kreiker, Andrzej Tarlecki, Moshe Y. Vardi, and Reinhard Wilhelm. Modeling, Analysis, and Verification - The Formal Methods Manifesto 2010 (Dagstuhl Perspectives Workshop 10482). *Dagstuhl Manifestos*, 1(1):21–40, 2011. ISSN 2193-2433.

[4] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, 2010.

[5] Yngve Lamo, Xiaoliang Wang, Florian Mantz, Wendy MacCaull, and Adrian Rutle. DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment. In *Computer and Information Science*, volume 429 of *Studies in Computational Intelligence*, pages 37–52. 2012.

[6] Zinovy Diskin. *Encyclopedia of Database Technologies and Applications*, chapter Mathematics of Generic Specifications for Model Management I and II. Information Science Reference, 2005.

[7] Zinovy Diskin, Boris Kadish, Frank Piessens, and Michael Johnson. Universal Arrow Foundations for Visual Modeling. In *1^{st} International Conference on Diagrammatic Representation and Inference*, 2000.

[8] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A formal approach to the specification and transformation of constraints in MDE. *JLAP*, 2012.

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

[10] Michael Barr and Charles Wells. *Category Theory for Computing Science (2^{nd})*. Prentice-Hall, Inc., 1995.

[11] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer. Springer-Verlag New York, Inc., 2006.

[12] Claudia Ermel, Enrico Biermann, Johann Schmidt, and Angeline Warning. Visual modeling of controlled emf model transformation using henshin. *ECEASST*, 32, 2010.

[13] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M. Küster. Analysis of Model Transformations via Alloy. In *Proc. of MoDeVVa*, 2007.

[14] Geri Georg, Kyriakos Anastasakis, Behzad Bordbar, Siv Hilde Houmb, Indrakshi Ray, and Manachai Toahchoodee. Verification and trade-off analysis of security properties in uml system models. *IEEE Trans. Softw. Eng.*, 36(3):338–356, 2010.

[15] Farida Mostefaoui and Julie Vachon. Verification of aspect-uml models using alloy. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 41–48. ACM Press, 2007.

[16] Bernardo F. B. Braga, João Paulo A. Almeida, Giancarlo Guizzardi, and Alessander Botti Benevides. Transforming ontouml into alloy: towards conceptual model validation using a lightweight formal method. *ISSE*, 6(1-2):55–63, 2010. doi: 10.1007/s11334-009-0120-5.

[17] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. In *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 158–171. Springer, 2009.

[18] Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between alloy specifications and uml class diagrams annotated with ocl. *Software and Systems Modeling*, pages 1–21, 2013.