Modeling and Reasoning with Multirelations, and their encoding in Alloy

Peiyuan Sun, Zinovy Diskin, Michał Antkiewicz, and Krzysztof Czarnecki

University of Waterloo
{p25sun,zdiskin,mantkiew,kczarnec}@gsd.uwaterloo.ca

Abstract. Multisets and multirelations arise naturally in modeling. In this paper, we present a sound and practical mathematical framework, which encodes multisets and multirelations using only ordinary sets and total functions. We implement the encoding as a multiconcepts library in Alloy, which is declarative, compatible with ordinary sets and relations, and can be incorporated into existing models seamlessly.

1 Introduction

Graphical notations are good for direct intuitive modeling but may be difficult to formalize and lack precision. Text-based modeling languages are well-amenable to formalization and formal reasoning, but the meaning of a textual specification may be not immediate and difficult to grasp. Integration of the two frameworks is a challenge; it needs understanding of how they can interact and how their mutual translation could be mediated. In this paper, we investigate a special instance of the story with the following two parties in the two roles.

The graphical modeling party is represented by the UML-style relational modeling, in which a relationship (an association in the UML parlance) between two classes is seen as a set of links represented by arrows. As in general the same two objects can be related by multiple links (the association ends are non-unique), we call such a relationship a *multirelation*. Note that even if we compose two ordinary relations, $R_1 : A \to B$ and $R_2 : B \to C$, the result is, in general, a multirelation because there may be multiple $b \in B$ such that $(a,b) \in R_1$ and $(b,c) \in R_2$. Converting this multirelation into an ordinary one means discarding some data and may be seen artificial and unsatisfactory from the modeling perspective.

The textual modeling party is Alloy, which is specially tailored for relational modeling and analysis. However, Alloy does not provide a direct way to work with multirelations, which requires the modeler to look for a workaround and makes our story intriguing for an Alloy user. Indeed, considering the universe of ordinary relations closed under composition is, as mentioned above, an artificial and often counter-intuitive imposition on relational modeling.

To integrate, we, first, formalize multirelations and operations over them in a category-theoretical framework of *spans* and span operations. Although the framework is well-known in category theory, its accurate presentation in the MDE literature seems to be novel and makes a theoretical contribution of the paper. (For instance,

the fact that the disjoint union of two sets is generally a multiset, and the notion of multiset product, in which elements' roles are freely combined, have not yet been considered in the UML literature up to our knowledge of it.) Second, as the span framework for relational modeling is based on sets and total functions that are readily modeled in Alloy, spans open the door for a seemingly natural Alloy encoding of multirelational modeling. However, implementing the entire multirelational algebra in Alloy as a proper (but conservative) extension of the ordinary relational algebra turned out non-trivial and required solving numerous specific problems of Alloy encoding of elementary operations over sets and functions. To ease using our solution for an Alloy user, we implemented a library of utility modules allowing the user to seamlessly integrate multirelations into existing models; conservativity then ensures that the ordinary relation part of the model is not broken and can be used further if needed.

Our plan for the paper is as follows. In Sect. 2 we present a simple running example of multirelational modeling, and show the main notions at work. Sect. 3 describes the mathematical framework. In Sect. 4, we demonstrate the usage of our Alloy module library by building an Alloy model for the running example. Sect. 5 presents the details of our encoding in Alloy, and in Sect. 6 we discuss how we evaluate the Alloy encoding. Related work is discussed in Sect. 7, and Sect. 8 concludes.

2 Multiconcepts naturally arise in modeling

We use *multiconcept* as a generic term referring to *multisets*, *multirelations*, and specific *multioperations* over them. In this section, we present a simple modeling scenario showing how naturally multiconcepts can appear in modeling. We then formulate several basic requirements for a multiconcept modeling framework.

2.1 Running example: seasonal sales and multirelations

The manager of a grocery store asks employee to prepare some bundles for the coming seasonal sale. A *bundle* contains several food *items*, and each item belongs to a certain product *category*. The manager imposes some rules on the bundle content, e.g., the following three:
(0) Every bundle must have (contains) at least two items
(1) Every bundle must have at least two dairy products;
(2) Every bundle must have items from at least two product categories.

To model the scenario, we identify three classes of objects: Bundle, Item, and Category, and two relations: contains : Bundle → Item and belongsTo : Item → Category as shown in Fig. 1(a) (ignore dashed blue arrows for a while). Note that a bundle may have repeated items according to the label 'non-unique' at the respective association end. A simple instance of this model is shown in Fig. 1(b). In this instance, the relation contains consists of five links c_i shown by arrows. Having two links relating the same pair ($B1$,*Bread*) means that the bundle $B1$ has two breads; we also say that the pair *(B1,Bread)* occurs twice in the relation, and the latter is then called a multirelation. The relation belongsTo is an ordinary relation consisting of three links/pairs b_i (by default, it is considered bearing the 'unique' label).

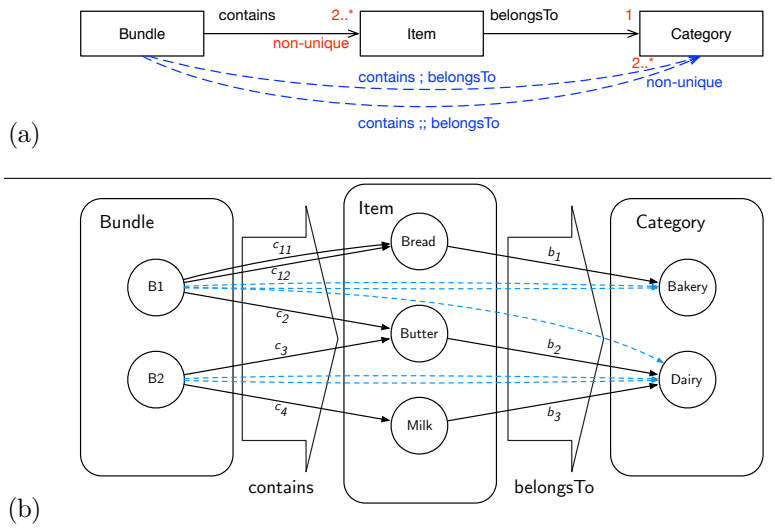


Fig. 1: Bundling: a model (a) and its instance (b)

In order to check the bundling rules described above, we need to compose the two relations and compute a new relation

$$\mathsf{contains;;belongsTo}:\mathsf{Bundle}\to\mathsf{Category},$$

(dashed lower arrow in Fig. 1a), where we use double semi-colon to denote the composition operation. For the instance in Fig. 1b, the composition would consists of five composed links shown by dashed blue arrows, e.g., $c_{11};b_1$, $c_{12};b_1$, etc.. As the bundle $B1$ has only one composed link to Dairy, it violates rule (1), while bundle $B2$ has two Dairy links ($c_3;b_2$ and $c_4;b_3$), and thus satisfies (1). As for rule (2), we can see that bundle $B1$ satisfies whereas bundle $B2$ violates it. A bundle containing one bread, one butter, and one milk would have satisfied both rules.

Now suppose that the bundle $B1$ has only one bread and hence contains would be an ordinary relation. Nevertheless, composition contains;;belongsTo of two ordinary relations would still be a multirelation containing two links from $B2$ to Diary. Hence, there are two types of composition of ordinary relations. One is precise and counts each pair of composable links as a separate composed link as we did above by creating two links from $B2$ to Diary. We call this composition *multijoin* and denote it by double semi-column. The other relational composition only deals with reachability, and links two elements (bundle B2 and Diary in the example) as soon as there is at least one composable pair of links between those elements, but only retains one link irrespective to the number of paths. We call this composition *ordinary join* and denote it by single semi-column. Clearly, ordinary join loses some information, which may be important to consider, e.g., for checking rule (1). Actually, this is a typical situation: computing the price of a bundle composed from prices of its items $\mathsf{price}:\mathsf{Item}\to\mathsf{int}$, or the weight of a bundle, or other aggregate queries (as they are called in the database literature), requires multijoins and multirelations. On the other

hand, there are situations whereby we do need the ordinary join, e.g., checking rule (2) in our example, where we are only concerned with reachability.

2.2 Seasonal sales and multisets

Unary multirelations or *multisets* is also a natural modeling concept. For example, in Fig. 1(b), mapping contains defines bundle $B1$ as a multiset of items, in which item *Bread* occurs twice, and we write $B1 = \{\{Bread, Bread, Butter\}\}$, where double brackets indicate that we deal with multisets rather than sets. Importantly, as a unary multirelation, bundle $B1$ is to be seen as a multi-*subset* of set Item so that the expression above implicitly states that item *Milk* is not included into $B1$. Hence, to be accurate, we should specify the type of a multiset and write $B1$:MSub(Item), where MSub(Item) denotes the set of all multisubsets of Item. Similarly, bundle $B2$ is another multisubset $\{\{Butter, Milk\}\}$ of the same type, which *happens to be* an ordinary (sub)set. The italicized reservation is important if, for example, we need to consider the union of two bundles, which should include item *Butter* twice: $B1 \uplus B2 = \{\{Bread, Bread, Butter, Butter, Milk\}\}$. Thus, union of two ordinary sets seen as multisets can result in a multiset (exactly like composition of two ordinary relations seen as multirelations can be a multirelation). On the other hand, in some contexts we may need an ordinary (non-counting) union operation, in which $B1 \cup B2 = \{\{Bread, Butter, Milk\}\}$. To distinguish these two operations, we will call them *ordinary* union and *multiunion* (similarly to contrasting *ordinary* join and *multijoin* in relational composition). Thus, the universe of ordinary (sub)sets is closed under ordinary union, but is not closed under multiunion. Multiunion is often called disjoint union, but the fact that disjoint union of two sets can result in a multiset is often not recognized.

2.3 Requirements

The discussion above leads to the following requirements for an effective multiconcepts framework. A modeler should be able to do the following on top of the ordinary sets and relations:

1. directly declare a multi(sub)set or a multirelation;
2. perform operations over multiconcepts;
3. control whether the result of an operation should be ordinary or multi.

For example, it should be possible to compute the multijoin of an ordinary relation with a multirelation, or the multijoin of two ordinary relations, or the multiunion of a set and a multiset. Furthermore, the syntax and semantics of the extended framework should be conservative w.r.t. the ordinary operations over ordinary objects, so that the modelers could add multiconcepts to their existing models without having to restructure them.

3 Mathematical framework

The diagram in Fig. 1(b) is intuitively simple: three boxes are sets, and two block arrows represent mappings between them. To make the diagram formal, we need

to formalize (i) the notion of mapping as a collection of links, and (ii) mapping composition. There are two ways of doing this. The first is well-known: a multirelation is a relation, in which each pair of elements is assigned with an integer called the *multiplicity* of the pair; correspondingly, multioperations amount to operations over numbers (see the accompanying TR [12] for details). We refer to this framework as *multiplicity-based* or *numeric*, but it is beyond our goals in the paper. The second approach is borrowed from category theory and is based on reification of links as indices, we call the approach *index-based* and present it below.

We will begin with index-based formalization of relations, because it allows us to introduce the approach in a natural way by discussing the example in Sect. 2. In this way we come to our major construct of *span*. Then we introduce the index-based version of multisets—the notion of a *family* of elements. We will also define operations over spans, operations over families, and mixed compositions of spans with families.

3.1 Relations as spans and operations over them.

We formalize the mapping contains in Fig. 1 by reifying all its constituent links as separate objects. This gives us a set Contains consisting of five elements c_i as shown in Fig. 2. The special nature of these elements (they represent links) is formalized by mapping each of them to the source and the target elements of the respective link. For example, as element $c_1 \in \mathsf{Contains}$ reifies link c_1 from $B1$ to Bread in Fig. 1, it is linked to $B1$ by the *source* link c_{1s}, and to Bread by the *target* link c_{1t}. All source links make a function $\mathsf{sleg}_{\mathsf{Contains}} : \mathsf{Bundle} \leftarrow \mathsf{Contains}$ called the *source leg*, and all target links make a function $\mathsf{tleg}_{\mathsf{Contains}} : \mathsf{Contains} \rightarrow \mathsf{Item}$ called the *target leg*. The triple $(\mathsf{Contains}, \mathsf{sleg}_{\mathsf{Contains}}, \mathsf{tleg}_{\mathsf{Contains}})$ is called a *span* with *head* set Contains and *legs* as above. Similarly, relation belongsTo is formalized by the span with the head set BelongsTo (see Fig. 2).

Definition 1 (span). A *(finite) span* R from a set A to a set B is a triple $(\mathsf{head}_R, \mathsf{sleg}_R, \mathsf{tleg}_R)$ with head_R a set called the *head*, and $\mathsf{sleg}_R : A \leftarrow \mathsf{head}_R$, $\mathsf{tleg}_R : \mathsf{head}_R \rightarrow B$ two functions called *legs*. In formal diagrams, we will often use a shorter notation (H_R, s_R, t_R) for span's components. We denote a span by a stroked arrow $R : A \nrightarrow B$, and will often use the same name for both the span and its head.

If a span represents a total single-valued relation, i.e., a function $f : A \rightarrow B$, its source leg is a bijection (e.g., the relation belongsTo in Fig. 1 is such). As the choice of the index set is arbitrary, we can take set A to be the head, and its identity $\mathsf{id}_A : A \rightarrow A$ as the source leg. Then the target leg is the function f itself.

Note also that sets A and B in Def. 1 are actually placeholders (formal parameters) for sets rather than actual sets. For example, if we want to specify a multirelation $R = \mathsf{Spouse}$ on a set Person[1], then we define $A = \mathsf{Person}$, $B = \mathsf{Person}$, $\mathsf{head}_{\mathsf{Spouse}} = \mathsf{marriageContract}$, $\mathsf{sleg}_{\mathsf{Spouse}} = \mathsf{spouse}_1$ and $\mathsf{tleg}_{\mathsf{Spouse}} = \mathsf{spouse}_2$, where functions spouse_1 and spouse_2 map a contract to the two spouses it binds. Formally, this procedure can be described as binding formal parameters, A, sleg_R etc. to actual

[1] the same two persons can be multiply related, if, e.g., they got diversed and then re-married again, and hence may have several marriage contracts

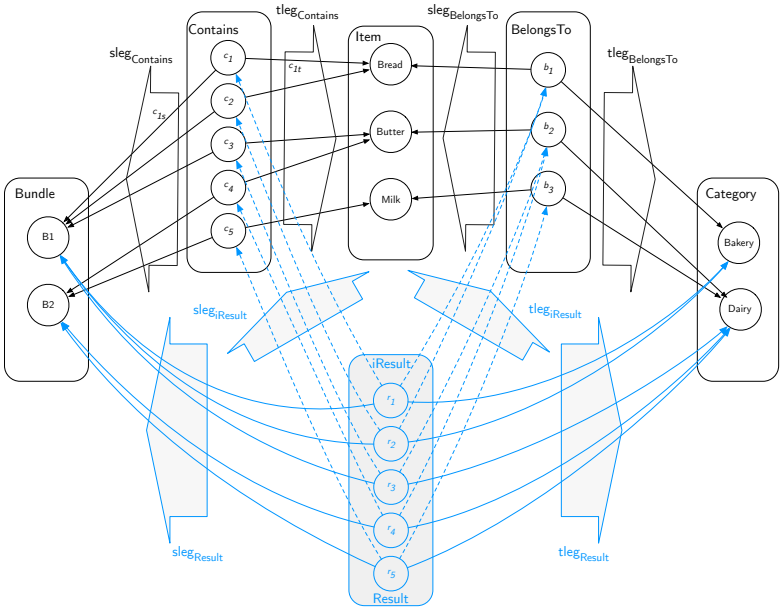


Fig. 2: Indexed-based formalization of Fig. 1

values Person, spouse_1 etc. Nodes and arrows in diagrams used in all our formal definitions below are formal parameters to be substituted by actual values (sets for nodes and functions for arrows), when applied in modeling situations.

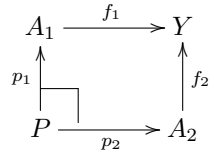
Now we proceed to the index-based formalization of sequential composition of relations. Our discussion of Fig. 1 showed that the core process is composition of links. As links now are reified as elements in the heads of the spans involved, first of all we need to find *composable pairs of links*. It is clear that links $c_i \in \mathsf{Contains}$ and $b_j \in \mathsf{BelongsTo}$ are composable iff the target of c_i is the source of b_j, i.e., $\mathsf{tleg}_{\mathsf{Contains}}(c_i) = \mathsf{sleg}_{\mathsf{BelongsTo}}(b_j)$. If this condition holds, the two links are composable, and we can create a new link from $\mathsf{sleg}_{\mathsf{Contains}}(c_i) \in \mathsf{Bundle}$ to $\mathsf{tleg}_{\mathsf{BelongsTo}}(b_j) \in \mathsf{Category}$. Thus, the set of all composable links iResult (where i stands for 'inner', and later we will also build an outer span)) is given by the following formula:

$$\mathsf{iResult} = \big\{(c_i, b_j) : \mathsf{tleg}_{\mathsf{Contains}}(c_i) = \mathsf{sleg}_{\mathsf{BelongsTo}}(b_j)\big\}.$$

This set is equipped with two projection functions selecting, resp., the first or the second element of pair (c_i, b_j), and we obtain a span iResult shown in Fig. 2 as the inner span. To finish the composition and obtain the resulting outer span Result, we need function composition: $\mathsf{sleg}_{\mathsf{Result}} = \mathsf{sleg}_{\mathsf{iResult}}; \mathsf{sleg}_{\mathsf{Contains}}$ and $\mathsf{tleg}_{\mathsf{Result}} = \mathsf{tleg}_{\mathsf{iResult}}; \mathsf{tleg}_{\mathsf{BelongsTo}}$ (see Fig. 2). Below we present an abstract formal specification of the procedure.

Selection of the composable links is provided by the operation called (in category theory) *pullback* of functions.

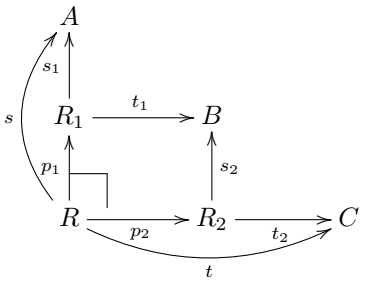
Definition 2 (pullback). The *pullback* of two functions with a common target, $f_1 : A_1 \to Y$ and $f_2 : Y \leftarrow A_2$ is a span with head $P = \{(a_1,a_2) \in A_1 \times A_2 : f_1(a_1) = f_2(a_2)\}$. The legs of the span are projections $p_i : P \to A_i$ with $p_i(a_1,a_2) = a_i$, $i = 1,2$. The diagram on the right shows the respective *pullback square* (note the angle near P denoting such squares).



It is easy to check that such a pullback square is commutative: $p_1;f_1 = p_2;f_2$.

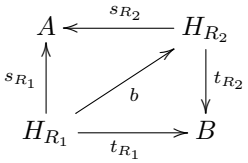
Now we can define span composition.

Definition 3 (span composition). Let $R_1 : A \nrightarrow B$, $R_2 : B \nrightarrow C$ be two composable spans. Their *(sequential) composition* is a span $R_1;;R_2 = (R,s,t) : A \nrightarrow C$ defined as shown on the right, where arc arrows (s and t) denote functions composed from two functions spanned by arcs (we will use this convention further on). In more detail, we first compute the pullback of functions t_1 and s_2 to select all pairs of composable links. Then we build the outer legs by composing $p_1;s_1$ and $p_2;t_2$.



If span R_2 represents a function (total single-valued relation), we can perform span composition more effectively by taking R_2's source leg to be the identity, and hence $\mathsf{tleg}_{R_2} = R_2$. Then we set $\mathsf{head}_{R_1;;R_2} = \mathsf{head}_{R_1}$, $\mathsf{sleg}_{R_1;;R_2} = \mathsf{sleg}_{R_1}$, and $\mathsf{tleg}_{R_1;;R_2} = \mathsf{tleg}_{R_1};\mathsf{tleg}_{R_2}$. This span is isomorphic to any other span representing relation R_2 with an arbitrary index set I bijective to B (because the relation is total and single-valued). For example, as relation $\mathsf{belongsTo}$ in our running example is a function, we can compose $\mathsf{Contains}$ and $\mathsf{BelongsTo}$ in this simpler way by identifying $b_{1,2,3}$ with *Bread*, *Butter* and *Milk* resp. The result will be isomorphic to the span specified in Fig. 1 in the following sense.

Definition 4 (span isomorphism). Two parallel spans $R_1 : A \nrightarrow B$, $R_2 : A \nrightarrow B$ are *isomorphic* if there is a bijection between their heads, $b : H_{R_1} \to H_{R_2}$, such that $s_{R_1} = b;s_{R_2}$ and $t_{R_1} = b;t_{R_2}$. The two commutativity conditions ensure that if a link $h \in H_{R_1}$ is mapped to a link $b(h) \in H_{R_2}$, then both links have the same source and target.

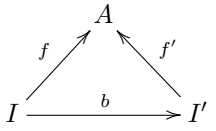


Thus, span composition is defined up to span isomorphism. Moreover, the process of relation indexing by a span is also defined up to isomorphism: we are free in choosing objects reifying links, but the source and target projection links of these objects are uniquely determined by the link being reified. In fact, everything in the indexing world is defined up to natural isomorphisms (bijections between index sets commuting with the respective functions). In this paper, we take particular representatives of equivalence classes defined by isomorphism like above, and perform operations over them. General results of category theory, in which standard operations over sets and functions are redefined via so called limits (e.g., pullback) and colimits (e.g., merge) up to isomorphism ensure that applying these operations to different but isomorphic representatives produces isomorphic results (see, e.g., [5]).

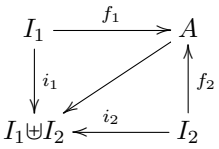
3.2 Multisets as families and operations over them.

We use the same indexing idea by reifying element *occurrences* as unique indices to distinguish the multiple occurrences of the same element. In a sense,

Definition 5 (families). Let A be a set (perhaps, infinite). A *(finite) family* over A is a function $f: I \to A$ from a finite *index* set I to A. The latter is called the *ground* set of family f, while the range set of function f, i.e., set $\{f(i): i \in I\} \subset A$, is often called the *active domain* of f (it is always finite as set I is such). To avoid confusion, we will use the term 'range set' rather than 'active domain'.
Two families, $f: I \to A$ and $f': I' \to A$, over the same ground set A are called *isomorphic*, and we write $f \cong f'$, if there is a bijection $b: I \to I'$ such that $b;f' = f$.



Definition 6 (multiunion/sum/merge). Given two families over A, $f_k: I_k \to A$, $k=1,2$, their *multiunion (*or *sum,* or *merge)* $f_1+f_2: I_1 \uplus I_2 \to A$ is given by the diagonal arrow in the commutative diagram on the right, where $\uplus$ denotes the operation of disjoint union, arrows i_1, i_2 are canonic injections, and the diagonal function is defined by $(f_1+f_2)(i) = f_1(i)$ if $i \in I_1$, and $(f_1+f_2)(i) = f_2(i)$ if $i \in I_2$.



We can use the operation of family multiunion for building disjoint union of sets (not surprisingly, as multiunion uses disjoint union of index sets). Given sets A and B, we form their union $U = A \cup B$ with two injections $i: A \to U$ and $j: B \to U$. These injections can be seen as two families over the same ground set. Summing them gives us a family $u: A \uplus B \to U$, whose index set is the disjoint union of A and B.

3.3 Mixed setting: Families and spans, ordinary and multi

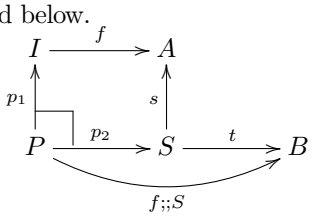
Connections between ordinary and multi(sub)sets are realized via the following two functions. Given a set A, we have function $\mathsf{drop}_A: \mathsf{MSub}(A) \to \mathsf{Sub}(A)$ from multi- to ordinary subsets of A that ignores indexes and only cares about the range set of a family. Conversely, a (trivial) function $\mathsf{lift}_A: \mathsf{MSub}(A) \leftarrow \mathsf{Sub}(A)$ makes an ordinary subset $X \subset A$ into a multiset by taking $I = X$ and $f(x) = x$ for any index x. Clearly, $\mathsf{drop}_A.\mathsf{lift}_A X = X$ for any $X \subset A$, but $\mathsf{lift}_A.\mathsf{drop}_A f \neq f$ as dropping discards the information about family f (where dot denotes function composition). Similarly, we have functions $\mathsf{drop}_{AB}: \mathsf{MRel}(A,B) \to \mathsf{Rel}(A,B)$ from the universe of all multirelations from A to B to the universe of all ordinary relations from A to B, and, conversely, $\mathsf{lift}_{AB}: \mathsf{MRel}(A,B) \leftarrow \mathsf{Rel}(A,B)$ defined in an obvious way.

It is easy to see that the ordinary union of two ordinary sets X,Y can be presented as $\mathsf{drop}(\mathsf{lift}X + \mathsf{lift}Y)$. Similarly, ordinary composition of two ordinary relations $X \subset A \times B$, $Y \subset B \times C$ is $\mathsf{drop}(\mathsf{lift}X;;\mathsf{lift}Y)$. Of course, an effective implementation of the ordinary operations should *not* follow these patterns.

In ordinary relational modleing, given a relation $R: A \nrightarrow B$, we often need to build the R-image of a subset $X \subset A$, and the R-preimage of a subset $Y \subset B$. In

multi-modeling, subsets are families and relations are spans, hence, we need the notions of the image and preimage of family w.r.t. a given span. Remarkably, both notions can be formally defined via pullbacks as described below.

Definition 7 (composing a family with a span). Given a family $f: I \to A$ and a span $S: A \nrightarrow B$, their *composition* is a family $f;;S: P \to B$ defined by the diagram on the right (recall that arc arrows denote functions obtained by composition of two functions spanned by the arc). This family is also called the *S-image* of f.



The *(inverse) composition* of span $S: A \nrightarrow B$ with a family $g: J \to B$ is a family $S;;g: P \to A$ defined by a diagram dual to the above: we begin with pullback of g and t, which gives us a pair of arrows (q_1, q_2), and then set $S;;g = q_1;s$. This family is called the *S-preimage of g*.

3.4 Composition as navigation

Span composition can be also defined in a more navigational way. Given a span $R: A \nrightarrow B$, we can represent it as a function $\phi: A \to \mathsf{MSub}(B)$ by represeting an element $a \in A$ as a "family" $a^*: \{*\} \to A$ with $a^*(*) = a$, and defining $\phi(a) = a^*;;R$. This is nothing but a special case of the general image-operation described in Def. 7. The latter can be described as a function $\phi^M_R : \mathsf{MSub}(A) \to \mathsf{MSub}(B)$. Now, if we have spans $R1: A \nrightarrow B$ and $R2: B \nrightarrow C$, we represent them as functions $\phi_{R1} : A \to \mathsf{MSub}(B)$ and $\phi_{R2} : B \to \mathsf{MSub}(C)$ resp. Span composition is then represented by the function composition $\phi_{R1}.\phi^M_{R2}$, which is not difficult to show equals to $\phi_{R1;;R2}$ (see e.g. [3] for an elementary proof). That is, given an element $a \in A$, its $R1;;R2$-image (which is a family (multiset) over C) can be computed either relationally as $(a^*;;R1);;R2 = a^*;;(R1;;R2)$, or navigationally as $a.\phi_{R1}.\phi^M_{R2}$. Note also that the special case when multirelation $R1$ is not defined on a is well treated without exclusion, because then multiset $\phi_{R1}(a)$ will be empty (i.e., an empty family given by an empty function $\emptyset: \emptyset \to B$), and $\phi^M_{R2}(\emptyset) = \emptyset$.

The description above actually shows that span composition effectively prevents the NULL-navigation safety issue occurring in many textual languages such as OCL [13]. Indeed, we use empty multisets to represent the case of non-existence similarly to how other languages use NULLs. Any composition with an empty multiset results in an empty multiset without special treatment. Hence, the navigation is always safe.

4 Demonstration

In this section, we demonstrate the usage of our index-based multiconcepts library in Alloy to model the scenario introduced in Section 2. The commented Alloy code below is the full model.

```
1 open multi                                    // a utility module providing the function drop
2
3 open mrel[Bundle, Item] as Contains           // declare a multirelation from Bundle to Item
4 open mrel[Item, Category] as BelongsTo        // declare a multirelation from Item to Category
5 open mrel[Bundle, Category] as Result         // to be the result of Contains;;BelongsTo
6
7 abstract sig Bundle {}                                                    // Bundle class
```

```
8  one sig B1, B2 extends Bundle {}                                    // Bundle instances
9  abstract sig Item {                                                 // Item class
10   belongsTo: set Category          // declare an ordinary relation from Item to Category
11 }
12 one sig Bread, Butter, Milk extends Item {}                         // Item instances
13 fact {
14   Bread.belongsTo = Bakery                                  // Set up belongsTo relation
15   Butter.belongsTo = Dairy
16   Milk.belongsTo = Dairy
17 }
18 abstract sig Category {}                                          // Category class
19 one sig Dairy, Bakery extends Category {}                       // Category instances
20
21 fact {
22   BelongsTo/liftedFrom[belongsTo]          // lift the ordinary relation to a multirelation
23   Result/composedFrom[Contains/get, BelongsTo/get]                // perform the multijoin
24   // rules
25   all b : Bundle | #(b<:Result/get:>Dairy) >= 2             // at least two dairy products
26   all b : Bundle | #(b.(drop[Result/get])) >= 2                 // at least two categories
27 }
28
29 assert AllHaveBread {              // based on our rules, all bundles should contain bread
30   all b: Bundle | some drop[b<:Contains/get].Bread
31 }
32
33 run {} for 6 Contains/Head, 3 BelongsTo/Head, 20 Result/Head
34 check AllHaveBread for 6 Contains/Head, 3 BelongsTo/Head, 20 Result/Head
```

Listing 1.1: The bundling model

The goal of this example is to show two typical usages of Alloy: instance finding and assertion checking. For instance finding, we ask the solver to find instances of the multirelation `Contains` (line 3) such that the rules (lines 25-26) are satisfied. Based on the rules, each bundle must contain some bread; the assertion `AllHaveBread` (line 29-31) is to check this fact.

The module `mrel` allows for declaring multirelations for the given domain and range signatures (lines 3-5). Such a multi-relation can be lifted from an ordinary relation by assigning each tuple in the ordinary relation a unique index. In our example, the ordinary relation `belongsTo` declared on line 10 is fixed on lines 13-17. The multirelation `BelongsTo` (declared at line 4) represents the same information as the ordinary relation `belongsTo` by lifting (line 22).

We need to multijoin `Contains` and `BelongsTo` to observe the categories of products in each bundle. Line 23 shows how an unconstrained multirelation `Contains` is multijoined with `BelongsTo`; the result is stored in `Result`. We then use the built-in domain and range restriction operators `<:` and `:>` and the function `drop` (drop the indices in a multirelation to make it ordinary) to state the rules "every bundle must contain at least two dairy products" (we care about multiplicity in this case) and "every bundle must contain items covering at least two product categories" (we do not care about multiplicity in this case). Fig. 3 shows a bundling instance generated by Alloy analyzer which satisfied the two rules.

5 Index-based multiconcepts library in Alloy

In this section, we present how we encode the index-based formalization of multiconcepts introduced in Section 3. We demonstate how we leverage language features of

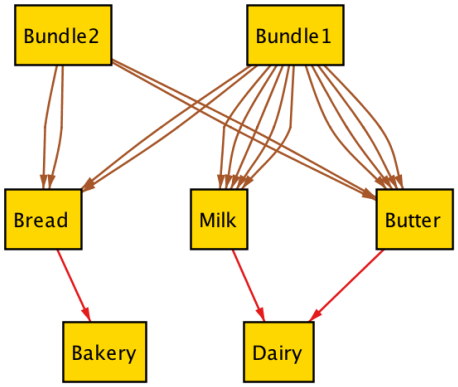


Fig. 3: A bundling instance generated by Alloy analyzer

Alloy to design a multiconcept library following the mathematical framework while emphasizing clarity and ease of use. The library comes in three parts: two parametric modules named `mset` and `mrel` for declaring multisets and multirelations; one plain module named `multi` providing operations over multiconcepts and a few utility functions.

5.1 Encoding of family and span

Family and span are naturally generic structures. By specifying the type of the ground set in a family or the types of the source and target in a span, we can obtain concrete family or span of specific types. Such polymorphism is provided by parametric modules in Alloy. We employ this language feature to encode family and span:

```
1 module mset [g]
2
3 sig Idx { f : one g }
4 fun get[] : Idx -> one g { f }
```

```
1 module mrel [s, t]
2
3 sig Head { sLeg: one s, tLeg: one t }
4 fun get[]: s -> Head -> t
5  { a:s, h:Head, b:t| h.sLeg=a && h.tLeg=b }
```

The first line of both listings declares a parametric module: module `mset` has one type parameter `g` standing for the ground set of a family; module `mrel` has two type parameters `s` and `t` standing for the source and target of a span.

Family is encoded by the index set `sig Idx`, which has a totally-defined single-valued relation `f` (restricted by the quantifier `one`) to ground set `g`. Span is encoded by the head set `sig Head`, which has two totally-defined single-valued relations (restricted by the quantifier `one`) `sLeg` and `tLeg` to source set `s` and target set `t`.

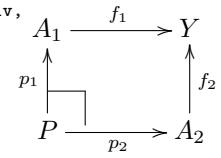
A family can be viewed as a totally-defined single-valued binary relation; a span can be viewed as a ternary relation by constructing a triple in which head set sits in the middle column along with source and target on the sides. Based on the views above, each module exposes a function `get` which returns its internal structure as a binary or a ternary relation (line 4). Returning them in such form has two benefits. First, in the module `multi`, we provide a few generic predicates and functions, which

work on both families and spans. These predicates and functions can be generic because we represent families and spans as ordinary binary and ternary relations and we can type the parameters of these predicates and functions as `univ->univ` or `univ->univ->univ`. Second, our encoding of multiconcepts is also automatically compatible with ordinary sets and relations. For example, a span can be effectively composed with a total single-valued relation by joining its target leg with that relation. Since we have returned the span as a ternary relation, the effective composition can simply be done by the built-in relational join operator (`.`).

5.2 Encoding of composition

The very important operation of composition over families and spans is pullback, implemented as a predicate in the following listing which takes all the sets and relations in the pullback diagram as input parameters.

```
1 // in module multi
2 pred pullback[ X1: univ, X2: univ, f1: X1 -> one univ, f2: X2 -> one univ,
3                P: univ, p1: P -> X1, p2: P -> X2 ] {
4   (no X1 or no X2) implies { no P } else {
5   all x1: X1 | all x2: X2 | x1.f1 = x2.f2 implies
6     { one p: P | p.p1 = x1 && p.p2 = x2 }
7   #P = #(f1.~f2)
8   }
9 }
```



In the pullback diagram above, we assume the sets `X1`, `X2` and `Y` with the functions `f1` and `f2` are given. By setting proper constraints, the solver will generate all correct instances in set `P` together with functions `p1` and `p2` to form a pullback square.

We first deal with the case when set `X1` or set `X2` is empty (line 4), which implies set `P` is also empty. The constraint in line 5 and 6 simply obeys the definition of pullback. We iterate the elements in set `X1` and `X2`. If elements `x1:X1` and `x2:X2` satisfy the condition `x1.f1 = x2.f2`, it implies that there must be one element `p` in set `P` which makes the pullback diagram commute by applying the constraint `p.p1=x1 && p.p2=x2`.

This constraint sets a lower bound to the solution space. With this constraint solely, the solver will generate all the correct instances in set `P`, and possibly some garbage instances due to the lack of an upper bound constraint. We set an upper bound to the solution space using the fact that the cardinality of set `P` is equal to the cardinality of the composition result `f1.~f2` according to the definition of pullback.

We use the pullback predicate to implement the composition. Before we go into details, we should notice that set P is fresh and disjoint with any other set in the pullback diagram, this is why pullback cannot be encoded as a function in Alloy because we cannot return a set that has not been declared before in a function. This is also the reason that we need to manually declare a multiset or a multirelation to hold the result of composition.

```
1  // in module mset [g]
2  open multi as m
3  // current family is composed from fami & span
4
5  pred composedFrom[fami: univ -> one univ,
6                    span: univ -> univ -> g ] {
7    let I = fami.idx, Hd = span.head,
8        sLeg = span.sleg, tLeg = span.tleg
9    | some p1: Idx->I, p2: Idx-> Hd
10   | pullback[I, Hd, fami, sLeg, Idx, p1, p2]
11  && f = p2.tLeg
12 }
```

```
1  // in module mrel [s, t]
2  open multi as m
3  // current span is composed from span1 & span2
4  pred composedFrom[ span1: s -> univ -> univ,
5                     span2: univ -> univ -> t ] {
6    let   Hd1=span1.head,   Hd2=span2.head,
7        sLeg1=span1.sleg, tLeg1=span1.tleg,
8        sLeg2=span2.sleg, tLeg2=span2.tleg
9    | some p1: Head -> Hd1, p2: Head -> Hd2
10   | pullback[Hd1,Hd2, tLeg1,sLeg2, Head,p1,p2]
11  && sLeg = p1.sLeg1 && tLeg = p2.tLeg2
12 }
```

Predicates `composedFrom` encodes the composition in Def. 3 and 6 respectively. They restrict the current family or span to be the result of the composition of the given input parameters. We use the utility functions to extract the components of a family or a span. We then apply the pullback predicate on the corresponding sets and relations (line 10) and get the result family or span (line 11). We rely on skolemization performed by Alloy which enables higher-order quantification with an existential quantifier to generate the relations `p1` and `p2` on the fly without explicitly declaring them (line 9).

5.3 Other multioperations

In our encoding, multiunion (merge) is simply a union of two families (assuming their index sets are disjoint):

```
1  // in module multi
2  fun merge[f1, f2: univ->one univ] : univ->one univ { f1 + f2 }
```

Since we transform a span to a ternary relation, it is straightforward to implement the operation inverse by simply flipping the first and third column of the ternary relation; the built-in domain and range restriction operators are directly applicable to the returned ternary relation.

```
1  // flip the 1st and 3rd column
2  fun inverse[span: univ->univ->univ] : span {
3    ter/flip13[span]
4  }
5  // For the domain and range restriction, we can use the built-in operators <: and :>
```

As discussed in Section 3.3, an ordinary set can be viewed as a multiset by assigning each element a unique index. In the same way, an ordinary relation can be seen as a multirelation. Therefore, we can transform an ordinary set or relation to a multiset or multirelation in our encoding. The operation *lift* is implemented as follows:

```
1  // in module mset [g]
2
3  pred liftedFrom[ G: g ] {
4    Idx.f = G && #Idx = #G
5  }
```

```
1  // in module mrel [s, t]
2
3  pred liftedFrom[r: s -> t] {
4    (~sleg).tleg = r && #Head = #r
5  }
```

Conversely, we can transform a multiset or multirelation to an ordinary set or relation by *dropping* the indices, which is encoded as follows:

```
1  // in module multi
2  fun drop[f: univ -> one univ] : univ { rel/ran[f] }
```

```
3
4 fun drop[span: univ -> univ -> univ] : univ -> univ {
5   ter/select13[span]
6 }
```

We can see that our library fullfils the requirements from Section 2.3: it provides 1) a way to directly declare a multiset or a multirelation; 2) an algebra of operations over multiconcepts; and 3) the lift and drop operations to control whether the result is ordinary or multi.

6 Evaluation

6.1 Correctness

We assess the correctness of our encoding through testing, which uses assertions to check if the result obtained from the encoded operation is equal to the expected result we manually compute. By running the assertions in Alloy, if a counter-example is found, that means the encoding of the operation is incorrect, otherwise, the encoding passes the test. All the encoding of operations have passed the tests which increases the confidence that the implementation is correct, and we have released the test suite together with the library [12].

6.2 Performance

One of the core of the Alloy reasoning framework is instance finding, but dealing with multiconcepts significantly increases the search space. In a nutshell, suppose we are given sets A,B and a relationship $R: A \nrightarrow B$. If R is ordinary, then for every pair $(a,b) \in A \times B$, there are only two possibilities: either $(a,b) \in R$ and hence there is a link from a to b, or R does not include such a link. In contrast, if R is a multirelation, for every pair (a,b) there are infinitely many possibilities for the number of links from a to b. Hence, multiplicity constraints play an important role in narrowing the search space in order to gain reasonal performance on reasoning.

We use the bundling model in Sect. 4 as the benchmark model. By manually controlling the scope of `Contain/Head` (Line 33), which sets an upper bound of the multiplicity of multirelation `Contain`, we investigate how the performance of Alloy analyzer's instance finding on the model relates with the multiplicity of a multirelation.

The results are shown in Table 1 and the visualization is shown in Fig. 4. The data includes the size of the model's CNF encoding (vars and clauses) and the total analysis time. The Alloy analyzer is configured to use the MiniSAT solver and runs on a 2.4Ghz Core i7 machine with 8GB RAM.

According to the diagram, the size of the model increases linearly with the scope of `Contains/Head`. The analysis time is fluctuating with the increase of scope but the overall analysis time is momentary when the scope is under 20. This result shows our Alloy multiconcept encoding is practical given a reasonable scope.

scope	vars	primary vars	clauses	analysis time (ms)
5	6860	314	13011	38
6	7496	340	14701	46
7	8128	366	16418	47
8	8756	392	18157	81
9	9378	418	19909	82
10	10000	444	21701	27
15	13012	574	31228	379
20	15748	704	41150	68

Table 1: Benchmark results (not exhaustively listed) and visualization

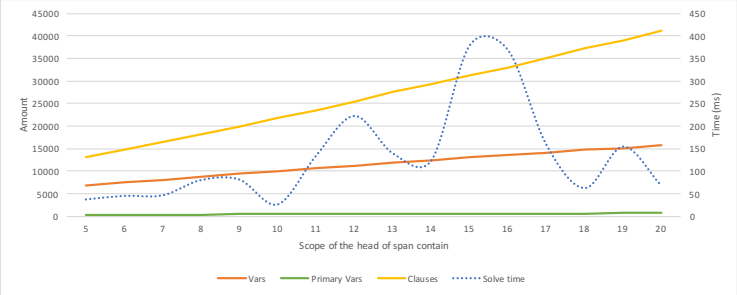


Fig. 4: Visualization of the benchmark result

7 Related Work

Translating UML class diagrams with OCL constraints (involving bags) into Alloy has been studied in [2] and [9]. In the former paper, bags/multisets were encoded in Alloy as sequences, which is inadequate as bags are not ordered. Neither of the papers considered translation of multirelations, which, as our example in Sect. 2 shows, restricts the practical applicability of the studies.

From the Alloy world side, the closest related work is a question: "Are there multisets in Alloy?" [1] on the website Stack Overflow. The selected answer to this question proposes to represent a multiset as a function from a set of elements to the natural numbers. For a given element, the function returns the multiplicity of the element. The answer also provides implementation of merge, intersection, and difference, which compute the new multiplicities of the results (add the multiplicities for the merge, subtract them for difference, and take their minimum for intersection). The answer does not provide any information about multirelations and other operations over them such as composition.

There is a large body of work about multisets treated numerically via multiplicities. Blizard presents a historical survey of the multiset theory [6], and traces back the idea of multisets as families to [11]. A category theoretic treatment of multisets and multirelations can be found in, resp., [10] and [7].

8 Conclusion

We have shown that simple categorical notions of family, span, and their composition via pullback can bridge the gap between graphical multirelational modeling and textual modeling in Alloy. Alloy users can now use multiconcepts in their models by importing our library module. There is previous work about transforming Alloy model to SMT [8]. Our encoding is also a blueprint for implementing multiconcept support for Satisfiability Modulo Theories (SMT) solvers, whereby only total functions are available.

In the future, we will use the presented encoding to implement support for multiconcepts as first-class constructs in the modeling language Clafer [4], which uses Alloy as one of the backends. In Clafer, modelers can directly declare multisets and multirelations; however, both the syntax of Clafer has to be extended with the new operators over multi-concepts and the translation to Alloy has to implement the correct semantics.

References

1. Are there multisets in Alloy? (2014), `http://stackoverflow.com/questions/23579928/are-there-multisets-in-alloy`, last accessed in Aug 2016
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software & Systems Modeling 9(1), 69–86 (2010)
3. Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wąsowski, A.: Partial instances via subclassing. In: International Conference on Software Language Engineering. pp. 344–364. Springer (2013)
4. Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wąsowski, A.: Clafer: unifying class and feature modeling. Software & Systems Modeling (2014)
5. Barr, M., Wells, C.: Category theory for computing science, vol. 49. Prentice Hall New York (1990)
6. Blizard, W.D., et al.: The development of multiset theory. Modern logic 1(4), 319–352 (1991)
7. Bruni, R., Gadducci, F.: Some algebraic laws for spans. Electr. Notes Theor. Comput. Sci. 44(3), 175–193 (2001)
8. El Ghazi, A.A., Taghdiri, M.: Relational reasoning via smt solving. In: International Symposium on Formal Methods. pp. 133–148. Springer (2011)
9. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using alloy revisited. In: MoDELS, pp. 592–607. Springer (2011)
10. Monro, G.: The concept of multiset. Mathematical Logic Quarterly 33(2), 171–178 (1987)
11. Rado, R.: The cardinal module and some theorems on families of sets. Annali di Matematica pura ed applicata 102(1), 135–154 (1975)
12. Sun, P., Diskin, Z., Antkiewicz, M., Czarnecki, K.: Modeling and reasoning with multisets and multirelations in Alloy. Tech. Rep. GSDLAB-TR 2016-01-22, Generative Software Development Lab, University of Waterloo (January 2016), `http://gsd.uwaterloo.ca/node/658`
13. Willink, E.D.: Safe navigation in ocl. In: OCL 2015–15th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations Workshop Proceedings. p. 81 (2015)