

IJS delovno poročilo
DP-15615
2026

Transformations of DEX Decision Tables: A Formal Framework

Marko Bohanec

Jožef Stefan Institute, Department of Knowledge Technologies
`marko.bohanec@ijs.si`

Abstract

This study examines transformations of decision tables in DEX (Decision EXpert), a qualitative multi-criteria decision modeling method. DEX models consist of hierarchical structures of attributes connected by decision tables, which map combinations of input attribute values to an output value. The dimensions of a decision table depend on the attributes involved and their discrete qualitative scales. Changes to the model structure, attributes, or their scales may alter not only the dimensions of the corresponding decision tables but also their contents, represented by elementary decision rules. This study proposes a formal theoretical framework for defining and analyzing DEX decision tables and their transformations. The framework provides precise mathematical definitions of DEX model components and decision table subtypes, and formalizes the requirements for preserving information and preferential dominance. Particular attention is devoted to standard decision tables, a subtype that accounts for the vast majority of DEX decision tables encountered in practice. A total of 25 transformations are identified and systematically described through four elements: a mathematical definition, a transformation algorithm, a proof of information preservation, and an illustrative example. Theoretical implications of extending the framework to decision tables with outcomes represented by probability value distributions are also discussed. Beyond its theoretical contributions, this work establishes a formal foundation for the further development and implementation of the DEX method.

Keywords: Multi-criteria model, method DEX, decision table, decision rule, decision table transformation.

Contents

Abstract	ii
1 Introduction	1
2 Basic Concepts	3
2.1 DEX attributes and scales	3
2.2 DEX decision table	3
2.2.1 Decision space	3
2.2.2 Decision table representation	3
2.2.3 Decision table subtypes	4
2.2.4 Monotone decision tables	5
2.2.5 Standard decision tables	5
2.2.6 Decision table example	5
2.2.7 Conceptual and notational conventions	6
3 Information and Dominance Preservation	8
3.1 Information preservation	9
3.1.1 Condition correspondence	9
3.1.2 Outcome-space translation	9
3.1.3 Corresponding rules	9
3.1.4 Strict information preservation	9
3.1.5 Weak information preservation	10
3.1.6 Preservation of non-entered rules	10
3.1.7 Summary	10
3.2 Dominance preservation	11
4 Transformations of Standard Decision Tables	13
4.1 Transformations due to changes of input attributes	13
4.1.1 Add/Insert an attribute	13
4.1.2 Duplicate an attribute	16
4.1.3 Delete an attribute	19
4.1.4 Exchange/Move input attributes	21
4.2 Transformations due to changes of an input attribute scale	22
4.2.1 Add/Insert input value	22
4.2.2 Duplicate input value	24
4.2.3 Delete input value	26
4.2.4 Merge input values	29
4.2.5 Copy input value	32
4.2.6 Move input values	34
4.2.7 Exchange input values	35
4.3 Transformations due to changes of output attribute scale	38
4.3.1 Add/Insert output value	38

4.3.2	Duplicate output value	39
4.3.3	Delete output value	40
4.3.4	Merge output values	42
4.3.5	Copy output value	44
4.3.6	Move output value	45
4.3.7	Exchange output values	47
4.4	Structural transformations	48
4.4.1	Flatten subtree	49
4.4.2	Decompose a table	52
4.5	Local transformations	55
4.6	Summary	56
5	On Transformations of Non-Standard Decision Tables	59
5.1	Comparing outcome values	59
5.2	Effect on individual transformations	60
5.3	Flatten	61
5.4	Decompose	61
5.5	Information-preservation proofs	61
5.6	Complexity	61
6	Conclusions	62
	References	65

1. Introduction

DEX (Decision EXpert) [1] is a qualitative multi-criteria decision modeling method, aimed at supporting decision makers in complex decision-making tasks. The central element of the approach is a *DEX model*, which consists of hierarchically organized variables, called *attributes*. All attributes are qualitative and can take values represented by words. The bottom-up *evaluation* (aggregation) of values in a DEX model is governed by user-defined *decision tables*. In principle, each internal node (aggregate attribute) has an associated decision table that maps the values of immediate subordinate attributes (inputs) to the value of that attribute (output).

DEX models are usually developed “manually” by the involved decision makers, domain experts, and decision analysts. They define the structure of attributes, their value scales, and the corresponding decision tables. Since this task is demanding, it is typically carried out with the help of software tools such as DEXiWin [2], <https://dex.ijs.si/dexisuite/dexiwin.html>. While the model content, in terms of attributes and decision rules, is provided by its users, DEXiWin provides model-editing functionality, helps ensure the consistency and completeness of decision tables, evaluates decision alternatives, supports decision analysis, and provides many other features needed for practical decision support.

In this report, we focus on DEX *decision tables*. Specifically, we study *transformations* that change the structure and contents of such tables. When a DEX model is being edited, various changes may be introduced that affect already defined decision tables. For instance, when a new attribute is added to an existing set of input attributes, the corresponding decision table has to be extended by one column. This not only changes the dimensions of the table but may also affect its contents, if it has already been defined. We aim to preserve as much of the information already contained in the table as possible.

The motivation for this study is both practical and theoretical. All transformations that may affect decision tables during DEX model editing must be properly supported and implemented in the software. It is therefore essential to understand each of these transformations, describe them in precise mathematical terms, and develop appropriate transformation algorithms. On the theoretical side, it is interesting that such a seemingly simple object as a decision table gives rise to such a rich set of possible transformations. Many of these transformations become non-trivial when the focus shifts from merely modifying table dimensions to adapting and preserving the information contained in the table. This makes decision table transformation both an interesting and relevant topic. In addition, the literature on decision table transformation is scarce [3], and we aim to help fill this gap, at least for decision tables specific to DEX.

This study builds on a previous study [3], where we already identified several categories of decision table transformations related to the set of input attributes, the scales of input attributes and the scale of an output attribute. For each category, typical transformations include adding, deleting, duplicating, copying and moving some element, and exchanging and merging two elements. In total, 18 transformations were identified and described there. We also informally addressed the principle of information preservation.

The goal of the present study is to extend the previous work in a more formal direction, proposing a *formal framework* of DEX-decision-table transformations. We made a complete and precise list of transformations, characterizing each one (unless trivial):

- *Formally* in mathematical terms, including:
 - mathematical expressions precisely describing changes of individual decision table components (attributes, rules, ...),
 - proof that the transformation indeed “to the maximum extent preserves the information contained in the original decision table”.
- *Algorithmically* with pseudocode of operations necessary to carry out the operation.
- With an illustrative *example*.

Additionally, we advanced the concept of *information preservation* and proposed a formal approach for its assessment and analysis.

The remaining part of this document is structured as follows. In section 2 we define basic concepts and notation: DEX models and their components, DEX decision tables, monotone and standard decision tables. Section 3 defines the principles of information and dominance preservation. Section 4 covers the main transformation categories: due to changes of input attributes (section 4.1), changes of input attributes’ scales (section 4.2), changes of output attribute’s scale (section 4.3), structural changes (section 4.4) and local changes (section 4.5). All these transformations are summarized in section 4.6. Changes necessary to extend decision tables’ outcomes from single values to value distributions are discussed, at a less formal level, in section 5. Section 6 concludes the report.

2. Basic Concepts

2.1 DEX attributes and scales

Attributes are DEX model variables that represent some observable or assessed properties of decision alternatives. All attributes in a DEX model are qualitative: each attribute is associated with a discrete *value scale* (or *scale* for short), which is typically composed of words, such as ‘low’, ‘medium’, ‘unacceptable’ or ‘good’.

Scales (and, consequently, attributes) can be *ordered* or *unordered*. An ordered scale implies that consecutive values (in the direction from left to right in *increasing* scales) denote more and more preferred (advantageous) values. Ordered scale values can be compared by preferential comparison operators $\prec, \preceq, \simeq, \succeq$, and $\succ$. An unordered scale has no intrinsic preferential order and its values can only be compared for equality $=$ or inequality $\neq$.

In this report, we consider only unordered and *increasingly* ordered scales. *Decreasingly* ordered scales can be simply reversed and can thus be omitted without loss of generality, while substantially simplifying the notation and theoretical analysis.

2.2 DEX decision table

2.2.1 Decision space

A DEX *decision table* (DT) maps the values of n *input attributes* $X = \{x_i, i = 1, \dots, n\}$ to the value of a single output attribute y . These attributes are associated with corresponding *value scales*: scales of $x_i, i = 1, \dots, n$:

$$S_i = \{v_{i,1}, v_{i,2}, \dots, v_{i,k_i}\}, \quad k_i = |S_i| > 0$$

and scale of y :

$$S_y = \{v_{y,1}, \dots, v_{y,k_y}\}, \quad k_y = |S_y| > 0$$

Input scales form the *input space*:

$$\mathbf{S} = S_1 \times S_2 \times \dots \times S_n, \quad |\mathbf{S}| = k_1 k_2 \dots k_n$$

A *decision space* is the tuple $DS = \langle X, y, \mathbf{S}, S_y \rangle$. It defines decision table dimensions, consisting of input attributes X , output attribute y , input space $\mathbf{S}$ and output scale S_y .

2.2.2 Decision table representation

A *decision table* T is a mapping:

$$T : \mathbf{S} \rightarrow V(S_y)$$

Here, $V(S_y)$ is a value space over the output, further detailed in section 2.2.3.

The table is represented by a set of *decision rules* of the form

$$r_{\mathbf{v}} = (\mathbf{v} \rightarrow o)$$

where $\mathbf{v} \in \mathbf{S}$ (called rule *condition*) and $o \in V(S_y)$ (*outcome*). Conversely, constituent parts of some rule $r = (\mathbf{v} \rightarrow o)$ are denoted: $\mathbf{v}(r) = \mathbf{v}$ and $o(r) = o$. $\mathbf{v}_i$ denotes the i -th element of vector $\mathbf{v}$, i.e. the one that corresponds to attribute $x_i, i = 1, \dots, n$.

Furthermore, the table representation R consists of rules that cover the *whole input space*:

$$R = \{r_{\mathbf{v}} : \forall \mathbf{v} \in \mathbf{S}\}$$

An additional bit of information is associated with each decision rule r , denoted $entered(r)$, which can be either *true* or *false*. The value *true* denotes decision rules defined by the user; these are considered “untouchable” and should be kept unchanged as much as possible. On the other hand, the outcomes of non-entered rules can be changed while transforming the table.

The subset of all entered rules in R is $E(R) = \{r \in R : entered(r)\}$.

The notation R_{cond} , where *cond* is some condition over $\mathbf{v} \in \mathbf{S}$, denotes a conditioned subset

$$R_{cond} = \{r_{\mathbf{v}} \in R : cond(\mathbf{v})\}$$

In summary, a DT is always fully bound to its decision space:

1. The conditional part of the table contains *all possible combinations* of the values of input attributes.
2. There are *no missing* conditions in a DEX decision table and all conditions are *distinct*.
3. There is only one outcome column that can hold values from the value space over y .

DTs are usually displayed in a tabular form (Table 2.1), with columns composed of n input columns and one output column, displaying one rule per row. Rules are normally sorted by increasing input attribute values, and possibly grouped by output attribute values.

2.2.3 Decision table subtypes

DT *subtypes* are characterized by the class of values $V(S_y)$ that can be assigned to the output attribute y . Normally, one would expect that each decision rule would assign a single output value to y , so that $V(S_y) \equiv S_y$. This would change the mapping T to:

$$T : \mathbf{S} \rightarrow S_y$$

Unfortunately, this is too limited for any practical purposes. At minimum, the outcome should facilitate expressing uncertainty, either explicitly, formulating the outcome as “unknown” or “undefined”, or implicitly, using value intervals or value distributions. This is why the DEX method [1] and supporting software [2] extend $V(S_y)$ to at least value intervals and value sets.

The following DT subtypes are particularly useful:

- When y is ordered: $V(S_y) \equiv I(S_y)$, where $I(S_y) = \{[l, h] : l, h \in S_y, l \preceq h\}$, i.e. value intervals over S_y .
- When y is unordered: $V(S_y) \equiv 2^{S_y}$, i.e. value sets over S_y .

Both subtypes may extend $V(S_y)$ to include the single value “*undefined*”:

$$V_u(S_y) = V(S_y) \cup \{undefined\}$$

DT subtypes with even more extended $V(S_y)$ include those that use probability or fuzzy distributions over S_y [4].

2.2.4 Monotone decision tables

When at least one input attribute is ordered, preferential comparison can be generalized to decision rules. Let $r_a, r_b \in R$ be two decision rules. Then, $r_a \succeq r_b$ if it holds:

- for all ordered attributes $x_i \in X$: $\mathbf{v}(r_a)_i \succeq \mathbf{v}(r_b)_i$, and
- for all unordered attributes $x_i \in X$: $\mathbf{v}(r_a)_i = \mathbf{v}(r_b)_i$.

A more strict relation is $r_a \succ r_b$, where:

- $r_a \succeq r_b$, and
- for at least one ordered attribute $x_i \in X$: $\mathbf{v}(r_a)_i \succ \mathbf{v}(r_b)_i$.

We say that r_a *dominates* r_b . Conversely, r_b is *dominated by* r_a (denoted $r_b \prec r_a$).

The *principle of dominance* says that whenever $r_a \succ r_b$ and output y is an ordered attribute, it should also hold $o(r_a) \succeq o(r_b)$. That is, whenever values of ordered input attributes increase, the outcome value should also increase or remain unchanged. When the principle of dominance holds for all dominated pairs of rules, the table is called *monotone*.

Monotonicity is a highly desired property in decision models. During the construction of DTs, they should be continuously checked for monotonicity, and the model developer should be notified whenever a violation is detected. While user-defined rules should be preserved even if they violate monotonicity, it is common practice to automatically restrict the permissible values of as-yet non-entered rules to intervals implied by the dominance principle.

Sometimes, the term *consistency* is used as a synonym for monotonicity.

2.2.5 Standard decision tables

A *standard decision table* (SDT) is the main DT subtype considered in this study. A DT is called standard when:

- y is *ordered*, and
- all *entered rules* $r \in E(R)$ have a fully defined outcome $o(r) \in S_y$ (i.e. a *single value from* S_y).

The values of *non-entered rules* are then determined from entered rules following the principle of dominance. In general this means that outcomes of non-entered rules are from $I(S_y) \cup \{\text{undefined}\}$. Proper intervals $I(S_y)$ can be set when the entered rules follow the principle of dominance, and the table is monotone. The value *undefined* is set to those rules whose value intervals cannot be determined due to a violated principle of dominance. An algorithm for interval assignment for non-entered SDT rules is presented in section 3.2.

The focus on SDTs is justified because they are by far the most prevalent DT subtype in DEX models. This author maintains a database of real-life DEX models that were made available for research purposes [5]. Currently, the database contains 8674 decision tables, of which 8654 (99.77%) are standard.

2.2.6 Decision table example

Let $X = \{A, B\}$, both with scale $S_A = S_B = \{\text{low}, \text{med}, \text{high}\}$ (increasingly ordered as $\text{low} \prec \text{med} \prec \text{high}$), and output y with scale $S_y = \{\text{bad}, \text{acc}, \text{good}\}$ ($\text{bad} \prec \text{acc} \prec \text{good}$). All 9 rules are entered (Table 2.1), forming a fully defined, monotone table.

#	<i>A</i>	<i>B</i>	<i>Y</i>
1	<i>low</i>	<i>low</i>	<i>bad</i>
2	<i>low</i>	<i>med</i>	<i>bad</i>
3	<i>low</i>	<i>high</i>	<i>acc</i>
4	<i>med</i>	<i>low</i>	<i>bad</i>
5	<i>med</i>	<i>med</i>	<i>acc</i>
6	<i>med</i>	<i>high</i>	<i>acc</i>
7	<i>high</i>	<i>low</i>	<i>acc</i>
8	<i>high</i>	<i>med</i>	<i>acc</i>
9	<i>high</i>	<i>high</i>	<i>good</i>

Table 2.1: Example of a fully entered, monotone decision table.

2.2.7 Conceptual and notational conventions

Sets as tuples. Above, attributes X and value scales S are defined as sets, implying that the order of elements was not important. However, in practical decision support, the order *is* important. For example, the sequential order of attributes determines the order of columns in decision tables and the visual placement of subtrees in a model. Similarly, the order of scale values affects decision table printouts. In Table 2.1, for example, the orders of columns and decision rules depend on the ordering of attributes and their value scales, respectively. Also, in ordered scales, the positional order of scale values implicitly determines their preferential order. In practical terms, attribute and value sets are therefore *tuples* rather than *sets*, and are in algorithms actually implemented as such.

For simplicity, we shall hereafter still use the set notation, but additionally assume the existence of a mapping $pos_S : S \rightarrow \mathbb{N}$ between set elements and their *positions* (ordinal numbers). For some set $S = \{s_1, s_2, \dots, s_n\}$:

$$pos_S(s_i) = i, \quad i \in \{1, 2, \dots, n\}$$

When clear from the context, the subscript S is dropped.

Ordered scales. For ordered scales $S = \{v_1, v_2, \dots, v_k\}$ we assume, unless specified otherwise, that the positional order of elements implicitly determines their preferential order:

$$v_a \preceq v_b \iff a \leq b, \quad a, b \in \{1, 2, \dots, k\}$$

Define $v_a \simeq v_b$ when $v_a \preceq v_b$ and $v_a \succeq v_b$. We also denote:

$$\begin{aligned} \text{low}(S) &= v_1, & \text{high}(S) &= v_k \\ \text{min}(S) &= 1, & \text{max}(S) &= k \end{aligned}$$

Intervals. In SDTs, value intervals are assigned to outcomes of non-entered rules. We define *interval comparison*:

$$[l, h] \preceq [l', h'] \iff l \preceq l' \text{ and } h \preceq h'$$

An interval that has equal bounds is identified with a single value:

$$[o, o] \equiv o, \quad o \in S$$

Dominance relations. Dominance relations between decision rules depend only on the corresponding conditions of those decision rules. Therefore, we can extend them to conditions themselves:

$$\mathbf{v}(r_a) \mathbf{P} \mathbf{v}(r_b) \iff r_a \mathbf{P} r_b, \quad r_a, r_b \in R$$

where ‘P’ is any of the preference comparison relations ‘ $\prec$ ’, ‘ $\preceq$ ’, ‘ $\simeq$ ’, ‘ $\succeq$ ’, or ‘ $\succ$ ’.

Names of attributes and scale values. In practice, attributes and scale values are *named*. These names appear instead of mathematical symbols in decision table printouts. For this purpose, $name(x)$ denotes a textual representation of some attribute or scale value x . Normally, the names within a single set should be distinct, but we leave this responsibility to the user. Mathematically distinct set elements remain distinct even when they bear the same name.

Pseudocode notation. In algorithms we assume that decision rules R are represented by a dictionary R that maps S (rule conditions) to tuples $(outcome, entered)$. Here, *outcomes* are values composed according to the SDT requirements (single value from S_y for entered rules and extended value intervals for non-entered rules); *entered* is a Boolean value indicating whether the corresponding rule is entered or not. We assume that R always contains *all possible rule conditions*, so that $R[v]$ always refers to the outcome corresponding to $v \in S$ (the outcome value may be *undefined*, though). To retrieve or set the *entered* status, we use the notation $entered(R[v])$ or $entered(r)$, where r is a rule.

In order to set up R , we may:

- $R \leftarrow R'$: copy an entire other representation R' to R ,
- $R \leftarrow \text{Init}(DS)$: initialize R according to decision space DS , or
- $R \leftarrow \text{Init}(S, S_y)$: initialize R according to given input space S and output scale S_y .

Any form of $\text{Init}()$ creates a properly-dimensioned full dictionary that maps all possible conditions to *undefined* with *entered=false*.

The notation $E(R)$ represents the subset of entered rules from R .

3. Information and Dominance Preservation

A decision table *transformation* changes the decision space DS , for example by adding or removing an attribute, changing a scale, and merging or splitting scale values. Also, it induces a corresponding change in the table representation R . Thus, a transformation is an operator acting on both DS and R :

$$\tau : (DS, R) \longrightarrow (DS', R')$$

with $DS = \langle X, y, \mathbf{S}, S_y \rangle \rightarrow DS' = \langle X', y, \mathbf{S}', S'_y \rangle$. Output attribute y itself is usually preserved.

Attribute-level transformations. These transformations, such as “add, delete, duplicate, merge input attributes”, act on $X \rightarrow X'$ and correspondingly $\mathbf{S} \rightarrow \mathbf{S}'$. For instance, the transformation “add attribute x_{n+1} with scale S_{n+1} ”:

$$\tau_{add}(x_{n+1}, S_{n+1}) : \mathbf{S} = S_1 \times \dots \times S_n \mapsto \mathbf{S}' = S_1 \times \dots \times S_n \times S_{n+1}$$

Scale-level transformations. Transformations, such as “add/delete/merge/reorder a value within a scale S_i ”, act on $S_i \rightarrow S'_i$, inducing $\mathbf{S} \rightarrow \mathbf{S}'$.

Output attribute scale transformations. These transformations are induced by changes of output attribute’s scale and cause a transition

$$S_y \rightarrow S'_y$$

Rule remapping. Each transformation τ must define how each new condition $\mathbf{v}' \in \mathbf{S}'$ obtains its outcome. This is defined by an *induced rule map*:

$$\rho_\tau : \mathbf{S}' \rightarrow V(S_y), \quad R' = \{r_{\mathbf{v}'} = (\mathbf{v}' \rightarrow \rho_\tau(\mathbf{v}')) \mid \forall \mathbf{v}' \in \mathbf{S}'\}$$

with the constraint that entered rules are preserved wherever a natural correspondence exists:

$$\forall r \in E(R) \text{ with a corresponding } \mathbf{v}' \in \mathbf{S}' : \quad \rho_\tau(\mathbf{v}') = o(r)$$

When R is monotone, then new non-entered outcomes should also satisfy monotonicity:

$$\forall r'_a, r'_b \in R' : r'_a \succ r'_b \Rightarrow o(r'_a) \succeq o(r'_b)$$

Composability. Since transformations are just functions on (DS, R) pairs, they compose naturally:

$$\tau_2 \circ \tau_1(DS, R) = \tau_2(\tau_1(DS, R))$$

This gives a clean way to express complex edits. This is useful both for formal proofs about the transformations and for an implementation architecture.

In what follows, we further develop these ideas. In section 3.1, we explore the notion of information preservation in greater depth, defining the concepts of condition and rule correspondence, and outlining three types of information preservation: strict, weak, and those of non-entered outcomes. In section 3.2, we further explore the principle of dominance and propose an algorithm for interval assignment for non-entered SDT rules, which is common for all specific transformations considered in section 4.

3.1 Information preservation

3.1.1 Condition correspondence

A transformation $\tau : (DS, R) \rightarrow (DS', R')$ generally changes the decision space itself $\mathbf{S} \rightarrow \mathbf{S}'$, possibly $S_y \rightarrow S'_y$, so R and R' do not live in the same space. We cannot just compare them rule-by-rule. We first need a way to say which new rule conditions correspond to which old conditions.

For a transformation τ , define a *condition correspondence relation*:

$$\varphi_\tau \subseteq \mathbf{S} \times \mathbf{S}'$$

where $(\mathbf{v}, \mathbf{v}') \in \varphi_\tau$ means: condition $\mathbf{v}'$ in the new table represents the same decision situation as $\mathbf{v}$ did in the old table, restricted to whatever the transformation leaves conceptually unchanged.

This is deliberately a relation, not a function, because correspondence can be:

- *one-to-one*: e.g. when reordering or renaming scale values;
- *one-to-many*: e.g. when adding a new input attribute: one old $\mathbf{v}$ corresponds to k_{n+1} new conditions $\mathbf{v} \circ w$, $w \in S_{n+1}$;
- *many-to-one*: e.g. when deleting an attribute, or merging two scale values into one, several old conditions collapse onto a single new one.

φ_τ must be supplied as part of the definition of each specific transformation. It is what makes “adding an attribute” mathematically different from “duplicating an attribute”.

3.1.2 Outcome-space translation

If a transformation also changes the output scale or value-space subtype

$$S_y \rightarrow S'_y \text{ or } V(S_y) \rightarrow V(S'_y)$$

outcomes themselves need translating. An *outcome translation map* is:

$$\psi_\tau : V(S_y) \rightarrow V(S'_y)$$

ψ_τ is identity whenever y and its scale/value-space are untouched by τ , which is the common case.

3.1.3 Corresponding rules

Using φ_τ , extend *correspondence* from conditions to *rules*:

$$r_{\mathbf{v}} \sim_\tau r'_{\mathbf{v}'} \iff (\mathbf{v}, \mathbf{v}') \in \varphi_\tau$$

3.1.4 Strict information preservation

The strongest, cleanest version of the principle applies to *entered rules only*. Those are precisely the “information” that a human put into the table and must be preserved as much as possible.

A transformation τ is *strictly information-preserving* if, for every $r_{\mathbf{v}} \in E(R)$ and every $r'_{\mathbf{v}'} \in R'$ with $r_{\mathbf{v}} \sim_\tau r'_{\mathbf{v}'}$, holds:

$$o(r'_{\mathbf{v}'}) = \psi_\tau(o(r_{\mathbf{v}})) \quad \text{and} \quad \text{entered}(r'_{\mathbf{v}'}) = \text{true}$$

That is, wherever a new condition corresponds to an old, user-entered condition, the transformed table must carry over both the outcome (translated into the new value space) and the entered status. This makes $E(R)$ meaningful across transformations; entered-ness is not just preserved as a label, it propagates.

3.1.5 Weak information preservation

Strict preservation can fail to be satisfiable when φ_τ is many-to-one and the corresponding old entered rules disagree. For example, merging two scale values $v_{i,1}, v_{i,2}$ into one v'_i : if two entered rules differed only in $x_i = v_{i,1}$ and $x_i = v_{i,2}$, but had different outcomes, they cannot both map onto the single new condition.

For this we need a kind of conflict-resolution or combination operator for combining outcomes across value-space subtypes (intervals, sets, distributions):

$$\bowtie: V(S_y) \times V(S_y) \rightarrow V(S_y)$$

For $\mathbf{v}' \in \mathbf{S}'$, let $C(\mathbf{v}') = \{\mathbf{v} \in \mathbf{S} : (\mathbf{v}, \mathbf{v}') \in \varphi_\tau, r_{\mathbf{v}} \in E(R)\}$ be the entered old conditions corresponding to $\mathbf{v}'$.

A transformation τ is *weakly (best-effort) information-preserving* if

$$o(r'_{\mathbf{v}'}) = \bigwedge_{\mathbf{v} \in C(\mathbf{v}')} \psi_\tau(o(r_{\mathbf{v}})) \quad \text{whenever } C(\mathbf{v}') \neq \emptyset$$

with

$$\text{entered}(r'_{\mathbf{v}'}) = \text{true} \iff |C(\mathbf{v}')| \geq 1$$

When $|C(\mathbf{v}')| = 1$ this reduces exactly to strict preservation. When $|C(\mathbf{v}')| > 1$, then ‘ $\bowtie$ ’ deterministically resolves any conflict, e.g., by taking the convex hull for intervals, the union for sets, a weighted mixture for distributions. For SDTs, which require a single-value outcomes of entered rules, ‘ $\bowtie$ ’ is assumed to either resolve to single values from S_y , or the conflict is explicitly resolved in some context-specific way (e.g. “Delete input attribute” and “Merge input values” in sections 4.1.3 and 4.2.4, respectively). All this assures that no entered information is silently discarded; it is combined rather than overwritten.

3.1.6 Preservation of non-entered rules

When $C(\mathbf{v}') = \emptyset$, there are no corresponding entered rules, or the corresponding rules are all non-entered. Consequently, there are no rules whose outcomes must be preserved exactly. Nevertheless, the principle of preserving as much information as possible still applies, albeit at a weaker level: outcomes should preferably remain as close as possible to those implied by the original table. This principle can be formulated as an optimization criterion rather than an equality constraint. We do not pursue this idea further here. Instead, we assume that no information from non-entered rules is explicitly preserved in the transformed table; such rules are handled solely according to the principle of dominance (section 3.2).

3.1.7 Summary

Information is *preserved* as:

- *strict equality*: on entered rules with unambiguous correspondence;
- *join-combination* ($\bowtie$): on entered rules with conflicting many-to-one correspondence;
- *through dominance*: on non-entered rules, aimed at monotonicity.

This gives:

- A hard, verifiable constraint that any correctly-defined transformation must satisfy for entered rules; useful for proving correctness of specific τ ’s.
- A soft, optimization-based guideline for everything else, compatible with the monotonicity requirement.
- The obligation that every specific transformation must define its own φ_τ , and ψ_τ if it touches y . This must be specified when we formalize individual transformations (section 4).

3.2 Dominance preservation

Consider a DT with ordered y and at least one ordered input attribute. According to section 2.2.4, the outcomes of non-entered rules are not free. Once the entered rules are known, non-entered outcomes are determined by the principle of dominance: whenever $r_a \succ r_b$, it must hold $o(r_a) \succeq o(r_b)$. Non-entered outcomes generally take values in $I(S_y) \cup \{undefined\}$: an interval bounded by dominance constraints from surrounding entered rules, or *undefined* if no consistent interval can be derived, for example when the entered rules themselves violate monotonicity.

To this end, each transformed SDT (DS', R') is hereafter assumed to be *post-processed* by an algorithm that enforces the principle of dominance, as much as possible, on the non-entered rules of R' .

For each non-entered rule $r \notin E(R')$, the principle of dominance forces

$$o(r) \succeq o(r_e) \quad \forall r_e \in E(R') : r_e \preceq r, \quad o(r) \preceq o(r_e) \quad \forall r_e \in E(R') : r_e \succeq r$$

So the tightest admissible interval is bounded below by the largest outcome among entered rules dominated by r , and bounded above by the smallest outcome among entered rules dominating r .

Bounds of a rule. For a set of entered rules $E = E(R)$ and any rule r (entered or not) let

$$\text{low}(r) = \max\{o(e) : e \in E, e \preceq r\}, \quad \text{high}(r) = \min\{o(e) : e \in E, e \succeq r\}$$

with $\text{low}(r) = \text{low}(S_y)$ if no entered rule lies below r , and $\text{high}(r) = \text{high}(S_y)$ if none lies above. Here ' $\preceq$ ' is the weak dominance of rules, which is transitive. "CompleteByDominance" assigns $[\text{low}(r), \text{high}(r)]$ to every non-entered rule r , or *undefined* if $\text{low}(r) \succ \text{high}(r)$.

(Lemma 1) A monotone $E(R)$ never yields *undefined*. *Proof.* Let $E(R)$ be monotone and $r \notin E(R)$. We show $\text{low}(r) \preceq \text{high}(r)$. If no entered rule lies below r , then $\text{low}(r) = \text{low}(S_y) \preceq \text{high}(r)$; symmetrically if none lies above. Otherwise $\text{low}(r) = o(r_l)$ and $\text{high}(r) = o(r_h)$ for some entered $r_l \preceq r \preceq r_h$. By transitivity $r_l \preceq r_h$, and by monotonicity of $E(R)$, $o(r_l) \preceq o(r_h)$. $\square$

(Lemma 2) The outcomes satisfy the interval order. If $E(R)$ is monotone, then $R' = \text{CompleteByDominance}(DS, E(R))$ satisfies the principle of dominance for *all* pairs of rules, entered or not: $r \succeq r'$ implies $[\text{low}(r), \text{high}(r)] \succeq [\text{low}(r'), \text{high}(r')]$, where an entered rule has the degenerate interval $[o(r), o(r)]$.

Proof. First, for an entered r we have $\text{low}(r) = \text{high}(r) = o(r)$: r itself is among the rules considered, and by monotonicity every other entered rule below (above) r has an outcome $\preceq$ ($\succeq$) $o(r)$. So the outcome interval of every rule is $[\text{low}(r), \text{high}(r)]$ (by Lemma 1 it is never *undefined*). Let $r \succeq r'$. By transitivity, every entered rule below r' is also below r , so the maximum defining $\text{low}(r)$ ranges over a superset and $\text{low}(r) \succeq \text{low}(r')$. Likewise every entered rule above r is above r' , so $\text{high}(r) \succeq \text{high}(r')$. By the definition of the interval order, $[\text{low}(r), \text{high}(r)] \succeq [\text{low}(r'), \text{high}(r')]$. $\square$

Consequently, in order to determine outcome intervals of non-entered rules, it is sufficient to consider only entered rules and their outcomes. There is no need to cross-examine non-entered rules, because they stay mutually monotone by construction. If $E(R)$ is monotone, so will be R .

Algorithm 1 Interval assignment for non-entered SDT rules

Algorithm CompleteByDominance(DS, E):

Input: DS = $\langle X, y, S, S_y \rangle$, with $S = S_1 \times \dots \times S_n$; S_y ordered

E: set of entered rules, one rule $(v \rightarrow o)$, $o \in S_y$,

Output: R: one rule for every $v \in S$, each with an entered flag and
an outcome $o(r) \in I(S_y) \cup \{\text{undefined}\}$

```
function Below(a, b):    #  $a \preceq b$  on conditions
    for i = 1 to n:
        if  $x_i$  is ordered:
            if  $\text{pos}_i(a[i]) > \text{pos}_i(b[i])$ : return false
        else:
            if  $a[i] \neq b[i]$ : return false
    return true

R  $\leftarrow$  Init(DS)
for each v in S:
    if there is a rule  $(v \rightarrow o)$  in E:
        R[v]  $\leftarrow$  (outcome = o, entered = true) # copied unchanged
    else:
        low  $\leftarrow$  low( $S_y$ ) # no lower bound: scale minimum
        high  $\leftarrow$  high( $S_y$ ) # no upper bound: scale maximum
        for each rule  $(u \rightarrow o_u)$  in E:
            if Below(u, v) and  $o_u \succ \text{low}$ : low  $\leftarrow$   $o_u$ 
            if Below(v, u) and  $o_u \prec \text{high}$ : high  $\leftarrow$   $o_u$ 
        if low  $\succ$  high:
            o  $\leftarrow$  undefined # entered rules around v violate dominance
        else if low = high:
            o  $\leftarrow$  low # interval degenerates to a single value
        else:
            o  $\leftarrow$  [low, high] # proper value interval
        R[v]  $\leftarrow$  (outcome = o, entered = false)
return R
```

Algorithm 1 creates a new decision table representation R in decision space DS, given a (sub)set of entered rules E. Because ‘ $\succeq$ ’, restricted to $E(R)$, is transitive, comparing r directly against every entered rule, rather than propagating bounds through other non-entered rules, already yields the tightest bounds. No fixed-point iteration is needed. $\text{low} \succ \text{high}$ signals that r sits “between” two entered rules whose outcomes already violate the dominance principle relative to each other through r ; hence assigning *undefined*. If r is not comparable to any entered rule on one side, the scale extremes act as the natural bounds.

The time complexity of the algorithm is $O(m \cdot \ell \cdot n)$, where $\ell = |E|$, $m = |S| - \ell$, and each comparison costs at most one pass over n attributes.

In cases when R is an already initialized and possibly partially developed decision table, we shall occasionally simplify the call to CompleteByDominance(R). Doing this, we assume that both arguments to Algorithm 1, DS and E(R), can be deduced from R or implicitly from the context, and that the code operates directly upon R without initializing it. This simplified call can return the same modified table ($R \leftarrow \text{CompleteByDominance}(R)$) or its copy after modification ($R' \leftarrow \text{CompleteByDominance}(R)$).

4. Transformations of Standard Decision Tables

This section defines transformations applicable to standard decision tables. We distinguish three main categories of transformations that affect a single decision table: those resulting from changes in the set of input attributes (section 4.1), changes in the scales of input attributes (section 4.2), and changes in the scale of the output attribute (section 4.3). Additionally, we address two structural transformations (section 4.4) that affect the model structure around the transformed tables. For each transformation, we describe the resulting changes to the decision space DS and its decision rules R , justify the preservation of information, and outline the corresponding transformation algorithm, illustrated with an example (trivial proofs, algorithms and examples are omitted). Finally, we mention some local modifications (section 4.5) that only modify the interior of decision tables without altering their dimensions.

4.1 Transformations due to changes of input attributes

Transformations in this section are induced by a change of the set of input attributes X . Let $DS = \langle X, y, \mathbf{S}, S_y \rangle$ be a decision space with $X = \{x_1, \dots, x_n\}$, $\mathbf{S} = S_1 \times \dots \times S_n$, and let R be a standard decision table over DS . Non-entered rules of R already carry the outcomes implied by the principle of dominance (intervals $I(S_y)$, possibly *undefined*).

4.1.1 Add/Insert an attribute

We introduce a new attribute x_{new} with scale $S_{new} = \{v_{new,1}, \dots, v_{new,k_{new}}\}$, $k_{new} = |S_{new}|$, to be inserted at position $q \in [1, n+1]$. We distinguish between *insertion* (when $q \leq n$) and *addition* or *appending* (when $q = n+1$).

New decision space is:

$$\begin{aligned} X' &= (x_1, \dots, x_{q-1}, x_{new}, x_q, \dots, x_n), & \mathbf{S}' &= S_1 \times \dots \times S_{q-1} \times S_{new} \times S_q \times \dots \times S_n \\ DS' &= \langle X', y, \mathbf{S}', S_y \rangle, & |\mathbf{S}'| &= |\mathbf{S}| \cdot k_{new} \end{aligned}$$

Projection. Define $\pi_q : \mathbf{S}' \rightarrow \mathbf{S}$ as the “remove q -th component” operator:

$$\pi_q(\mathbf{v}') = \mathbf{v}' \setminus q$$

For fixed $\mathbf{v} \in \mathbf{S}$, $\pi_q^{-1}(\mathbf{v}) = \mathbf{v}' : \pi_q(\mathbf{v}') = \mathbf{v}$ has exactly k_{new} elements, one for each value of x_{new} inserted at position q . Write $\mathbf{v} \circ_q v'$ for the vector obtained by inserting value $v' \in S_{new}$ into $\mathbf{v}$ at position q .

Rule construction. Since we know nothing about how x_{new} affects the outcome, the only information-preserving choice is to keep the outcome of a rule unchanged as x_{new} varies. For every $\mathbf{v} \in \mathbf{S}$ and every $v' \in S_{new}$, define $\mathbf{v}' = \mathbf{v} \circ_q v'$ and

$$entered(r'_{\mathbf{v}'}) := entered(r_{\mathbf{v}}), \quad o(r'_{\mathbf{v}'}) := o(r_{\mathbf{v}}) \text{ whenever } entered(r_{\mathbf{v}})$$

Thus, the transformed set of entered rules is

$$E(R') = \{r'_{\mathbf{v}'} : \mathbf{v}' \rightarrow c \mid r_{\mathbf{v}} \in E(R), c = o(r_{\mathbf{v}}), v' \in S_{new}, \mathbf{v}' = \mathbf{v} \circ_q v'\}$$

Outcomes for rules of R' that are not in $E(R')$ are determined afterwards from $E(R')$ by the principle of dominance over $\mathbf{S}'$ (section 3.2), treating x_{new} as ordered or unordered according to its own scale type.

Algorithm. The “Add/Insert attribute” pseudocode is shown as Algorithm 2. Building the new raw table R' requires one pass, constant work per cell of time complexity $O(|\mathbf{S}| \cdot k_{new})$. After building the table, “CompleteByDominance” determines the outcomes of non-entered rules.

Algorithm 2 Add/Insert a new input attribute

```
function AddInputAttribute(R, S, X, x_new, S_new, q, S_y):
    # R : original SDT, given as dict v -> (outcome, entered)
    # S : [S_1, ..., S_n]    (list of scales, in attribute order)
    # X : [x_1, ..., x_n]    (attribute names)
    # x_new : new attribute
    # S_new : scale of x_new
    # q      : insertion position, 1 <= q <= n+1
    # S_y    : output scale

    X' ← Insert_at(X, q, x_new)
    S' ← insert_at(S, q, S_new)

    R' ← Init(S', S_y)

    for each v' in R': # all conditions in R', i.e. cartesian product of S'
        v ← remove_component(v', q) # projection pi_q
        rule ← R[v]
        if entered(rule):
            R'[v'] ← (outcome = o(rule), entered = true)
        else:
            R'[v'] ← (outcome = undefined, entered = false)

    R' ← CompleteByDominance(R')
    return R'
```

Proof of information preservation. We show three things: (a) every entered rule of R is preserved verbatim in R' (no information lost or altered), (b) fixing the new attribute’s value exactly recovers R (nothing spurious is added at the level of entered knowledge), and (c) if R is monotone, so is R' .

(Lemma 1) Bijection. For every $\mathbf{v} \in \mathbf{S}$, the map $v' \mapsto \mathbf{v} \circ_q v'$ is a bijection $S_{new} \rightarrow \pi_q^{-1}(\mathbf{v})$. *Proof.* $\mathbf{S}' = S_1 \times \dots \times S_{new} \times \dots \times S_n$ is a Cartesian product. Fixing every coordinate except position q determines a unique element for each choice of the q -th coordinate, and distinct choices give distinct vectors. $\square$

(Lemma 2) Entered-rule preservation. The map $r_{\mathbf{v}} \mapsto r'_{\mathbf{v} \circ_q v'} : v' \in S_{new}$ sends $E(R)$ to $E(R')$ bijectively onto $E(R) \times S_{new}$ (in cardinality), and every image rule carries exactly the outcome of $r_{\mathbf{v}}$.

Proof. Immediate from the construction: $entered(r'_{\mathbf{v}'}) = entered(r_{\pi_q(\mathbf{v}')})$ and $o(r'_{\mathbf{v}'}) = o(r_{\pi_q(\mathbf{v}')})$ whenever the right-hand side is entered. By Lemma 1 each $\mathbf{v}$ has exactly k_{new} preimages, all entered iff $r_{\mathbf{v}}$ is, all with outcome $o(r_{\mathbf{v}})$. Hence $|E(R')| = k_{new} \cdot |E(R)|$ and no entered value of R is dropped, duplicated inconsistently, or changed. $\square$

This is exactly the requirement on entered rules: none is lost or modified.

(Lemma 3) Slice recoverability. For any fixed $w \in S_{new}$, let $R'_w = \{r'_{\mathbf{v}'} \in R' : \mathbf{v}'_q = w\}$. Then the map $\mathbf{v} \mapsto \mathbf{v} \circ_q w$ is an isomorphism of decision tables $R \cong R'_w$. The same entered/non-entered status, the same outcomes.

Proof. Direct from the definitions: for each $\mathbf{v}$, $entered(r'_{\mathbf{v} \circ_q w}) = entered(r_{\mathbf{v}})$ and, when entered, $o(r'_{\mathbf{v} \circ_q w}) = o(r_{\mathbf{v}})$. Since $\mathbf{v} \mapsto \mathbf{v} \circ_q w$ is injective (Lemma 1) with image exactly R'_w 's domain, this is a bijection preserving all rule content. $\square$

Lemma 3 says: restrict R' to any single value of the new attribute and you get back exactly the original table. The transformation does not fabricate any dependency between x_{new} and y ; it merely replicates the existing knowledge along an uninformed new dimension.

(Theorem) Monotonicity preservation. If R satisfies the principle of dominance (is monotone), then $E(R')$ also satisfies the principle of dominance in $\mathbf{S}'$; consequently the dominance closure of $E(R')$ yields a well-defined (non-contradictory) SDT R' .

Proof. Let $r'_{\mathbf{a}}, r'_{\mathbf{b}} \in E(R')$ with $r'_{\mathbf{a}} \succ r'_{\mathbf{b}}$, i.e. $\mathbf{a}, \mathbf{b} \in \mathbf{S}'$ agree on all unordered attributes, weakly dominate on all ordered attributes, and strictly dominate on at least one. Let $\mathbf{v}_a = \pi_q(\mathbf{a})$, $\mathbf{v}_b = \pi_q(\mathbf{b})$.

- If x_{new} is *unordered*: dominance forces $a_q = b_q$ (equal new-attribute value), so the strict dominance must come from the original coordinates, giving $\mathbf{v}_a \succ \mathbf{v}_b$ in $\mathbf{S}$ (dominance restricted to a fixed value of x_{new} mirrors dominance in $\mathbf{S}$, since dropping an equal coordinate does not affect comparisons on the rest). By monotonicity of R , $o(r_{\mathbf{v}_a}) \succeq o(r_{\mathbf{v}_b})$, and by Lemma 2, $o(r'_{\mathbf{a}}) = o(r_{\mathbf{v}_a}) \succeq o(r_{\mathbf{v}_b}) = o(r'_{\mathbf{b}})$.
- If x_{new} is *ordered*: two sub-cases.
 - The strict dominance arises (at least partly) from the original coordinates, i.e. $\mathbf{v}_a \succeq \mathbf{v}_b$ with strict inequality somewhere among $x_1, \dots, x_n$: same argument as above gives $o(r'_{\mathbf{a}}) \succeq o(r'_{\mathbf{b}})$.
 - The strict dominance arises only from x_{new} ($a_q \succ b_q$) while $\mathbf{v}_a = \mathbf{v}_b = \mathbf{v}$: then by Lemma 2, $o(r'_{\mathbf{a}}) = o(r_{\mathbf{v}}) = o(r'_{\mathbf{b}})$, so $o(r'_{\mathbf{a}}) \succeq o(r'_{\mathbf{b}})$ holds trivially (equality).

In every case $o(r'_{\mathbf{a}}) \succeq o(r'_{\mathbf{b}})$, so $E(R')$ satisfies the principle of dominance. $\square$

Summary. The “Add/Insert attribute” transformation preserves the information content of R in the required sense:

1. every entered rule’s value survives unchanged (Lemma 2);
2. the original table is exactly recoverable as any fixed-value slice of R' along x_{new} (Lemma 3), confirming that nothing about the y - x_{new} relationship is invented;
3. monotonicity: a well-defined, contradiction-free dominance closure for the non-entered rules is inherited automatically from R (Theorem), so R' is a valid SDT whose non-entered outcomes are exactly the tightest intervals consistent with the lifted entered rules, neither over- nor under-committing.

This also explains, formally, the practical limitation that this transformation adds a dimension but not discriminative information. This is precisely why “Duplicate input” transformation (next section) is often preferred when more informative defaults are desired.

Example. Let us use the DT from Table 2.1. Insert a new attribute C with scale $S_C = \{no, yes\}$ ($k_{new} = 2$), appended at position $q = 3$, so that $x_{new} = x_3 = C$. New decision space is:

$$X' = \{A, B, C\}, \quad \mathbf{S}' = S_A \times S_B \times S_C, \quad |\mathbf{S}'| = 9 \times 2 = 18$$

New rules are constructed so that every rule $r_{\mathbf{v}} \in R$ (all are entered) is reproduced once for $C=no$ and once for $C=yes$, with the same outcome. No information about C ’s effect exists yet, so none is invented. The resulting decision rules are shown in Table 4.1.

#	A	B	C	Y
1	<i>low</i>	<i>low</i>	<i>no</i>	<i>bad</i>
2	<i>low</i>	<i>low</i>	<i>yes</i>	<i>bad</i>
3	<i>low</i>	<i>med</i>	<i>no</i>	<i>bad</i>
4	<i>low</i>	<i>med</i>	<i>yes</i>	<i>bad</i>
5	<i>low</i>	<i>high</i>	<i>no</i>	<i>acc</i>
6	<i>low</i>	<i>high</i>	<i>yes</i>	<i>acc</i>
7	<i>med</i>	<i>low</i>	<i>no</i>	<i>bad</i>
8	<i>med</i>	<i>low</i>	<i>yes</i>	<i>bad</i>
9	<i>med</i>	<i>med</i>	<i>no</i>	<i>acc</i>
10	<i>med</i>	<i>med</i>	<i>yes</i>	<i>acc</i>
11	<i>med</i>	<i>high</i>	<i>no</i>	<i>acc</i>
12	<i>med</i>	<i>high</i>	<i>yes</i>	<i>acc</i>
13	<i>high</i>	<i>low</i>	<i>no</i>	<i>acc</i>
14	<i>high</i>	<i>low</i>	<i>yes</i>	<i>acc</i>
15	<i>high</i>	<i>med</i>	<i>no</i>	<i>acc</i>
16	<i>high</i>	<i>med</i>	<i>yes</i>	<i>acc</i>
17	<i>high</i>	<i>high</i>	<i>no</i>	<i>good</i>
18	<i>high</i>	<i>high</i>	<i>yes</i>	<i>good</i>

Table 4.1: Table 2.1 after adding attribute C , $S_C = \{no, yes\}$.

4.1.2 Duplicate an attribute

Pick some attribute $x_s \in X$, $s \in [1, n]$ to be *duplicated*. We create a new attribute x' with $S' = S_s$, and insert it into X . Without loss of generality we insert it at the end (position $q = n + 1$), giving the new decision space

$$DS' = \langle X', y, \mathbf{S}', S_y \rangle, \quad X' = \{x_1, \dots, x_n, x'\}, \quad \mathbf{S}' = S_1 \times \dots \times S_n \times S'$$

Since $|S'| = |S_s| = k_s$, the table grows by a factor k_s : $|\mathbf{S}'| = |\mathbf{S}| \cdot k_s$. Every original condition $\mathbf{v} \in \mathbf{S}$ splits into k_s new conditions

$$\mathbf{v}' = \mathbf{v} \circ v', \quad v' \in S'$$

Rule construction. Unlike plain “Add/Insert attribute”, here the new attribute’s scale is not “unknown”; it is a copy of x_s ’s scale, so we know how the new attribute should relate to the existing value v_s . We preserve information only where the new attribute value agrees with the value the duplicated attribute already had in that rule. Elsewhere, we have no basis for a value and leave the rule non-entered.

Formally, let $w = \mathbf{v}_s$ (the s -th component of $\mathbf{v}$). Then

$$E(R') = \{r_{\mathbf{v}'} : \mathbf{v}' \rightarrow o(r_{\mathbf{v}}) \mid r_{\mathbf{v}} \in E(R), w = \mathbf{v}_s, \mathbf{v}' = \mathbf{v} \circ w\}$$

All rules $r_{\mathbf{v}'}$ with $\mathbf{v}'_{n+1} \neq \mathbf{v}'_s$ remain non-entered; their outcomes are subsequently obtained by applying the principle of dominance to $E(R')$.

Algorithm. The “Duplicate attribute” pseudocode is shown in Algorithm 3. The inner loop resembles the one in Algorithm 2 (k_s copies per original rule), but the entered flag is now set selectively, only when $v'_s = v_s$, rather than on all k_s copies.

Algorithm 3 Duplicate an input attribute

```
function DuplicateInput(R, X, S, S_y, s):
    # R: original table (rules v -> o, with entered flags)
    # X, S: input attributes and their scales
    # S_y: output scale
    # s: index of attribute to duplicate

    X' ← X + [X[s]]          # append copy of attribute x_s
    S' ← S + [S[s]]          # its scale equals S_s

    R' ← Init(S', S_y)       # new empty table over S'
                                # all rules initially non-entered

    for each v in R:
        rule ← R[v]
        for each value v_s in S[s]: # for all scale values
            v' ← Concatenate(v, v_s)
            if entered(rule) and v_s == v'[s]:
                R'[v'] ← (outcome = o(rule), entered = true)
            # else: leave R'[v'] non-entered and handle it below

    R' ← CompleteByDominance(R')
    return R'
```

Proof of information preservation. We show two things: (a) the transformation exactly preserves the original table’s information, and (b) it preserves monotonicity.

Information preservation (diagonal recovery). Define the *diagonal restriction* of R' as the sub-table obtained by fixing $x' = x_s$, i.e. selecting all $\mathbf{v}' = \mathbf{v} \circ v_s$ with $\mathbf{v}'_{n+1} = \mathbf{v}'_s$. By construction, for every $r_{\mathbf{v}} \in E(R)$ the rule $r_{\mathbf{v} \circ v_s} \in E(R')$ has $o(r_{\mathbf{v} \circ v_s}) = o(r_{\mathbf{v}})$. Hence

$$R'|_{x'=x_s} = R$$

the original table can be read off exactly from the diagonal slice of R' . No entered rule’s outcome is altered or lost.

Monotonicity is preserved. Assume R satisfies the principle of dominance. Let $r'_{a'}, r'_{b'} \in E(R')$ be two distinct entered rules of R' , with $a' = a \circ a_s$ and $b' = b \circ b_s$, where $r_a, r_b \in E(R)$ (only diagonal rules are entered by construction). We must show: if $a' \succ b'$ then $o(r'_{a'}) \succeq o(r'_{b'})$.

Compare a' and b' component-wise. For $i = 1, \dots, n$ the comparison is identical to comparing a_i and b_i . The extra $(n+1)$ -th component compares $a'_{n+1} = a_s$ against $b'_{n+1} = b_s$ — but a_s, b_s

are already the s -th components of a, b , which were already compared among $i = 1, \dots, n$. So the $(n+1)$ -th coordinate contributes no new information: it is weakly/strictly ordered exactly as coordinate s already is. Consequently

$$a' \succ b' \iff a \succ b$$

Since $a \succ b \Rightarrow o(r_a) \succeq o(r_b)$ (as R is monotone) and $o(r'_{a'}) = o(r_a)$, $o(r'_{b'}) = o(r_b)$, we get $o(r'_{a'}) \succeq o(r'_{b'})$ whenever $a' \succ b'$. Hence $E(R')$ satisfies the principle of dominance.

Because $E(R')$ is monotone, the subsequent dominance-based assignment of the non-entered (off-diagonal) rules to $I(S_y)$ is well defined, exactly as assumed for any standard table with $E \neq \emptyset$. $\square$

Example. Take Table 2.1 and duplicate attribute B , producing $B' = B$ with the same scale $S_{B'} = S_B$. New decision space is:

$$X' = (A, B, B'), \quad \mathbf{S}' = S_A \times S_B \times S_{B'}, \quad |\mathbf{S}'| = 9 \times 3 = 27$$

Only rows where $B = B'$ retain the original, fully entered outcome (Table 4.2). All rows with $B \neq B'$ become non-entered (marked “-”) and are later bounded by dominance.

#	A	B	B'	Y
1	<i>low</i>	<i>low</i>	<i>low</i>	<i>bad</i>
2	<i>low</i>	<i>low</i>	<i>med</i>	-
3	<i>low</i>	<i>low</i>	<i>high</i>	-
4	<i>low</i>	<i>med</i>	<i>low</i>	-
5	<i>low</i>	<i>med</i>	<i>med</i>	<i>bad</i>
6	<i>low</i>	<i>med</i>	<i>high</i>	-
7	<i>low</i>	<i>high</i>	<i>low</i>	-
8	<i>low</i>	<i>high</i>	<i>med</i>	-
9	<i>low</i>	<i>high</i>	<i>high</i>	<i>acc</i>
10	<i>med</i>	<i>low</i>	<i>low</i>	<i>bad</i>
11	<i>med</i>	<i>low</i>	<i>med</i>	-
12	<i>med</i>	<i>low</i>	<i>high</i>	-
13	<i>med</i>	<i>med</i>	<i>low</i>	-
14	<i>med</i>	<i>med</i>	<i>med</i>	<i>acc</i>
15	<i>med</i>	<i>med</i>	<i>high</i>	-
16	<i>med</i>	<i>high</i>	<i>low</i>	-
17	<i>med</i>	<i>high</i>	<i>med</i>	-
18	<i>med</i>	<i>high</i>	<i>high</i>	<i>acc</i>
19	<i>high</i>	<i>low</i>	<i>low</i>	<i>acc</i>
20	<i>high</i>	<i>low</i>	<i>med</i>	-
21	<i>high</i>	<i>low</i>	<i>high</i>	-
22	<i>high</i>	<i>med</i>	<i>low</i>	-
23	<i>high</i>	<i>med</i>	<i>med</i>	<i>acc</i>
24	<i>high</i>	<i>med</i>	<i>high</i>	-
25	<i>high</i>	<i>high</i>	<i>low</i>	-
26	<i>high</i>	<i>high</i>	<i>med</i>	-
27	<i>high</i>	<i>high</i>	<i>high</i>	<i>good</i>

Table 4.2: Table 2.1 after duplicating attribute B .

The diagonal $B = B'$ reproduces Table 2.1 exactly. The off-diagonal rules are left for the dominance step. Duplicating an attribute flags the “disagreement” combinations ($B \neq B'$) for the modeler’s attention, while silently preserving everything already known about B ’s effect on the diagonal. This is why duplication is generally preferred to plain “Add/Insert”.

4.1.3 Delete an attribute

Given $DS = \langle X, y, \mathbf{S}, S_y \rangle$, deleting input attribute $x_q \in X$, $q \in [1, n]$ produces

$$X' = \{x_1, \dots, x_{q-1}, x_{q+1}, \dots, x_n\}, \quad \mathbf{S}' = S_1 \times \dots \times S_{q-1} \times S_{q+1} \times \dots \times S_n$$

Thus, $DS' = \langle X', y, \mathbf{S}', S_y \rangle$ with $|\mathbf{S}'| = |\mathbf{S}|/k_q$. Every original condition projects onto a reduced one by dropping the q -th coordinate:

$$\pi_q : \mathbf{S} \rightarrow \mathbf{S}', \quad \pi_q(\mathbf{v}) = \mathbf{v} \setminus q$$

This projection is k_q -to-1: exactly k_q different $\mathbf{v} \in \mathbf{S}$ collapse onto each $\mathbf{u} \in \mathbf{S}'$.

For each $\mathbf{u} \in \mathbf{S}'$, collect the outcomes contributed by original *entered* rules that map onto it:

$$J(\mathbf{u}) = \{o(r_{\mathbf{v}}) \mid r_{\mathbf{v}} \in E(R), \mathbf{v} \setminus q = \mathbf{u}\}$$

Since we restrict attention to standard tables, an entered rule must carry a single value $c \in S_y$. Deletion can therefore only safely produce a new entered rule where the collapsed original rules *unanimously agree*:

$$E(R') = \{r_{\mathbf{u}} : \mathbf{u} \rightarrow c \mid \mathbf{u} \in \mathbf{S}', J(\mathbf{u}) = \{c\}\}$$

If $J(\mathbf{u}) = \emptyset$ (no entered source rule) or $|J(\mathbf{u})| > 1$ (conflicting outcomes), $r_{\mathbf{u}}$ is left non-entered; its value is obtained by applying the principle of dominance.

This is the standard-table specialization of the general *join* operator $join : c \in V(S_y) \rightarrow V(S_y)$ applicable for other non-ordered outcome types (value sets and distributions):

$$join(J(\mathbf{u})) = \bigcup_{c \in J(\mathbf{u})} c$$

which would keep the whole union instead of discarding conflicting entries:

Algorithm. The “Delete attribute” pseudocode is shown in Algorithm 4.

Information preservation

(P1) Soundness. Every entered rule of R' reflects information that was unambiguously present in R : it is entered only where all values of the deleted attribute x_q led to the same outcome. Nothing is asserted where the original table disagreed, so no information is fabricated. This is the maximal amount of certain information that can survive removing a dimension.

Proof: Immediate from the definition of $J(\mathbf{u})$ and $E(R')$. $\square$

(P2) Monotonicity is preserved for fully entered tables. If R is monotone, and $R = E(R)$, then $E(R')$ satisfies the principle of dominance.

Proof. Let $r'_a : \mathbf{u}_a \rightarrow c_a$ and $r'_b : \mathbf{u}_b \rightarrow c_b$ be two entered rules of R' with $\mathbf{u}_a \succ \mathbf{u}_b$. By construction $J(\mathbf{u}_a) = \{c_a\}$ and $J(\mathbf{u}_b) = \{c_b\}$: every entered rule of R projecting to $\mathbf{u}_a$ (resp. $\mathbf{u}_b$) carries outcome c_a (resp. c_b), independently of the value taken by x_q . Choose a value $w \in S_q$ for which both $(\mathbf{u}_a, w)$ and $(\mathbf{u}_b, w)$ happen to be entered witnesses (such a common w exists whenever the table was fully entered, and more generally whenever the two projected groups’ witness sets overlap). Set $\mathbf{v}_a = \mathbf{u}_a \circ w$, $\mathbf{v}_b = \mathbf{u}_b \circ w \in \mathbf{S}$. Since $\mathbf{u}_a \succ \mathbf{u}_b$ on the $n-1$ shared attributes and the q -th coordinates are equal, $\mathbf{v}_a \succ \mathbf{v}_b$ in $\mathbf{S}$. Monotonicity of R then gives $o(r_{\mathbf{v}_a}) \succeq o(r_{\mathbf{v}_b})$, i.e. $c_a \succeq c_b$, i.e. $o(r'_a) \succeq o(r'_b)$. Hence $E(R')$ satisfies the principle of dominance, and the subsequent dominance-based filling of its non-entered cells is well defined. $\square$

Algorithm 4 Delete an input attribute

```
function DeleteInput(R, X, S, S_y, q):  
    X' ← X without x[q]  
    S' ← S without S[q]  
    groups ← empty map from u in S' to list of outcomes  
  
    for each v in R:  
        rule ← R[v]  
        if entered(rule):  
            u ← v without q-th component # projection  
            groups[u].append(o(rule))  
  
    R' ← Init(S', S_y) # new empty table over S' and S_y  
    for each u in S':  
        if u in groups:  
            J ← set(groups[u])  
            if |J| == 1:  
                R'[u] ← (outcome = single value from J, entered = true)  
            # else: conflicting source outcomes -> leave non-entered  
        # else: no source outcomes -> leave non-entered  
  
    R' ← CompleteByDominance(R')  
    return R'
```

(P3) Monotonicity is not always preserved. *Proof* by counterexample: Consider input attributes $\{A, Q\}$, $S_A = \{low, high\}$, $S_Q = \{lo, hi\}$, $S_y = \{0, 1\}$, with entered rules $(high, lo) \rightarrow 0$ and $(low, hi) \rightarrow 1$. This is monotone, because the two rules are incomparable. After deleting Q , both A -rows are entered with $high \rightarrow 0$ and $low \rightarrow 1$, which violates dominance. $\square$

#	A	B	Y
1	low	low	bad
2	low	med	bad
3	low	high	bad
4	med	low	bad
5	med	med	acc
6	med	high	acc
7	high	low	bad
8	high	med	acc
9	high	high	good

Table 4.3: Example of a fully entered, monotone decision table, adapted from Table 2.1.

Example. Take Table 4.3 and delete attribute B , producing Table 4.4. The 9 original rules collapse into three groups of 3 rules, each group corresponding to one R' rule. Only the first group contains equal Y outcomes and thus survives as an entered rule. The remaining two cells are non-entered due to value conflicts, to be subsequently bounded by dominance.

#	A	$J(\mathbf{u})$ (Y outcomes seen)	Y'
1	<i>low</i>	<i>bad, bad, bad</i>	entered: <i>bad</i>
2	<i>med</i>	<i>bad, acc, acc</i>	conflict: non-entered
3	<i>high</i>	<i>bad, acc, good</i>	conflict: non-entered

Table 4.4: Result of deleting attribute B from Table 4.3.

4.1.4 Exchange/Move input attributes

These two transformations of inputs are primarily of practical importance. They do not change any decision rules, but just reshuffle the positions of input attributes, affecting the order of columns and rules in the decision table. *Exchanging* means exchanging positions of two attributes in X . *Moving* means repositioning some attribute $x_s \in X$ to a new position $q \in [1, n], q \neq s$, shifting positions of other attributes if necessary.

Both transformations act only on the *positions* of input attributes in X . They do not add or remove any attributes, and do not touch any scale S_i or S_y . Consequently, they never invent information. Unlike “Add”, “Duplicate” or “Delete”, they require no *join* and no dominance-based inference at all.

Formally, a *position permutation* is a bijection $\sigma : 1, \dots, n \rightarrow 1, \dots, n$. It reorders attributes to

$$X' = \{x_{\sigma(1)}, x_{\sigma(2)}, \dots, x_{\sigma(n)}\}, \quad \mathbf{S}' = S_{\sigma(1)} \times \dots \times S_{\sigma(n)}$$

giving $DS' = \langle X', y, \mathbf{S}', S_y \rangle$. Note that n and individual S_i ’s are unchanged, only their positions.

Then, σ induces a vector mapping $\varphi_\sigma : \mathbf{S} \rightarrow \mathbf{S}'$:

$$\varphi_\sigma(\mathbf{v}) = \mathbf{v}', \quad v'_j = v_{\sigma(j)}, \quad j = 1, \dots, n$$

The transformed table is

$$R' = \{r_{\mathbf{v}'} : \mathbf{v}' \rightarrow o(r_{\mathbf{v}}) \mid r_{\mathbf{v}} \in R, \mathbf{v}' = \varphi_\sigma(\mathbf{v}), \}, \quad entered(r_{\mathbf{v}'}) = entered(r_{\mathbf{v}})$$

Exchange (p, q) swaps two positions $p, q \in [1, n], p \neq q$:

$$\sigma_{\text{exch}}(j) = \begin{cases} q, & j = p, \\ p, & j = q, \\ j, & \text{otherwise.} \end{cases}$$

Move ($s \rightarrow q$) relocates the attribute at position s to position q ($s \neq q$), shifting the attributes between s and q by one:

$$\sigma_{\text{move}}(j) = \begin{cases} s, & j = q, \\ j + 1, & s \leq j < q, \\ j, & j < s \text{ or } j > q \end{cases} \quad (s < q), \quad \sigma_{\text{move}}(j) = \begin{cases} s, & j = q, \\ j - 1, & q < j \leq s, \\ j, & j < q \text{ or } j > s \end{cases} \quad (s > q)$$

Both σ_{exch} and σ_{move} are permutations of $1, \dots, n$, hence special cases of the general φ_σ construction above.

Algorithm. Outlined as Algorithm 5.

Information preservation and Example. Trivial, omitted.

Algorithm 5 Exchange or Move input attributes

```
function PermuteInputs(R,  $\sigma$ ):  
  R'  $\leftarrow$  Init(R)  
  for each rule r = ( $v \rightarrow o$ ) in R:  
    for j = 1 to n:  
      v'[j]  $\leftarrow$  v[ $\sigma(j)$ ]  
    R'[v']  $\leftarrow$  (outcome = o, entered = entered(r))  
  return R'  
  
function ExchangeInputs(R, p, q):  
   $\sigma \leftarrow$  identity(1..n)  
   $\sigma[p] \leftarrow q$   
   $\sigma[q] \leftarrow p$   
  return PermuteInputs(R,  $\sigma$ )  
  
function MoveInput(R, s, q):  
   $\sigma \leftarrow$  identity(1..n)  
  if s < q:  
    for j  $\leftarrow$  s to q-1:  $\sigma[j] \leftarrow j+1$   
     $\sigma[q] \leftarrow s$   
  if s > q:  
    for j = q+1 to s:  $\sigma[j] \leftarrow j-1$   
     $\sigma[q] \leftarrow s$   
  return PermuteInputs(R,  $\sigma$ )
```

4.2 Transformations due to changes of an input attribute scale

Transformations in this section are induced by a change of some value scale $S_s \in \mathbf{S}, s \in [1, n]$. This includes adding some value to and deleting some value from S_s , duplicating and merging values, and changing the position of values.

In all cases, the original set of attributes X stays intact. However, the associated scale S_s changes to S'_s . The new decision space is $DS' = \langle X, y, \mathbf{S}', S_y \rangle$, where

$$\mathbf{S}' = S_1 \times S_2 \times \cdots \times S'_s \times \cdots \times S_n$$

4.2.1 Add/Insert input value

Consider one input attribute $x_s \in X, s \in [1, n]$, with scale

$$S_s = \{v_{s,1}, v_{s,2}, \dots, v_{s,k_s}\}$$

The transformation *inserts* a new value $v^* \notin S_s$ at position $p \in [1, k_s + 1]$, giving the new scale

$$S'_s = \{v_{s,1}, \dots, v_{s,p-1}, v^*, v_{s,p}, \dots, v_{s,k_s}\}, \quad k'_s = k_s + 1$$

$p = k_s + 1$ is the special case of *adding* v^* at the end.

If S_s is ordered, S'_s is ordered too, keeping the relative order of the old values unchanged and placing v^* so that $v_{s,p-1} \prec v^* \prec v_{s,p}$ (boundary terms omitted when $p = 1$ or $p = k_s + 1$).

The new decision space is $DS' = \langle X, y, \mathbf{S}', S_y \rangle$ with

$$\mathbf{S}' = S_1 \times \cdots \times S'_s \times \cdots \times S_n, \quad |\mathbf{S}'| = |\mathbf{S}| \cdot \frac{k_s + 1}{k_s}$$

Since $S_s \subset S'_s$, every old condition $\mathbf{v} \in \mathbf{S}$ is also a valid condition in S' . We can reuse the old entered rules unchanged and simply mark all new conditions, where $x_s = v^*$, as non-entered:

$$E(R') = \{r' = (\mathbf{v} \rightarrow o) \mid r = (\mathbf{v} \rightarrow o) \in E(R)\}$$

$$\text{entered}(r_{\mathbf{w} \circ_s v^*}) = \text{false} \quad \text{for every } \mathbf{w} \in S_1 \times \dots \times S_{s-1} \times S_{s+1} \times \dots \times S_n$$

where $\mathbf{w} \circ_s v^*$ denotes the vector obtained by inserting v^* at position s into $\mathbf{w}$. As always for SDTs, once $E(R')$ is fixed, the outcomes of all non-entered rules of R' are processed according to the dominance principle.

Note that this is the “blind” counterpart of “Duplicate input value” (section 4.2.2): nothing is known about v^* a priori, so no entered rule can involve it; only the dominance closure gives it (partial) information, inherited from its neighbors $v_{s,p-1}$ and $v_{s,p}$ in S'_s .

Algorithm. Pseudocode is shown in Algorithm 6.

Algorithm 6 Add/Insert value to an input attribute

```
function AddInsertInputValue(R, s, v_star, p):
    # R: SDT over DS = <X, y, S, S_y>, with E(R) its entered rules
    # s: index of the attribute whose scale is extended
    # v_star: new scale value, v_star not in S_s
    # p: insertion position, 1 <= p <= k_s + 1

    S_s' ← insert v_star into S_s at position p # |S_s'| = k_s + 1
    S' ← S_1 x ... x S_s' x ... x S_n           # replaces S_s with S_s'
    R' ← Init(S', S_y)

    # 1. carry over all old entered rules unchanged
    for each r = (v -> o) in E(R):
        R'[v] ← (outcome = o, entered = true)

    # 2. create the new (non-entered) rules for x_s = v_star
    for each w in S_1 x ... x S_{s-1} x S_{s+1} x ... x S_n:
        v' ← insert v_star into w at position s
        R'[v] ← (outcome = undefined, entered = false)

    R' ← CompleteByDominance(R')
    return R'
```

Information preservation

(P1) Completeness. R' is a valid SDT over S' : it contains exactly one rule per $\mathbf{v}' \in S'$.

Proof. S' can be partitioned as $S' = \{\mathbf{v}' \in S' : \mathbf{v}'_s \in S_s\} \dot{\cup} \{\mathbf{v}' \in S' : \mathbf{v}'_s = v^*\}$. The identity map on coordinates $i \neq s$ is a bijection between the first part and $\mathbf{S}$ (step 1 of the pseudocode enumerates it exactly, once per $\mathbf{v} \in \mathbf{S}$). The second part is exactly $\{\mathbf{w} \circ_s v^* : \mathbf{w} \in \prod_{i \neq s} S_i\}$, enumerated once each in step 2. Hence the union produced by the algorithm covers S' with no duplicates and no omissions. $\square$

(P2) Information preservation. For every $r = (\mathbf{v} \rightarrow o) \in E(R)$, the corresponding rule $r' \in R'$ has $\mathbf{v}(r') = \mathbf{v}(r)$, $o(r') = o(r)$ and $\text{entered}(r') = \text{true}$.

Proof. Immediate from step 1: entered rules are copied verbatim, since $S'_i = S_i$ for $i \neq s$ and $S_s \subset S'_s$ leaves the meaning of every old value $v_s \in S_s$ unchanged. $\square$

(P3) No new violations among old rules. Let r_a, r_b be two rules built from old values ($\mathbf{v}_{a,s}, \mathbf{v}_{b,s} \in S_s$). Then $r_a \succ r_b$ in $\mathbf{S}'$ iff $r_a \succ r_b$ in $\mathbf{S}$.

Proof. Dominance is defined pointwise via $\succeq$ on each coordinate (background definition of $\succ$). For $i \neq s$ the scales, hence the comparison, are unchanged. For $i = s$, by construction S'_s preserves the relative order of the elements of S_s (only v^* is newly inserted between two consecutive old values, or at an end); hence for any $v_{s,i}, v_{s,j} \in S_s$, $v_{s,i} \preceq v_{s,j}$ holds in S_s iff it holds in S'_s . Consequently the truth value of $r_a \succ r_b$ is identical in $\mathbf{S}$ and $\mathbf{S}'$. $\square$

Corollary. If $E(R)$ satisfies the principle of dominance in R (i.e. R is monotone), then the embedded set $E(R')$ (P2) satisfies the principle of dominance in R' as well; by (P3) no dominance relation between old rules is created or destroyed, so no old pair can newly violate the principle. Hence “CompleteByDominance” is applied to a consistent entered set and is well defined; it can only add information (interval bounds) for the newly introduced conditions with $x_s = v^*$, and, because those bounds are computed strictly from S'_s -neighbors $v_{s,p-1}, v_{s,p}$ and the (unchanged) rest of the vector, it never contradicts any previously entered outcome. This establishes that “Add/Insert input value” is a correct, information-preserving transformation. $\square$

Example. Take $X = \{A, B, C\}$, all with scale *low, high* ($low \prec high$), and output scale $S_y = bad, acc, good$ ($bad \prec acc \prec good$). Let R (Table 4.5) be fully entered ($E(R) = R$), with outcome given by the number of *high* values among A, B, C ($0 \rightarrow bad, 1 \rightarrow bad, 2 \rightarrow acc, 3 \rightarrow good$). This R is monotone.

#	A	B	C	Y
1	low	low	low	bad
2	low	low	high	bad
3	low	high	low	bad
4	low	high	high	acc
5	high	low	low	bad
6	high	low	high	acc
7	high	high	low	acc
8	high	high	high	good

Table 4.5: Decision table to illustrate decision table transformations.

Now apply “Add/Insert input value” to $x_s = C$: insert $v^* = med$ at position $p = 2$, so $S'_C = \{low, med, high\}$, $k'_C = 3$, and $|\mathbf{S}'| = 2 \cdot 2 \cdot 3 = 12$. The algorithm copies the 8 old rows unchanged and with entered status. 4 new rows are created for $C = med$, initially non-entered/undefined. “CompleteByDominance” then applies dominance closure. The result is shown in Table 4.6.

4.2.2 Duplicate input value

Consider attribute $x_s \in X$, $s \in [1, n]$, with scale

$$S_s = \{v_{s,1}, v_{s,2}, \dots, v_{s,k_s}\}$$

Choose a value $v = v_{s,d} \in S_s$, $d \in [1, k_s]$, to be *duplicated*. The transformation creates a new value $v' \notin S_s$, inserted immediately adjacent to v (at position $d + 1$):

$$S'_s = \{v_{s,1}, \dots, v_{s,d}, v', v_{s,d+1}, \dots, v_{s,k_s}\}$$

As before, $k'_s = k_s + 1$, $DS' = \langle X, y, \mathbf{S}', S_y \rangle$ and

$$\mathbf{S}' = S_1 \times \dots \times S'_s \times \dots \times S_n, \quad |\mathbf{S}'| = |\mathbf{S}| \cdot \frac{k_s + 1}{k_s}$$

#	A	B	C	Y'
1	low	low	low	bad (entered)
2	low	low	med	[bad, bad] = bad
3	low	low	high	bad (entered)
4	low	high	low	bad (entered)
5	low	high	med	[bad, acc]
6	low	high	high	acc (entered)
7	high	low	low	bad (entered)
8	high	low	med	[bad, acc]
9	high	low	high	acc (entered)
10	high	high	low	acc (entered)
11	high	high	med	[acc, good]
12	high	high	high	good(entered)

Table 4.6: Table 4.5 after inserting value *med* to attribute *C*.

The defining difference from “Add/Insert input value” is *semantic*: v' is meant to behave exactly as v , thus every rule that would use v' is required to reproduce the outcome of the corresponding rule that uses v . This holds not just for entered rules, but as a general requirement on the transformed table. Formally, the new entered set $E(R')$ is the union of unchanged old rules

$$\{r' = (\mathbf{v} \rightarrow o) \mid r = (\mathbf{v} \rightarrow o) \in E(R)\}$$

and duplicated rules, where $\mathbf{v}[x_s := v']$ denotes $\mathbf{v}$ with its s -th component replaced by v' :

$$\{r' = (\mathbf{v}[x_s := v'] \rightarrow o) \mid r = (\mathbf{v} \rightarrow o) \in E(R), \mathbf{v}_s = v\}$$

In other words, every entered rule whose condition uses $x_s = v$ is copied to an equally entered rule with $x_s = v'$, keeping the same outcome. All remaining non-entered conditions of R' are then completed by the principle of dominance. As shown below, this closure automatically reproduces the same outcome for v' -rules as for the corresponding v -rules even where R was itself incomplete.

Algorithm. Pseudocode is shown in Algorithm 7. Steps 1 and 2 are formulated sequentially, but can be combined within a single loop for efficiency.

Information preservation

(P1) Completeness. R' is a valid SDT over $\mathbf{S}'$ (one rule per condition). *Proof:* Identical as in section 4.2.1. $\square$

(P2) Information preservation of old rules. For $r = (\mathbf{v} \rightarrow o) \in E(R)$, the copy $r' \in R'$ has $\mathbf{v}(r') = \mathbf{v}(r)$, $o(r') = o(r)$, $entered(r') = \text{true}$. *Proof:* Immediate from step 1. $\square$

(P3) Monotonicity is preserved. If $E(R)$ satisfies the principle of dominance, then so does $E(R')$. In particular, every duplicated rule is consistent with all other entered rules, and $o(r') = o(r)$ holds by construction.

Proof. Let $\kappa : S' \rightarrow S$ replace v' by v in the s -th component of a condition and leave everything else unchanged. By steps 1 and 2, every entered rule $a \in E(R')$ satisfies $\kappa(a) \in E(R)$ and $o(a) = o(\kappa(a))$. Moreover, κ is order-preserving, $a \succeq b \Rightarrow \kappa(a) \succeq \kappa(b)$, because v' is adjacent to v (and order-equivalent to it): every value above (below) v' is above (below) v . For an unordered x_s this is trivial. Now let $a, b \in E(R')$ with $a \succ b$. Then $\kappa(a) \succeq \kappa(b)$. If $\kappa(a) = \kappa(b)$, then $o(a) = o(b)$. Otherwise $\kappa(a) \succ \kappa(b)$ are entered rules of $E(R)$, hence $o(a) = o(\kappa(a)) \succeq o(\kappa(b)) = o(b)$. $\square$

Algorithm 7 Duplicate value of an input attribute

```
function DuplicateInputValue(R, s, v, p):  
    # R: SDT over DS = <X, y, S, S_y>, with E(R) its entered rules  
    # s: index of the attribute whose scale is extended  
    # v: value in S_s to duplicate  
    # p: insertion position for the duplicate (adjacent to v)  
  
    v' ← new value, v' not in S_s  
    S_s' ← insert v' into S_s at position p, with v' ~ v  
    S' ← S_1 x ... x S_s' x ... x S_n      # replaces S_s with S_s'  
    R' ← Init(S', S_y)  
  
    # 1. carry over all old entered rules unchanged  
    for each r = (c → o) in E(R):  
        R'[c] ← (outcome = o, entered = true)  
  
    # 2. duplicate every entered rule that used x_s = v  
    for each r = (c → o) in E(R):  
        if c[s] == v:  
            w ← c with component s replaced by v'  
            R'[w] ← (outcome = o, entered = true)  
  
    R' ← CompleteByDominance(R')  
    return R'
```

(Corollary) Full-entry preservation. If R is fully entered ($E(R) = R$), then R' constructed above is fully entered as well; every condition in $\mathbf{S}'$ is covered by either step 1 or step 2, with no residual undefined/interval outcomes. This is the key qualitative difference from “Add/Insert input value”, where new rows are necessarily left non-entered and at best resolved to intervals.

Example. Take again Table 4.5 and duplicate $v = \text{high}$ of $x_s = C$: create $v' = \text{high2}$ and insert it adjacent to high , with $\text{high} \simeq \text{high2}$, so $S'_C = \{\text{low}, \text{high}, \text{high2}\}$, $|\mathbf{S}'| = 2 \cdot 2 \cdot 3 = 12$. Table 4.7 shows the result of duplication.

4.2.3 Delete input value

Consider $x_s \in X$ with scale $S_s = \{v_{s,1}, \dots, v_{s,k_s}\}$, $k_s > 1$. Choose some $v \in S_s$ and *delete* it, so that

$$S'_s = S_s \setminus \{v\}, \quad k'_s = k_s - 1, \quad \mathbf{S}' = S_1 \times \dots \times S'_s \times \dots \times S_n, \quad |\mathbf{S}'| = |\mathbf{S}| \cdot \frac{k_s - 1}{k_s}$$

Notice that deleting the only value of a scale ($k_s = 1$) is inadmissible, since it would violate $|S_i| > 0$ for a decision space.

All rules whose condition uses $x_s = v$ are removed; there is no longer a place for them in $\mathbf{S}'$. Everything else (rules, entered or not) is untouched at the level of conditions. Formally, the surviving entered set is

$$E(R') = \{r = (\mathbf{v} \rightarrow o) \in E(R) : \mathbf{v}_s \neq v\} = E(R) \cap R_{x_s \neq v}$$

As always for an SDT, the remaining non-entered rules are completed by dominance.

#	<i>A</i>	<i>B</i>	<i>C</i>	<i>Y'</i>
1	<i>low</i>	<i>low</i>	<i>low</i>	<i>bad</i> (entered)
2	<i>low</i>	<i>low</i>	<i>high</i>	<i>bad</i> (entered)
3	<i>low</i>	<i>low</i>	high2	bad (entered, duplicated)
4	<i>low</i>	<i>high</i>	<i>low</i>	<i>bad</i> (entered)
5	<i>low</i>	<i>high</i>	<i>high</i>	<i>acc</i> (entered)
6	<i>low</i>	<i>high</i>	high2	acc (entered, duplicated)
7	<i>high</i>	<i>low</i>	<i>low</i>	<i>bad</i> (entered)
8	<i>high</i>	<i>low</i>	<i>high</i>	<i>acc</i> (entered)
9	<i>high</i>	<i>low</i>	high2	acc (entered, duplicated)
10	<i>high</i>	<i>high</i>	<i>low</i>	<i>acc</i> (entered)
11	<i>high</i>	<i>high</i>	<i>high</i>	<i>good</i> (entered)
12	<i>high</i>	<i>high</i>	high2	good (entered, duplicated)

Table 4.7: Table 4.5 after duplicating value *high* of attribute *C*.

Note that this transformation, unlike “Add/Insert” and “Duplicate input attribute value”, only ever removes rules. It introduces no new conditions. It is therefore “straightforward” as far as which rules disappear, but because entered rules with $x_s = v$ may have been the dominance neighbors that pinned down the outcomes of other surviving non-entered rules, deleting v can legitimately change (i.e., widen) the derived outcomes of rules that remain in the table, even though no rule’s condition changes. Only entered rules are guaranteed to keep their exact outcome. Non-entered outcomes are recomputed, not merely inherited.

Algorithm. Pseudocode is shown in Algorithm 8.

Algorithm 8 Delete value of an input attribute

```

function DeleteInputValue(R, s, v):
    # R: SDT over DS = <X, y, S, S_y>, with E(R) its entered rules
    # s: index of the attribute whose scale loses a value
    # v: value in S_s to delete

    assert |S_s| > 1 # deleting the last value is inadmissible

    S_s' ← S_s \ { v } # k_s' = k_s - 1
    S' ← S_1 x ... x S_s' x ... x S_n
    R' ← Init(S', S_y)

    for each r = (c -> o) in E(R):
        if c[s] != v: # rules and entered rules with x_s = v are dropped
            R'[c] ← (outcome = o, entered = true)

    R ← CompleteByDominance(R')
    return R'

```

Information preservation

(P1) Completeness. R' contains exactly one rule per element of S' .

Proof. $\mathbf{S}' = \{\mathbf{v} \in \mathbf{S} : \mathbf{v}_s \neq v\}$ under the natural identification of $S_1 \times \dots \times S'_s \times \dots \times S_n$ with this subset of $\mathbf{S}$. The construction enumerates precisely these conditions (via E' for the entered ones, and via “CompleteByDominance” for the rest), so R' is a complete, distinct-condition table over $\mathbf{S}'$. $\square$

(P2) Exact preservation of surviving entered rules. For every $r = (\mathbf{v} \rightarrow o) \in E(R)$ with $\mathbf{v}_s \neq v$, the same rule appears unchanged in $E(R')$, i.e. $\mathbf{v}(r') = \mathbf{v}(r)$, $o(r') = o(r)$, $\text{entered}(r') = \text{true}$. *Proof:* Immediate from the pseudocode: such rules are copied verbatim, and no other step of the algorithm modifies an already-entered rule’s outcome. $\square$

(P3) Monotonicity is inherited, not just preserved. If $E(R)$ satisfies the principle of dominance in $\mathbf{S}$, then $E(R')$ satisfies it in $\mathbf{S}'$.

Proof. $E(R') \subseteq E(R)$ (P2), and for any $r_a, r_b \in E(R')$ (hence $\subseteq \mathbf{S}' \subset \mathbf{S}$ under the natural embedding, since $S'_i = S_i$ for $i \neq s$ and $S'_s \subset S_s$), the dominance relation ‘ $\succ$ ’ between r_a and r_b computed in $\mathbf{S}'$ coincides with the one computed in $\mathbf{S}$ (a sub-order of a totally ordered scale restricted to a subset is exactly the restriction of that order). Since $r_a \succ r_b \Rightarrow o(r_a) \succeq o(r_b)$ already held in R (monotonicity of $E(R)$), it continues to hold for the same pair in R' . Restriction to a subset of pairs of an already-monotone relation cannot create a violation. $\square$

Remark. Non-entered outcomes are not guaranteed preserved. Unlike (P2–3), nothing guarantees that a non-entered rule $r' \in R' \setminus E(R')$ keeps its previous outcome from R . Its dominance bounds are recomputed from $E(R')$, which may have lost exactly the entered rule(s) with $x_s = v$ that previously pinned them; the resulting interval can only stay the same or become weaker (wider, never tighter or *undefined*, since $E(R') \subseteq E(R)$ implies the set of admissible dominance constraints can only shrink. This is illustrated below.

Example. Continue the running example: $A, B \in \{low, high\}$, $C \in \{low, med, high\}$ ($low \prec med \prec high$), $S_y = \{bad, acc, good\}$. Take Table 4.6 produced by “Add/Insert input value”, which contains entered rules only for $C \in \{low, high\}$, with $C = med$ left non-entered. Then apply “Delete input value” $v = low$ on C , giving $S'_C = \{med, high\}$, $|\mathbf{S}'| = 2 \cdot 2 \cdot 2 = 8$. The 4 rows with $C = low$ are dropped (all were entered). The result is presented in Table 4.8. The surviving entered set consists of rules numbered 2, 4, 6 and 8. Recomputing all non-entered rules (1, 3, 5 and 7) leaves the former three rules unchanged, but widens rule 7 from $[acc, good]$ to $[bad, good]$.

#	A	B	C	Y'
1	low	low	med	bad (non-entered, unchanged)
2	low	low	high	bad (entered)
3	low	high	med	[bad, acc] (non-entered, unchanged)
4	low	high	high	acc (entered)
5	high	low	med	[bad, acc] (non-entered, unchanged)
6	high	low	high	acc (entered)
7	high	high	med	[bad, good] (non-entered, widened)
8	high	high	high	good(entered)

Table 4.8: Table 4.6 after deleting value *low* from attribute *C*.

4.2.4 Merge input values

Consider $x_s \in X$ with scale $S_s = \{v_{s,1}, \dots, v_{s,k_s}\}$, $k_s > 1$. Choose two distinct values $v_a, v_b \in S_s$ (normally, but not necessarily, adjacent) and merge them into a single new value $v_{ab} \notin S_s$:

$$S'_s = (S_s \setminus \{v_a, v_b\}) \cup \{v_{ab}\}, \quad k'_s = k_s - 1, \quad \mathbf{S}' = S_1 \times \dots \times S'_s \times \dots \times S_n, \quad |\mathbf{S}'| = |\mathbf{S}| \cdot \frac{k_s - 1}{k_s}$$

Unlike “Delete input value” (section 4.2.3), where a value simply disappears, here the two pre-transformation conditions are *combined* rather than discarded. For every $\mathbf{w} \in \prod_{i \neq s} S_i$, the two old conditions $\mathbf{w} \circ_s v_a$ and $\mathbf{w} \circ_s v_b$ collapse onto the single new condition $\mathbf{w} \circ_s v_{ab}$. A new rule for it must reconcile whatever was known about both. Define the *join set*

$$J(\mathbf{w}) = \{o(r) : r \in E(R), \mathbf{v}(r) \in \mathbf{w} \circ_s v_a, \mathbf{w} \circ_s v_b\}, \quad |J(\mathbf{w})| \in \{0, 1, 2\}$$

Here, only entered pre-transformation rules contribute. A non-entered pre-transformation rule contributes nothing. The *join operator* for an interval-valued SDT outcome space is the tightest interval containing all observed values:

$$join(J) = \begin{cases} \text{(leave unassigned)}, & J = \emptyset \\ [\min J, \max J] \subseteq I(S_y), & J \neq \emptyset \end{cases}$$

assuming ordering by $\preceq$ on S_y . The merged rule is then

$$r'_{\mathbf{w}} = (\mathbf{w} \circ_s v_{ab} \rightarrow join(J(\mathbf{w})))$$

$$entered(r'_{\mathbf{w}}) = \text{true} \iff J(\mathbf{w}) \neq \emptyset \text{ and } \min J(\mathbf{w}) = \max J(\mathbf{w})$$

The merge yields an entered singleton outcome exactly when the two source facts agree, or only one existed. Conversely, it yields a genuine, non-entered interval when the two entered facts conflict. It yields no direct information (left non-entered, undetermined) when neither pre-transformation rule was entered. All conditions with $x_s \notin \{v_a, v_b\}$ carry over unchanged:

$$E(R') = \{r = (\mathbf{v} \rightarrow o) \in E(R) : \mathbf{v}_s \notin \{v_a, v_b\}\} \cup \{r'_{\mathbf{w}} : J(\mathbf{w}) \neq \emptyset, \min J(\mathbf{w}) = \max J(\mathbf{w})\}$$

Algorithm. Pseudocode is shown in Algorithm 9.

Algorithm 9 Merge two values of an input attribute

```
function MergeInputValues(R, s, v_a, v_b):
  # R: SDT over DS = <X, y, S, S_y>, with E(R) its entered rules
  # s: index of the attribute whose scale loses two values, gains one
  # v_a, v_b: distinct values in S_s to be merged

  v_ab ← new value, v_ab not in S_s
  S_s' ← { S_s \ {v_a, v_b} } ∪ { v_ab }
  # assuming v_ab takes the position of v_a, and v_b is removed
  S' ← S_1 x ... x S_s' x ... x S_n
  R' ← Init(S', S_y)

  # 1. carry over old entered rules untouched by the merge
  for each r = (v → o) in E(R):
    if v[s] != v_a and v[s] != v_b:
      R'[v] ← (outcome = o, entered = true)

  # 2. build the join for every merged condition
  for each w in S_1 x ... x S_{s-1} x S_{s+1} x ... x S_n:
    J ← {}
    for u in {v_a, v_b}:
      w_u ← w with w[s] replaced by u
      r_u ← R[w_u]
      if entered(r_u):
        J ← J ∪ { outcome(r_u) }

    v' ← w with v_ab inserted at position s
    if J != {}:
      lo ← min(J)
      hi ← max(J)
      if lo == hi: # enter singleton
        R'[v'] ← (outcome = lo, entered = true)
      # else: leave v' non-entered and resolve later
    # else if J == {}: leave v' non-entered

  R' ← CompleteByDominance(R')
  return R'
```

Information preservation

(P1) Completeness. R' has exactly one rule per element of $\mathbf{S}'$. *Proof:* As before: $\mathbf{S}'$ splits into $\{\mathbf{v}' \in \mathbf{S}' : \mathbf{v}'_s \neq v_{ab}\}$, in bijection with $\{\mathbf{v} \in \mathbf{S} : \mathbf{v}_s \notin \{v_a, v_b\}\}$ (step 1), and $\{\mathbf{v}' \in \mathbf{S}' : \mathbf{v}'_s = v_{ab}\}$, in bijection with $\prod_{i \neq s} S_i$ (one rule per $\mathbf{w}$, step 2/3). $\square$

(P2) Exact preservation of unmerged entered rules. For $r \in E(R)$ with $\mathbf{v}(r)_s \notin \{v_a, v_b\}$, the copy in $E(R')$ has identical condition, outcome, and *entered* = true. *Proof:* immediate from step 1. $\square$

(P3) join is the minimal information-preserving combination. For any $\mathbf{w}$ with $J(\mathbf{w}) \neq \emptyset$, $\text{join}(J(\mathbf{w})) = [\min J(\mathbf{w}), \max J(\mathbf{w})]$ is the smallest interval in $I(S_y)$ containing every value

of $J(\mathbf{w})$.

Proof. Any interval $[l, h]$ containing all of $J(\mathbf{w})$ must satisfy $l \preceq c \preceq h$ for every $c \in J(\mathbf{w})$, hence $l \preceq \min J(\mathbf{w})$ and $h \succeq \max J(\mathbf{w})$; so $[\min J, \max J] \subseteq [l, h]$ for every admissible $[l, h]$, i.e. it is the unique smallest such interval. $\square$

This shows that merge loses no more information than strictly necessary to reconcile v_a 's and v_b 's known facts into one condition, in keeping with the information-preservation principle; when $\min J = \max J$ nothing at all is lost, which is exactly the case the algorithm marks as entered.

(P4) Monotonicity is not always preserved. *Proof* by counterexample: Let a monotone decision table contain entered rules $(high, c2) \rightarrow 0$ and $(low, c3) \rightarrow 1$, which are incomparable and thus do not violate monotonicity. After merging $c2$ and $c3$, $(high, c23) \rightarrow 0$ is above $(low, c23) \rightarrow 1$, making the table non-monotone.

#	A	B	C	Y
1	low	low	low	bad
2	low	low	med	bad
3	low	low	high	acc
4	low	high	low	bad
5	low	high	med	acc
6	low	high	high	acc
7	high	low	low	bad
8	high	low	med	acc
9	high	low	high	acc
10	high	high	low	acc
11	high	high	med	acc
12	high	high	high	good

Table 4.9: Decision table to illustrate merging and copying attribute values.

(A, B)	$o(\cdot, med)$	$o(\cdot, high)$	$J(\mathbf{w})$	$join$	entered
(low, low)	bad	acc	$\{bad, acc\}$	$[bad, acc]$	no
$(low, high)$	acc	acc	$\{acc\}$	acc	yes
$(high, low)$	acc	acc	$\{acc\}$	acc	yes
$(high, high)$	acc	good	$\{acc, good\}$	$[acc, good]$	no

Table 4.10: Joining outcomes $C = \{med, high\}$ from Table 4.9.

#	A	B	C'	Y'
1	low	low	low	bad (entered)
2	low	low	med+high	$[bad, acc]$ (non-entered)
3	low	high	low	bad (entered)
4	low	high	med+high	acc (entered, sources agreed)
5	high	low	low	bad (entered)
6	high	low	med+high	acc (entered, sources agreed)
7	high	high	low	acc
8	high	high	med+high	$[acc, good]$ (non-entered, sources conflicted)

Table 4.11: Table 4.9 after merging values $C = \{med, high\}$.

Example. Take Table 4.9, defined with $A, B \in \{low, high\}$, $C \in \{low, med, high\}$, $S_y = \{bad, acc, good\}$, fully entered and monotone. Apply “Merge input values” to C , merging the adjacent values $v_a = med$, $v_b = high$ into $v_{ab} = med+high$: $S'_C = \{low, med+high\}$, $|S'| = 2 \cdot 2 \cdot 2 = 8$.

For each (A, B) , join the outcomes at $C = med$ and $C = high$, as shown in Table 4.10. This gives the resulting Table 4.11. Two of the four merged rules (4 and 6) collapse to an exact, entered value because the two source facts agreed. The other two (2 and 8) are first downgraded to the tightest interval spanning the conflicting facts and then post-processed by “CompleteByDominance”.

4.2.5 Copy input value

Consider $x_s \in X$ with scale $S_s = \{v_{s,1}, \dots, v_{s,k_s}\}$, $k_s > 1$. Choose a source value $v = v_{s,d}$ and a target value $u = v_{s,q}$, $q \neq d$. This transformation *copies* v onto u : every entered rule indexed by v is repeated for u , and the entered rules previously indexed by u are discarded.

Neither the attributes nor the scales change: $DS' = DS$. The value u keeps its name and its place in S_s ; in particular, it does not become order-equivalent to v , which distinguishes “Copy” from “Duplicate input value” (section 4.2.2). Outcomes are never combined, which distinguishes it from “Merge input values” (section 4.2.4). Only the entered rules of the slice $x_s = u$ are replaced:

$$E(R') = \{(\mathbf{w} \circ_s x \rightarrow o) \in E(R) : x \neq u\} \cup \{(\mathbf{w} \circ_s u \rightarrow o) : (\mathbf{w} \circ_s v \rightarrow o) \in E(R)\}$$

Here $\mathbf{w}$ ranges over $\prod_{i \neq s} S_i$ and $\mathbf{w} \circ_s x$ inserts x at position s . The correspondence of conditions (section 3.1.1) is $\varphi_\tau = \{(\mathbf{c}, \mathbf{c}) : c_s \neq u\} \cup \{(\mathbf{w} \circ_s v, \mathbf{w} \circ_s u)\}$ and ψ_τ is the identity; conditions with $x_s = u$ have no image.

Only entered rules are transferred. The outcomes of non-entered rules are *not* copied but recomputed by “CompleteByDominance”, because the entered rules around $\mathbf{w} \circ_s u$ differ from those around $\mathbf{w} \circ_s v$. Since rules are both removed (slice u) and added (slice u), the recomputed intervals may become narrower or wider.

Interposed values. For an ordered S_s let

$$I(v, u) = \{x \in S_s \setminus \{u\} : pos(v) < pos(x) \leq pos(u) \text{ or } pos(u) \leq pos(x) < pos(v)\}$$

be the set of values that lie between v and u (for a strictly ordered scale, the values strictly between positions d and q , so $I = \emptyset$ iff $|d - q| = 1$). For an unordered x_s define $I(v, u) = \emptyset$.

Relation to other transformations. “Copy” equals “Delete input value” u (section 4.2.3), followed by “Add/Insert input value” u at position q (section 4.2.1), followed by entering the copied rules (section 4.5). This is correct because “CompleteByDominance” depends only on the set of entered rules, so the intermediate completions do not matter. The composition with “Duplicate” is *not* equivalent.

Algorithm. Pseudocode is shown in Algorithm 10.

Algorithm 10 Copy an input attribute value

```
function CopyInputValue(DS, R, s, d, q):  
    # DS = <X, y, S, S_y>  
    # s: index of attribute x_s  
    # d: position of source value v  
    # q: position of target value u; d != q  
    v ← S_s[d]  
    u ← S_s[q]  
  
    # 1. keep all the original rules other than slice u  
    R' ← Init(DS)  
    for each (c → o) in E(R):  
        if c[s] != u:  
            R'[c] ← R[c]  
  
    # 2. copy the entered rules of slice v to slice u  
    for each (c → o) in E(R):  
        if c[s] == v:  
            c_u ← c with c[s] replaced by u  
            R'[c_u] ← (outcome = o, entered = true)  
  
    R' ← CompleteByDominance(R')  
    return R'
```

Information preservation

(P1) Well-formedness. $E(R')$ contains at most one rule per condition, hence R' is a valid SDT over DS . *Proof.* Conditions with $x_s \neq u$ come only from the first part of $E(R')$, conditions with $x_s = u$ only from the second part, and $\mathbf{w} \mapsto \mathbf{w} \circ_s u$ is injective. $\square$

(P2) Information preservation. Every entered rule of R not indexed by u is preserved exactly (same outcome, entered), and every entered rule indexed by v additionally appears for u . The only information lost is $\{r \in E(R) : \mathbf{v}(r)_s = u\}$. Hence the transformation is strictly information-preserving w.r.t. φ_τ and ψ_τ , and it is not reversible. *Proof.* Immediate from the definition of $E(R')$. $\square$

(P3) Monotonicity, sufficient condition. If $E(R)$ satisfies the principle of dominance and $I(v, u) = \emptyset$, then $E(R')$ satisfies it, so R' contains no *undefined* outcome.

Proof. Let $a = (\mathbf{c}_a \rightarrow o_a)$, $b = (\mathbf{c}_b \rightarrow o_b) \in E(R')$ with $\mathbf{c}_a \succeq \mathbf{c}_b$. Four cases:

- both old ($c_s \neq u$): both belong to $E(R)$.
- both new: $c_{a,s} = c_{b,s} = u$, so $\mathbf{c}_a = \mathbf{w}_a \circ_s u$, $\mathbf{c}_b = \mathbf{w}_b \circ_s u$ and $\mathbf{w}_a \succeq \mathbf{w}_b$. Their sources $\mathbf{w}_a \circ_s v \succeq \mathbf{w}_b \circ_s v$ lie in $E(R)$, hence $o_a \geq o_b$.
- a new, b old with $x = c_{b,s} \neq u$: x_s is ordered (otherwise a, b are incomparable), $\mathbf{w}_a \succeq \mathbf{w}_b$ and $\text{pos}(u) \geq \text{pos}(x)$. As $x \notin I$, $\text{pos}(v) < \text{pos}(x)$ is impossible, so $\text{pos}(v) \geq \text{pos}(x)$. Therefore $\mathbf{w}_a \circ_s v \succeq \mathbf{c}_b$, both in $E(R)$, and $o_a \geq o_b$.
- a old, b new: symmetric ($\text{pos}(x) \geq \text{pos}(u)$ and $x \notin I$ give $\text{pos}(x) \geq \text{pos}(v)$).

R' then has no *undefined* outcome by the lemma of section 3.2. $\square$

(P4) The condition is necessary, and where conflicts arise. If x_s is ordered, $I(v, u) \neq \emptyset$ and $|S_y| \geq 2$, there is a monotone R for which $E(R')$ violates the principle of dominance. For every R , each violating pair consists of a copied rule and a rule indexed by some $x \in I(v, u)$.

Proof. Take $x \in I$. If $\text{pos}(v) < \text{pos}(x) \leq \text{pos}(u)$, let the (fully entered) R be $o(\mathbf{c}) = 2$ if $\text{pos}(c_s) \geq \text{pos}(x)$ and 1 otherwise. It is monotone, $o(\mathbf{w} \circ_s v) = 1$ and $o(\mathbf{w} \circ_s x) = 2$, but $\mathbf{w} \circ_s u \succeq \mathbf{w} \circ_s x$ receives outcome $1 < 2$. The case $\text{pos}(u) \leq \text{pos}(x) < \text{pos}(v)$ is symmetric with $o = 2$ iff $\text{pos}(c_s) \geq \text{pos}(v)$. The second statement follows from the proof of (P3): the first two cases never fail, and the last two fail only if $\text{pos}(v) < \text{pos}(x)$ (resp. $\text{pos}(x) < \text{pos}(v)$), i.e. $x \in I$. $\square$

Example 1 (adjacent values). Reuse Table 4.9. Apply “Copy input value” $v = \text{med}$ (position $d = 2$) to position $q = 3$ (previously *high*). Every rule with $C = \text{high}$ is overwritten, both outcome and entered status, with the corresponding $C = \text{med}$ rule. Table 4.12 shows the result.

#	A	B	C	Y'
1	low	low	low	bad
2	low	low	med	bad
3	low	low	high	bad (was acc, overwritten)
4	low	high	low	bad
5	low	high	med	acc
6	low	high	high	acc (coincidentally unchanged)
7	high	low	low	bad
8	high	low	med	acc
9	high	low	high	acc (coincidentally unchanged)
10	high	high	low	acc
11	high	high	med	acc
12	high	high	high	acc (was good, overwritten)

Table 4.12: Table 4.9 after copying value $C = \text{med}$ to position 3 in S_C .

Example 2 (non-adjacent values). Copy $v = \text{low}$ ($d = 1$) onto $u = \text{high}$ ($q = 3$) in the same table. Now $I = \{\text{med}\}$, and the new rules with $C = \text{high}$ (rows 3, 6, 9, 12) get the outcomes *bad, bad, bad, acc*. Two conflicts appear: $(\text{low}, \text{high}, \text{high}) \rightarrow \text{bad}$ against $(\text{low}, \text{high}, \text{med}) \rightarrow \text{acc}$, and $(\text{high}, \text{low}, \text{high}) \rightarrow \text{bad}$ against $(\text{high}, \text{low}, \text{med}) \rightarrow \text{acc}$. In both pairs the rule with $C = \text{high}$ dominates but has a worse outcome.

4.2.6 Move input values

Consider $x_s \in X$ with ordered scale represented as a sequence $S_s = (v_{s,1}, \dots, v_{s,k_s})$, according to the interpretation in section 2.2.7. So far the order has always coincided with set membership; here the *order itself* is the object being changed. Choose $v = v_{s,d}$ and a target position $p \in [1, k_s]$, $p \neq d$. The *move* relocates v from position d to position p , shifting the intervening values by exactly one slot to close the gap:

$$S'_s = \begin{cases} (v_{s,1}, \dots, v_{s,d-1}, v_{s,d+1}, \dots, v_{s,p}, v, v_{s,p+1}, \dots, v_{s,k_s}), & p > d \\ (v_{s,1}, \dots, v_{s,p-1}, v, v_{s,p}, \dots, v_{s,d-1}, v_{s,d+1}, \dots, v_{s,k_s}), & p < d \end{cases}$$

As a set, $S'_s = S_s$; no value is added or removed, $k'_s = k_s$, $|S'| = |S|$, and the decision space's dimensions are unchanged: $DS' = \langle X, y, \mathbf{S}, S_y \rangle$. What changes is only the preference order $\preceq_s$ that S_s carries. Define the set of values crossed by the move as exactly those whose relative

order with respect to v reverses:

$$Crossed(v, d, p) = \begin{cases} \{v_{s,d+1}, \dots, v_{s,p}\}, & p > d \\ \{v_{s,p}, \dots, v_{s,d-1}\}, & p < d \end{cases}$$

Because the domain $\mathbf{S} = S_1 \times \dots \times S_n$ is unaffected as a set, entered rules are left completely untouched:

$$E(R') = E(R)$$

The transformation reduces to a *relabeling of the order relation* $\preceq_s \rightarrow \preceq'_s$; nothing in $E(R)$ is added, removed, or reassigned. Its only (potential) effect is on whether $E(R')$ satisfies the principle of dominance under the new order $\preceq'_s$, which may lead to changing of the *undefined* status of some non-entered rules.

Algorithm. Pseudocode is shown in Algorithm 11.

Algorithm 11 Move input attribute values

```
function MoveInputValue(R, s, v, p):
    # R: SDT over DS = <X, y, S, S_y>
    # s: index of the attribute whose scale order changes
    # v: value in S_s to relocate
    # p: target position, p != d (current position of v)

    d ← current position of v in S_s
    Crossed ← values strictly between positions d and p (per direction)
    S_s' ← S_s with v moved from position d to p # same SET, new order
    S' ← S_1 x ... x S_s' x ... x S_n          # = S as a set

    R' ← Init(S', S_y)
    for each r = (c -> o) in E(R):
        w ← c with updated value indices at c[s]
        R'[w] ← (outcome = o, entered = true)

    R' ← CompleteByDominance(R')
    return R'
```

Information preservation. Omitted.

Example. Reuse Table 4.9. Apply “Move input values” on S_C : move $v = med$ (position 2) to position 3. Then $Crossed(med, 2, 3) = \{v_{s,3}\} = \{high\}$, and the new order is $S'_C = (low, high, med)$. The resulting Table 4.13 represents exactly the same mapping, but with the changed order and meaning of ‘ $\succ$ ’ for C . The table is not monotone after the move, with rules 2 and 11 the main sources of dominance violation.

4.2.7 Exchange input values

Consider $x_s \in X$ with ordered scale represented as a sequence $S_s = (v_{s,1}, \dots, v_{s,k_s})$. Choose two distinct values $v_a = v_{s,i}$, $v_b = v_{s,j}$, $i < j$ and *exchange* their positions directly, leaving every other value exactly where it was:

$$S'_s = (v_{s,1}, \dots, v_{s,i-1}, v_b, v_{s,i+1}, \dots, v_{s,j-1}, v_a, v_{s,j+1}, \dots, v_{s,k_s})$$

#	<i>A</i>	<i>B</i>	<i>C'</i>	<i>Y'</i>
1	<i>low</i>	<i>low</i>	<i>low</i>	<i>bad</i>
2	<i>low</i>	<i>low</i>	<i>high</i>	<i>acc</i>
3	<i>low</i>	<i>low</i>	<i>med</i>	<i>bad</i>
4	<i>low</i>	<i>high</i>	<i>low</i>	<i>bad</i>
5	<i>low</i>	<i>high</i>	<i>high</i>	<i>acc</i>
6	<i>low</i>	<i>high</i>	<i>med</i>	<i>acc</i>
7	<i>high</i>	<i>low</i>	<i>low</i>	<i>bad</i>
8	<i>high</i>	<i>low</i>	<i>high</i>	<i>acc</i>
9	<i>high</i>	<i>low</i>	<i>med</i>	<i>acc</i>
10	<i>high</i>	<i>high</i>	<i>low</i>	<i>acc</i>
11	<i>high</i>	<i>high</i>	<i>high</i>	<i>good</i>
12	<i>high</i>	<i>high</i>	<i>med</i>	<i>acc</i>

Table 4.13: Table 4.9 after moving $C = med$ to position 3.

As with “Move” (section 4.2.6), this changes only the order relation $\preceq_s \rightarrow \preceq'_s$: as a set, $S'_s = S_s$, so $S' = S$ and $DS' = DS$. Consequently the decision table itself is left completely untouched: outcomes “drag along” with their values, not with their positions:

$$R' = R \quad (\text{identical rules: same } \mathbf{v}(r), o(r), \text{entered}(r) \text{ for every condition})$$

“Exchange” differs from “Move” in how order relations are affected. Because v_a and v_b swap ends of the interval $[i, j]$ while every value strictly inside that interval stays fixed, an intervening value’s side relative to both endpoints flips, not just relative to one moving value as in “Move”. Define the affected set:

$$Between(v_a, v_b) = \{u = v_{s,m} : i < m < j\}$$

The pairs of values whose relative order is reversed by the transformation are then:

$$\{(v_a, v_b)\} \cup \{(v_a, u) : u \in Between(v_a, v_b)\} \cup \{(v_b, u) : u \in Between(v_a, v_b)\}$$

All other pairs of values (including any two values both outside $[i, j]$, both inside $Between(v_a, v_b)$, or one inside/one outside) retain their exact original order relation, since neither endpoint’s position, relative to them, changes side.

Algorithm. Pseudocode is shown in Algorithm 12.

Algorithm 12 Exchange input attribute values

```
function ExchangeInputValues(R, s, v_a, v_b):  
  # R: SDT over DS = <X, y, S, S_y>, with E(R) its entered rules  
  # s: index of the attribute whose scale order changes  
  # v_a, v_b: distinct values in S_s to swap positions  
  
  i ← position of v_a in S_s  
  j ← position of v_b in S_s  
  assume i < j  
  Between ← values at positions strictly between i and j  
    # untouched, but reversed vs. v_a, v_b  
  
  S_s' ← S_s with v_a and v_b swapped, all other positions unchanged  
  S' ← S_1 x ... x S_s' x ... x S_n # = S as a set  
  
  R' ← Init(S', S_y)  
  for each r = (c -> o) in E(R):  
    w ← c with updated value indices at c[s]  
    R'[w] ← (outcome = o, entered = true)  
  
  R' ← CompleteByDominance(R')  
  return R'
```

Information preservation. Omitted.

Example. Reuse Table 4.9. Apply “Exchange input values” on S_C : swap $v_a = low$ (position 1) and $v_b = high$ (position 3). Then $Between(low, high) = \{med\}$ (position 2), and the new order is $S'_C = (high, med, low)$. Position 1 (least preferred by convention) now holds *high*, position 3 (most preferred) now holds *low*. The result is shown in Table 4.14, with outcomes of swapped rules printed in bold. The table is not monotone any more.

#	A	B	C'	Y'
1	low	low	high	acc
2	low	low	med	bad
3	low	low	low	bad
4	low	high	high	acc
5	low	high	med	acc
6	low	high	low	bad
7	high	low	high	acc
8	high	low	med	acc
9	high	low	low	bad
10	high	high	high	good
11	high	high	med	acc
12	high	high	low	acc

Table 4.14: Table 4.9 after exchanging values $C = low$ and $C = high$.

4.3 Transformations due to changes of output attribute scale

The third category of decision table transformations are due to changes of the output attribute's scale from S_y to S'_y . The new decision space is therefore

$$DS' = \langle X, y, \mathbf{S}, S'_y \rangle$$

Both X and $\mathbf{S}$, and hence the entire condition space, are *unchanged*; the row count (number of rules) is invariant. However, S_y does change, potentially affecting the DT outcomes.

Output-value transformations follow the same general pattern as input-value transformations (section 4.2) and include “Add/Insert”, “Duplicate”, “Delete”, “Merge”, “Copy”, “Move” and “Exchange”. This correspondence is important because, within a DEX model, the output attribute of a decision table may serve as an input attribute of another table at the next higher level in the hierarchy. A change to the scale of such an attribute therefore requires corresponding transformations of both tables.

4.3.1 Add/Insert output value

Let $DS = \langle X, y, \mathbf{S}, S_y \rangle$ and let R be an SDT over it, with $S_y = \{v_{y,1}, \dots, v_{y,k_y}\}$. Choose an insertion position $p \in [1, k_y + 1]$ and a new value $v^* \notin S_y$. Define

$$S'_y = \{v_{y,1}, \dots, v_{y,p-1}, v^*, v_{y,p}, \dots, v_{y,k_y}\}, \quad k'_y = k_y + 1$$

with the relative order of the old values preserved and $v_{y,p-1} \prec v^* \prec v_{y,p}$, and boundary terms omitted at the ends. When $p = k_y + 1$, we call the transformation “Add output value”, otherwise “Insert output value”.

Since $S_y \subset S'_y$ via the order-preserving identity embedding $\iota : S_y \hookrightarrow S'_y$, every legal old outcome remains legal, unmodified, in the new value space: a singleton $o(r) \in S_y$ is trivially in S'_y ; an interval $o(r) = [l, h] \in I(S_y)$ with $l, h \in S_y$ is literally the same pair $[l, h]$, now read as an interval of $I(S'_y)$; *undefined* stays *undefined*. Consequently, the transformation can be defined as the identity on the entered rule set:

$$E(R') = E(R) \quad (\text{same conditions, same singleton outcomes, same entered flags})$$

This is followed by the usual dominance post-processing.

The disadvantage is that no transformed rule ever maps to v^* , as a direct consequence of this construction. Achieving that may require additional editing of the table.

Algorithm. Pseudocode is shown in Algorithm 13.

Algorithm 13 Add/Insert output attribute value

```
function AddInsertOutputValue(R, v_star, p):  
    # R: SDT over DS = <X, y, S, S_y>  
    # v_star: new output value, v_star not in S_y  
    # p: insertion position, 1 <= p <= k_y + 1  
  
    S_y' ← insert v_star into S_y at position p # preserve order on old values  
    DS' ← <X, y, S, S_y'> # X and S unchanged  
  
    R' ← Init(S, S_y')  
  
    for each r = (v → o) in E(R):  
        R'[v] ← R[v] # entered rules copied verbatim, unchanged  
  
    R' ← CompleteByDominance(R')  
    return R'
```

Information preservation. Omitted.

Example. Reuse the partly defined Table 4.6. Apply “Add/Insert output value”: insert $v^* = mid$ at position $p = 3$, i.e., strictly between *acc* and *good*: $S'_y = (bad, acc, mid, good)$. All 12 original conditions are unchanged: the 8 entered rows are untouched, and the 4 derived rows recompute by “CompleteByDominance” to the identical bounds as before.

4.3.2 Duplicate output value

Pick an index $j \in [1, k_y]$, i.e., a value $v_j \in S_y$ to *duplicate*. Introduce a fresh value $v'_j \notin S_y$ and insert it immediately after v_j :

$$S'_y = \{v_1, \dots, v_j, v'_j, v_{j+1}, \dots, v_{k_y}\}, \quad k'_y = k_y + 1$$

with $v_j \prec v'_j \prec v_{j+1}$, or v'_j becomes the new top element if $j = k_y$. All other order relations of S_y are left intact. The input side is untouched: $DS' = \langle X, y, S, S'_y \rangle$, so the condition/rule set is unchanged, and only outcomes change.

Moreover, outcomes that do not include v_j can stay unchanged. The only question is how to treat rules with the original outcome v_j . Since v_j and v'_j are duplicates, the “cleanest” solution would be to assign the interval outcome $[v_j, v'_j]$ instead of v_j . While this would be admissible for decision tables that allow interval outcomes for all rules, it is not possible in SDTs, where entered rules are required to have single-valued outcomes. We may resolve this conflict by:

- leaving rules with the outcome v_j intact; such “Duplicate” is equivalent to “Add/Insert”, and none of the entered rules would map to v'_j after the transformation.
- changing the entered status of such rules to *false*, leaving them for dominance post-processing.

Both resolutions are not ideal and can potentially cause information loss: in the first case by not making the two outcomes equivalent, and in the second case by converting an already entered rule to non-entered, which contradicts their “untouchable” status postulated with SDTs.

Hereby we informally take the second approach.

Algorithm. Pseudocode is shown in Algorithm 14.

Algorithm 14 Duplicate output attribute value

```

function DuplicateOutputValue(R, S_y, j):
    # R : SDT over DS = <X, y, S, S_y>
    # S_y : output scale [v_1, ..., v_k], increasingly ordered
    # j : index of the value to duplicate, 1 <= j <= k_y

    v_j ← S_y[j]
    v_j' ← fresh duplicate of v_j, with different name(v_j')
    S_y' ← S[1..j] ++ [v_j'] ++ S[j+1..k_y] # insert copy right after v_j

    R' ← Init(S, S_y')
    for each r = (v → o) in R: # all rules, o can be undefined
        r' ← r # copy both o and entered status
        if o == v_j:
            entered(r') ← false
        R'[v] ← r'

    R' ← CompleteByDominance(R')
    return R'

```

Information preservation. Omitted.

Example. Take Table 4.5. Duplicate *acc*, producing *acc'*, so that $S'_y = \{bad \prec acc \prec acc' \prec good\}$. Table 4.15 shows three outcome columns: the original *Y* outcomes, the “clean” *Y'* outcomes that would be assigned if the table facilitated intervals in full, and the actual outcomes obtained after post-processing the table by “CompleteByDominance”.

#	<i>A</i>	<i>B</i>	<i>C</i>	original <i>Y</i>	“clean” <i>Y'</i>	entered (after)	post-processed <i>Y'</i>
1	<i>low</i>	<i>low</i>	<i>low</i>	<i>bad</i>	<i>bad</i>	yes	<i>bad</i>
2	<i>low</i>	<i>low</i>	<i>high</i>	<i>bad</i>	<i>bad</i>	yes	<i>bad</i>
3	<i>low</i>	<i>high</i>	<i>low</i>	<i>bad</i>	<i>bad</i>	yes	<i>bad</i>
4	<i>low</i>	<i>high</i>	<i>high</i>	<i>acc</i>	[<i>acc</i> , <i>acc'</i>]	no	[<i>bad</i> , <i>good</i>]
5	<i>high</i>	<i>low</i>	<i>low</i>	<i>bad</i>	<i>bad</i>	yes	<i>bad</i>
6	<i>high</i>	<i>low</i>	<i>high</i>	<i>acc</i>	[<i>acc</i> , <i>acc'</i>]	no	[<i>bad</i> , <i>good</i>]
7	<i>high</i>	<i>high</i>	<i>low</i>	<i>acc</i>	[<i>acc</i> , <i>acc'</i>]	no	[<i>bad</i> , <i>good</i>]
8	<i>high</i>	<i>high</i>	<i>high</i>	<i>good</i>	<i>good</i>	yes	<i>good</i>

Table 4.15: Processing “Duplicate” $acc \rightarrow acc'$ of *Y* on Table 4.5.

4.3.3 Delete output value

Let $DS = \langle X, y, \mathbf{S}, S_y \rangle$ and *R* an SDT over it, $S_y = \{v_{y,1}, \dots, v_{y,k_y}\}$, $k_y > 1$. Choose $v = v_{y,e} \in S_y$ to *delete*:

$$S'_y = S_y \setminus \{v\}, \quad k'_y = k_y - 1, \quad DS' = \langle X, y, \mathbf{S}, S'_y \rangle$$

Again, *X* and *S* are unchanged: $|R'| = |R|$, and only the permissible outcome values shrink.

For entered rules, the effect is stark, and pinpoints the main difficulty of this transformation. An entered SDT rule carries a single outcome value $o(r) \in S_y$. If $o(r) \neq v$, nothing changes: the label survives trivially in S'_y . If $o(r) = v$, no substitute label can be recovered from the

rule itself: there is no information left to fall back on, so the rule is *demoted*: its outcome is discarded, and it is marked non-entered:

$$E(R') = \{r = (\mathbf{v} \rightarrow o) \in E(R) : o \neq v\}$$

Remark: Relation to the general, non-SDT case. For output spaces where non-entered outcomes are stored composite objects (intervals, sets, or probability distributions), deleting v can be done by directly editing the stored object (shrinking an interval’s bound, dropping a set element, zeroing a probability) without recomputation. This shortcut does not apply here: in SDT, non-entered outcomes are not stored explicitly, they are only marked and derived. Thus, “stripping v ” for an SDT reduces exactly to re-running the “CompleteByDominance” closure over $E(R')$ and S'_y . The two approaches are not guaranteed to produce the same answer once entered rules have been demoted.

Algorithm. Pseudocode is shown in Algorithm 15.

Algorithm 15 Delete output attribute value

```
function DeleteOutputValue(R, v):
    # R: SDT over DS = <X, y, S, S_y>
    # v: output value to delete, |S_y| > 1

    S_y' ← S_y \ { v }

    R' ← Init(S, S_y')

    for each r = (c → o) in E(R):
        if o != v:
            R'[c] ← (outcome = o remapped to S_y', entered = entered(r))
        # else: r is dropped: implicitly demoted to non-entered / undefined

    R' ← CompleteByDominance(R')
    return R'
```

Information preservation

(P1) No structural change. R' has the same conditions and row count as R . *Proof:* Immediate from definition. $\square$

(P2) Exact preservation of unaffected entered rules. For $r \in E(R)$ with $o(r) \neq v$: the corresponding $r' \in R'$ has $\mathbf{v}(r') = \mathbf{v}(r)$, $o(r') = o(r)$, $\text{entered}(r') = \text{true}$. *Proof:* Immediate, since $o(r) \in S'_y$. $\square$

(P3) Rules mapping exactly to v cannot be preserved. For $r \in E(R)$ with $o(r) = v$: no valid entered outcome for r exists in S'_y .

Proof. By the SDT definition, an entered rule’s outcome must be a single value from S_y . Since $v \notin S'_y$, $o(r) = v$ is no longer an admissible label, and r carries no further information beyond that single value from which any substitute could be inferred. The only information-preserving move available is to demote r to non-entered and let the general closure mechanism derive whatever residual bound the rest of the table still supports. $\square$

(P4) Monotone consistency is inherited. If $E(R)$ is monotone under $\preceq_y$, then $E(R')$ is monotone under $\preceq'_y$ (the restriction of $\preceq_y$ to S'_y).

Proof. $E(R') \subseteq E(R)$ (P2/3), every surviving outcome is literally unchanged (P2), and $\preceq'_y$ is exactly $\preceq_y$ restricted to $S_y \setminus \{v\}$. A sub-order of a total order restricted to a subset is the restriction of that order (same argument as P3). Hence any dominance pair among survivors that satisfied the principle in R still does in R' ; restriction cannot create a new violation. This guarantees “CompleteByDominance” is applied to a consistent entered set and is well-defined. $\square$

(P5) Information can only weaken, and may weaken severely. Since $E(R') = E(R) \setminus \{r : o(r) = v\}$ has strictly fewer entered comparators whenever v was used by at least one entered rule, every non-entered outcome of R' can only be the same or wider than what the same closure would give with those comparators still present. In the extreme, if v was the sole label separating multiple rules from its neighbors, its deletion can force many previously well-bounded rules to the maximally uninformative interval $[\text{low}(S'_y), \text{high}(S'_y)]$. $\square$

Example. Take Table 4.5. Apply “Delete output value” $v = acc$, so that $S'_y = (bad, good)$. This yields Table 4.16. There, the rules 4, 6 and 7, which were originally *acc*, are first demoted and then post-processed by dominance.

#	A	B	C	Y
1	low	low	low	bad
2	low	low	high	bad
3	low	high	low	bad
4	low	high	high	$[bad, good]$ (derived)
5	high	low	low	bad
6	high	low	high	$[bad, good]$ (derived)
7	high	high	low	$[bad, good]$ (derived)
8	high	high	high	good

Table 4.16: Delete $v = acc$ from Y in Table 4.5.

4.3.4 Merge output values

Similarly to “Merge input values” (section 4.2.4), this transformation turns two values $v_a, v_b \in S_y$ into a new single value v_{ab} of the transformed scale S'_y . In this case, all references to either v_a or v_b in the outcomes of entered decision rules are transformed to v_{ab} .

Conceptually, “Merge” is the inverse of “Duplicate” (section 4.3.2): instead of splitting one position into two, it collapses two positions into one, shrinking the scale. However, due to the difficulty of “Duplicate” that cannot create “clean” entered value intervals in SDTs, “Merge” and “Duplicate” are not exact inverses of each other.

Pick two *adjacent* positions $a, a+1 \in [1, k_y]$. Adjacency is required: merging non-adjacent values would leave whatever lies strictly between them with an undefined relation to the new merged value, which is exactly the kind of order-breaking situation identified later with “Move” (section 4.3.6) and “Exchange” (section 4.3.7). “Merge”, by contrast, is safe due to this precondition.

Introduce a fresh value v_{ab} for replacing both v_a and v_b , and define the position-collapsing map

$$\mu(i) = \begin{cases} i & i < a \\ a & a \leq i \leq a+1 \\ i-1 & i > a+1 \end{cases}$$

Generalize this map from positions to single values $v \in S_y$:

$$\mu(v) = u \in S_y : \text{pos}_{S_y}(u) = \mu(\text{pos}_{S_y}(v))$$

The new scale has one fewer position:

$$S'_y = \{v_1, \dots, v_{a-1}, v_{ab}, v_{a+2}, \dots, v_{k_y}\}, \quad k'_y = k_y - 1$$

with order $\prec'$ inherited from $\prec$ via μ (still total, since μ is monotone non-decreasing). Input space $\mathbf{S}$ is unaffected.

Every outcome is re-expressed using μ : $\psi(o) = \mu(o)$ for a single value, $\psi([l, h]) = [\mu(l), \mu(h)]$ for an interval, and $\psi(\text{undefined}) = \text{undefined}$.

The representation is transformed to:

$$R' = \{r'_v : v \rightarrow \psi(o(r_v)) \mid v \in \mathbf{S}\} \quad \text{entered}(r'_v) = \text{entered}(r_v)$$

Since μ maps both v_a and v_b to the same value v_{ab} , any entered rule that mapped to v_a or v_b now maps to v_{ab} , remaining a single value, still entered. No information is lost about the fact that these rules were entered, only the finer distinction between v_a and v_b disappears, which is precisely the intended semantics of merging.

Algorithm. Pseudocode is shown in Algorithm 16.

Algorithm 16 Merge output attribute values

function MergeOutputValues(R, S, a):

 # R : SDT

 # S : output scale [v_1, ..., v_k_y]

 # a : position of the first of the two values to merge;

 # merges positions a, a+1

 v_ab ← fresh scale value

 S_y' ← S[1..a-1] ++ [v_ab] ++ S[a+2..k_y] # collapse two positions to one

 mu(i) ← i if i < a
 ← a if i == a or i == a+1
 ← i - 1 if i > a+1

 R' ← Init(space of R, S_y')

 for each r = (v → o) in E(R):

 (l, h) ← AsInterval(o) # l = h if single value

 o' ← Interval(mu(l), mu(h))

 # degenerates to single value mu(l) if mu(l) == mu(h)

 R'[v] ← (outcome = o, entered = entered(r))

 R' ← CompleteByDominance(R')

 return R'

Information preservation

(P1) S'_y is a valid scale. μ is monotone non-decreasing and surjective onto $\{1, \dots, k_y - 1\}$, collapsing exactly the adjacent pair $(a, a+1)$; hence S'_y (positions $1, \dots, k_y - 1$ with the induced order) is a well-defined total order. $\square$

(P2) R' is a well-formed SDT. Conditions (S) are unchanged, so coverage is preserved. For any rule, $\psi(o)$ applied to a single value o yields $\mu(o)$, a single position in S'_y . Therefore, entered rules, whose flag is copied unchanged, still carry single-valued outcomes, satisfying the SDT requirement. Non-entered outcomes remain intervals or *undefined*. $\square$

(P3) Dominance is always preserved, merge never introduces a violation. Let $r_a \succ r_b$ (input dominance), so in R , $o(r_a) \succeq o(r_b)$, i.e. $\text{pos}(o(r_a)) \geq \text{pos}(o(r_b))$. Since μ is monotone non-decreasing,

$$\text{pos}(o(r_a)) \geq \text{pos}(o(r_b)) \Rightarrow \mu(\text{pos}(o(r_a))) \geq \mu(\text{pos}(o(r_b)))$$

That is, $\psi(o(r_a)) \succeq' \psi(o(r_b))$. So every dominance relation that held in R still holds in R' . Merging can only turn a strict preference into an equality, when r_a, r_b 's outcomes were exactly the merged pair v_a, v_b , but this is never a violation. $\square$

(P4) R' is maximally informative among scales of size $k_y - 1$ obtained by identifying v_a with v_b . Every rule not previously involving v_a or v_b is carried over with only a position relabeling and no information loss. Every rule that did involve v_a or v_b is mapped to the unique value v_{ab} that any such merge must produce, by definition of “Merge”. There is no alternative assignment that would preserve more information while still honoring that v_a and v_b are no longer distinguishable. $\square$

Example. Take Table 4.5 and merge y values *acc* and *good* into *acc+good*. The result (Table 4.17) is straightforward: all rules remain entered and there are no induced dominance violations.

#	A	B	C	Y'
1	<i>low</i>	<i>low</i>	<i>low</i>	<i>bad</i>
2	<i>low</i>	<i>low</i>	<i>high</i>	<i>bad</i>
3	<i>low</i>	<i>high</i>	<i>low</i>	<i>bad</i>
4	<i>low</i>	<i>high</i>	<i>high</i>	<i>acc+good</i>
5	<i>high</i>	<i>low</i>	<i>low</i>	<i>bad</i>
6	<i>high</i>	<i>low</i>	<i>high</i>	<i>acc+good</i>
7	<i>high</i>	<i>high</i>	<i>low</i>	<i>acc+good</i>
8	<i>high</i>	<i>high</i>	<i>high</i>	<i>acc+good</i>

Table 4.17: Merging $v = \text{acc}$ and $v = \text{good}$ of Y in Table 4.5.

4.3.5 Copy output value

In scale S_y , pick a source position $d \in [1, k_y]$ and a distinct destination position $q \in [1, k_y]$, $d \neq q$. *Copying* v_d onto v_q means: the position q is kept (i.e., not deleted, reordered, or merged with d , which are all different transformations), but its label is overwritten to match v_d 's label:

$$S'_y = S_y \text{ (same positions, same order } \prec), \quad \text{name}'(i) = \begin{cases} \text{name}(d) & i = q \\ \text{name}(i) & i \neq q \end{cases}$$

S'_y has the same cardinality $k'_y = k_y$ and the same order relation as S_y ; the only thing that changes is that positions d and q now carry an identical label, even though they remain two distinct, differently-ordered scale elements: $\text{pos}(v_d) \neq \text{pos}(v_q)$, and $v_d \prec v_q$ or $v_q \prec v_d$ still holds according to their original order.

Since no position is added, removed, or reordered, and dominance/outcome computations operate on positions, not on labels, the rule set is unaffected:

$$R' = R \quad (\text{same conditions, same ordinal outcomes, same } entered(\cdot) \text{ flags})$$

The only observable change is “cosmetic”: wherever an outcome equal to v_q is displayed, it now shows the label $name(d)$, indistinguishable in print from outcomes at v_d , even though internally v_d and v_q remain different and comparable scale elements.

This transformation is not very useful by itself, but it is required for consistency with “Copy input value” (section 4.2.5) one level up the model hierarchy, where y plays the role of an input attribute of another table. Copying a value there requires this operation to be defined here, too.

Algorithm, Information preservation and Example. Trivial, omitted.

4.3.6 Move output value

Move, when applied to an output scale S_y , changes the positional order of its elements, usually with the aim to change their implied preferential order. Take a source position $p \in [1, k_y]$ and target position $p' \in [1, k_y]$, $p \neq p'$. Moving $v_p \in S_y$ to position p' means removing it from its current slot and reinserting it at p' , shifting the values in between by one slot. This transformation keeps the individual elements $v \in S_y$ intact (including their $name(v)$), but changes their positions $pos(v)$, implicitly affecting the relation ‘ $\prec$ ’. This is unlike “Copy” (section 4.3.5), which changes a label at a fixed position, or “Duplicate” (section 4.3.2), which inserts a new position.

Formally, define the position permutation $\pi : [1, k_y] \rightarrow [1, k_y]$:

$$\pi(q) = \begin{cases} p' & q = p \\ q - 1 & p < q \leq p' \quad (\text{moving forward, } p < p') \\ q + 1 & p' \leq q < p \quad (\text{moving backward, } p' < p) \\ q & \text{otherwise} \end{cases}$$

The new scale S'_y has the same value set as S_y but a new order $\prec'$:

$$v \prec' w \iff \pi(pos(v)) < \pi(pos(w))$$

The input space $\mathbf{S}$ remains intact. Also, decision rules are “untouched” in the following sense: a rule r that mapped to outcome $o(r) = o = acc$ still maps to acc , where $acc = name(o)$. However, the positions $pos(o) = pos(acc)$ before and after “Move” might be different.

Accordingly,

$$E(R') = E(R)$$

while monotonicity and dominance might be affected by the changed preference order. Because $\prec'$ can invert the relative order of the moved value against values it passed over, some previously dominance-consistent entered pairs may now conflict; per the SDT convention, such conflicts are resolved by assigning *undefined* to the non-entered rules caught between them, while the conflicting entered rules themselves are kept intact.

Algorithm. Pseudocode is shown in Algorithm 17. Even though the entered rules do not change (in mathematical terms), rule outcomes (single values and intervals) might still need some implementation-dependent reindexing. To this end, this algorithm employs **Reindex(o)** on outcomes of all entered rules being transformed.

Algorithm 17 Move output attribute value

function MoveOutputValue(R, S, p, p'):

 assert p != p'

 S_y' ← splice S:

 remove value at p, reinsert at p', shift intervening values by one

 R' ← Init(space of R, S_y')

 for each r = (v → o) in E(R):

 R'[v] ← (outcome = Reindex(o), entered = true)

 R' ← CompleteByDominance(R')

 return R'

Information preservation

(P1) S'_y is a valid scale. π is a bijection on $\{1, \dots, k_y\}$ (a single-element list-splice), so S'_y has the same cardinality and is totally ordered by $\prec'$. $\square$

(P2) R' is a well-formed SDT. Conditions (S) are unchanged, so full coverage holds. Entered rules keep the same single values (assured by `Reindex()`). Non-entered outcomes are produced by “CompleteByDominance”, which by definition returns values in $I(S'_y) \cup \text{undefined}$. Hence R' satisfies the SDT structural requirements. $\square$

(P3) Dominance is not generally preserved. Take two entered rules $r_a, r_b \in E(R)$ with input dominance $r_a \succ r_b$, so originally $\text{pos}(o(r_a)) \geq \text{pos}(o(r_b))$.

- If $\text{pos}(o(r_a))$ and $\text{pos}(o(r_b))$ are both outside $[\min(p, p'), \max(p, p')]$, π is the identity on both, so the relation is unaffected.
- If both lie strictly inside the shifted sub-range (excluding p itself), they are shifted by the same uniform offset, so their relative order is preserved.
- If one of them is exactly v_p (the moved value) and the other lies in the range it was moved across, their relative order is *inverted* by construction of π : the moved element jumps from one end of the interval to the other, crossing every value strictly between.

So dominance is preserved for all pairs except those directly involving the moved value and something it “passed over”. Where both rules of such a pair are entered, the conflict is kept; any non-entered rule that would have been sandwiched between them is set to *undefined* by “CompleteByDominance”, since no valid interval satisfies both endpoints. $\square$

(P4) Given the fixed entered set, R' is maximally informative. Since $E(R') = E(R)$ is held fixed and “CompleteByDominance” returns, by definition, the tightest interval consistent with $E(R')$ and $\prec'$ (or *undefined* exactly where no valid interval exists), no other valid SDT sharing the same entered rules and the same new order S'_y can carry more information. $\square$

Example. Take Table 4.5 again. Move *acc* from position $p = 2$ to $p' = 3$:

$$\pi(1) = 1, \quad \pi(2) = 3, \quad \pi(3) = 2 \quad \Rightarrow \quad S'_y = \{bad(1) \prec good(2) \prec acc(3)\}$$

While outcome names stay the same (see Table 4.18), their positions (shown in parenthesis) change. This change causes monotonicity violations of rule 8 against rules 4, 6 and 7.

#	<i>A</i>	<i>B</i>	<i>C</i>	<i>Y'</i>	changed
1	<i>low</i>	<i>low</i>	<i>low</i>	<i>bad</i> (1)	no
2	<i>low</i>	<i>low</i>	<i>high</i>	<i>bad</i> (1)	no
3	<i>low</i>	<i>high</i>	<i>low</i>	<i>bad</i> (1)	no
4	<i>low</i>	<i>high</i>	<i>high</i>	<i>acc</i> (3)	yes
5	<i>high</i>	<i>low</i>	<i>low</i>	<i>bad</i> (1)	no
6	<i>high</i>	<i>low</i>	<i>high</i>	<i>acc</i> (3)	yes
7	<i>high</i>	<i>high</i>	<i>low</i>	<i>acc</i> (3)	yes
8	<i>high</i>	<i>high</i>	<i>high</i>	<i>good</i> (2)	yes

Table 4.18: Move *acc* of *Y* from position 2 to 3 from Table 4.5.

4.3.7 Exchange output values

Exchange is a restricted, “local” case of “Move” (section 4.3.6): instead of splicing one value out and shifting a whole range, exactly two positions swap ranks. Every other position, including any values strictly between them, keeps its original rank number.

Pick two distinct positions $p, q \in [1, k_y]$, $p \neq q$, and $p < q$ without loss of generality. Define the permutation

$$\pi(i) = \begin{cases} q & i = p \\ p & i = q \\ i & i \neq p, q \end{cases}$$

The new scale keeps the same label set, but reorders it:

$$S'_y = S_y \text{ (same labels),} \quad v \prec' w \iff \pi(pos(v)) < \pi(pos(w))$$

As with “Move”, input space and rule labels are untouched. Entered rules are untouchable and copied over verbatim (same label, same conditions).

$$E(R') = \{r'_{\mathbf{v}} : \mathbf{v} \rightarrow o(r_{\mathbf{v}}) \mid r_{\mathbf{v}} \in E(R)\}$$

Non-entered outcomes, which were derived by dominance under the old order, must be recomputed under $\prec'$.

Algorithm. Pseudocode is shown in Algorithm 18.

Algorithm 18 Exchange output attribute values

```
function ExchangeOutputValues(R, S, p, q):  
    assert p != q  
  
    S_y' ← S with elements of positions p and q swapped  
    R' ← Init(space of R, S_y')  
  
    for each r = (v → o) in E(R):  
        R'[v] ← (outcome = Reindex(o), entered = true)  
  
    R' ← CompleteByDominance(R')  
    return R'
```

Information preservation

(P1) S'_y is a valid scale. A transposition is a bijection on $\{1, \dots, k_y\}$, so S'_y is a totally ordered set of the same cardinality. $\square$

(P2) R' is a well-formed SDT. Same argument as “Move” (section 4.3.6). $\square$

(P3) Dominance is preserved for every pair not involving v_p or v_q . This preserves strictly more relations than “Move”. For positions $i, j \notin \{p, q\}$, $\pi(i) = i$ and $\pi(j) = j$, so their relative order in S'_y is identical to their order in S_y , regardless of whether i, j lie inside or outside the window (p, q) . Hence for any two rules $r_a \succ r_b$ (input dominance) with $o(r_a), o(r_b) \notin \{v_p, v_q\}$, $o(r_a) \succeq o(r_b)$ in S_y directly implies $o(r_a) \succeq' o(r_b)$ in S'_y : no violation possible.

Violations can arise only where $o(r_a)$ or $o(r_b) \in \{v_p, v_q\}$:

- v_p vs. v_q themselves always reverse ($\pi(p) = q > \pi(q) = p$, but originally $p < q$).
- v_p or v_q vs. any v_x inside $x \in [p + 1, q - 1]$ always reverse, since originally $p < x < q$, but $\pi(p) > \pi(x)$ or $\pi(x) > \pi(q)$, respectively.
- v_p or v_q vs. any v_x outside $[p, q]$ never flips, since $\pi(x) = x$ and $\pi(p), \pi(q) \in \{p, q\} \subset [p, q]$, preserving the same side of the comparison.

Where a genuine violation occurs between two entered rules, both are kept. Any non-entered rule sandwiched between them is set to *undefined*, exactly as for “Move”. $\square$

(P4) Given the fixed entered set, R' is maximally informative. Same argument as “Move”. $\square$

Example. Take Table 4.5 again. Exchange S_y positions 1 and 3. The result is shown in Table 4.19. Similarly to in Table 4.18 the changes induced inconsistencies between the rules, for example rule 4 against rules 1, 2, and 3. There are as many as 16 violating pairs.

4.4 Structural transformations

Structural transformations reach beyond a single decision table and operate on a *subtree* of connected SDTs.

#	A	B	C	Y'	changed
1	low	low	low	bad(3)	yes
2	low	low	high	bad(3)	yes
3	low	high	low	bad(3)	yes
4	low	high	high	acc(2)	no
5	high	low	low	bad(3)	yes
6	high	low	high	acc(2)	no
7	high	high	low	acc(2)	no
8	high	high	high	good(1)	yes

Table 4.19: Exchange outcomes *bad* and *good* in Table 4.5.

4.4.1 Flatten subtree

Flatten converts a whole subtree of connected decision tables into a single “flat” decision table. Only *entered* rules are composed: the non-entered rules of the subtree tables, and the intervals they carry, are ignored. The non-entered rules of the flat table are then obtained, as for any SDT, by dominance (section 3.2). Consequently, every entered rule of the result is single-valued by construction, and no interval arithmetic is needed.

Let us first extend the notation. A *subtree of attributes* is a rooted tree $\mathcal{T}$ with root y and node set N . Every internal node $z \in N$ (an *aggregate attribute*) has children $Ch(z) = (z_1, \dots, z_{m_z})$ and an associated SDT

$$T_z : S_{z_1} \times \dots \times S_{z_{m_z}} \rightarrow V(S_z)$$

with entered rules $E(T_z)$ and an *ordered* scale S_z (SDT requirement). Leaves of $\mathcal{T}$ are the *basic attributes* $B(\mathcal{T}) = \{x_1, \dots, x_n\}$; they carry a scale, which may be ordered or unordered, but no table. For $z \in N$ let $B(z)$ denote the basic attributes in the subtree rooted at z , so $B(z) = \{z\}$ for a leaf and $B(z) = \biguplus_i B(z_i)$ otherwise. For $\mathbf{x} \in \prod_{b \in B(z)} S_b$ let $\mathbf{x}|_{z_i}$ be its projection onto $B(z_i)$.

The value of z that follows from *entered rules only* is the partial function $\Phi_z : \prod_{b \in B(z)} S_b \rightarrow S_z$:

$$\Phi_z(\mathbf{x}) = \begin{cases} x_z & z \text{ is a leaf} \\ o & z \text{ internal, } \mathbf{v} = (\Phi_{z_1}(\mathbf{x}|_{z_1}), \dots, \Phi_{z_{m_z}}(\mathbf{x}|_{z_{m_z}})) \text{ is defined and } (\mathbf{v} \rightarrow o) \in E(T_z) \\ \perp & \text{otherwise.} \end{cases}$$

Here $\perp$ only says that the value is not determined by entered rules. It is not an SDT outcome and must not be confused with *undefined*. The flat table is

$$E^{flat} = \{(\mathbf{x} \rightarrow \Phi_y(\mathbf{x})) : \mathbf{x} \in \prod_{b \in B(y)} S_b, \Phi_y(\mathbf{x}) \neq \perp\}$$

$$DS^{flat} = \langle B(y), y, \prod_{b \in B(y)} S_b, S_y \rangle$$

$$Flat(y) = CompleteByDominance(DS^{flat}, E^{flat})$$

In words: a flat rule is entered if and only if every table along the path from the basic attributes to the root contributes an entered rule, and its outcome is the result of this chain. All other rules are non-entered.

Algorithm. Pseudocode is shown in Algorithm 19. Assignments are sets of pairs (attribute, value), so the assignments of the children can be united. The work is proportional to the sizes of the sets F_z , each at most $\prod_{b \in B(z)} |S_b|$, plus the completion step.

Algorithm 19 Flatten subtree

```
function EnteredFlat(z):
    # returns  $F_z = \{ (x, o) : x \text{ in } \prod_{b \in B(z)} S_b, \Phi_z(x) = o \}$ 
    if z is a leaf:
        return  $\{ (\{z = w\}, w) : w \text{ in } S_z \}$ 
    for each child  $z_i$  of z:
         $F_i \leftarrow \text{EnteredFlat}(z_i)$ 
        index  $F_i$  by value:  $F_i[w] \leftarrow \{ x : (x, w) \text{ in } F_i \}$ 
     $F \leftarrow \{ \}$ 
    for each entered rule  $(v_1, \dots, v_m \rightarrow o) \text{ in } E(T_z)$ :
        for each  $(x_1, \dots, x_m) \text{ in } F_1[v_1] \times \dots \times F_m[v_m]$ :
             $F \leftarrow F \cup \{ (x_1 \cup \dots \cup x_m, o) \}$ 
    return F

function FlattenSubtree(y):
     $F \leftarrow \text{EnteredFlat}(y)$ 
     $E_{\text{flat}} \leftarrow \{ (x \rightarrow o) : (x, o) \text{ in } F \}$ 
     $DS_{\text{flat}} \leftarrow \langle B(y), y, \prod_{b \in B(y)} S_b, S_y \rangle$ 

    return CompleteByDominance( $DS_{\text{flat}}$ ,  $E_{\text{flat}}$ )
```

Information preservation

(P1) $Flat(y)$ is a well-formed SDT over DS^{flat} . E^{flat} contains at most one rule per condition, each with a single value of S_y . *Proof.* Φ_y is a function: $\mathbf{x}$ determines $\mathbf{x}|_{z_i}$, hence the child values $\mathbf{v}$, hence the unique rule of $E(T_z)$ with condition $\mathbf{v}$. By induction, $\text{EnteredFlat}(z)$ returns exactly the graph of Φ_z . “CompleteByDominance” then assigns the remaining rules. $\square$

(P2) Entered rules are exactly the chains of entered rules. A flat rule is entered iff, at every internal node z , the combination of values produced by its children is the condition of an entered rule of T_z , and its outcome is the value obtained by evaluating the subtree with these rules. *Proof.* Immediate from the definition of Φ_y . $\square$

(P3) Monotonicity is preserved. If $E(T_z)$ satisfies the principle of dominance for every internal node z , then E^{flat} satisfies it, and $Flat(y)$ contains no *undefined* outcome. *Proof.* By induction on the depth of z we show: if $\mathbf{x} \succeq \mathbf{x}'$ and both $\Phi_z(\mathbf{x}), \Phi_z(\mathbf{x}')$ are defined, then $\Phi_z(\mathbf{x}) \succeq \Phi_z(\mathbf{x}')$. For a leaf this is trivial. For an internal node, the child values satisfy $\mathbf{v} \succeq \mathbf{v}'$ by induction (unordered leaves are equal; internal children have ordered scales), and $\mathbf{v}, \mathbf{v}'$ are conditions of entered rules of T_z . Hence $o \succeq o'$. The absence of *undefined* follows from the lemma of section 3.2. $\square$

Information loss. “Flatten” is not information-preserving in the full sense. The following is lost:

- the non-entered rules of the subtree tables and their intervals;
- entered rules of T_z whose condition is never produced from entered rules of the children.

The intervals of the non-entered flat rules are derived by dominance from E^{flat} only, so they may be wider than those obtained by propagating the intervals of the subtree tables. The transformation is not reversible.

More complete information preservation would require explicitly accounting for non-entered outcome intervals in the tables being combined. This would ultimately entail explicitly *entering* these intervals into T_z . This is not possible with SDTs, which restrict entered outcomes to single values, but would be feasible with extended decision tables that support interval-valued outcomes.

Example. Consider the subtree $X_{12} = X_{12}(X_1, X_2)$, $Y = Y(X_{12}, X_3)$, with $X_1, X_2, X_3 \in \{low, high\}$ and $X_{12} \in \{lo, mi, hi\}$. There are two decision tables, connected by X_{12} : Table 4.20, where X_{12} is an input attribute, and Table 4.21, where X_{12} is the output.

#	X_{12}	X_3	Y'	entered
1	<i>lo</i>	<i>low</i>	<i>bad</i>	yes
2	<i>lo</i>	<i>high</i>	<i>bad</i>	yes
3	<i>mi</i>	<i>low</i>	<i>[bad, acc]</i>	no
4	<i>mi</i>	<i>high</i>	<i>acc</i>	yes
5	<i>hi</i>	<i>low</i>	<i>[bad, good]</i>	no
6	<i>hi</i>	<i>high</i>	<i>good</i>	yes

Table 4.20: Decision table that maps X_{12} and X_3 to Y .

#	X_1	X_2	X_{12}
1	<i>low</i>	<i>low</i>	<i>lo</i>
2	<i>low</i>	<i>high</i>	<i>mi</i>
3	<i>high</i>	<i>low</i>	<i>mi</i>
4	<i>high</i>	<i>high</i>	<i>hi</i>

Table 4.21: Decision table that maps X_1 and X_2 to X_{12} (all rules are entered).

Only the entered rules are used: all rules of Table 4.21 and rules 1, 2, 4 and 6 of Table 4.20. Rules 3 and 5 of Table 4.20 (intervals) are ignored. For each of the $2 \times 2 \times 2 = 8$ combinations of (X_1, X_2, X_3) , X_{12} is determined by Table 4.21 and then looked up in Table 4.20. Combinations 1, 2, 4, 6 and 8 reach entered rules and give entered flat rules. Combinations 3, 5 and 7 reach the non-entered rules (mi, low) , (mi, low) and (hi, low) and remain non-entered. Their outcomes are obtained by dominance from the five entered rules, e.g. for combination 7 the nearest entered rule below is rule 1 (*bad*) and above it is rule 8 (*good*), giving *[bad, good]*. The result is Table 4.22. Column (X_{12}) is shown for reference only and is not part of the flat table. In this example the intervals coincide with those that would follow from propagating the intervals of Table 4.20; in general the dominance intervals can be wider.

#	X_1	X_2	X_3	Y'	entered	(X_{12})
1	<i>low</i>	<i>low</i>	<i>low</i>	<i>bad</i>	yes	<i>lo</i>
2	<i>low</i>	<i>low</i>	<i>high</i>	<i>bad</i>	yes	<i>lo</i>
3	<i>low</i>	<i>high</i>	<i>low</i>	<i>[bad, acc]</i>	no	<i>mi</i>
4	<i>low</i>	<i>high</i>	<i>high</i>	<i>acc</i>	yes	<i>mi</i>
5	<i>high</i>	<i>low</i>	<i>low</i>	<i>[bad, acc]</i>	no	<i>mi</i>
6	<i>high</i>	<i>low</i>	<i>high</i>	<i>acc</i>	yes	<i>mi</i>
7	<i>high</i>	<i>high</i>	<i>low</i>	<i>[bad, good]</i>	no	<i>hi</i>
8	<i>high</i>	<i>high</i>	<i>high</i>	<i>good</i>	yes	<i>hi</i>

Table 4.22: A flattened decision table over (X_1, X_2, X_3) .

4.4.2 Decompose a table

This transformation is an approximate reverse of “Flatten subtree”, restricted to splitting off exactly two input attributes at a time. It is a decision-table instance of classical *function decomposition* [6]. The algorithm below is the SDT-table analogue of the column-partition step used in HINT (Hierarchical INduction Tool) [7]. In contrast with “Flatten”, “Decompose” uses the completed table, including the non-entered closure intervals.

Let $T_y : S_{A_1} \times S_{A_2} \times \mathbf{S}_{rest} \rightarrow V(S_y)$ be an SDT with $n \geq 3$ inputs, $\mathbf{S}_{rest} = S_{x_3} \times \dots \times S_{x_n}$. Represent T_y as a *partition matrix* M :

- rows, indexed by $i \in S_{A_1} \times S_{A_2}$ (size $p = |S_{A_1}| \cdot |S_{A_2}|$),
- columns, indexed by $j \in \mathbf{S}_{rest}$ (size $q = |\mathbf{S}_{rest}|$),
- entries $M[i][j] = o(r_{(i,j)}) \in I(S_y) \cup \{\text{undefined}\}$, the outcome of T_y at $(A_1, A_2) = i, X_{rest} = j$, possibly dominance-completed.

Write $M[i][\cdot]$ for row i as a vector of q outcomes.

Two partition-table entries $o_1, o_2 \in I(S_y) \cup \{\text{undefined}\}$ are *equal*, denoted $o_1 = o_2$, if and only if they represent the same single value in S_y or the same interval in $I(S_y)$. By convention, *undefined* is equal to itself.

Row equivalence. Define, on the row index set $S_{A_1} \times S_{A_2}$:

$$i \sim i' \iff M[i][\cdot] = M[i'][\cdot]$$

This is a genuine equivalence relation (reflexive, symmetric, transitive, i.e. vector equality). Let $S_C = (S_{A_1} \times S_{A_2}) / \sim = \{c_1, \dots, c_t\}$ be the resulting classes; this is the constructed scale of the new intermediate attribute C , of size $t = |S_C| \leq p$.

Order on S_C . Define a pointwise order on row-vectors: $M[i][\cdot] \preceq M[i'][\cdot]$ iff $M[i][j] \preceq M[i'][j]$ for every column j (interval order on entries: $[l, h] \preceq [l', h']$ iff $l \preceq l'$ and $h \preceq h'$). In comparisons, interpret *undefined* as a special entry, lower than any regular value or interval. Since class members share an identical vector, this induces a well-defined partial order on classes:

$$c \preceq c' \iff M[i][\cdot] \preceq M[i'][\cdot] \text{ for (any) } i \in c, i' \in c'$$

Tables H and G .

$$H : S_{A_1} \times S_{A_2} \rightarrow S_C, \quad H(a_1, a_2) = [(a_1, a_2)]_{\sim}$$

$$G : S_C \times \mathbf{S}_{rest} \rightarrow V(S_y), \quad G(c, \mathbf{x}_r) = M[i][\mathbf{x}_r] \text{ for any } i \in c$$

G is well-defined precisely because all rows in a class agree on every column, by the definition of ‘ $\sim$ ’.

Entered flags. Consistent with the information-preservation principle used throughout this section:

- For H : whole row must be user-committed:

$$\text{entered}_H(a_1, a_2) = \bigwedge_{j \in \mathbf{S}_{rest}} \text{entered}_Y((a_1, a_2), j)$$

- For G : any one committed representative suffices, since the value is shared by the whole class:

$$\text{entered}_G(c, \mathbf{x}_r) = \bigvee_{i \in c} \text{entered}_Y(i, \mathbf{x}_r)$$

Algorithm. Pseudocode is shown in Algorithm 20.

Algorithm 20 Decompose decision table

```

function DecomposeTable(Ty, A1, A2):
    Xrest ← InputsOf(Ty) \ { A1, A2 }
    M      ← BuildMatrix(Ty, rows = SA1 x SA2, cols = SXrest)

    # 1. Row equivalence classes
    classes ← {}                # map: row-vector -> class id
    for i in SA1 x SA2:
        v ← M[i][*]
        if v not in classes: classes[v] ← NewClassId()
    SC ← Values(classes)        # the new scale, size t

    # 2. Order classes (pointwise dominance of their row-vectors)
    order ← PartialOrderByPointwiseDominance(classes)
    if not IsTotal(order):
        order ← LinearExtension(order) # topological sort -> total order
    SC ← SortByOrder(SC, order)

    # 3. Build H
    H ← Init(SA1 x SA2, SC)
    for (a1, a2) in SA1 x SA2:
        c ← classes[ M[(a1,a2)][*] ]
        e ← AND( entered_Y((a1,a2), j) for j in SXrest )
        H[(a1,a2)] ← ( outcome = c, entered = e )
    H ← CompleteByDominance(H)

    # 4. Build G
    G ← Init(SC x SXrest, Sy)
    for c in SC:
        i ← AnyRepresentative(c)
        for xr in SXrest:
            o ← M[i][xr]
            e ← OR( entered_Y(i', xr) for i' in c )
            G[(c, xr)] ← ( outcome = o, entered = e )
    G ← CompleteByDominance(G)

    return (G, H, SC)

```

Information preservation

(P1) H and G are well-formed SDTs. H : covers every $(a_1, a_2) \in S_{A_1} \times S_{A_2}$ exactly once (loop over the full product); output scale S_C is by construction a totally ordered discrete set (P3 below); entered rules carry a single class value, which satisfies SDT structure. G : covers every $(c, \mathbf{x}_r) \in S_C \times \mathbf{S}_{rest}$ exactly once; well-defined regardless of representative choice, by definition of ' $\sim$ '; outcomes lie in $I(S_y) \cup \text{undefined}$, exactly as in T_y . $\square$

(P2) Exactness: $G(H(A_1, A_2), X_{rest}) = T_y(A_1, A_2, X_{rest})$ for every input combination. Fix $(a_1, a_2, \mathbf{x}_r)$, let $i = (a_1, a_2)$ and $c = H(i) = [i]_{\sim}$. By definition, $G(c, \mathbf{x}_r) = M[i'][\mathbf{x}_r]$ for

some representative $i' \in c$. Since $i \sim i'$ means $M[i][\cdot] = M[i'][\cdot]$ on every column, in particular $M[i][\mathbf{x}_r] = M[i'][\mathbf{x}_r]$. Hence $G(H(i), \mathbf{x}_r) = M[i][\mathbf{x}_r] = T_y(a_1, a_2, \mathbf{x}_r)$. The decomposition is exact (not an approximation) by construction. $\square$

(P3) Minimality: $|S_C| = t$ is the smallest possible size for any exact decomposition $Y = G'(C', X_{rest})$, $C' = H'(A_1, A_2)$. Suppose some exact decomposition uses $|S_{C'}| < t$. Since t is the number of distinct row-vectors and $|S_{C'}| < t$, by pigeonhole some two rows $i \neq i'$ with $M[i][\cdot] \neq M[i'][\cdot]$ (hence $i \not\sim i'$) must be assigned the same value $H'(i) = H'(i') = c'$. Since they differ in at least one column j_0 (i.e. $M[i][j_0] \neq M[i'][j_0]$), exactness of G' requires simultaneously $G'(c', j_0) = M[i][j_0]$ and $G'(c', j_0) = M[i'][j_0]$ — a contradiction, as G' is a function of (c', j_0) alone. Hence no exact decomposition can use fewer than t values; S_C as constructed is minimal. $\square$

Example. We reuse the flattened Table 4.22 constructed in the previous section. “Decompose” should invert “Flatten”, up to attribute and value names, and entered flags. Recall: $(X_1, X_2, X_3) \rightarrow Y^{flat}$, $X_1, X_2, X_3 \in \{low, high\}$, $Y^{flat} \in \{bad, acc, good\}$.

Decompose with $A_1 = X_1$, $A_2 = X_2$, $X_{rest} = \{X_3\}$. Build the partition matrix with rows (X_1, X_2) and columns X_3 (Table 4.23).

(X_1, X_2)	$X_3=low$	$X_3=high$
(low, low)	<i>bad</i>	<i>bad</i>
$(low, high)$	$[bad, acc]$	<i>acc</i>
$(high, low)$	$[bad, acc]$	<i>acc</i>
$(high, high)$	$[bad, good]$	<i>good</i>

Table 4.23: Partition matrix, derived from Table 4.22.

Rows $(low, high)$ and $(high, low)$ have identical vectors, so they merge into one class; the other two rows are singletons. This gives exactly $t = 3$ classes:

$$S_C = \{c_1, c_2, c_3\}, \quad c_1 = \{(low, low)\}, \quad c_2 = \{(low, high), (high, low)\}, \quad c_3 = \{(high, high)\}$$

Pointwise order in this case is $(bad, bad) \preceq ([bad, acc], acc) \preceq ([bad, good], good)$, so $c_1 \prec c_2 \prec c_3$ is already total (P3 holds cleanly here).

In this way, “Decompose” effectively constructed a new attribute C with scale $S_C = \{c_1, c_2, c_3\}$. C corresponds to the previous attribute X_{12} with scale $S_{X_{12}} = \{lo, mi, hi\}$. There is a one-to-one correspondence between scale values, despite different names.

Using this partition matrix approximately reproduces the original two Tables 4.20 and 4.21. The new tables G and H are shown in Tables 4.24 and 4.25, respectively. Decision rules match. The only differences are a different name for the newly constructed attribute (C instead of X_{12}), and different labels of the corresponding scale values. This is because the table being decomposed does not contain this information, preventing “Decompose” from reconstructing it. Also, reconstructed entered flags are not equal to the original ones.

#	C	X_3	Y'	entered
1	c_1	<i>low</i>	<i>bad</i>	yes
2	c_1	<i>high</i>	<i>bad</i>	yes
3	c_2	<i>low</i>	$[bad, acc]$	no
4	c_2	<i>high</i>	<i>acc</i>	yes
5	c_3	<i>low</i>	$[bad, good]$	no
6	c_3	<i>high</i>	<i>good</i>	yes

Table 4.24: Table G after decomposing Table 4.22.

#	X_1	X_2	C	entered
1	<i>low</i>	<i>low</i>	c_1	yes
2	<i>low</i>	<i>high</i>	c_2	no
3	<i>high</i>	<i>low</i>	c_2	no
4	<i>high</i>	<i>high</i>	c_3	no

Table 4.25: Table H after decomposing Table 4.22.

4.5 Local transformations

In this study, we focus on transformations invoked by changing table *dimensions* in terms of input attributes, attribute scales or model structure around a table. Nevertheless, other, less “invasive” transformations should also be mentioned, as they play an important role in the decision-modeling process:

Enter/Retract a single rule. These transformations are actually the main ones used to define the *contents* of decision tables, which is then operated upon by other transformations. For some condition $\mathbf{v}$ and corresponding rule $r = (\mathbf{v} \rightarrow o)$, *Enter* means setting the value of $o \in S_y$ and $\text{entered}(r) = \text{true}$. Conversely, *Retract* of an entered rule toggles its $\text{entered}(r)$ to *false*. Every such change is followed by “CompleteByDominance”. Usually, these transformations are carried out iteratively in interaction with the model creator, until the table contents are considered satisfactory. See [3] for more information about decision table creation.

Toggle scale order. *Preferential order* of a scale can be changed from *ordered* to *unordered* and vice versa. Whenever the order of any scale associated with a decision table (i.e., any input or output scale) changes, it may affect non-entered rules and thus requires post-processing using “CompleteByDominance”. While the change from *ordered* to *unordered* only widens non-entered intervals, the change in the other direction potentially induces dominance violations, causing the assignment of *undefined* to some non-entered rules.

Replace scale. *Replacing* some scale S with another scale S' boils down to two cases. When $|S| = |S'|$ (calling the scales *assignment compatible*), the transformation is trivial: just replace S with S' , assuming the same value positions and implied preference order. If one scale is ordered and the other is not, “Scale type conversion” occurs at the same time. The second case $|S| \neq |S'|$ would require defining an explicit mapping of values from one scale to another, which is difficult and is generally discouraged. In practice, it is usually carried out as a sequence of more elementary scale transformations.

These three transformations (with the two variants of the first and the simple version of the third) are already implemented in the DEXiWin software [2]. However, there are still other transformations that are not.

Restrict attribute. This transformation would *restrict conditions* of the table on one specific, fixed value of an entire input attribute, discarding the other rows outright rather than aggregating them, and reducing the table’s arity by one. This is distinct from “Delete input value” (removes one scale value, keeps the attribute) and from “Delete input attribute” (which aggregates across all of an attribute’s values via the join operator).

Consistency repair. None of our transformations so far resolve a dominance violation between two entered rules. A genuinely different transformation would take an inconsistent $E(R)$ and propose a minimal change (revert some rule(s) to non-entered, or suggest new values that restore monotonicity). This is an optimization problem, not a mechanical rewrite.

Instead of a formula that reads off the new table directly, it searches for some “optimal” way to restore monotonicity.

These two transformations are good candidates for future implementation.

Another important set of “transformations” relate to representing decision tables in compact, human-readable forms. So far, DEX already provides a number of such methods, from approximating a full table by a parametric aggregation function (e.g. a weighted-sum), to “compactifying” the table in terms of aggregate (complex) decision rules. However, there are still representational methods not tested with DEX. These transformations are of a fundamentally different character and would require a separate study.

The same holds for methods that learn decision tables from data [6] and revise existing decision tables to comply with new data evidence [8].

4.6 Summary

This section addressed transformations of standard decision tables. They are summarized in

- Table 4.26: Transformations of input attributes
- Table 4.27: Transformations of input scales
- Table 4.28: Transformations of output scale
- Table 4.29: Structural transformations (section 4.4)
- Table 4.30: Local transformations (section 4.5)

Sect.	Transformation	Signature	Effect	Monotonicity	Reversible?
4.1.1	Add/Insert input attribute	Insert x_{new} at position q ; every $\mathbf{v} \in \mathbf{S}$ fans out to k_{new} copies, with the same outcome on every copy	$\mathbf{S} \rightarrow \mathbf{S} \cdot k_{new}$	Always preserved	Yes: any fixed-value slice along x_{new} exactly recovers R
4.1.2	Duplicate input	Copy $x_s \rightarrow x'$; $S' = S_s$; only the “diagonal” ($x' = x_s$) copy inherits entered status	$\mathbf{S} \rightarrow \mathbf{S} \cdot k_s$	Always preserved	Yes: diagonal slice $R'_{x'=x_s} = R$
4.1.3	Delete input	Project away x_q ; entered only where all k_q collapsed witnesses agree	$\mathbf{S} \rightarrow \mathbf{S}/k_q$	Always preserved	No (lossy in general)
4.1.4	Exchange/Move inputs	Position permutation σ on X (transposition = “Exchange”, splice = “Move”)	$\mathbf{S}$ unchanged	Always preserved	Yes (σ^{-1})

Table 4.26: Transformations of input attributes.

Sect.	Transformation	Signature	k'_s	Monotonicity	Reversible?
4.2.1	Add/Insert input value	Insert v^* at position p ; new rows non-entered, closed via CompleteByDominance	$k_s + 1$	Always preserved	Yes (“Delete” undoes)
4.2.2	Duplicate input value	$v_j \rightarrow \{v_j, v'_j\}$	$k_s + 1$	Always preserved	Yes (“Merge”)
4.2.3	Delete input value	Rows with $x_s = v$ removed outright	$k_s - 1$	Preserved for fully entered tables	No (lossy)
4.2.4	Merge input values	$v_a, v_b \rightarrow v_{ab}$; outcomes of collapsed rows joined	$k_s - 1$	Generally not preserved	No (lossy)
4.2.5	Copy input value	Entered rules not indexed by u preserved; those indexed by u lost	k_s	Preserved iff x_s is unordered or $I(v, u) = \emptyset$	No (overwrites)
4.2.6	Move input value	Splice v to new position	k_s	Conditionally (pairs spanning the shift range only)	Yes (inverse splice)
4.2.7	Exchange input values	Transpose two positions	k_s	Conditionally (pairs touching p or q only)	Yes (self-inverse)

Table 4.27: Transformations of input scales.

Sect.	Transformation	Signature	k'_y	Monotonicity	Reversible?
4.3.1	Add/Insert output value	$S'_y = S_y \cup \{v^*\}$, order-preserving; $E(R') = E(R)$ verbatim	$k_y + 1$	Always preserved	Yes
4.3.2	Duplicate output value	$v_j \rightarrow \{v_j, v'_j\}$	$k_y + 1$	Always preserved, with possible <i>widening</i>	No (lossy)
4.3.3	Delete output value	Entered rules with $o(r) = v$ demoted (no substitute derivable); rest recomputed	$k_y - 1$	Always preserved, with possible <i>widening</i>	No (lossy)
4.3.4	Merge output values	Adjacent $v_a, v_b \rightarrow v_{ab}$	$k_y - 1$	Always preserved (collapsing is order-safe)	No (lossy)
4.3.5	Copy output value	Relabel position d to $name(s)$; positions unchanged	k_y	Always preserved (identity on positions)	Yes (value labels lost)
4.3.6	Move output value	Splice value from p to p'	k_y	Conditionally (pairs spanning the shift range)	Yes
4.3.7	Exchange output values	Transpose ranks p, q	k_y	Conditionally (pairs touching p or q)	Yes

Table 4.28: Transformations of output attribute values.

Transformation	Signature	Direction	Exactness
Flatten subtree	$Flat(y) = T_y \circ (T_{z_1}, \dots, T_{z_m})$, recursive to the leaves	Many tables $\rightarrow$ one	Entered chains preserved; sub-table intervals lost
Decompose a table	$Y = G(X_{rest}, C=H(A_1, A_2)),$ $S_C = (S_{A_1} \times S_{A_2})/\sim$	One table $\rightarrow$ two, for a <i>chosen</i> pair A_1, A_2	Exact and minimal in S_C

Table 4.29: Structural transformations of decision tables.

Transformation	Signature	Dominance	Monotonicity	Reversible?
Enter rule	$E(R') = E(R) \cup \{\mathbf{v}_0 \mapsto o_0\}$	—	Not guaranteed	Yes
Retract rule	$E(R') = E(R) \setminus \{\mathbf{v}_0 \mapsto o_0\}$	—	Always safe (can only widen)	Yes of a prior “Enter” on a previously free rule
Toggle order ordered$\rightarrow$unordered	$ordered(S_s) : true \rightarrow false$	Dominance predicate loosens	Always preserved (can only widen)	Yes when “Toggle order unordered $\rightarrow$ ordered” with same $\prec$
Toggle order unordered$\rightarrow$ordered	Assuming existing value order	Dominance predicate tightens	Not guaranteed	Yes
Replace scale (same-size)	Replace $S_s \rightarrow S'_s$ May trigger “Toggle order”	—	Preserved with equally-ordered scales	Yes

Table 4.30: Local decision table transformations.

5. On Transformations of Non-Standard Decision Tables

In previous sections, we restricted the study to standard decision tables. In this section we discuss extending STD to non-standard decision tables. Particularly interesting are the following DT subtypes, characterized by the types of outcomes $V(S_y)$:

1. value *intervals* (applicable to all rules, including entered ones),
2. value *sets*, which are interesting for unordered output scales, or
3. probability or fuzzy *distributions* of outcome values.

In this section, we investigate how these extensions of $V(S_y)$ might affect the formal framework for transformations, the corresponding algorithms, and the proofs of information preservation. For example, value intervals, sets, and probability distributions are not generally comparable in terms of preference. How does this affect the proposed approach? Similarly, how can monotonicity constraints be applied to extended outcome representations?

We address these questions less formally than in section 4. A thorough investigation would require detailed formulations, analogous to those developed for SDTs; such an undertaking is beyond the scope of this study. Nevertheless, our aim is to provide informative answers to most of the key questions concerning non-standard decision tables. Specifically, we focus on using probability distributions, which is a powerful general representation that can be, with some assumptions, used in place of intervals and sets.

5.1 Comparing outcome values

Trying to force a *total* order on the probability simplex over $S, k = |S|$, would be a mistake: any total order requires collapsing a $(k - 1)$ -dimensional object to a line (e.g. via expected rank, entropy-adjusted score, etc.).

The natural, defensible order over S is *first-order stochastic dominance* (FOSD):

$$p \succeq q \iff F_p(v) \leq F_q(v) \quad \forall v \in S, \quad F_p(v) = \sum_{x \preceq v} p(x)$$

where F_p is the *cumulative distribution function* (CDF).

This says “ p puts at least as much mass on the high end, at every threshold.” It is exactly the order under which $p \succeq q \iff \mathbb{E}_p[u] \geq \mathbb{E}_q[u]$ for every monotone utility u . It is also only a *partial* order: many pairs of distributions are *incomparable*. This incomparability is not a bug to work around, but rather the honest reflection of “even though the scale is ordinal, we cannot rank everything”.

Although that FOSD is only partial, the space of CDFs on a finite ordered set, under the pointwise order, *is a lattice*: pointwise min and max of two valid CDFs are themselves valid CDFs. Translating back:

- *join* ($p \vee q$, the least distribution FOSD-dominating both) = the distribution whose CDF is the pointwise min of F_p, F_q ,
- *meet* ($p \wedge q$, the greatest distribution FOSD-dominated by both) = pointwise max of F_p, F_q .

This is exactly the operation the “CompleteByDominance” needs whenever it currently takes a scalar max/min over dominated/dominating entered rules. The generalization is mechanical: replace scalar max/min with CDF-wise min/max. So the interval $I(S) = [l, h]$ that non-entered SDT rules carry generalizes to a pair of bounding CDFs $[F_{lo}, F_{hi}]$. This is precisely a *p-box* [9]. In its simplest form, a p-box is defined by a pair of bounding cumulative distribution functions $[\underline{F}(x), \overline{F}(x)]$ that circumscribe an imprecisely known true distribution function $F(x)$:

$$\underline{F}(x) \leq F(x) \leq \overline{F}(x) \quad \forall x \in S_x$$

A p-box represents the set of all distributions consistent with the bounds, exactly the same semantics our intervals had (“the true value could be anywhere in here”). This means:

- Everything we built for SDTs (e.g., dominance closure and the “CompleteByDominance”) is literally the *degenerate/vertex special case* of this richer structure (single values = degenerate p-boxes). The distributional theory should be built once, at the p-box level, with SDT recovered as a restriction.
- The algorithms mostly survive with a local edit: scalar ops $\rightarrow$ CDF-wise ops. The character of consistency-checking changes: instead of a binary “does/doesn’t violate” we now get a three-way outcome: dominates, is dominated, or *incomparable*. Then, “incomparable entered rules” is a new, legitimate state that is not a violation and should be treated explicitly.

5.2 Effect on individual transformations

When it comes to specific transformations, some get easier and some get genuinely harder.

“Merge”, “Delete” and “Add” on scales get cleaner, not harder. Merging $v_a, v_b \rightarrow v_{ab}$ becomes literal mass-pooling, $p'_{ab} = p_a + p_b$, unambiguous and exact; no “join a set of ≤ 2 outcomes” heuristic is needed (but is still required in the interval/set case). Deleting v_j becomes conditioning/renormalizing the remaining mass, $p'_i = p_i / (1 - p_j)$ for $i \neq j$. This is well-defined whenever $p_j < 1$, only degenerating to the old “destructive” case at the point mass $p_j = 1$. Adding a value just extends every distribution with $p_{new} = 0$. Distributions make these operations more graceful than the scalar/interval versions, because there’s now a principled way to redistribute partial belief instead of an all-or-nothing choice.

“Duplicate”, “Copy”, “Move”, “Exchange” get harder due to a new ambiguity, not just extended notation. Duplicating v_j into v_j, v'_j raises a real question: if $p_j > 0$, how should that mass split between the two copies? We now have a menu of legitimate options: leave all mass at v_j by convention; split proportionally to neighboring resolved mass; represent the ambiguity explicitly as a two-vertex p-box/credal set spanning “ p_j at v_j ” and “ p_j at v'_j ”. This needs an explicit, justified default in the eventual formalization; it is a modeling decision, not something forced by the math. “Move/Exchange” are mechanically the same “reindex, recompute closure” as before, but recomputing FOSD relations after reordering is $O(k)$ pointwise-CDF comparisons per pair rather than one scalar comparison, and, since FOSD was already only partial before the reorder, the bookkeeping of what becomes newly comparable/incomparable is more involved than the old “monotone or not” check.

5.3 Flatten

“Flatten” needs an assumption the SDT framework never needed: *independence*; no assumption was needed about how children relate to each other. Propagating *distributions* through T_z to get an actual output distribution requires a genuine pushforward:

$$p_z(v) = \sum_{\mathbf{u}: T_z(\mathbf{u})=v} \prod_i p_i(u_i),$$

and that product form is only correct if the children’s uncertainties are *independent*. DEX’s per-attribute-scale structure gives no information about dependency between siblings, so this becomes an explicit modeling assumption the project has to own: either declare independence by default (simplest, likely wrong in general), or extend the framework with an explicit joint/copula structure between siblings (much heavier). SDT-based “Flatten” dodged this question entirely; distribution-based “Flatten” cannot.

5.4 Decompose

Exact equivalence becomes the wrong tool, almost by necessity. “Decompose” relied on exact equality of row-vectors to define ‘ $\sim$ ’. With distribution-valued outcomes, exact equality of two elicited probability vectors is a measure-zero event in practice (rounding, elicitation noise). So the distributional setting essentially forces the move to *approximate decomposition*: clustering “close enough” row-distributions rather than requiring identity.

5.5 Information-preservation proofs

Information-preservation proofs must become quantitative. Previously “preserve as much information as possible” was mostly a qualitative argument: “this interval is a subset of that one, hence weakly more informative”, i.e., a partial order on informativeness itself. Distributions give a real numeric handle: *Shannon entropy* [10] of a resolved distribution, or *p-box width* (area between the bounding CDFs) for an unresolved one. This means proofs can move from “no less informative” to “quantifiably how much was lost,” and it gives approximate/optimization-flavored transformations (approximate decomposition, consistency repair, table compactification into weights) an actual objective function to minimize rather than just a correctness property to verify.

5.6 Complexity

Inevitably, processing of distributions increases from $O(1)$ (SDT, interval) to $O(k)$, and every dominance check goes from a scalar comparison to $O(k)$ CDF comparisons. This is usually combined with the existing combinatorial processing of $\prod_i |S_i|$ rows. Fortunately, real-life DTs are relatively small and very rarely exceed a few hundred rules. At this scale, the increase from $O(1)$ to $O(k)$ does not pose any serious issue.

6. Conclusions

Decision tables are an important component of DEX decision models. They represent the decision maker’s preferential knowledge about how the elementary properties of decision alternatives affect their overall evaluation. As a DEX model is developed, its decision tables undergo various changes. Decision rules may be added to or deleted from a table, resulting in local modifications. Other changes, such as adding, deleting, or rearranging attributes in the model, or modifying attribute scales, additionally affect the dimensions of decision tables. If we disregarded the contents of the tables, these changes would amount to routine mechanical modifications along the three dimensions of a decision table: input attributes, the scales of input attributes, and the scale of the output attribute. However, when the table contents matter and the goal is to preserve as much existing information as possible during a transformation, the task becomes considerably more complex. Seemingly simple operations, such as “Add input value”, may need to be distinguished into several operations, such as “Add/Insert” and “Duplicate”, each with a different semantic interpretation and a different effect on the table contents. The task is further complicated by the need to consider the principle of dominance and to ensure the monotonicity of decision tables whenever possible.

In this study, we proposed a formal framework for analyzing decision table transformations, focusing specifically on standard decision tables (STD), which account for the vast majority (over 99%) of DEX decision tables encountered in practice. STD is defined for an ordered output attribute and requires each entered decision rule to have a single-value outcome. The outcomes of non-entered rules are determined according to the principle of dominance and are generally represented as value intervals, or as *undefined* in cases of dominance violation.

After defining the necessary framework components (section 2) and formulating the principles of information preservation (section 3.1) and dominance preservation (section 3.2), we identified and described 25 SDT transformations:

- Five due to changes of input attributes (section 4.1): “Add/Insert”, “Duplicate”, “Delete”, “Move”, and “Exchange”.
- Seven due to changes of input scales (section 4.2): “Add/Insert”, “Duplicate”, “Delete”, “Merge”, “Copy”, “Move”, and “Exchange”.
- Seven due to changes of output scales (section 4.3): “Add/Insert”, “Duplicate”, “Delete”, “Merge”, “Copy”, “Move”, and “Exchange”.
- Two structural changes (section 4.4): “Flatten” and “Decompose”.
- Four local changes (section 4.5): “Enter” and “Retract” decision rule, “Toggle” scale order and “Replace” the scale.

It is important that transformations corresponding to changes in input and output scales are mutually consistent. When applied to an aggregate non-root attribute, such transformations affect both the upper decision table, in which the attribute serves as an input, and the lower decision table, in which it serves as the output.

Compared with the previous study [3], the present study adopts a more formal and systematic approach, providing precise mathematical definitions of DEX model components, decision tables, and decision rules. It explicitly defines the different subtypes of decision tables and identifies the SDT as the most common subtype. The concepts of information preservation and dominance preservation are also formalized. For the first time, transformations applicable to SDTs are systematically described in terms of four elements: a mathematical definition, a transformation algorithm, a proof of information preservation, and an illustrative example. Two novel categories of transformations are also introduced: structural transformations, which modify the model structure surrounding a decision table (“Flatten” and “Decompose”), and local transformations, which modify individual decision rules or replace a scale with another compatible scale.

Beyond satisfying theoretical curiosity, this work is important for paving the way toward a comprehensive implementation of the DEX method. The current implementation, DEXiWin [2], already supports most of the identified transformations, but lacks the theoretical foundation that this study provides. In particular, DEXiWin’s handling of non-entered rules and information preservation could benefit substantially from the results of this research, which addresses these issues at a level of systematic detail not previously achieved. Another avenue for future research is to extend the range of decision table (DT) subtypes to accommodate outcomes represented by probability distributions and fuzzy sets of values. Although DEXiWin already supports these representations to some extent, the present study indicates that extending the transformation framework to accommodate them is non-trivial and requires a similarly systematic and formal approach.

Acknowledgement

This work has been financially supported by the Slovenian Research and Innovation Agency for the programme Knowledge Technologies, research core funding No. P2-0103.

References

- [1] M. Bohanec, “DEX (Decision EXpert): A qualitative hierarchical multi-criteria method,” in *Multiple Criteria Decision Making: Techniques, Analysis and Applications*, A. J. Kulka-rni, Ed. Singapore: Springer Nature Singapore, 2022, pp. 39–78, ISBN: 978-981-16-7414-3. DOI: [10.1007/978-981-16-7414-3_3](https://doi.org/10.1007/978-981-16-7414-3_3)
- [2] M. Bohanec, “DEXiWin: DEX decision modeling software, User’s manual, Version 1.2,” Institut Jožef Stefan, Ljubljana, Report IJS DP-14747, 2024. [Online]. Available: https://kt.ijs.si/MarkoBohanec/pub/2024_DP14747_DEXiWin.pdf
- [3] M. Bohanec, “Transformations of decision tables in qualitative multi-criteria decision modeling method DEX,” Institut Jožef Stefan, Ljubljana, Report IJS DP-14761, 2024. [Online]. Available: https://kt.ijs.si/MarkoBohanec/pub/2024_DP14761_Transformations.pdf
- [4] N. Trdin and M. Bohanec, “Extending the multi-criteria decision making method DEX with numeric attributes, value distributions and relational models,” *Central European Journal of Operations Research*, vol. 26, no. 1, pp. 1–41, 2018. DOI: [10.1007/s10100-017-0468-9](https://doi.org/10.1007/s10100-017-0468-9)
- [5] M. Bohanec, “Multi-criteria DEX models: An overview and analysis,” in *SOR-2017: 14th International Symposium on Operational Research in Slovenia*, L. Zadnik Stirn et al., Eds., Ljubljana: Slovenian Society Informatika, Section for Operational Research, Sep. 2017, pp. 155–160. [Online]. Available: https://kt.ijs.si/MarkoBohanec/pub/2017_SOR_DEXmodels.pdf
- [6] B. Zupan, M. Bohanec, J. Demšar, and I. Bratko, “Learning by discovering concept hierarchies,” *Artificial Intelligence*, vol. 109, no. 1–2, pp. 211–242, 1999. DOI: [10.1016/S0004-3702\(99\)00008-9](https://doi.org/10.1016/S0004-3702(99)00008-9)
- [7] M. Bohanec and B. Zupan, “A function-decomposition method for development of hierarchical multi-attribute decision models,” *Decision Support Systems*, vol. 36, no. 3, pp. 215–233, 2004. DOI: [10.1016/S0167-9236\(02\)00148-3](https://doi.org/10.1016/S0167-9236(02)00148-3)
- [8] M. Žnidaršič and M. Bohanec, “Revision of qualitative multi-attribute decision models,” in *DSS2004: IFIP International Conference on Decision Support Systems: Decision Support in an Uncertain and Complex World*, R. Meredith, G. Shanks, D. Arnott, and S. Carlson, Eds., 2004, pp. 881–888.
- [9] S. Ferson, V. Kreinovich, L. Ginzburg, K. Sentz, and D. S. Myers, “Constructing probability boxes and Dempster-Shafer structures,” Sandia National Laboratories, Albuquerque, New Mexico, Tech. Rep. SAND2002-4015, 2003.
- [10] A. K. Nanda and S. Chowdhury, “Shannon’s entropy and its generalisations towards statistical inference in last seven decades,” *International Statistical Review*, vol. 88, no. 3, pp. 657–675, 2020. DOI: [10.1111/insr.12374](https://doi.org/10.1111/insr.12374)