Modeling and Reasoning with Multisets and Multirelations in Alloy

by

Peiyuan Sun

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

Abstract

Multisets and multirelations arise naturally in modeling; however, most modeling languages either have limited or completely lack support for multisets and multirelations. Alloy, for instance, is a lightweight relational modeling language which provides automatic analysis of models. In Alloy, ordinary sets and relations are the only first-class language semantic constructs; therefore to work with multisets and multirelations, modelers need to invent ad-hoc ways to encode these multiconcepts or rely on a third-party library that provides their implementations, assuming there is such one. In fact, such a library has been missing for Alloy, and implementing a fully functional multiconcepts library is challenging, especially when it is required to encode an algebra of operations over multiconcepts. This thesis presents two sound and practical mathematical formalizations of multiconcepts, namely, index-based and multiplicity-based, which encode multisets and multirelations using only basic concepts such as ordinary sets, total functions and natural numbers. We implement two generic multiconcepts libraries in Alloy based on the corresponding formalizations. Each library has a carefully designed interface and can be seamlessly integrated into existing relational models. We also perform an empirical evaluation on both implementations; the result shows multiplicity-based encoding is more scalable in terms of performance; thus, it is more preferable in practice.

Acknowledgements

I would first like to express my gratitude to my supervisor Krzysztof Czarnecki for the guidance and support throughout my master study.

Furthermore, I would like to thank Zinovy Diskin and Michał Antkiewicz for the collaboration and inspiring input on this research topic.

I would also like to thank Grant Weddell and Derek Rayside as the readers of this thesis, and I am grateful to their valuable comments on this thesis.

Finally, I must express my very profound gratitude to my mother for providing me with unfailing support and continuous encouragement throughout my years of study. Thank you.

Dedication

This is dedicated to my mother and all the people I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A multiset (also known as bag) is a collection in which an element can occur multiple times. A multirelation is a relation in which the same two elements can be related more than once. We will refer to constructs with multiple occurrences as multiconcepts. Multiconcepts often appear in modeling (see Sect. 1.1 for a motivating example), and some standard modeling languages, such as UML and OCL, have partial support for multiconcepts by allowing multiple occurrences (e.g., `nonunique` annotated association in UML and the bag collection type in OCL), but usually lack operations over multiconcepts. In other modeling languages such as Alloy, only ordinary sets and relations are first-class language constructs, which means there is no direct way to work with multiconcepts and encoding is required. Furthermore, in order to effectively model with multiconcepts, one must also encode an algebra of operations over multiconcepts (like relational algebra for ordinary relations).

As far as we know, a systematic and principled encoding of multiconcepts in Alloy is not present in literature. This thesis presents two theoretically sound and practically feasible multiconcept formalizations. One is commonly known as the numeric approach, using natural numbers to represent multiplicities (number of links between two objects); the other formalization is under category-theoretical framework of span operations. Although the framework is well-known in category theory, its accurate presentation in the MDE literature and application to Alloy is novel. Both formalizations are based on simple mathematical concepts: sets, relations (more specifically, total functions) and natural numbers, which can be readily modeled in Alloy. Algebraic operations over multiconcepts are covered completely in both formalization; however, implementing the entire multirelational algebra in Alloy as a proper (but conservative) extension of the ordinary relational algebra turns out to be non-trivial and requires solving numerous specific problems of Alloy encoding of elementary operations over sets and functions. To ease using our solution in Alloy, we implemented both formalizations as generic libraries allowing users to seamlessly integrate multirelations into existing models with minimal effort; conservativity then ensures that the ordinary relation part of the model is not broken.

In a nutshell, our Alloy library implementations cover multisets, multirelations, and an algebra of operations over them. The encodings are composable, and compatible with ordinary sets and relations — operations over the latter appear as specializations of their multiconcept counterparts, which means one can integrate our multiconcept encodings without breaking the existing model and losing the reasoning ability of Alloy analyzer.

1.1 A Running Example

We start with a simple scenario to motivate the concept of multisets and multirelations in domain modeling. We will keep using this example in the future chapters:

> A manager of a grocery store asks the employees 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 two rules on the content of each bundle:

- Every bundle must contain at least two dairy products;
- Every bundle must contain items from at least two product categories.

To model this scenario, we identify three classes of objects: Bundle, Item, and $\mathsf{Category}$, and two relations between them, $\mathsf{contains}\colon \mathsf{Bundle} \to \mathsf{Item}$ and $\mathsf{belongsTo}\colon \mathsf{Item} \to \mathsf{Category}$. A simple instance of this model is shown in Fig. 1.1. In this instance, the relation $\mathsf{contains}$ consists of five links c_i shown by arrows. Having two links relating the same pair $(B1, \mathit{Bread})$ of elements 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 $\mathsf{belongsTo}$ is an ordinary relation (a Cantorian set of pairs) consisting of three links/pairs b_i.

In order to check the bundling rules given above, we need to compose the two relations resulting in a new relation

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

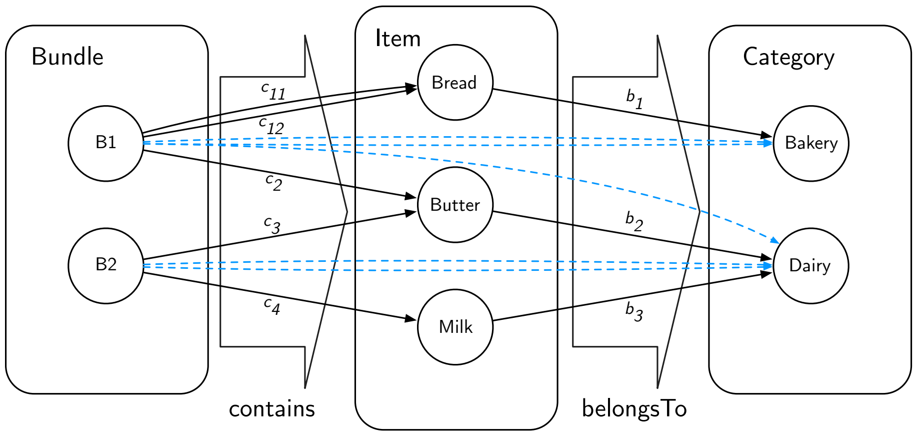


Figure 1.1: An instance of bundling

where we use semi-colon to denote the composition operation. For the instance in Fig. 1.1, the composition would consist of five composed links shown by dashed blue arrows (e.g., $c_{11};b_1$ and $c_{12};b_1$). As the bundle $B1$ has only one composed link to Dairy, it violates the first rule, while bundle $B2$ has two Dairy links ($c_3;b_2$ and $c_4;b_3$), and thus satisfies the first rule. Also, we can see that the bundle $B1$ satisfies the second rule (at least two categories), whereas the bundle $B2$ violates the rule. A bundle containing one bread, one butter, and one milk would be a valid instance satisfying 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 could still be a multirelation containing two links from $B2$ to Diary. Actually, 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 $c_3;b_2$ and $c_4;b_3$, from $B2$ to Diary. The other one is known as traditional composition of relations in relational algebra, which 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 (from $B2$ to Diary). The former composition may result in a multirelation (and we call such composition *multijoin*), the latter is always an ordinary relation (and we call it *ordinary join*). Clearly, this latter composition loses some information, which may be important to consider, e.g., for checking the first rule in our example. In fact, this is a typical situation: computing the price of a bundle composed from prices of its items price: Item $\rightarrow Int$, or the weight of a bundle, or other aggregate queries (as they are called in the database literature), requires multijoin and multirelations. On the other hand, there are situations whereby ordinary join is apt enough. Consider, e.g., checking the second rule in our example when we are only concerning with reachability.

1.1.1 Requirements

The discussion above leads to the following requirements for an effective multiconcepts framework in a modeling environment. Modelers should be able to do the following on top of ordinary sets and relations:

- directly declare a multiset or a multirelation;
- perform operations over multiconcepts, especially composition;
- 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. 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.

1.2 Contributions

On the theoretical side, we revisit the common multiset and multirelation theory based on numerical encoding; in addition, we present another formalization based on category theory. Both formalizations are general purpose and can be used as blueprints for modeling languages to implement utility libraries to support multiconcepts.

Another main contribution is our multiplicity-based and index-based multiconcepts libraries in Alloy, both of which support multiconcepts declaration and a complete set of operations over multiconcepts. The multiplicity-based implementation has better performance and is preferable to be used in practice. We release our multiconcept libraries to Alloy community and users can work with multiconcepts by simply importing our libraries. Additionally, since our implementations cover algebraic operations over multiconcepts, these encoded operations can be used as semantic primitives to design a modeling language which treats multisets and multirelations as first-class constructs and uses Alloy as the target language for code generation.

1.3 Related Work

There is a large body of work about multisets treated numerically via multiplicities. Blizzard presents a historical survey of the multiset theory [7], and traces back the idea of multisets as families to [17]. A category-theoretic treatment of multisets and multirelations can be found in, resp., [8] and [15].

Multisets and multirelations have not been researched extensively in the field of modeling languages. We did not found any literature presenting a systematic study on multiconcepts in Alloy or any other language. The closest related work is a discussion thread on the website Stack Overflow; the title of the discussion is "Are there multisets in Alloy?" [1], asking if there is an explicit notion of bags provided in Alloy or any workaround that can model multisets. Two answers are given in the thread, each of which attempts to model multisets using the numerical approach, using a function returning the multiplicity of a given element. The answers also provide implementations of operations such as union and intersection on multisets; however, the proposed implementations are flawed, yielding incorrect results in certain cases. No other operations such as composition are provided in the discussion.

Multisets and multirelations arise when translating a modeling language which supports multiconcepts to another language. Anastasakis et al. discuss a model transformation from UML with OCL constraints to Alloy [2]. In their transformation approach, the bag collection type in OCL is encoded using the built-in sequence utility module in Alloy. Even though sequences allow multiple occurrences of the same element in a collection, this approach is not optimal since by definition bags are unordered collections while sequences are ordered. Besides, to the best of our knowledge, none of the UML to Alloy translation approaches in existing literature (including [2] and [14]) covers the `nonunique` annotation on associations, which allows declaring multirelations.

Multisets and multirelations are used in various fields. Feijs et al. have developed a relational approach for software architecture analysis [11] and further generalized the approach with multirelations [10]. Petri Nets and its variations such as Coloured Petri Nets use multisets to define the concept *plates*, one of the components in Petri Nets [13] [16]. Robles et al. model Petri Nets in Alloy[18], using multisets implicitly. Baldan's thesis [5] on Petri Nets and graph grammars includes a categorical formalization of multiconcepts, which is the core concept to define morphisms of Petri nets.

1.4 Structure of the Thesis

The rest of the thesis is organized into four chapters. Chapter 2 presents both formalizations of multiconcepts, starting with the numeric approach followed by the index-based categorical framework. Chapter 3 briefly introduces Alloy and then demonstrates the implementations of both formalizations in Alloy in detail. Chapter 4 evaluates both Alloy implementations in terms of usability and performance; a system architecture example is also presented as an application of multiconcepts. In Chapter 5 we conclude.

Parts of Chapter 2 and Chapter 3 were published as a paper [19] .

Chapter 2

Mathematical Foundation

In this chapter we present two different approaches to formalize multisets and multirelations. The first one is based on numeric-valued functions; we will refer to it as *multiplicity-based* or *numeric*. The second is borrowed from category theory and it is based on reification of links as indices; we will call the approach *index-based*. We introduce the two formal approaches and then discuss their relationships from the formal and modeling perspectives.

2.1 Multiplicity-based Formalization

2.1.1 Multisets and Operations

Representing multisets as numeric-valued function is a common approach. Intuitively, the approach represents a multiset as a mapping from a set of elements (ground set) to a set of numbers to carry the information about the multiplicity of each element. The formal definition is shown as follows.

Definition 1 **(Multiset)**
A *(finite) multiset* over a set A is a function $m\colon A \to \mathbb{N}$ ($\mathbb{N}$ denotes the set of natural numbers including zero). Set A is called the *ground*, and m is the *multiplicity* function.

As we do not exclude zero multiplicities, m actually defines a multi-subset of the ground. The set $\{a{\in}A\colon\ m(a) \neq 0\}$ is called the *base* of m, and it is an ordinary subset. Thus, we define a function $\mathsf{drop}_A\colon \mathsf{MSub}(A) \to \mathsf{Sub}(A)$ from multi- to ordinary subsets of A. Conversely, a (trivial) function $\mathsf{lift}_A\colon \mathsf{MSub}(A) \leftarrow \mathsf{Sub}(A)$ makes an ordinary subset into a multi-one with all multiplicities equal to 1. Subindex A will be often skipped. Clearly, $\mathsf{drop}.\mathsf{lift}X = X$ for any subset X, but $\mathsf{lift}.\mathsf{drop}A \neq A$ as dropping discards the information about A (where dot denotes function composition).

Definition 2 (**Union and Intersection**)
Let m and n be two arbitrary multisets over the same ground set A. There are two union operations. The first, *max-union*, is denoted by $\mathsf{max}(m, n)$ and defined by $\mathsf{max}(m, n)(a) = \mathsf{max}(m(a), n(a))$. Second, *add-union*, is denoted by $m+n$ and defined by $(m+n)(a) = m(a) + n(a)$. There are two intersection operations. The first, *min-intersection*, is denoted by $\mathsf{min}(m, n)$ and defined by $\mathsf{min}(m, n)(a) = \mathsf{min}(m(a), n(a))$. Second, *mult-intersection*, is denoted by $m{\times}n$ and defined by $(m{\times}n)(a) = m(a) \times n(a)$.

Let us see what these operation mean for ordinary sets lifted to multisets. Suppose we have two subsets $X, Y \subset A$. It is easy to see that $\mathsf{drop}(\mathsf{max}(\mathsf{lift}X, \mathsf{lift}Y)) = X \cup Y$ and $\mathsf{drop}(\mathsf{min}(\mathsf{lift}X, \mathsf{lift}Y)) = X \cap Y$, which explains the names of the operations.

Now note that $\mathsf{lift}X + \mathsf{lift}Y$ is a representation of the disjoint union $X \uplus Y$—we will discuss this in more detail when we introduce our indexed formalism. It is also worth noting that $\mathsf{lift}X + \mathsf{lift}Y$ is a multi- rather than ordinary set, and class $\mathsf{Sub}(A)$ where A is an ordinary set is not closed under add-unions. The usual disjoint union notation hides this fact and can be confusing. (Indeed, with a precise categorical formalization of the disjoint union as the coproduct operation, disjoint union is a *diagram* of sets and functions rather than a set.)

Finally, $\mathsf{lift}X \times \mathsf{lift}Y$ coincides with $\mathsf{min}(\mathsf{lift}X, \mathsf{lift}Y)$ and thus amounts to the same set $X \cap Y$.

2.1.2 Multirelations and Operations

Base on the definition of multisets, a (binary) multirelation is defined as a multiset over a Cartesian product.

Definition 3 (**Multirelation**)
A binary *multirelation* from set A to set B is a function $m\colon A \times B \to \mathbb{N}$. Number $m(a, b)$ is called the *multiplicity* of pair(a, b). Sets A and B are called the *source* and, respectively, the *target* of m. To stress the different roles of sets A and B, we denote a multirelation by an arrow $m\colon A \to_{\mathbb{N}} B$.

Similarly to multisets, we have a function $\mathsf{drop}_{AB}\colon \mathsf{MRel}(A, B) \to \mathsf{Rel}(A, B)$ from the universe of all multirelations from A to B to the universe of all ordinary relations from A

to B. Conversely, any ordinary relation can be trivially seen as a multirelation whose all multiplicities are equal to 1. We call this operation $\mathsf{lift}_{AB}\colon \mathsf{MRel}(A,B) \leftarrow \mathsf{Rel}(A,B)$. Index A and B will be often skipped.

It is convenient to specify the multiplicity function of a multirelation $m\colon A \to_{\mathbb{N}} B$ by a matrix, whose rows and columns are indexed by elements of sets A and B resp. For example, in our sales scenario in Sect. 1.1, multirelations contains and belongsTo can be defined by the following matrices.

contains	Bread	Butter	Milk
B1	2	1	0
B2	0	1	1

belongsTo	Bakery	Dairy
Bread	1	0
Butter	0	1
Milk	0	1

Clearly, drop(belongsTo) $\cong$ belongsTo $\cong$ lift(belongsTo), which is trivially true for any ordinary relation.

Two multirelations are called *(sequentially) composable* if the target of the first is the source of the second. For example, contains and belongsTo are composable.

Definition 4 (Sequential Composition)
Let $m\colon A \to_{\mathbb{N}} B$ and $n\colon B \to_{\mathbb{N}} C$ be two composable relations. First, for a given $a \in A$, we define function $m_{a\_}\colon B \leftarrow \mathbb{N}$ by setting $m_{a\_}(b) = m(a,b)$, and similarly for a given $c \in C$, we have function $n_{\_c}\colon B \to \mathbb{N}$ with $n_{\_c}(b) = n(b,c)$. Now *composed* multirelation $m{\times}n\colon A \to_{\mathbb{N}} C$ is defined by the following formula: for any $a \in A$ and $c \in C$, $(m{\times}n)(a,c) = \sum_{b\in B} m_{a\_}(b) \cdot n_{\_c}(b)$.

If multiplicities are specified as matrices, then it is easy to see that the matrix of the composition is given by matrix multiplication.

In the sales example, if the multirelation result denotes the composition contains;belongsTo, then by the definition above:

$$\begin{array}{lcl}
\mathsf{result}(B1, Bakery) & = & 2\cdot 1 + 1\cdot 0 + 0\cdot 0 = 2 \\
\mathsf{result}(B1, Dairy) & = & 2\cdot 0 + 1\cdot 1 + 0\cdot 1 = 1 \\
\mathsf{result}(B2, Bakery) & = & 0\cdot 1 + 1\cdot 0 + 1\cdot 0 = 0 \\
\mathsf{result}(B2, Dairy) & = & 0\cdot 0 + 1\cdot 1 + 1\cdot 1 = 2
\end{array}$$

It is easy to see that if $R1 \subset A{\times}B$ and $R2 \subset B{\times}C$ are ordinary relations, then $\mathsf{lift}R1{\times}\mathsf{lift}R2$ is, in general, a multi- rather than an ordinary relation from A to C. However, dropping multiplicities makes its an ordinary relation known as relational composition. Thus, the latter is $\mathsf{drop}(\mathsf{lift}R1 \times \mathsf{lift}R2)$.

2.2 Index-based Formalization

This section introduces the index-based formalization. We first introduce the concept of a span, explaining through the example from Sect. 1.1. 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.

2.2.1 Multirelations as Spans

We formalize the mapping $\mathsf{contains}$ in Fig. 1.1 by reifying all its constituent links as separate objects. This gives us a set $\mathsf{Contains}$ consisting of five elements c_i as shown in Fig. 2.1. 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. 1.1, it is linked to $B1$ by the *source* link c_{1s}, and to Bread by the *target* link c_{1t}.

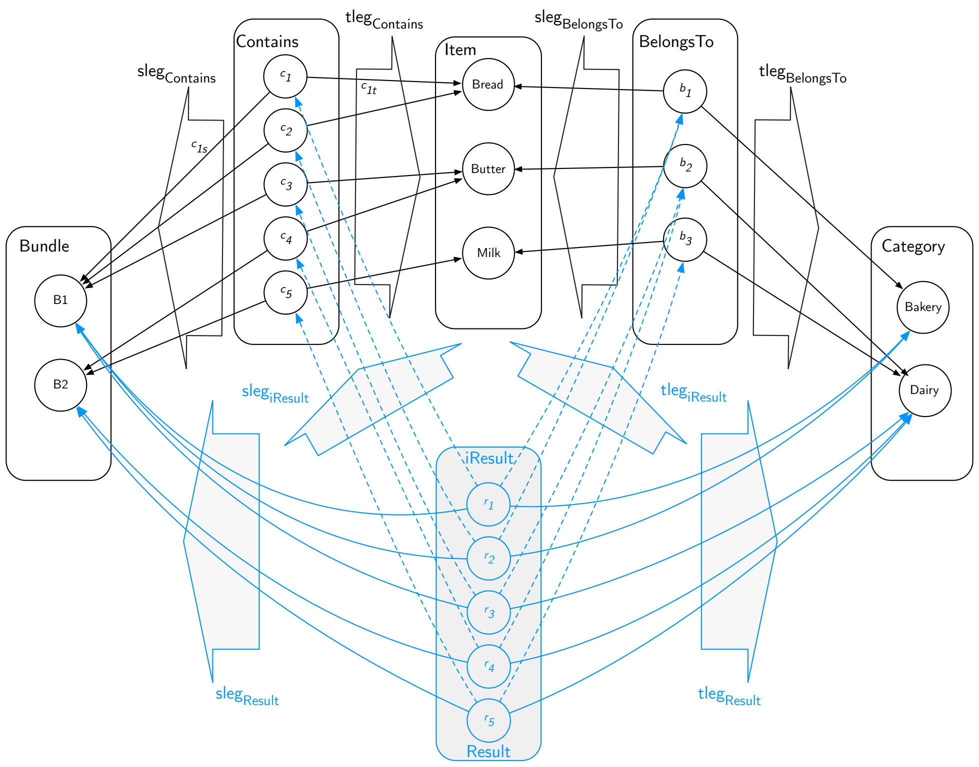
All source links make a function $\mathsf{sleg}_{\mathsf{Contains}}\colon \mathsf{Bundle} \leftarrow \mathsf{Contains}$ called the *source leg*, and all target links make a function $\mathsf{tleg}_{\mathsf{Contains}}\colon \mathsf{Contains} \rightarrow \mathsf{Item}$ called the *target leg*. The triple $(\mathsf{Contains}, \mathsf{sleg}_{\mathsf{Contains}}, \mathsf{tleg}_{\mathsf{Contains}})$ is called a *span* with the *head* set $\mathsf{Contains}$ and *legs* as above. Similarly, relation $\mathsf{belongsTo}$ is formalized by the span with the head set $\mathsf{BelongsTo}$ (see Fig. 2.1).

Definition 5 (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 being a set called the *head*, and $\mathsf{sleg}_R\colon A \leftarrow \mathsf{head}_R$, $\mathsf{tleg}_R\colon \mathsf{head}_R \rightarrow B$, 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 $R\colon A \nrightarrow B$, 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 $f\colon A \rightarrow B$, its source leg is a bijection (e.g., the relation $\mathsf{belongsTo}$ in Fig. 1.1 is such). As the choice of the index set is arbitrary, we can take set A to be the head, and its identity $\mathsf{id}_A\colon A \rightarrow A$ as the source leg. Then the target leg is the function f itself.

Note also that sets A and B in Def. 5 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 $A = \mathsf{Person}$, $B = \mathsf{Person}$, $\mathsf{head}_{\mathsf{Spouse}} = \mathsf{marriageContract}$,

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



Contains
Item
BelongsTo
Bundle
Category
$\mathsf{sleg}_{\mathsf{Contains}}$
$\mathsf{tleg}_{\mathsf{Contains}}$
$\mathsf{sleg}_{\mathsf{BelongsTo}}$
$\mathsf{tleg}_{\mathsf{BelongsTo}}$
c_1
c_2
c_3
c_4
c_5
c_{1t}
c_{1s}
Bread
Butter
Milk
b_1
b_2
b_3
B1
B2
Bakery
Dairy
$\mathsf{sleg}_{\mathsf{iResult}}$
$\mathsf{tleg}_{\mathsf{iResult}}$
iResult
r_1
r_2
r_3
r_4
r_5
Result
$\mathsf{sleg}_{\mathsf{Result}}$
$\mathsf{tleg}_{\mathsf{Result}}$

Figure 2.1: A bundling instance after reification

$\mathsf{sleg}_{\mathsf{Spouse}} = \mathsf{spouse}_1$ 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 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. 1.1 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 $\mathsf{iResult}$ (where i stands for

$$\begin{tikzcd}
A_1 \arrow[r, "f_1"] & Y \\
P \arrow[u, "p_1"] \arrow[r, "p_2"'] \arrow[ur, phantom, "\lrcorner", very near start] & A_2 \arrow[u, "f_2"']
\end{tikzcd}$$

Figure 2.2: Pullback square

'inner', as later we will also build an outer span) is given by the following formula:

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

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 $\mathsf{iResult}$ shown in Fig. 2.1 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. 2.1). 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 6 (**Pullback**)
The *pullback* of two functions with a common target, $f_1\colon A_1 \to Y$ and $f_2\colon Y \leftarrow A_2$ is a span with head $P = \{(a_1, a_2) \in A_1{\times}A_2\colon\ f_1(a_1) = f_2(a_2)\}$. The legs of the span are projections $p_i\colon P \to A_i$ with $p_i(a_1, a_2) = a_i$, $i = 1, 2$. Fig. 2.2 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 7 (**Span Composition**)
Let $R_1\colon A \nrightarrow B$, $R_2\colon B \nrightarrow C$ be two composable spans. Their *(sequential) composition* is a span $R_1;; R_2 = (R, s, t)\colon A \nrightarrow C$ defined as shown in Figure 2.3, 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 $\mathsf{tleg}_{R_2}{=}R_2$.

$$\begin{tikzcd}
A & & \\
R_1 \arrow[u, "s_1"'] \arrow[r, "t_1"] & B & \\
R \arrow[u, "p_1"] \arrow[r, "p_2"'] \arrow[uu, bend left, "s"] \arrow[rr, bend right, "t"'] \arrow[ur, phantom, "\lrcorner", very near start] & R_2 \arrow[u, "s_2"'] \arrow[r, "t_2"'] & C
\end{tikzcd}$$

Figure 2.3: Span composition

$$\begin{tikzcd}
A & H_{R_2} \arrow[l, "s_{R_2}"'] \arrow[d, "t_{R_2}"] \\
H_{R_1} \arrow[u, "s_{R_1}"] \arrow[ur, "b"'] \arrow[r, "t_{R_1}"'] & B
\end{tikzcd}$$

Figure 2.4: Span isomorphism

Then we set $\mathsf{head}_{R_1;R_2}$=head_{R_1}, $\mathsf{sleg}_{R_1;R_2}$=sleg_{R_1}, and $\mathsf{tleg}_{R_1;R_2}$=tleg_{R_1};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. 2.1 in the following sense.

Definition 8 (Span Isomorphism)
Two parallel spans $R_1 \colon A \nrightarrow B$, $R_2 \colon A \nrightarrow B$ are *isomorphic* if there is a bijection between their heads, $b \colon H_{R_1} \to H_{R_2}$, such that $s_{R_1} = b; s_{R_2}$ and $t_{R_1} = b; t_{R_2}$ (as shown in Fig. 2.4). 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

$$\begin{tikzcd}
 & A & \\
I \arrow[ur, "f"] \arrow[rr, "b"] & & I' \arrow[ul, "f'"']
\end{tikzcd}$$

Figure 2.5: Family and family isomorphism

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., [6]).

2.2.2 Multisets as Families and Operations over Them

We use the same indexing idea of reifying element *occurrences* as unique indices to distinguish the multiple occurrences of the same element. This leads us to the concept of a *family*, which we use to represent multisets.

Definition 9 (Families and Family Isomorphism)
Let A be a set (perhaps, infinite). A *(finite) family* over A is a function $f\colon I \to A$ 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 $\{f(i)\colon\ i{\in}I\} \subset A$, 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, $f\colon I \to A$ and $f'\colon I' \to A$, over the same ground set A are called *isomorphic*, and we write $f \cong f'$, if and only if there is a bijection $b\colon I \to I'$ such that $b; f' = f$ (as shown in Fig. 2.5).

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 $i\colon A \to U$ and $j\colon B \to U$. These injections can be seen as two families over the same ground set. Summing them gives us a family $u\colon A \uplus B \to U$, whose index set is the disjoint union of A and B.

$$\begin{tikzcd}
I \arrow[r, "f"] & A \\
P \arrow[u, "p_1"] \arrow[r, "p_2"] \arrow[ur, phantom, "\lrcorner", very near start] \arrow[rr, bend right, "f;;S"'] & S \arrow[u, "s"] \arrow[r, "t"] & B
\end{tikzcd}$$

Figure 2.6: Composing family with span

2.2.3 Mixed Setting: Families/Spans and Ordinary/Multi

Connections between ordinary and multi(sub)sets are realized via the following two functions. Given a set A, we have function $\mathsf{drop}_A\colon \mathsf{MSub}(A) \to \mathsf{Sub}(A)$ from multi- to ordinary subsets of A that ignores indexes and only cares about the range set of a family. Conversely, a (trivial) function $\mathsf{lift}_A\colon \mathsf{MSub}(A) \leftarrow \mathsf{Sub}(A)$ 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 $\mathsf{drop}_{AB}\colon \mathsf{MRel}(A,B) \to \mathsf{Rel}(A,B)$ from the universe of all multirelations from A to B to the universe of all ordinary relations from A to B, and, conversely, $\mathsf{lift}_{AB}\colon \mathsf{MRel}(A,B) \leftarrow \mathsf{Rel}(A,B)$ defined in an obvious way.

It is easy to see that the ordinary union of two ordinary sets X, Y can be presented as $\mathsf{drop}(\mathsf{lift}X \cup \mathsf{lift}Y)$. Similarly, ordinary composition of two ordinary relations $X \subset A\times B$, $Y \subset B\times C$ is $\mathsf{drop}(\mathsf{lift}X;;\mathsf{lift}Y)$.

In ordinary relational modeling, given a relation $R\colon A \nrightarrow B$, 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 10 (Composing a Family with a Span)
Given a family $f\colon I \to A$ and a span $S\colon A \nrightarrow B$, their *composition* is a family $f;;S\colon P \to B$ defined by the diagram in Fig. 2.6 (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 $S\colon A \nrightarrow B$ with a family $g\colon J \to B$ is a family $S;; g\colon P \to A$ defined by a diagram dual to the one in Fig. 2.6: 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*.

2.2.4 Composition as Navigation

Span composition can be also defined in a navigational style. Given a span $R\colon A \nrightarrow B$, we can represent it as a function $\phi\colon A \to \mathsf{MSub}(B)$ by represeting an element $a \in A$ as a "family" $a^*\colon \{*\} \to A$ with $a^*(*) = a$, and defining $\phi(a) = a^*;; R$. This is nothing but a special case of the general image-operation described in Def. 10. The latter can be described as a function $\phi_R^M\colon \mathsf{MSub}(A) \to \mathsf{MSub}(B)$. Now, if we have spans $R1\colon A \nrightarrow B$ and $R2\colon B \nrightarrow C$, we represent them as functions $\phi_{R1}\colon A \to \mathsf{MSub}(B)$ and $\phi_{R2}\colon B \to \mathsf{MSub}(C)$, respectively. Span composition is then represented by the function composition $\phi_{R1}.\phi_{R2}^M$, which is not difficult to show as equal 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 $a.\phi_{R1}.\phi_{R2}^M$. 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 $\emptyset\colon \emptyset \to B$), and $\phi_{R2}^M(\emptyset) = \emptyset$.

The description above shows that span composition effectively prevents the NULL-navigation safety issue occurring in many textual languages such as OCL[20]. 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.

2.3 Indices vs. Multiplicities

We compare the expressive power of the two frameworks, first for the unary constructs (families and multisets), and then for binary (spans and multirelations).

Families vs. Multisets. Every family f determines its *multiplicity function* $m_f\colon A \to \mathbb{N}$ by setting $m_f(a) = \#f^{-1}(a)$ for all $a{\in}A$, where the operator $\#$ denotes ordinary set cardinality. Note that as we do not require f to be surjective, multiplicity may be equal to zero for some elements. It is easy to see that isomorphic families provide the same multiplicity function: $f \cong f'$ implies $m_f = m_{f'}$. Conversely, if two families over A have the same multiplicity function, then they are isomorphic (because any two sets of the same cardinality can be bijectively related). It is easy to see for any multiset, we have $m_{f_m} = \mathrm{m}$, and for any family f, we have $f_{(m_f)} \cong f$ (in the sense of Definition 9). Thus, the notions of multiset and family are equivalent.

Theorem 1
Let f_1, f_2 be two arbitrary families over the same ground set.

(a) $f_1 \cong f_2$ iff $m_{f_1} = m_{f_2}$.

(b) $m_{f_1+f_2} = m_{f_1} + m_{f_2}$. Thus, merging families is the indexed counterpart of the add-union of multisets.

(c) $m_{f_1 \times f_2} = m_{f_1} \times m_{f_2}$. Thus, product of families is the indexed counterpart of the mult-intersection of multisets.

Spans vs. Multirelations. Given a multirelation $m\colon A \to_{\mathbb{N}} B$, we build a span $S_m\colon A \nrightarrow B$ as follows. For all $a{\in}A$, $b{\in}B$, let I_{ab} be any set with cardinality $m(a,b)$, and let $I_m = \biguplus_{a\in A, b\in B} I_{ab}$ be their disjoint union. We define functions $s_m\colon I_m \to A$ and $t_m\colon I_m \to B$ by setting $s_m(i) = a$ and $t_m(i) = b$ if $i{\in}I_{ab}$. Now we define span $S_m = (I_m, s_m, t_m)$. Conversely, given a span $S\colon A \nrightarrow B$, we define a multirelation $m_S\colon A{\times}B \to \mathbb{N}$ by setting for all $a{\in}A$, $b{\in}B$

$$m_S(a,b) = \#\left\{h \in \mathsf{head}_S\colon\ a = \mathsf{sleg}_S(h) \text{ and } \mathsf{tleg}_S(h) = b\right\}.$$

Now it is easy to see that for any multirelation, we have $m_{(S_m)} = m$, and for any span S, we have $S_{(m_S)} \cong S$ (in the sense of Definition 8). Thus, wrt. formal expressiveness, the notions of multirelation and span are equivalent (up to the indexing set isomorphism). Moreover, we can prove a much stronger result that $S_{m\times n} \cong S_m; S_n$, and $m_{S_1;S_2} = m_{S_1} \times m_{S_2}$. Thus, the correspondence between spans and multirelations is compatible with operations over them.

Chapter 3

Multiconcepts Encoding in Alloy

Alloy modeling language and analyzer were created by Daniel Jackson and his team [12]. Instead of focusing on theorem proving like B, VDM and Z, Alloy emphasizes automatic analysis of models. The Alloy language itself has a lot of similarities to object-oriented programming languages; language features such as as single inheritance, subtyping type system and navigational access style are supported in Alloy. The underlying logic system of Alloy is first-order logic with relational algebra and transitive closure, which is easy to understand and powerful enough to express specifications upon Alloy models.

3.1 Overview of Alloy

3.1.1 Sets, Relations and Atoms

Sets and relations, which are simple abstract concepts yet powerful enough to describe any complex system, are the building blocks of models in Alloy. Conceptually, Alloy unifies sets and relations by treating sets as unary relations. Alloy does not distinguish sets and elements (scalars) by introducing the concept *atom*. An atom is essentially a unary relation (set) containing a single tuple, which is not divisible and remains uninterpreted. Breaking down the distinction between sets and elements might be counter-intuitive, but in practice it brings a nice uniformity on the language design such that the syntax can be simplified. The example below demonstrates a set (Item), a relation (belongsTo) and an atom (Bread) in textual notation:

$$\begin{aligned}
\mathsf{Item} &= \{(Bread), (Butter), (Milk)\}\\
\mathsf{belongsTo} &= \{(Bread, Bakery), (Butter, Dairy)\}\\
\mathsf{Bread} &= \{(Bread)\}
\end{aligned}$$

In terms of syntax, keyword `sig` (signature) declares a set of atoms. Relations are declared as fields of signatures. The number of atoms in a introduced set is not fixed, which could even be empty set. Alloy also supports subtyping by the keyword `extends`, representing a subset of atoms inside the superset. All subsets are ensured to be disjoint if they are introduced by the keyword `extends`.

3.1.2 Logic and Constraints

Alloy relies on a first-order relational logic system, which can be automatically analyzed by limiting the scope (number of atoms in each set) of a model. The purpose of constraints in Alloy is to state specifications on a model or to check if a certain property in a model holds true. There are three forms of constraints in Alloy: *fact*, *predicate* and *assertion*. Facts are logic expressions to state that some properties in a model are always true. Predicates are parametrized logic expressions, which can accept and apply on different inputs. Assertions are used to command the Alloy analyzer to check if the property stated in an assertion always holds true in a model; the analyzer will search in a finite solution space (since the scope is limited), trying to find a counter-example.

3.1.3 Type System and Polymorphism

Generally, polymorphism mechanisms provided in programming languages are the key feature to implement generic codes. In Alloy, the type system supports subtypes; by the nature of subtyping, subtype polymorphism (inclusion polymorphism) is supported. For any two types S and T, if there is a subtype relation between them (denoted as $S <: T$), it means that any context expecting a term of type T can safely accept a term of type S. The *top* type, which is the super type of every other type, is defined by the keyword `univ` in Alloy. Using top type `univ` in type signatures as type placeholder is one common way to write generic predicates or functions in Alloy; the side effect is the loss of type safety.

Alloy also supports a certain degree of parametric polymorphism by the feature called *parametric module*. Module is the unit of reusable code in Alloy. In every module, a list of type parameters can be declared at the beginning and instantiated when the module is being imported. This mechanism of parametrization is more of a textual substitution approach like macros in the C programming language. Instantiating a module is just replacing all the type parameters in the module with given concrete types.

3.1.4 Integer

Alloy supports integers with special treatments. In spite of the fact that atoms in Alloy is uninterpreted, there is a library coming along with Alloy distribution which treats integers as a special class of atoms. Each raw integer value of type `int` is wrapped within an atom of type `Int` (like the `Integer` wrapper class of raw type `int` in Java). Integer in Alloy needs to be carefully taken when using because of the limit of bitwidth; overflow often happens and unexpected instance might be given in the analyzer.

3.1.5 Command and Scope

In an Alloy model, each set has a *scope*, which makes an upper bound of the number of atoms in it. Consequently, the total possible instances of the model is finite; thus theoretically, a model can be automatically analyzed by enumerating all the possible instances. Alloy relies on *small-scope hypothesis* [12], arguing that the deficiencies of a model are highly possible to be found given only a small scope. In practice, to analyze or verify an Alloy model, one needs to issue a command along with the scopes of signatures declared in the model. There are two kinds of commands: command `run` will instruct the Alloy Analyzer to find all the valid instances that satisfy all the constraints stated in the model; command `check` followed with an assertion will instruct the analyzer to try to find a counter-example that contradicts the assertion. However, even if no counter-example is found, it does not necessarily prove that the assertion is always true because of the limit in small-scope hypothesis.

3.2 Index-based Multiconcepts Implementation

Sets and relations in Alloy are ordinary and there is no first-class support for multiconcepts. In order to work with multisets and multirelations, encodings are necessary. In Chapter 2 we demonstrate two mathematical approaches to formalize multiconcepts; both are encodable in Alloy since the concepts used in the formalizations are nothing but sets, relations and natural numbers, all of which are supported in Alloy. We start by presenting the index-based encoding, and then we present the multiplicity-based encoding. We have two objectives for the implementation: firstly the implementation should be generic and reusable; secondly the implementation should expose a clean interface so that users can import and use the multiconcept implementation with minimal efforts.

3.2.1 Family and Span

Before we dive into the actual implementation of index-based multisets and multirelations, we start drafting from the perspective of types: the type of a multiset or a multirelation should be a type constructor (a special kind of type which has type parameters and can be instantiated by given concrete types), denoted as follows.

$$\begin{aligned} \text{multiset} &:: \text{MSet}(t) \\ \text{multirelation} &:: \text{MRel}(s, t) \end{aligned}$$

To interpret, the type of a multiset has a type parameter t, which can be instantiated with a concrete type, for example, a primitive type named Item; then we obtain a type MSet(Item) representing the type of a multiset of Item. The instantiation mechanism applies to the multirelation type in the same way such as MRel(Bundle, Item).

The discussion of types of multisets and multirelations is trying to shed light on the implementation of multiconcepts in Alloy. Conceptually multisets and multirelations are generic data types; families and spans are structures which represent multiconcepts; therefore they should also reflect the generality. Polymorphism in programming languages is often used to implement generic data structure; there is no exception in Alloy: parametric polymorphism provided by *parametric module* is naturally a proper technique to model families and spans.

```
1 module mset [t]
2
3 sig Idx {
4   f : one t
5 }
6
7 fun get[] : Idx -> t {
8   f
9 }
```

```
1  module mrel [s, t]
2
3  sig Head {
4    sLeg: one s,
5    tLeg: one t
6  }
7
8  fun get[]: s -> Head -> t {
9    a: s, h: Head, b: t |
10     h.sLeg = a && h.tLeg = b
11 }
```

Both modules declare type parameters at the beginning, which can be instantiated on demand. The structure of families and spans are constructed from the index set. For families, we create a signature named `Idx`, in which there is a relation `f` pointing to another set of type `t` (`t` is a type parameter). The modifier `one` ensures that relation `f` is a total function, since every element in set `Idx` must be linked with exactly one element in another set. A span is implemented in a similar way: a index set named `Head` containing two totally-defined single-valued functions `sLeg` and `tLeg` pointing to source `s` and target `t` as the mathematical definition of a span.

The structure of a family or a span can be flattened as a plain relation in tabular form. A family can be viewed as a totally-defined single-valued binary relation; a span can be viewed as a ternary relation, in which head set sits in the middle column along with source and target sets on the sides. Based on the views above, each module exposes a function `get` which transforms its internal structure to a binary or a ternary relation (line 7 and line 8 resp.).

There are two benefits of the transformation described above: firstly we could write the parameter type and return type of a function or a predicate operating on a family or a span without knowing the internal structure of our encoding (the internal structure is not visible until the parametric module has been instantiated by the time of importing); second, we gain the interoperability between ordinary concepts and multiconcepts automatically. Since a span is transformed to a ternary relation, it can be joined with another total single-valued binary relation using the built-in dot operator resulting in another ternary relation which is just the ternary form of another span.

On the other hand, we can still retrieve the components of a family or a span from the returned binary or ternary relations. Alloy provides utility modules to work with binary and ternary relations in the default distribution. We use them to manipulate binary and ternary relations to implement the helper functions, see module `multi` in Appendix A.

3.2.2 Span Composition

The importance of composition cannot be overstated. For ordinary relations, it is stated as "the quintessential relational operator is composition." [12] in the Alloy book. This importance also applies to the composition operation over multiconcepts. In the index-based formalization, it contains two steps to perform composition:

1. perform the pullback operation;
2. functionally compose the legs to get the final result.

We first implement the pullback operation. In the pullback square diagram on the right, the given sets and relations are X_1, X_2, Y, f_1 and f_2; the calculated parts are P, p_1 and p_2 by the pullback operation. The result of a pullback operation can be understood as the information of the pairs of elements from X_1 and X_2 pointing to the same element in set Y. The Alloy code for pullback operation is shown as follows.

$$\begin{tikzcd}
X_1 \arrow[r, "f_1"] & Y \\
P \arrow[u, "p_1"] \arrow[r, "p_2"'] \arrow[ur, phantom, "\lrcorner" very near start] & X_2 \arrow[u, "f_2"']
\end{tikzcd}$$

```
1  pred pullback[ X1: univ, X2: univ,
2                 f1: X1 -> one univ, f2: X2 -> one univ,
3                  P: univ, p1: P -> X1, p2: P -> X2 ]
4  {
5    (no X1 or no X2) implies {
6      no P
7    } else {
8      all x1: X1 | all x2: X2 | x1.f1 = x2.f2 implies
9       { one p: P | p.p1 = x1 && p.p2 = x2 }
10     #P = #(f1.~f2)
11   }
12 }
```

We first deal with the case when set `X1` or set `X2` is empty (line 5), which implies set `P` is also empty. The constraint in line 8 and 9 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 atoms in set `P` and possibly some unexpected atoms 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 reuse the pullback predicate to implement the composition operation. 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 it is impossible to generate a set that has not been declared ahead 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.

Once we have the pullback predicate, we could use it as a helper sub-predicate to implement the composition predicate. The composition predicate is located in module `mset` and `mrel` to state that the current family or span is the result of composition from other families and spans.

```
// in module mset [g]
open multi

// current family is composed from fami & span
pred composedFrom[fami: univ -> one univ, span: univ -> univ -> g ] {
  let I = fami.idx, Hd = span.head,
      sLeg = span.sleg, tLeg = span.tleg
    | some p1: Idx->I, p2: Idx-> Hd
    | pullback[I, Hd, fami, sLeg, Idx, p1, p2] // Step 1, pullback
    && f = p2.tLeg // Step 2, functional compostion of legs
}
```

```
// in module mrel [s, t]
open multi

// current span is composed from span1 & span2
pred composedFrom[span1: s -> univ -> univ, span2: univ -> univ -> t ] {
  let Hd1=span1.head,   Hd2=span2.head,
      sLeg1=span1.sleg, tLeg1=span1.tleg,
      sLeg2=span2.sleg, tLeg2=span2.tleg
    | some p1: Head -> Hd1, p2: Head -> Hd2
    | pullback[Hd1,Hd2, tLeg1,sLeg2, Head,p1,p2] // Step 1, pullback
    && sLeg = p1.sLeg1 && tLeg = p2.tLeg2 // Step 2, functional composition of legs
}
```

3.2.3 Multiplicity of Elements

In the index-based encoding, we can specify the multiplicity of a element in a multiset by indirectly putting constraint on the multiplicity of indices pointing to the element; the same approach applies to span-structured multirelations. We expose a predicate named `moe` in module `mset` and `mrel` respectively as follows:

```
module mset [g]

pred moe[e: g, n: Int] {
  #(f.e) = n
}
```

```
module mrel [s, t]

pred moe[source: s, target: t, n: Int] {
  #(source.(get[]).target) = n
}
```

3.2.4 Union, Intersection and Merge

Let A and B be two multisets. If an element e occurs x times in A and y times in B, it occurs $x + y$ times in the result of merge $A + B$. In our encoding, it is simply a union of two families (assuming their index sets are disjoint). For the union of two multisets, if element e occurs x times in A and y times in B, it occurs $max(x, y)$ times in the result of union $A \cup B$. Similarly, e occurs $min(x, y)$ times in the result of intersection $A \cap B$.

Set comprehension is used to implement union and intersection operations as functions. Merge (or disjoint union) can be simply encoded as the union of two families.

```
// max-union, element with larger multiplicity wins
fun union[f1, f2: univ->one univ] : univ->one univ {
  let I = idx[f1]+idx[f2], G = grd[f1]+grd[f2] |
    { i: I, g: G | (i in idx[f1] && g = i.f1 && #f1:>g >= #f2:>g) or
                   (i in idx[f2] && g = i.f2 && #f1:>g <  #f2:>g ) }
}
// min-intersection, element with smaller multiplicity wins
fun intersection[f1, f2: univ->one univ] : univ->one univ {
  let I = idx[f1]+idx[f2], G = grd[f1]+grd[f2] |
    { i: I, g: G | (i in idx[f1] && g = i.f1 && #f1:>g <= #f2:>g) or
                   (i in idx[f2] && g = i.f2 && #f1:>g >  #f2:>g ) }
}
// disjoint union, simply a union of two families
fun merge[f1, f2: univ->one univ] : univ->one univ {
  f1 + f2
}
```

3.2.5 Domain and Range Restriction, Inverse

Since we can 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 transformed ternary relation.

```
// flip the 1st and 3rd column
fun inverse[span: univ->univ->univ] : span {
  ter/flip13[span]
}
```

3.2.6 Lift and Drop

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:

```
// in module mset [g]

pred liftedFrom[ G: g ] {
  Idx.f = G && #Idx = #G
}
```

```
// in module mrel [s, t]

pred liftedFrom[r: s -> t] {
  (~sleg).tleg = r && #Head = #r
}
```

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

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

fun drop[span: univ -> univ -> univ] : univ -> univ {
  ter/select13[span]
}
```

3.3 Multiplicity-based Multiconcepts Implementation

3.3.1 Multiplicity-based Representation

In the multiplicity-based multiconcepts library, we mainly leverage the top type and covariance in Alloy's subtyping system to implement generic representations and operations for multisets and multirelations. The universal type for sets and binary relations in Alloy are `univ` and `univ->univ`. What we need to do is augmenting the type with an extra column to carry the multiplicity information, which results in types `Int->univ` and `univ->Int->univ` (the position of type `Int` can be arbitrary but it is convenient to implement some operations when type `Int` is in the middle). In other words, any expression with subtype of `Int->univ` or `univ->Int->univ` can be considered as a representation of a multiset or a multirelation (with one extra constraint applied). For example, a valid encoding of a multirelation is shown below,

```
abstract sig Bundle {
  contain : Int -> Item
}
abstract sig Item {}
```

the expression `contain` refers to a ternary relation with type `Bundle->Int->Item` which is a subtype of `univ->Int->univ`. Another way to construct a valid ternary relation could be using a global singleton such as follows,

```
abstract sig Bundle {}
abstract sig Item {}
one sig Sing {
  contains : Bundle->Int->Item
}
```

In this setting, we declare a singleton `Sing` with a field named `contains`. The expression `Sing.contains` then yields a relation with type `Bundle->Int->Item` representing a multirelation between Bundle and Item. As we can see, our implementation is not limited to one specific form to encode multiplicity-based multiconcepts. Additionally, the following constraint needs to be applied to ensure that the representation is essentially a positive numeric-valued function.

```
pred mRel[r: univ->Int->univ] {
  let dom = r.dom | let ran = r.ran |
    all a:dom, b:ran | no a.r.b || (one a.r.b && a.r.b > 0)
4 }
```

The constraint requires that given a ternary relation r of type `univ->Int->univ`, for any pair in the Cartesian product of the domain and range set, either there is no integer value associated or there is only one integer which must be greater than zero. Only with this constraint applied, ternary relation r can be a legitimate representation of a multirelation.

3.3.2 Matrix Multiplication as Composition

Composition for numeric encoding is matrix multiplication as we defined in Sect. 2.1.2. From the perspective of type, we compose two relations of types `T->Int->U` and `U->Int->V` resulting in a relation of type `T->Int->V`, where `T`, `U` and `V` are three type parameters. The encoding of matrix multiplication is shown below with explanation afterward.

```
 1 module matrix[T, U, V]
 2
 3 open util/integer
 4 open util/ternary
 5
 6 fun id3[x: univ] : x -> x -> x {
 7   { a, b, c: x | a = b && b = c }
 8 }
 9
10 fun mjoin[m1: T->Int->U, m2: U->Int->V] : T->Int->V {
11   { t:T, n:Int, v:V |
12       let r = (t.m1).(id3[U]).(m2.v) | // r : Int -> U -> Int
13         (some r) && n = (sum u:U | mul[(select12[r]).u, u.(select23[r])]) }
14 }
```

For demonstration purpose, we introduce three type parameters `T`, `U` and `V` (line 1) at the beginning; we open two utility module `integer` and `ternary` for integer multiplication and ternary relation operations (line 3 and 4). Before the composition implementation, we implement a helper function named `id3` to generate a ternary identity relation.

We use set comprehension to encode the matrix multiplication. To multiply two matrices `m1` and `m2` of type `T->Int->U` and `U->Int->V`, the result is a set of triple `(t:T, n:Int, v:V)` (line 11), where elements `t, n, v` satisfies a certain condition (described in line 11, 12). To state the condition, we bind a intermediate ternary relation of type `Int->U->Int` to an identifier `r`, which is obtained by joining the last two columns of `m1`, a ternary identity relation of all atoms in `U` and the first two column of `m2`; then we require that r is non-empty and the integer value n for the corresponding elements t and v is the summation of the multiplication of the integer in first column and last column in relation r for each u.

3.3.3 Union, Intersection and Merge

The union, intersection and merge operations for multisets are implemented by arithmetic calculation on the multiplicity of elements.

```
fun union[f1, f2: Int -> univ] : Int -> univ {
  let base = (ran[f1]+ran[f2]) |
    { n : Int, e : base | (some f1.e || some f2.e) && n = larger[f1.e, f2.e] }
}

fun intersect[f1, f2: Int -> univ] : Int -> univ {
  let base = (ran[f1]+ran[f2]) |
    { n : Int, e : base | (some f1.e && some f2.e) && n = smaller[f1.e, f2.e] }
}

fun merge[f1, f2: Int -> univ] : Int -> univ {
  let base = (ran[f1]+ran[f2]) |
    { n : Int, e : base | n = add[f1.e, f2.e] }
}
```

3.3.4 Domain and Range Restriction, Inverse

The domain and range restriction is similar to index-based encoding. Built-in operators <: and :> can be directly used since the encoding is merely a ternary relation. The function flip13 in the library ternary can be used as inversion.

3.3.5 Lift and Drop

Operation *lift* can be encoded as a function, which returns a ternary relation by associating multiplicity one to each pair in the original ordinary relation. Operation *drop* is similar to the implementation in index-based encoding.

```
fun lift[r: univ->univ] : univ->Int->univ {
  let dom = r.univ, ran = univ.r |
    { a:dom, n:Int, b:ran | a->b in r && n = 1}
}

fun drop[r: univ -> Int -> univ] : univ->univ {
  select13[r]
}
```

Chapter 4

Evaluation

4.1 Demonstration

In this section, we demonstrate the usage of both multiconcept implementations to build the bundling model introduced in Sect. 1.1 in Alloy. The full Alloy models are shown in Listing 4.1 and Listing 4.2.

Both multiconcept implementations are capable to model the grocery sale scenario and express the bundling rules; Alloy analyzer finds correct instances for both models. However, in terms of usability, the multiplicity-based implementation has one noticeable advantage: all the algebraic operations over multiconcepts are encoded as functions, thus a sequence of operations can be written as a series of function invocations. In the contrary, due to previously discussed limitation, composition in the index-based implementation has to be encoded as a predicate, which means every intermediate result in a series of compositions has to be declared explicitly with the composition predicate applied. Obviously, the composition operation in index-based implementation is less convenient to use than the one in multiplicity-based implementation.

Another advantage of multiplicity-based implementation is its visualization. Alloy Analyzer can visualize instances of a model using graphical representations. When displaying a ternary relation of type `univ->Int->univ`, the analyzer will automatically put the integer on the arc. For example, the visualization of an instance of the bundling model is shown in Fig. 4.1 without any manual configuration. It is clear to see the multiplicity of each relation. In this particular instance, bundle B1 has 2 bread, 2 milk and bundle B2 has 1 bread, 1 milk, 1 butter; both bundles satisfy the rules we stated in the model. To

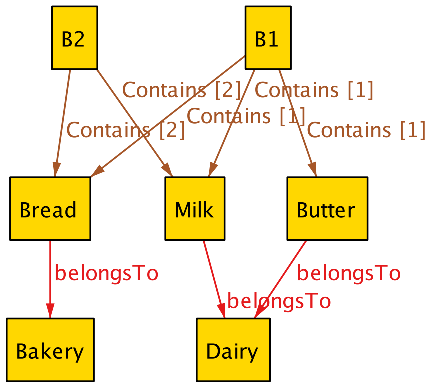


Figure 4.1: Visualization of multiplicity-based model

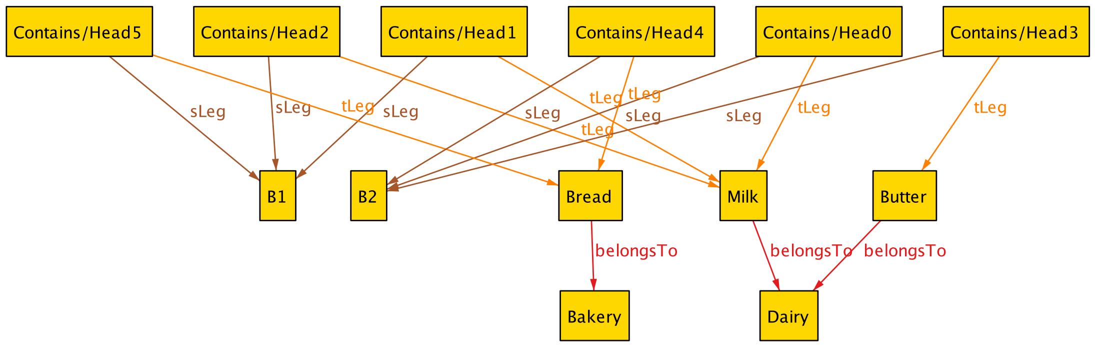


Figure 4.2: Visualization of index-based model

compare, Fig. 4.2 shows the visualization of an instance of the bundling model with the index-based implementation. Without any configuration, it is not as comprehensible as the multiplicity-based visualization since every index is displayed separately, which makes calculating the multiplicity of each relation an extra step. Even though we could improve the visualization (see Fig. 4.3) by hiding the head set of the span and displaying the transformed ternary relation from the span, it still needs to calculate the number of links. According to the two advantages discussed above, the multiplicity-based implementation is more preferable in terms of usability.

```
// import multiplicity-based multiconcepts module
open matrix[Bundle, Item, Category]

// Setup sets Bundle, Item and Category
abstract sig Bundle {
  Contains: Int -> Item // Declare ternary relation Contain
}
one sig B1, B2 extends Bundle {}
abstract sig Item {
  belongsTo: set Category // Declare an ordinary relation belongsTo
}
one sig Bread, Butter, Milk extends Item {}
fact {
  Bread.belongsTo = Bakery
  Butter.belongsTo = Dairy
  Milk.belongsTo = Dairy
}

abstract sig Category {}
one sig Dairy, Bakery extends Category {}

fact {
  // Restrict ternary relation to be a multirelation
  mRel[Contains]
  // Lift ordinary relation belongsTo to a multirelation
  let BelongsTo = lift[belongsTo] |
  // Compute the composition
  let Result = matmul[Contains, BelongsTo] |
  // State the bundling rules
  all b : Bundle | b.Result.Dairy >= 2 && #(b.(drop[Result])) >= 2
}

run {} for 4 but 4 Int
```

Listing 4.1: Bundling model based on multiplicity-based implementation

```
// Utility module of index-based multiconcepts implementation
open multi
// declare a multirelation from Bundle to Item
open mrel[Bundle, Item] as Contains
// declare a multirelation from Item to Category
open mrel[Item, Category] as BelongsTo
// declare a multirelation to hold the composition result
open mrel[Bundle, Category] as Result

// Setup sets Bundle, Item and Category
abstract sig Bundle {}
one sig B1, B2 extends Bundle {}
abstract sig Item {
  belongsTo: set Category // Declare an ordinary relation belongsTo
}
one sig Bread, Butter, Milk extends Item {}
fact {
  Bread.belongsTo = Bakery
  Butter.belongsTo = Dairy
  Milk.belongsTo = Dairy
}
abstract sig Category {}
one sig Dairy, Bakery extends Category {}

fact {
  // State that mrel BelongsTo is lifted from belongsTo
  BelongsTo/liftedFrom[belongsTo]
  // State that mrel Result is composed from Contains and BelongsTo
  Result/composedFrom[Contains/get, BelongsTo/get]
  // State the rules
  all b : Bundle | #(b<:Result/get:>Dairy) >= 2
  all b : Bundle | #(b.(drop[Result/get])) >= 2
}

// for better visualization
fun disp1[] : univ->univ->univ {
  Contains/get
}

run {} for 6 Contains/Head, 3 BelongsTo/Head, 7 Result/Head
```

Listing 4.2: Bundling model based on index-based implementation

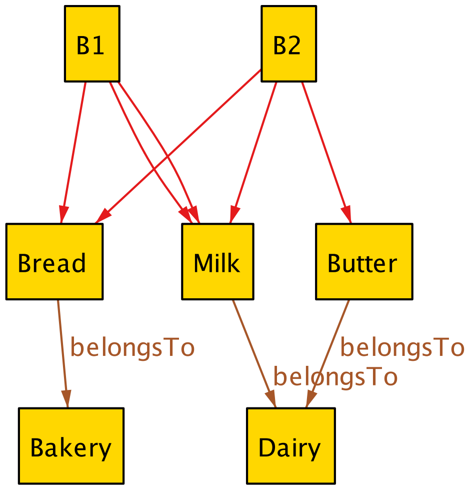


Figure 4.3: Visualization of index-based model with extra configuration

4.2 Performance

In this section, we make a fair comparison of the performance between index-based and multiplicity-based multiconcept implementations in Alloy. The benchmark method we adopt is using each multiconcept library to encode the same model; then we record and compare the running time for the Alloy analyzer to execute each model. We focus on measuring the performance of the composition operation since it has the most complex implementation in both libraries and is the most important operation in multirelational algebra.

The model we choose is an abstract matrix multiplication computation of two all-one matrices. First we measure the cost of encoding two all-one matrices using each multiconcept library; next we compose two all-ones matrices to compare the performance of the composition implementation in each library. We also scale up the dimension of the matrices to test how the performance of each library scales with the size of the model.

For instance, the multiplication of two 3×3 all-ones matrices is shown below in mathematical notations. We encode this multiplication process using our multiconcept implementation. The common setup is shown in Listing 4.3.

$$\begin{array}{c|ccc} & b_1 & b_2 & b_3 \\ a_1 & 1 & 1 & 1 \\ a_2 & 1 & 1 & 1 \\ a_3 & 1 & 1 & 1 \end{array} \times \begin{array}{c|ccc} & c_1 & c_2 & c_3 \\ b_1 & 1 & 1 & 1 \\ b_2 & 1 & 1 & 1 \\ b_3 & 1 & 1 & 1 \end{array} = \begin{array}{c|ccc} & c_1 & c_2 & c_3 \\ a_1 & 3 & 3 & 3 \\ a_2 & 3 & 3 & 3 \\ a_3 & 3 & 3 & 3 \end{array}$$

Figure 4.4: All-ones matrices multiplication when dimension is 3×3

```
abstract sig A {}
one sig a1, a2, a3 extends A {}

abstract sig B {}
one sig b1, b2, b3 extends B {}

abstract sig C {}
one sig c1, c2, c3 extends C {}
```

Listing 4.3: Common setup for 3×3 matrix

Firstly, we created three disjoint sets with names `A`, `B` and `C`; each set contains three elements (atoms). We model two all-one matrices using our index-base and multiplicity-based multirelation implementation, shown below.

```
open matrix[A, B, C]

one sig Sing {
  mrel1 : A -> Int -> B,
  mrel2 : B -> Int -> C,
  mrel3 : A -> Int -> C // for composition
}

fact {
  Sing.mrel1 = a1->1->b1 + a1->1->b2 + a1->1->b3
             + a2->1->b1 + a2->1->b2 + a2->1->b3
             + a3->1->b1 + a3->1->b2 + a3->1->b3

  Sing.mrel2 = b1->1->c1 + b1->1->c2 + b1->1->c3
             + b2->1->c1 + b2->1->c2 + b2->1->c3
             + b3->1->c1 + b3->1->c2 + b3->1->c3

  Sing.mrel3 = mjoin[Sing.mrel1, Sing.mrel2] // for composition
}
```

Listing 4.4: Multiplicity-based encoding of two 3×3 all-ones matrices

```
open mrel[A, B] as mrel1
open mrel[B, C] as mrel2
open mrel[A, C] as mrel3

fact {
  mrel1/moe[a1, b1, 1] && mrel1/moe[a1, b2, 1] && mrel1/moe[a1, b3, 1]
  mrel1/moe[a2, b1, 1] && mrel1/moe[a2, b2, 1] && mrel1/moe[a2, b3, 1]
  mrel1/moe[a3, b1, 1] && mrel1/moe[a3, b2, 1] && mrel1/moe[a3, b3, 1]

  mrel2/moe[b1, c1, 1] && mrel2/moe[b1, c2, 1] && mrel2/moe[b1, c3, 1]
  mrel2/moe[b2, c1, 1] && mrel2/moe[b2, c2, 1] && mrel2/moe[b2, c3, 1]
  mrel2/moe[b3, c1, 1] && mrel2/moe[b3, c2, 1] && mrel2/moe[b3, c3, 1]

  mrel3/composedFrom[mrel1/get, mrel2/get] // for compostion
}
```

Listing 4.5: Index-based encoding of two 3×3 all-ones matrices

The lines with comments is to set up the composition. We first evaluate the cost of encoding two all-ones matrices. The model without the commented lines will be executed. The scope is set to be the minimum number that Alloy analyzer can find an instance. The Alloy analyzer is configured to use the MiniSAT with unsat core solver and runs on a 2.4Ghz Core i7 machine with 8GB RAM. The result is presented in Table 4.1.

Encoding	Matrix Size	Var	Primary Var	Clauses	Generation time	Analysis time
Numeric	2	135	128	134	5	9
	3	151	144	150	10	17
	4	519	512	518	103	89
Index	2	436	40	877	28	12
	3	2188	126	5441	64	30
	4	6340	288	17157	212	77

Table 4.1: Result of modeling two all-ones matrices, time in ms

From the benchmark result we could see that the multiplicity-based encoding performs slightly better than index-based encoding. An interesting observation is that there are less difference between var and primary var in multiplicity-based encoding than that in the index-based encoding, where *primary var* corresponds to instances declared in a model and *var* reflects the variables that are needed to encode the constraints into SAT formula.

Next we test the performance of the composition operation in each encoding. We benchmark the composition of two multirelations represented by all-ones matrices, scaling up the dimension from 2×2. The result is shown in Table 4.2.

Encoding	Matrix Size	Var	Primary Var	Clauses	Generation time	Analysis time
Numeric	2	2314	96	8137	27	10
	3	7453	216	26586	62	14
	4	47898	768	194265	732	76
Index	2	2354	144	5487	61	46
	3	21501	801	55185	582	305620
	4	115282	2912	321756	5721	Suspended

Table 4.2: Result of composition over two all-ones matrices, time in ms

We could clearly see the performance of composition operation encoding in both encodings. The composition operation in multiplicity-based encoding performs very well and scales reasonably with the size of the model. In contrast, the performance of composition operation in index-based encoding is acceptable when the size of the model is small (2×2); however, the performance degenerates dramatically with the increase of the model size; the model cannot be analyzed in reasonable time (beyond 20 minutes) when the matrix size is 4×4, in which case we have to manually suspend the analysis.

From the benchmark result we could see that the multiplicity-based implementation outperform the index-based implementation enormously when composition operation is used in the model. Combined with the usability discussion in previous section, we think the multiplicity-based library is more practical and preferable to use when modeling with multiconcepts in Alloy.

4.3 Application

Multiconcepts are useful in many areas, one example is in software architecture analysis. It has been studied that it is beneficial to attribute the links between different components in a large software system with a number [10], which essentially are multirelations. With only ordinary relations, a typical software architecture diagram turns out almost fully meshed; The distinction between important connections and minor or trivial connections can barely be observed. In contrast, the quantitative information in a multirelation explicitly shows the importances of different connections, in which way the recognition and understanding of a system architecture diagram could be potentially improved.

More interestingly, a theory named 'lifting' can transforms a low-level component inter-dependency relation in a system to a high-level one. The theory of lifting is proposed under the algebra of ordinary relations at first [11]; further study shows that it can be generalized to the context of multirelations, turning out to be a transformation aggregating meaningful quantitative information.

We demonstrate the application of multirelation in Alloy using an imaginary system architecture scenario. The Alloy model is shown in Listing 4.6. The system consists of `Subsystems` and each `Subsystem` consists of several `Components`. The ordinary relation `partOf` defines the ownership of each component; the multirelation `use` is to describe the inter-dependency relation among `Components`. We will use the lifting theory to transform the low level `use` relation to a high level relation `Use` among `Subsystems`.

The lifting theory in the multirelation context is defined as follows: given a 'use' relation r with type $\mathrm{MRel}(\mathrm{T},\mathrm{T})$ and a 'part-of' relation p with type $\mathrm{Rel}(\mathrm{T},\mathrm{U})$, the expression $\mathsf{lift}(p^{-1});; r;; \mathsf{lift}(p)$ (recall that lift is the operation which transforms an ordinary relation to a multirelation) yields a multirelation with type $\mathrm{MRel}(\mathrm{U},\mathrm{U})$, representing the 'use' relation in a higher level with quantitative information aggregated. Line 28 in listing 4.6 is the lifting transformation expression in Alloy with our multiplicity-based multiconcepts implementation. Two instances of the architecture model are shown in Fig. 4.6 and Fig. 4.5; one can clearly see the low-level and high-level view of the system architecture.

```
1  open matrix[univ, univ, univ]
2
3  abstract sig Component {
4    partOf : Subsystem,
5    use : Int -> Component
6  }
7
8  abstract sig Subsystem {
9    Use : Int -> Subsystem
10 }
11
12 one sig Subsystem1, Subsystem2, Subsystem3 extends Subsystem {}
13 abstract sig ComGroup1, ComGroup2, ComGroup3 extends Component {}
14
15 one sig Component1, Component2, Component3 extends ComGroup1 {}
16 one sig Component4, Component5, Component6 extends ComGroup2 {}
17 one sig Component7, Component8, Component9 extends ComGroup3 {}
18
```

```
fact {
  ComGroup1.partOf = Subsystem1
  ComGroup2.partOf = Subsystem2
  ComGroup3.partOf = Subsystem3
}

fact {
  mRel[use]
  mRel[Use]
  Use = lift[~partOf].mjoin[use].mjoin[lift[partOf]]
}
```

Listing 4.6: A system architechure model with two layers

Obviously, with numbers on the arcs, we have more visibility on how heavily a part of the system is used by the others. What is more, the topography of the system graph is still preserved since we can drop a multirelation to an ordinary relation; one can even put graph property constraints on the model such as requiring that the high-level subsystem graph is acyclic, in which case, the instance in Fig. 4.5 is an invalid instance. Such analysis of system architectures are presented and proved to be practical and useful in existing literatures, see [10] and [11].

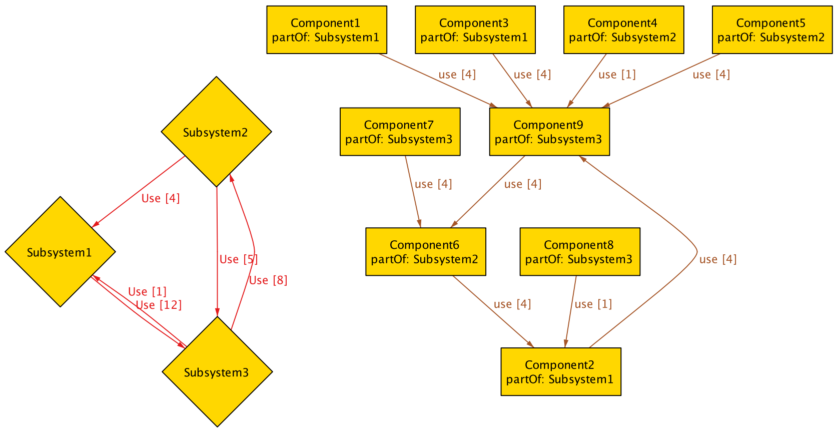


Figure 4.5: An instance of system architecture model with lifting

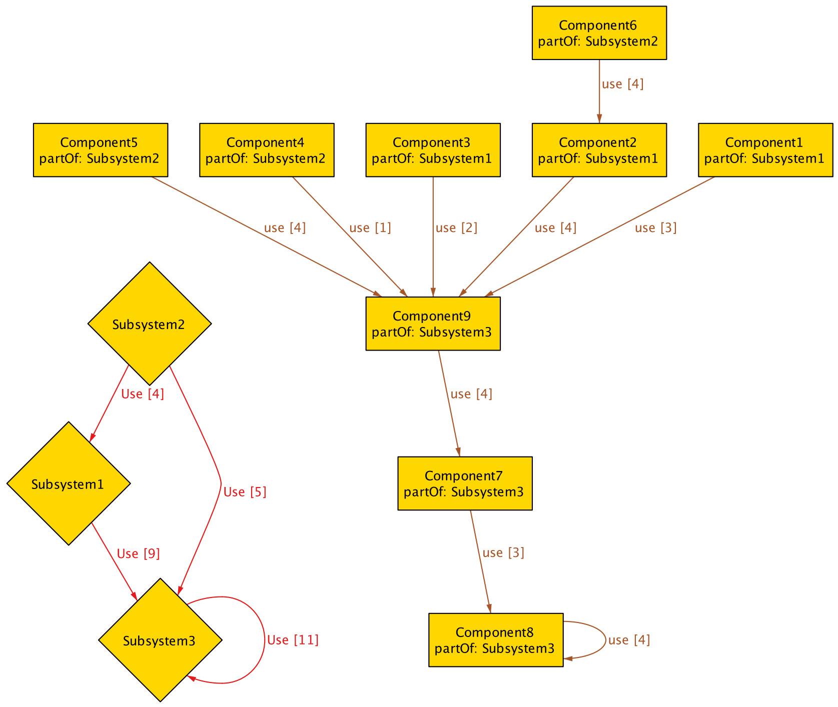


Figure 4.6: Another instance of system architecture model with lifting

Chapter 5

Conclusion and Future Work

In this thesis, a systematic and principled framework for modeling with multisets and multirelations is presented, including theoretical foundations, practical implementations and applications. In terms of theory background, we revisited the common multiplicity-based approach and presented a less-known index-based formalization based on the concept of spans in category theory. Both formalizations have covered the representation of multiconcepts and a set of algebraic operations. We have also shown that both formalization have the same expressiveness.

We have implemented two multiconcepts libraries in Alloy according to the mathematical formalizations. Both libraries support multiconcept declarations and algebraic operations. Implementations are presented in detail, which could be inspiring for other modeling languages to implement a similar multiconcepts library. Both libraries expose a carefully-designed interface, which is easy to use and allows Alloy users to model with multiconcepts by simply importing our library. We release our multiconcepts libraries as a contribution to the Alloy community. We have also evaluated both libraries concerning the usability and performance. Multiplicity-based implementation has a nicer usage pattern and scales better with the increase of the model size; therefore, is more preferable and applicable in practical modeling. Finally, as a by-product, we have demonstrated how to implement a few important concepts and operations from category theory in Alloy: family, span, and their composition via pullback.

In the future, there are several potential extensions of our work on multiconcepts. Transitive closure is defined for ordinary relations but remains unclear for multirelations; a further investigation on a theoretical explanation for transitive closure in the context of multirelations might be valuable. Satisfiability Modulo Theories (SMT) solvers provide similar constructs to Alloy; sets, total functions, integers and first-order logic are available in SMT. One could attempt to implement multiconcepts in SMT using our Alloy implementation as blueprints. The multiconcept implementation in Alloy can also be used as semantic primitives to implement a modeling language which has built-in multiconcept support. Clafer [4], which uses Alloy as one of the back-end solver, can declare span-shaped multirelations. One could extend the syntax and semantic of Clafer to support operations over multiconcepts natively.

References

[1] Are there multisets in Alloy?, 2014. http://stackoverflow.com/questions/23579928/are-there-multisets-in-alloy, last accessed in Aug 2016.

[2] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software & Systems Modeling*, 9(1):69–86, 2010.

[3] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Partial instances via subclassing. In *International Conference on Software Language Engineering*, pages 344–364. Springer, 2013.

[4] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Clafer: unifying class and feature modeling. *Software & Systems Modeling*, 2014.

[5] Paolo Baldan. *Modelling concurrent computations: from contextual Petri nets to graph grammars.* PhD thesis, PhD thesis, Department of Computer Science, University of Pisa, 2000. Available as technical report n. TD-1/00, 2000.

[6] Michael Barr and Charles Wells. *Category theory for computing science*, volume 49. Prentice Hall New York, 1990.

[7] Wayne D Blizard et al. The development of multiset theory. *Modern logic*, 1(4):319–352, 1991.

[8] Roberto Bruni and Fabio Gadducci. Some algebraic laws for spans. *Electr. Notes Theor. Comput. Sci.*, 44(3):175–193, 2001.

[9] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via smt solving. In *International Symposium on Formal Methods*, pages 133–148. Springer, 2011.

[10] L Feijs and RL Krikhaar. Relation algebra with multi-relations. *International Journal of Computer Mathematics*, 70(1):57–74, 1998.

[11] Loe Feijs, René Krikhaar, and R Van Ommering. A relational approach to support software architecture analysis. *Software: Practice and Experience*, 28(4):371–400, 1998.

[12] Daniel Jackson. *Software Abstractions: logic, language, and analysis.* MIT press, 2012.

[13] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer Science & Business Media, 2013.

[14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class diagrams analysis using alloy revisited. In *MoDELS*, pages 592–607. Springer, 2011.

[15] GP Monro. The concept of multiset. *Mathematical Logic Quarterly*, 33(2):171–178, 1987.

[16] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[17] Richard Rado. The cardinal module and some theorems on families of sets. *Annali di Matematica pura ed applicata*, 102(1):135–154, 1975.

[18] Jonathan A Robles and Geoffrey A Solano. Modeling petri nets using alloy. In *TENCON 2012-2012 IEEE Region 10 Conference*, pages 1–6. IEEE, 2012.

[19] Peiyuan Sun, Zinovy Diskin, Michał Antkiewicz, and Krzysztof Czarnecki. Modeling and reasoning with multirelations, and their encoding in Alloy. In *OCL 2016–16th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations Workshop Proceedings*, 2016.

[20] Edward D Willink. Safe navigation in ocl. In *OCL 2015–15th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations Workshop Proceedings*, page 81, 2015.

APPENDICES

Appendix A

Alloy Index-based Multiconcepts Library

```
/** Parametric module to declare a multiset of type g */
module mset [g]
open multi as m

abstract sig Idx {
  f : one g
}

// plain binary relation representation
fun get[] : Idx -> one g { f }

// multiplicity Of element
pred moe[e: g, n: Int] {
  #(f:>e) = n
}

// is isomorphic to another family
pred isomorphicTo[ family: univ -> one univ ] {
  let I = family.idx |
  some r: I->Idx | bijection[r, I, Idx] && r.f = family
}

// current lift from an ordinary set
pred liftedFrom[ s: univ ] {
  Idx.f = s && #Idx = #s
}
```

```

// current family is the composition result from another family and span
pred composedFrom[fami: univ -> one univ,
                  span: univ -> univ -> g ] {
  let I = fami.idx, sLeg = span.sleg, tLeg = span.tleg, Hd = span.head
  | some p1: Idx->I, p2: Idx-> Hd
  | pullback[I, Hd, fami, sLeg, Idx, p1, p2]
 && f = p2.tLeg
}
```

```
/** Parametric module to declare a multirelation from type s to type t */
module mrel [s, t]
open multi as m

sig Head { sLeg : one s, tLeg : one t }

// fact { some Head }

// ternary relation representation
fun get[] : s -> Head -> t {
  { a: s, h: Head, b: t |  h.sLeg = a and h.tLeg = b}
}

// multiplicity Of element
pred moe[source: s, target: t, n: Int] {
  #(get[].mdom[source].mran[target]) = n
}

// current span is lift from an ordinary binary relation
pred liftedFrom[r: s -> t] {
  (~sLeg).tLeg = r && #Head = #r
}

// current span is the composition result from another two spans.
pred composedFrom[ span1 : s -> univ -> univ,
                  span2 : univ -> univ -> t ] {
  let sLeg1=span1.sleg, tLeg1=span1.tleg, Hd1 = span1.head,
      sLeg2=span2.sleg, tLeg2=span2.tleg, Hd2 = span2.head
  | some p1: Head -> Hd1, p2: Head -> Hd2
  | pullback[Hd1, Hd2, tLeg1, sLeg2, Head, p1, p2]
 && sLeg = p1.sLeg1 && tLeg = p2.tLeg2
}
```

```
/** utility functions for family and span */
module multi

open util/relation as rel
open util/ternary  as ter

// return the index set of family f
fun idx[f: univ -> one univ] : univ {
  rel/dom[f]
}

// return the ground set of family f
fun grd[f: univ -> one univ] : univ {
  rel/ran[f]
}

// return the head of the span
fun head[span: univ -> univ -> univ] : univ {
  ter/mid[span]
}

// return the source leg of the span
fun sleg[span: univ -> univ -> univ] : univ -> univ {
  ~(ter/select12[span])
}

// return the target leg of the span
fun tleg[span: univ -> univ -> univ] : univ -> univ {
  ter/select23[span]
}

// inverse, by flipping the 1st and 3rd column
fun inverse[span: univ->univ->univ] : span {
  ter/flip13[span]
}

// domain restriction, return the ternary relation starting with e
fun mdom[span: univ->univ->univ, e: univ] : span {
  e <: span
}

// range restriction, return the ternary relation ending with e
fun mran[span: univ->univ->univ, e: univ] : span {
  span :> e
}
```

```

// max-union, element with larger multiplicity are selected in the result
fun union[f1, f2: univ->one univ] : univ->one univ {
  let I = idx[f1]+idx[f2], G = grd[f1]+grd[f2] |
    { i: I, g: G | (i in idx[f1] && g = i.f1 && #f1:>g >= #f2:>g) or
                   (i in idx[f2] && g = i.f2 && #f1:>g <  #f2:>g ) }
}

// min-intersection, element with smaller multiplicity are selected in the result
fun intersection[f1, f2: univ->one univ] : univ->one univ {
  let I = idx[f1]+idx[f2], G = grd[f1]+grd[f2] |
    { i: I, g: G | (i in idx[f1] && g = i.f1 && #f1:>g <= #f2:>g) or
                   (i in idx[f2] && g = i.f2 && #f1:>g >  #f2:>g ) }
}

// disjoint union, simply a union of two families
fun merge[f1, f2: univ->one univ] : univ->one univ {
  f1 + f2
}

// drop indices and return ordinary set or relation
fun drop[f: univ -> one univ] : univ.f {
  rel/ran[f]
}

fun drop[span: univ -> univ -> univ] : ((span.univ).univ) -> (univ.(univ.span)) {
  ter/select13[span]
}

// pullback
pred pullback[ X: univ, Y: univ, f: X -> one univ, g: Y -> one univ,
               P: univ, p1: P -> X, p2: P -> Y ] {
  (no X or no Y) implies { no P } else {
  all x: X | all y: Y | x.f = y.g implies
    { one p: P | p.p1 = x && p.p2 = y }

  all p: P | p.p1.f = p.p2.g
  //#P = #(f.~g)
  }
}
```

Appendix B

Alloy Numeric-based Multiconcepts Library

```
module multi

open util/relation
open util/ternary

// apply this predicate to any ternary relation
// to make it a valid multirelation representation
pred mRel[r: univ->Int->univ] {
  let dom = r.dom | let ran = r.ran |
    all a:dom, b:ran | no a.r.b || (one a.r.b && a.r.b > 0)
}

// generate a tenary identity relation
fun id3[x: univ] : x -> x -> x {
  { a, b, c: x | a = b && b = c }
}

// matrix multiplication as composition
fun mjoin[m1: univ->Int->univ, m2: univ->Int->univ] : univ->Int->univ {
  let T = dom[m1], U = ran[m1], V = ran[m2] |
  { t:T, n:Int, v:V |
      let r = (t.m1).(id3[U]).(m2.v) | // r : Int -> U -> Int
        (some r) && n = (sum u: U | mul[(select12[r]).u, u.(select23[r])]) }
}

// lift an orinary relation to a multirelation
```

```
fun lift[r: univ->univ] : univ->Int->univ {
  let dom = r.univ, ran = univ.r |
  { a:dom, n:Int, b:ran | a->b in r && n = 1}
}

// drop the multiplicity information
fun drop[m: univ -> Int -> univ] : univ->univ {
  select13[m]
}

// inverse a multirelation
fun inverse[m : univ->Int->univ] : univ->Int->univ {
  flip13[m]
}

// max union
fun union[f1, f2: Int -> univ] : Int -> univ {
  let base = (ran[f1]+ran[f2]) |
    { n : Int, e : base | (some f1.e || some f2.e) && n = larger[f1.e, f2.e] }
}

// min intersection
fun intersect[f1, f2: Int -> univ] : Int -> univ {
  let base = (ran[f1]+ran[f2]) |
    { n : Int, e : base | (some f1.e && some f2.e) && n = smaller[f1.e, f2.e] }
}

// merge
fun merge[f1, f2: Int -> univ] : Int -> univ {
  let base = (ran[f1]+ran[f2]) |
    { n : Int, e : base | n = add[f1.e, f2.e] }
}
```