Quasi-Finite Domains: Dealing with the Infinite in Mass Customization

Albert Haag[1]

Abstract. In this paper we propose to relax finiteness in relational tables and tabular constraints in a controlled way. We preserve the syntactic representation of a row in a table as a tuple of symbols. Some of these symbols refer to an atomic value as usual. Others, which we call *quasi-finite symbols* (QF-symbols), refer to infinite subsets of an underlying infinite domain. Practical examples for QF-symbols are references to (uncountable) real-valued intervals and *wildcards* representing countably infinite sets. Our goal is to provide a simple and smooth extension of the tabular paradigm, predominant in business, that is compatible with compression of the table to *c-tuples* [14] or to a *variant decomposition diagram* [11], and is amenable to constraint processing, such as local propagation.

The approach is based on organizing the QF-symbols pertaining to each product property in a *specialization relation* [8, 9]. A specialization relation is a partial ordering that expresses specificity of meaning. A QF-symbol can be ignored in the presence of a more special one.

To ensure that the sets represented by two distinct QF-symbols pertaining to the same domain are disjoint, we further require that it must be possible to represent the *intersection* and *set-differences* of QF-symbols. In order to be able to remove duplicates implicated by a disjunction of QF-symbols from result sets of queries, we require that it is possible to represent their *normalized set-union*.

QF-symbols may refer to any objects as long as the above requirements are met, e.g. regular expressions (unary predicates), rectangles (geometric shapes), etc.

1 Introduction

This work expands on a common theme: that data in tabular form is a natural, non-proprietary medium for communicating between interrelated business processes within an enterprise, as well as between enterprises. We focus on *mass customization* (MC), which we take to be *mass production* with a lot size of one. A non-configurable product, amenable to mass production, can have *product variants*[2]. For example, mass produced ballpoint pens come in several colors, but are otherwise identical. The offered colors are in a one-to-one correspondence with the manufactured ballpoint variants. The business attributes are maintained once for the generic ballpoint pen. Only one generic bill of materials that covers all possibilities needs to be maintained. For each variant the value of an additional product property, *color*, is needed to determine which ink filling and matching cap is used in assembling the variant[3].

MC adds *customization* to this setting by placing the emphasis on *individualization*, i.e. there will be many variants of a product and the business is prepared to produce only a single unit of each one on demand (lot size one). Accordingly, the product properties that distinguish the variants are central and potentially numerous[4]. In other work [13] we discuss the MC setting in more detail and show that compression of *variant tables*, tables listing combinations of product features, is a key element in managing the exponential explosion of the number of variants caused by the increase in the number of customization choices, which production technology now enables. Here, we propose to add to the expressivity of variant tables by presenting a *quasi-finite* (QF) framework that allows dealing with infinite sets of choices within the tabular paradigm.

If the domains of all descriptive properties are finite, then the number of variants is finite as well. Leaving the reference to the underlying generic MC product aside, each variant is defined by a value assignment to the properties, which we represent as a *relational tuple* (*r-tuple*). If the number of offered variants is not large, these r-tuples can be maintained as rows in a database table or spreadsheet, which then acts as a product model comprised of a single tabular constraint. If desired or needed, this overall *variant table* can conceptually be split into smaller tables that together form a *constraint satisfaction problem* (CSP). Each CSP variable corresponds to a product property and each CSP solution to an offered product variant[5]. We refer to any tabular constraint on product properties as a *variant table*.

Variant tables are a form of modeling that is very acceptable to a business. Their downside is that they may not scale with a growing number of choices for individualization. However, we show in [13] that expected regularities in the product variants will allow a compressed form of the table to scale. Here, our choice of the compressed form is a *variant decomposition diagram* (VDD) [10, 11], and the associated *c-tuples*, a term adopted from [14] and used there for a Cartesian product of sets of values.

However, infinite domains occur in MC practice, and neither tables of r-tuples nor classic CSP approaches allow infinite sets. In this paper we propose to relax finiteness in variant tables, and by extension in the associated constraint processing, in a controlled way.

[1] Product Management GmbH, Germany, email: albert@product-management-haag.de

[2] We use SAP terminology pertaining to the handling of products and product variants, citing [6] as a general reference. A brief sketch of the history of the SAP Variant Configurator is given in [7]

[3] The SAP tables relevant for configuration are listed in [6], Appendix A

[4] For simplicity of exposition, we disregard the possibility of needing to deal with variant structures, i.e. variants that have variants as parts. Our approach here addresses tabular data in general and could be extended to variant structures if needed.

[5] In practice, product models are not limited to use only tabular constraints. However, the reasoning here shows that product variants could be exclusively expressed in tables in the finite case. The single overall table listing all variants can be seen the the result of *compiling* the product model, e.g. to a *decision diagram* [2].

Our goal is to provide a simple and smooth extension of the tabular paradigm that retains its acceptance in business, and, particularly, allows compression to a VDD and c-tuples. We preserve the syntactic representation of a row in a table as a tuple of symbols, while allowing some of these, which we call *quasi-finite symbols* (QF-symbols), to refer to infinite sets. A symbol for a *real-valued interval*, which is uncountably infinite by definition, is an example of a QF-symbol. A *wildcard* symbol that refers to an infinite domain is also a QF-symbol. In contrast, a *wildcard* for a finite domain is just an *alias* for the finite set of values. We treat this as "syntactic sugar" and equivalent to the expanded set of values.

The approach we take here is illustrated by example in Section 4 and based on the following ideas[6]:

- Each property domain is defined by a finite set of symbols:
  - a finite domain by its *values*, which we refer to as *r-symbols*,
  - an infinite domain by one or more (disjoint) QF-symbols.
- We represent a value assignment to the product properties as a tuple of symbols. If the tuple contains only r-symbols, it is an r-tuple. If it also contains QF-symbols, we call it a QF-tuple. Both r-tuples and QF-tuples are interpreted a special cases of a c-tuple, where an r-symbol in the tuple is treated as a singleton set.
- We adapt the concept of a *specialization relation* from [8, 9] to QF-symbols. When queries or constraint solving need to consider two different QF-symbols for the same property simultaneously, they can ignore both symbols and focus instead on the more *special* symbol for their set intersection.
- Only a finite number of QF-symbols is needed, which can be derived in advance from the product model (e.g. the property domains and the variant tables)[7].
- Compression to a VDD, and through that to c-tuples, can be done as in the finite case, if we can ensure certain requirements are met.

One difference between a QF-symbol and an r-symbol is that the former still allows choice, i.e. it can be *specialized* or restricted further when required. The consequence is that some constraints may need to be formulated in non-tabular form, e.g. to express that for two real-valued properties *length* and *width* it should hold that: $length \geq width$. These constraints can be seen as inter-property predicates. Whereas we discuss unary intra-property predicates as QF-symbols, we will not deal with other inter-property predicates in this paper, except to note in passing that restricting real-valued intervals with numeric linear (in)equalities is an established technique (see Section 8.2) that can be smoothly integrated with our intended processing.

We show how configuration queries over variant tables with QF-symbols can be meaningfully supported. We also believe that the concept of specialization relations is an important bridge to constraint processing in general. The idea of defining a specialization relation via the *subset-relation* can be inverted: given a set of symbols from the column of a table that correspond to elements of a partial order, such that a unique greatest successor and a unique least common predecessor exists for any two elements, these symbols can be treated in a like manner to QF-symbols for purposes of queries and constraint processing, if we are willing to interpret the partial order as a specialization relation.

[6] This extends the simple processing of real-valued intervals and wildcards proposed in [10, 11] for a *set-labeled* VDD.

[7] If further QF-symbols are generated dynamically externally, the specialization relation will have to be extended dynamically. Nevertheless, at any given time a finite number of symbols will be needed.

As stated, the goal of this work is to smoothly extend the tabular paradigm, not to compete with other dedicated problem solving approaches, and we do not make any such comparisons here. The *quasi-finite* (QF) approach has not yet been tried in the field. Therefore, we cannot present results. Given that the VDD processing remains syntactically alike to the finite case, and given our positive experiences with specialization relations in other endeavors, we are confident that performance is not the issue. Instead, it will be a primary concern to establish usefulness in practice and evaluate acceptance by the business community.

The paper is structured as follows:

- We summarize a database approach to configuration in Section 2 and the topic of compression to VDDs and c-tuples in Section 3.
- We illustrate all ideas using an extensive example based on an MC T-shirt in Section 4.
- Constructing VDDs from QF-tuples is akin to constructing them from c-tuples. This topic is beyond the scope of this paper. However, we summarize the basic problem of ensuring disjoint c-tuples (QF-tuples) in Section 5.
- We look at the motivating examples of QF-symbols and how they meet our requirements in Section 6 in some detail.
- We discuss queries to variant tables with QF-symbols in Section 7.
- We present our ideas on specialization relations and their relation to constraint processing in Section 8. In particular we show that local propagation works seamlessly.
- We also believe that using QF-symbols (and perhaps c-tuples in general) directly in the definition of a product variant has business benefits, which we discuss in Section 9.
- We provide a summary and an outlook in Section 10.

2 Configuration in the Database Paradigm

The easiest MC business setting is when the business offering is a small finite set of product variants actually represented in extensional form in a relational database table or spreadsheet. Even when this is not possible, due to the size such a table would have, tabular constraints can be used to define the valid variants.

The extensional form of a tabular constraint naturally supports various data queries such as (1) and (2), here formulated in SQL, which are the most relevant for configuration as discussed in [11][8].

The query in (1) returns a result set of all variants matching the user's criteria[9]. The k product properties are denoted by $v_1, \ldots, v_k$. The variant table is denoted as $\langle vtab \rangle$. $\langle R_j \rangle$ denotes a subset of the domain D_j for product property v_j. The values of interest to a user when configuring can be communicated in the *WHERE* clause.

```
SELECT * FROM ⟨vtab⟩
       WHERE ⟨v1⟩ IN ⟨R1⟩ AND ...⟨vk⟩ IN ⟨Rk⟩;  (1)
```

The query in (2) returns the domain restriction for property v_j under the *WHERE* clause.

```
SELECT DISTINCT ⟨vj⟩ FROM ⟨vtab⟩
       WHERE ⟨v1⟩ IN ⟨R1⟩ AND ...⟨vk⟩ IN ⟨Rk⟩;  (2)
```

[8] While the approach here may be extended to cover further SQL queries, this is beyond the scope of this paper.

[9] In the SQL syntax, an *IN* term in the *WHERE* clause need not be specified where no restriction is intended. However, for purposes of representing a query condition as a c-tuple (see Section 3), we will substitute $R_j = D_j$ for an omitted *IN* term

These queries can also be done to further filter the result sets of previous queries (see [11, 10]).

To sum up: tabular constraints in extensional form can be evaluated using database queries. In [11] we have shown that this extends to tables represented as VDDs in a way that also guarantees the efficiency of the queries. We now have to show here how to handle the queries (1) and particularly (2) in conjunction with QF-symbols.

3 C-Tuples, Table Compression, and Decision Diagrams

The discussion of compression in this section is illustrated with examples using a simple T-shirt in Section 4.

In the finite case a variant can be represented as an *r-tuple*. If we substitute sets for values in this tuple, the tuple is no longer relational, but represents the Cartesian set of all r-tuples that can be formed as combinations using values from the sets. We call such a Cartesian tuple a *c-tuple*[10]. As a tuple we denote it by $\mathbf{C} = \langle C_1, C_2, \ldots, C_k \rangle$, where $C_j \subset D_j$ and D_j is the domain of the product property $v_j \in \{v_1, \ldots, v_k\}$. As a Cartesian set it would be written as $\mathbf{C} = C_1 \times C_2 \times \ldots \times C_k$.

In the context of the above definition, we don't care whether an element C_j of a c-tuple is finite or infinite. Note that the set of r-tuples represented by a c-tuple is uncountable if one of the sets in the c-tuple refers to a real-valued interval.

The *WHERE* clause with the k *IN* operators in (1) and (2) itself describes a c-tuple $\langle R_1, \ldots, R_k \rangle$, which expresses the set of values the user (the problem solving agent) believes in. We will refer to this c-tuple as the *query condition*. We will allow a query condition to be any c-tuple from our variant domain.

C-tuples offer a way to compress tables. For example, if the set of all variants is totally unconstrained, this can be represented by a single c-tuple, which is the Cartesian product of the product domains. With constraints, there will be more c-tuples, but often a c-tuple representation is much more compact than the extensional form [12]. For this reason, c-tuples are already used both formally and informally in configuration practice.

In the case of finite domains, a set of c-tuples can be further compressed to a decision diagram (DD). We use the form of a *Variant Decomposition Diagram* (VDD). As introduced in [10, 11], a VDD is a binary rooted *Directed Acyclic Graph* (DAG), where each node has a label denoting the assignment of a property to a value (r-symbol). Here we will allow QF-symbols in node labels as well. Each node has two emanating links, *HI* and *LO*, which we characterize as follows given a fixed ordering of the product properties: $v_1, \ldots, v_k$:[11]

- the *HI*-link of a node points to a node for the next product property v_{j+1} or to the terminal sink $\top$ (*true*) from last column nodes.
- the *LO*-link points to an alternate value assignment for the same product property v_j or to the terminal sink $\bot$ (*false*).

We will call a chain of nodes linked via *LO*-links an *l-chain*. If more than one QF-symbol appears in an l-chain, the QF-symbols must denote disjoint sets, in order to allow a unique decision for a node, given a value assignment.

Nodes in an l-chain that all have a common HI-link represent the disjunction of their value assignments and could be merged into one *set-labeled node*. In [10, 11] we introduced a VDD with *set-labeled* nodes, where a node was labeled with a finite set of r-symbols representing a disjunction of value assignments. Since any node labeled with such a finite set can be re-expanded into an l-chain of regular VDD nodes nodes that assign the symbols one at a time, we do not propose to use VDDs with set-labeled nodes in practice. We use them here in Section 4 to simplify the exposition.

A VDD is functionally equivalent to the extensional form of the table it represents from the perspective of the queries (1) and (2) relevant for configuration, see [11]. The extension of these queries to include QF-symbols is the topic of Section 7. A VDD can also support counting the number of tuples in a table or a result set of a query and access a tuple directly by its position in the table/result set.

4 T-Shirt Example

4.1 Classic Finite T-Shirt Variants

In [10] the concepts of representing a variant table using a VDD are illustrated using an example of a simple T-Shirt. We use this example here both to illustrate the concepts discussed so far, and also to illustrate the proposed extension to infinite sets.

The simple T-shirt has the three properties *Imprint* (v_1), *Size* (v_2), and *Color* (v_3) with the finite domains:

- $\{MIB$(Men in Black), STW(Save the Whales)$\}$
- $\{L$(Large), M(Medium), S(Small)$\}$
- $\{Black, Blue, Red, White\}$

Only 11 variants are valid due to constraints that state that MIB implies $Black$ and STW implies $\neg S$(Small). Table 1 is the extensional form of the variant table, which is small enough to be used as the only and definitive representation of the variants for the purposes of both business and configuration. It encodes the underlying CSP as a single tabular constraint.

The query (1) can be used to filter the variants to the set matching any given selection criteria (*query condition*) $\langle R_!, \ldots, R_k \rangle$. For example, if the user needs a small (*S*) sized T-shirt, there is only one solution (the first row in Table 1). Alternatively, if a *Red* T-shirt is desired, there are two variants that satisfy this (eighth and ninth rows), and the domains are restricted as follows: $Imprint \in \{STW\}$, $Size \in \{Medium, Large\}$, and Color $\in \{Red\}$ by applying the query (2) for each property in turn.

Table 1. Simple T-shirt

Imprint	Size	Color
MIB	S	Black
MIB	M	Black
MIB	L	Black
STW	M	Black
STW	L	Black
STW	M	White
STW	L	White
STW	M	Red
STW	L	Red
STW	M	Blue
STW	L	Blue

[10] We adapt the term from [14], which investigates direct compression to c-tuples.

[11] Under these assumptions, a *multi-valued decision diagram* (MDD), a more widely known form of a DD [3, 1], can be mapped to a VDD. This is further detailed in [11]

Figure 1 depicts a VDD with set-labeled nodes for Table 1. The HI-links in each path from the root to the sink $\top$ in the VDD in Figure 1 define a c-tuple. The set of all c-tuples that can be formed is disjoint and is a way to represent Table 1 in compressed form. Table 2 lists the two c-tuples needed to represent the 11 variants.

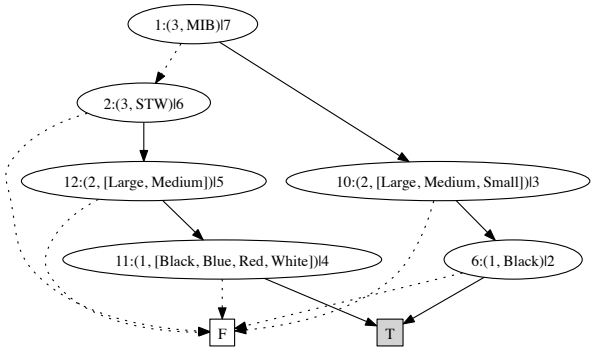

Figure 1. VDD of T-shirt with set-labeled nodes

Table 2. C-tuples for simple T-shirt

Imprint	Size	Color
MIB	S;M;L	Black
STW	M;L	Black;White;Red;Blue

4.2 T-Shirt with Infinite Domains

To illustrate the use of infinite sets, we modify the example to allow an arbitrary user-provided image as an imprint on a white T-shirt. An image is identified at runtime via a file name. The file name must refer to a processable graphic, which is taken to mean that only a jpg or a tiff format can be accepted. Hence, the domain is the infinite set of all legal file names that match the regular expression $\langle$*img-filename*$\rangle = *.jpg| * .tiff$.

We also add a property *Scale* to capture a factor to be used to scale the image printed on the T-shirt. For the vintage prints MIB and STW we require $Scale = 1$. For the user-provided images, the scale can be arbitrarily chosen by the user as a floating point number in the range 0.5 to 1 (the interval $[0.5, 1.0]$).

Table 3 lists the c-tuples needed to describe this setting. The product property *Scale* has here been placed as the first property v_1. The other properties are now v_2 (*Imprint*), v_3 (*Size*), and v_4 (*Color*).

Table 3. C-tuples for simple T-shirt with infinite domains

Scale	Imprint	Size	Color
1.0	MIB	S;M;L	Black
1.0	STW	M;L	Black;White;Red;Blue
$[0.5, 1.0]$	$\langle$img-filename$\rangle$	S;M;L	White

Table 4. Split c-tuples for simple T-shirt with infinite domains

Scale	Imprint	Size	Color
1.0	MIB	S;M;L	Black
1.0	STW	M;L	Black;White;Red;Blue
$[1.0]$	$\langle$img-filename$\rangle$	S;M;L	White
$[0.5, 1.0)$	$\langle$img-filename$\rangle$	S;M;L	White

We now discuss how to construct a VDD with set-labeled nodes for these c-tuples. Figure 2 shows the result[12] .

1. We construct a root node ν_1 labeled $\langle v_1, [1.0]\rangle$ starting with the first c-tuple. This node will be used for both the first and second c-tuples in Table 3.

2. We construct the *l-chain* (chain of *LO*-links) for the root node:
   - We pointed out in Section 3 that nodes linked in an *l-chain* need to have disjoint set labels. This is illustrated here. It will be a problem if we label ν_2 with $[0.5, 1.0]$ (third c-tuple), because then the value $Scale = 1.0$ does not allow deciding uniquely for either ν_1 or ν_2. So we split $[0.5, 1.0]$ into the two disjoint c-tuples.

     $$\begin{array}{llll} [1.0] & \langle\text{img-filename}\rangle & \{S, M, L\} & White \\ [0.5, 1.0) & \langle\text{img-filename}\rangle & \{S, M, L\} & White \end{array}$$

   - Table 4 shows all the c-tuples to be handled in constructing the VDD. The new third c-tuple is covered by ν_1. We label the second node ν_2, linked from ν_1 via its *LO*-link, with the half-open interval $[0.5, 1.0)$ from the fourth c-tuple. This handles the first column.

3. We next process the rest of the three tuples in Table 4 that start with $C_1 := [1.0]$. We create:
   - ν_3, the target for the *HI*-link of ν_1, labeled with $\langle v_2, MIB\rangle$
   - ν_4, the target for the *LO*-link of ν_3, labeled with $\langle v_2, STW\rangle$
   - ν_5, the target for the *LO*-link of ν_4, labeled with $\langle v_2, \langle$img-filename$\rangle\rangle$

4. It is straightforward to handle the third column for the above three c-tuples: nodes ν_3, ν_4, and ν_5 have their *HI*-links pointing to nodes ν_6, ν_7, and ν_8, respectively with the labels depicted in Figure 2:
   - The first of the c-tuples allows only the color *Black* (node ν_9).
   - The second allows all colors (node ν_{10}), and
   - the third allows only the color *White* (node ν_{11}).

   This completely handles the first three c-tuples.

5. It is now trivial to finish the VDD. Node ν_2 still needs to be processed with respect to the last (fourth) c-tuple. But the columns two to four are identical to those in the third c-tuple. Node ν_5 was already constructed for this.

We note that we skirted the issue that the product property *Imprint* (v_2) allows both values from a finite list, e.g. $\{MIB, STW\}$, as well as arbitrary "additional" values ($\langle$img-filename$\rangle$). This is not

[12] To reduce the size needed to display the graph, the terminal sink $\perp$ has been omitted. Conceptually, it terminates all chains of LO-links. Also, the nodes $\nu_1, \nu_2, \ldots \nu_n$ are identified by "n1", "n2", . . . "nn"

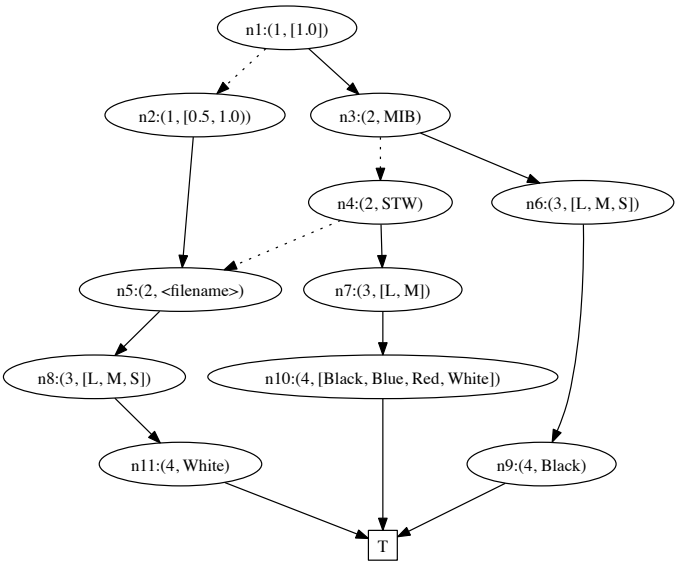

Figure 2. VDD of T-shirt with non-finite set-labeled nodes

uncommon in practice where a "standard" solution is modeled with a predefined finite domain, but *additional values* are allowed (see [6]). The sets $\{MIB\}$, $\{STW\}$ and ⟨img-filename⟩ are effectively treated as disjoint by the VDD due to the constructed *l-chain* of nodes ν_3, ν_4, and ν_5). This could be formally ensured by augmenting the QF element ⟨img-filename⟩ to ⟨img-filename⟩ $\cap \neg\{MIB, STW\}$ (see Section 6).

Lastly, we informally discuss some exemplary queries. QF queries are the subject of Section 7. The query to Table 1 for small (*S*) T-Shirts yielded a result set consisting of one r-tuple (the first row). The domains for the three product properties were restricted to $\{MIB\}$, $\{S\}$, and $\{Black\}$. The same query condition against Table 4 yields three c-tuples (the first, third and fourth c-tuple). Each of these c-tuples in the result set must be intersected with the query condition to eliminate the sizes medium (*M*) and large (*L*) that are in the c-tuples but excluded by the query condition. Consequently, the domains for the four product properties are restricted to $[0.5, 1.0]$ $\{MIB,$ ⟨img-filename⟩$\}$, $\{S\}$, and $\{Black, White\}$[13].

Similarly, the query to Table 1 for *Red* T-Shirts yielded a result set consisting of two r-tuples (the eighth and ninth row). Against Table 4 the result set consists of one (the second) c-tuple. After intersection with the query condition the four product properties are restricted to $[1.0]$ $\{STW\}$, $\{M, L\}$, and $\{Red\}$.

Instead, if the query condition simply specifies a file name for a particular image, *my-img.jpg*, then the last two c-tuples would be the result set. They agree completely except in the first column. As there is no need for the split here (as there was when constructing the original VDD), the two tuples could be combined into one[14]:

$$\langle [0.5, 1.0], \langle\text{img-filename}\rangle, \{S, M, L\}, White\rangle$$

[13] Formed by collecting all symbols occurring for each column and calculating the the *union*. The result of the union of the two intervals is *normalized* (see Section 6)

[14] This reduction is actually required where we want the tuples in a result set to be distinct, i.e. to have been *normalized*.

The result set intersected with the external condition is then:

$$\langle [0.5, 1.0], \{\text{my-img.jpg}\}, \{S, M, L\}, White\rangle$$

If the query formulates the additional restriction $Scale \in [0.25, 0.75]$, then the result set intersected with the external condition is:

$$\langle [0.5, 0.75], \{\text{my-img.jpg}\}, \{S, M, L\}, White\rangle$$

5 Excursion on the Construction of VDDs from C-Tuples

A c-tuple $\mathbf{C}$ can be decomposed into its *head* (the first element C_1) and its *tail* $\mathbf{T}$, which is also a c-tuple. We denote this by $\mathbf{C} := C_1|\mathbf{T}$:

$$C := \langle C_1, \ldots, C_k\rangle = C_1|\langle C_2, \ldots, C_k\rangle = C_1|\mathbf{T} \qquad (3)$$

When constructing a (partial) VDD from a list of c-tuples $\mathbf{C}_1, \ldots \mathbf{C}_m$ an *l-chain* for the head (root) node is constructed using the first elements $C_{11}, C_{21}, \ldots, C_{m1}$. As discussed in Section 3 and evident from the example in Section 4.2 these elements must be disjoint.

If there are two c-tuples $\mathbf{C}_i, \mathbf{C}_{i'}$ with the same tail, i.e.

$$\mathbf{C}_i = C_{i1}|\mathbf{T} \quad \text{and} \quad \mathbf{C}_{i'1} = C_{i'1}|\mathbf{T}$$

then their first elements must be merged to yield one c-tuple

$$\mathbf{C}' = (C_{i1} \cup C_{i'1})|\mathbf{T}$$

We can ensure disjointness of any other pair of c-tuples $\mathbf{C}_i, \mathbf{C}_{i'}$ with differing tails

$$\mathbf{C}_i = C_{i1}|\mathbf{T}_i \quad \text{and} \quad \mathbf{C}_{i'1} = C_{i'1}|\mathbf{T}_{i'}$$

by replacing them with the three c-tuples $\mathbf{C}_a, \mathbf{C}_b, \mathbf{C}_c$ in (4) (a c-tuple with an empty element is considered empty and can be disregarded):

$$\begin{aligned} \mathbf{C}_a &= (C_{i1} \setminus C_{i'1})|\mathbf{T}_i \\ \mathbf{C}_b &= (C_{i'1} \setminus C_{i1})|\mathbf{T}_{i'} \\ C_c &= (C_{i1} \cap C_{i'1})|(\mathbf{T}_i \cup \mathbf{T}_{i'}) \end{aligned} \qquad (4)$$

As the example in Section 4.2 shows, the c-tuple heads show up directly as labels of set-labeled nodes. We already stated that a set-labeled node labeled with a finite set of symbols (r-symbols or QF-symbols) can be expanded to an l-chain of regular VDD nodes.

6 Operations with Non-Finite Elements in C-Tuples

As the discussion in Section 5 and the example in Section 4 make clear, it will be necessary to both split and combine c-tuples when constructing a VDD and result sets. Therefore, we need the following operations on c-tuple elements $C_{ij}, C_{i'j}$ pertaining to the same product property v_j:

- set intersection: $C_{ij} \cap C_{i'j}$
- set union: $C_{ij} \cup C_{i'j}$
- negation with respect to the overall domain: $\neg C_{ij} := D_j \setminus C_{ij}$
- set difference: $C_{ij} \setminus C_{i'j} = C_{ij} \cap \neg C_{i'j}$

For finite sets this is a given. For QF-symbols that are used in the labels of VDD nodes, we must ensure that these operations are well defined and fit in our QF framework.

In the following subsections we look at this in detail for the infinite elements we propose to add:

- Real-valued intervals
- Unconstrained countably infinite sets
- Sets of exclusions, particularly *finite* exclusion sets.

Where the domain underlying negation needs to be made clear we will denote negation as:

$$\neg C := \overline{C}^D = D \setminus C$$

6.1 Real-Valued Intervals and the *Xnumeric* Datatype

We denote a real-valued interval using conventional mathematical notation, e.g. $[a, b)$ for a half-open interval with a closed lower bound a and an open upper bound b. This is the set of all real numbers x such that $x >= a \wedge x < b$. We allow lower and upper infinity, denoted by $-\inf$ and $+\inf$, with open bounds. A single real number x can be encoded as a singleton interval $[x]$. All other interval bounds can be open or closed

We define an *xnumeric* to be a finite list of real-valued intervals representing the union of its elements in a *normalized* form. *Normalized* means that the intervals in the list are disjoint, separable, and in ascending order, e.g. the set of intervals $\{[0.5, 1.0), [1.0]\}$ is disjoint and ascending, but it is not separable. In normalized form it is just $[0.5, 1.0]$. (Remark: The interval $\{[0.5, 1.0), (1.0, 2.0]\}$ is separable and thus normalized, because its two intervals are separated by the "gap" of the singleton interval $[1.0]$.)

For *xnumerics* it is straightforward to ensure normalization. First, any intervals that are non-disjoint or not separable can be merged into one interval. Since the remaining intervals are disjoint, they can be ordered. Hence the union of two xnumeric is just the set union followed by normalization. The intersection is just the list of pairwise intersections. Because the xnumeric is ordered due to normalization, this operation is efficient in the sense that it is not necessary to actually intersect all pairs.

The set union of two intervals is not necessarily again an interval, hence we need the concept of an *xnumeric*.

The *unconstrained xnumeric* is the interval $(-\inf, +\inf)$. The negation of an xnumeric is the set difference to this unconstrained set. It is formed by inverting the finite number of "gaps" between the intervals in the xnumeric. For example

$$\neg\{[0.5, 1.0), (1.0, 2.0]\} = \{(-\inf, 0.5), [1.0], (2.0, +\inf)\}$$

Remark: a finite set of real number values can be represented as an xnumeric using singleton intervals. All interaction with finite sets is covered by the above operations defined for xnumerics.

An xnumeric is a list of QF-symbols (intervals) representing their set union. A set-labeled node for an xnumeric can be expanded to an *l-chain* of nodes with interval labels.

6.2 Countably Infinite Sets and Domains

Examples of countably infinite domains are the list of all integers or all strings. This requires that each product property is associated with an immutable datatype. We consider a domain D or any $C \subset D$ to be qualified by a unary *predicate* (condition) that filters out disallowed values at run-time (e.g. a regular expression for a string). Any value fulfilling the predicate (e.g. any string matching the regular expression) is an acceptable value. Examples for qualifying predicates for an integer datatype are: *positive_p*, *even_p* or *odd_p*.[15]

[15] In the absence of more specialized predicates, $\langle true \rangle$ is taken as the default predicate.

A unary predicate can be represented by its name (a symbol), which serves as the QF-symbol identifying it. In the example in Section 4.2, we used the notation $\langle$*img-filename*$\rangle$ to refer to a regular expression for legal file names.

The set operations translate into logical operations for predicates. The union of two infinite sets qualified by predicates π_1 and π_2 is just a set qualified with the disjunction $\pi_1 \vee \pi_2$. Similarly, intersection translates to $\pi_1 \wedge \pi_2$, and negation to $\neg\pi_1$.

Again, we require normalization to reduce a complex logical expression by removing any redundant elements. It has yet to be determined what works best in practice here. From a theoretical view, we might require a *disjunctive normal form* (DNF). The overall predicate could then be represented as a list (finite set) of conjunctions. A set-labeled node for such a list can be expanded to an *l-chain* of nodes, as for xnumerics. Each such node would be labeled by a conjunction of predicates, which would be treated as an indivisible QF-symbol.

We must also deal with set unions between finite sets and countably infinite sets. In the example in Section 4, the standard imprints for the T-shirt formed a finite set $\{MIB, STW\}$, but "additional values" were then allowed, which were specified by the QF-symbol $\langle$img-filename$\rangle$. The domain of the property imprint is just the union of these sets. Generally, the domain D for a product property with a *non-xnumeric* datatype is $D = \{F, \pi\} := F \cup \pi$, where F is a finite set of values, π a predicate representing an countable infinite set, and both F and π respect the datatype assigned to the product property.[16]

The set-operations then become:

- set intersection: $\{F, \langle\pi\rangle\} \cap \{F', \langle\pi'\rangle\} = \{F \cap F', \langle\pi \wedge \pi'\rangle\}$
- set union: $\{F, \langle\pi\rangle\} \cup \{F', \langle\pi'\rangle\} = \{F \cup F', \langle\pi \vee \pi'\rangle\}$
- negation: $\neg\{F, \langle\pi\rangle\} := \overline{F}^{\langle\pi\rangle} \cap \neg\langle\pi\rangle$
- set difference: $\{F, \langle\pi\rangle\} \setminus \{F', \langle\pi'\rangle\} = \{F'', \neg F', \langle\pi\rangle \setminus \langle\pi'\rangle\}$
  - where F'' is the finite set $F \setminus F' \cup F \setminus \langle\pi'\rangle$, and
  - the finite set $\neg F' = \overline{F'}^{\langle\pi\rangle}$ is an *exclusion set* of all values in F' that lie in $\langle\pi\rangle$ (see Section 6.3).

6.3 Exclusions and Exclusion Sets

An *exclusion* of a value x from a property domain D is a way of stating $D \setminus \{x\}$. It means that x is considered to be invalid, which we will denote by $\neg x$. For real-valued domains, exclusions can be directly formulated as xnumerics, e.g., $\{(-\inf, x)(x, +\inf)\}$ would exclude the real number x. For a finite domain or an xnumeric domain, we can simply positively represent the set $D \setminus \{x\}$. For a countably infinite domain, we need further expressiveness. Given a countably infinite domain D for a product property and a finite set of values $E \subset D$, we introduce an *exclusion set* $\neg E := \overline{E}^D := D \setminus E$. An exclusion set $\neg E$ can be merged with a unary predicate π by removing any values from E that do not satisfy the predicate π, i.e. $\neg E \cap \langle\pi\rangle \subset \neg E$ is a reduced exclusion set. In order to keep the exposition simple, we will ignore this reduction and denote $\neg E \cap \langle\pi\rangle$ also by $\neg E$.

For two exclusion sets $\neg E, \neg E'$, the required set operations are inverted:

- set intersection: $\neg E \cap \neg E' = \neg(E \cup E')$
- set union: $\neg E \cup \neg E' = \neg(E \cap E')$
- negation: $\neg\neg E = D \setminus (D \setminus \neg E) = E$

[16] Ideally, F and π will be disjoint. Either F or π can be empty. We define the predicate $\langle$*false*$\rangle$ to represent the empty set.

- set difference: $\neg E \setminus \neg E' = E' \setminus E$

Finite exclusion sets are needed in order to meet our requirements of negation of finite sets against infinite domains. The concept can also be extended to infinite exclusion sets. Indeed, a negated unary predicate corresponds to such a set. For example, if the set of all prime numbers is represented by the unary predicate $\langle prime\_p \rangle$, the $\neg\langle prime\_p \rangle$ represents exclusion of all prime integers.

In either case, a reference to an exclusion set is treated as a QF-symbol. For example, for a predicate π, $\neg\pi$ is the symbol representing the exclusion of all values in π.

7 Queries on Quasi-Finite VDDs

In the classic finite case, the result set $\mathfrak{R}$ of the query (1) is a finite set of *r-tuples*. In the QF framework, it is a finite set of *QF-tuples* that may contain both value symbols and QF-symbols. A QF-symbol in the result set must be specialized to conform to the the query condition, e.g. by set intersection with the query condition. Problem solving (PS) must expect the remaining degree of non-determinism.

The query (2) contains the keyword *DISTINCT*. This means any duplicates must be removed from the result set for the particular column (property). We see replacing QF-symbols by their *normalized union* akin to removing duplicates. Therefore, the symbols in the result set, both QF-symbols and r-symbols, must be replaced by their normalized union, which ensures also that remaining symbols are pairwise disjoint.

8 Constraint Processing with Quasi-Finite Symbols

8.1 Specialization Relations

Given two QF-symbols ϕ_1, ϕ_2 for the same CSP variable, we regard ϕ_2 to be more *special* than ϕ_1 if ϕ_2 denotes a subset of ϕ_1. This leads to a partial ordering (PO) of the symbols that occur in the variant tables, which we call a *specialization relation*, introduced and motivated for another context in [9]. Generally, a *specialization relation* on a set of facts expresses specificity of meaning, characterized by the following three properties:

- *Problem solving* (PS) need not consider an otherwise valid fact in the presence of a more special one (*procedural-subsumption property*). This property requires the acquiescence of PS.
- A fact is logically implied by any of its specializations (*semantic-compatibility property*).
- Negation inverts specialization (symmetry-under-negation property).

The facts we deal with in this paper are assignments of r-symbols and QF-symbols to a CSP variable. The PS we consider consists of queries to the table and constraint processing, particularly local propagation of constraints. From the perspective of queries we additionally need to be able to aggregate the result sets into a normalized form, e.g. delete duplicate r-symbols, replace two QF-symbols by a more general one representing their union, etc. We have shown in Section 6 that the QF-symbols we primarily envision meet these requirements.

We can also turn the reasoning around and define a PO of symbols pertaining to the same property domain as a specialization relation if we can show that it has the above properties and if we also guarantee the following:

- There is a unique symbol $\perp$ (*false*) that is a special of all other symbols. This is a QF-symbol for the empty set.
- For any two symbols in the PO, there exists a unique symbol for a *greatest common successor/special* (the "*intersection*").
- For any two symbols in the PO, there exists a unique symbol for a *least common predecessor/general* (the "*normalized union*")[17].
- There is a unique top-level symbol Ω that represents the entire domain. For any symbol ϕ in the PO, there exists a symbol $\neg\phi$ in the PO, such that the least common predecessor of ϕ and $\neg\phi$ is Ω and the greatest common successor is $\perp$. $\neg\phi$ denotes the negation of ϕ.[18]

For example, we could arrange images in a PO and declare it a specialization relation, paying some attention to fulfill the requirements in the spirit of the intended PS.

Specialization relations provide some conceptual and practical benefits:

- They can be pre-calculated and stored in a graph. This may be more performant than calculating intersections and unions on the fly.
- They provide a general concept to adapt PS to QF-symbols: PS must simply be prepared to specialize the assignment of a QF-symbol to a CSP variable.
- They generalize to other objects, e.g. shapes, images, taxonomies, etc.

8.2 Local Propagation with QF-Symbols

If a product model contains multiple constraints, *local propagation* [4] can be used to restrict the domains of the product properties to a state of *arc consistency*. Any domain restriction of a product property is propagated to all constraints that reference the same product property. The process continues until no further restrictions are possible. For a tabular constraint, the query (2) can be used to determine the domain restrictions, which are then propagated (see Section 7).

The QF framework fits nicely in this scheme. A c-tuple comprised of finite sets of symbols that may include the normalized union of QF-symbols may be used as a query condition. The domain restrictions that result from the query (2) with this query condition may again contain the normalized union of QF-symbols. The c-tuple formed from the resulting domain restriction for each column can be smoothly propagated to other constraints.

If the product model is entirely made-up of tabular constraints, the local propagation of QF-symbols is covered by our approach. It is also straightforward to include constraints representing numeric linear (in)equalities when propagating real-valued intervals.[19] Extending the propagation of QF-symbols yet further is a topic of future work.

8.3 General Constraint Solving

Extending existing problem solvers to deal with QF-symbols, will require an analysis of the particular methods employed. However, a

[17] We had noted that the union of two QF-symbols need not itself be a QF-symbol. Here, however, it is an advantage to be able to have a least common predecessor/general representing the *union* as part of the PO. A more detailed treatment of specialization relations is deferred to a discussion using practical examples when they arise.

[18] We need "negation" primarily to ensure a "set-difference" operation to be able to split two symbols into a disjoint triple of symbols as in (4) in Section 5.

[19] This is implemented in the SAP product configurators ([6]).

main common idea is that constraint problem solving will make use of the concept of *specialization relations*. Instead of exploring the validity of a simple value assignment, an assignment of a variable to a QF-symbol can be specialized. When considering such an assignment and a constraint on the same variable, the greatest common special must be substituted in the assignment. A forced specialization to the empty set would invalidate an assignment. The solutions found by constraint solving may contain (specialized) QF-symbols, i.e. exhibit a degree of non-determinism that cannot be avoided and is to be expected.

9 Indeterminism in Variants and Sub-Variants

A product variant is classically defined as an r-tuple, a value assignment to each of its product properties, but this is neither ideal nor sufficient in practice. Some degree of indeterminism in a variant is needed when a variant is to be further specialized in a later business process (e.g., at the customer's site). For example, a pump may be sold with a connection that fits several different sizes of hoses. The end customer may have to make a manual adjustment for the particular hose they want to attach by cutting off a part of the provided connector. The pump being sold to the customer by the business is a variant of their MC "pump" product that allows further individualization at the customer's site.

The number of sub-variants can be infinite, e.g. for the frequency a radio receiver may be tuned to. As built, the property *Frequency* would be described by a list of real-valued intervals for possible reception bands, e.g. $\{[7.2, 7.45], [9.4, 9.9], [11.6, 12.1]\}$ (MHz).

Allowing a variant to be defined by a c-tuple solves the above problem. However, c-tuples that define variants must be distinguished from those that are merely the by-product of compression. This would be an open MC business topic.

10 Summary and Outlook

This work extends a common theme: that data in tabular form is not only natural for modeling variants, but also a natural, non-proprietary medium for communicating between interrelated business processes within an enterprise, as well as between enterprises. Compression of tables is essential in MC for letting variant tables scale with a growing number of choices, which production technology now enables [13]. C-tuples are a transparent yet powerful form of compression that is transparent and upon which non-proprietary exchange formats can be based. This is addressed in other work, e.g. [13]. Here, we propose to add to the expressivity of variant tables by presenting a *quasi-finite* (QF) framework that allows dealing with infinite sets of choices within the tabular paradigm.

We believe the QF framework presented here meets these expectations. The main idea is that problem solving will deal with the QF-symbols representing infinite sets via *specialization relations*. Instead of exploring the validity of a value assignment of a value (feature) to a variable (product property), the assignment to a QF-symbol can be specialized. A forced specialization to the empty set would invalidate an assignment.

The discussed techniques involving c-tuples and VDDs using QF-symbols has not yet been deployed in practice. The individual ingredients: c-tuples, VDDs, and the management of partial orders (POs) for specialization relations have all been applied with positive results. As already mentioned, the question of how to define practical normalization of predicates is open. However, we believe that the primary open issue is to verify that it actually meets the expectations of MC business. This also includes evaluating the need of integrating with other types of constraints, such as linear (in)equality constraints, and the business value of indeterminism in product variants.

ACKNOWLEDGEMENTS

I would like to thank my daughter Laura and the reviewers for their comments, which helped improve this paper considerably.

REFERENCES

[1] Jérôme Amilhastre, Hélène Fargier, Alexandre Niveau, and Cédric Pralet, 'Compiling csps: A complexity map of (non-deterministic) multivalued decision diagrams', *International Journal on Artificial Intelligence Tools*, **23**(4), (2014).

[2] Henrik Reif Andersen, Tarik Hadzic, John N. Hooker, and Peter Tiedemann, 'A constraint store based on multivalued decision diagrams', In Bessiere [5], pp. 118–132.

[3] Rüdiger Berndt, Peter Bazan, Kai-Steffen Jens Hielscher, Reinhard German, and Martin Lukasiewycz, 'Multi-valued decision diagrams for the verification of consistency in automotive product data', in *2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012*, eds., Antony Tang and Henry Muccini, pp. 189–192. IEEE, (2012).

[4] C. Bessiere, 'Constraint propagation', in *Handbook of Constraint Programming*, eds., F. Rossi, P. van Beek, and T. Walsh, chapter 3, Elsevier, (2006).

[5] Christian Bessiere, ed. *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*. Springer, 2007.

[6] U. Blumöhr, M. Münch, and M. Ukalovic, *Variant Configuration with SAP, second edition*, SAP Press, Galileo Press, 2012.

[7] A. Haag, 'Chapter 27 - Product Configuration in SAP: A Retrospective', in *Knowledge-Based Configuration*, eds., Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen, 319 – 337, Morgan Kaufmann, Boston, (2014).

[8] Albert Haag, 'Konzepte zur praktischen handhabbarkeit einer atms-basierten problemlösung', in *Das PLAKON-Buch, Ein Expertensystemkern für Planungs- und Konfigurierungsaufgaben in technischen Domänen*, eds., Roman Cunis, Andreas Günter, and Helmut Strecker, volume 266 of *Informatik-Fachberichte*, 212–237, Springer, (1991).

[9] Albert Haag, *The ATMS - an assumption based problem solving architecture utilizing specialization relations*, Ph.D. dissertation, Kaiserslautern University of Technology, Germany, 1995.

[10] Albert Haag, 'Column oriented compilation of variant tables', in *Proceedings of the 17th International Configuration Workshop, Vienna, Austria, September 10-11, 2015.*, eds., Juha Tiihonen, Andreas A. Falkner, and Tomas Axling, volume 1453 of *CEUR Workshop Proceedings*, pp. 89–96. CEUR-WS.org, (2015).

[11] Albert Haag, 'Managing variants of a personalized product', *Journal of Intelligent Information Systems*, 1–28, (2016).

[12] Albert Haag, 'Assessing the complexity expressed in a variant table', in *Proceedings of the 19th International Configuration Workshop, La Defense, France, September 14-15, 2017.*, pp. 20–27, (2017).

[13] Albert Haag and Laura Haag, 'Empowering the use of variant tables in mass customization', in *Proceedings of the MCP-CE 2018 conference, Novi Sad, Serbia, September 19-21, 2018.*, p. (Submitted), (2018).

[14] G. Katsirelos and T. Walsh, 'A compression algorithm for large arity extensional constraints', In Bessiere [5], pp. 379–393.