J Intell Inf Syst (2017) 49:59–86
DOI 10.1007/s10844-016-0432-5

Managing variants of a personalized product

Practical compression and fast evaluation of variant tables

Albert Haag[1]

Received: 15 November 2015 / Revised: 7 July 2016 / Accepted: 9 September 2016 /
Published online: 11 October 2016

Abstract A product *variant table* is a table that lists legal combinations of product features. Variant tables can be used to constrain the variability offered for a personalized product. The concept of such a table is easy to understand. Hence, variant tables are natural to use when ensuring the completeness and correctness of a quote/order for a customizable product. They are also used to filter out *inadmissible* choices for features in an interactive specification (configuration) process. Variant tables can be maintained as relational (database) tables, using spreadsheets, or in proprietary ways offered by the product modeling environment. Variant tables can become quite large. A way of compressing them is then sought that supports a space-efficient representation and a time-efficient evaluation. The motivation of this work is to develop a simple approach to *compress/compile* a variant table into an *easy to read, but possibly hard to write* form that can be deployed in a business setting at acceptable cost and risk in a similar manner as a database. The main result is a simple compression and evaluation scheme for an individual variant table called a *Variant Decomposition Diagram* (*VDD*). A VDD supports efficient consistency checks, the filtering of inadmissible features, and iteration over the table. A simple static heuristic for decomposition order is proposed that suggests itself from a "column oriented viewpoint". This heuristic is not always optimal, but it has the advantage of allowing fast compilation of a variant table into a VDD. Compression results for a publicly available model of a Renault Megane are given. With the proposed heuristic the VDD is a specialization of a *Zero-suppressed (binary) Decision*

I want to thank the reviewers and my daughter Laura for substantial constructive criticism of the original manuscript, all of which I have tried to incorporate here. Any remaining shortcomings are my own.
Disclaimer: While the motivation for this work lies in my past at SAP and experiences with its product configurators (Blumöhr et al. 2012; Haag 2014), all work on this article and its precursors (Haag 2015a, b) was performed privately after transition into partial retirement without a work obligation at SAP. The implementation is not related to SAP software in any way. It is neither endorsed by SAP nor does it reflect ongoing SAP development.

✉ Albert Haag
albert.haag@t-online.de
[1] Present address: Sonnenwendstrasse 50, 67098 Bad Dürkheim, Germany

Diagram (*ZDD*) (Knuth 2011) and also maps to a *Multi-valued Decision Diagram* (*MDD*) (Andersen et al. 2007; Berndt et al. 2012).

Keywords Variant tables · Arc consistency · Decision diagrams · Table compression · Column-oriented decomposition · VDD · ZDD · BDD · MDD

1 Introduction and motivation

Mass customization is about producing personalized variants of a product. These variants share a common basic structure, but with individually differing features. The individualized features are described by assigning a value (typically a string or number) to a product property. In natural language, the terms *feature* and *product property* are more or less synonymous and can refer either to a named observable such as *Color*, or to the individually instantiated manifestation "$Color = `Red'$". To distinguish this, the following terminology is used in the sequel: A product property name (an observable property) is referred to as a *characteristic*.[1] The term *feature* is used to designate an individual manifestation: the assignment of a *value* to a *characteristic*.

With this terminology, all variants of a product share the same characteristics but may differ in their individual features (the values assigned to the characteristics). It is natural to represent the set of variants in tabular form: Each column of the table corresponds to a characteristic. A row in the table represents a combination of features, i.e., it is a tuple of value assignments to the given characteristics. A table that lists legal combinations of product features will be called a (product) *variant table*.[2] The concept of a variant table is easy to understand. Hence, variant tables constitute a basic element in product modeling.

From a business perspective, variant tables are used as constraints to verify the validity of the chosen set of features in a quote/order for a personalized product. This also includes filtering out features that can no longer be consistently added to an emerging interactive specification, which will here be referred to as a *configuration*. Variant tables listing the variants actually sold are an important source for business analytics, which may require scanning the entire table.

In practice, common ways of maintaining a variant table are: in a database, in the form of a spreadsheet, or in proprietary ways offered by the business software. However, when variant tables become large, a way of compressing them is sought that supports a space-efficient representation and a time-efficient evaluation.

When using a database, one approach at this is *normalization*: A table is split into smaller chunks that capture independent relations between characteristics. Another approach is to switch to a column-oriented database (Stonebraker et al. 2005), which may offer a more compact representation and faster evaluation than relational databases.

When using a spreadsheet, the most obvious approach at space-efficiency is to identify Cartesian subsets $D_1 \times \ldots \times D_k$ of the table that can each be represented by one tuple of cells, here called a *c-tuple* as in Katsirelos and Walsh (2007). All values of each component set D_j would be listed in a single spreadsheet cell.

[1] This terminology is that used in SAP ERP (*Enterprise Resource Planning*) logistics.

[2] This terminology is that used in SAP Variant Configurator (SAP VC ((Blumöhr et al. 2012))). Variant tables are important elements of SAP VC product models.

Also, techniques to *compile* logical formulae into either an (ordered) *Binary Decision Diagram* (*BDD*) (Knuth 2011; Hadzic 2004) or an (ordered) *Multi-valued Decision Diagram* (MDD) (Andersen et al. 2007; Andersen et al. 2010; Berndt et al. 2012) can be applied to variant tables (seen as expressing a logical relation). These techniques can result in a compact representation that is very efficient to evaluate.

The motivation for this work was to look for a practical alternative to using a database or a spreadsheet for maintaining and evaluating large variant tables. The author's belief is that a solution that addresses variant tables directly and individually can be integrated and used in legacy environments (such as the SAP VC configurator) more easily and with less risk than shifting the entire implementation to a new technology.

The main result is a simple compression and evaluation scheme for an individual variant table. The underlying data structure is called a *Variant Decomposition Diagram* (VDD), because it represents a decomposition of the table into subtables. A VDD also yields a natural compression of a variant table into *c-tuples* allowing a compact representation in a spreadsheet. Besides the fact that the decomposition is based on identifying common occurrences of the same feature within a column of the table, there is no technical connection to column-oriented databases.

Structurally, a VDD is a form of *binary decision diagram* (Knuth 2011). The size of a VDD constructed for a given variant table will depend on the order of the decomposition decisions (this corresponds to an ordering of the product *features*). Another result of this work is a simple static heuristic for making decomposition decisions when constructing a VDD. This heuristic suggested itself from a "column oriented point of view" and is called the *(preferred) column heuristic* accordingly (see Section 5).

The column heuristic has so far yielded very acceptable compression and has the advantage of allowing fast compilation of variant tables (see Section 7). This makes it more attractive from a practical perspective than a slower approach with more optimal compression. Fast compilation is important when considering a VDD as an alternative to a database or a spreadsheet. It means that a VDD can be quickly regenerated from an underlying transparent representation after a change to the table content.

When using the *column heuristic*, the resulting VDD turns out as a specialization of a *Zero-suppressed (binary) Decision Diagram* (*ZDD*) (Knuth 2011) and also maps to an MDD (see Sections 4.4 and 6.4).[3]

Finding a good ordering of decision variables can be exponentially complex for decision diagrams, and heuristics that work are often application domain dependent (Andersen et al. 2010; Berndt 2016; Matthes et al. 2012; Hadzic et al. 2008). In some cases, no single standard agnostic search method for a good variable ordering is satisfactory (Matthes et al. 2012). This was also the author's experience with an initial attempt to compile a variant table into a BDD. Hence, the proposed simple static heuristic for constructing a VDD for a variant table is noteworthy in itself.

The VDD approach was developed in parallel with a Java implementation. This VDD implementation was applied to tables that resemble real customer data (SAP VC variant tables) in Haag (2015b). This both verified the functionality and substantiated the claim for space and time efficiency of the approach. The further results given here in Section 7 confirm this claim. The implementation has not yet been productively deployed.

[3]The structure of a VDD as a *binary decision diagram* and the relation to a ZDD and MDD motivates an alternate interpretation of the acronym VDD as *Variant binary Decision Diagram*.

Whereas the VDD approach would apply to tabular data in general, the restriction here to variant tables (and hence also the use of the more specific terminology) seems justified by the conjecture that variant tables may be particularly amenable to the compression techniques presented here, because their content will usually encode some regularity in the product variants they describe. Accordingly, the functionality currently implemented for a VDD is tailored to the mass customization/product configuration setting. A VDD supports efficient database-like queries for the filtering of inadmissible features (see Section 3) and consistency checks. It allows iteration over the table and over the result sets of queries.

Two examples are used for illustrative purposes. One is a T-shirt that is to be offered in a web shop on the internet. This is presented in Section 2 and illustrates the role that variant tables can play in a mass customization business. The other is used in Section 7 to evaluate VDD compression of variant tables. It is for a "real" product model of the Renault Megane that was made publicly available for performance testing in a constraint programming context. It is cited in Amilhastre et al. (2002) and is available at http://www.itu.dk/research/cla/externals/clib/.

Section 3 looks at the the central filtering function and (*arc*) *consistency*. The approach at VDD construction (table decomposition) and evaluation is treated in Section 4. The proposed simple static heuristic is discussed in Section 5. Section 6 handles a further reduction of a VDD using *set-labeled* nodes. This is used in constructing an external representation of variant table as Cartesian tuples (*c-tuples*), and a mapping to an MDD. Finally, Section 8 concludes with a summary, an outlook of future work, and open issues.

2 T-shirt example

The T-shirt example is not based on any actual web-shop. This allows both simplification and the exploration of ideas beyond what may actually be available today at a particular company's web site. However, the observations, particularly about the use of variant tables, are meant to be "real", and apply to mass customized products in general, as evidenced by the Renault Megane model for a car.

We take a T-shirt to be fully described by the product characteristics:[4]

$$Style \quad Fabric \quad Size \quad Color \quad Imprint \quad ImprintColor \quad Price \tag{1}$$

2.1 Very simple T-shirt customization

We assume that the business starts simple by selling a set of pre-defined T-shirt variants from stock. Only one style (*Standard*) and one fabric (*Cotton*) are offered for a standard price (9.99 USD).

The other characteristics have the following value domains:

- $\{Large, Medium, Small\}$ for *Size*,
- $\{Black, Blue, Red, White\}$ for *Color*,
- $\{MIB$(Men in Black), STW(Save the Whales)$\}$ for *Imprint*, and
- $\{White, Green\}$ for *ImprintColor*.[5]

[4]Characteristics that are freely specifiable, such as personal names, are outside the scope of variant tables and are omitted from the discussion here for the most part.

[5]We take *ImprintColor* to be *White* for *MIB*, and *Green* for *STW*.

Table 1 Variant table VT_SIMPLE_T_SHIRT for the simple T-shirt

Style	Fabric	Size	Color	Imprint	ImprintColor	Price(USD)
Standard	Cotton	Small	Black	MIB	White	9.99
Standard	Cotton	Medium	Black	MIB	White	9.99
Standard	Cotton	Large	Black	MIB	White	9.99
Standard	Cotton	Medium	Black	STW	Green	9.99
Standard	Cotton	Large	Black	STW	Green	9.99
Standard	Cotton	Medium	White	STW	Green	9.99
Standard	Cotton	Large	White	STW	Green	9.99
Standard	Cotton	Medium	Red	STW	Green	9.99
Standard	Cotton	Large	Red	STW	Green	9.99
Standard	Cotton	Medium	Blue	STW	Green	9.99
Standard	Cotton	Large	Blue	STW	Green	9.99

Not all combinations of *Size*, *Color*, and *Imprint* are possible due to constraints that state that for imprints MIB is only available for the color $Black$ and STW does not fit on $small$ shirts.The valid variants of this simple T-shirt are listed in Table 1.[6]

2.2 Extended T-shirt customization

We now imagine the following scenario for growing the business:

- The stock expands to three fabrics (*Cotton/Synthetic/Mixed*) and three styles (*No sleeve/Half sleeve/Full sleeve*). Cotton shirts are only available in *Half sleeve* and *Full sleeve*.
- The sizes *XS*, *XL*, and *XXL* are added. For toddlers, sizes *3T* and *4T* are added as well. Toddler shirts are only available in fabric *Cotton*.
- The business offers T-shirts in different colors. The colors *Black*, *Blue*, *Red*, and *White* are stocked. These shirts cost the standard price. T-shirts can be dyed in four other colors *Pink*, *Purple*, *Green*, and *Yellow*. These colors cost extra. Also, the dye used to obtain the colors depends on the fabric. For simplicity, let us assume that the price will only depend on the fabric of the T-shirt and the dye used. *Dye* is added to the list of characteristics (1), but is not shown in user interaction specifying a T-shirt.
- The possible predefined imprints are expanded and listed in a catalog. The association of *Imprint* and *ImprintColor* is encoded in the colored image of each catalog entry. Let us assume that there are 100 black imprints, 90 blue ones, 90 red ones, and 50 green ones. Generally, the color of an imprint must differ from the color of the T-shirt.
- The imprints *MIB* and *STW* from the previous section are discontinued.[7]

Tables 2, 3, 4, and 5 are examples of how variant tables relating to the above three items might be formulated in compressed form in a spreadsheet using multi-valued cells. The wildcard symbol '*' is used as an abbreviation to stand for "all possible values", which demonstrates an expressiveness desired in practice.

[6]The boxed subtable of Table 1 is structurally identical to the example used in Haag (2015a, b), which is an adaptation of an example from Andersen et al. (2010).

[7]This is for simplicity of exposition. so as not to clutter up the variant tables (Tables 2, 3, 4, and 5).

Table 2 Compressed relation of Style, Fabric, and Size

Style	Fabric	Size
{HalfSleeve, FullSleeve}	Cotton	*
*	{ mixed, synthetic}	{ XS, S, M, L, XL, XXL}

"*" is a wild card expressions that stands for the entire domain

A table explicitly listing all possible variants would be obtained by an *equi-join* operation ($\bowtie$) on the four tables (if they were expanded to relational form) using all matching column names (2).

$$\text{Overall Variant Table} = \text{Table 2} \bowtie \text{Table 3} \bowtie \text{Table 4} \bowtie \text{Table 5} \tag{2}$$

We assume that the business operates with a stock of "basic" T-shirts. The description of the stocked items may not require all the characteristics. For example, the basic shirts will not need the characteristics *Imprint*, *ImprintColor*, *Dye*, and *Price*. This might be a business argument for having several variant tables. Nevertheless, the distinction between stocked variants and products made-to-order is fluid, as any produced variant could be placed in stock when an order is returned. The business may be easier to operate with a single table (the left-hand side of (2)) where this is possible. However, it must also be able to deal with the case of multiple variant tables (the right-hand side of (2)). The 113 tables in the Renault Megane model (Section 7) are an example of such a model.

2.3 Scalability - configuring T-shirts

Our web shop must scale with demand. This entails automation as far as possible. Quotes/orders must be verified as being complete and correct. Complete means that all

Table 3 Compressed relation of Fabric, Color, and Dye

Fabric	Color	Dye
*	{Black, Blue, Red, White }	none
Cotton	Green	GRCD#1
{ Mixed, Synthetic}	Green	GRSD#2
Cotton	Purple	PUCD#3
{ Mixed, Synthetic}	Purple	PUSD#4
Cotton	Pink	PICD#5
{ Mixed, Synthetic}	Pink	PISD#6
Cotton	Yellow	YCD#7
{ Mixed, Synthetic}	Yellow	YSD#8

"*" is a wild card expressions that stands for the entire domain

Table 4 Compressed relation of Color, Imprint, and Dye

Color	Imprint	Dye
Black	*230 non-black imprints*	none
Blue	*240 non-blue imprints*	none
Red	*240 non-red imprints*	none
White	*All 330 imprints*	none
Green	*280 non-green imprints*	{ GRCD#1, GRSD#2 }
Pink	*All 330 imprints*	{ PICD#3, PISD#4 }
Purple	*All 330 imprints*	{ PUCD#5, PUSD#6 }
Yellow	*All 330 imprints*	{ YCD#7, YSD#8 }

characteristics of a T-Shirt are specified.[8] A check whether a complete given T-shirt configuration (specification) is consistent with (2) is easy, whether the product model in (2) is given by the joined left-hand side, or by the group of four tables involved in the join.

There are two approaches to restricting users to consistent choices during an interactive specification of their desired T-shirt.

In the first approach the user is guided in making choices in a way that ensures a complete and consistent result. For example, a catalog of stocked basic T-shirts is presented first. The choice of a basic T-shirt will already define many of the features (such as *Style*, *Size*, and *Fabric*). The user is then allowed to choose a color for the T-shirt. If the choice of a color depends on the already chosen base T-shirt, the application will only offer colors compatible with the chosen base shirt. If the choice of a color has other effects on other features, e.g., *Price*, then these effects are displayed in advance alongside the color. Finally, the user may choose the imprint from a catalog and is shown the resulting price for the personalized T-shirt.

In the alternate approach, the user is presented with the relevant characteristics listed above, perhaps filled with defaults matching their personal profile. For each characteristic the possible values are visible to the customer. The user can make choices or exclusions for any characteristic in any order. At any stage in this configuration process, the currently valid domains are always updated and displayed accordingly. In an advanced setting, the user could actually choose an inadmissible, grayed-out value. In this case, the user interface needs to be able to guide the user as to what choices need to be modified to enable this selection.[9]

Programming is needed in the first approach to implement the guiding procedure. In the second scenario, techniques are needed to establish the admissible choices in each state. A central mechanism for this (see Section 3) is the filtering function associated with a variant table.

This second approach is a form of *product configuration*, which has been a research and development topic of its own for quite a while (Felfernig et al. 2014). For the T-shirt example given here, the VDD approach and propagation to establish *arc consistency* suffice as a simple configurator (see Section 3.4).

[8] In the example there are no "optional" characteristics. A value must be specified for each one.

[9] This is one topic of Haag and Riemann (2011), albeit in the context of a software configuration.

Table 5 Compressed relation of Fabric, Dye, and Price

Fabric	Dye	Price
Cotton	none	10.99
{ Mixed, Synthetic }	none	9.99
Cotton	GRCD#1	17.99
{ Mixed, Synthetic }	GRSD#2	16.99
Cotton	PICD#3	19.99
{ Mixed, Synthetic }	PISD#4	18.99
Cotton	PUCD#5	17.99
{ Mixed, Synthetic }	PUSD#6	16.99
Cotton	YCD#7	15.99
{ Mixed, Synthetic }	YSD#8	14.99

3 Filtering function of a variant table

If the user has chosen the color *Red* for their T-shirt in the example in Section 2.1, then there are only two rows in Table 1 left to be considered. All other choices must be compatible with these two variants. The admissible choices can be *filtered* accordingly: *Size* is restricted (filtered) to the set $\{Medium, Large\}$, and *Imprint* is restricted to the singleton set $\{STW\}$.

3.1 Basic definition of a filtering function

The number of columns of the variant table, k, is referred to as its *arity*. Let $v_1 \ldots v_k$ denote the column characteristics of a variant table. Let a finite domain D_j be given for each v_j. Then $\mathbf{D} = D_1 \times \ldots \times D_k$ is the *solution space*, the set of all conceivable variants. Cartesian sets like $\mathbf{D}$ are here referred to as *cuboids* and denoted by boldface capital letters.

An arbitrary subset $\mathfrak{X} \subseteq \mathbf{D}$ consists of a set of tuples $(x_1 \ldots x_k)$ that can be interpreted as the rows of a table. Thus the overall set of valid variants can always be seen as a table, although it may not be feasible or useful to explicitly represent it as such if it is very large. The term *variant table* will be taken to refer to subsets of the solution space that are intended to be explicitly formulated and maintained as a table. A variant table will be denoted by $\mathfrak{T}$. In general, sets $\mathfrak{X} \subseteq \mathbf{D}$ that are not Cartesian will be denoted by *fraktur* capital letters.

$\mathfrak{X} \subseteq \mathbf{D}$ can also be represented as an $r \times k$ matrix/array $\left(x_{ij}\right)$, where r denotes the number of tuples (relational table rows), i is the index of the tuple (row), and j is the index of the column. I.e., the value x_{ij} is for the characteristic v_j.

With these definitions, the projection of $\mathfrak{X}$ onto each v_j is defined by (3):

$$\pi_j(\mathfrak{X}) := \bigcup_{i=1}^{r} \{x_{ij} \in \mathfrak{X}\} \tag{3}$$

The smallest encompassing cuboid of $\mathfrak{X}$ is made up of all its column projections

$$\pi(\mathfrak{X}) := \pi_1(\mathfrak{X}) \times \ldots \times \pi_k(\mathfrak{X}) \tag{4}$$

If only one variant table $\mathfrak{T}$ is being considered, we can take $D_j = \pi_j(\mathfrak{T})$, i.e., the global domains are just made up of the values that occur in $\mathfrak{T}$, because any feature not referenced in $\mathfrak{T}$ cannot occur in a product variant.

Let a domain restriction $R_j \subseteq D_j$ be given for each v_j. Then

$$\mathbf{R} = R_1 \times \ldots \times R_k \tag{5}$$

is called the resulting overall domain restriction.

Given $\mathfrak{X} \subseteq \mathbf{D}$ and $\mathbf{R}$, only the tuples in $\mathfrak{X} \cap \mathbf{R}$ are still valid combinations. Accordingly, for each column v_j only the values in $\pi_j(\mathfrak{X} \cap \mathbf{R})$ are still *admissible* (part of any valid solution). The filtering function (6) of $\mathfrak{X}$, denoted by $f_{\mathfrak{X}}$, is defined as

$$\begin{aligned} f_{\mathfrak{X}} : 2^{\mathbf{D}} &\to 2^{\mathbf{D}} \\ \mathbf{R} &\mapsto \pi(\mathfrak{X} \cap \mathbf{R}) \end{aligned} \tag{6}$$

$f_{\mathfrak{X}}$ restricts (filters) $\mathbf{R}_j$ to $\pi_j(\mathfrak{X} \cap \mathbf{R}) =: \widehat{R}_j$.

3.2 Filtering using a database

As pointed out, if the user has chosen the color *Red* for their T-shirt in the example in Section 2.1, then the there are only two rows in Table 1 left to be considered. These two rows can be determined with the database SQL query

```
SELECT * FROM VT_SIMPLE_T_SHIRT WHERE Color = 'Red';
```

The required projections can also be obtained directly with SQL means:

```
SELECT DISTINCT Size FROM VT_SIMPLE_T_SHIRT WHERE Color = 'Red';
SELECT DISTINCT Imprint FROM VT_SIMPLE_T_SHIRT WHERE Color = 'Red';
```

Somewhat abusing formal notation, the filtering function $f_{\mathfrak{T}}$ of a variant table $\mathfrak{T}$ could be formulated as k SQL queries as in (7).

$$\texttt{SELECT DISTINCT } \langle v_j \rangle \texttt{ FROM } \langle \mathfrak{T} \rangle \texttt{ WHERE } \langle v_1 \rangle \texttt{ IN } \langle R_1 \rangle \texttt{ AND } \ldots \langle v_k \rangle \texttt{ IN } \langle R_k \rangle; \tag{7}$$

The queries (7) are "idealized", because, whereas database systems do well with very large tables, it may not be practical to formulate an actual SQL query as in (7) due to its potential complexity. For this reason, dedicated algorithms operating on an in memory representations of a table are mostly used for implementing the filtering function of a variant table. The STR algorithm (Lecoutre 2011) is an example of such a special algorithm.

3.3 Filtering using binary decision diagrams

From a logical perspective, each possible feature (assignment of a value x to a characteristic v_j) can be seen as a logical proposition, denoted by $p(v_j, x)$. $p(v_j, x)$ is considered as *false* if the feature is not present in the variant being considered and *true* otherwise. Thus, each proposition $p(v_j, x)$ is naturally associated with a Boolean variable, which, for simplicity, we may also denote by $p(v_j, x)$.

There are $s := \sum_{j=1}^{k} |D_j|$ distinct features referenced in the product model. Hence, given any truth assignment to these s features, a variant table $\mathfrak{T}$, seen as a logical expression, will evaluate to 1 ($\top$/*true*) if a row exists with all its propositions being *true*. Otherwise, it evaluates to 0 ($\bot$/*false*). In this sense, $\mathfrak{T}$ defines a Boolean function:

$$\mathcal{F}_{\mathfrak{T}} : 2^s \to \{0, 1\} \tag{8}$$

$\mathcal{F}_{\mathfrak{T}}$ can be represented as a binary decision diagram in various ways (Knuth 2011), which promise a compact representation. When such a representation is found, it offers a means for fast evaluation of the filtering function.

The size of such a diagram depends on the order in which the Boolean variables are considered. The complexity of finding a good variable ordering is high, and heuristics are employed in practice. Thus construction of a compact decision diagram is time consuming and must be performed in advance of any evaluation. Accordingly, such an approach is referred to as a *compilation approach*. The form of a binary decisions diagram considered here is a VDD, see Section 4.

3.4 Arc consistency

If there is more than one variant table in the product model (as in the Renault model in Section 7), these tables could be joined as in (2). This joined table would explicitly list all valid variants. If it is not opportune to calculate this join, then an approximation of the filtering function of the overall joined table can be obtained via *constraint propagation* among the individual variant tables. Any reduction of a column domain achieved in filtering with respect to a given variant table must be applied to all other tables that reference the same characteristic to see if further filtering can occur. When constraint propagation has reached a state where no further filtering can occur, this is called *arc consistency*. A treatment of arc consistency is given in Bessiere (2006).

Arc consistency is a weaker concept than explicitly calculating all valid variants. However, practical experience with product configurators has established that it is fully adequate in practical applications in the overwhelming majority of cases.[10]

4 Variant table decomposition

Let $\mathfrak{T}$ be a variant table of arity k with column characteristics $v_1 \ldots v_k$. Let $D_j := \pi_j(\mathfrak{T})$ be a finite domain for v_j (the set of values occurring in the j-th column of $\mathfrak{T}$).

4.1 Column oriented decomposition

The basic idea to decompose $\mathfrak{T}$ is very simple. Choose a column characteristic v_j and a value $x \in D_j$. Then $\mathfrak{T}$ can be decomposed into three parts:

- $B(\mathfrak{T}, v_j, x)$: the collection of cells in the column for v_j that contain the value x,
- $\mathfrak{L}(\mathfrak{T}, v_j, x)$: the rows in $\mathfrak{T}$ that don't have value x in column j, and
- $\mathfrak{R}(\mathfrak{T}, v_j, x)$: what remains after removing $B(\mathfrak{T}, v_j, x)$ and $\mathfrak{L}(\mathfrak{T}, v_j, x)$ from $\mathfrak{T}$.

Table 6 is a copy of Table 1 with the three parts identified as follows:

- The cells in $B(VT\_SIMPLE\_T\_SHIRT, v_5, MIB)$ are boxed.
- The cells in $\mathfrak{L}(VT\_SIMPLE\_T\_SHIRT, v_5, MIB)$ are displayed *slanted.*
- The cells in $\mathfrak{R}(VT\_SIMPLE\_T\_SHIRT, v_5, MIB)$ are in **boldface**.

Table 6 is sorted in a way that the cells in $B(VT\_SIMPLE\_T\_SHIRT, v_5, MIB)$ form a contiguous block of values (boxed). Regardless of the sorting of a variant table, $B(\mathfrak{T}, v_j, x)$

[10] Of course, it is possible to construct counter examples.

Table 6 Decomposition of VT_SIMPLE_T_SHIRT for *MIB*

Style	Fabric	Size	Color	Imprint	ImprintColor	Price(USD)
Standard	**Cotton**	**Small**	**Black**	MIB	**White**	**9.99**
Standard	**Cotton**	**Medium**	**Black**	MIB	**White**	**9.99**
Standard	**Cotton**	**Large**	**Black**	MIB	**White**	**9.99**
Standard	*Cotton*	*Medium*	*Black*	*STW*	*Green*	*9.99*
Standard	*Cotton*	*Large*	*Black*	*STW*	*Green*	*9.99*
Standard	*Cotton*	*Medium*	*White*	*STW*	*Green*	*9.99*
Standard	*Cotton*	*Large*	*White*	*STW*	*Green*	*9.99*
Standard	*Cotton*	*Medium*	*Red*	*STW*	*Green*	*9.99*
Standard	*Cotton*	*Large*	*Red*	*STW*	*Green*	*9.99*
Standard	*Cotton*	*Medium*	*Blue*	*STW*	*Green*	*9.99*
Standard	*Cotton*	*Large*	*Blue*	*STW*	*Green*	*9.99*

will be referred to as the *value block* for v_j and x. $\mathfrak{L}(\mathfrak{T}, v_j, x)$ is called the *left subtable* of $\mathfrak{T}$ for v_j and x, and $\mathfrak{R}(\mathfrak{T}, v_j, x)$ is called the *right subtable* of $\mathfrak{T}$ for v_j and x.[11]

$\mathfrak{L}(\mathfrak{T}, v_j, x)$ and $\mathfrak{R}(\mathfrak{T}, v_j, x)$ can be decomposed in turn. The decomposition process can be continued until only empty subtables remain. The question, which feature (column and value) to decompose on at each non-empty subtable, will depend on a suitable heuristic (see Section 5).

Each value block $B(\mathfrak{T}, v_j, x)$ represents all occurrences of the feature $(v_j = x)$ in $\mathfrak{T}$ and, as already noted, corresponds to a logical proposition $p(v_j, x)$ (see Section 3.3). If $p(v_j, x)$ is *true*, then the evaluation of the Boolean function for $\mathfrak{T}$ (8) depends only on the further evaluation of a sub-function for $\mathfrak{R}(\mathfrak{T}, v_j, x)$. If it is *false*, then it depends only on the further evaluation of $\mathfrak{L}(\mathfrak{T}, v_j, x)$.

In the sequel $p(v_j, x)$ will be referred to as a *feature*, even where it functions as a *proposition* or *Boolean variable*.

4.2 Variant (Binary) Decision Diagram - VDD

The recursive decomposition of $\mathfrak{T}$ leads to a decomposition tree. The tree's root represents the entire table. It is labeled by the value block $B(\mathfrak{T}, v_{j_1}, x_1)$ for the feature used to start the decomposition. There are two branches from this root. One, termed the *left child*, represents the left subtable $\mathfrak{L}(\mathfrak{T}, v_{j_1}, x_1)$. The other, termed the *right child*, represents the right subtable $\mathfrak{R}(\mathfrak{T}, v_{j_1}, x_1)$.

Each of these children can in turn have children if it can be decomposed further. An empty left child is labeled by a special symbol $\bot$ (*false*). An empty right child is labeled by a special symbol $\top$ (*true*). This can be reasoned as follows. If a (sub)table has only one column, then its right subtable is always empty, regardless of which feature is chosen for further decomposition. The empty right subtable must evaluate to *true* (denoted by the special symbol $\top$), since the (sub)table must evaluate to *true* when the chosen feature holds. If a (sub)table consists of only one value block then its right and left subtables are empty. The

[11] Observe that $\mathfrak{R}(\mathfrak{T}, v_j, x)$ is a sub-space − not a subset − of the overall solution space $\mathbf{D}$, i.e.,

$$\mathfrak{R}(\mathfrak{T}, v_j, x) \subseteq D_1 \times \ldots D_{j-1} \times D_{j+1} \times \ldots \times D_k \not\subset D_1 \times \ldots \times D_k$$

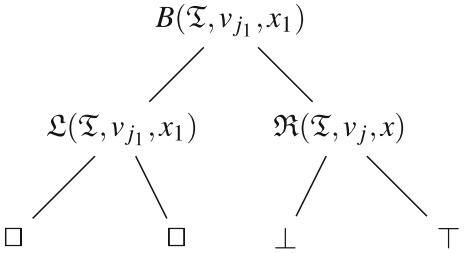


Fig. 1 Basic scheme of a decomposition

left subtable must evaluate to *false* (denoted by the special symbol $\perp$), since the (sub)table must evaluate to *false* when the (only) feature in the (sub)table does not hold.

Figure 1 is a depiction of this. The further decomposition of $\mathfrak{R}(\mathfrak{T}, v_j, x)$ is shown as two empty children. This means that it represents a table that consists of only one value block as discussed above.

Identical subtables may arise at different points in the decomposition tree. A goal of minimal representation is to represent these multiple occurrences only once by transforming the decomposition tree into a *binary rooted directed acyclic graph*, which is called a *VDD*. VDD stands for *Variant Decomposition Diagram*. It might also stand for *Variant Decision Diagram*.[12] All leaves can be identified with one of two predefined nodes also labeled $\perp$ and $\top$. Subsequently, all nodes labeled by the same feature $p(v_j, x)$ that have identical children are represented by re-using one shared node.[13]

By conventions following (Knuth 2011), a link to the left child is called a *LO*-link, drawn with a dotted line, preferably to the left, and a link to the right child is called a *HI*-link, drawn with a filled line, preferably to the *right*. A *LO*-link is followed when the feature in the node label is disbelieved/*false*. A *HI*-link is followed when it is believed/*true*. The terminal nodes $\perp$ and $\top$ are called *sinks*.

Figure 2 shows the VDD obtained for Table 1 using the *preferred column heuristic* proposed in Section 5. There are two nodes with multiple parents. Both of these refer to $Color =$ '*Black*' (v_4). These are the nodes with re-use.

A VDD node labeled $p(v_j, x)$ will itself be denoted as

$$\nu(v_j, x) \tag{9}$$

4.3 Evaluation of a VDD

Given a VDD $\mathfrak{V}$, a node $\nu(v_j, x)$ in $\mathfrak{V}$ as defined in (9), and an overall domain restriction $\mathbf{R}$ as defined in (5), $\nu(v_j, x)$ can be marked as *out* if $x \notin R_j$. This requires the implementation of an efficient membership test in the implementation.

A path in $\mathfrak{V}$ from the root node ν_A to the sink $\top$ is *admissible* if it does not contain a node marked *out*. $\mathfrak{V} \sqcap \mathbf{R}$ denotes the set of paths in $\mathfrak{V}$ that are admissible under a given restriction $\mathbf{R}$. If there are no such paths, then $\mathbf{R}$ is inconsistent given $\mathfrak{V}$.

[12] The term *Variant Decomposition Diagram* stresses the central column oriented decomposition. The term *Variant Decision Diagram* stresses the fact that it is a form of binary decision diagram (although not a BDD (Knuth 2011)).

[13] The VDD implementation identifies re-use of nodes on the fly during VDD construction. *Algorithm R* in (Knuth 2011) gives an explicit specification in the context of a BDD.

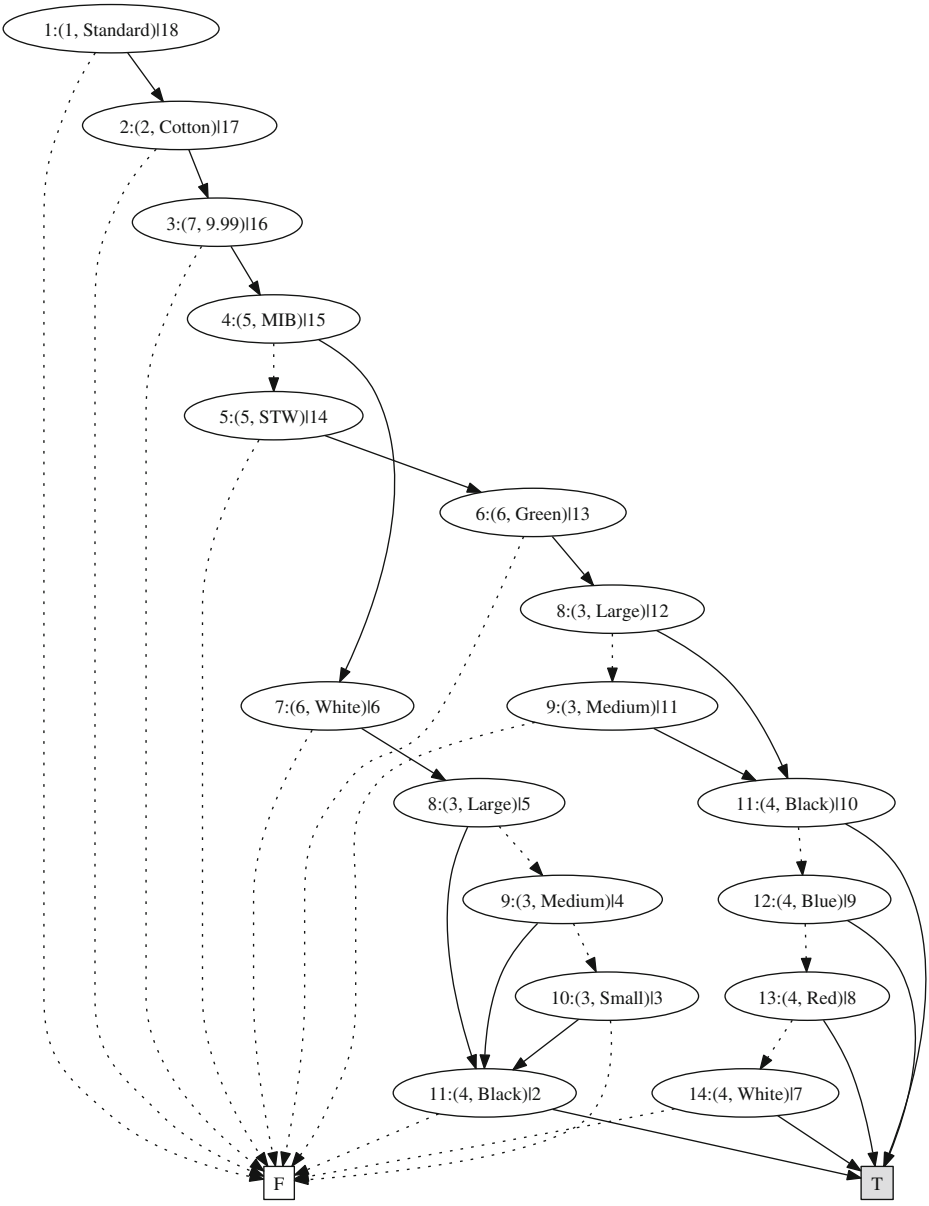


Fig. 2 VDD of table VT_SIMPLE_T_SHIRT using the preferred column heuristic (see Section 5)

The complexity of the evaluation can be bounded by the following two observations:

– Let n_j be the number of distinct nodes in $\mathfrak{V}$ that pertain to the j-th column of $\mathfrak{T}$. Then n_j membership tests in R_j must be performed during the evaluation of $\mathfrak{V}$ for the j-th column to determine nodes that can be marked as *out*. This must be done for any column that is properly restricted, i.e., where $\pi_j(\mathfrak{T}) \not\subset R_j$.

– An *admissible node* is a node that lies on an admissible path. Let n be the overall number of nodes in $\mathfrak{V}$. The question of which nodes are admissible is related to the problem of counting the admissible paths.[14] After determining which nodes are *out*, this has the remaining complexity of $O(n)$ (c.f., *Algorithm C* in Knuth (2011)).

Each admissible path in $\mathfrak{V}$ corresponds to an admissible value tuple in $\mathfrak{T}$. Hence, $\mathfrak{V} \sqcap \mathbf{R}$ directly corresponds to $\mathfrak{T} \cap \mathbf{R}$.

Having found all admissible nodes, it remains to actually construct the resulting domain restrictions $\widehat{R}_j$. This entails forming the union of all node labels of admissible nodes for the j-th column. From a database perspective, we can interpret $\widehat{R_j}$ as the result set of the conceptual SQL query (7). An iterator over the admissible paths can be implemented by starting from a first admissible path and then finding further admissible paths via backtracking over the admissible nodes.

4.4 Zero-suppressed (binary) Decision Diagram - ZDD

A ZDD is a cousin of a BDD (Knuth 2011). Both have the same data structure. The interpretation of a ZDD node is different from that of a BDD node, but both depend on a total ordering of the decision variables. In a BDD, if a node for the p-th Boolean variable in this ordering is directly linked to another node for the q-th variable, any variable that falls between these two is treated as being indifferent (*true* or *false*) in that path. In a ZDD any such variable is treated as being *false*.

A VDD also shares the same basic data structure as a BDD and ZDD. But, as the column oriented decomposition process described in Section 4.1 allows complete freedom of choosing the next feature to further decompose on at any given point, a VDD does not depend on an underlying ordering of the features. In a VDD, any feature not in the subtable represented by a node is treated as *false* in the subgraph below it.

If we totally order the set of features in some fashion and require that the decomposition respects this ordering, i.e., we always choose the first feature in this order that is applicable in a given subtable, then the resulting VDD is also a ZDD. To see this: Given two linked nodes $v(v_{j_1}, x_1)$ and $v(v_{j_2}, x_2)$ (i.e., the *HI*-link of $v(v_{j_1}, x_1)$ points to $v(v_{j_2}, x_2)$), let v_{j_3} be any variable that falls between v_{j_1} and v_{j_2} in the given variable ordering. Then $\forall x \in D_{j_3}$: $p(v_{j_3}, x) = false$ on any path starting at $v(v_{j_1}, x_1)$, because if a node $v(v_{j_3}, x^*)$ occurs on a path after $v(v_{j_1}, x_1)$ it would have to have been placed before $v(v_{j_2}, x_2)$.

Heuristics that do not follow an overall ordering of the features have been successfully tried in the VDD implementation, which does not assume that a VDD is a ZDD. However, the conceptual advantage of dealing with a ZDD is that all results and algorithms known for ZDDs apply (Knuth 2011; Mishchenko 2001).

The currently favored *preferred column heuristic* in Section 5 is based an an ordering of the features. Figure 2 depicts a VDD for the simple T-shirt (Table 1) that is also a ZDD.[15]

[14] Counting will determine whether a node has paths without nodes marked *out* to $\top$. Such a node is admissible if at least one of its parents is admissible.

[15] Each node in Fig. 2 is labeled in the form $\langle n : (j, val)\rangle|m$. The labels are artifacts of the current VDD implementation. (j, val) designates the feature $p(v_j, x)$, n is the ordinal giving the position of this feature in the overall total ordering of the features, and m is the node number assigned during construction (a unique identifier − not necessarily in the natural order).

There are 13 features in Table 1. Their order is

$$p(v_1, Standard),\ p(v_2, Cotton),\ p(v_7, 9.99),\ p(v_5, MIB),$$
$$p(v_5, STW),\ p(v_6, Green),\ p(v_6, White),\ p(v_3, Large),\ p(v_3, Medium)$$
$$p(v_3, Small),\ p(v_4, Black),\ p(v_4, Blue),\ p(v_4, Red),\ p(v_4, White)$$

5 Heuristics

The size of a VDD obtained for a table $\mathfrak{T}$ depends on which feature is chosen for decomposition at each step. Let S be the set of features occurring in $\mathfrak{T}$. If $|S| = s$, then there are $s!$ (s factorial) ways of ordering S. If we base the decomposition decisions on this ordering, each different ordering may lead to a differently sized VDD. There are even more ways of constructing a VDD if we do not follow such a global ordering (and are outside the realm of a ZDD).

The usual way of finding a good ordering is to start from a good guess and then improve on this by exploring local perturbations of the ordering. If the initial guess is not already good, this exploration can be expensive in terms of computational resources (both time and space). The heuristic proposed in Algorithm 1 is the "good guess". So far it seems surprisingly satisfactory without further tweaking.

Algorithm 1 *(Preferred Column Heuristic)*

1. *Sort the k columns of* $\mathfrak{T}$ *by* $|D_j|$ *ascending (largest number of values last)*
2. *Make the root node of the VDD choosing the first value in the first column for decomposition*
3. *While nodes with a non-terminal subtable remain: Always choose the first value in their first column for decomposition*

A *column heuristic* is a generalization of the *preferred column heuristic* that substitutes any given sorting of the columns in the first step. In particular, the natural order of the columns implicit in the definition of $\mathfrak{T}$ may be used. Note that the size of a VDD constructed using a *column heuristic* does not depend on the ordering of the features in D_j. To see this, note that the value block used in decomposition *slices* $\mathfrak{T}$ horizontally.[16] The slices obtained overall with respect to all values in the first column are the same, regardless of the order of the values in a column. In Table 6 there are two "slices" for column four (values *MIB* and *STW*). Transposing them does not affect their children (the respective subtables).

6 Compression, merged VDD nodes, and MDDs

6.1 Set-labeled VDD nodes

Given a solution space $\mathbf{D}$ and a variant table $\mathfrak{T} \subseteq \mathbf{D}$, a Cartesian subset (*cuboid*) $\mathbf{C} \subseteq \mathbf{D}$ will be more specially referred to as a *c-tuple* if $\mathbf{C} \subseteq \mathfrak{T}$. $\mathfrak{T}$ can be maintained in a compressed form in a spreadsheet using disjoint c-tuples. Conversely, possibilities of finding a

[16] The terminology of horizontal slicing suggests itself, but is also inspired by Gharbi et al. (2014).

compression of $\mathfrak{T}$ into disjoint c-tuples is sought, i.e., for exporting the table externally to a spreadsheet. A heuristic approach to this is presented in Katsirelos and Walsh (2007).

The VDD implementation optionally allows compressing a VDD that was constructed using a *column heuristic* (see Section 5) by merging several nodes into one set-labeled node. Let $\mathfrak{V}$ be a VDD constructed with a *column heuristic*. Let $v^*(v_j, X)$ be a node labeled with the set $X := \{x_1 \ldots x_h\} \subseteq D_j$. Then $v^*(v_j, X)$ represents the disjunction $p(v_j, x_1) \vee \ldots \vee p(v_j, x_h)$. A tuple of nodes in a path from the root node of a VDD to the sink $\top$ represents a c-tuple (not a value tuple) if it contains set-labeled nodes.

Note that a VDD constructed with a column heuristic has the following properties not necessarily applicable to a VDD in general:

- All nodes linked via *LO*-links will always pertain to the same characteristic v_j. This follows from the fact that the columns of any non-empty left subtable of a table $\mathfrak{T}$ are the same as the columns of $\mathfrak{T}$. The heuristic says to always choose from the first column.
- A node $v(v_j, x)$ is always $(j - 1)$ *HI*-links distant from the root node. This follows by iterating the argument that if a table $\mathfrak{T}$ is decomposed by a feature $p(v_1, x)$ referencing its first column, then its right child will be from the right subtable $\mathfrak{R}(\mathfrak{T}, v_1, x)$, which has the second column of $\mathfrak{T}$ as its first column by construction.

The subgraph composed of a node and all its direct descendants that can be reached from the node following only *LO*-links is defined as the *l-chain* of the node. All nodes in an *l-chain* will pertain to the same v_j. A set of nodes in an *l-chain* that all have the same right child (*HI*-link) represents the disjunction of the tuples for the admissible paths though the nodes in the set.

Figure 3 is a graph of the T-shirt using set-labeled nodes (the top three nodes for *Standard* (*Style*), *Cotton* (*Fabric*), and *9.99* (*Price*) have been omitted for space reasons).[17] It shows a reduction by merging nodes of the VDD in Fig. 2. Each of the three nodes that are sets correspond to an *l-chain* in Fig. 2:

- The nodes numbered 11 and 12 (*Medium* and *Large*) in Fig. 2 − with the common right child node number 10 (*Black*) − can be merged into one node (with node number 16 in Fig. 3).
- The nodes numbered 3, 4 and 5 (*Small*, *Medium*, and *Large*) in Fig. 2 − with the common right child node 2 (*Black*) − can be merged into one node (with node number 3 in Fig. 3).
- The nodes numbered 7, 8, 9 and 10 (*White*, *Red*, *Blue* and *Black*) in Fig. 2 − with the common right child $\top$ − can be merged into one node (with node number 5 in Fig. 3).

After merging nodes, a VDD constructed with the *preferred column heuristic* yields comparable compression to c-tuples for the simple examples given in Katsirelos and Walsh (2007). It would be interesting to see if this generalizes to more complex tables. However, there is a key difference in the measurement of compression in the two approaches. In Katsirelos and Walsh (2007) the goal is to find a minimal number of c-tuples. In constructing a VDD, the goal is always to minimize the number of nodes. This is not the same objective, as VDD paths may share nodes (which would correspond to c-tuples sharing common

[17] A merged set-labeled node is assigned an ordinal number ("variable number") n in the depicted label $(\langle n : (j, val)\rangle | m)$ outside the range used for numbering the features.

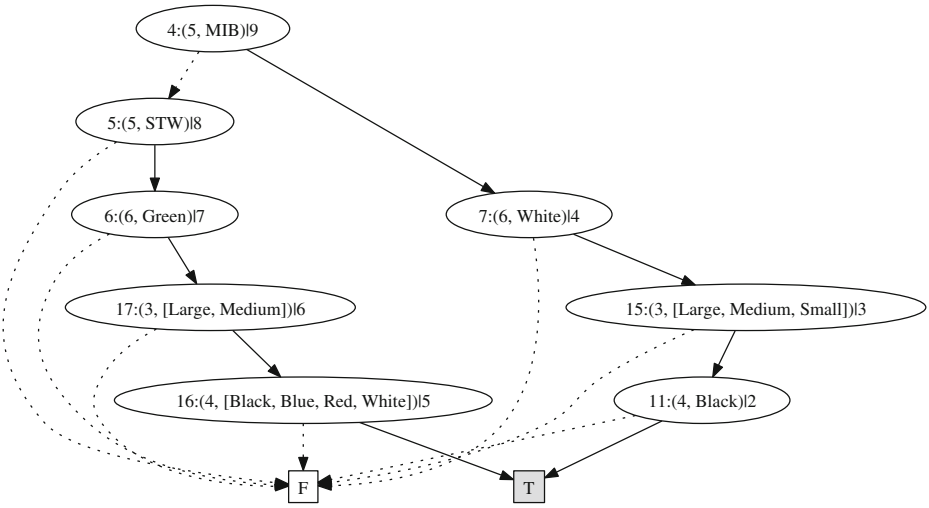


Fig. 3 VDD of simple T-shirt from Fig. 2 with set-labeled nodes

tails). For example, the compression of a variant table containing a column with a unique variant identification number to c-tuples will have a c-tuple for every variant, but it can be compressed using a VDD when there are shared tails.

The VDD implementation provides iteration in two stages for VDDs with set-valued nodes: an iteration over the admissible c-tuples and then an iteration over the c-tuple. Iteration over the c-tuples allows generating an external representation of $\mathfrak{T}$ in the form of c-tuples.

6.2 Evaluation of a VDD with set-Labeled nodes

As in Section 4.3, let $\mathfrak{V}$ be a VDD and $\mathbf{R}$ be an overall domain restriction as defined in (5). If $\mathfrak{V}$ has set-labeled nodes, this requires a different evaluation in the following two respects.

- For a set-labeled node $v^*(v_j, X)$ in $\mathfrak{V}$. $v^*(v_j, X)$ is marked as *out* if $X \cap R_j = \emptyset$. This requires the implementation of an intersects test in the implementation (instead of the membership test $x \in R_j$). The intersects test can be implemented efficiently if D_j is sorted, but it is more costly than the membership test.
- In a VDD $\mathfrak{V}$ with set-labeled nodes an admissible path corresponds to an *admissible c-tuple*, not an *admissible value tuple*. But, not all value tuples in an admissible c-tuple need to be in $\mathbf{R}$. To see this, suppose that the variant table $\mathfrak{T} \subset \mathbf{D}$ compresses to a single c-tuple $\mathbf{C} = \mathfrak{T}$. If $\mathbf{R} \cap \mathfrak{T} \neq \emptyset$, then $\mathbf{C}$ is an admissible c-tuple. But, if $\mathbf{R} \not\subset \mathfrak{T}$, there are value tuples in the admissible c-tuple $\mathbf{C}$ that are not in $\mathbf{R}$. In this case the result set (7) is obtained by additionally intersecting each admissible c-tuple $\mathbf{C}$ with $\mathbf{R}$.

The number of nodes may be significantly reduced by merging to set-labeled nodes. However, whether this will yield a performance gain in evaluation over the unmerged VDD is an open question (see Sections 7.3 and 7.4).

6.3 Continuous intervals and unbounded domains in VDD node labels

In business practice, it may be required to allow continuous real-valued intervals to occur in both the restrictions R_j and (non-relational) variant table cells.[18] The VDD implementation allows and handles this by providing for VDD nodes $v^*(v_j, I)$ labeled with continuous intervals.

During the variant table decomposition, when constructing a VDD, an interval is treated as an atomic unit like a value, and the interval is used to label the node. Merging of nodes extends naturally to interval nodes. The label of a merged node is the set union of the node labels of the nodes that were merged. A union of two intervals may result in a set represented by two intervals, or it may be that the intervals can be merged into one. So the labels of nodes must allow a general form expressing a union of several intervals.

During evaluation of a VDD, a node that references one or more intervals in its label is treated like other set-valued nodes by applying an intersects test. Iteration over the VDD again yields c-tuples, but these may now contain components containing continuous intervals. It is not meaningful to further iterate over a continuous interval itself, or over an infinite set in general.

A *"wild card"* ("*") expression referring to a non-numeric domain is another useful construct. Table 2 is an example how a wild card expression could be used externally as a syntactic short-cut to listing the entire global domain. However, a wild card expression could also be used for an *unbounded* domain, denoted here by Ω. Any intersection $R_j \cap \Omega$ yields R_j. An example of a characteristic with an unbounded domain for a T-shirt would be *ImprintText*, which would allow specifying "free text". Numeric intervals may also be unbounded or partially unbounded, i.e., $-\infty$ and $+\infty$ are legal interval bounds.

6.4 Multi-valued Decision Diagrams (MDDs)

One currently favored way of compiling the constraints in a product model is to an (ordered) *Multi-valued Decision Diagram* (*MDD*) (Andersen et al. 2007; Hadzic et al. 2008; Berndt 2016; Berndt et al. 2012). An MDD is a *non-binary rooted directed acyclic graph*. Each MDD node is labeled by a non-Boolean *variable* mv_j from an ordered set $\{mv_1 \ldots mv_p\}$. An MDD node can have several outgoing links. A link from a node for variable mv_j always leads to a node for mv_{j+1}, or to the sink $\top$ (see Andersen et al. (2007)).

If a variant table $\mathfrak{T}$ can be compiled in several different ways, it is obvious that these underlying representations for $\mathfrak{T}$ can be mapped to one another. For a VDD compiled with a column heuristic and merging to set-labeled nodes, there is a direct mapping to an MDD: Let the MDD variables mv_j correspond to the k characteristics v_j. Map each VDD node $v^*(v_j, X)$ to an MDD node for mv_j with an outgoing link for each element $x_i \in X$ to a node for mv_{j+1}. Figure 4 is an MDD obtained from the VDD depicted in Fig. 3 with set-labeled nodes. Conversely, it is also possible to map an MDD representing a table to a VDD by inverting the above mapping.[19]

Different approaches suggest different heuristics and different evaluations. The simple column oriented heuristic may not suggest itself when thinking in terms of MDDs. Indeed,

[18] A continuous interval states that any feature with a value from the interval is considered as feasible (an infinite disjunction).

[19] In Section 3.1 it was observed that any subset $\mathfrak{X}$ of the solution set **D** can be seen as a "table", so this correspondence generalizes to MDDs representing arbitrary sets of constraints.

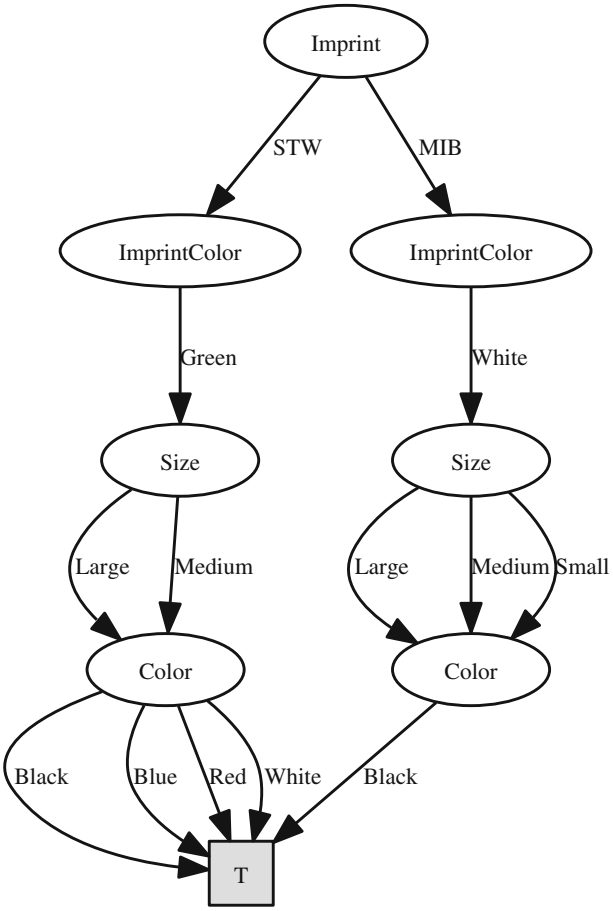


Fig. 4 MDD corresponding to VDD in Fig. 3 with set-labeled nodes

in Andersen et al. (2010) BDD packages are suggested for finding a good variable ordering. The MDD is then extracted from the BDD. The evaluation for set-labeled nodes using an intersects test given in Section 6.1 may not suggest itself in the MDD paradigm either.

Thus, the question of which representation (VDD or MDD) is more adequate for compiling and evaluating a variant table has not yet been determined. VDDs bring together concepts from databases, c-tuple compression, and decision diagrams for a restricted application domain. This appears to be very fruitful. Applications using MDDs have been focussed on compiling all constraints in a product model at once, which may be a harder problem (Andersen et al. 2010; Berndt 2016).

Implementational complexity is also a concern for business software that is to be deployed in practice. The VDD implementation (see Section 7.1) is based on a binary data structure, which was straightforward to implement without additional machinery. A comparison of this with the implementational complexity of existing MDD packages has not yet been done.

7 Empirical results

7.1 The java VDD implementation

The functionality, performance, and simplicity of the approach presented here is validated by a VDD implementation in Java. This was designed to be able to replace variant table handling in a legacy configuration environment such as the SAP VC or the SAP IPC.

The VDD implementation has not yet been deployed in practice, but it was tested for functional correctness against 238 variant tables as described in Haag (2015b). Functional correctness means, that the results of performing the same configuration with and without VDDs yielded identical results. The performance results obtained there show the promise of the VDD approach.

The compression and evaluation results in the following subsections are obtained using this implementation. Time measurements used the maximal accuracy available in Java, `System.nanoTime()`, on an Apple Mac mini with 2.5 GHz Intel Core i5 and 8GB memory.[20] However, all results are given rounded to milliseconds (milli).

The VDD implementation currently supports the following data types for characteristics: *string*, *integer*, *float* (Java *double*), and *numeric interval*. The data type *decimal* can be substituted either using strings, integers, or doubles, but this data type should receive more support in the future. As the examples show, *string* is a very natural data type for business applications.

7.2 Compression results for the T-shirt

The simple T-shirt is easily represented as in Table 1. Maintenance of the T-shirt variants in the extended T-shirt example of Section 2.2 is more tedious. A business may choose to maintain the four separate tables (Tables 2, 3, 4, and 5) in compressed form as c-tuples. However, it is also possible to maintain the *equi-join* of these four tables (the lefthand side of (2)) as a database table with more than 100000 rows.

Table 7 gives the compression results for each of these T-shirt tables. In each case, the compilation into a VDD using the preferred column heuristic (see Section 5) was from a relational representation of the table.[21]

The higher the compression ratio, the more advantageous the VDD is compared to a relational table. For the joined T-shirt table the compression ration is 99.95 %. Here, the VDD promises huge performance benefits (see Section 7.4). The ratio of the number of features s to the number of table cells $N = kr$ is also important. In the extreme case that $s = N$ no compression is possible. However, a VDD is then not worse off than a column oriented database.

7.3 Compression of the Renault Megane (RM) variant tables

The *Renault Megane* (*RM*) model can be represented as 113 variant tables. Table 8 gives summary information about the distribution of the sizes of the parameters k (*arity*), s (number of distinct features), and $N = kr$ (overall size) of each table. Overall, the sizes of the

[20] Memory is not an issue so far. The same tests with similar results were performed on a MacBook Pro with 2.5 GHz Intel Core i5 and 4GB memory.

[21] The fact that the number of c-tuples differs from that evident in Tables 2, 3, 4, and 5 is due to the effect of the heuristic employed in the table compilation.

Table 7 Compression of T-shirt tables

Table	k	r	s	n	$compr\%$	$t(milli)$	n^*	c
Simple T-shirt (Table 1)	7	11	13	16	79.22%	0	10	2
Styles (Table 2)	3	52	14	14	91.03%	1	6	3
Dyes (Table 3)	3	24	21	37	48.61%	1	28	15
Imprints (Table 4)	3	2007	333	441	92.68%	31	14	5
Prices (Table 5)	3	15	20	20	55.56%	0	17	8
Extended T-shirt (join)	8	120120	373	528	99.95%	2233	134	57

Legend: k is the *arity*, r the number of *rows*, s the number of distinct features, n the number of unmerged VDD nodes, $compr\% := (N - n)/N$ (with $N := kr$, the number of table cells) the compression ratio, and t the compilation time. Also given are n^* the number of nodes of the merged VDD and c the number of c-tuples that result from this

RM variant tables are less diverse than those of the 238 variant tables described in Haag (2015b).

Table 9 gives the average compression results for all 113 tables in the same format as Table 7 for T-shirts. Compression results are given for five individual tables: the table with the the largest *arity* (k), largest number of rows (r), smallest, median, and largest number of features (s). The other 108 RM tables are sorted by s and divided into four equal blocks. The average compression is given for each block.

Again, the compression ratio of the largest table is 99.95 %. Small tables are less compressible than big ones, but the overall average is close to 80 %. This means that preferred column heuristic is doing extremely well with the RM variant tables.

In order to have a simple test for the quality of the *preferred column ordering* when constructing a VDD, the results it produces were compared with those produced by using the *natural column heuristic* (that uses the order of columns externally given in the definition of the variant table). The two column heuristics will not differ, if the table is already in the preferred column order. However, if a significant proportion of tables compiles better in natural order than for the preferred "best guess", then this is a signal that it would be worthwhile to pursue a further search for a better column order.

It turns out that for the RM variant tables the preferred column heuristic is strictly dominated by the natural column heuristic for 58 variant tables. Thus, for the RM variant tables the natural column order in which the tables are presented is better in about half of the tables. This was not the case for the 238 variant tables evaluated in Haag (2015b). On average, the preferred column heuristic does outperform the natural column heuristic by 40 nodes.

Table 8 Size statistics on 113 Renault Megane (RM) variant tables

Range	k	r	s	N
Average	5	1724	35	11400
Minimum	2	3	4	6
Median	6	79	27	364
Maximum	10	48721	99	292326

Legend: column definitions k, r, and s as for Table 7. $N = kr$ is the number of table cells

Table 9 Compression of RM variant tables with preferred column heuristic

Table	*k*	*r*	*s*	*n*	*compr%*	*t(milli)*	*n**	c
Average	5	1724	35	92	77.83 %	31	48	24
Min s	2	3	4	5	16.67 %	0	4	2
Median s	6	79	24	68	85.65 %	1	49	22
Max s	9	164	99	288	80.49 %	4	229	56
Max k	10	342	57	343	89.97 %	4	295	119
Max kr	6	48721	87	142	99.95 %	1007	52	45
Avg tables 1 – 27	4	41	14	32	60.90 %	0	21	11
Avg tables 28 – 54	6	68	25	69	81.73 %	3	51	21
Avg tables 55 – 81	4	213	36	67	73.95 %	1	28	15
Avg tables 82 – 109	6	6617	63	178	95.35 %	120	73	45

Legend: column definitions as for Table 7

Table 10 shows the compression results using the natural column heuristic. It shows the overall average and results for the five individual tables given in Table 9 (the first six lines). The preferred column heuristic loses in two of the five individual cases shown (*Median s* and *Max k*), as can be seen when comparing with Table 9.

Finally, it remains to evaluate the effect of merging nodes. For the largest table (*Max N*) with 292326 table cells, the heuristics performed as follows: The natural column heuristic produced 310 unmerged and 44 merged; The preferred column heuristic produced 142 unmerged and 58 merged. Although the preferred column heuristic is better than the natural column heuristic without merging, merging is better using the VDD produced with the natural column heuristic than using the smaller one produced with the preferred column heuristic. The best merged VDD with only 44 set-valued nodes has about a factor of three less nodes than the best non-merged VDD. But, as noted, there is some additional effort in run-time evaluation. The net effect of this trade-off is still under evaluation (see Section 7.4). The best merged VDD with 44 nodes is small enough to be meaningfully visualized graphically. Figure 5 gives a flavor of this. Due to limitations on the size of figures here the graph is reproduced too small to easily decipher the node labels, but gives an impression of its structure.

For completeness, a non column oriented heuristic called *h1* was also applied to the RM variant tables.[22] There were four tables for which *h1* outperformed the preferred column heuristic and 15 for which *h1* outperformed the natural column heuristic. However, in no case was *h1* better than the best choice of the preferred or natural column heuristic.

So, in summary, the preferred column heuristic wins overall, but other heuristics (both column oriented and non-column oriented) cannot be precluded from consideration. A search for a better column order should be provided in the VDD implementation in the future (where constraints on compilation time allow this).[23] The search for general (non-column) heuristics would be a more advanced topic.

[22] Heuristic *h1* is the first heuristic implemented (see Haag (2015b)). It was discarded, because it has non-linear compile times, which makes its application to large tables unattractive.

[23] Berndt (2016) provides some results on this for MDDs that may prove useful here as well.

Table 10 Compression of RM variant tables with the natural column heuristic

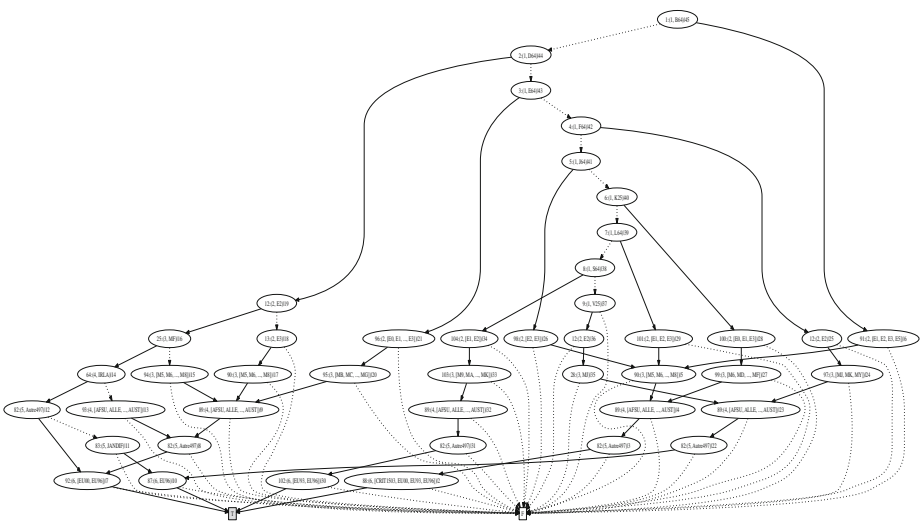
Table	*k*	*r*	*s*	*n*	*compr%*	*t(milli)*	*n**	c
Average	5	1724	35	132	78.37 %	29	47	22
Min s	2	3	4	5	16.67 %	0	4	2
Median s	6	79	24	62	86.92 %	0	51	20
Max s	9	164	99	356	75.88 %	2	349	113
Max k	10	342	57	337	90.15 %	3	275	155
Max kr	6	48721	87	310	99.89 %	1285	44	13

Legend: column definitions as for Table 7

7.4 Performance measurements for filtering the RM tables

The runtime performance results given here are intended as a statement of what can be expected in practice. The focus is on the filtering function and *arc consistency*, because these are at the core of what is needed to support (interactive) product configuration (see Section 2). The VDD implementation (see Section 7.1) does not currently include a propagation algorithm for arc consistency. Hence, a naive constraint propagation algorithm was implemented outside of the actual VDD implementation. "Naive" means that no effort was made to in any way optimize the propagation, and better performance results could be expected after tuning this algorithm.

All VDDs for the performance tests were compiled using the *preferred column heuristic* − with and without merged set-valued nodes. There are overall 99 characteristics used in the RM model. The index $p = 1 \ldots 99$ will be used to refer to characteristics in this total set (v_p) and their domains (D_p). The index $j = 1 \ldots k$ will be used to refer to characteristics of a particular variant table with arity k, as hitherto.



44 nodes represent 292326 cells (6 × 48721)

Fig. 5 Merged VDD of largest table in Renault Megane (RM) model

Table 11 Performance results for domain restrictions

	unmerged			*merged*		
Averages	n	t	$t_N = t/n$	n	t	$t_N = t/n$
overall	92	2	0.04	48	3	0.15
Small VDDs	30	1	0.08	8	1	0.31
Medium VDDs	67	1	0.02	41	3	0.09
Large VDDs	151	2	0.02	81	5	0.06

Legend: VDDs sorted ascending by number of VDD nodes n (separately for *merged* and *unmerged* compilation), t average time for 100 filterings per table in milliseconds, $t_N = t/n$ average time per node in milliseconds

Three different evaluations were measured for each of the RM tables $\mathfrak{T}_q, q = 1 \ldots 113$:[24]

1. Simple filtering was used to determine all column domains D_j of $\mathfrak{T}_q$ without any prior restriction ($\mathbf{R} = \Omega$).
2. Simple filtering was applied to $\mathfrak{T}_q$ with a restriction $R_1 \subset D_1$ of the first column to the first half of its domain.
3. Constraint propagation to achieve arc consistency using all RM tables was applied after setting the first value in D_1 of T_q ($p(v_1, x_1) = true$).

For the first two evaluations, all unit tests were repeated 100 times in order to get results meaningfully expressible in milliseconds. For the first evaluation, the total (summed) time needed to perform this test 100 times for each table was 14 milliseconds (unmerged) and 21 milliseconds (merged). For the second evaluation, the total (summed) time needed was 231 milliseconds (unmerged) and 401 milliseconds (merged). For these two evaluations, measurements were also performed for each VDD separately.

Table 11 gives averages (for 100 repetitions of the test), both for VDDs with and without merged set-labeled nodes. The VDDs for the tables are sorted by number of nodes (ascending), separately for the VDDS with and without merged nodes. The VDDs are divided into three equal groups by size. Averages are given for the overall set of VDDs, the first third (*small VDDs*), the second third (*medium VDDs*), and the last third (*large VDDs*). The evaluation time of a VDD is expected to mainly depend the number of its nodes (see Section 4.3). Table 11 also shows averages for evaluation time per node. The results show that for smaller tables some other effects are noticeable that lead to somewhat higher evaluation times.

For the third evaluation, constraint propagation was first done in an initial state without external restrictions to determine the "real" domains of the 99 characteristics v_p, $p = 1 \ldots 99$, and these domains were cached. It took 42 milliseconds to achieve this initial arc consistency both for unmerged and merged VDDs (see Table 12). The performance of merged and unmerged VDDs did not differ much in the further tests. Table 12 gives the overall average, the minimum, median, and the maximum times (in milliseconds) of achieving arc consistency after restricting one characteristic to its first value and setting all other domains to the cached initial domains.

[24] The 113 RM tables are extracted from a model expressly published to enable performance measurements (Andersen et al. 2010). As the focus in Andersen et al. (2010) is somewhat different, the results reported there do not directly compare.

Table 12 Performance results for arc consistency

	unmerged *t (millis)*	*merged* *t (millis)*
Initial	42	42
Average	3	4
Min	0	0
Median	2	4
Max	14	15

Legend: *t* evaluation time in milliseconds

From a practical perspective, these results are very good. Table 12 shows that the average times for constraint propagation are below 10 milliseconds, and the longest time is below 50 milliseconds (below 20 once the initial domains have been determined).

8 Summary and conclusions

In effect, the VDD approach presented here offers a reduced database functionality for querying variant tables as in (7), as well as iteration over the result sets. The functionality provided is tuned to checking the consistency of a product configuration and supporting the filtering of inadmissible product features. This is needed for *constraint propagation* to achieve *arc consistency*. The idea of decomposing a variant table into subtables and reducing the ensuing decomposition tree to a *binary* directed acyclic graph as set forth in Section 4 has proven very fruitful. Large variant tables from actual product models have so far been substantially compressible. In this respect the VDD compression competes both with the compression inherent in a column-oriented database, as well as that obtainable with other decision diagrams (Knuth 2011).[25]

As pointed out in Section 4.4, a VDD shares the same data structure as a ZDD, but is not in itself necessarily a ZDD. The VDD implementation (see Section 7.1) is not based on a VDD being a ZDD. However, the current heuristics used to construct VDDs do result in these VDDs being ZDDs. The conceptual advantage of this is that all results and algorithms given for ZDDs in Knuth (2011) then apply. Accordingly, a description of the own algorithms used for constructing a ZDD from the decomposition tree has been omitted here.

Besides querying a VDD, the maintenance of variant tables is an important practical aspect. Since variant tables are often very compressible, maintenance in a spreadsheet in the form of *c-tuples* suggests itself. It is easy to expand such c-tuples into a relational form of *value (feature) tuples*. The converse is the subject of research, such as Katsirelos and Walsh (2007). The *column heuristics* proposed in Section 5 together with merging VDD nodes to set-labeled nodes (Section 6.1) offer a way to obtain an external representation of a variant table in compressed form as c-tuples, and this is also a contribution to this research.

A *column heuristic* is a proposed static heuristic for constructing a VDD with fast compilation times (see Section 7.3). Moreover, the *preferred column heuristic* has yielded very

[25] It would be interesting to investigate whether a column-oriented database could in itself be directly used in constraint propagation. This was proposed by a colleague at SAP some time ago, but has not been followed up on.

acceptable overall compression (although not necessarily optimal). This means that it facilitates table maintenance through a spreadsheet-like external representation, which can be compiled back to a VDD quick enough to be acceptable in user interaction (see Section 7.4).

Table maintenance with VDDs directly (without generating an explicit set of c-tuples) is a current development topic for the VDD implementation. The rationale for this is that some variant tables cannot be compressed using c-tuples, but may be compressible using VDDs. For example, if a column *VariantID* is added for the extended T-shirt in (1) that assigns a unique identifying number to each possible T-shirt variant, then any representation as a list of tuples would have one tuple for each variant, and is therefore not compressible. However, a compression using a VDD is still possible for shared subtables.

VDDs support the following features that either exist in the legacy environment of the SAP VC ((Blumöhr et al. 2012)) or are required by customers:

- *String* and *numeric* data types (see Section 7.1 for details)
- Numeric intervals in table cells and restrictions (see Section 6.3)
- Negative tables (listing excluded combinations of product features)

Negative variant tables are the subject of Haag (2015a), but this is considered out of scope here, as the compression possible due to negation did not prove to be additionally beneficial over the straight VDD compression described here.

As pointed out in Section 6.4, a VDD constructed with a column heuristic and subsequent merging to set-labelled nodes allows a direct mapping to and from an MDD. The VDD implementation has so far not been compared with MDD implementations. But, in the event that an MDD package is deployed in a business, the heuristic approach to generating VDDs could be used to generate MDDs there. Conversely, a VDD offers a binary representation of an MDD. This could be used in evaluating the MDD should that prove opportune. For example, the complexity of evaluation in a VDD directly depends on its number of nodes (see Sections 4.3 and 4.3), which allows precise performance predictions.

Besides the topic of providing direct support for variant table maintenance, the following topics are also subjects for future work:

- Improved performance for the evaluation of set-labeled nodes.
- Dynamic determination of a best column order. For MDDs this is treated in Berndt (2016).
- Providing support for an *equi-join* ($\bowtie$) operation on tables as in (2). This is ongoing development.
- Using VDDs to store problem solving states inside (legacy) arc consistency and constraint satisfaction algorithms. For MDDs this is the topic of Andersen et al. (2007).
- More closely comparing the MDD approaches with the VDD approach.

The goal of the work with MDDs in Berndt (2016) and Berndt et al. (2012) and with BDDs in Matthes et al. (2012) is to check consistency given very large sets of constraints. A competing approach looked at there is to use SAT solving (determining Boolean satisfiability). As the goal here is different, SAT-solving was not considered.

To sum up: The current state of the VDD implementation meets the goal of providing fast compression and evaluation of variant tables in a form that is suitable for deployment in a legacy environment with low cost and risk. As tables are treated individually, it is always possible to fall back on the existing legacy solution for any tables that may not be compilable to a VDD. The fast compile times are important in an interactive table maintenance setting.

Rather surprisingly, evaluation of merged VDDs with set-labeled nodes is currently slower than their non-merged counterparts. The author believes this is due to a suboptimal implementation, and one topic of future work is trying to improve this. The sizes of the actual files used to store a VDD is noticeably smaller using set-labeled nodes, so they do provide a definite advantage.

Finally, a more general research topic would be to more formally investigate commonalities between the following three approaches:

- BDDs in various flavors ((Knuth 2011))
- Compression into c-tuples and constraint slicing ((Gharbi et al. 2014; Katsirelos and Walsh 2007))
- Read optimized databases (such as column-oriented databases ((Stonebraker et al. 2005))).

The VDD approach has elements of all three of the above. There are differences in the underlying technology that needs to be deployed and in the heuristics that suggest themselves.

References

Amilhastre, J., Fargier, H., & Marquis, P. (2002). Consistency restoration and explanations in dynamic CSPs Application to configuration. *Artificial Intelligence*, *135*(1-2), 199–234. doi:10.1016/S0004-3702(01)00162-X.

Andersen, H., Hadzic, T., & Pisinger, D. (2010). Interactive cost configuration over decision diagrams. *Journal of Artificial Intelligence Research (JAIR)*, *37*, 99–139. doi:10.1613/jair.2905.

Andersen, H.R., Hadzic, T., Hooker, J.N., & Tiedemann, P. (2007). A constraint store based on multivalued decision diagrams. In *Bessiere* (Bessiere 2007) (pp. 118–132). doi:10.1007/978-3-540-74970-7_11.

Berndt, R. (2016). Decision diagrams for the verification of consistency in automotive product data. Hochschulschrift, dissertation, thesis, Friedrich-Alexander-Universitt Erlangen-Nrnberg (FAU). http://nbn-resolving.de/urn:nbn:de:bvb:29-opus4-69013, http://d-nb.info/108242644X/34, https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/6901. Zusammenfassung in deutscher Sprache.

Berndt, R., Bazan, P., Hielscher, K.J., German, R., & Lukasiewycz, M. (2012). Multi-valued decision diagrams for the verification of consistency in automotive product data, In Tang, A., & Muccini, H. (Eds.) *2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012, pp. 189–192. IEEE*. doi:10.1109/QSIC.2012.43.

Bessiere, C. (2006). Constraint propagation. In Rossi, F., van Beek, P., & Walsh, T. (Eds.) *Handbook of Constraint Programming, chap. 3. Elsevier*.

Bessiere, C. (ed.) (2007). Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings, Lecture Notes in Computer Science, vol. 4741 Springer.

Blumöhr, U., Münch, M., & Ukalovic, M. (2012). Variant Configuration with SAP, second edition. SAP Press Galileo Press.

Felfernig, A., Hotz, L., Bagley, C., & Tiihonen, J. (eds.) (2014). Knowledge-Based Configuration. Morgan Kaufmann, Boston. doi:10.1016/B978-0-12-415817-7.00027-X.

Gharbi, N., Hemery, F., Lecoutre, C., & Roussel, O. (2014). Sliced table constraints: Combining compression and tabular reduction. In Simonis, H. (Ed.) *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8451, pp. 120–135. Springer*. doi:10.1007/978-3-319-07046-9_9.

Haag, A. (2014). Chapter 27 - Product Configuration in SAP: A Retrospective, In Felfernig, A., Hotz, L., Bagley, C., & Tiihonen, J. (Eds.) *Knowledge-Based Configuration, pp. 319 – 337. Morgan Kaufmann, Boston*. doi:10.1016/B978-0-12-415817-7.00027-X.

Haag, A. (2015a). Arc consistency with negative variant tables. In Tiihonen et al. 2015*, pp. 81–87*. http://ceur-ws.org/Vol-1453/13_Haag_ArcConsistencyWithNegative_Confws-15_p81.pdf.

Haag, A. (2015b). Column oriented compilation of variant tables. In Tiihonen et al. 2015*, pp. 89–96*. http://ceur-ws.org/Vol-1453/14_Haag_ColumnOrientedCompilationOf_Confws-15_p89.pdf.

Haag, A., & Riemann, S. (2011). Product configuration as decision support: The declarative paradigm in practice. *AI EDAM*, *25*(2), 131–142. doi:10.1017/S0890060410000582.

Hadzic, T. (2004). A bdd-based approach to interactive configuration. In Wallace, M. (Ed.) *principles and practice of constraint programming - CP 2004, 10th international conference, CP 2004, toronto, Canada, September 27 - October 1, 2004, Proceedings, Lecture Notes in Computer Science, vol. 3258, p. 797. Springer*.

Hadzic, T., Hansen, E.R., & O'Sullivan, B. (2008). Layer compression in decision diagrams. In *20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2008), November 3-5, 2008, Dayton, Ohio, USA, Volume 1, pp. 19–26. IEEE Computer Society*. doi:10.1109/ICTAI.2008.92.

Katsirelos, G., & Walsh, T. (2007). A compression algorithm for large arity extensional constraints. In *Bessiere* (Bessiere 2007) (pp. 379–393).

Knuth, D. (2011). *The Art of Computer Programming, vol. 4A Combinatorial Algorithms Part 1, chap. Binary Decision Diagrams,* (pp. 202–280). Boston: Pearson Education.

Lecoutre, C. (2011). STR2: optimized simple tabular reduction for table constraints. *Constraints*, *16*(4), 341–371.

Matthes, B., Zengler, C., & Küchlin, W. (2012). An improved constraint ordering heuristics for compiling configuration problems. In Mayer, W., & Albert, P. (Eds.) *Proceedings of the Workshop on Configuration at ECAI 2012, Montpellier, France, August 27, 2012, CEUR Workshop Proceedings, vol. 958, pp. 36–40. CEUR-WS.org*. http://ceur-ws.org/Vol-958/paper7.pdf.

Mishchenko, A. (2001). *An introduction to zero-suppressed binary decision diagrams*. Tech. rep.: Portland State University.

Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., & Zdonik, S.B. (2005). C-store: A column-oriented DBMS. In Böhm, K., Jensen, C.S., Haas, L.M., Kersten, M.L., Larson, P., & Ooi, B.C. (Eds.) *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005, pp. 553–564. ACM*. http://www.vldb2005.org/program/paper/thu/p553-stonebraker.pdf.

Tiihonen, J., Falkner, A.A., & Axling, T. (eds.) (2015). Proceedings of the 17th International Configuration Workshop, Vienna, Austria, September 10-11, 2015, CEUR Workshop Proceedings, vol. 1453. CEUR-WS.org. http://ceur-ws.org/Vol-1453.