Informatiesystemen

Bol.com

February 20, 2018

Introduction

Information systems have become a crucial part of any organization. Banks almost fully exist out of information systems, governments keep all data of citizens in their databases and hospitals need data for treating their patients. In almost every part of our daily lives, information systems play a prominent role. That is why the creation of information systems must be done very carefully. For designing such a system, we use a modeling technique. This technique allows us to model each part of the information system so that we can detect errors early and even create multiple possible solutions and pick the best one. This report will cover a model for the website Bol.com, the large on-line department store. This department store features a lot of different products and thus a lot of data with their respective databases.

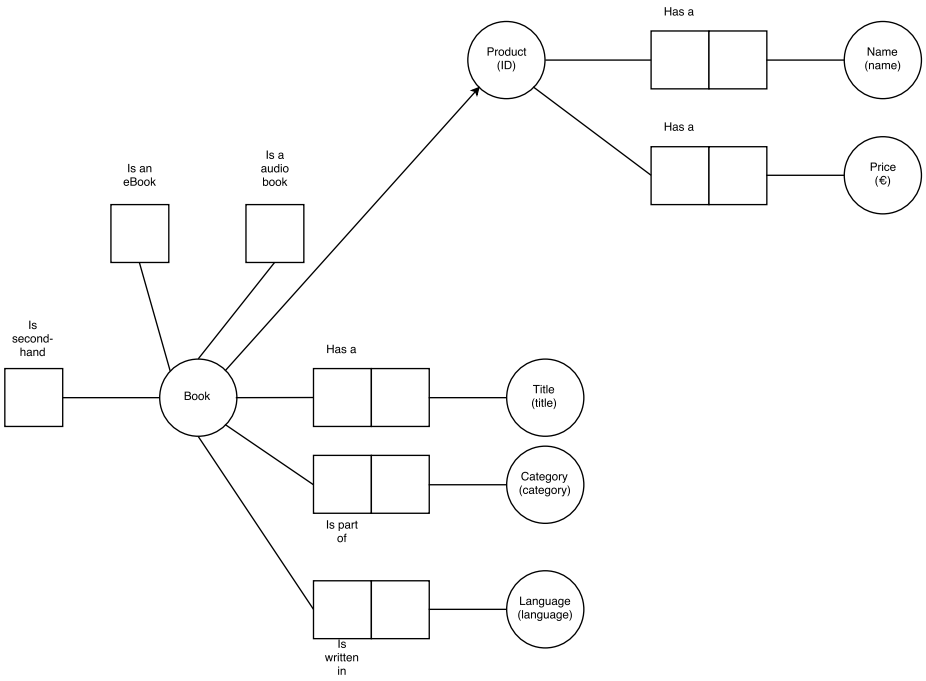

Figure 1: Basic model for books

Books

The Bol.com store features a lot of different products, such as electronics, clothing and toys. This chapter will focus on the category books. This category consist of, you probably already guessed it, books. A basic model for books is provided in figure 1. This model shows two main object types, namely *Product* and *Book*. The *Product* object type is identified by an ID. All products on Bol.com are identified by this ID. Also, all products on Bol.com have a *Price* and a $Name$ as it shows on the web page of a specific product. There are a few choices made in this model that will be explained now, for example the two unary fact types $Is\ an\ eBook$ and $Is\ a\ audio\ book$. These two fact types are modeled as a unary fact type but could also be modeled as a binary fact type like in figure 2. There are no advantages between the two, but the modeling in figure 1 implicitly assumes that if a *Book* is not a audio book or eBook, the *Book* is a normal old fashioned book. This is not the case in figure 2 because if a *Book* plays a role in this fact type, it is either a *Audio book*, *eBook* or *Book*[1]. This model also contains a specialization of *Product* and *Book*. This means a *Book* is a special type of *Product*. All other products at Bol.com can be specialized in such a relation. We can assume that an object type *Electronics* probably exists and is also a specialization of *Product* but because *Electornics* have other relations with other fact types (such as the screen size) the model needs to have a specialization to each category. This is also why the model contains no specialization to each of the book categories. The model shows a binary fact type between *Book* and *Category*. This binary fact type could also have been modeled as a specialization but that would have been a lot less clear because the categories of books are not different enough to make a separate specialization for each of them. In other words, there is no book category that has other fact types than the other ones.

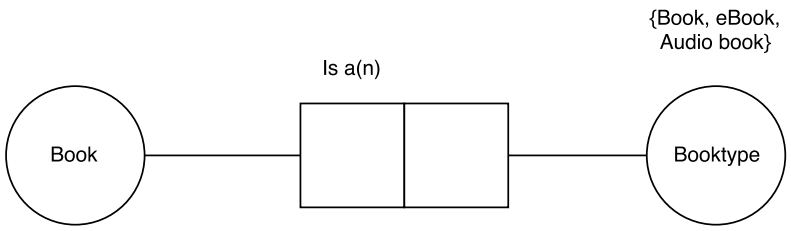

Figure 2: Binary relation for books

Books, more interesting

The basic model was somewhat incomplete. We could, for example, have a book $a1$ in the population of *Book* that is present in $Is\ an\ eBook$ and $Is\ a\ audio\ book$. To fix these errors, the model needs to include constraints. One of these constraints has to be a uniqueness constraint over the roles $Is\ an\ eBook$ and $Is\ a\ audio\ book$. The other added constraints are not that special, some uniqueness and total role constraints are added to make the model more complete.

[1]Actually, a *Book* can also be absent in this fact type due to constraints but this will be covered in the next chapter.

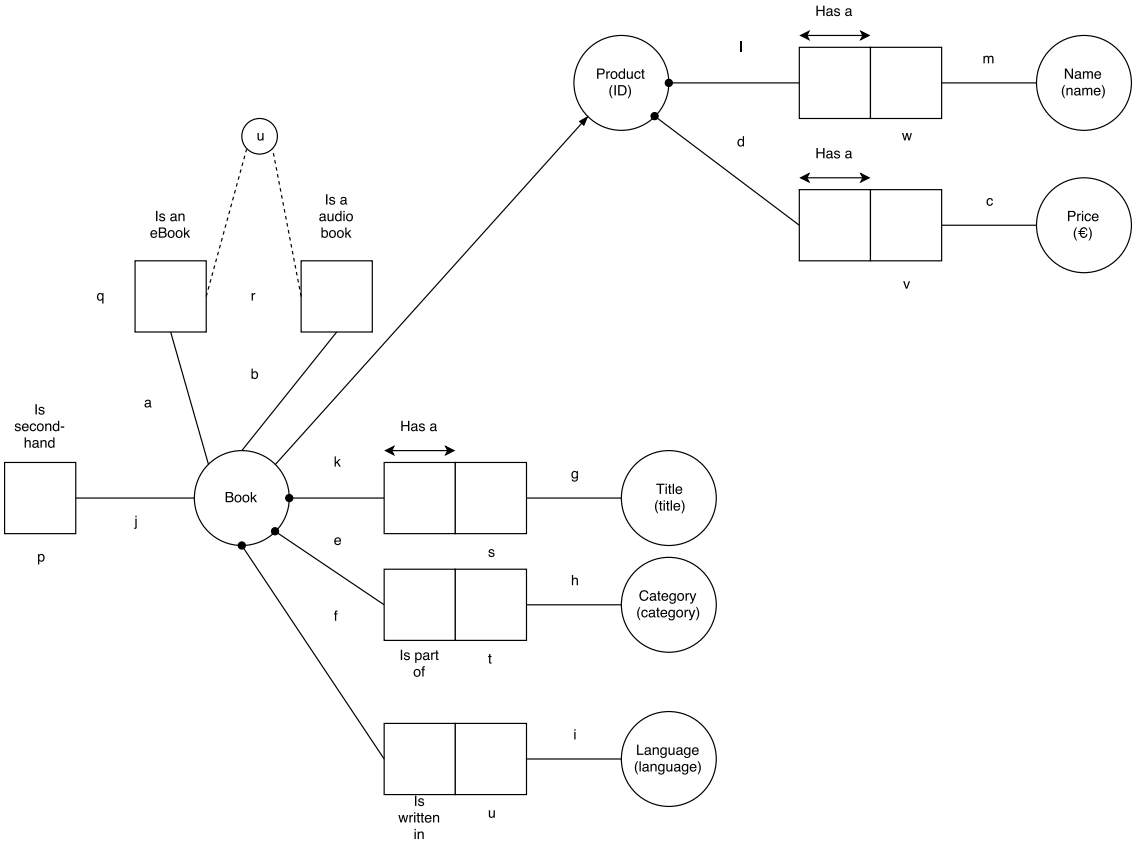

Figure 3: Basic model for books with constraints

We will further examine this model by defining the basic components of it in mathematical terms.

$$
\begin{aligned}
\mathcal{P} &= \{a, b, c, d, e, f, g, h, i, j, k, l, m\} \\
\mathcal{F} &= \{p, q, r, s, t, u, v, w\} \\
\mathcal{S} &= \emptyset \\
\mathcal{L} &= \{ID, title, category, language, name, €\} \\
\mathcal{E} &= \{Book, Product, Title, Category, Language, Name, Price\} \\
\mathcal{G} &= \emptyset \\
\mathcal{C} &= \emptyset \\
\mathcal{O} &= \mathcal{L} \cup \mathcal{F} \cup \mathcal{E}
\end{aligned}
$$

where $p = \{j\}$, $q = \{a\}$, $r = \{b\}$, $s = \{k, g\}$, $t = \{e, h\}$, $u = \{f, i\}$, $w = \{l, m\}$, $v = \{d, c\}$. The Base function is applied below:

$$
\begin{aligned}
\mathsf{Base}(a) &= Book \\
\mathsf{Base}(b) &= Book \\
\mathsf{Base}(c) &= Price \\
\mathsf{Base}(d) &= Product \\
\mathsf{Base}(e) &= Book \\
\mathsf{Base}(f) &= Book \\
\mathsf{Base}(g) &= Title \\
\mathsf{Base}(h) &= Category \\
\mathsf{Base}(i) &= Language \\
\mathsf{Base}(j) &= Book \\
\mathsf{Base}(k) &= Book \\
\mathsf{Base}(l) &= Product \\
\mathsf{Base}(m) &= Name
\end{aligned}
$$

There are no *power types* in this model so the function Elt can not be applied here. This model also lacks generalizations so the Gen function can not be applied. The function Spec however can be applied to the specialization *Book*:

$$Book \ \mathsf{Spec} \ Product$$

The domain of € is considered to be:

$$\mathsf{Dom}(€) = \{x | x \in \mathbb{R}^+ \text{ and } x \text{ is rounded down to two decimals}\}$$

Because of the specialization defined above, the population of *Book* depends upon the population of *Product*. The specialization ensures that all the elements in the population of *Book* are also in *Product*.

$$Pop(Book) \subseteq Pop(Product)$$

Recommendations

When viewing the page of the book *Oorsprong* at Bol.com, recommendations appear at the side of the page. These recommendations are either selected by Bol.com themselves or by their automated system that checks what customers that already bought *Oorsprong* also bought. These recommendations are very important for Bol.com as they could provoke the user to buy more than just that book that the user clicked on. All these recommendations are kept in a database that links books together. The recommendations are modeled in figure 4. The model introduces two binary fact types, one for the automated selection and one for the Bol.com selection. We want to make sure that a combination in *Automated* is not in *Selected* and vice versa. The figure contains four uniqueness constraints named A, B, C and D. We can ask ourself, what is the best way to model this recommendations feature?

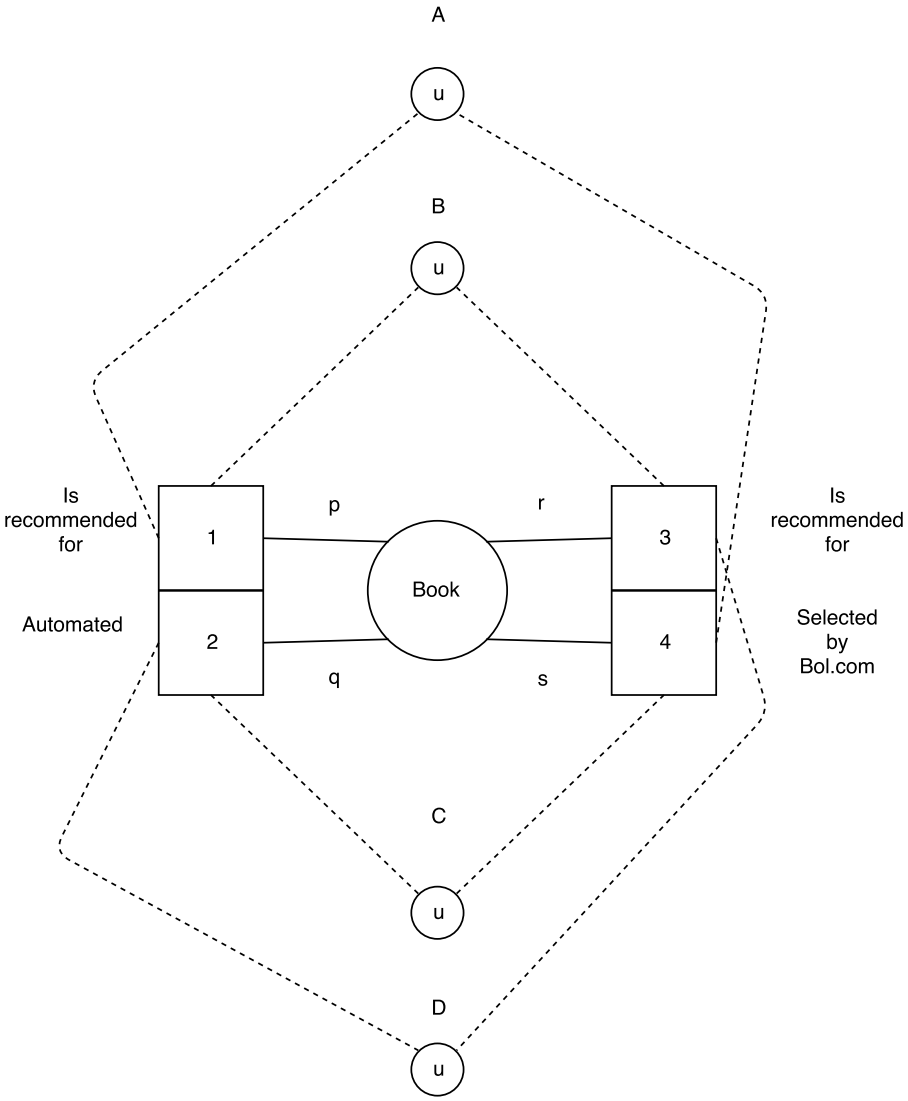

Figure 4: Model for recommendations

Constraint A

Let's begin by exploring uniqueness constraint A. When choosing constraint A as the active constraint, we can create a new model by joining *Automated* with *Selected* and using the condition $q = r$.

$$Automated \bowtie_{q=r} Selected$$

The resulting model is shown in figure 5. A sample population is given: $Pop(f) = \{\{p : a1, q : a2, r : a3\}, \{p : a1, q : a2, r : a4\}\}$. In the first model, this population would be: $Pop(Automated) = \{\{p : a1, q : a3\}, \{p : a1, q : a4\}\}$ and $Pop(Selected) = \{\{r : a3, s : a2\}, \{r : a4, s : a3\}\}$. In this context, this would mean the following. Let's say the automated system selects a book $a1$ that will be in the recommended section of $a3$ and a Bol.com employee selects a book $a2$ that needs to have book $a3$ in its recommended section. Then there can be no more than one book $a3$. Because Bol.com wants to have more than one of such books $a3$ this is the wrong constraint for this situation.

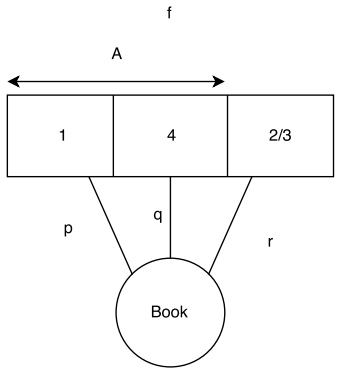

Figure 5: Constraint A

Constraint B

Constraint B is a very different constraint than constraint A. This constraint makes sure that the combinations made in p and r are unique. We can, just as with constraint A, join *Automated* with *Selected* by using the condition $q = s$.

$$Automated \bowtie_{q=s} Selected$$

This join can be seen in figure 6. The semantic meaning of this constraint is the following. If the automated system selects a book $a1$ to be recommended for a book $a2$ and the non-automated system selects a book $a3$ to be recommended for a book $a2$, there can be no book $a4$ in the place of book $a2$. In other words, if the automated system and the non-automated system select two books $a1$ and $a3$ this combination can only be linked to one book. Because Bol.com want to have more than one of such books $a2$, this is the wrong constraint for this situation.

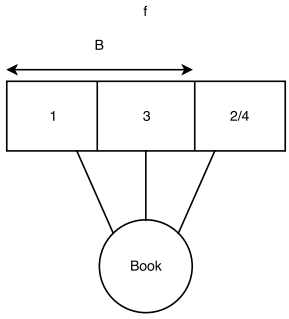

Figure 6: Constraint B

Constraint C

The next constraint we look at is constraint C. The model we get by joining the two fact types *Automated* and *Selected* on condition $p = r$ can be seen in figure 7. This constraint is almost semantically equivalent to the previous constraint. This constraint ensures that a combination of books selected by the automated system and the non-automated system can only have one

recommended book. This is also not the constraint we are looking for when modeling the recommendation system of Bol.com.

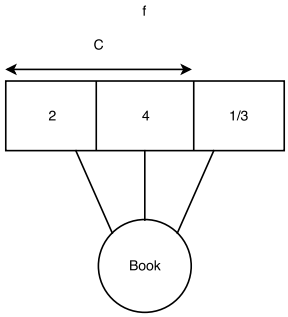

Figure 7: Constraint C

Constraint D

The last constraint we look at is constraint D. As in the previous sections, we first join the two fact types on the condition $p = s$. The model we get out of this join is almost equal to the model in Constraint A. The only difference is that the roles *Automated* and *Selected* are switched. This means that this last constraint is also not the one we are looking for.

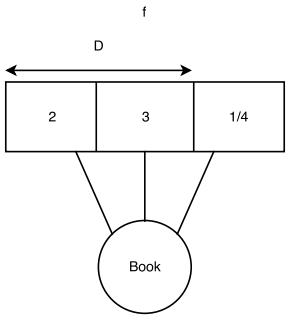

Figure 8: Constraint D

Exclusion constraint

As discussed in the previous subsection, all the constraints shown in figure 4 are not suitable for the model we want to make, namely that if a book-book combination $\{p : a1, q : a2\}$ exists in *Automated*, it does not exist in *Selected* and vice versa. In other words, a combination of books is either in *Selected* or in *Automated*. We do not need a uniqueness constraint but we need an exclusion constraint. This is shown in figure 9. This exclusion constraint makes sure that a

combination of $\{p : a1, q : a2\}$ in *Automated* is not in *Selected*. A sample population may look like this: $Pop(Automated) = \{\{p : a1, q : a2\}, \{p : a3, q : a4\}\}$ and $Pop(Selected) = \{\{r : a1, q : a3\}, \{r : a2, q : a3\}\}$. We could not add $\{r : a1, s : a2\}$ to *Selected* because that would violate the exclusion constraint over *Automated* and *Selected*.

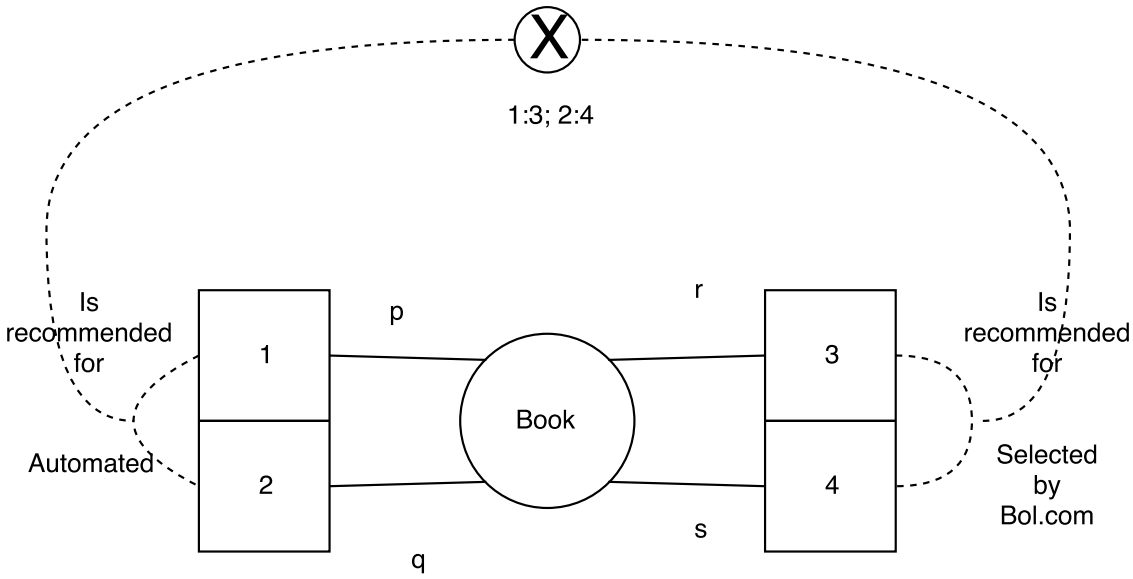

Figure 9: Exclusion constraint

Computers

The web shop Bol.com also sells all sorts of computers. From laptops, to desktops to tablets. In this section, we will take a look at the laptop and desktop computers. These groups are modeled in figure 10.

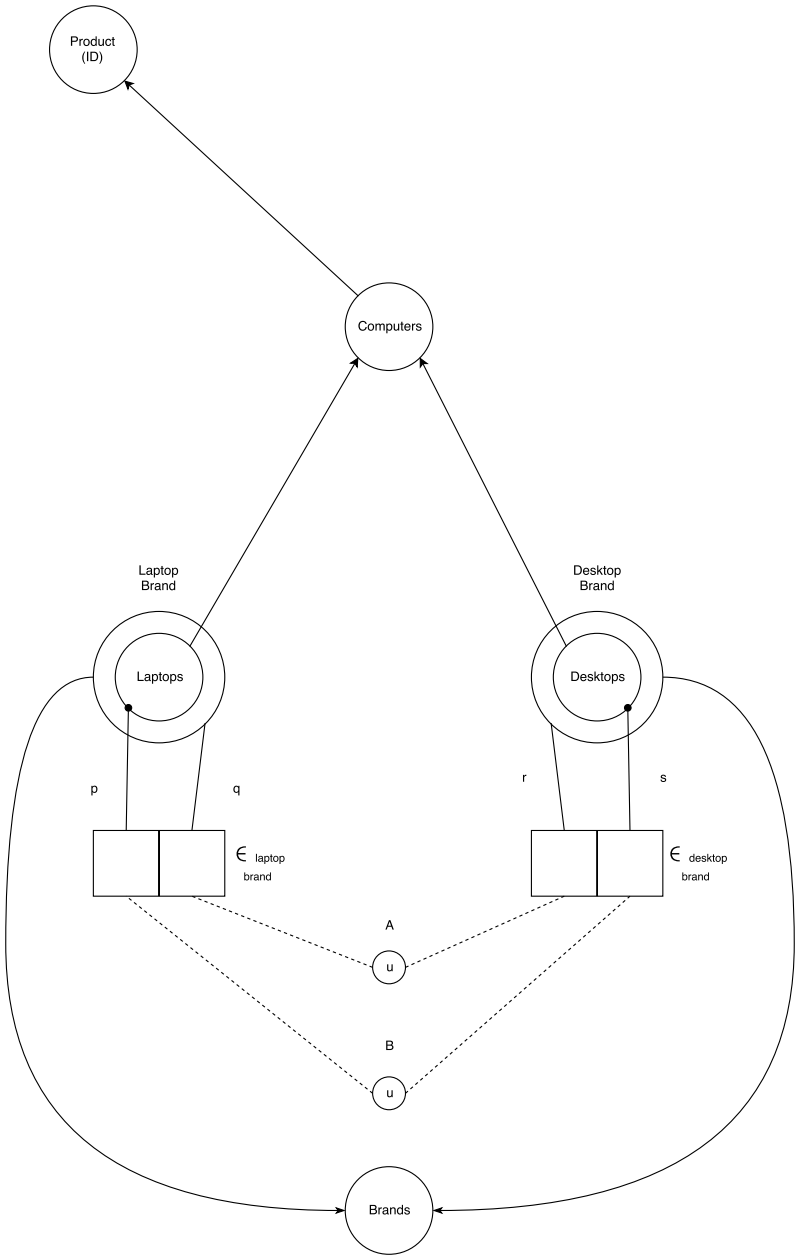

Figure 10: Computers

We can see that each of these groups of products is a specialization of the object type *Computers*. These specializations are then grouped together by brand in either of them. We are going to take a look at the two uniqueness constraints at the bottom of the model, constraint A and constraint B. What impact do they have on the population of *Laptop Brand* and *Desktop Brand*? Before looking at the two constraints we can create a population for all object

types. A sample population is given below:

$$
\begin{aligned}
Pop(Computers) &= \{c1, c2, c3, c4, c5, c6, c7\} \\
Pop(Laptops) &= \{c1, c2, c3, c4\} \\
Pop(Desktops) &= \{c5, c6, c7\} \\
Pop(Laptop\ Brand) &= \{\{q : b1, p : (c1)\}, \{q : b2, p : (c2, c3)\}, \{q : b3, p : (c4)\}\} \\
Pop(Desktop\ Brand) &= \{\{r : b4, s : (c5, c6)\}, \{r : b2, s : (c7)\}\}
\end{aligned}
$$

This sample population will be changed in the following subsections.

Constraint A

Constraint A is a uniqueness constraint over the two "group" ends of the two relations. In this model, it is hard to see what impact this constraint has over the populations of the fact types. We can rewrite the model to make it a little more clear what happens when adding either of the constraints. This rewritten model is shown in figure 11.

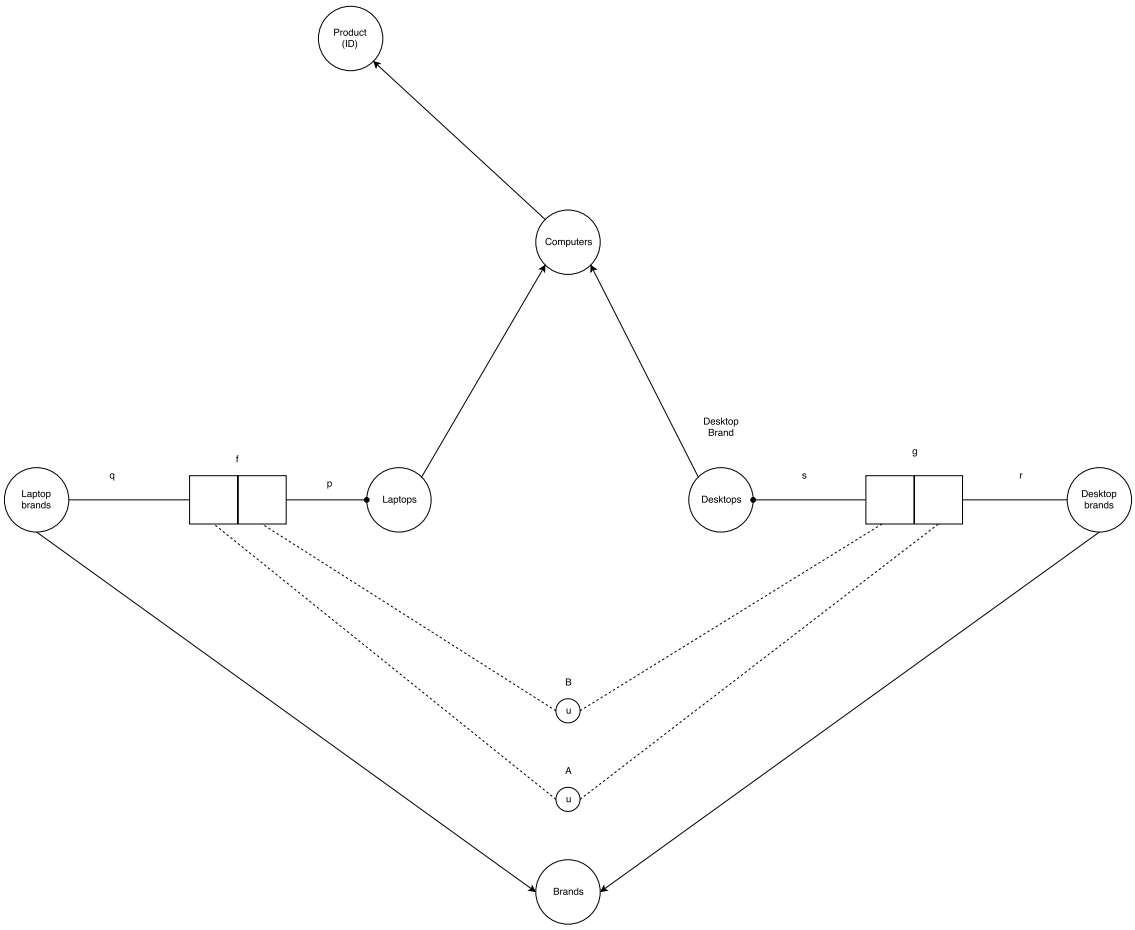

Figure 11: Computers without groups

Now, it is easy to see that a population of a group can not contain more than one of the same computers. The sample population of the previous section is thus still valid because no computer is in Laptops and in Desktops. This makes sure that no computer can exist in either a group of

Desktops or Laptops. For illustrative purposes, a non-allowed population is given below:

$$\begin{aligned} Pop(Computers) &= \{c1, c2, c3, c4, c5, c6, c7\} \\ Pop(Laptops) &= \{c1, c2, c3, c4\} \\ Pop(Desktops) &= \{c1, c3, c5, c6, c7\} \\ Pop(Laptop\ Brand) &= \{\{q : b1, p : (c1, c2, c3)\}, \{q : b2, p : (c4)\}\} \\ Pop(Desktop\ Brand) &= \{\{r : b3, s : (c1, c3, c5)\}, \{r : b4, s : (c6, c7)\}\} \end{aligned}$$

This population is not allowed because $b1$ and $b3$ share two computers, namely $c1$ and $c3$. An allowed population would look like this:

$$\begin{aligned} Pop(Computers) &= \{c1, c2, c3, c4, c5, c6, c7\} \\ Pop(Laptops) &= \{c1, c2, c3, c4\} \\ Pop(Desktops) &= \{c1, c5, c6, c7\} \\ Pop(Laptop\ Brand) &= \{\{q : b1, p : (c1, c2, c3)\}, \{q : b2, p : (c4)\}\} \\ Pop(Desktop\ Brand) &= \{\{r : b3, s : (c1, c5)\}, \{r : b4, s : (c6, c7)\}\} \end{aligned}$$

This population is allowed because $b1$ and $b3$ only share one computer instead of two. We could translate this population to a population of the second model by changing all groups to binary relations:

$$\begin{aligned} Pop(Computers) &= \{c1, c2, c3, c4, c5, c6, c7\} \\ Pop(Laptops) &= \{c1, c2, c3, c4\} \\ Pop(Desktops) &= \{c1, c5, c6, c7\} \\ Pop(Brands) &= \{b1, b2, b3, b4\} \\ Pop(Laptop\ Brands) &= Pop(Brands) \\ Pop(Dekstop\ Brands) &= Pop(Brands) \\ Pop(f) &= \{\{q : b1, p : c1\}, \{q : b1, p : c2\}, \{q : b1, p : c3\}, \{q : b2, p : c4\}\} \\ Pop(g) &= \{\{r : b3, s : c1\}, \{r : b3, s : c5\}, \{r : b4, s : c6\}, \{r : b4, s : c7\}\} \end{aligned}$$

Constraint B

Constraint B is a uniqueness constraint over the two "item" ends of the two relations. Constraint B is also specified in figure 11. This constraint is harder to understand than the previous one. Therefor, we join relations f and g. This is possible because Laptop brands and Desktop brands are type related. We join f and g by: $q = r$. This is done in figure 12.

$$f \bowtie_{q=r} g$$

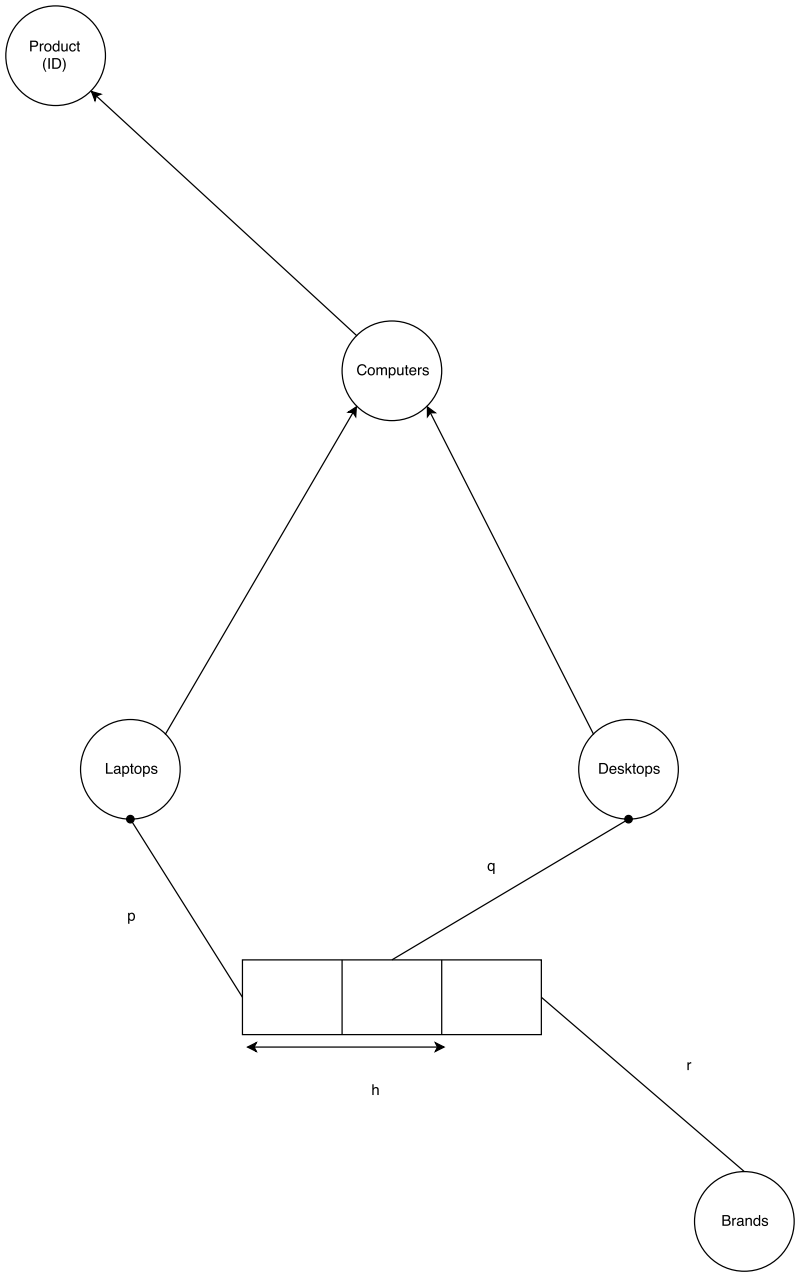

Figure 12: Joined computers

To explore the effect this constraint has on our model, we create a non-allowed population:

$$\begin{aligned} Pop(Computers) &= \{c1, c2, c3, c4, c5, c6, c7\} \\ Pop(Laptops) &= \{c1\} \\ Pop(Desktops) &= \{c5, c6\} \\ Pop(Brands) &= \{b1, b2\} \\ Pop(h) &= \{\{p : c1, q : c5, r : b1\}, \{p : c1, q : c6, r : b1\}, \{p : c1, q : c5, r : b2\}\} \end{aligned}$$

This can be translated to the original model:

$$
\begin{aligned}
Pop(Computers) &= \{c1, c2, c3, c4, c5, c6, c7\} \\
Pop(Laptops) &= \{c1\} \\
Pop(Desktops) &= \{c5, c6\} \\
Pop(Brands) &= \{b1, b2\} \\
Pop(Laptop\ Brand) &= \{\{q : b1, p : (c1)\}, \{q : b2, p : (c1)\}\} \\
Pop(Desktop\ Brand) &= \{\{r : b1, s : (c5, c6)\}, \{r : b2, s : (c5)\}\}
\end{aligned}
$$

This sample population is non-allowed because the population of h contains the combination $p : c1, q : c5$ twice. This means that population is not allowed if two brands contain two or more of the same computers.

Strengths and weaknesses

We have seen a couple of applications of the theory on some models. With these models, we have studied the corresponding populations and discovered what certain constraints did to those populations. At this point, we will look into the strengths and weaknesses of the modeling technique used so far. The modeling technique we used provides a lot of ways to create a model for specific information systems but are there also points at which this modeling technique is poor? Let's find out.

Specializations

In section Books on page 2 we discussed a basic model for books. This basic model included a specialization named *Book*. This specialization made sure that the population of *Book* was also in *Product*.

$$Pop(Books) \quad \subseteq \quad Pop(Product)$$

This means that we can also identify books in *Book* by an ID. This way, we can specify a price for a product while also keeping all properties of books intact and without essentially copying fact types for each specialization. All product categories at the Bol.com webshop are specified as a specialization in the Bol.com model. This is shown in figure 13.

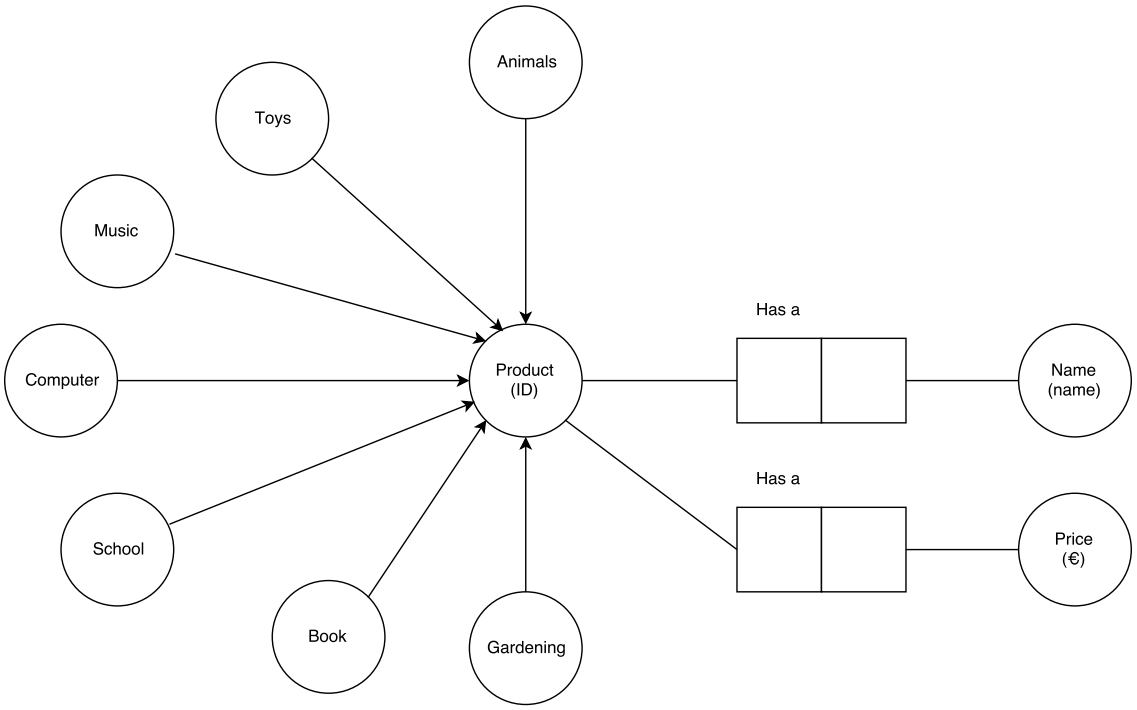

Figure 13: Specializations

This modeling technique is nice, because for each category we can define other fact types. For example, a book is written in a language (as specified in figure 1) but it makes no sense to define a fact type for languages for each element in the population of *Computer*. The amount of products Bol.com is selling is ever increasing, and so are the categories. The problem with the specialization lies in the fact that creating a new category is hard when using this modeling technique. To create a new category, it requires us to add a whole new specialization instead of just changing populations. To create a new category, we thus need to change the model and that is not desirable. What we want to happen when creating a new category is that we only need to change the populations because that is a rather easy thing to do. Can we extend the current modeling technique to change the need of an extra specialization? Or maybe a better question, is this even desirable? The latter question points to the fact that if we want to create a new category, we also need to add all sorts of fact types related to that new category. Therefor, we can not come across the need to actually modify the model when adding a new category. The question now is, can it be done any different? A specialization is effectively a subset of the parent object type. That is why we may be able to create a binary fact type to represent each category, but then, would we be able to distinguish each different type in that binary fact type. The two types that are in the relationship of the binary fact type do not change so we can not create distinction with such a binary fact type. Therefor we need to create a unary fact type for each category and from thereon out use constraints to ensure all products present in such a type are also present in all their fact types that relate to their category. This however, creates a unreadable model and moreover does not take away the need to modify the model when adding a category. The solution to this problem contains an edit to our current modeling technique. We need to add a new binary fact type with a predefined set of categories. With this new fact type, we can create constraints on object types that play a role in the new fact type. A constraint can contain certain conditions building upon our category type. An image of this is shown below in figure 14.

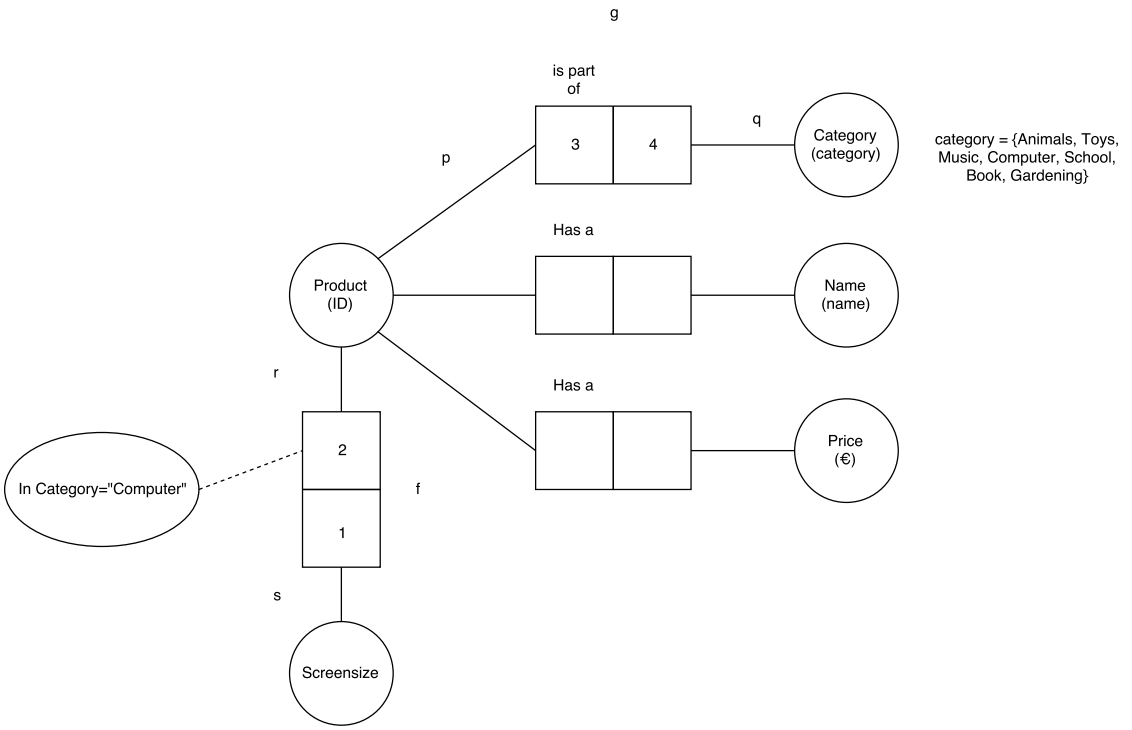

Figure 14: The new constraint

This figure shows us the new constraint over the binary fact type f. The constraint is connected to the $Product$ side of this fact type. The new constraint ensures that the population of f only contains products that are also present in the population of g, and moreover are present in g with a relation to the element $Computer$. This constraint thus not only ensures that two elements are present in both fact types but also ensure that a specific element of an object type has a relation with that element. We need to specify how this constraint can be used and what populations it ensures. Let's first look at a population this constraint denies. A disallowed population is shown below:

$$
\begin{aligned}
Pop(Product) &= \{p1, p2, p3, p4\} \\
Pop(Screensize) &= \{17, 15, 13\} \\
Pop(g) &= \{\{p : p1, q : Animals\}, \{p : p2, q : Music\}, \{p : p3, q : Music\}, \{p : p4, q : Computer\}\} \\
Pop(f) &= \{\{r : p1, s : 17\}, \{r : p4, s : 15\}\}
\end{aligned}
$$

This population is disallowed due to $p1$ being present in the population of f but not being present in g with $Computer$. When deleting $p1$ from $Pop(f)$ we create a valid population for this model.

Implementation

For this new constraint, we will define a couple of rules for how to use it. First of, the constraint consists of two things, an object type with the set of categories and the actual constraint over a fact type. A single category object type can serve multiple constraints. There is no need to create new category types for every constraint we add. First of, let's implement the category type. This type is a special binary fact type which connects two object types, one object type must have a population defined in an enumeration. This enumeration must be predefined because we will use it later in the model. The fact type that connects these two object types has an implicit constraint.

This are necessary because otherwise we will see duplicates in the fact type. The only purpose of the fact type g is to identify which product is in which category. That is why the binary fact type contains an implicit constraint defined below in figure 15.

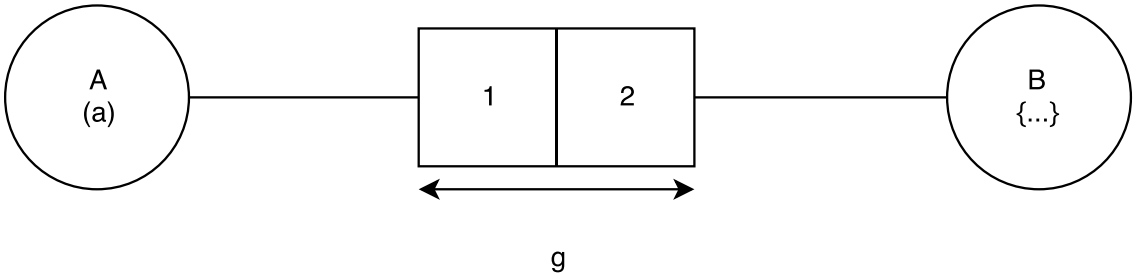

Figure 15: Enumeration fact type

The implicit constraint is a uniqueness constraint over fact type g. This constraint ensures that there are no doubles in the population of g. The population of g does not have to contain doubles because if a specific element $a1$ of A is already in g connected with an element $b1$ of B, the element $a1$ is considered to be in the category $b1$. Adding this combination another time to g will not add any meaning in terms of categories. Let's now define the constraint itself by using the enumeration fact type. The new constraint is connected to one side of a fact type. The constraint can only be connected to the A side of a fact type connected to A. The constraint can contain a lot of conditions based upon the category enumeration. In figure 15 the constraint contains the text: $In\ Category = "Computer"$. Before going into what this piece of text can contain, we will first give some examples:

- In Category="Computer"
- In Category="Music" or Category="Computer"
- In Category="Music" and Category="Computer"
- In Category="Music" and Category2="Computer"

These statements will be elaborated further by firstly defining the grammar.

$$
\begin{aligned}
< statement > &= In < categoryEqual > \\
< categoryEqual > &= < categoryEqual > < andOr > < categoryEqual > \\
&\mid (< categoryEqual >) \\
&\mid < category > = " < element > " \\
< category > &= \text{—Category—} \\
< element > &= \text{—Element—} \\
< andOr > &= and \mid or
\end{aligned}
$$

Where —Category— is a name of a category object type and —Element— is an element of the population of that same —Category— type. We can see that every $< statement >$ starts with In. After that, the recursive definition of $< categoryEqual >$ ensures that there is no limit to the amount of statements placed in $< categoryEqual >$. After all, we are just creating a predicate which takes an element of a type connected to all $< category >$ types and then return either $true$ or $false$. If the predicate returns $true$, the element is allowed to play a role in the constrained fact type, otherwise it is not. For every $< categoryEqual >$ that is chosen to be $< category > = " < element > "$, it is assumed that the $< element >$ is actually an element of

$< category >$, otherwise the predicate is invalid and will always return $false$. The actual object type that is connected to the constrained fact type must also be connected to all categories that the constraint contains in its predicate. A couple of invalid fact types are shown below in figure 16.

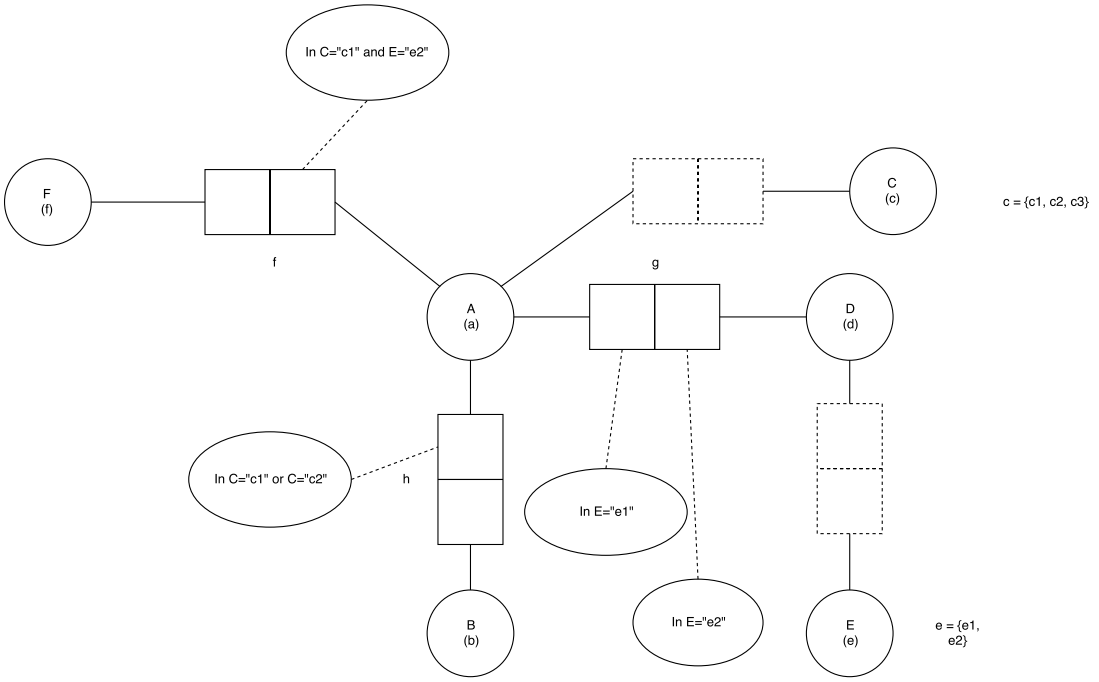

Figure 16: Which ones are wrong?

This figure shows four of our new constraints and introduces a weird fact type with dotted lines. This new fact type shows us all category fact types in the model. We will use this way of modeling a category fact type from now on so it is clear to see which fact types are category ones. Let's look at the constraints. First we will look at the constraint over fact type h, this constraint uses the category type C over the A side of the fact type. We can see that A is connected to the category object type C with a category fact type, that is why this constraint is valid. Let's now look at the constraint over f, we can see that it uses the category object types C and E. The constraint is connected to the A side of the fact type f. This means we should check A for category fact type connections to C and E. From the previous constraint, we already know it is connected to C so we start looking at E. We can see that A is not connected to E so the constraint over f is invalid. The next constraints we look at is the ones over g, the A side of g has a constraint with predicate $In\ E = "e1"$. We thus need to check A for connections to E but as discussed previously, this is not the case so this constraint is also invalid. The last constraint is over the same fact type but the D part of it. D is connected to E with a category fact type so we can conclude that this constraint is valid.

Join

We have seen the syntax and semantic meaning of the newly created category constraint but a rather important question remains, can we join fact types that contain a category constraint? We will figure this out by creating a new model that we will try to join later on. This new model is shown in figure 17.

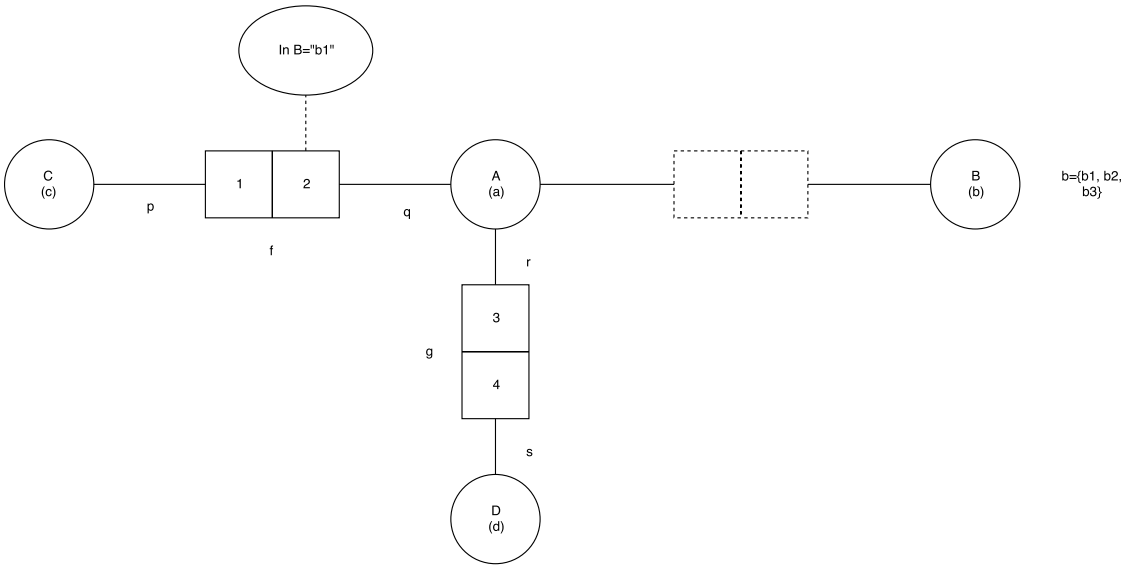

Figure 17: Joining

This figure shows one category constraint over fact f. We will join the fact types f and g with condition $q = r$. After the join, we can create a ternary fact type as seen in figure 18.

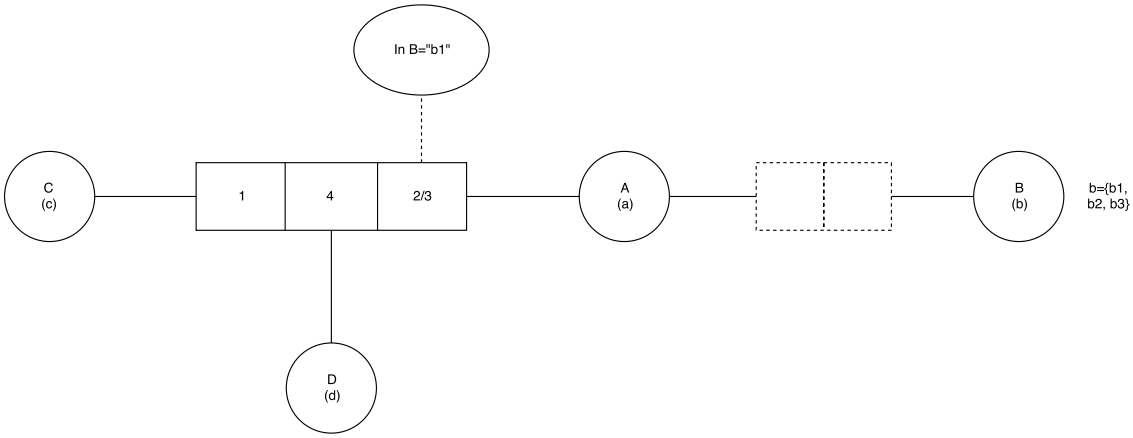

Figure 18: Joined fact types

The constraint that was present in figure 17 is now also present in the fact type of figure 18. The join is thus not all that spectacular when dealing with the new constraint. However, what if there was also a category constraint over the fact type g? Let's look at what two constraints over different fact types do when joined. First we define a new model (that is slightly different from the model in figure 17). This new model is showed below in figure 19

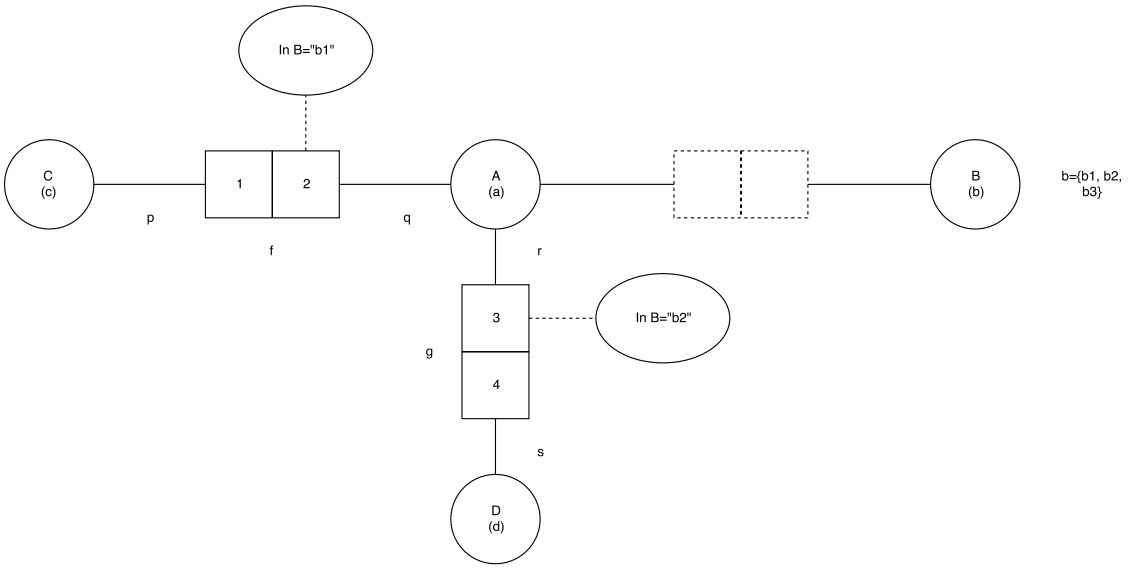

Figure 19: Different join

This model introduces a new constraint over $g3$. This creates difficulty when joining the two fact types because both constraints will be over the same side of the new ternary fact type. These two constraints can be written in only one constraint by using the *and* keyword that is present in the grammar. With *and* and some bracketing we will create a single constraint that is semantically equal to two separate constraints. Before explaining more, we will first look at the newly created ternary fact type in figure 20.

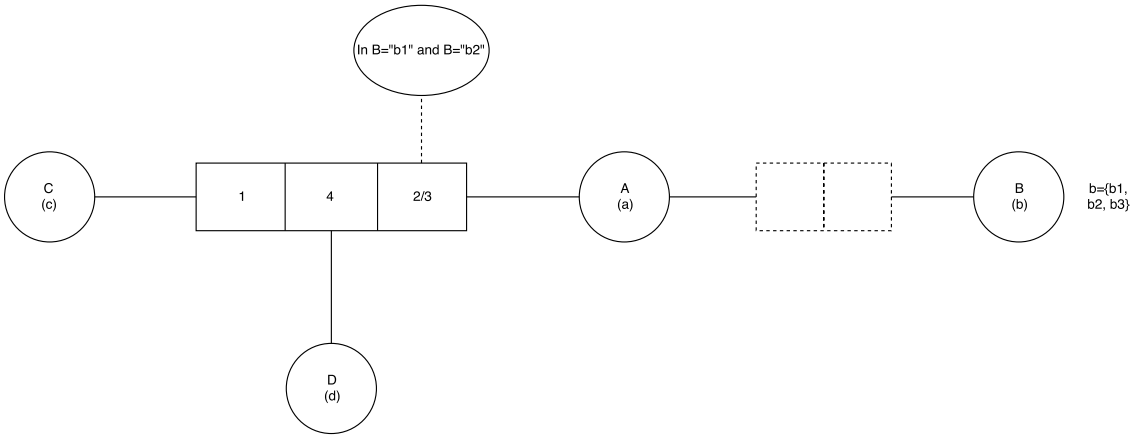

Figure 20: Different joined fact types

As seen in figure 20, we joined the two constraint into one constraint by using *and*. We can define a rule for joining two category constraints into one if the two constraints are over the same part of a fact type. This can be done by editing the constraints like this:

$$\begin{aligned} Constraint1 &= In\ < categoryEqual1 > \\ Constraint2 &= In\ < categoryEqual2 > \\ JoinedConstraints &= In\ (< categoryEqual1 >)\ and\ (< categoryEqual2 >) \end{aligned}$$

This way, the semantics of both individual statements can be combined in one single statement.

Applicability

Now that we have defined the new constraint, let's look at the applicability of it. In the beginning of this chapter, we discussed how specializations can be used to define categories and when defining a new category, the model needs te be changed. Did we come across this by defining the new category constraint? Yes and no. By using the new constraint, adding a new category has become very easy. Just changing the population of the category object type and some category constraints can create a new category. This has only one disadvantage. When creating a new category, we may want to add fact type that is not yet in the model. If this is the case, the model needs to be changed anyhow. However, the model changes are much less radical. Another disadvantage lies in the fact that all fact types that were first split across all specializations are now connected to one object type. This may make the model a lot less readable. But in general, the amount of space the models takes on paper will decrease because of deleting a lot of object types and replacing them by one category object type.

Discussion

The newly created constraint is very handy when creating categories for object types but there are also situations in which specializations are maybe more handy to use. The last question we will ask ourselves, has this constraint created more possibilities for models? We will try to answer this question by using the model of figure 17. This model features two fact types for which one of these types has the new category constraint. We could look at the category type as a lot of unary fact types. For each element in the set the category type, we could create a unary fact type. When an element of A plays a role in a unary fact type, it is considered to be in the category that unary fact type represents. A set constraint can then make sure that the populations of the fact types that had a category constraint stay the same. This can be done when a category constraint does contain only one statement, or more statements connected with *and*. When *or* is used, we have no way of defining the semantics of the *or* statement in terms of other constraints.