US 20170206304A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2017/0206304 A1

GOODMAN et al. (43) Pub. Date: Jul. 20, 2017

(54) **RESOLVING CONFIGURATION CONFLICTS USING A MULTI-VALUED DECISION DIAGRAM**

(71) Applicant: **FORD MOTOR COMPANY**, Dearborn, MI (US)

(72) Inventors: **Bryan Roger GOODMAN**, Northville, MI (US); **Melinda Kaye HUNSAKER**, Canton, MI (US); **David Mark NEWTON**, Koeln (DE); **Yu-Ning LIU**, Ann Arbor, MI (US); **Essam Mahmoud SABBAGH**, Dearborn Heights, MI (US); **Rickie Allan SPRAGUE**, Gladwin, MI (US); **Yakov M. FRADKIN**, Farmington Hills, MI (US)

(21) Appl. No.: **15/294,149**

(22) Filed: **Oct. 14, 2016**

Related U.S. Application Data

(60) Provisional application No. 62/280,609, filed on Jan. 19, 2016, provisional application No. 62/352,463, filed on Jun. 20, 2016.

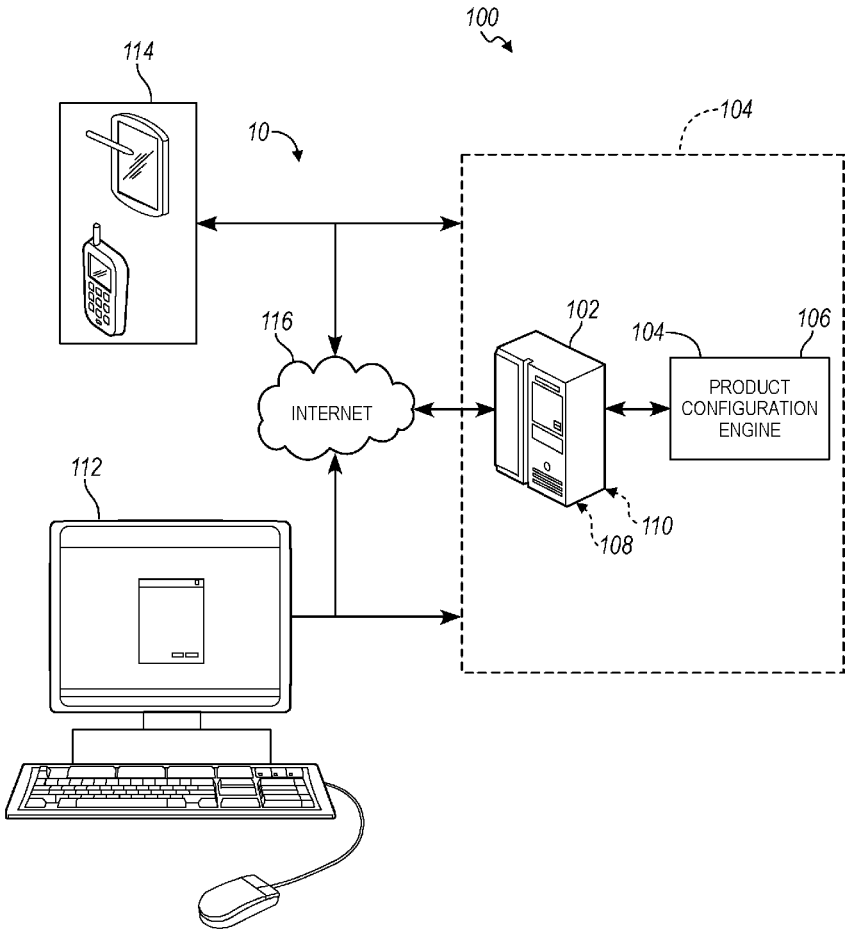
Publication Classification

(51) **Int. Cl.**
G06F 17/50 (2006.01)

(52) **U.S. Cl.**
CPC ***G06F 17/5095*** (2013.01)

(57) **ABSTRACT**

A system is provided with a memory device and a processor. The memory device is adapted to store data representative of a multi-valued decision diagram (MDD) specifying a buildable space of all possible valid configurations of a vehicle. The processor is in communication with the memory and is programmed to identify an invalid configuration, and to generate a restricted buildable space, including to determine an edit distance of each complete path indicative of a number of features to change the invalid configuration of that path to one of the valid configurations, identify a minimum of the edit distances, and remove configurations having edit distances larger than the minimum. The processor is further programmed to identify at least one feature to change the invalid configuration to at least one valid configuration based on the restricted buildable space; and to generate output indicative of the at least one feature to change.



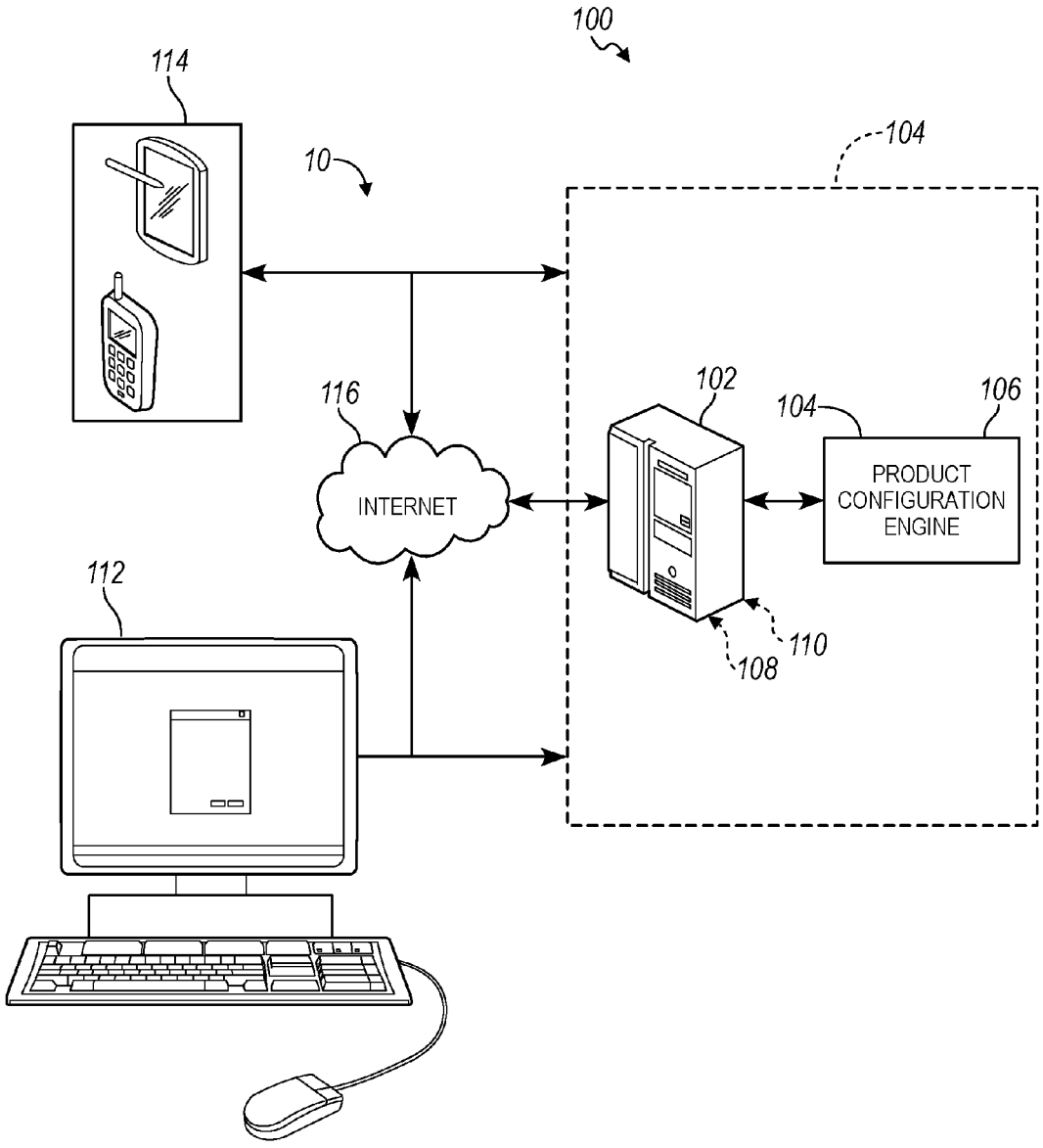

100
114
10
104
116
102
104
106
INTERNET
PRODUCT CONFIGURATION ENGINE
112
110
108

FIG. 1

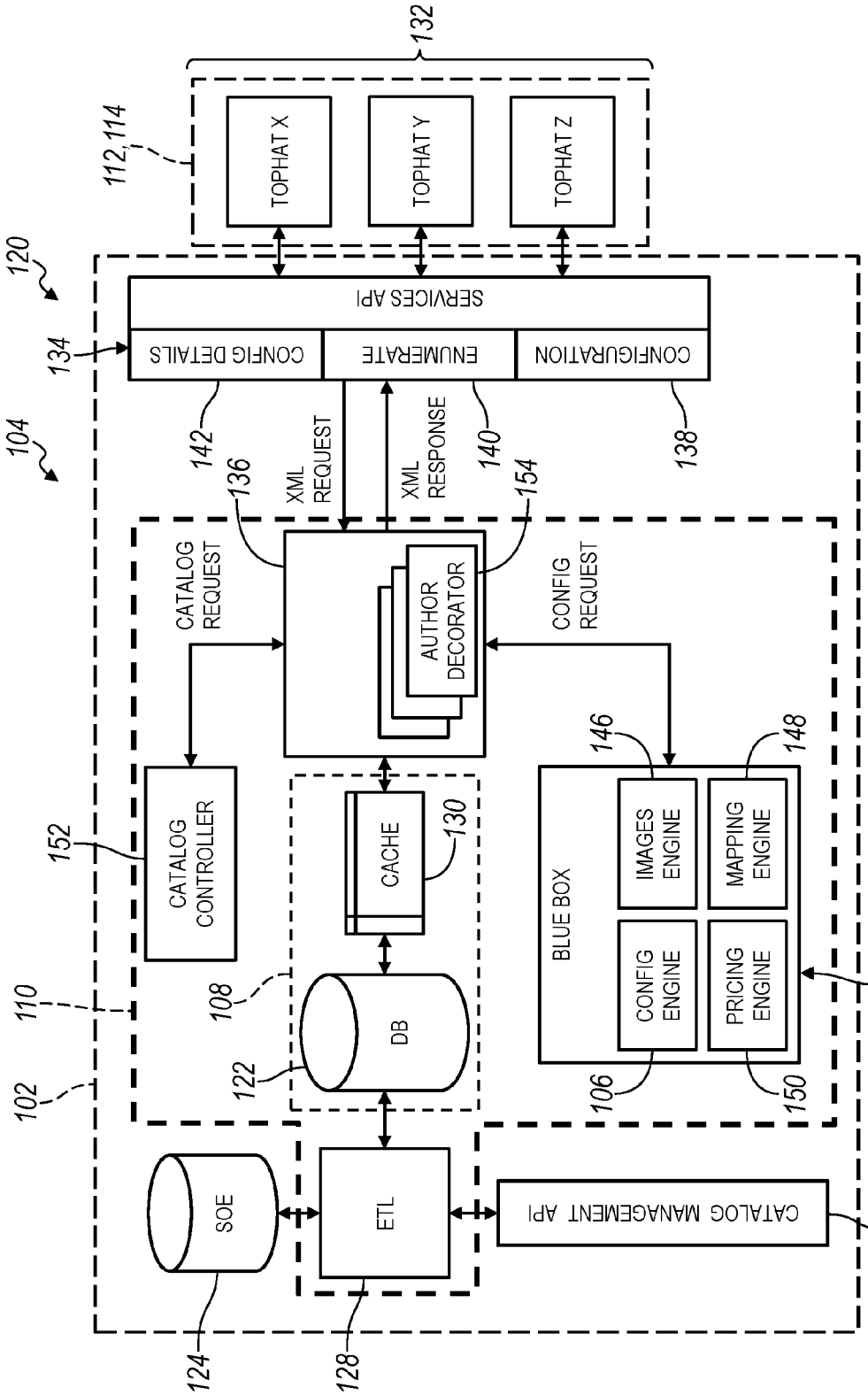

132
112,114
TOPHAT X
TOPHAT Y
TOPHAT Z
120
SERVICES API
134
CONFIG DETAILS
ENUMERATE
CONFIGURATION
104
142
136
XML REQUEST
XML RESPONSE
140
154
138
CATALOG REQUEST
AUTHOR DECORATOR
CONFIG REQUEST
146
148
152
CATALOG CONTROLLER
CACHE
130
BLUE BOX
IMAGES ENGINE
MAPPING ENGINE
110
108
DB
CONFIG ENGINE
PRICING ENGINE
144
102
122
106
150
SOE
ETL
CATALOG MANAGEMENT API
126
124
128

FIG. 2

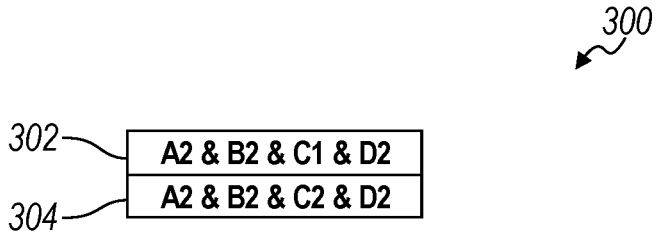


FIG. 3

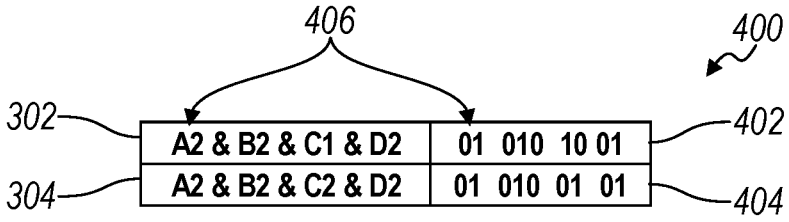


FIG. 4

500

COLUMN	0	1	2	3	4	5	6	7	8
FAMILY	A	A	B	B	B	C	C	D	D
FEATURE	A1	A2	B1	B2	B3	C1	C2	D2	D2

FIG. 5

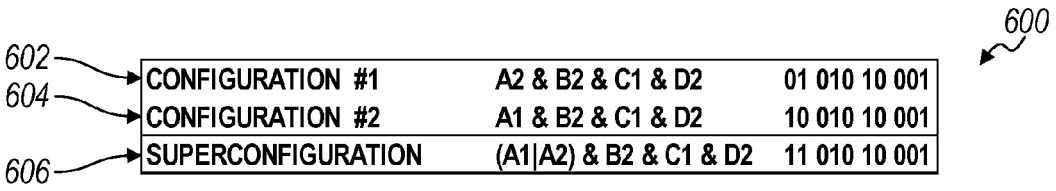


FIG. 6

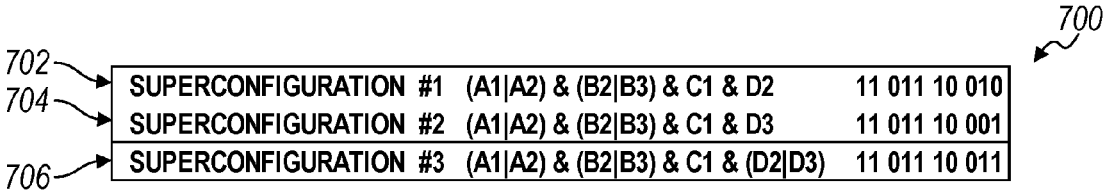


FIG. 7

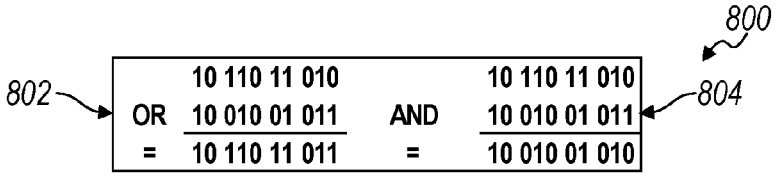


FIG. 8

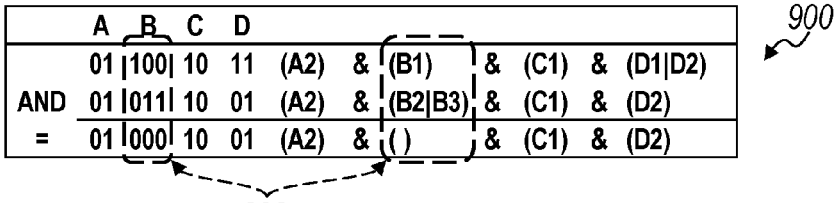


FIG. 9

1000

1308

PKG.400	PKG.401	PKG.402	RADIO.STD	RADIO.DLX	RADIO.NAV	PAINT.WHITE	PAINT.RED	PAINT.BLUE	TRIM.STONE	TRIM.CHARCOAL	TRIM.RUBY	MOONRF.LESS	MOONRF.VISTA
1	0	0	1	0	0	1	1	0	1	0	0	1	0
0	1	1	0	1	1	0	0	1	0	1	0	1	1
0	1	1	0	1	0	0	1	1	0	0	1	1	1
0	1	0	0	0	1	0	1	1	0	0	1	1	1
0	0	1	0	0	1	0	1	0	0	0	1	1	1

FIG. 10

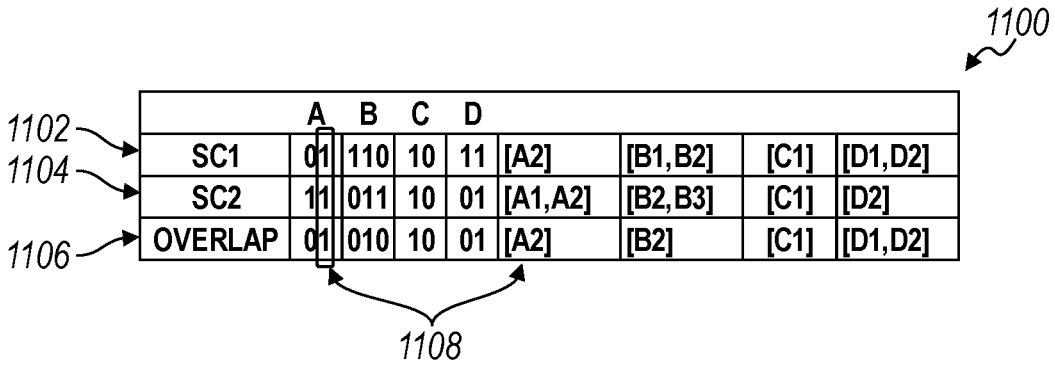


FIG. 11

1200

FEATURE COMBINATION	FEATURE MASK				
	A	B	C	D	E
A2 OR A3	011	111	111	111	11
B3	111	001	111	111	11
(A2 OR A3) AND B3	011	001	111	111	11

FIG. 12

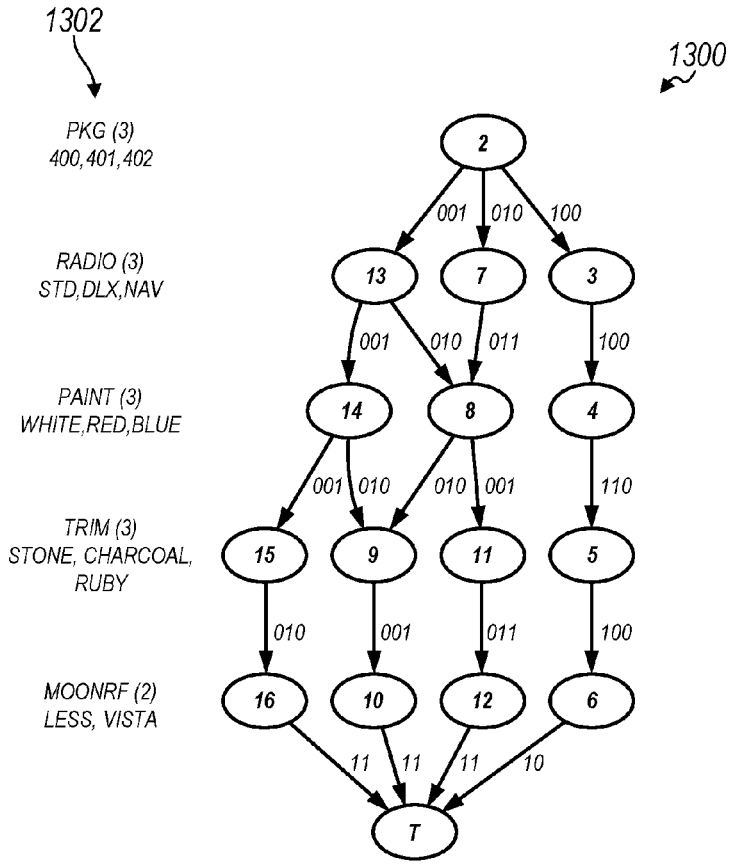


FIG. 13

1400

VEHICLE	# FAMILIES	# CONFIGURATIONS	MATRIX ROWS	MDD NODES	MATRIX SIZE	MDD SIZE
VEHICLE A	28	813,568	19	19	20KB	20KB
VEHICLE B	84	147,348,398,616,576	264,358	11,960	23,026KB	766KB
VEHICLE C	92	27,254,571,868,219,300	OUT OF MEMORY	4,860	N/A	313KB

FIG. 14

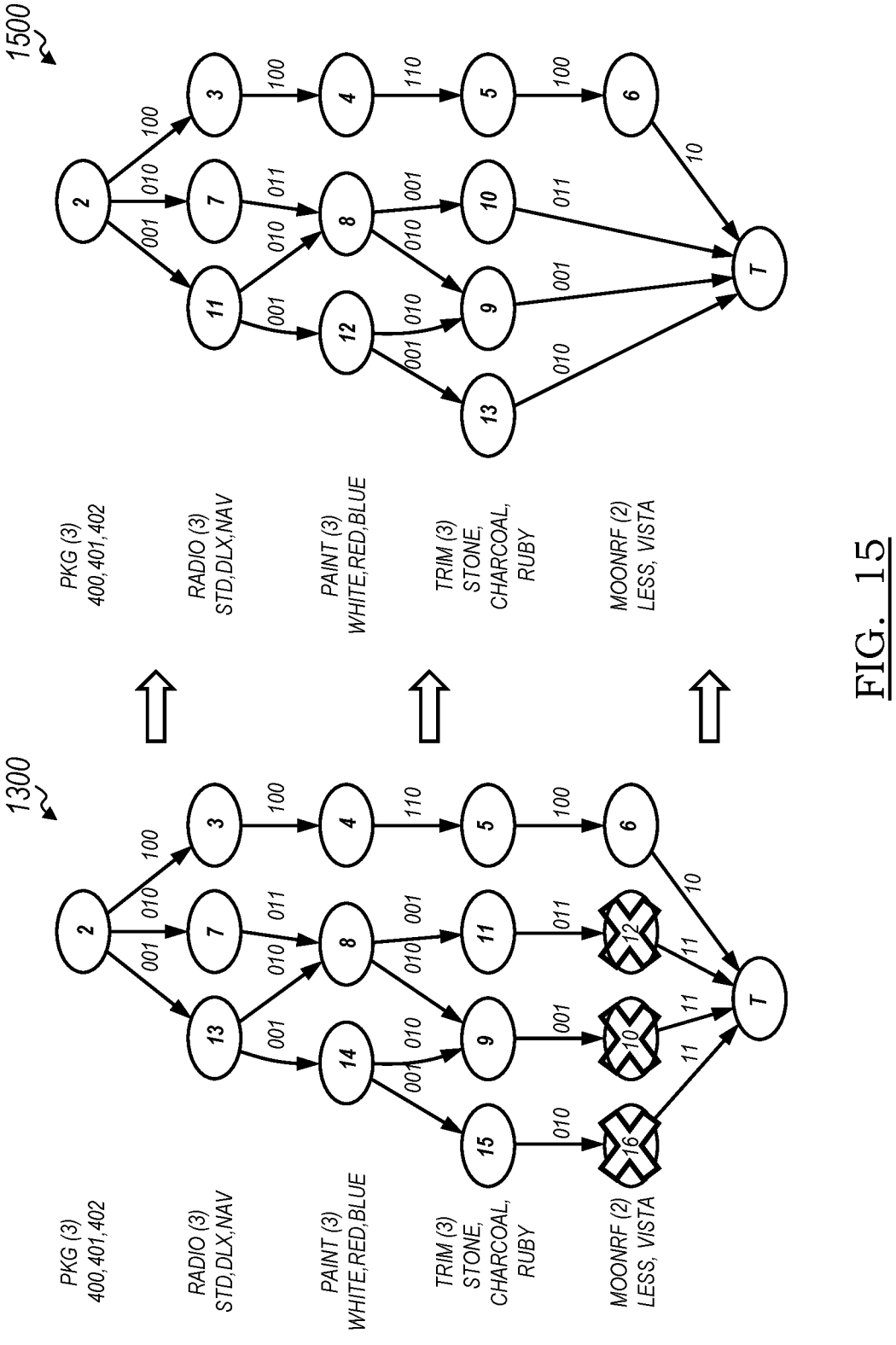

1500
1300
PKG (3)
400,401,402
RADIO (3)
STD,DLX,NAV
PAINT (3)
WHITE,RED,BLUE
TRIM (3)
STONE,
CHARCOAL,
RUBY
MOONRF (2)
LESS, VISTA
T

FIG. 15

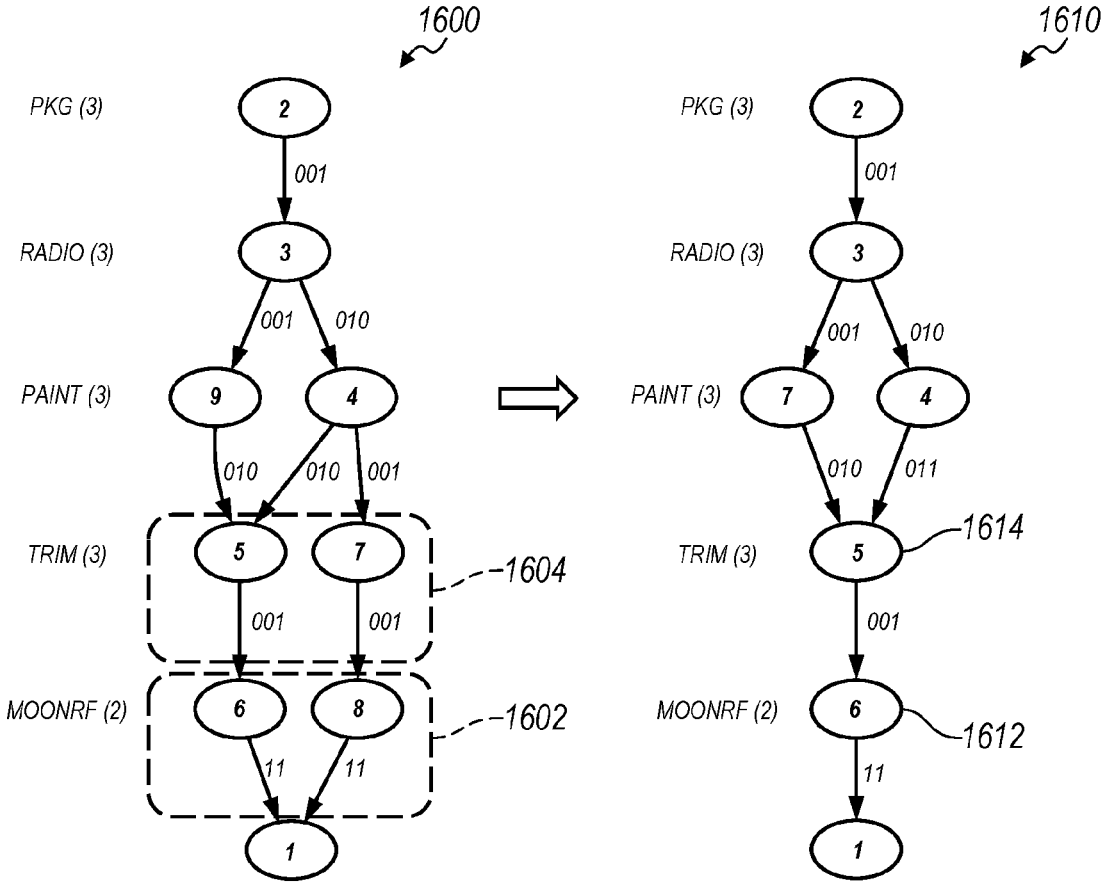

1600
PKG (3)
2
001
RADIO (3)
3
001
010
PAINT (3)
9
4
010
010
001
TRIM (3)
5
7
1604
001
001
MOONRF (2)
6
8
1602
11
11
1
1610
PKG (3)
2
001
RADIO (3)
3
001
010
PAINT (3)
7
4
010
011
TRIM (3)
5
1614
001
MOONRF (2)
6
1612
11
1

FIG. 16

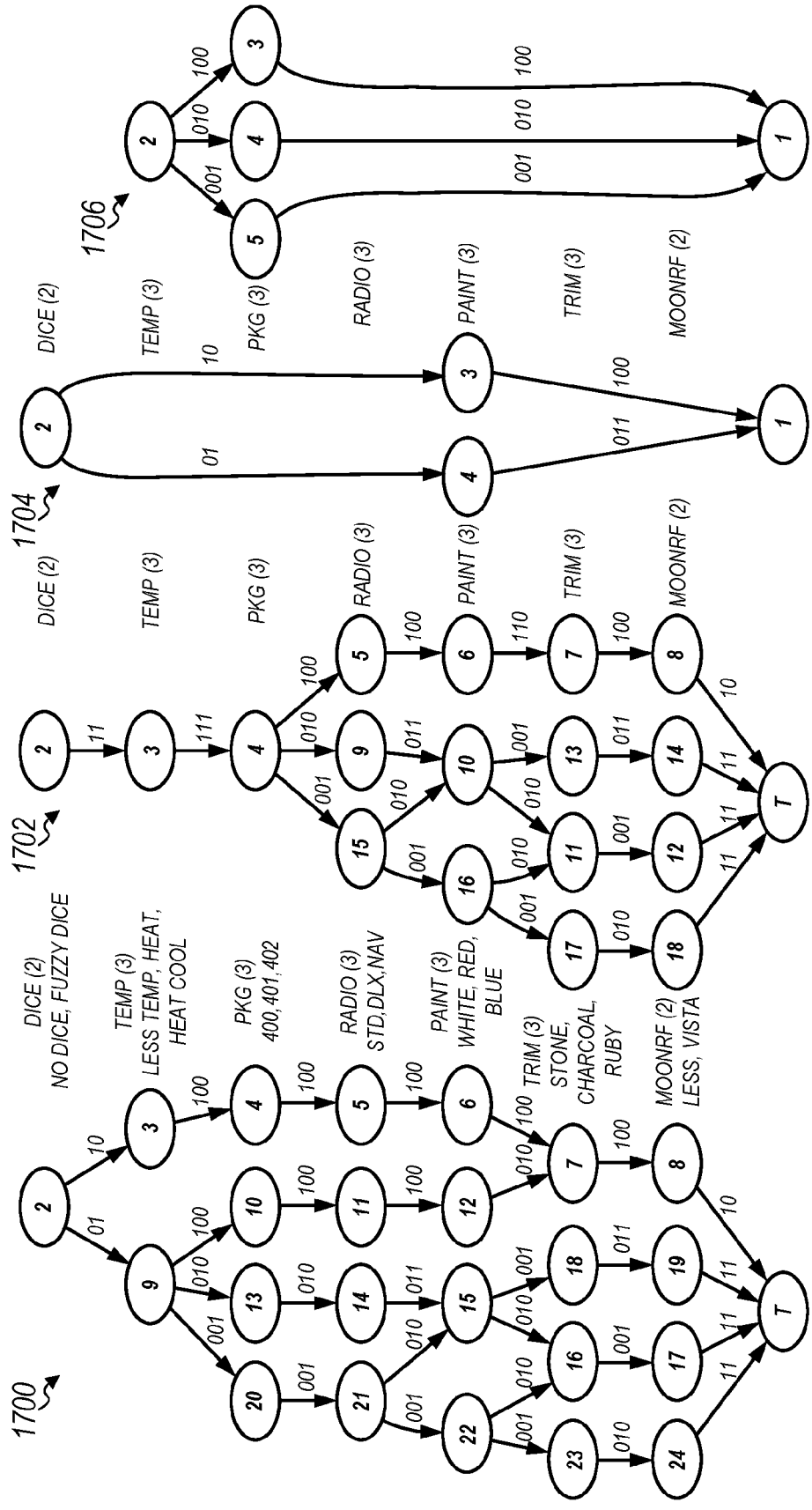

1700
1702
1704
1706
DICE (2)
NO DICE, FUZZY DICE
TEMP (3)
LESS TEMP, HEAT, HEAT COOL
PKG (3)
400,401,402
RADIO (3)
STD,DLX,NAV
PAINT (3)
WHITE, RED, BLUE
TRIM (3)
STONE, CHARCOAL, RUBY
MOONRF (2)
LESS, VISTA
T

FIG. 17

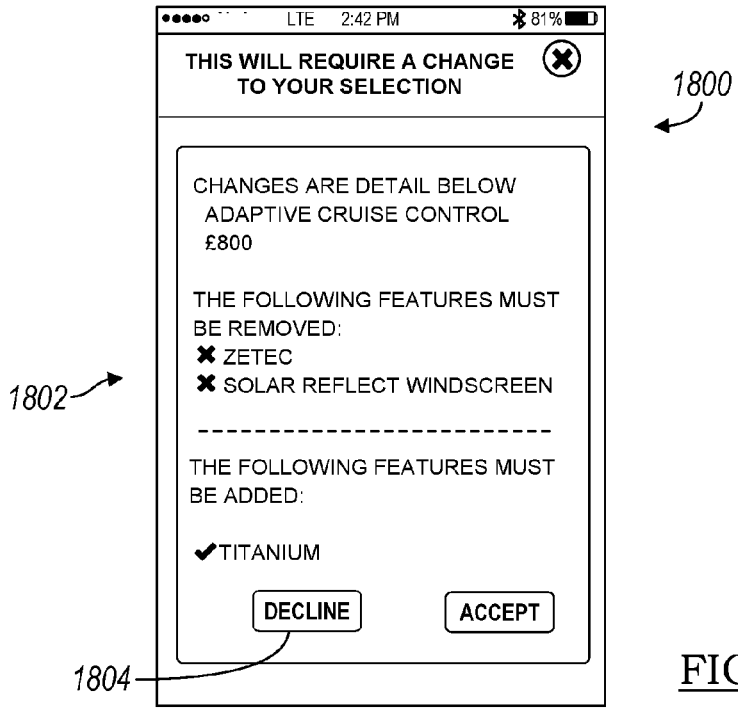

LTE 2:42 PM 81%
THIS WILL REQUIRE A CHANGE TO YOUR SELECTION
1800
CHANGES ARE DETAIL BELOW
ADAPTIVE CRUISE CONTROL
£800
THE FOLLOWING FEATURES MUST BE REMOVED:
ZETEC
SOLAR REFLECT WINDSCREEN
THE FOLLOWING FEATURES MUST BE ADDED:
TITANIUM
DECLINE
ACCEPT
1802
1804

FIG. 18

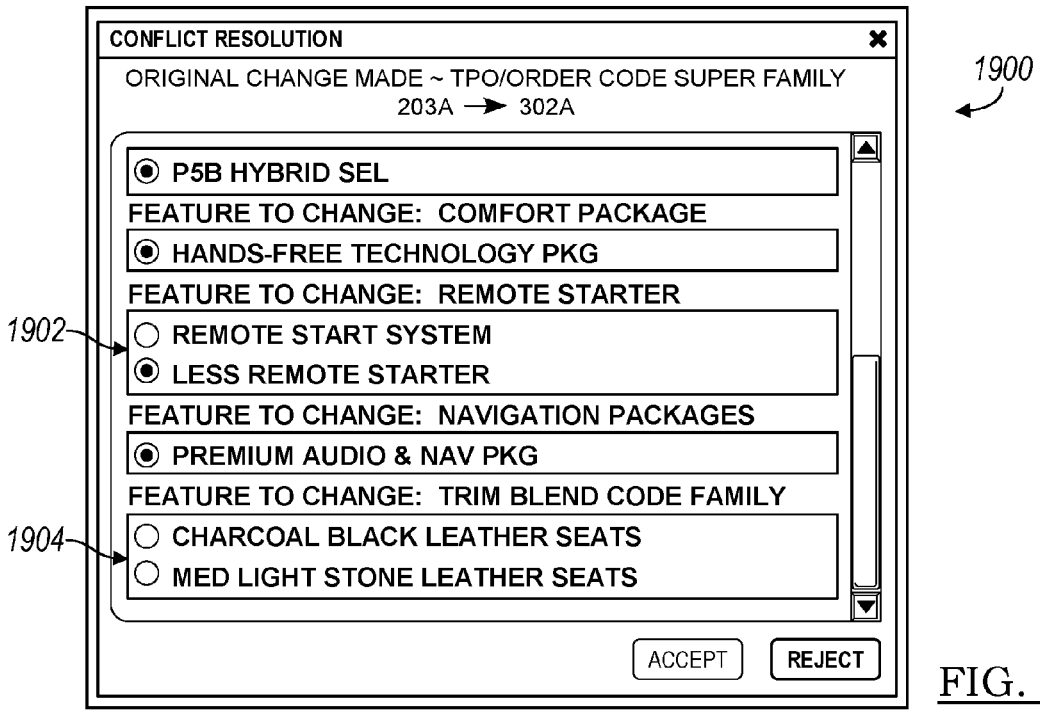

CONFLICT RESOLUTION
ORIGINAL CHANGE MADE ~ TPO/ORDER CODE SUPER FAMILY
203A → 302A
1900
P5B HYBRID SEL
FEATURE TO CHANGE: COMFORT PACKAGE
HANDS-FREE TECHNOLOGY PKG
FEATURE TO CHANGE: REMOTE STARTER
REMOTE START SYSTEM
LESS REMOTE STARTER
FEATURE TO CHANGE: NAVIGATION PACKAGES
PREMIUM AUDIO & NAV PKG
FEATURE TO CHANGE: TRIM BLEND CODE FAMILY
CHARCOAL BLACK LEATHER SEATS
MED LIGHT STONE LEATHER SEATS
ACCEPT
REJECT
1902
1904

FIG. 19

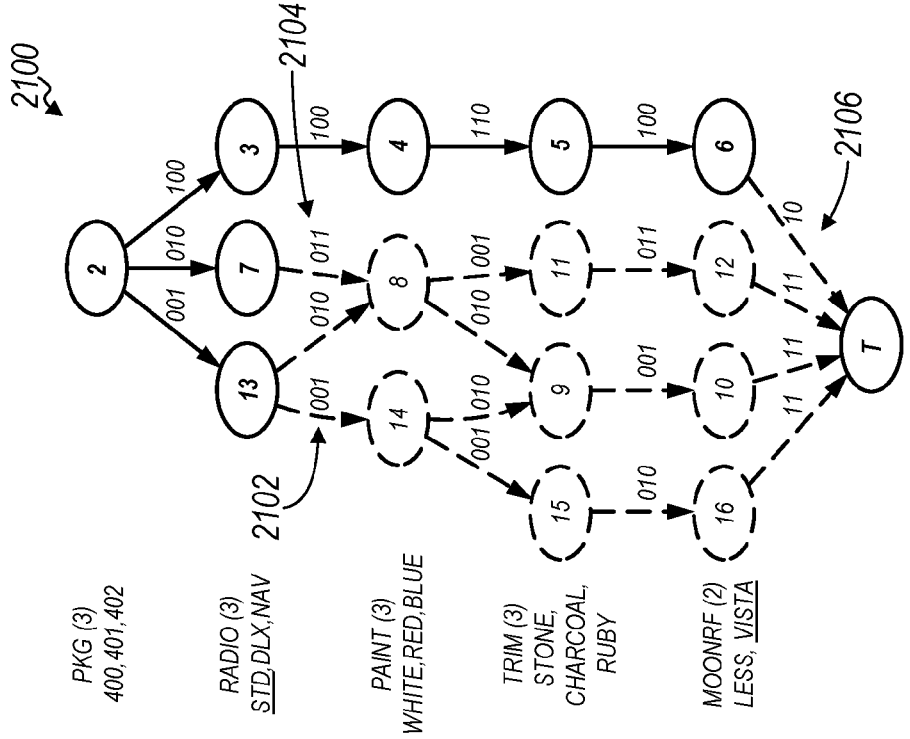

2100
2104
2106
2102
PKG (3)
400,401,402
RADIO (3)
STD,DLX,NAV
PAINT (3)
WHITE,RED,BLUE
TRIM (3)
STONE,
CHARCOAL,
RUBY
MOONRF (2)
LESS, VISTA
T

FIG. 21

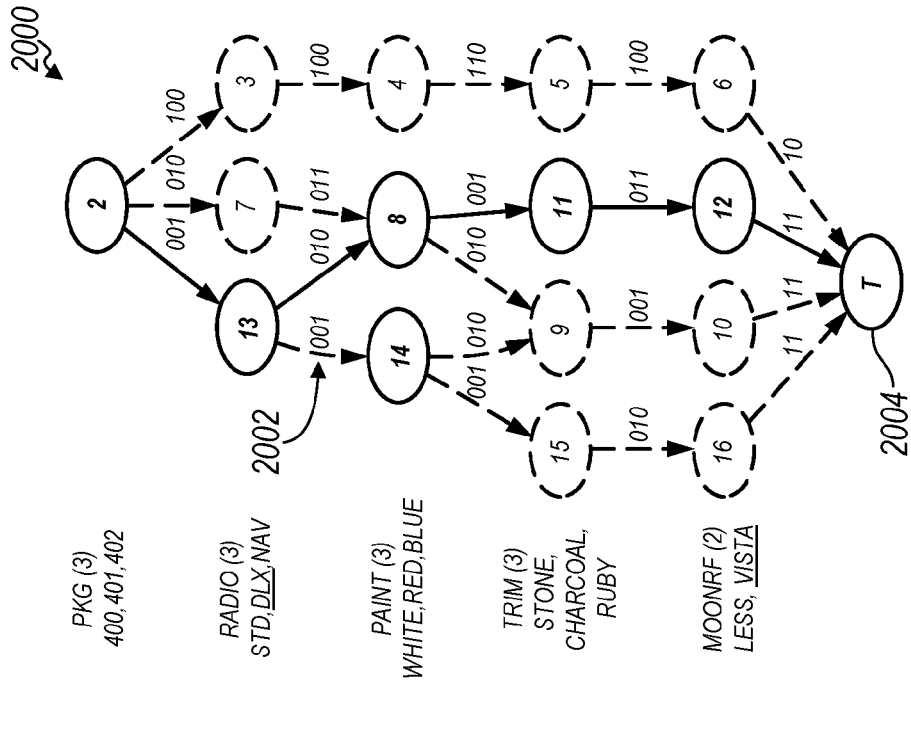

2000
2002
2004
PKG (3)
400,401,402
RADIO (3)
STD,DLX,NAV
PAINT (3)
WHITE,RED,BLUE
TRIM (3)
STONE,
CHARCOAL,
RUBY
MOONRF (2)
LESS, VISTA
T

FIG. 20

2200

PKG.400	PKG.401	PKG.402	RADIO.STD	RADIO.DLX	RADIO.NAV	PAINT.WHITE	PAINT.RED	PAINT.BLUE	TRIM.STONE	TRIM.CHARCOAL	TRIM.RUBY	MOONRF.LESS	MOONRF.VISTA
0	1	0	0	0	1	0	1	1	0	0	1	0	1
0	0	1	0	0	1	0	1	0	0	0	1	0	1

FIG. 22

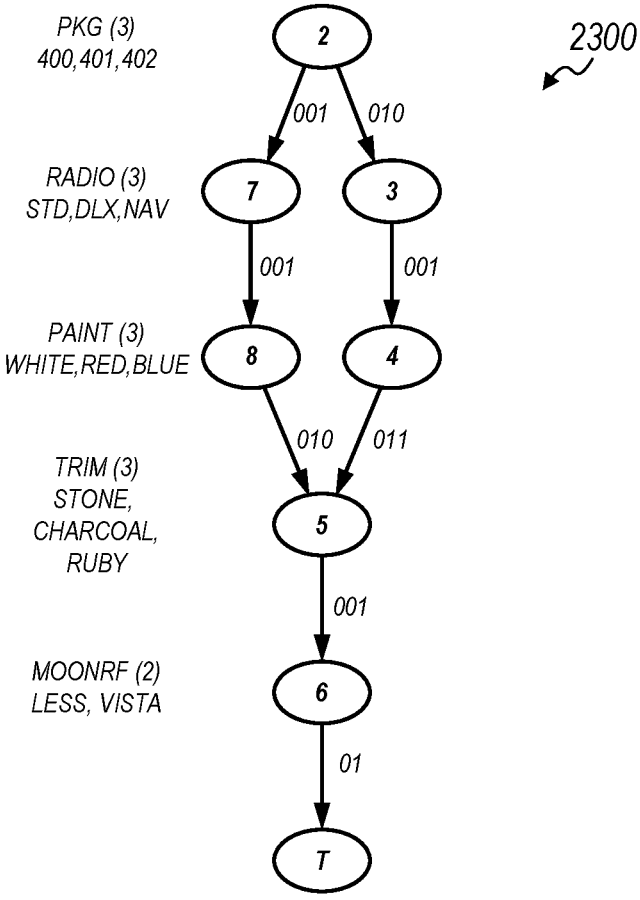


FIG. 23

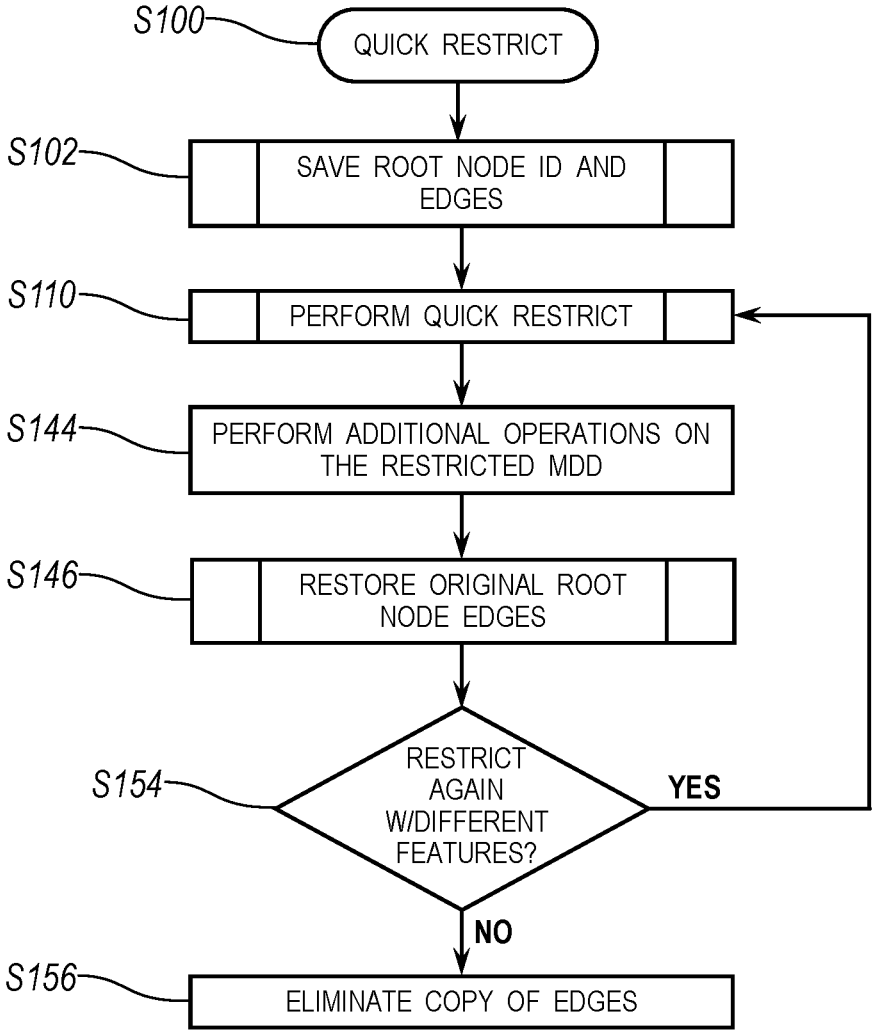

S100
QUICK RESTRICT
S102
SAVE ROOT NODE ID AND EDGES
S110
PERFORM QUICK RESTRICT
S144
PERFORM ADDITIONAL OPERATIONS ON THE RESTRICTED MDD
S146
RESTORE ORIGINAL ROOT NODE EDGES
S154
RESTRICT AGAIN W/DIFFERENT FEATURES?
YES
NO
S156
ELIMINATE COPY OF EDGES

FIG. 24

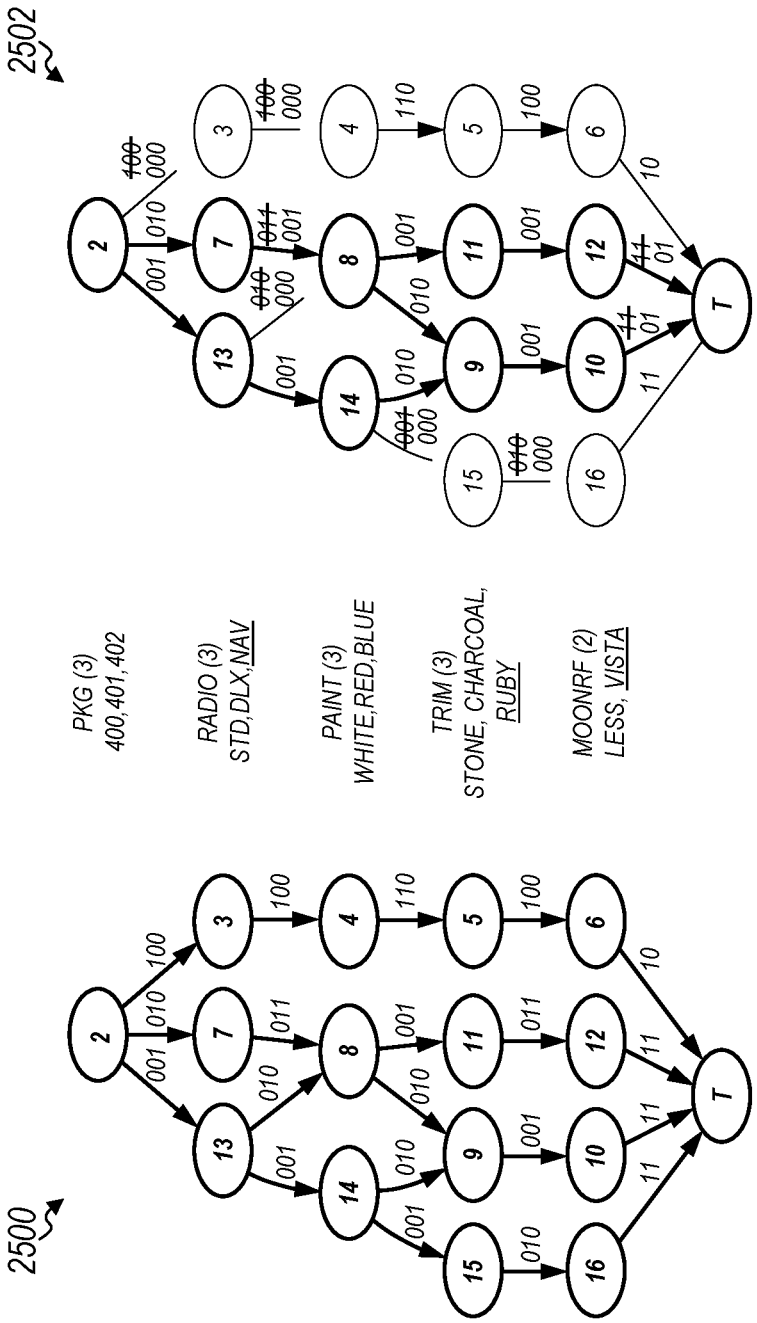

2502
2500
PKG (3)
400,401,402
RADIO (3)
STD,DLX,NAV
PAINT (3)
WHITE,RED,BLUE
TRIM (3)
STONE, CHARCOAL,
RUBY
MOONRF (2)
LESS, VISTA
T

FIG. 25

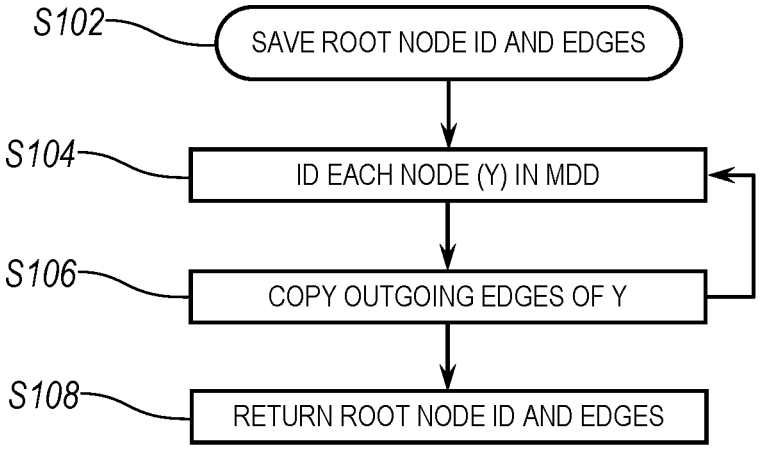

S102
SAVE ROOT NODE ID AND EDGES
S104
ID EACH NODE (Y) IN MDD
S106
COPY OUTGOING EDGES OF Y
S108
RETURN ROOT NODE ID AND EDGES

FIG. 26

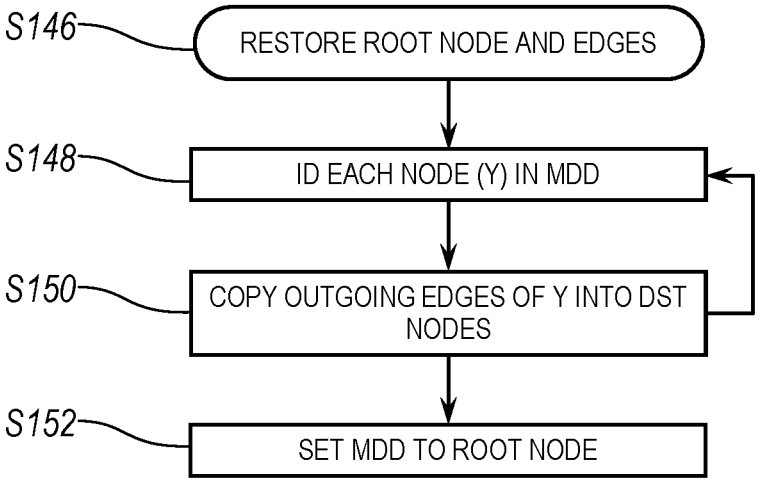

S146
RESTORE ROOT NODE AND EDGES
S148
ID EACH NODE (Y) IN MDD
S150
COPY OUTGOING EDGES OF Y INTO DST NODES
S152
SET MDD TO ROOT NODE

FIG. 28

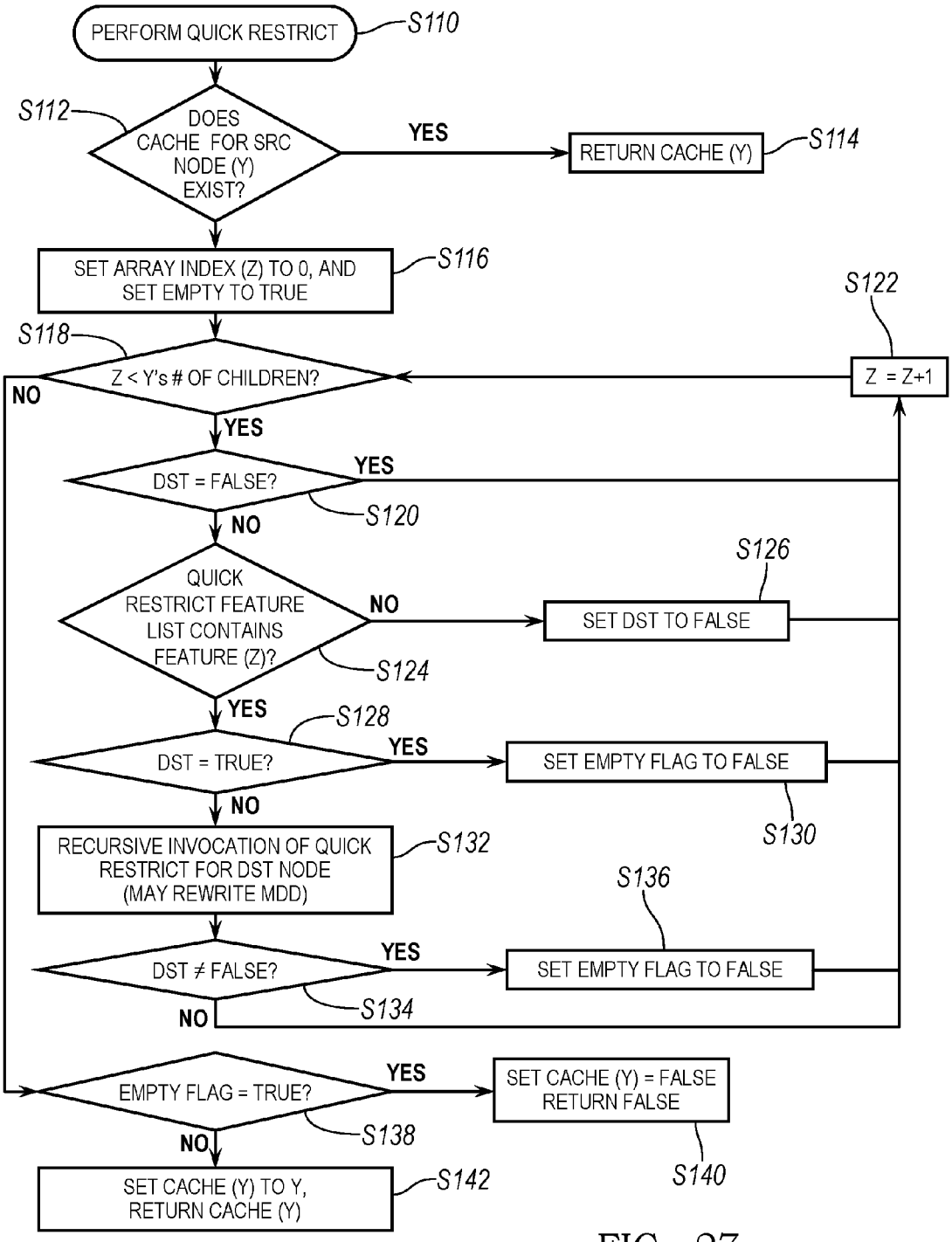

PERFORM QUICK RESTRICT
S110
S112
DOES CACHE FOR SRC NODE (Y) EXIST?
YES
RETURN CACHE (Y)
S114
SET ARRAY INDEX (Z) TO 0, AND SET EMPTY TO TRUE
S116
S118
Z < Y's # OF CHILDREN?
NO
YES
S122
Z = Z+1
DST = FALSE?
YES
S120
NO
S126
QUICK RESTRICT FEATURE LIST CONTAINS FEATURE (Z)?
NO
SET DST TO FALSE
S124
YES
S128
DST = TRUE?
YES
SET EMPTY FLAG TO FALSE
S130
NO
RECURSIVE INVOCATION OF QUICK RESTRICT FOR DST NODE (MAY REWRITE MDD)
S132
S136
DST ≠ FALSE?
YES
SET EMPTY FLAG TO FALSE
S134
NO
EMPTY FLAG = TRUE?
YES
SET CACHE (Y) = FALSE RETURN FALSE
S138
NO
S140
SET CACHE (Y) TO Y, RETURN CACHE (Y)
S142

FIG. 27

2900

TIMES (MS) FOR 1000 OPERATIONS			QUICK-RESTRICT WITH DOMAIN AND RESET		READ-ONLY RESTRICTED DOMAIN	
MARKET	VEHICLE	MDD # NODES	TOTAL	AVERAGE	TOTAL	AVARAGE
US	A	1,023	79	0.079	75	0.075
US	B	8,842	604	0.604	194	0.194
US	C	96,576	5,656	5.656	1,319	1.319
GREAT BRITAIN	D	235,508	13,494	13.494	3,672	3.672
GERMANY	E	491,712	24,608	24.608	6,418	6.418

FIG. 29

3000

PKG.400	PKG.401	PKG.402	TRIM.STONE	TRIM.CHARCOAL	TRIM.RUBY
1	0	0	1	0	0
0	1	1	0	1	0
0	1	1	0	0	1
0	1	0	0	0	1
0	0	1	0	0	1

FIG. 30

3100

PKG.400	PKG.401	PKG.402	TRIM.STONE	TRIM.CHARCOAL	TRIM.RUBY
1	0	0	1	0	0
0	1	1	0	1	1

FIG. 31

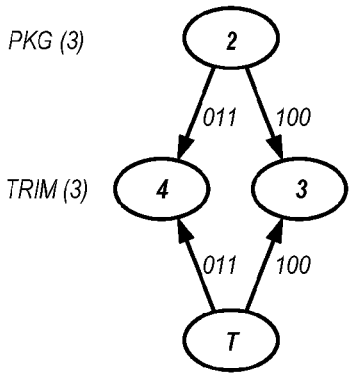


FIG. 32

3300

[BLUE, CHARCOAL]
[BLUE, RUBY]
[RED, RUBY]
[RED, STONE]
[WHITE, STONE]

FIG. 33

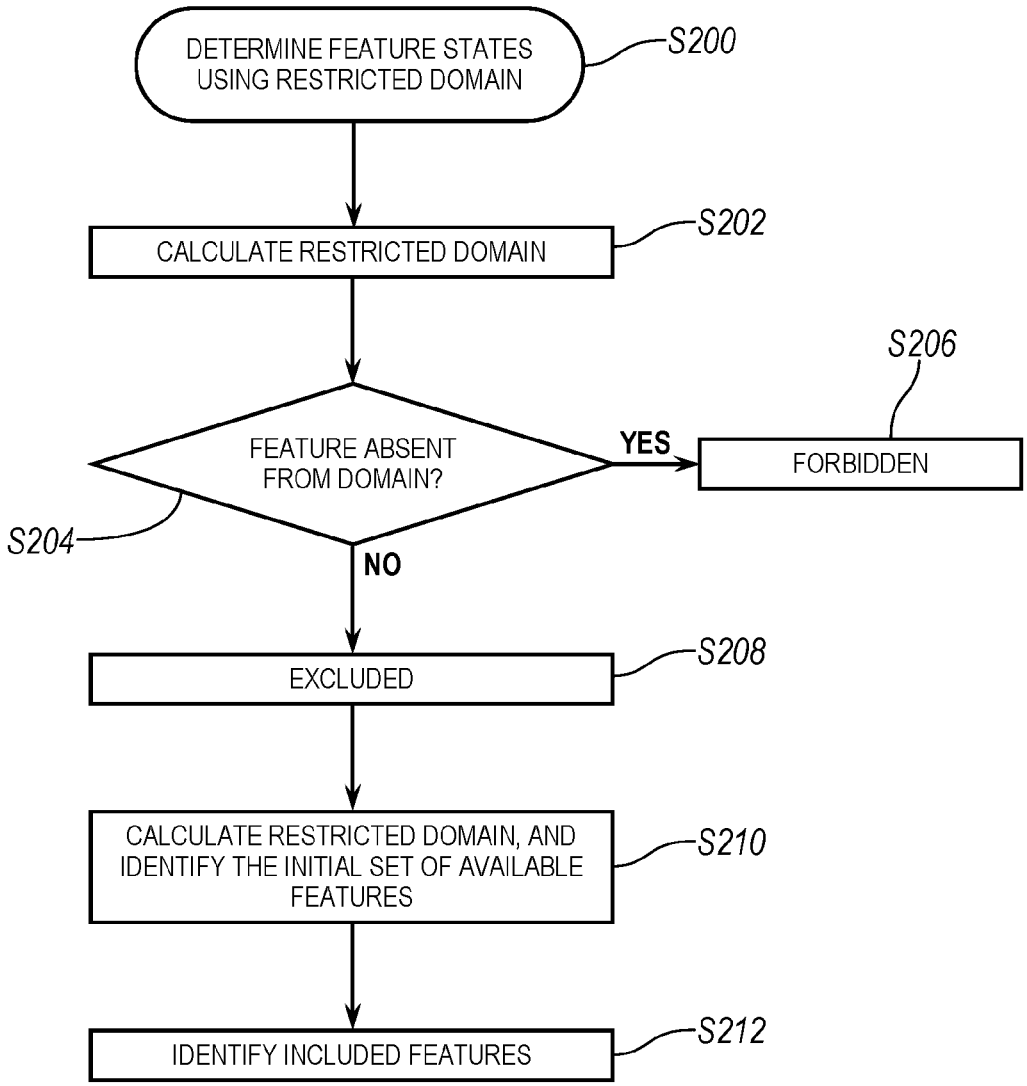

DETERMINE FEATURE STATES USING RESTRICTED DOMAIN
S200
CALCULATE RESTRICTED DOMAIN
S202
FEATURE ABSENT FROM DOMAIN?
S204
YES
S206
FORBIDDEN
NO
EXCLUDED
S208
CALCULATE RESTRICTED DOMAIN, AND IDENTIFY THE INITIAL SET OF AVAILABLE FEATURES
S210
IDENTIFY INCLUDED FEATURES
S212

FIG. 34

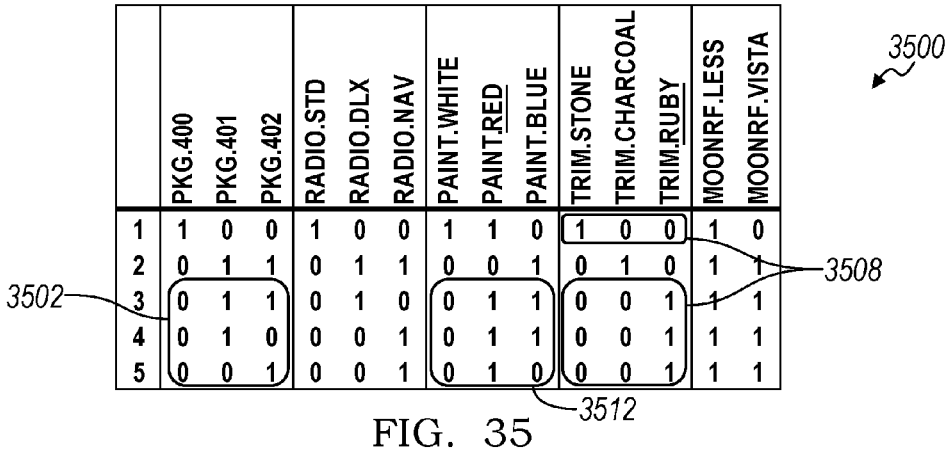


FIG. 35

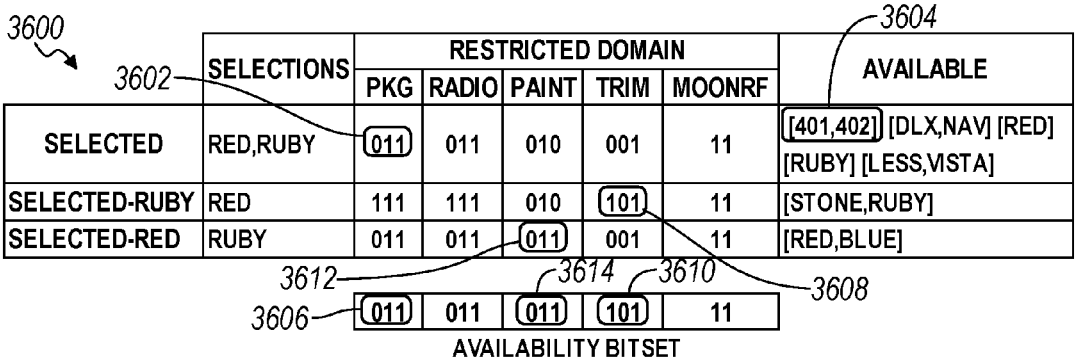


FIG. 36

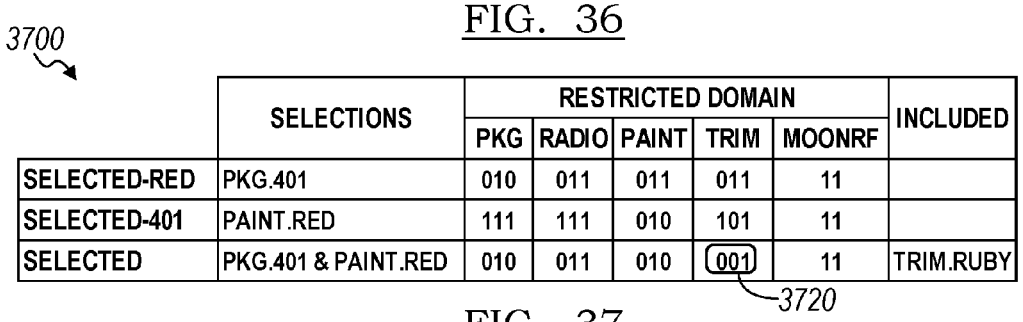


FIG. 37

3800

STATE	FEATURES
SELECTED	PAINT.RED TRIM.RUBY
INCLUDED	
AVAILABLE	PKG.401, PKG.402, RADIO.DLX, RADIO.NAV, PAINT.BLUE, TRIM.STONE, MOONRF.LESS, MOONRF.VISTA
EXCLUDED	PKG.400, RADIO.STD, PAINT.WHITE, TRIM.CHARCOAL

FIG. 38

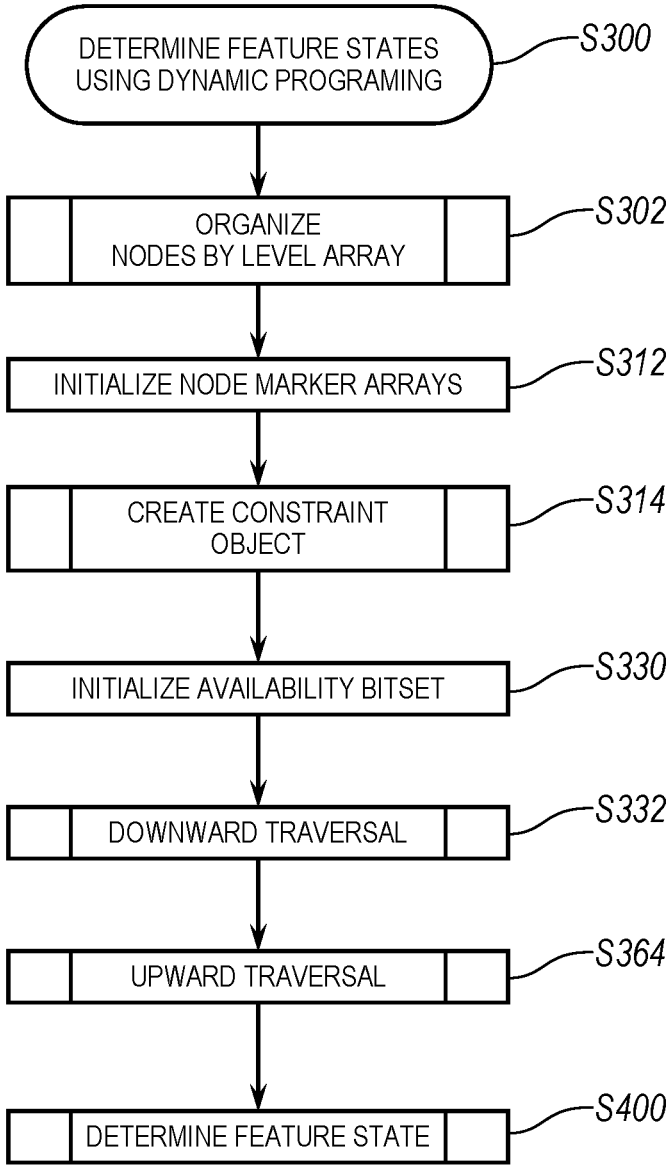

DETERMINE FEATURE STATES USING DYNAMIC PROGRAMING
S300
ORGANIZE NODES BY LEVEL ARRAY
S302
INITIALIZE NODE MARKER ARRAYS
S312
CREATE CONSTRAINT OBJECT
S314
INITIALIZE AVAILABILITY BITSET
S330
DOWNWARD TRAVERSAL
S332
UPWARD TRAVERSAL
S364
DETERMINE FEATURE STATE
S400

FIG. 39

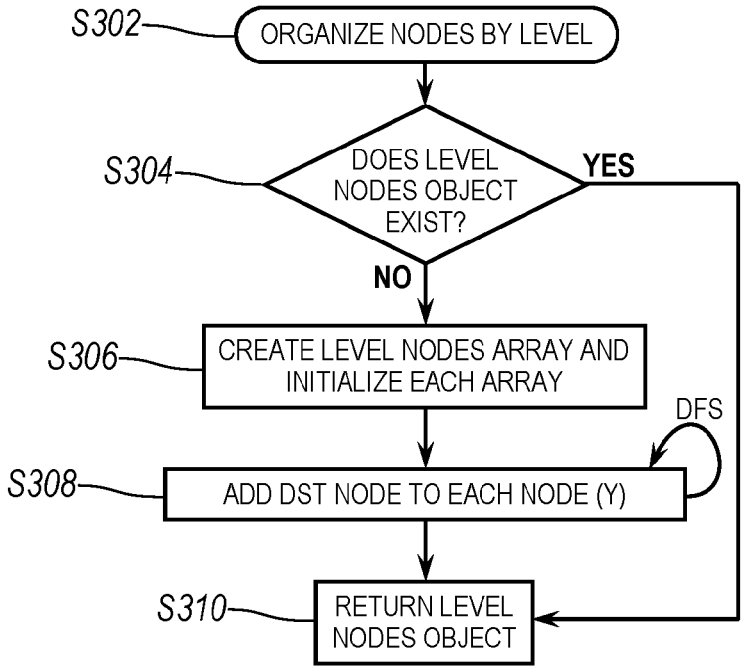

S302
ORGANIZE NODES BY LEVEL
S304
DOES LEVEL NODES OBJECT EXIST?
YES
NO
S306
CREATE LEVEL NODES ARRAY AND INITIALIZE EACH ARRAY
DFS
S308
ADD DST NODE TO EACH NODE (Y)
S310
RETURN LEVEL NODES OBJECT

FIG. 40

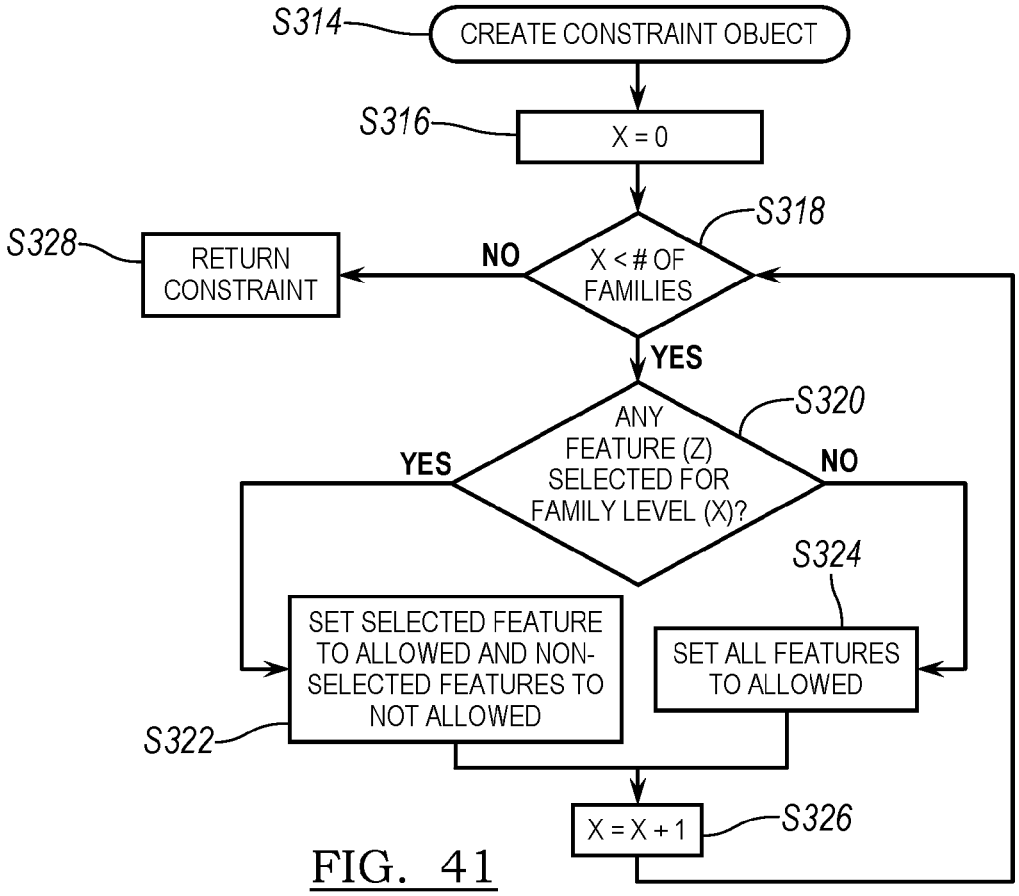

S314
CREATE CONSTRAINT OBJECT
S316
X = 0
S318
X < # OF FAMILIES
NO
S328
RETURN CONSTRAINT
YES
S320
ANY FEATURE (Z) SELECTED FOR FAMILY LEVEL (X)?
YES
NO
S322
SET SELECTED FEATURE TO ALLOWED AND NON-SELECTED FEATURES TO NOT ALLOWED
S324
SET ALL FEATURES TO ALLOWED
X = X + 1
S326

FIG. 41

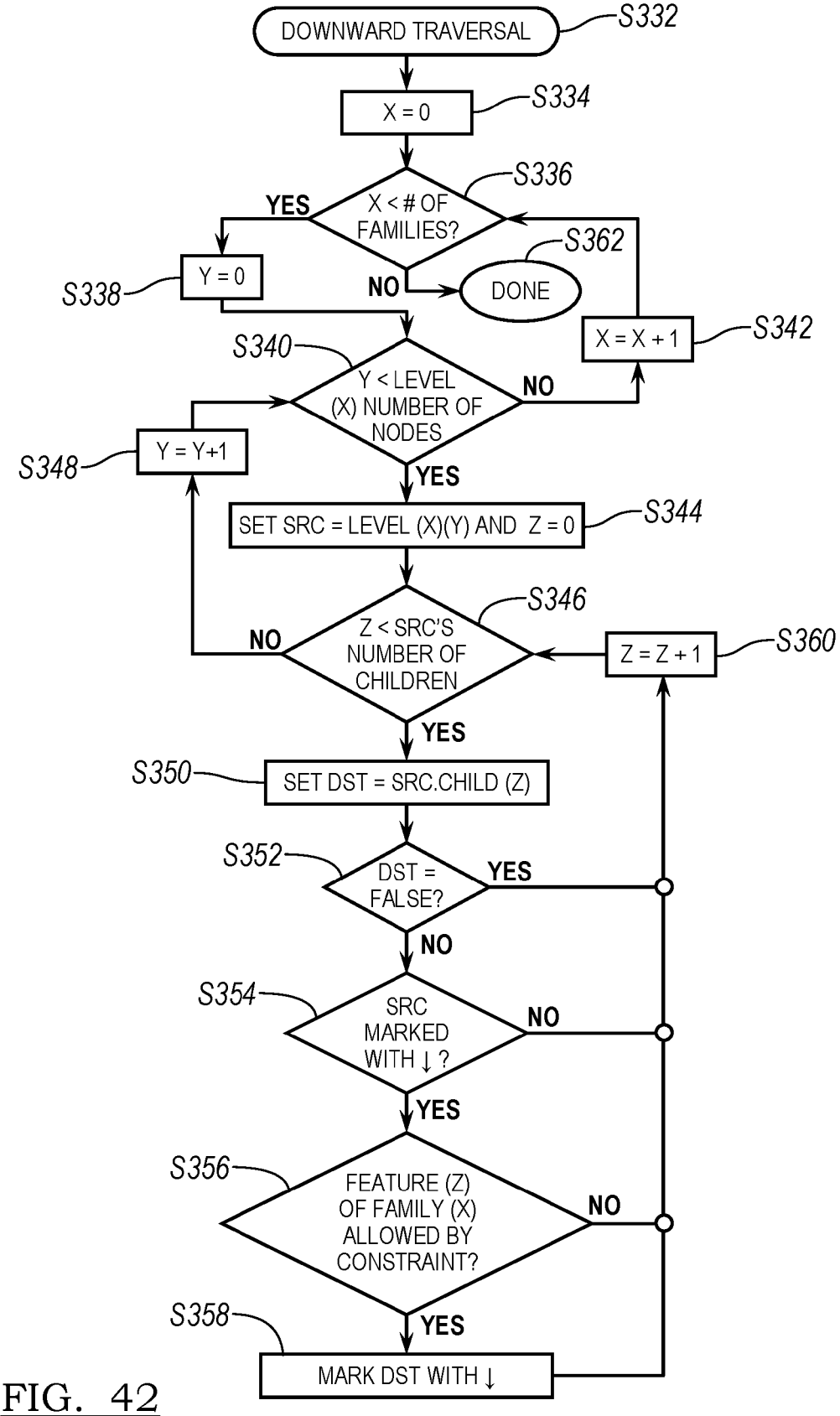

DOWNWARD TRAVERSAL
S332
X = 0
S334
X < # OF FAMILIES?
S336
YES
NO
DONE
S362
Y = 0
S338
X = X + 1
S342
Y < LEVEL (X) NUMBER OF NODES
S340
NO
Y = Y+1
S348
YES
SET SRC = LEVEL (X)(Y) AND Z = 0
S344
Z < SRC'S NUMBER OF CHILDREN
S346
NO
Z = Z + 1
S360
YES
SET DST = SRC.CHILD (Z)
S350
DST = FALSE?
S352
YES
NO
SRC MARKED WITH ↓ ?
S354
NO
YES
FEATURE (Z) OF FAMILY (X) ALLOWED BY CONSTRAINT?
S356
NO
YES
MARK DST WITH ↓
S358

FIG. 42

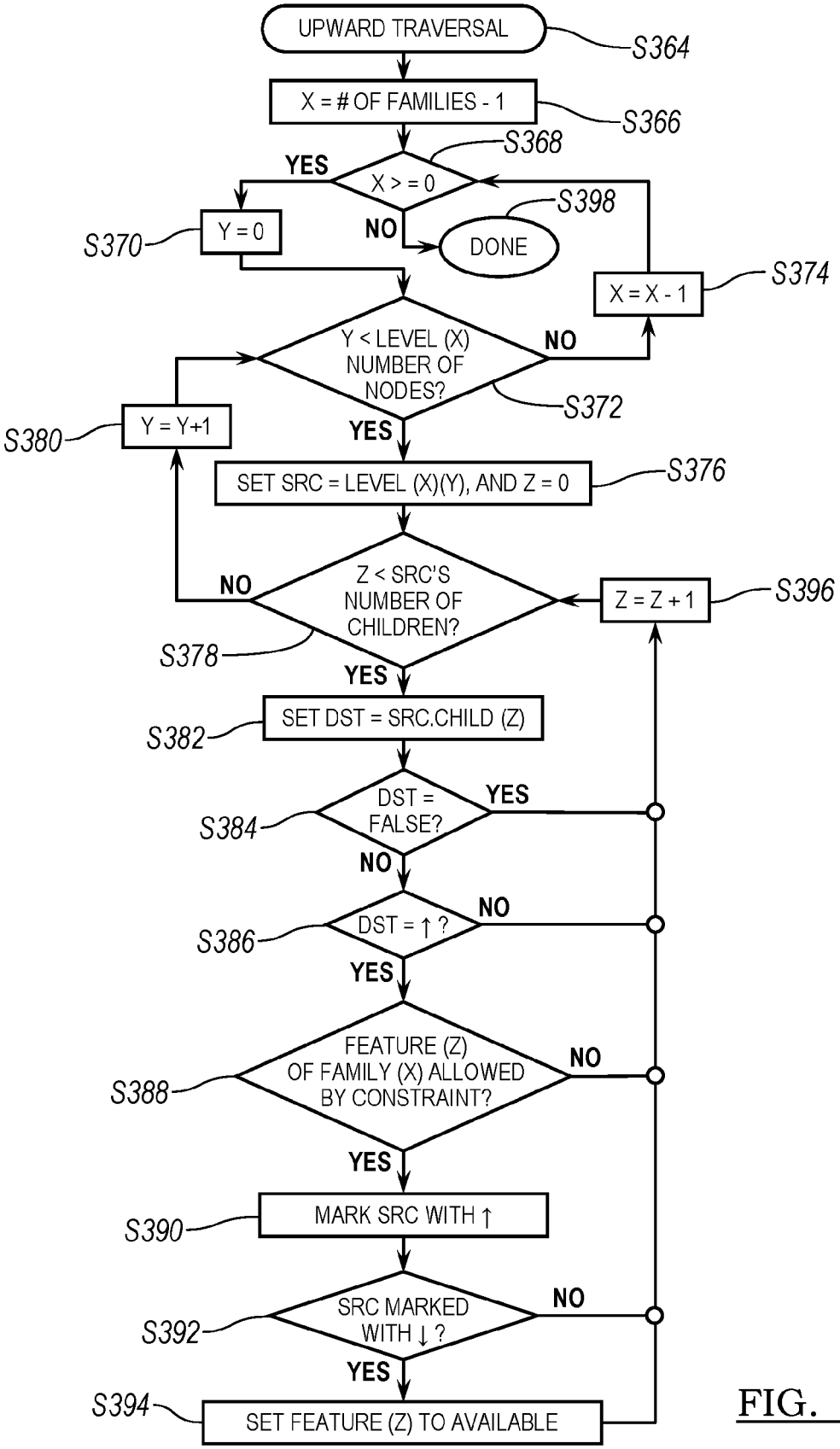

UPWARD TRAVERSAL
S364
X = # OF FAMILIES - 1
S366
S368
YES
X > = 0
S398
NO
DONE
S370
Y = 0
S374
X = X - 1
Y < LEVEL (X) NUMBER OF NODES?
NO
S372
S380
Y = Y+1
YES
SET SRC = LEVEL (X)(Y), AND Z = 0
S376
Z < SRC'S NUMBER OF CHILDREN?
NO
Z = Z + 1
S396
S378
YES
SET DST = SRC.CHILD (Z)
S382
DST = FALSE?
YES
S384
NO
DST = ↑ ?
NO
S386
YES
FEATURE (Z) OF FAMILY (X) ALLOWED BY CONSTRAINT?
NO
S388
YES
MARK SRC WITH ↑
S390
SRC MARKED WITH ↓ ?
NO
S392
YES
SET FEATURE (Z) TO AVAILABLE
S394

FIG. 43

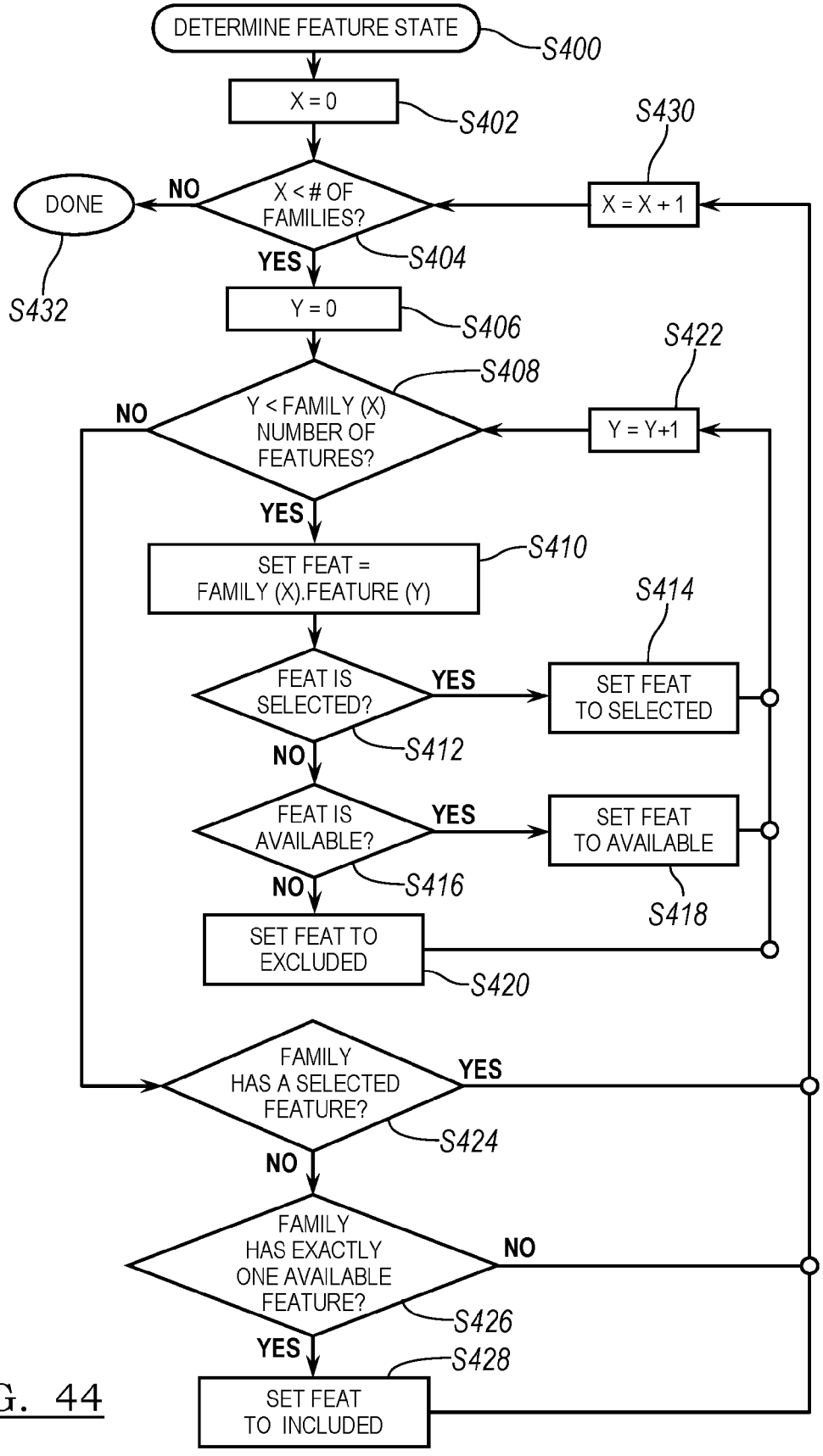

DETERMINE FEATURE STATE
S400
X = 0
S402
S430
X = X + 1
DONE
NO
X < # OF FAMILIES?
S404
YES
S432
Y = 0
S406
S422
S408
NO
Y < FAMILY (X) NUMBER OF FEATURES?
Y = Y+1
YES
SET FEAT = FAMILY (X).FEATURE (Y)
S410
S414
FEAT IS SELECTED?
YES
SET FEAT TO SELECTED
S412
NO
FEAT IS AVAILABLE?
YES
SET FEAT TO AVAILABLE
S416
NO
S418
SET FEAT TO EXCLUDED
S420
FAMILY HAS A SELECTED FEATURE?
YES
S424
NO
FAMILY HAS EXACTLY ONE AVAILABLE FEATURE?
NO
S426
YES
S428
SET FEAT TO INCLUDED

FIG. 44

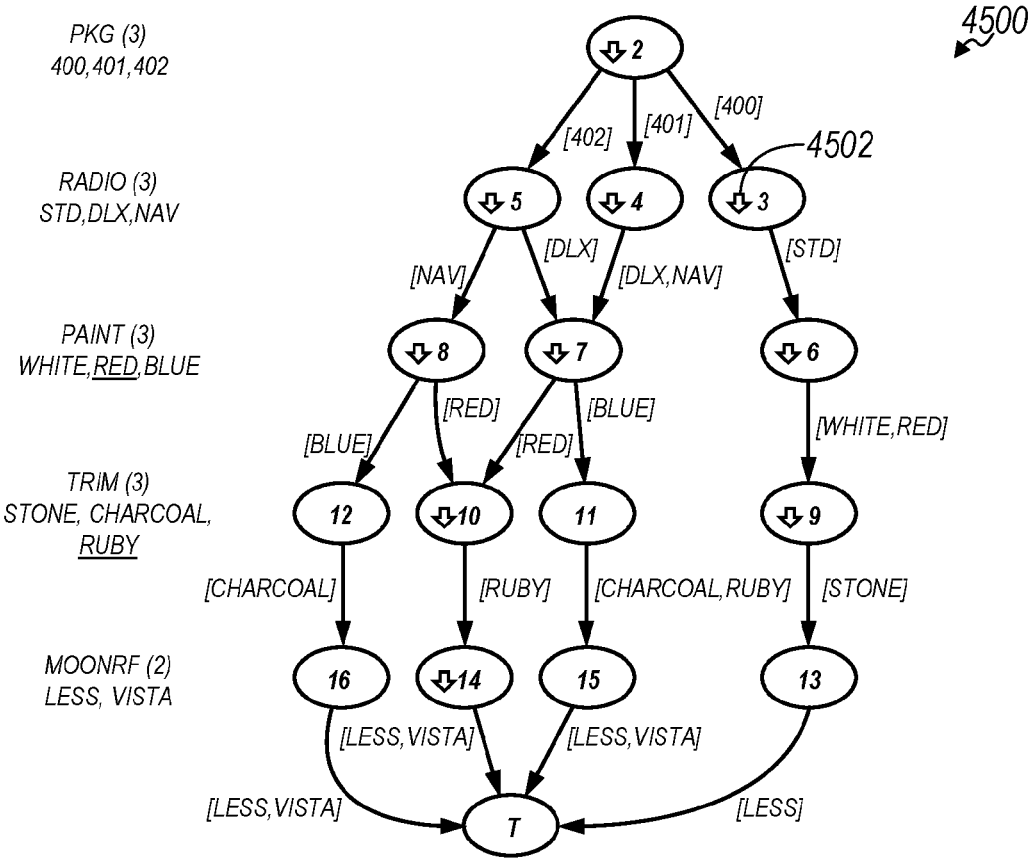

4500
PKG (3)
400,401,402
RADIO (3)
STD,DLX,NAV
PAINT (3)
WHITE,RED,BLUE
TRIM (3)
STONE, CHARCOAL,
RUBY
MOONRF (2)
LESS, VISTA
2
[402]
[401]
[400]
4502
5
4
3
[NAV]
[DLX]
[DLX,NAV]
[STD]
8
7
6
[BLUE]
[RED]
[RED]
[BLUE]
[WHITE,RED]
12
10
11
9
[CHARCOAL]
[RUBY]
[CHARCOAL,RUBY]
[STONE]
16
14
15
13
[LESS,VISTA]
[LESS,VISTA]
[LESS,VISTA]
[LESS]
T

FIG. 45

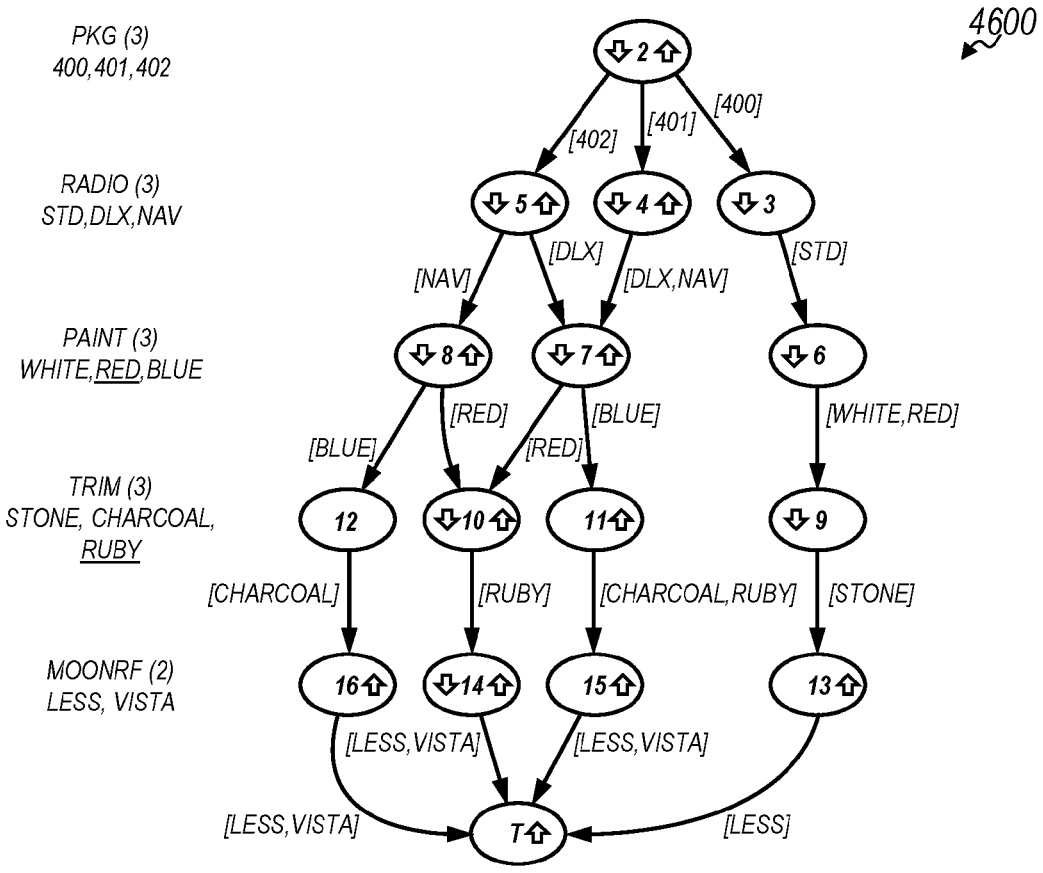

4600
PKG (3)
400,401,402
RADIO (3)
STD,DLX,NAV
PAINT (3)
WHITE,RED,BLUE
TRIM (3)
STONE, CHARCOAL,
RUBY
MOONRF (2)
LESS, VISTA
2
[402]
[401]
[400]
5
4
3
[NAV]
[DLX]
[DLX,NAV]
[STD]
8
7
6
[BLUE]
[RED]
[RED]
[BLUE]
[WHITE,RED]
12
10
11
9
[CHARCOAL]
[RUBY]
[CHARCOAL,RUBY]
[STONE]
16
14
15
13
[LESS,VISTA]
[LESS,VISTA]
[LESS,VISTA]
T
[LESS]

FIG. 46

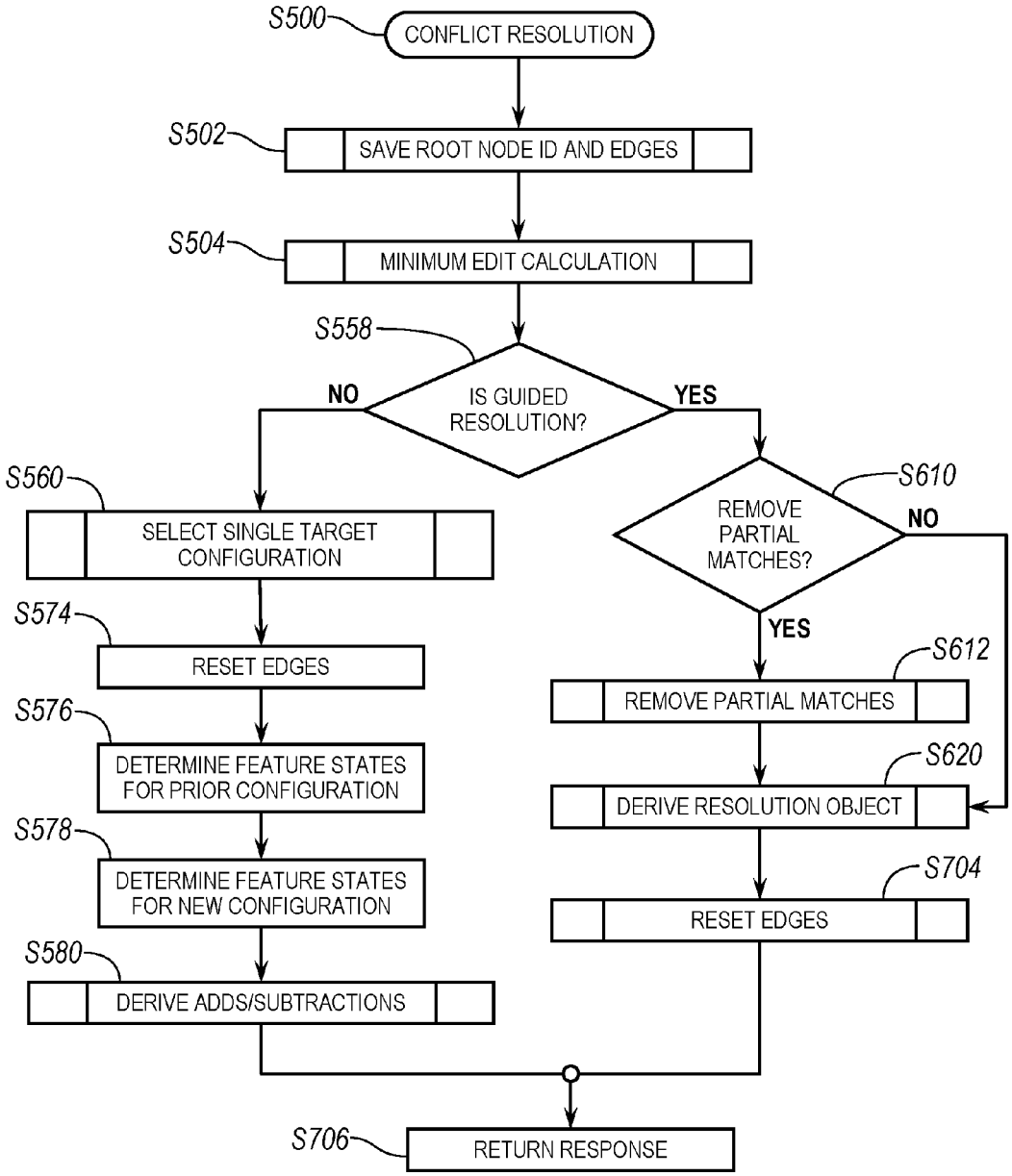

S500
CONFLICT RESOLUTION
S502
SAVE ROOT NODE ID AND EDGES
S504
MINIMUM EDIT CALCULATION
S558
IS GUIDED RESOLUTION?
NO
YES
S560
SELECT SINGLE TARGET CONFIGURATION
S574
RESET EDGES
S576
DETERMINE FEATURE STATES FOR PRIOR CONFIGURATION
S578
DETERMINE FEATURE STATES FOR NEW CONFIGURATION
S580
DERIVE ADDS/SUBTRACTIONS
S610
REMOVE PARTIAL MATCHES?
NO
YES
S612
REMOVE PARTIAL MATCHES
S620
DERIVE RESOLUTION OBJECT
S704
RESET EDGES
S706
RETURN RESPONSE

FIG. 47

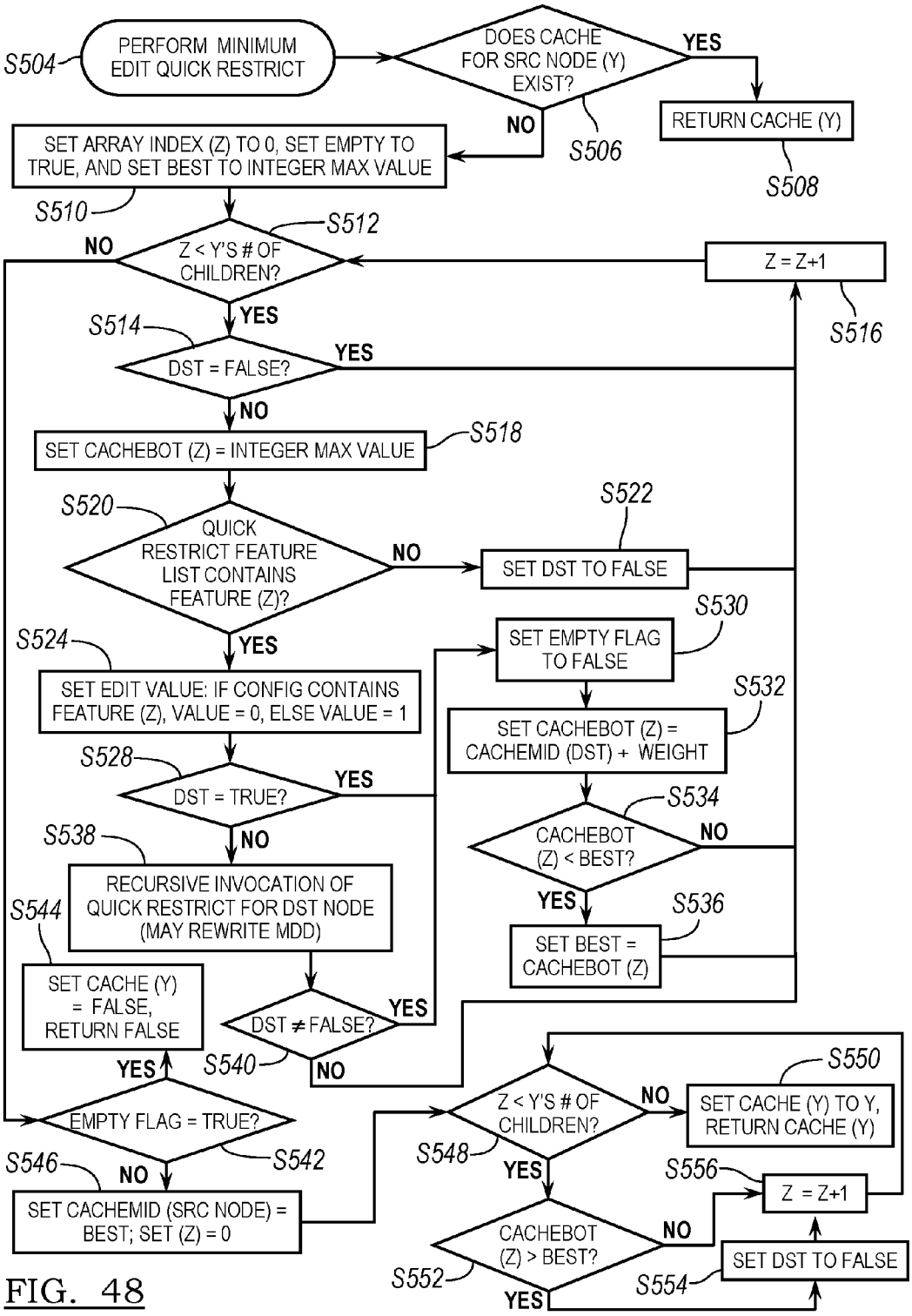

S504
PERFORM MINIMUM EDIT QUICK RESTRICT
DOES CACHE FOR SRC NODE (Y) EXIST?
YES
NO
S506
RETURN CACHE (Y)
S508
SET ARRAY INDEX (Z) TO 0, SET EMPTY TO TRUE, AND SET BEST TO INTEGER MAX VALUE
S510
S512
Z < Y'S # OF CHILDREN?
NO
YES
Z = Z+1
S516
S514
DST = FALSE?
YES
NO
SET CACHEBOT (Z) = INTEGER MAX VALUE
S518
S520
QUICK RESTRICT FEATURE LIST CONTAINS FEATURE (Z)?
NO
S522
SET DST TO FALSE
YES
S524
SET EDIT VALUE: IF CONFIG CONTAINS FEATURE (Z), VALUE = 0, ELSE VALUE = 1
S528
DST = TRUE?
YES
NO
S530
SET EMPTY FLAG TO FALSE
S532
SET CACHEBOT (Z) = CACHEMID (DST) + WEIGHT
S534
CACHEBOT (Z) < BEST?
NO
YES
S536
SET BEST = CACHEBOT (Z)
S538
RECURSIVE INVOCATION OF QUICK RESTRICT FOR DST NODE (MAY REWRITE MDD)
S544
SET CACHE (Y) = FALSE, RETURN FALSE
DST ≠ FALSE?
YES
S540
NO
YES
EMPTY FLAG = TRUE?
S542
S546
NO
SET CACHEMID (SRC NODE) = BEST; SET (Z) = 0
S548
Z < Y'S # OF CHILDREN?
NO
S550
SET CACHE (Y) TO Y, RETURN CACHE (Y)
YES
S556
Z = Z+1
CACHEBOT (Z) > BEST?
NO
S552
YES
S554
SET DST TO FALSE

FIG. 48

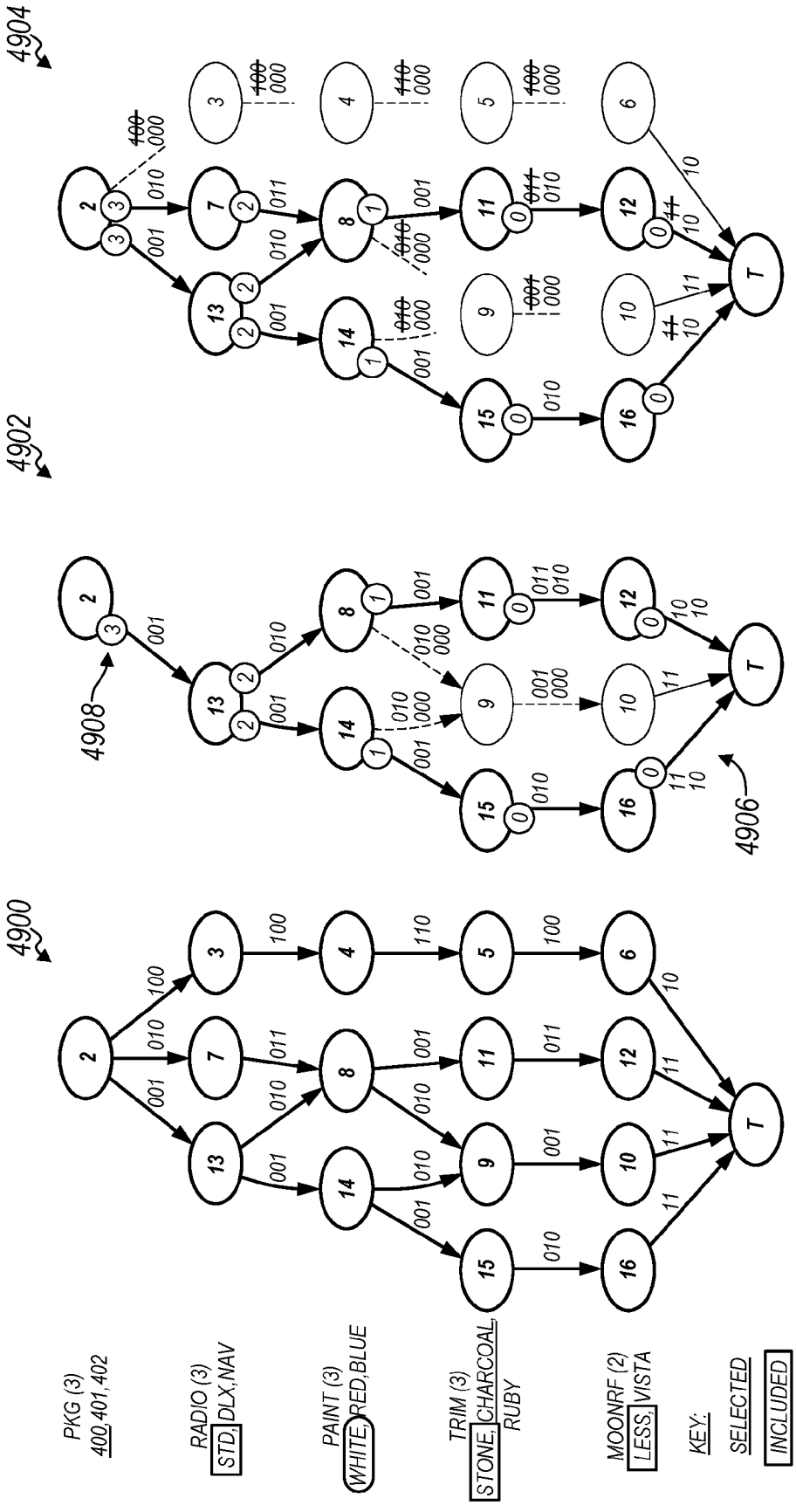

4900
4902
4904
4906
4908
PKG (3)
400, 401, 402
RADIO (3)
STD, DLX, NAV
PAINT (3)
WHITE, RED, BLUE
TRIM (3)
STONE, CHARCOAL
RUBY
MOONRF (2)
LESS, VISTA
KEY:
SELECTED
INCLUDED
DEFAULT

FIG. 49

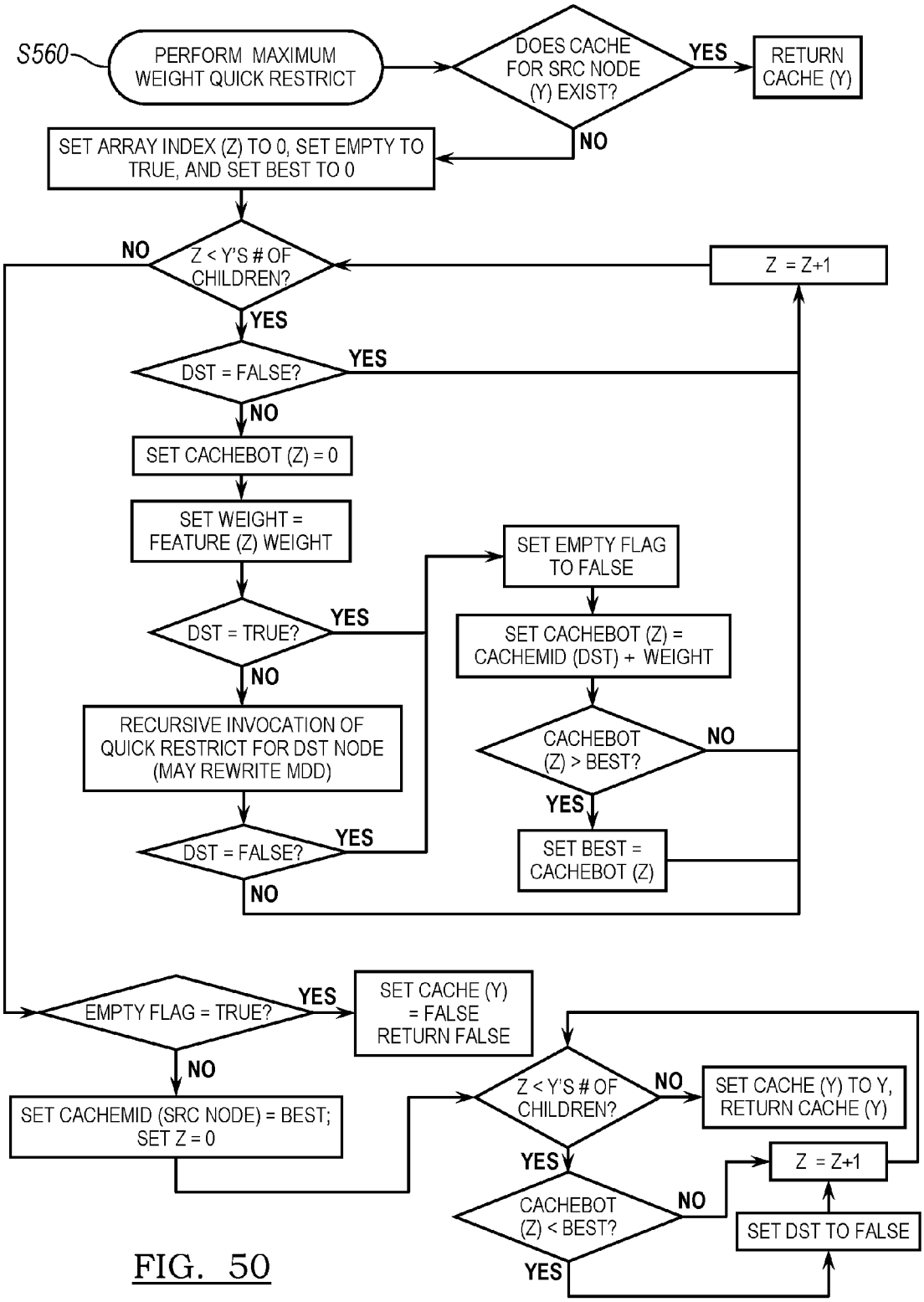

S560
PERFORM MAXIMUM WEIGHT QUICK RESTRICT
DOES CACHE FOR SRC NODE (Y) EXIST?
YES
RETURN CACHE (Y)
NO
SET ARRAY INDEX (Z) TO 0, SET EMPTY TO TRUE, AND SET BEST TO 0
NO
Z < Y'S # OF CHILDREN?
Z = Z+1
YES
DST = FALSE?
YES
NO
SET CACHEBOT (Z) = 0
SET WEIGHT = FEATURE (Z) WEIGHT
SET EMPTY FLAG TO FALSE
DST = TRUE?
YES
SET CACHEBOT (Z) = CACHEMID (DST) + WEIGHT
NO
RECURSIVE INVOCATION OF QUICK RESTRICT FOR DST NODE (MAY REWRITE MDD)
CACHEBOT (Z) > BEST?
NO
YES
DST = FALSE?
YES
SET BEST = CACHEBOT (Z)
NO
EMPTY FLAG = TRUE?
YES
SET CACHE (Y) = FALSE RETURN FALSE
NO
SET CACHEMID (SRC NODE) = BEST; SET Z = 0
Z < Y'S # OF CHILDREN?
NO
SET CACHE (Y) TO Y, RETURN CACHE (Y)
YES
Z = Z+1
CACHEBOT (Z) < BEST?
NO
SET DST TO FALSE
YES

FIG. 50

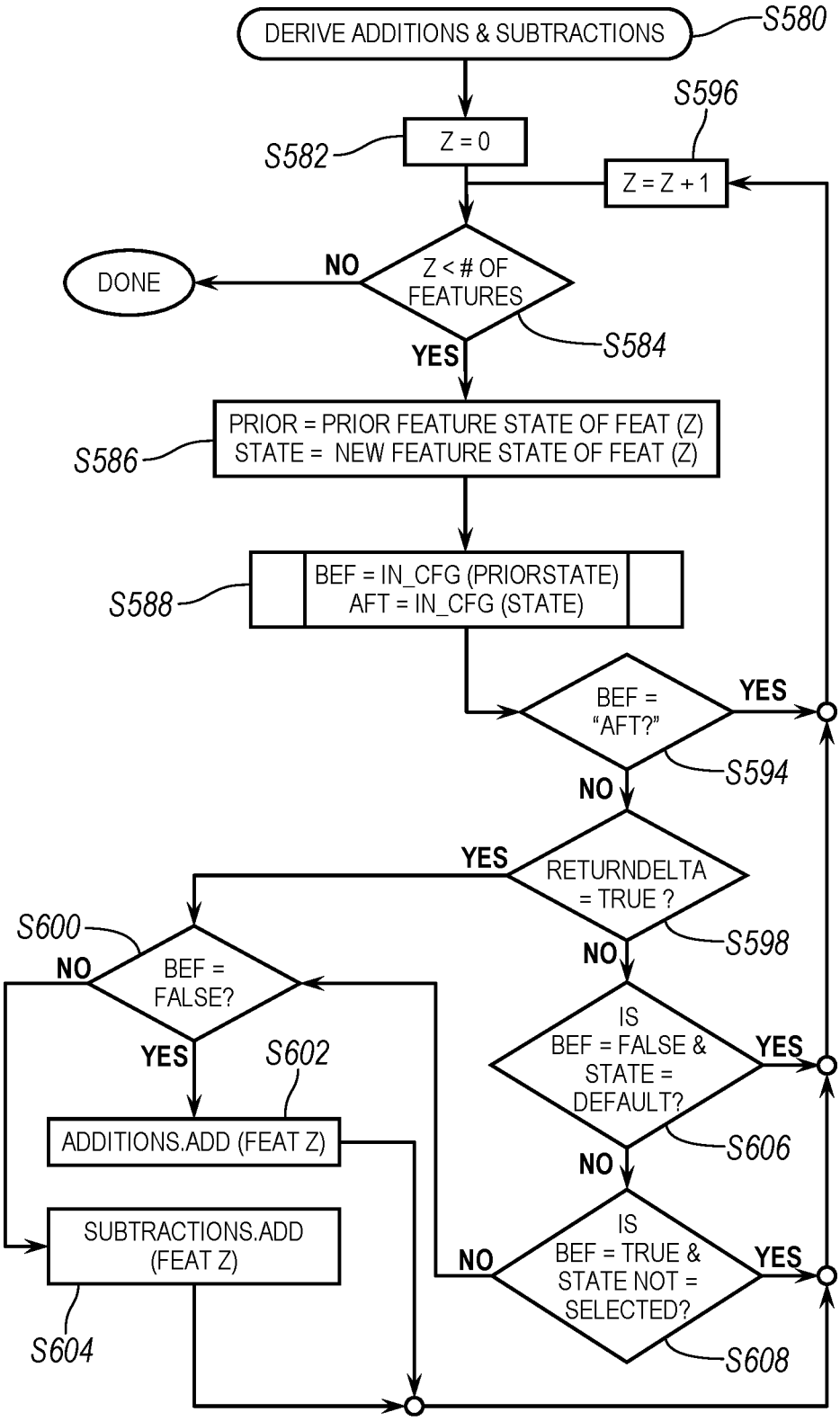

DERIVE ADDITIONS & SUBTRACTIONS
S580
S596
S582
Z = 0
Z = Z + 1
DONE
NO
Z < # OF FEATURES
S584
YES
PRIOR = PRIOR FEATURE STATE OF FEAT (Z)
STATE = NEW FEATURE STATE OF FEAT (Z)
S586
BEF = IN_CFG (PRIORSTATE)
AFT = IN_CFG (STATE)
S588
BEF = "AFT?"
YES
S594
NO
RETURNDELTA = TRUE ?
YES
S598
NO
S600
NO
BEF = FALSE?
YES
S602
IS BEF = FALSE & STATE = DEFAULT?
YES
S606
NO
ADDITIONS.ADD (FEAT Z)
SUBTRACTIONS.ADD (FEAT Z)
IS BEF = TRUE & STATE NOT = SELECTED?
NO
YES
S604
S608

FIG. 51

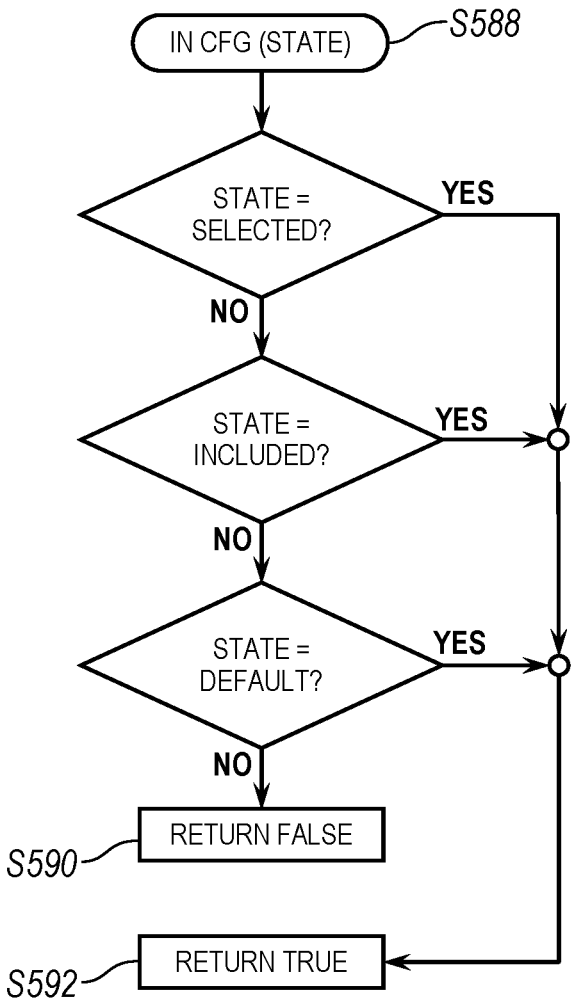

IN CFG (STATE)
S588
STATE = SELECTED?
YES
NO
STATE = INCLUDED?
YES
NO
STATE = DEFAULT?
YES
NO
RETURN FALSE
S590
RETURN TRUE
S592

FIG. 52

5302 5304 5300

PRIOR CONFIG	SELECTED (400) INCLUDED (STD, STONE, LESS) DEFAULT (WHITE)
NEW CONFIG	SELECTED (CHARCOAL) INCLUDED (BLUE) DEFAULT (401,DLX, LESS)

RETURN DELTA	ADDITIONS AND SUBSTRACTIONS
FALSE	<ADDITIONS>
	<FEATURE ID = "BLUE" GROUP = "PAINT" DESCRIPTION = "BLUE" STATE = "INCLUDED"/>
	</ADDITIONS>
	<SUBTRACTIONS>
	<FEATURE ID = "400" GROUP = "PACKAGE" DESCRIPTION = "400" STATE = "SELECTED"/>
	</SUBTRACTIONS>
TRUE	<ADDITIONS>
	<FEATURE ID = "401" GROUP = "PACKAGE" DESCRIPTION = "401" STATE = "DEFAULT"/>
	<FEATURE ID = "DLX" GROUP = "RADIO" DESCRIPTION = "DELUXE" STATE = "DEFAULT"/>
	<FEATURE ID = "BLUE" GROUP = "PAINT" DESCRIPTION = "BLUE" STATE = "INCLUDED"/>
	<FEATURE ID = "CHARCOAL" GROUP = "TRIM" DESCRIPTION = "CHARCOAL" STATE = "SELECTED"/>
	</ADDITIONS>
	<SUBTRACTIONS>
	<FEATURE ID = "400" GROUP = "PACKAGE" DESCRIPTION = "400" STATE = "SELECTED"/>
	<FEATURE ID = "STD" GROUP = "RADIO" DESCRIPTION = "STANDARD" STATE = "INCLUDED"/>
	<FEATURE ID = "WHITE" GROUP = "PAINT" DESCRIPTION = "WHITE" STATE = "DEFAULT"/>
	<FEATURE ID = "STONE" GROUP = "TRIM" DESCRIPTION = "STONE" STATE = "INCLUDED"/>
	</SUBTRACTIONS>

5306

FIG. 53

5400

CONFLICT RESOLUTION
TO SELECT TRIM CHARCOAL, YOU MUST:
• REMOVE PACKAGE 400 • ADD PAINT BLUE OK CANCEL

FIG. 54

5500

CONFLICT RESOLUTION
TO SELECT TRIM CHARCOAL, YOU MUST CHANGE:
• PACKAGE FROM 400 TO 401 • RADIO FROM STANDARD TO DELUXE • PAINT FROM WHITE TO BLUE OK CANCEL

FIG. 55

5600

010 011 001 010 10	[401, DLX, BLUE, CHARCOAL, LESS]
	[401, NAV, BLUE, CHARCOAL, LESS]
001 011 001 010 10	[402, DLX, BLUE, CHARCOAL, LESS]
	[402, NAV, BLUE, CHARCOAL, LESS]

FIG. 56

5700

CONFLICT RESOLUTION

TO ADD CHARCOAL, YOU MUST:

CHANGE PACKAGE 400 TO ONE OF:

- ☑ 401
- ☐ 402

CHANGE RADIO STD TO ONE OF:

- ☐ DELUXE
- ☑ NAVIGATION

CHANGE PAINT FROM WHITE TO BLUE.

OK CANCEL

FIG. 57

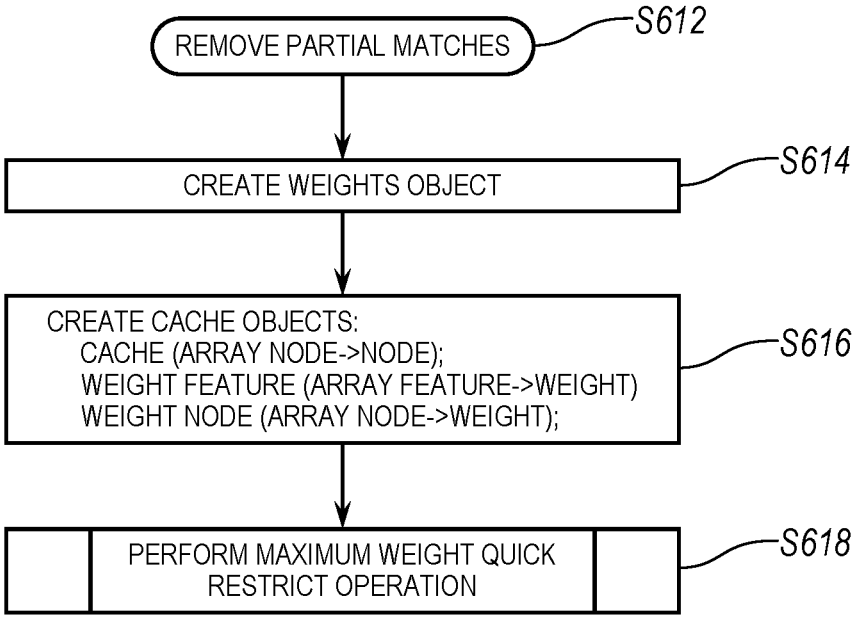

REMOVE PARTIAL MATCHES
S612
CREATE WEIGHTS OBJECT
S614
CREATE CACHE OBJECTS:
CACHE (ARRAY NODE->NODE);
WEIGHT FEATURE (ARRAY FEATURE->WEIGHT)
WEIGHT NODE (ARRAY NODE->WEIGHT);
S616
PERFORM MAXIMUM WEIGHT QUICK RESTRICT OPERATION
S618

FIG. 58

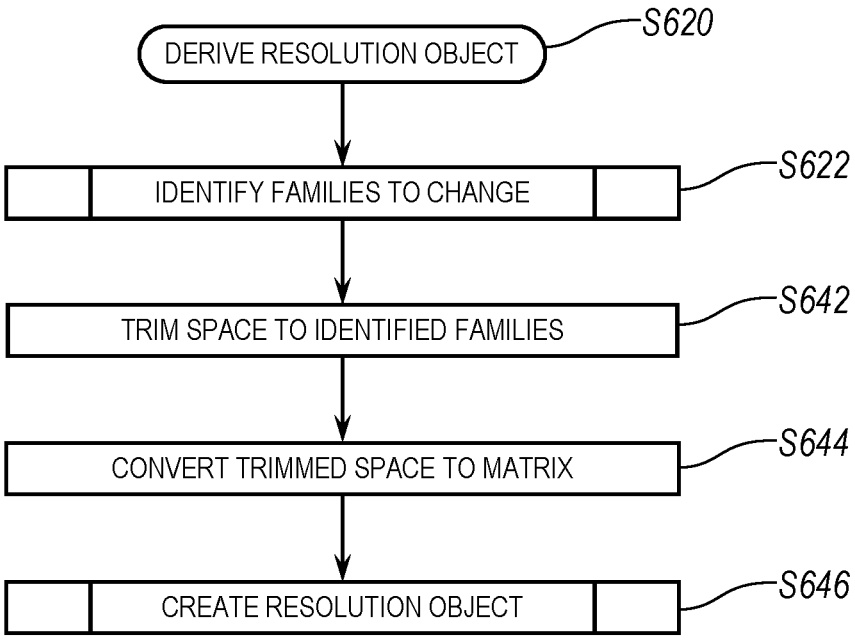

DERIVE RESOLUTION OBJECT
S620
IDENTIFY FAMILIES TO CHANGE
S622
TRIM SPACE TO IDENTIFIED FAMILIES
S642
CONVERT TRIMMED SPACE TO MATRIX
S644
CREATE RESOLUTION OBJECT
S646

FIG. 59

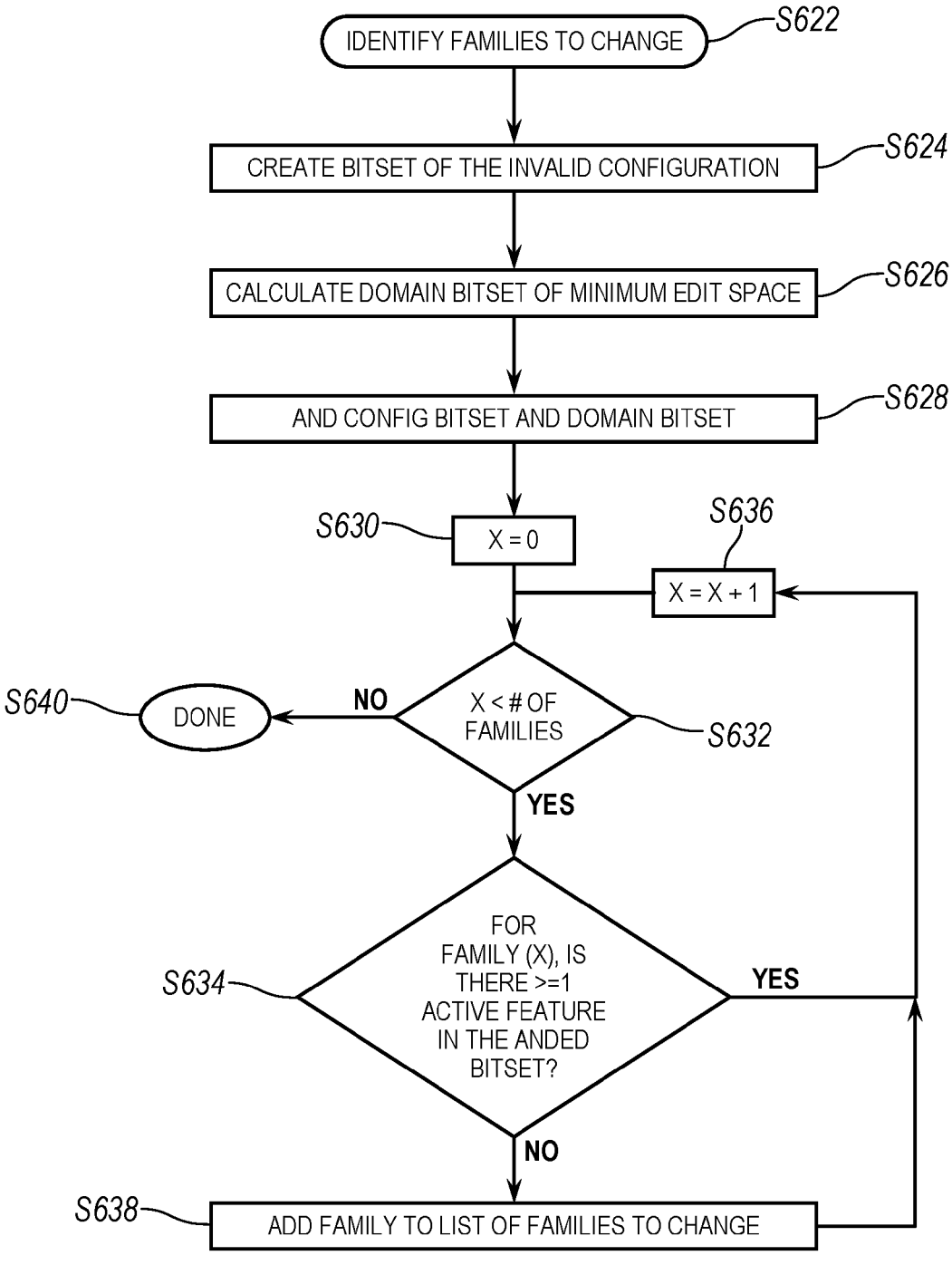

IDENTIFY FAMILIES TO CHANGE
S622
CREATE BITSET OF THE INVALID CONFIGURATION
S624
CALCULATE DOMAIN BITSET OF MINIMUM EDIT SPACE
S626
AND CONFIG BITSET AND DOMAIN BITSET
S628
S630
X = 0
S636
X = X + 1
X < # OF FAMILIES
S632
NO
S640
DONE
YES
FOR FAMILY (X), IS THERE >=1 ACTIVE FEATURE IN THE ANDED BITSET?
S634
YES
NO
S638
ADD FAMILY TO LIST OF FAMILIES TO CHANGE

FIG. 60

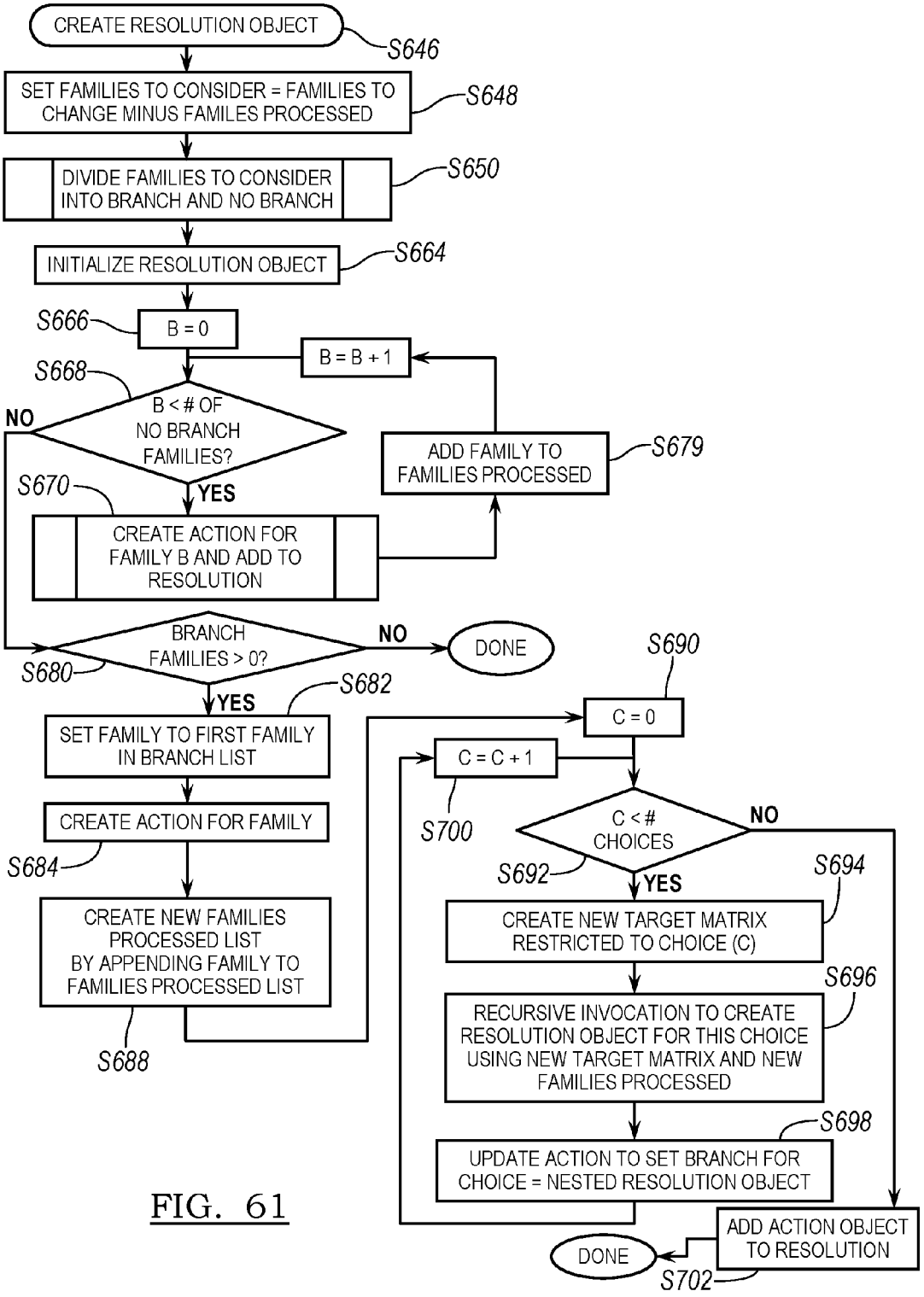

CREATE RESOLUTION OBJECT
S646
SET FAMILIES TO CONSIDER = FAMILIES TO CHANGE MINUS FAMILES PROCESSED
S648
DIVIDE FAMILIES TO CONSIDER INTO BRANCH AND NO BRANCH
S650
INITIALIZE RESOLUTION OBJECT
S664
S666
B = 0
B = B + 1
S668
B < # OF NO BRANCH FAMILIES?
NO
YES
ADD FAMILY TO FAMILIES PROCESSED
S679
S670
CREATE ACTION FOR FAMILY B AND ADD TO RESOLUTION
BRANCH FAMILIES > 0?
S680
NO
DONE
YES
S682
SET FAMILY TO FIRST FAMILY IN BRANCH LIST
CREATE ACTION FOR FAMILY
S684
CREATE NEW FAMILIES PROCESSED LIST BY APPENDING FAMILY TO FAMILIES PROCESSED LIST
S688
S690
C = 0
C = C + 1
S700
C < # CHOICES
S692
NO
YES
S694
CREATE NEW TARGET MATRIX RESTRICTED TO CHOICE (C)
S696
RECURSIVE INVOCATION TO CREATE RESOLUTION OBJECT FOR THIS CHOICE USING NEW TARGET MATRIX AND NEW FAMILIES PROCESSED
S698
UPDATE ACTION TO SET BRANCH FOR CHOICE = NESTED RESOLUTION OBJECT
ADD ACTION OBJECT TO RESOLUTION
DONE
S702

FIG. 61

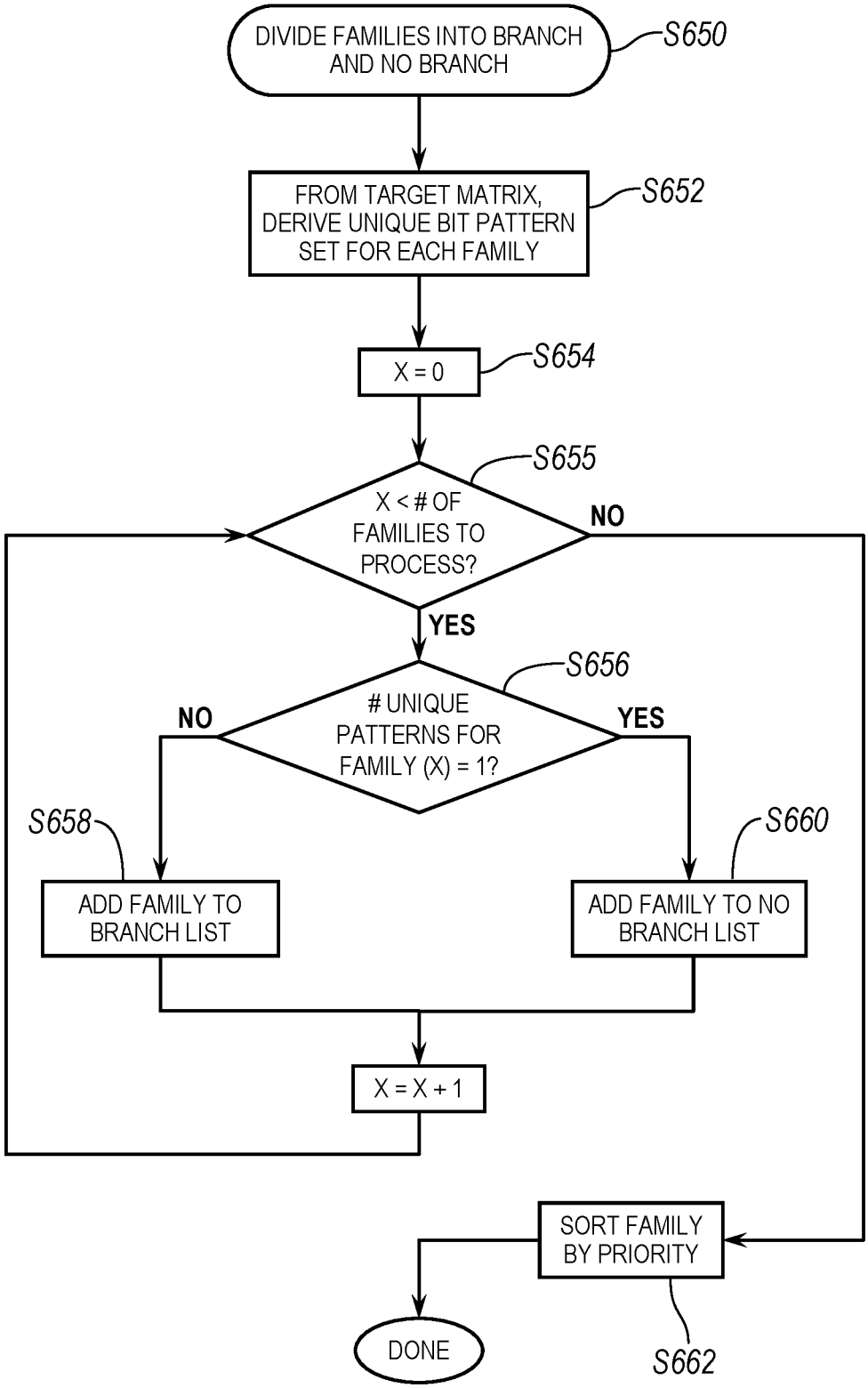

DIVIDE FAMILIES INTO BRANCH AND NO BRANCH
S650
FROM TARGET MATRIX, DERIVE UNIQUE BIT PATTERN SET FOR EACH FAMILY
S652
X = 0
S654
S655
X < # OF FAMILIES TO PROCESS?
NO
YES
S656
UNIQUE PATTERNS FOR FAMILY (X) = 1?
NO
YES
S658
ADD FAMILY TO BRANCH LIST
S660
ADD FAMILY TO NO BRANCH LIST
X = X + 1
SORT FAMILY BY PRIORITY
DONE
S662

FIG. 62

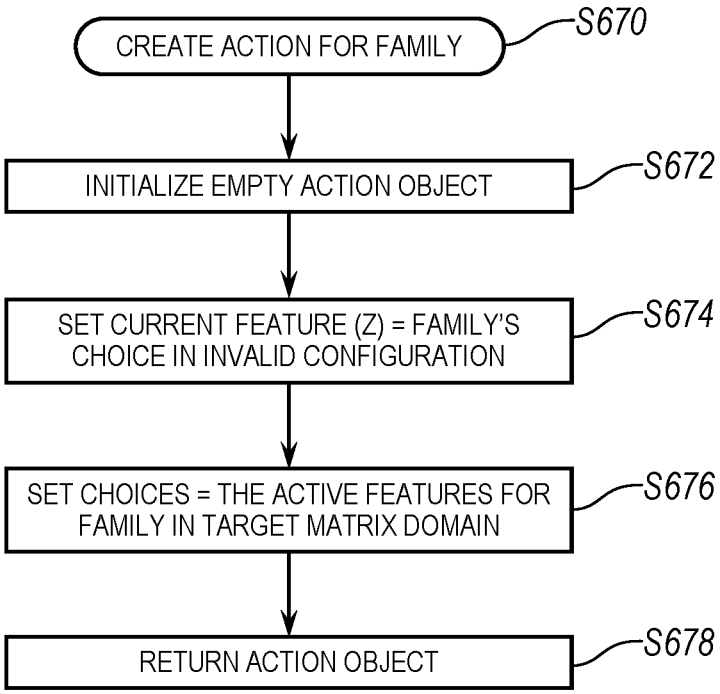

CREATE ACTION FOR FAMILY
S670
INITIALIZE EMPTY ACTION OBJECT
S672
SET CURRENT FEATURE (Z) = FAMILY'S CHOICE IN INVALID CONFIGURATION
S674
SET CHOICES = THE ACTIVE FEATURES FOR FAMILY IN TARGET MATRIX DOMAIN
S676
RETURN ACTION OBJECT
S678

FIG. 63

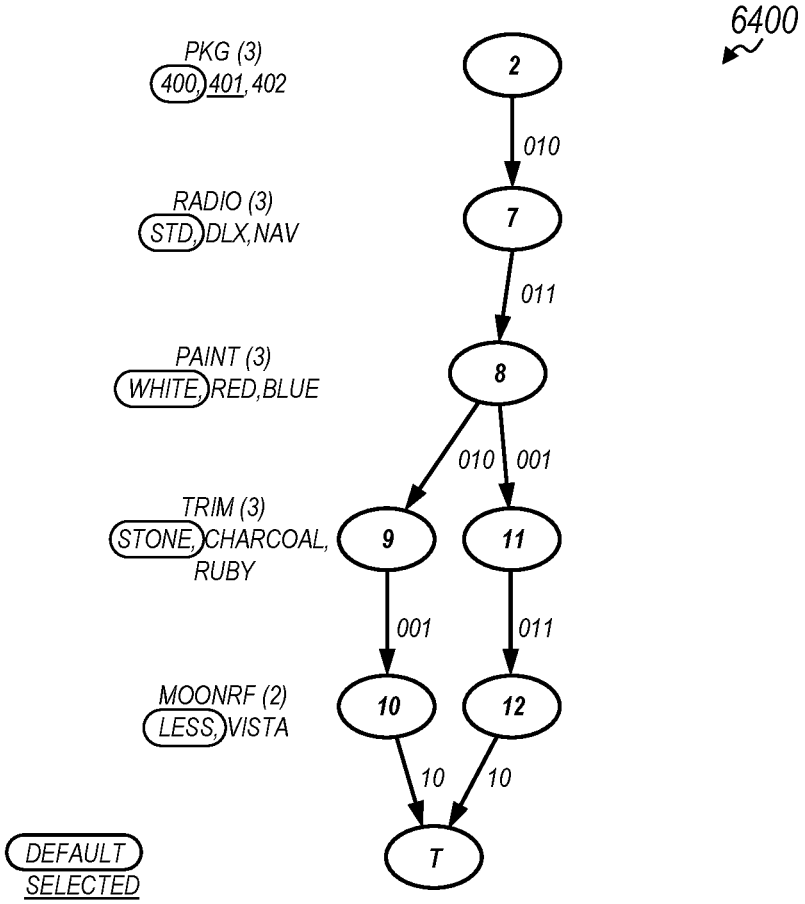

6400
PKG (3)
400, 401, 402
RADIO (3)
STD, DLX, NAV
PAINT (3)
WHITE, RED, BLUE
TRIM (3)
STONE, CHARCOAL, RUBY
MOONRF (2)
LESS, VISTA
DEFAULT
SELECTED
2
010
7
011
8
010
001
9
11
001
011
10
12
10
10
T

FIG. 64

6500

INVALID CONFIG				
010	100	100	100	10

FIG. 65

6600

MINIMUM EDIT SPACE				
PKG	RADIO	PAINT	TRIM	MOONRF
010	011	010	001	10
010	011	001	011	10

FIG. 66

6700

PKG	RADIO	PAINT	TRIM	MOONRF	
010	011	011	011	10	MINIMUM EDIT
010	100	100	100	10	INVALID CONFIG
010	000	000	000	10	AND

6702

FIG. 67

6800

TRIMMED MINIMUM EDIT SPACE		
RADIO	PAINT	TRIM
011	010	001
011	001	011

FIG. 68

6900

TRIMMED MINIMUM EDIT SPACE		
RADIO	PAINT	TRIM
011	010	001

FIG. 69

7000

TRIMMED MINIMUM EDIT SPACE		
RADIO	PAINT	TRIM
011	001	011

FIG. 70

7100

```
RESOLUTION {
 ACTIONS [
  ACTION { RADIO, [STD], [DLX, NAV] } }
  ACTION { PAINT, [WHITE], [RED, BLUE],
   BRANCHES {
    RED ->  RESOLUTION {
                ACTIONS [ ACTION { TRIM, [STONE], [RUBY] } ]
              }
    BLUE ->  RESOLUTION {
                ACTIONS [ ACTION { TRIM, [STONE], [CHARCOAL, RUBY] } ]
              }....
```

FIG. 71

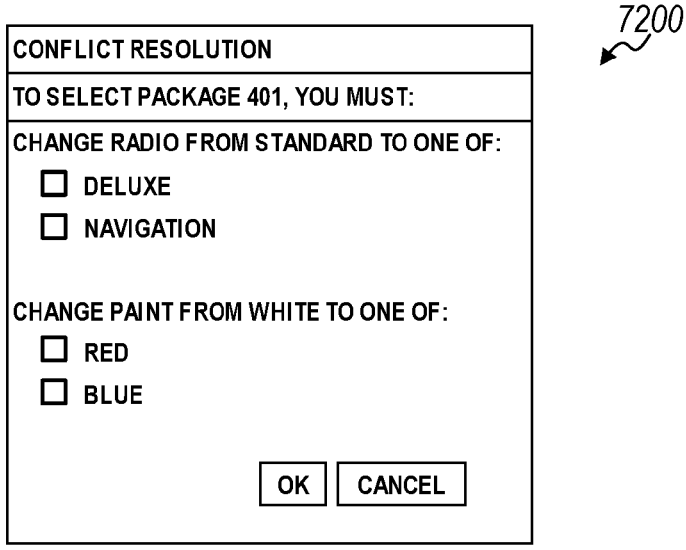


FIG. 72

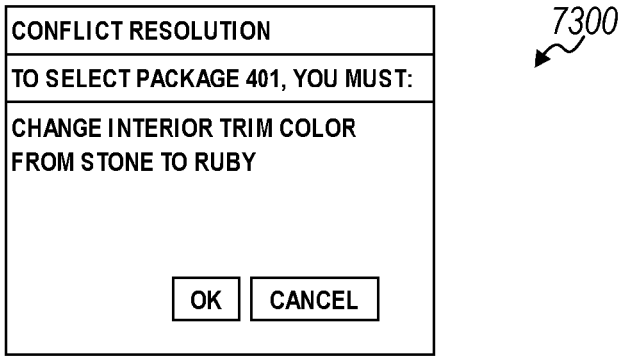


FIG. 73

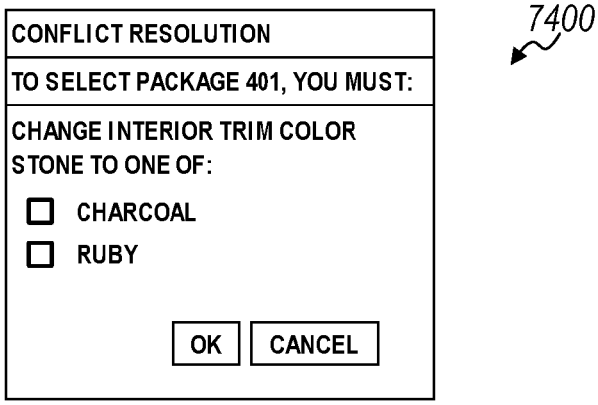


FIG. 74

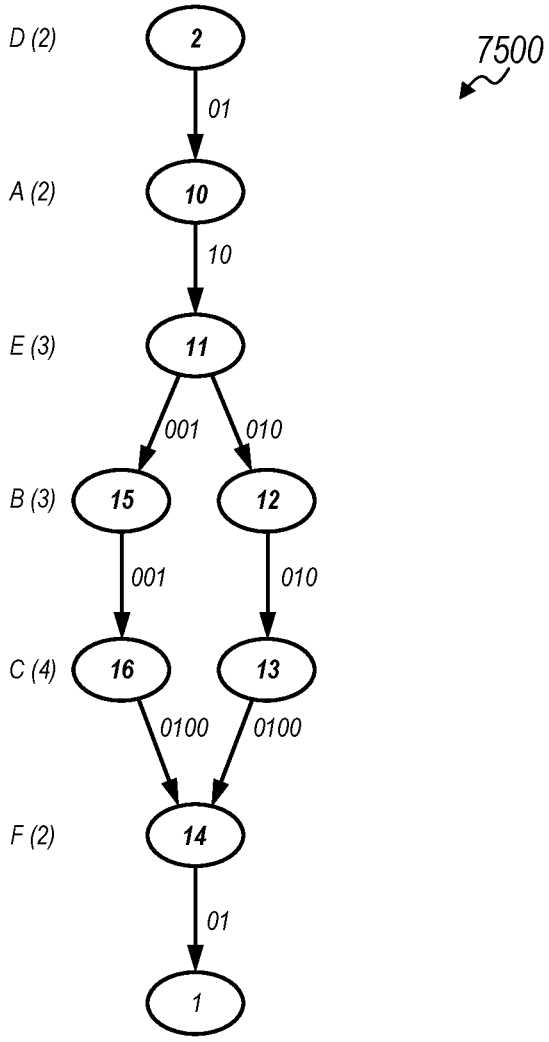


FIG. 75

7600

PATH	TARGET CONFIG	WEIGHT
2-10-11-12-13-14-T	D2 A1 E2 B2 C2 F2	010100
2-10-11-15-16-14-T	D2 A1 E3 B3 C2 F2	000110

FIG. 76

7700

E	F	Y	T	R	P	S	A
01	10	001001	10	1000	101	000011	01
10	01	000110	11	0100	100	001000	01
10	01	001001	11	1000	101	000011	01
10	01	001001	11	0001	111	000100	10
10	01	110000	11	0010	111	110000	01

FIG. 77

7800

E	F	Y	T	R	P	S	A
10	01	000110	01	0100	100	001000	01
10	01	001001	01	1000	101	000011	01
10	01	001001	01	0001	111	000100	10
10	01	110000	01	0010	111	110000	01

FIG. 78

7900

E	F	Y	T	R	P	S	A
10	01	001000	01	1000	100	000010	01

FIG. 79

8000

INVALID	E2	F1	Y3	T2	R1	P1	S5	A2
TARGET 1	E1	F2	Y1	T2	R3	P1	S1	A2

FIG. 80

8100

INVALID	E2	F1	Y3	T2	R1	P1	S5	A2
TARGET 2	E1	F2	Y3	T2	R1	P1	S5	A2

FIG. 81

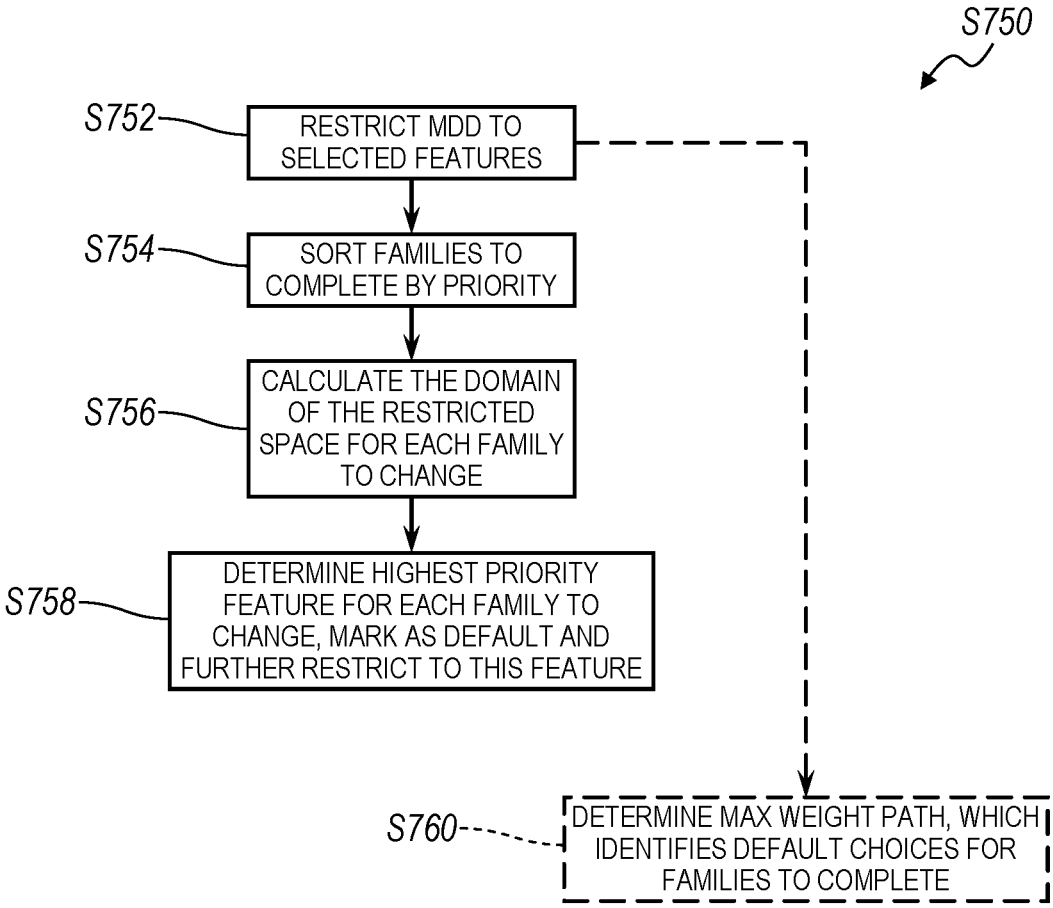

S750
S752
RESTRICT MDD TO SELECTED FEATURES
S754
SORT FAMILIES TO COMPLETE BY PRIORITY
S756
CALCULATE THE DOMAIN OF THE RESTRICTED SPACE FOR EACH FAMILY TO CHANGE
S758
DETERMINE HIGHEST PRIORITY FEATURE FOR EACH FAMILY TO CHANGE, MARK AS DEFAULT AND FURTHER RESTRICT TO THIS FEATURE
S760
DETERMINE MAX WEIGHT PATH, WHICH IDENTIFIES DEFAULT CHOICES FOR FAMILIES TO COMPLETE

FIG. 82

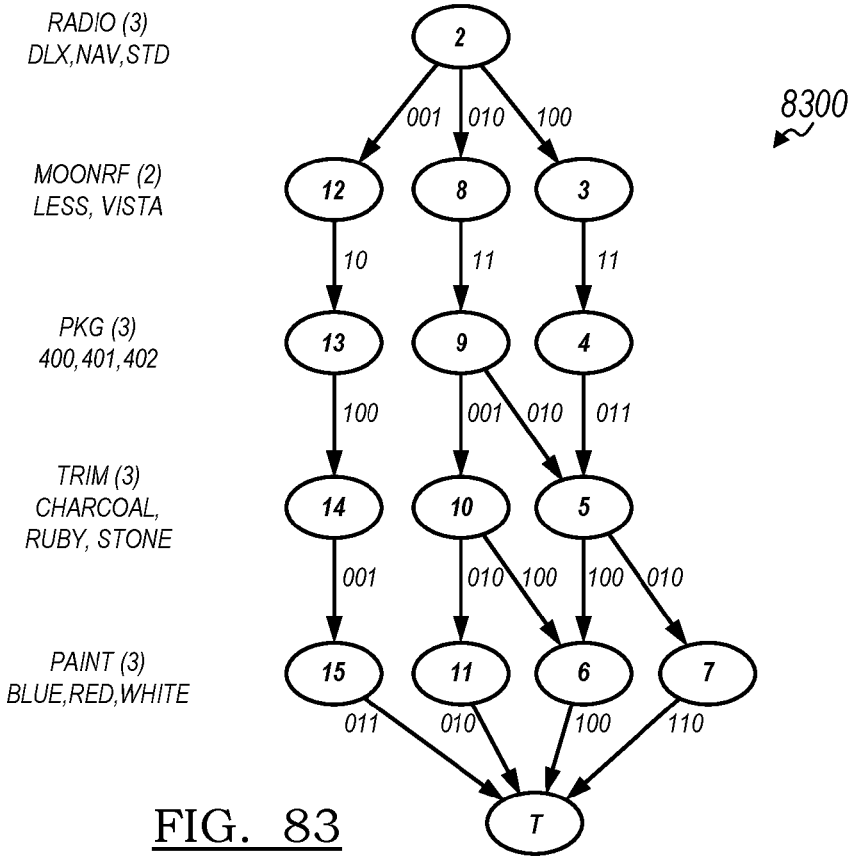


FIG. 83

8400

ALT SEQUENCE STRUCTURE		
FEATURE	INDEX IN FAMILY	INDEX IN STRUCTURE
PKG.400	0	0
PKG.401	1	1
PKG.402	2	2
RADIO.STD	0	3
RADIO.DLX	1	4
RADIO.NAV	2	5
PAINT.WHITE	0	6
PAINT.RED	1	7
PAINT.BLUE	2	8
TRIM.STONE	0	9
TRIM.CHARCOAL	1	10
TRIM.RUBY	2	11
MOONRF.LESS	0	12
MONNRF.VISTA	1	13

FIG. 84

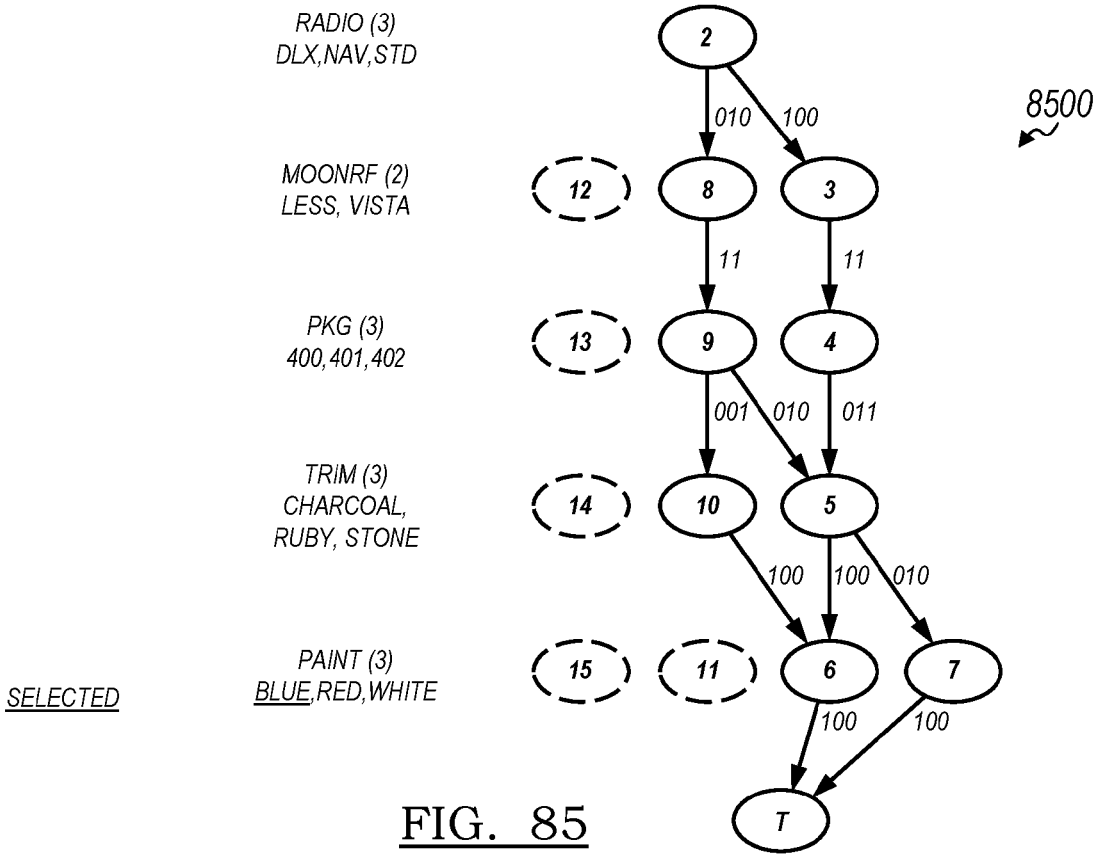


FIG. 85

8600

PATH	WEIGHT
6-T	000 000 001 000 00
10-6-T	000 000 001 010 00
9-10-6-T	001 000 001 010 00

FIG. 86

8700

PATH	WEIGHT
5-RUBY-7-T	000 000 001 001 00
5-CHARCOAL-6-T	000 000 001 010 00

8702

FIG. 87

8800

PATH	WEIGHT
9-402-10-6-T	001 000 001 010 00
9-401-5-6-T	010 000 001 010 00

8802

FIG. 88

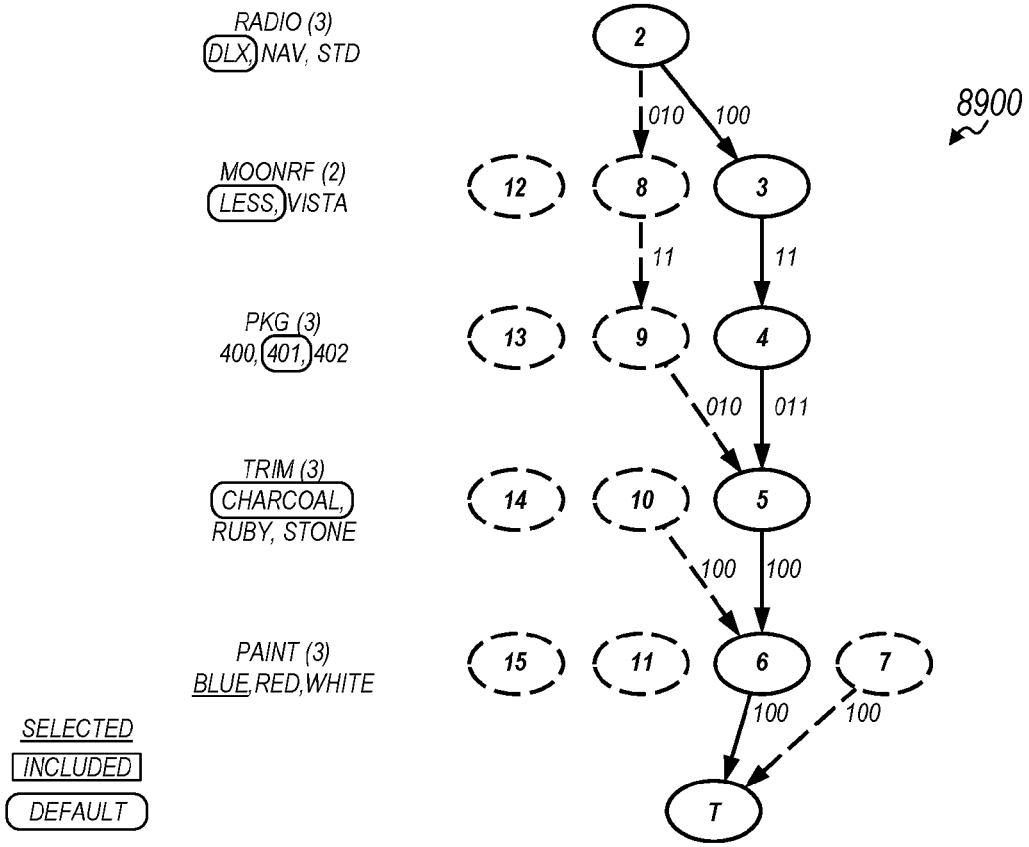


FIG. 89

9000

PATH	WEIGHT
2-NAV-8-9-5-6-T	010 001 001 010 10
2-DLX-3-4-5-6-T	010 010 001 010 10

9002

FIG. 90

9100

BUILDABLE SPACE																	
PKG.400	PKG.401	PKG.402	RADIO.STD	RADIO.DLX	RADIO.NAV	PAINT.WHITE	PAINT.RED	PAINT.BLUE	TRIM.STONE	TRIM.CHARCOAL	TRIM.RUBY	MOONRF.LESS	MOONRF.VISTA	DICE.NODICE	DICE.FUZZYDICE	TEMP.LESSTEMP	TEMP.HEAT	TEMP.HEATCOOL
1	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	1	0	0
0	1	0	0	1	1	0	1	0	0	0	1	1	1	0	1	0	1	0
0	1	0	0	1	1	0	0	1	0	1	1	1	1	0	1	0	1	0
0	0	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0	0	1
0	0	1	0	1	0	0	0	1	0	1	1	1	1	0	1	0	0	1
0	0	1	0	0	1	0	1	0	0	0	1	1	1	0	1	0	0	1
0	0	1	0	0	1	0	0	1	0	1	0	1	1	0	1	0	0	1
1	0	0	1	0	0	1	0	0	1	0	0	1	0	1	0	1	0	0

FIG. 91

9200

	STANDARD FEATURE CONDITIONS																	
	PKG.400	PKG.401	PKG.402	RADIO.STD	RADIO.DLX	RADIO.NAV	PAINT.WHITE	PAINT.RED	PAINT.BLUE	TRIM.STONE	TRIM.CHARCOAL	TRIM.RUBY	MOONRF.LESS	MOONRF.VISTA	DICE.NODICE	DICE.FUZZYDICE	TEMP.LESSTEMP	TEMP.HEAT	TEMP.HEATCOOL
9202 PKG	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
9204 RADIO	1	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
MOONRF	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
9206 DICE	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	1	1	1
	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1
9208 TEMP	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
9210	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1

FIG. 92

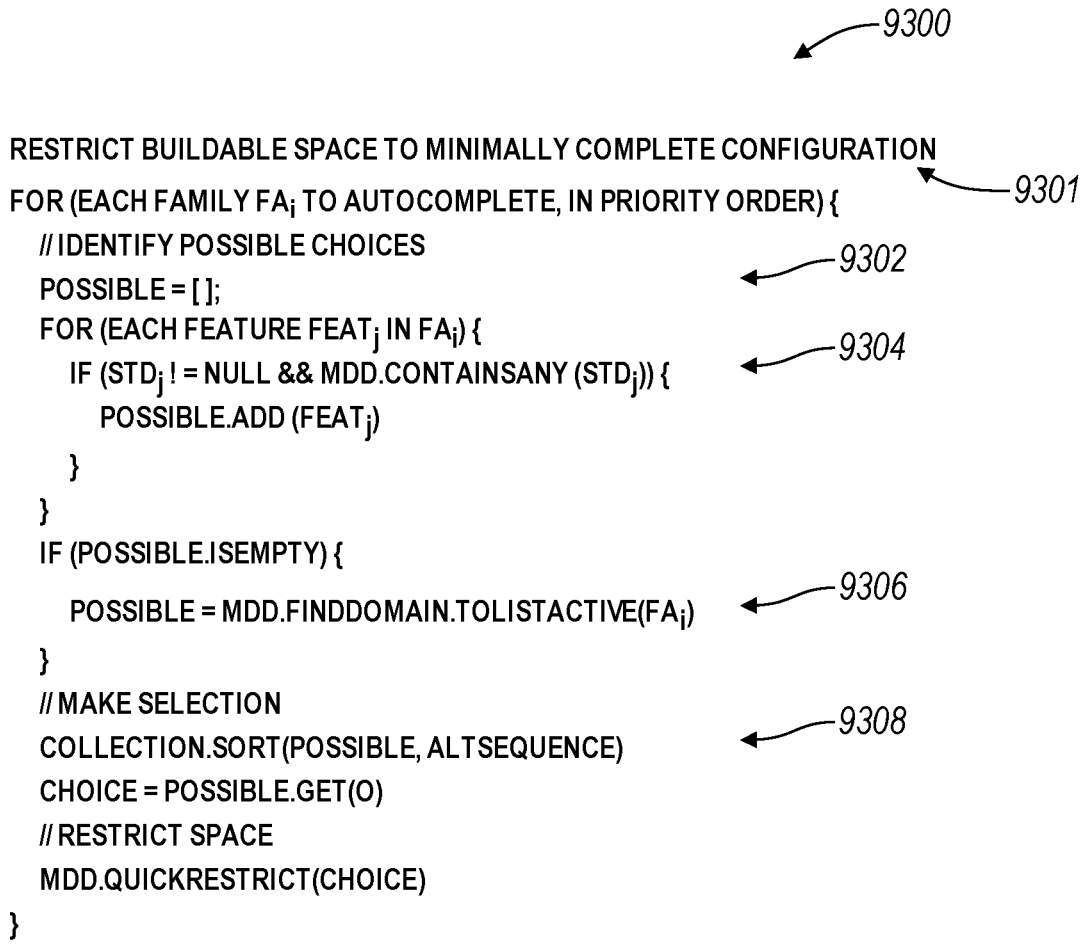

9300
RESTRICT BUILDABLE SPACE TO MINIMALLY COMPLETE CONFIGURATION
FOR (EACH FAMILY FA$_i$ TO AUTOCOMPLETE, IN PRIORITY ORDER) {
9301
// IDENTIFY POSSIBLE CHOICES
POSSIBLE = [ ];
9302
FOR (EACH FEATURE FEAT$_j$ IN FA$_i$) {
IF (STD$_j$! = NULL && MDD.CONTAINSANY (STD$_j$)) {
9304
POSSIBLE.ADD (FEAT$_j$)
}
}
IF (POSSIBLE.ISEMPTY) {
POSSIBLE = MDD.FINDDOMAIN.TOLISTACTIVE(FA$_i$)
9306
}
// MAKE SELECTION
COLLECTION.SORT(POSSIBLE, ALTSEQUENCE)
9308
CHOICE = POSSIBLE.GET(0)
// RESTRICT SPACE
MDD.QUICKRESTRICT(CHOICE)
}

FIG. 93

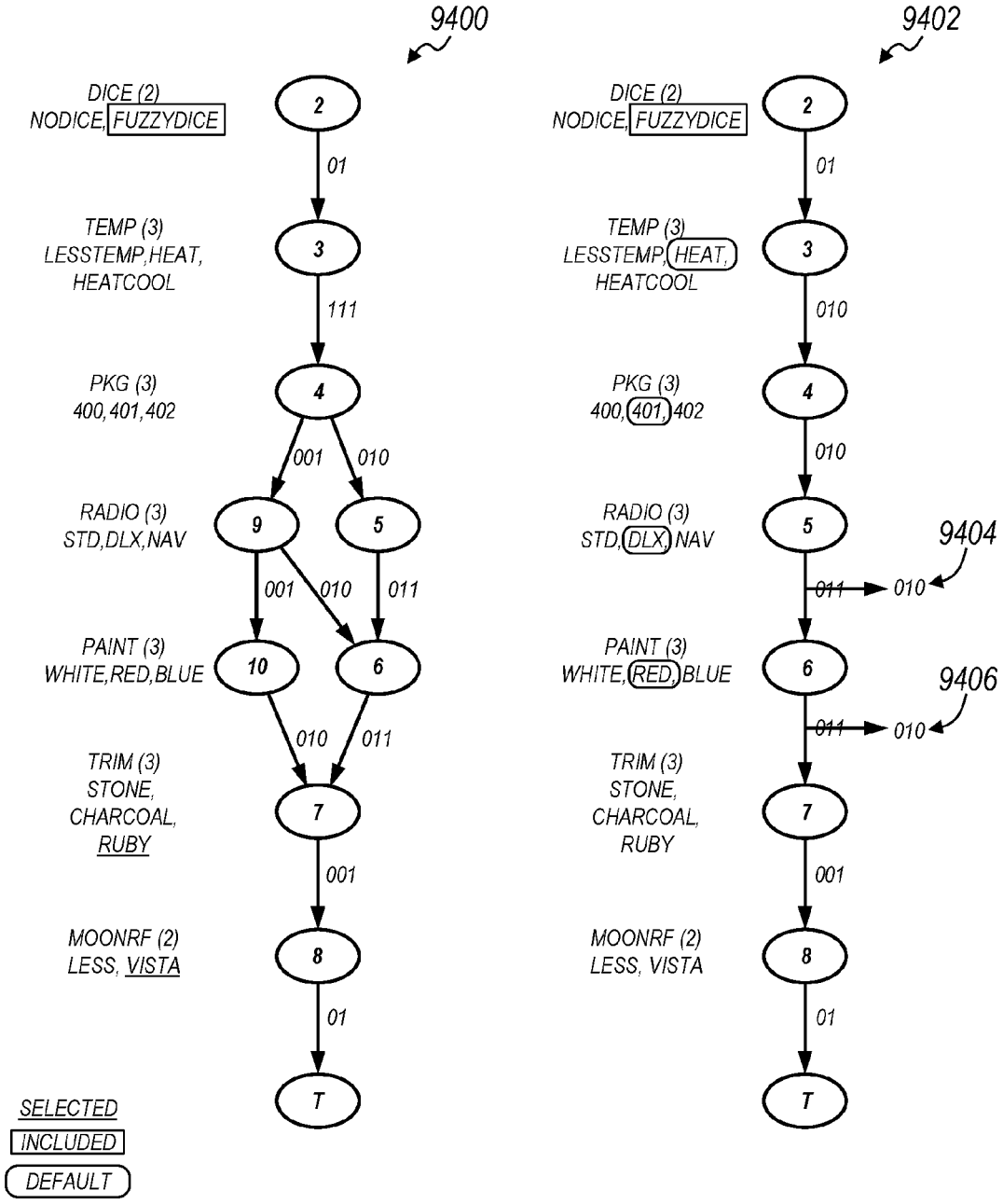

9400
DICE (2)
NODICE, FUZZYDICE
2
01
TEMP (3)
LESSTEMP, HEAT,
HEATCOOL
3
111
PKG (3)
400, 401, 402
4
001
010
RADIO (3)
STD, DLX, NAV
9
5
001
010
011
PAINT (3)
WHITE, RED, BLUE
10
6
010
011
TRIM (3)
STONE,
CHARCOAL,
RUBY
7
001
MOONRF (2)
LESS, VISTA
8
01
T
9402
DICE (2)
NODICE, FUZZYDICE
2
01
TEMP (3)
LESSTEMP, HEAT,
HEATCOOL
3
010
PKG (3)
400, 401, 402
4
010
RADIO (3)
STD, DLX, NAV
5
9404
011
010
PAINT (3)
WHITE, RED, BLUE
6
9406
011
010
TRIM (3)
STONE,
CHARCOAL,
RUBY
7
001
MOONRF (2)
LESS, VISTA
8
01
T
SELECTED
INCLUDED
DEFAULT

FIG. 94

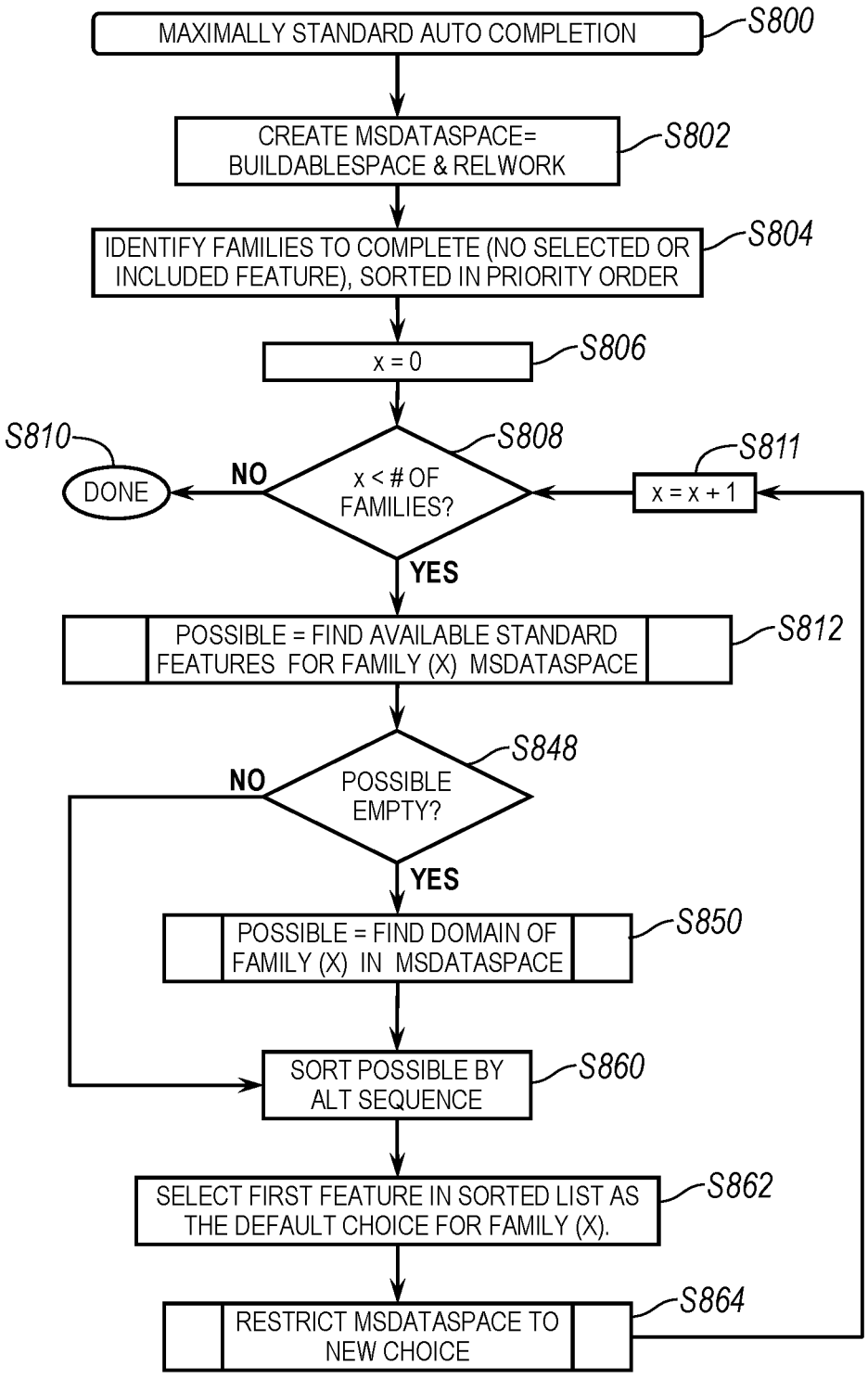

MAXIMALLY STANDARD AUTO COMPLETION
S800
CREATE MSDATASPACE= BUILDABLESPACE & RELWORK
S802
IDENTIFY FAMILIES TO COMPLETE (NO SELECTED OR INCLUDED FEATURE), SORTED IN PRIORITY ORDER
S804
x = 0
S806
S810
DONE
NO
x < # OF FAMILIES?
S808
S811
x = x + 1
YES
POSSIBLE = FIND AVAILABLE STANDARD FEATURES FOR FAMILY (X) MSDATASPACE
S812
S848
NO
POSSIBLE EMPTY?
YES
POSSIBLE = FIND DOMAIN OF FAMILY (X) IN MSDATASPACE
S850
SORT POSSIBLE BY ALT SEQUENCE
S860
SELECT FIRST FEATURE IN SORTED LIST AS THE DEFAULT CHOICE FOR FAMILY (X).
S862
RESTRICT MSDATASPACE TO NEW CHOICE
S864

FIG. 95

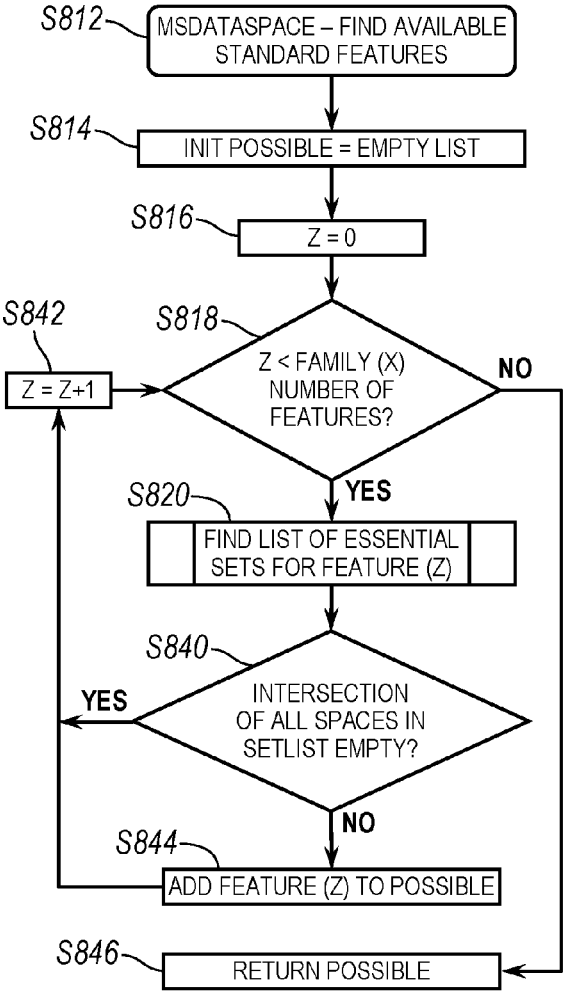

S812
MSDATASPACE – FIND AVAILABLE STANDARD FEATURES
S814
INIT POSSIBLE = EMPTY LIST
S816
Z = 0
S842
S818
Z = Z+1
Z < FAMILY (X) NUMBER OF FEATURES?
NO
YES
S820
FIND LIST OF ESSENTIAL SETS FOR FEATURE (Z)
S840
YES
INTERSECTION OF ALL SPACES IN SETLIST EMPTY?
NO
S844
ADD FEATURE (Z) TO POSSIBLE
S846
RETURN POSSIBLE

FIG. 96

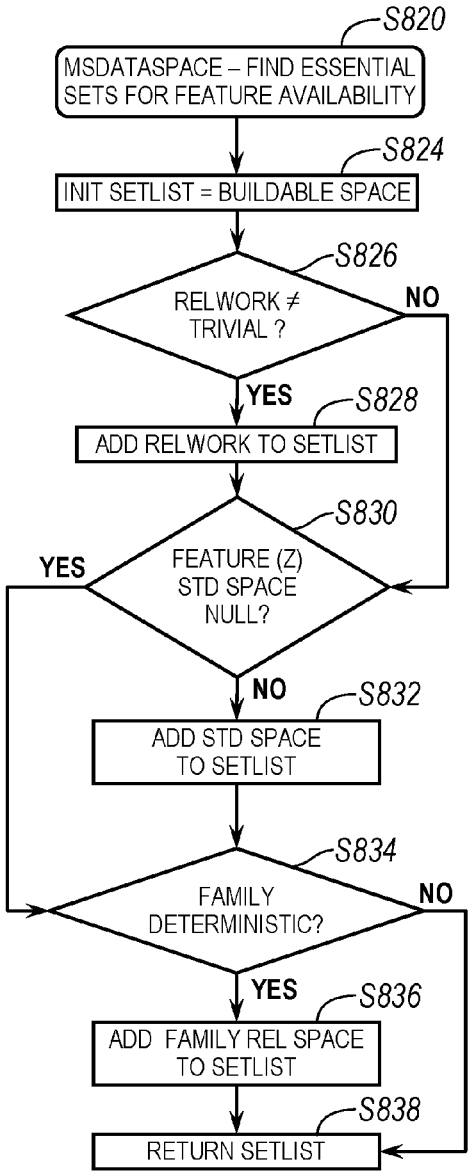

S820
MSDATASPACE – FIND ESSENTIAL SETS FOR FEATURE AVAILABILITY
S824
INIT SETLIST = BUILDABLE SPACE
S826
RELWORK ≠ TRIVIAL ?
NO
YES
S828
ADD RELWORK TO SETLIST
S830
YES
FEATURE (Z) STD SPACE NULL?
NO
S832
ADD STD SPACE TO SETLIST
S834
FAMILY DETERMINISTIC?
NO
YES
S836
ADD FAMILY REL SPACE TO SETLIST
S838
RETURN SETLIST

FIG. 97

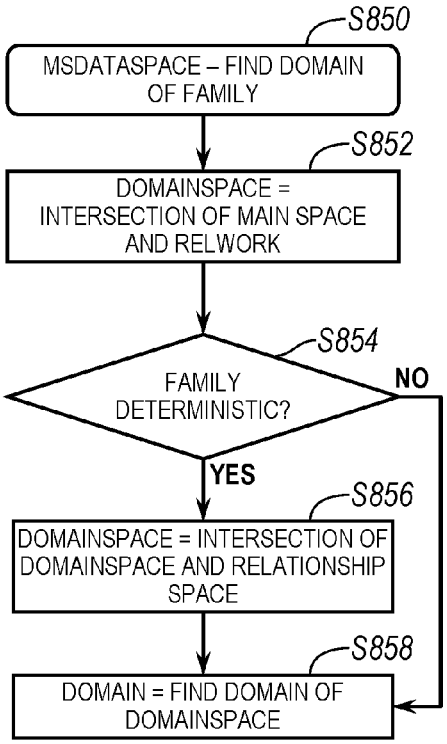

S850
MSDATASPACE – FIND DOMAIN OF FAMILY
S852
DOMAINSPACE = INTERSECTION OF MAIN SPACE AND RELWORK
S854
FAMILY DETERMINISTIC?
NO
YES
S856
DOMAINSPACE = INTERSECTION OF DOMAINSPACE AND RELATIONSHIP SPACE
S858
DOMAIN = FIND DOMAIN OF DOMAINSPACE

FIG. 98

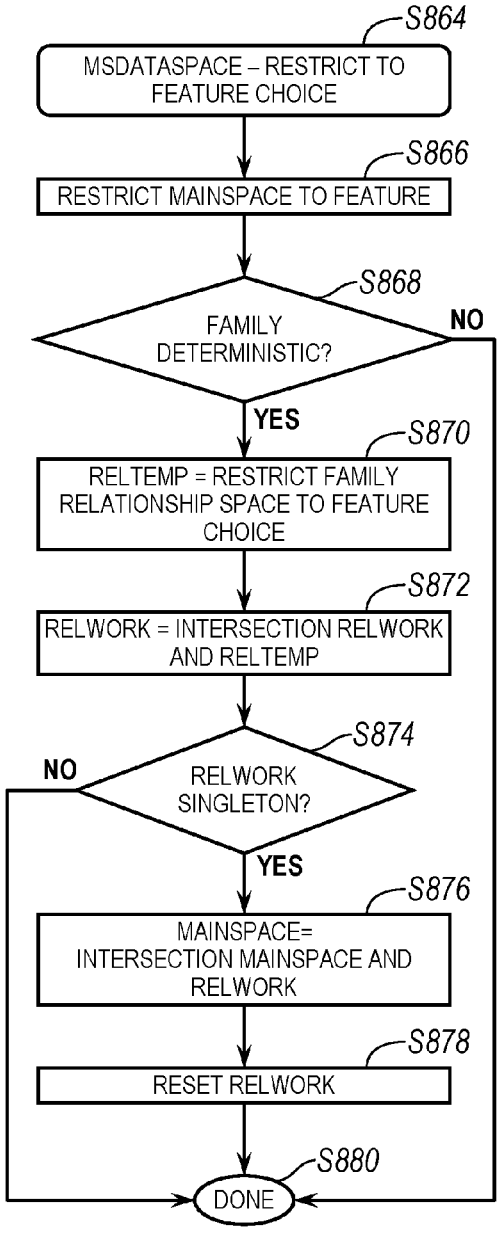
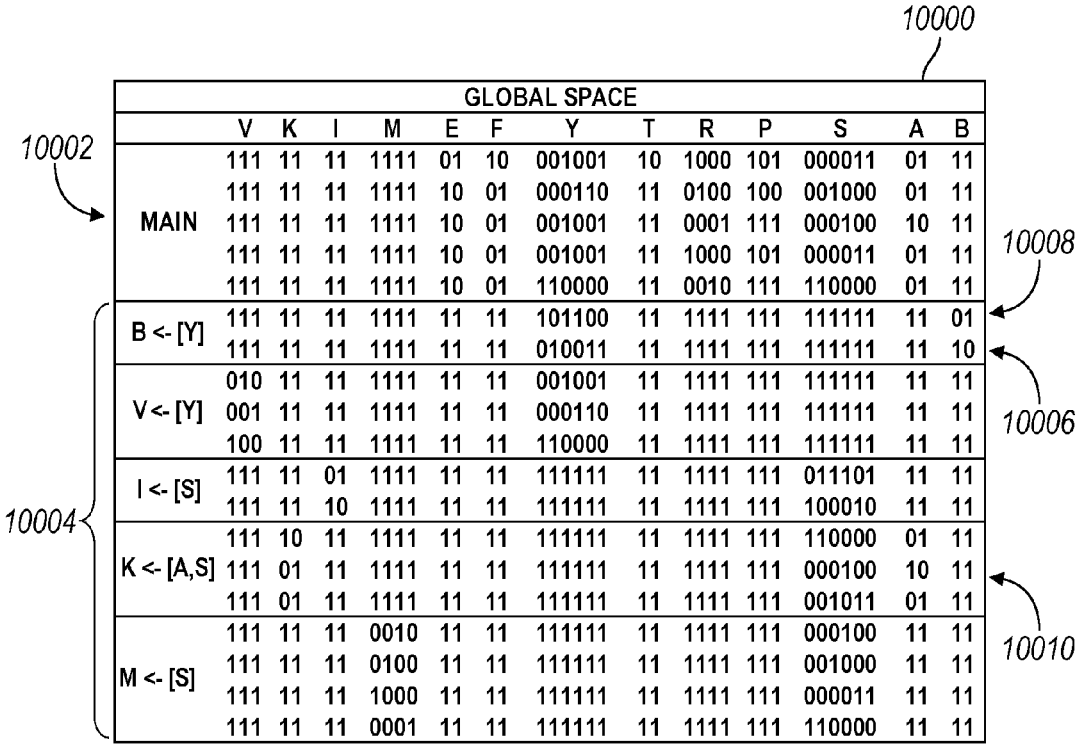
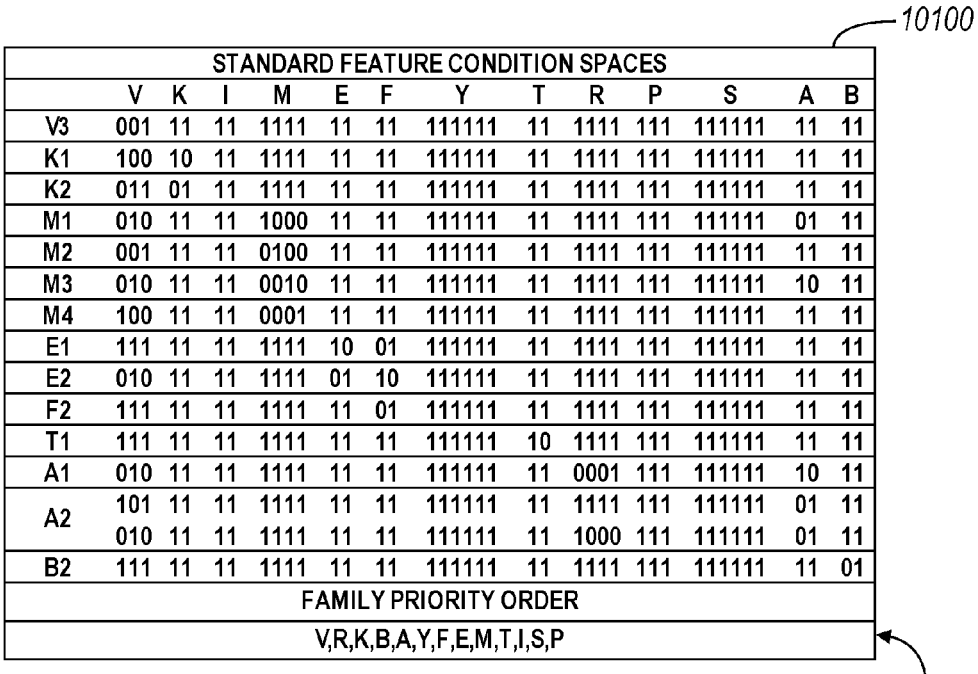

S864
MSDATASPACE – RESTRICT TO FEATURE CHOICE
S866
RESTRICT MAINSPACE TO FEATURE
S868
FAMILY DETERMINISTIC?
NO
YES
S870
RELTEMP = RESTRICT FAMILY RELATIONSHIP SPACE TO FEATURE CHOICE
S872
RELWORK = INTERSECTION RELWORK AND RELTEMP
S874
NO
RELWORK SINGLETON?
YES
S876
MAINSPACE= INTERSECTION MAINSPACE AND RELWORK
S878
RESET RELWORK
S880
DONE

FIG. 99



GLOBAL SPACE													
	V	K	I	M	E	F	Y	T	R	P	S	A	B
MAIN	111	11	11	1111	01	10	001001	10	1000	101	000011	01	11
	111	11	11	1111	10	01	000110	11	0100	100	001000	01	11
	111	11	11	1111	10	01	001001	11	0001	111	000100	10	11
	111	11	11	1111	10	01	001001	11	1000	101	000011	01	11
	111	11	11	1111	10	01	110000	11	0010	111	110000	01	11
B <- [Y]	111	11	11	1111	11	11	101100	11	1111	111	111111	11	01
	111	11	11	1111	11	11	010011	11	1111	111	111111	11	10
V <- [Y]	010	11	11	1111	11	11	001001	11	1111	111	111111	11	11
	001	11	11	1111	11	11	000110	11	1111	111	111111	11	11
	100	11	11	1111	11	11	110000	11	1111	111	111111	11	11
I <- [S]	111	11	01	1111	11	11	111111	11	1111	111	011101	11	11
	111	11	10	1111	11	11	111111	11	1111	111	100010	11	11
K <- [A,S]	111	10	11	1111	11	11	111111	11	1111	111	110000	01	11
	111	01	11	1111	11	11	111111	11	1111	111	000100	10	11
	111	01	11	1111	11	11	111111	11	1111	111	001011	01	11
M <- [S]	111	11	11	0010	11	11	111111	11	1111	111	000100	11	11
	111	11	11	0100	11	11	111111	11	1111	111	001000	11	11
	111	11	11	1000	11	11	111111	11	1111	111	000011	11	11
	111	11	11	0001	11	11	111111	11	1111	111	110000	11	11

FIG. 100



STANDARD FEATURE CONDITION SPACES													
	V	K	I	M	E	F	Y	T	R	P	S	A	B
V3	001	11	11	1111	11	11	111111	11	1111	111	111111	11	11
K1	100	10	11	1111	11	11	111111	11	1111	111	111111	11	11
K2	011	01	11	1111	11	11	111111	11	1111	111	111111	11	11
M1	010	11	11	1000	11	11	111111	11	1111	111	111111	01	11
M2	001	11	11	0100	11	11	111111	11	1111	111	111111	11	11
M3	010	11	11	0010	11	11	111111	11	1111	111	111111	10	11
M4	100	11	11	0001	11	11	111111	11	1111	111	111111	11	11
E1	111	11	11	1111	10	01	111111	11	1111	111	111111	11	11
E2	010	11	11	1111	01	10	111111	11	1111	111	111111	11	11
F2	111	11	11	1111	11	01	111111	11	1111	111	111111	11	11
T1	111	11	11	1111	11	11	111111	10	1111	111	111111	11	11
A1	010	11	11	1111	11	11	111111	11	0001	111	111111	10	11
A2	101	11	11	1111	11	11	111111	11	1111	111	111111	01	11
	010	11	11	1111	11	11	111111	11	1000	111	111111	01	11
B2	111	11	11	1111	11	11	111111	11	1111	111	111111	11	01
FAMILY PRIORITY ORDER													
V,R,K,B,A,Y,F,E,M,T,I,S,P													

FIG. 101

10200

CONFIGURATION	RELEVANT MDDs	CALL
{E1, T1}	MASK,MAIN	MAIN.CONTAINSANY (MASK)
{B1, E1, I1,T1}	MASK,MAIN,RELB,RELI	MAIN.CONTAINSANYPARRALLEL (MASK,RELB,RELI)
{K1, V1}	MASK,MAIN,RELK,RELV	MAIN.CONTAINSANYPARRALLEL (MASK,RELK,RELV)

FIG. 102

10300

REL B												
V	K	I	M	E	F	Y	T	R	P	S	A	B
111	11	11	1111	11	11	010011	11	1111	111	111111	11	10

FIG. 103

10400

MAIN SPACE												
V	K	I	M	E	F	Y	T	R	P	S	A	B
111	11	11	1111	01	10	000001	10	1000	101	000011	01	10
111	11	11	1111	10	01	000010	11	0100	100	001000	01	10
111	11	11	1111	10	01	000001	11	0001	111	000100	10	10
111	11	11	1111	10	01	000001	11	1000	101	000011	01	10
111	11	11	1111	10	01	010000	11	0010	111	110000	01	10

10402

FIG. 104

REL K												
V	K	I	M	E	F	Y	T	R	P	S	A	B
111	01	11	1111	11	11	111111	11	1111	111	001011	01	11
111	01	11	1111	11	11	111111	11	1111	111	000100	10	11

10500

FIG. 105

MAIN SPACE												
V	K	I	M	E	F	Y	T	R	P	S	A	B
111	01	11	1111	10	01	000010	11	0100	100	001000	01	10
111	01	11	1111	10	01	000001	11	1000	101	000011	01	10
111	01	11	1111	10	01	000001	11	0001	111	000100	10	10
111	01	11	1111	10	01	010000	11	0010	111	110000	01	10
111	01	11	1111	01	10	000001	10	1000	101	000011	01	10
RELWORK												
111	01	11	1111	11	11	111111	11	1111	111	001011	01	11
111	01	11	1111	11	11	111111	11	1111	111	000100	10	11

10600

10602

FIG. 106

REL M												
V	K	I	M	E	F	Y	T	R	P	S	A	B
111	11	11	0100	11	11	111111	11	1111	111	001000	11	11

10700

FIG. 107

RELWORK												
111	01	11	0100	11	11	111111	11	1111	111	001000	01	11

10800

FIG. 108

MAIN SPACE												
V	K	I	M	E	F	Y	T	R	P	S	A	B
111	01	11	0100	10	01	000010	11	0100	100	001000	01	10
RELWORK												
111	11	11	1111	11	11	111111	11	1111	111	111111	11	11

10900

FIG. 109

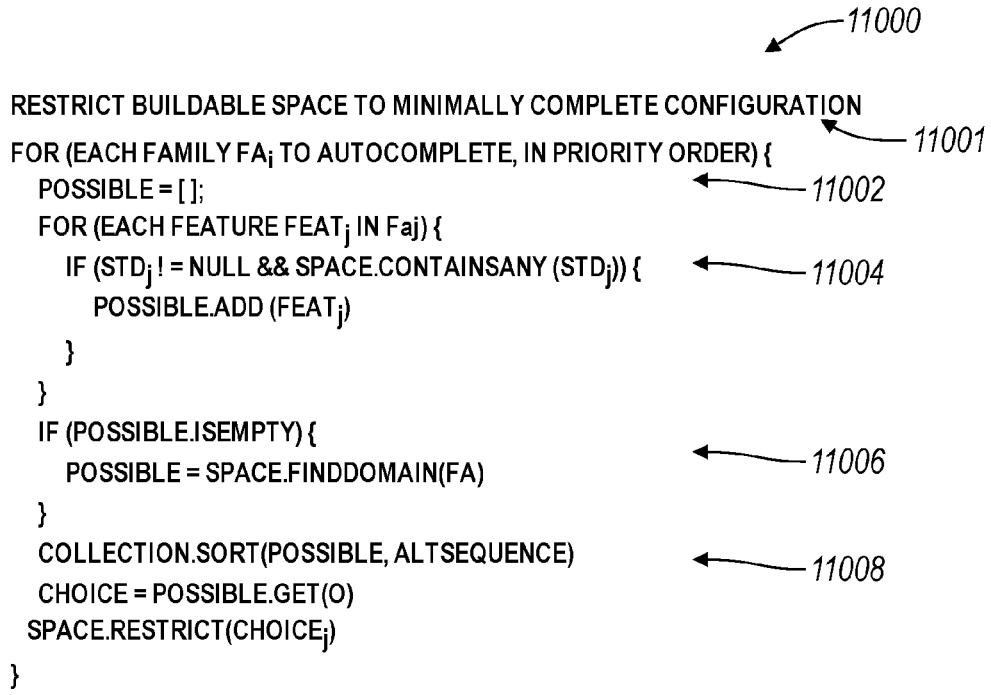


FIG. 110

MAIN SPACE												
V	K	I	M	E	F	Y	T	R	P	S	A	B
111	11	11	1111	10	01	001000	11	0001	100	000100	10	11
111	11	11	1111	10	01	001000	11	1000	100	000011	01	11
RELWORK												
111	11	11	1111	11	11	111111	11	1111	111	111111	11	11

11100

FIG. 111

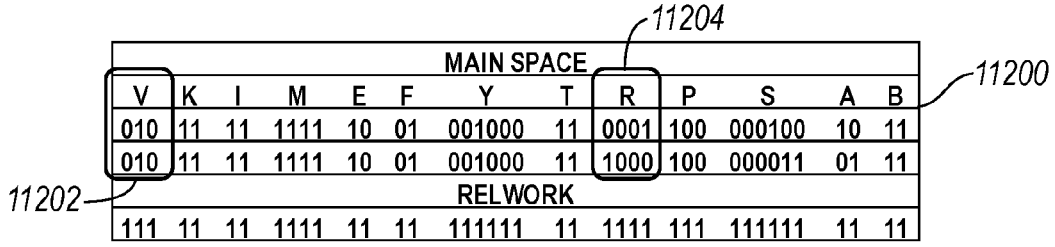


FIG. 112

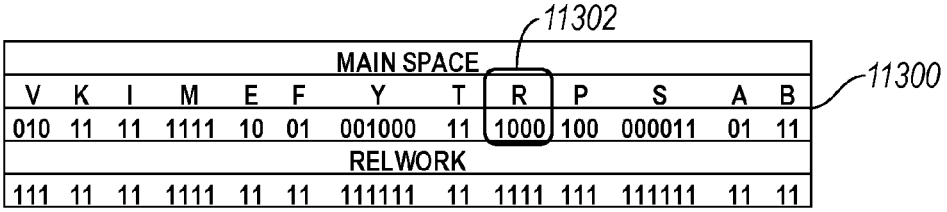


FIG. 113

MAIN SPACE												
V	K	I	M	E	F	Y	T	R	P	S	A	B
111	01	11	1111	10	01	001000	11	1000	100	000011	01	11
RELWORK												
111	01	11	1111	11	11	111111	11	1111	111	000100	10	11
111	01	11	1111	11	11	111111	11	1111	111	001011	01	11

11400

FIG. 114

MAIN SPACE												
V	K	I	M	E	F	Y	T	R	P	S	A	B
010	01	11	1111	10	01	001000	11	1000	100	000011	01	01
RELWORK												
111	01	11	1111	11	11	111111	11	1111	111	000100	10	11
111	01	11	1111	11	11	111111	11	1111	111	001011	01	11

11500

FIG. 115

MAIN SPACE												
V	K	I	M	E	F	Y	T	R	P	S	A	B
010	01	11	1000	10	01	001000	11	1000	100	000011	01	01
RELWORK												
111	11	11	1111	11	11	111111	11	1111	111	111111	11	11

11600

FIG. 116

MAIN SPACE												
V	K	I	M	E	F	Y	T	R	P	S	A	B
010	01	10	1000	10	01	001000	10	1000	100	000010	01	01
RELWORK												
111	11	11	1111	11	11	111111	11	1111	111	111111	11	11

11700

FIG. 117

RESOLVING CONFIGURATION CONFLICTS USING A MULTI-VALUED DECISION DIAGRAM

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of U.S. provisional application Ser. No. 62/280,609 filed Jan. 19, 2016 and U.S. provisional application Ser. No. 62/352,463 filed Jun. 20, 2016, the disclosures of which are hereby incorporated in their entirety by reference herein.

TECHNICAL FIELD

[0002] One or more embodiments generally relate to systems and methods for configuring a product.

BACKGROUND

[0003] Product configuration is an aspect of industries that offer customizable products with a wide variety of features. The process of selecting a configuration, or features that include a configuration, is used in multiple aspects of marketing and sales, order management and production planning, and product development. Examples include virtually constructing an ideal product (e.g., a vehicle) using a build-and-price application or selecting features for a prototype.

[0004] The product definition or product offering in the automotive industry is often of staggering dimensionality and size. It is common for a vehicle to be offered with thirty or more optional feature categories, such as paint color, engine size, radio type, and wheel style. Allowing a user to quickly explore a complex space that could include more than 10^{30} valid configurations is a challenging problem in constraints programming.

SUMMARY

[0005] In one embodiment, a system is provided with a memory device and a processor. The memory device is adapted to store data representative of a multi-valued decision diagram (MDD) specifying a buildable space of all possible valid configurations of a vehicle. The MDD includes a root node, a truth node, and at least one level of intervening nodes. Each level of the MDD corresponds to a family of mutually-exclusive features represented by at least one node. Each intervening node of a level connects to nodes of a next adjacent level by outgoing edges having labels indicating valid features of the family and to nodes of a prior adjacent level by incoming edges that are outgoing edges of the prior adjacent level, such that a complete path from the root node through the outgoing edges to the truth node defines at least one of the valid configurations. The processor is in communication with the memory and is programmed to identify an invalid configuration, and to generate a restricted buildable space, including to determine an edit distance of each complete path indicative of a number of features to change the invalid configuration of that path to one of the valid configurations, identify a minimum of the edit distances, and remove configurations having edit distances larger than the minimum. The processor is further programmed to identify at least one feature to change the invalid configuration to at least one valid configuration based on the restricted buildable space; and to generate output indicative of the at least one feature to change.

[0006] In another embodiment, a method is provided for storing, in a memory, data representative of a multi-valued decision diagram (MDD) specifying a buildable space of all possible valid configurations of a vehicle. The MDD includes a root node, a truth node, and at least one level of intervening nodes, each level of the MDD corresponding to a family of mutually-exclusive features represented by at least one node. Each intervening node of a level connects to nodes of a next adjacent level by outgoing edges having labels indicating valid features of the family and to nodes of a prior adjacent level by incoming edges that are outgoing edges of the prior adjacent level, such that a complete path from the root node through the outgoing edges to the truth node defines at least one of the valid configurations. Input indicative of a non-guided resolution is received. An invalid configuration is identified. A restricted buildable space is generated, including: an edit distance of each complete path is determined that is indicative of a number of features to change the invalid configuration of that path to one of the valid configurations, a minimum of the edit distances is identified, and configurations having edit distances larger than the minimum are removed. A further restricted buildable space having a single target configuration is generated, including: a weight of each path in the restricted buildable space is determined that is indicative of a priority of its features, the maximum weight of all paths is identified, and configurations having weights that are less than the maximum weight are removed. At least one feature to change the invalid configuration to at least one valid configuration is identified, based on the further restricted buildable space. And output indicative of the at least one feature to change is generated.

[0007] In yet another embodiment, a method is provided for storing, in a memory, data representative of a multi-valued decision diagram (MDD) that specifies a buildable space of all possible valid configurations of a vehicle. The MDD includes a root node, a truth node, and at least one level of intervening nodes. Each level of the MDD corresponds to a family of mutually-exclusive features represented by at least one node. Each intervening node of a level connecting to nodes of a next adjacent level by outgoing edges having labels indicating valid features of the family and to nodes of a prior adjacent level by incoming edges that are outgoing edges of the prior adjacent level, such that a complete path from the root node through the outgoing edges to the truth node defines at least one of the valid configurations. Input is received that is indicative of a guided resolution. An invalid configuration is identified. A restricted buildable space is generated that has multiple target configurations, including: an edit distance is determined of each complete path indicative of a number of features to change the invalid configuration of that path to one of the valid configurations, a minimum of the edit distances is identified, and configurations are removed that have edit distances larger than the minimum. At least one family to change is identified, including: a bitset of the invalid configuration is determined, a domain bitset of the restricted space is determined, a bitwise conjunction of the invalid configuration bitset and the restricted space bitset is calculated, and the at least one family to change is identified, as a family having no active bits in the bitwise conjunction. A message is provided to a user that displays at least one feature to change from each family to change, to resolve the at least one feature.

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] FIG. **1** is a block diagram of a product configuration system, according to one or more embodiments;

[0009] FIG. **2** is an application programming interface illustrating an application of the product configuration system of FIG. **1** including a configuration engine;

[0010] FIG. **3** is a table illustrating configurations expressed in conjunctive normal form;

[0011] FIG. **4** is a table illustrating configurations expressed in conjunctive normal form and in binary form;

[0012] FIG. **5** is a table illustrating mappings of the configurations of FIG. **4**;

[0013] FIG. **6** is a table illustrating multiple configurations and a superconfiguration;

[0014] FIG. **7** is a table illustrating multiple superconfigurations;

[0015] FIG. **8** is a table illustrating the interaction of superconfigurations;

[0016] FIG. **9** is another table illustrating the interaction of superconfigurations;

[0017] FIG. **10** is a table depicting a buildable space according to one or more embodiments;

[0018] FIG. **11** is a table depicting overlapping configurations;

[0019] FIG. **12** is a table depicting a feature mask;

[0020] FIG. **13** is a multi-valued decision diagram (MDD) representing the buildable space of FIG. **10**, according to one or more embodiments;

[0021] FIG. **14** is a comparison table;

[0022] FIG. **15** is a diagram illustrating a reduction of the MDD of FIG. **13**;

[0023] FIG. **16** is a diagram illustrating merging of duplicate MDD nodes;

[0024] FIG. **17** is a diagram illustrating MDD compression with deterministic families;

[0025] FIG. **18** is a window displayed to a user based on a conflict resolution procedure of the configuration engine of FIG. **2** according to one embodiment;

[0026] FIG. **19** is a window displayed to the user based on a conflict resolution procedure of the configuration engine of FIG. **2** according to another embodiment;

[0027] FIG. **20** is a diagram illustrating a containsAny operation performed by the configuration engine according to one embodiment;

[0028] FIG. **21** is another diagram illustrating a containsAny operation performed by the configuration engine according to another embodiment;

[0029] FIG. **22** is a table listing a restricted buildable space of the buildable space in FIG. **10**;

[0030] FIG. **23** is a diagram illustrating a restricted buildable space of the buildable space in FIG. **13** and the table of FIG. **22**;

[0031] FIG. **24** is a flowchart illustrating a method for evaluating an MDD using reversible restrictions, according to one or more embodiments;

[0032] FIG. **25** is a diagram illustrating an example of various steps of the method of FIG. **24**;

[0033] FIG. **26** is a flowchart illustrating a subroutine of the method of FIG. **24**;

[0034] FIG. **27** is another flowchart illustrating a subroutine of the method of FIG. **24**;

[0035] FIG. **28** is yet another flowchart illustrating a subroutine of the method of FIG. **24**;

[0036] FIG. **29** is a comparison table;

[0037] FIG. **30** is a table illustrating a reduction of the buildable space of FIG. **10**;

[0038] FIG. **31** is a table illustrating a projected space after the overlap has been removed and the space has been compressed;

[0039] FIG. **32** is a diagram illustrating the table of FIG. **30**;

[0040] FIG. **33** is a table illustrating combinations of features of the buildable space of FIG. **13**;

[0041] FIG. **34** is a flowchart illustrating a method for determining MDD feature states according to one or more embodiments;

[0042] FIG. **35** is a table illustrating a set of superconfigurations;

[0043] FIG. **36** is an example of a table illustrating a restricted domain after various steps of the method of FIG. **34**;

[0044] FIG. **37** is another example of a table illustrating a restricted domain after various steps of the method of FIG. **34**;

[0045] FIG. **38** is a table illustrating an example of the results of the method of FIG. **34**;

[0046] FIG. **39** is a flowchart illustrating another method for determining MDD feature states according to one or more embodiments;

[0047] FIG. **40** is a flowchart illustrating a subroutine of the method of FIG. **39**;

[0048] FIG. **41** is another flowchart illustrating a subroutine of the method of FIG. **39**;

[0049] FIG. **42** is yet another flowchart illustrating a subroutine of the method of FIG. **39**;

[0050] FIG. **43** is another flowchart illustrating a subroutine of the method of FIG. **39**;

[0051] FIG. **44** is yet another flowchart illustrating a subroutine of the method of FIG. **39**;

[0052] FIG. **45** is a diagram illustrating an example of various steps of the method of FIG. **39**;

[0053] FIG. **46** is another diagram illustrating an example of various steps of the method of FIG. **39**;

[0054] FIG. **47** is a flowchart illustrating a method for resolving conflicts between configurations according to one or more embodiments;

[0055] FIG. **48** is a flowchart illustrating a subroutine of the method of FIG. **47**;

[0056] FIG. **49** is a diagram illustrating an example of various steps of the method of FIG. **47**;

[0057] FIG. **50** is another flowchart illustrating a subroutine of the method of FIG. **47**;

[0058] FIG. **51** is yet another flowchart illustrating a subroutine of the method of FIG. **47**;

[0059] FIG. **52** is still yet another flowchart illustrating a subroutine of the method of FIG. **47**;

[0060] FIG. **53** is a table illustrating an example of additions and subtractions for the diagram of FIG. **49** according to various steps of the method of FIG. **47**;

[0061] FIG. **54** is a window illustrating an example of additions and subtractions that are displayed to the user based on various steps of the method of FIG. **47** according to one embodiment;

[0062] FIG. **55** is a window illustrating an example of additions and subtractions that are displayed to the user based on various steps of the method of FIG. **47** according to another embodiment;

[0063] FIG. **56** is a table illustrating a compression of three superconfigurations to two superconfigurations, according to one embodiment;

[0064] FIG. **57** is a window illustrating an example of a resolution object that is displayed to the user based on various steps of the method of FIG. **47** according to another embodiment;

[0065] FIG. **58** is another flowchart illustrating a subroutine of the method of FIG. **47**;

[0066] FIG. **59** is yet another flowchart illustrating a subroutine of the method of FIG. **47**;

[0067] FIG. **60** is still yet another flowchart illustrating a subroutine of the method of FIG. **47**;

[0068] FIG. **61** is another flowchart illustrating a subroutine of the method of FIG. **47**;

[0069] FIG. **62** is yet another flowchart illustrating a subroutine of the method of FIG. **47**;

[0070] FIG. **63** is still yet another flowchart illustrating a subroutine of the method of FIG. **47**;

[0071] FIG. **64** is a diagram illustrating an example of various steps of the method of FIG. **47**;

[0072] FIG. **65** is a table listing an invalid configuration as a bitset;

[0073] FIG. **66** is a table illustrating the minimum edit space of the configuration of FIG. **65** converted to a matrix, as determined by various steps of the method of FIG. **47**;

[0074] FIG. **67** is a table illustrating a bitwise conjunction (AND) of the domain of the edit space from FIG. **66** with the invalid configuration from FIG. **65**;

[0075] FIG. **68** is a table illustrating the minimum edit space from FIG. **66** after it is trimmed to the families to change from FIG. **67**;

[0076] FIG. **69** is an example target matrix for a selection of a feature as determined by various steps of the method of FIG. **47**;

[0077] FIG. **70** is an example target matrix for a selection of another feature as determined by various steps of the method of FIG. **47**;

[0078] FIG. **71** is software code illustrating an example final resolution object, according to one or more embodiments;

[0079] FIG. **72** is a window illustrating an example prompt provided to the user as part of a guided resolution;

[0080] FIG. **73** is a window illustrating an example of another prompt provided to the user as part of the guided resolution;

[0081] FIG. **74** is a window illustrating an example of yet another prompt provided to the user as part of the guided resolution;

[0082] FIG. **75** is a diagram illustrating an example of various steps of the remove partial matches subroutine of FIG. **58**;

[0083] FIG. **76** is a table illustrating an example of various steps of the remove partial matches subroutine of FIG. **58**;

[0084] FIG. **77** is a table illustrating an example of various steps of the minimum edit space calculation of FIG. **48**;

[0085] FIG. **78** is another table illustrating an example of various steps of the minimum edit space calculation of FIG. **48**;

[0086] FIG. **79** is yet another table illustrating an example of various steps of the minimum edit space calculation of FIG. **48**;

[0087] FIG. **80** is still yet another table illustrating an example of various steps of the minimum edit space calculation of FIG. **48**;

[0088] FIG. **81** is another table illustrating an example of various steps of the minimum edit space calculation of FIG. **48**;

[0089] FIG. **82** is a flow chart illustrating a method for automatically completing a configuration according to one or more embodiments;

[0090] FIG. **83** is a diagram illustrating a buildable space;

[0091] FIG. **84** is a table that defines an alternate sequence structure for defining path weights of the buildable space of FIG. **83**;

[0092] FIG. **85** is a diagram illustrating an example of various steps of the method of FIG. **82** performed on the buildable space of FIG. **83**;

[0093] FIG. **86** is table illustrating path weight;

[0094] FIG. **87** is another table illustrating path weight;

[0095] FIG. **88** is yet another table illustrating path weight;

[0096] FIG. **89** is a diagram illustrating an example of various steps of the method of FIG. **82** performed on the buildable space of FIG. **85**;

[0097] FIG. **90** is another table illustrating path weight;

[0098] FIG. **91** is a table illustrating a matrix that defines the same buildable space as the diagram of FIG. **17**;

[0099] FIG. **92** is a table illustrating the standard feature conditions for the product definition defining the buildable space of FIG. **91**;

[0100] FIG. **93** is software code illustrating a method for automatically completing a configuration using a maximally standard algorithm according to one or more embodiments;

[0101] FIG. **94** is a diagram illustrating an example of various steps of the method of FIG. **93**;

[0102] FIG. **95** is a flowchart illustrating another method for automatically completing a configuration using a maximally standard algorithm according to one or more embodiments;

[0103] FIG. **96** is a flowchart illustrating a subroutine of the method of FIG. **95**;

[0104] FIG. **97** is another flowchart illustrating a subroutine of the method of FIG. **95**;

[0105] FIG. **98** is yet another flowchart illustrating a subroutine of the method of FIG. **95**;

[0106] FIG. **99** is still yet another flowchart illustrating a subroutine of the method of FIG. **95**;

[0107] FIG. **100** is a table illustrating a buildable space;

[0108] FIG. **101** is a table illustrating standard feature conditions;

[0109] FIG. **102** is a table illustrating an example of various steps of the method of FIG. **95**;

[0110] FIG. **103** is another table illustrating an example of various steps of the method of FIG. **95**;

[0111] FIG. **104** is yet another table illustrating an example of various steps of the method of FIG. **95**;

[0112] FIG. **105** is still yet another table illustrating an example of various steps of the method of FIG. **95**;

[0113] FIG. **106** is another table illustrating an example of various steps of the method of FIG. **95**;

[0114] FIG. **107** is yet another table illustrating an example of various steps of the method of FIG. **95**;

[0115] FIG. **108** is still yet another table illustrating an example of various steps of the method of FIG. **95**;

[0116] FIG. **109** is another table illustrating an example of various steps of the method of FIG. **95**;

[0117] FIG. **110** is software code illustrating an example of various steps of the method of FIG. **95**;

[0118] FIG. **111** is another table illustrating an example of various steps of the method of FIG. **95**;

[0119] FIG. **112** is yet another table illustrating an example of various steps of the method of FIG. **95**;

[0120] FIG. **113** is still yet another table illustrating an example of various steps of the method of FIG. **95**;

[0121] FIG. **114** is another table illustrating an example of various steps of the method of FIG. **95**;

[0122] FIG. **115** is yet another table illustrating an example of various steps of the method of FIG. **95**;

[0123] FIG. **116** is still yet another table illustrating an example of various steps of the method of FIG. **95**; and

[0124] FIG. **117** is another table illustrating an example of various steps of the method of FIG. **95**.

DETAILED DESCRIPTION

[0125] As required, detailed embodiments of the present invention are disclosed herein; however, it is to be understood that the disclosed embodiments are merely exemplary of the invention that may be embodied in various and alternative forms. The figures are not necessarily to scale; some features may be exaggerated or minimized to show details of particular components. Therefore, specific structural and functional details disclosed herein are not to be interpreted as limiting, but merely as a representative basis for teaching one skilled in the art to variously employ the present invention.

[0126] With reference to FIG. **1**, a product configuration system is illustrated in accordance with one or more embodiments and generally referenced by numeral **100**. The product configuration system **100** includes a server **102** and a configurator application **104**. The configurator application **104** includes a configuration engine **106**. The server **102** includes memory **108** and a processor **110** for storing and operating the configurator application **104**. The product configuration system **100** communicates with a user device, such as a personal computer **112** and/or a mobile device **114** (e.g., a tablet, a mobile phone, and the like), each of which include memory and a processor. The configurator application **104** includes a "front-end" application (shown in FIG. **2**) that may be installed to the user device **112**, **114** from a computer-readable storage medium such as a CD-ROM, DVD or USB thumb drive. Alternatively, the front-end application may be downloaded from the server **102** to the client device **112**, **114** via an internet connection **116**. The design and efficiency of the application **104** therefore allows it to be optimized to run on multiple various operating system platforms and on devices having varying levels of processing capability and memory storage.

[0127] The configurator application **104** allows a user to explore a product offering, where the product is defined by selecting multiple features. A common example is a build-and-price website that allows a user to customize a product by choosing features such as size and color. The configuration engine **106** validates the customized product (i.e., the configuration) as the user selects different features.

[0128] The product offering, or product definition, includes all of the allowed ways of combining features (or parts, options, or attributes) in order to make a complete product. For example, a company might sell a product in two levels (e.g., base and luxury), A1 and A2, and in three colors, B1, B2 and B3. Further, the company only offers the base model, A1, in one color, B1. Features are grouped into families, in this case family A-"level" and family B-"color." The product configuration space includes the following four configurations: A1B1, A2B1, A2B2, and A2B3. Product definition often comprises rules or constraints that limit or allow relationships between features. In this example the rule might be "A1 requires B1." A complex product can be defined by thousands or even tens of thousands of rules.

[0129] The configurator application **104** allows the user to select options or features to create or modify a configuration. When a user changes a configuration by adding and/or removing a feature, the configuration engine **106** validates the new configuration. To perform this validation, the configuration engine **106** determines if the selected configuration fits within the allowed product space. If the new configuration is invalid, the configuration engine **106** either prompts the user to make changes, or make the changes itself according to a predefined hierarchy. For example, the monitor of the personal computer **112** shown in FIG. **1** depicts a message window that is displayed to the user indicating changes to resolve the conflict between selected features. The validation and conflict resolution is performed quickly (e.g., less than 2 seconds) and uses little memory and processing. Ideally, this allows the user to explore the buildable space of allowable product configurations, often while viewing information associated with the configuration such as price or images.

[0130] Users interact with front-end applications that present the product information, e.g., on the monitor of their pc **112**, and allow them to explore the buildable space. Although, the configuration engine **106** is different from the front-end application that is displayed on the pc **112**, the applications interact closely with each other. Examples of front-end applications include build-and-price websites, mobile shopping applications, interactive shopping kiosks, and back-office order entry and management systems. When these front-end applications populate product information, they can communicate with the configuration engine **106**. The only people who interact directly with the configuration engine **106** are back-office users tending to the data stored in the server **102**. The product configuration application **104** primarily interacts with other applications.

[0131] In the computer science field, exploration of a configuration space that is defined by rules or constraints is called constraint programming. A configuration space can also be defined by a model that is a representation of the possible product configurations. Many methods exist in the literature to solve configuration problems, with some using a constraints programming approach and others operating on product definition models.

[0132] Referring to FIG. **2**, an application programming interface (API) is illustrated in accordance with one or more embodiments, and generally referenced by numeral **120**. The API **120** illustrates the inputs, steps, and outputs of the configurator application **104**. The configurator application **104** is contained within the server **102** and the user device (pc **112** and/or mobile device **114**) according to one embodiment, and may be implemented using hardware and/or software control logic as described in greater detail herein.

[0133] The configurator application **104** includes a database (DB) **122** for storing data as a catalog entry. Each catalog entry will store the vehicle attributes (make, model,

year, etc.), buildable product space, and extended feature attributes (images, prices, detailed descriptions, sales codes, etc.). The data may be directly processed as is shown by a product definition data source **124**, or it may come through a catalog management API **126**. The configurator application **104** also includes an extract, transform and load (ETL) process **128** that provides an interface between the DB **122** and the product definition data source **124** and the catalog management API **126**. The configurator application **104** accesses data from the DB **122** and temporarily saves a copy of it in cache memory **130**.

[0134] The configurator application **104** includes one or more "front-end" applications or "top hats" **132** that are accessible from the user device **112**, **114**. Examples of such "front-end" applications **132** include consumer build and price sites, management lease ordering sites, and dealer ordering applications.

[0135] The configurator application **104** includes a service API **134** and a web service controller **136** for coordinating the user's requests. The service API **134** includes a configuration service **138**, an enumeration service **140** and a configuration details service **142**.

[0136] The configurator application **104** includes a plurality of processing engines **144**, for optimizing the data provided to the front-end applications **132**. The engines include: the configuration engine **106**, an image engine **146**, a mapping engine **148** and a pricing engine **150**. The image engine **146** provides one or more images associated with features of the configuration. The mapping engine **148** provides feature codes from one or more feature code dictionaries. For example, manufacturing systems and sales systems may use different codes for the same feature, and the mapping engine translates between the different coding schemes. The pricing engine **150** provides pricing associated with the complete configurations and with individual features. The pricing engine **150** can also provide changes in pricing associated with changes in the configuration or features.

[0137] The configurator application **104** also includes a catalog controller **152** that lists which product definitions are available. The catalog controller **152** receives a catalog request (e.g., what products are available), and provides a response (e.g., product A, product B and product C). Then the user selects a product, and the front end application **132** submits a configuration load request. Then the web service controller **136** returns a default configuration.

[0138] The web service controller **136** is a simple object access protocol (SOAP) web service using XML messages, according to one embodiment. The service API **134** provides an XML request to the web service controller **136** based on each user selection made in the front-end applications **132**. In other embodiments the web service controller **136** uses other protocol.

[0139] The web service controller **136** provides an XML response to the service API **134** in response to each XML request. The web service controller **136** parses each XML request, makes appropriate calls to the processing engines **144** or catalog controller **152**, and transforms the output of the configuration engine **106**, the image engine **146**, the mapping engine **148** and the pricing engine **150** into the appropriate XML response. The web service controller **136** includes author decorators **154** that are responsible for building up each portion of the response, and includes logic to retrieve data from the database **122** via the cache **130** and save the data as a working copy.

[0140] The configuration engine **106** operates with reference to a valid buildable space. The buildable space is also referred to as the product offering and is defined in terms of features that are grouped into mutually exclusive family sets. All of the possible features are grouped into families such that a valid configuration will contain one and only one feature from each family.

[0141] In one example product definition of a vehicle, (Vehicle A), exterior paint color is defined by the family PAA and its features are Green (PNWAD), Magnetic (PN4DQ), Blue1 (PNMAE), Red1 (PN4A7), Red2 (PNEAM), Black (PN3KQ), Silver (PN4AG), Orange (PN4AH), White (PNYW3), and Blue2 (PN4MAG).

[0142] The configurator application **104** uses extended feature attributes, or metadata associated with a feature, such as feature name and feature description when presenting the information to a user. However, the configuration engine **106** uses feature and family codes. A fully qualified feature is its "family dot feature" code, such as PAA.PNEAM to represent the paint family with paint color Red2.

[0143] Each point of variation in the product specification can be considered a dimension where a value is selected. The large number of these variation points on a motor vehicle results in many valid configurations. Traditionally the large number of valid configurations in this buildable space has been responsible for much computational effort to perform the configuration process, which may be referred to as "the curse of dimensionality." The configuration system **100** provides improvements over traditional systems because it operates efficiently even on highly complex buildable spaces, e.g. those containing more than 10^{24} valid configurations.

[0144] A configuration is a particular product instance specified by a set of features. Each feature belongs to exactly one family. There may be at most one feature selected from each family. A complete configuration contains exactly one feature from each and every family. A partial configuration will include features from a subset of the families, and one or more families will not have a feature choice specified.

[0145] In a build and price application, the configuration engine **106** will typically work in complete configurations. Partial configurations are useful for searching or filtering configurations, as would be done to find and amend orders in an order management system.

[0146] Some features are optional, such as moonroof, and a user may order a vehicle without the feature. But, a complete configuration must still contain a choice for the family. A "less feature" is used to specify the absence of an option. For example, Vehicle A, which has an optional power moonroof, the family (CHA) contains two features: without the moonroof (i.e., "less moonroof"—CHAAA) and with the moonroof (i.e., "moonroof"—CHAAC). The buildable space defines all valid configurations.

[0147] With reference to FIGS. **3-5**, a configuration can be expressed in a variety of different forms. A configuration can be expressed in conjunctive normal form (CNF). For example, in one embodiment, a product is defined by four families: A, B, C and D; where A, C, and D each have two possible choices, and B has three possible choices. If there are no restrictions on what features can occur together (other than a single feature choice for each family), the set of choices can be defined as (A1|A2) & (B1|B2|B3) & (C1|C2)

& (D1|D2). For a full configuration, each clause will reduce to a single literal or selection for each family (e.g., A1, B1, C1, D1).

[0148] FIG. **3** depicts a CNF Table **300** with two possible configurations. The CNF table **300** includes a first CNF configuration **302** and a second CNF configuration **304**.

[0149] With reference to FIG. **4**, a configuration can also be expressed in binary form by a string of 0's and 1's, as depicted by Table **400**. In binary form, a zero (0) indicates the absence of a feature in a configuration, and a one (1) indicates the presence of a feature in the configuration. For example, a first binary configuration **402** illustrates the first CNF configuration **302** in binary form, and a second binary configuration **404** illustrates the second CNF configuration **304** in binary form. The literal A2 is replaced with "01", as generally referenced by numeral **406**. Each column maps to a feature in the family: '0' in the first position indicates the feature A1 is not present in the first binary configuration **402**, and '1' in the second position indicates the feature A2 is present in the first binary configuration **402**. The first CNF configuration **302** (A2 & B2 & C1 & D2) is represented by the first binary configuration **402**, where the ampersand (&) symbol is replaced by a space as a delimiter between each family, i.e., "01 010 10 01."

[0150] A configuration space can be very large. For example, a product definition with around 100 independent features could have over 10^{27} possible configurations. If each configuration was stored as an uncompressed set of 0's and 1's, it would take more than 12 billion Exabytes of memory to represent them all—which is not practical with the present day state of the art computer hardware. Therefore the configurator application **104** uses compressed representation to facilitate computational tractability and efficiency within available memory.

[0151] A bit is the basic unit of information in computing. It can only have one of two values and is commonly represented as 0 or 1, or the Boolean values of false and true. A bit set represents a vector or array of bits. For example, Java has a native utility class called "BitSet" that stores a set of bits with an array of values of a primitive 64-bit datatype called long.

[0152] With the rightmost position of the bit set index 0 and leftmost position index 8, 100101001 is equal to $1*2^8+1*2^5+1*2^3+1*2^0=256+32+8+1=297$. Thus the bit set 100101001 can be stored using the long (or integer) 297. An array of 2 long values (−8929791190748339525, 8791258171646) can represent a bit set of 107 features: 11 01 11 010 100 01 10 00010 00010 111 01 010 01 11 000100 000011 00100 00010 00010 11111 1100 01000001 001 01 11 11 11 01 10 11 11 11.

[0153] The Java BitSet class allows for varying bit set lengths. But, for the configurator application **104**, the set of features is fixed for a given vehicle. Thus, the configuration engine **106** defines its own fixed length bit set object. To store a configuration, a feature bit set is used. The feature bit set associates a feature object with each bit position. This mapping is defined in a structure object that defines a list of families with each family defining a list of features.

[0154] FIG. **5** illustrates a table **500** that defines the mappings for the bit sets in table **400**. The bit sets shown in table **400** are printed in blocked format. The bits for each family are grouped together with no delimiter and family blocks are separated by a space. The bit sets can also be printed in strict binary format with no extra spacing, e.g., the first binary configuration **402** of Table **400** could be rewritten as 010101001. Alternatively, the bit sets can be printed in strict binary format with the same delimiter between every feature and no extra delimiter between families, e.g., the first binary configuration **402** of Table **400** could be rewritten as 0 1 0 1 0 1 0 0 1. Blocked format allows for better human readability; however, space-delimited is a viable option when viewing the data in a table with labeled columns and importing bit set data into an Excel worksheet for analysis.

[0155] With reference to FIGS. **6-10**, one or more configurations may be compressed and represented by a "superconfiguration." FIG. **6** includes a table **600** listing a first configuration **602**: (A2, B2, C1, D2), and a second configuration **604**: (A1, B2, C1, D2). Because the values are identical in all but one family 'A', the configurations **602**, **604** can be compressed into a single superconfiguration **606**. The table **600** shows both the CNF and bit set forms of the configurations and superconfigurations. Thus, a configuration is just a superconfiguration with only one possible value per family.

[0156] Referring to FIG. **7**, two superconfigurations can also be compressed as shown in table **700**. A first superconfiguration **702** and a second superconfiguration **704** are compressed into a third superconfiguration **706**.

[0157] With reference to FIG. **8**, bit sets, configurations and superconfigurations can be interacted using bitwise steps as shown in table **800**. Referring to "OR" step **802**, when OR-ing two superconfigurations the resulting value for each family is the union of the family set from each superconfiguration. Thus, if any value in a family set (column) is "1", then the union of the family set is "1." And referring to "AND" step **804**, when AND-ing two superconfigurations the resulting value for each family is the intersection of the family set from each superconfiguration. Thus, if all values in a family set (column) are "1", then the union of the family set is "1."

[0158] Referring to FIG. **9**, when AND-ing two superconfigurations, the resulting superconfiguration is invalid if any family has no active bits, as shown in table **900**. The intersection for family B is empty causing the configuration to be invalid, as referenced by numeral **902**.

[0159] With reference to FIG. **10**, the configuration system **100** expresses buildable space with a set of superconfigurations that define all valid configurations, according to one or more embodiments. This compressed representation allows for more efficient storing and processing of a buildable space. A non-overlapping set of super configurations can be stored in a matrix **1000**, with each row stored as a bit set.

[0160] Referring to FIG. **11**, the configuration engine **106** does not use overlapping superconfigurations because they include redundant information which could lead to incorrect calculations. Thus, all basic steps performed by the configuration engine **106** rely on the fact that the buildable space contains no overlap. Two superconfigurations are said to overlap if there exists one or more configurations that are defined by both superconfigurations. Table **1100** shows a first superconfiguration **1102** that overlaps with a second superconfiguration **1104**. Row **1106** identifies all of the overlapping features. For example, both superconfigurations **1102**, **1104** include feature A2, as represented by numeral **1108**, and therefore overlap.

[0161] With reference to FIG. **12**, a superconfiguration can be used to define a feature mask. A feature mask for a single family is created by setting the active bits in the family to zero to define the constrained features and setting all bits to 1 for the remaining families. A feature mask for multiple families is the AND of the masks for each family. Table **1200** shows several examples. A feature mask can be used to search a configuration space (contains) or limit a configuration space (restrict). Feature masks can be used to encode a feature condition in order to associate data with features, such as descriptions, prices and images.

[0162] Referring to FIG. **13**, the configuration system **100** uses a multi-valued decision diagram (MDD) to represent the buildable space, according to one or more embodiments. An MDD representing the same buildable space as the superconfigurations matrix **1000** (FIG. **10**) is shown in accordance with one or more embodiments, and generally referenced by numeral **1300**. An MDD is capable of representing the configuration space for large and highly complex product offerings much more compactly than a matrix.

[0163] The MDD **1300** includes nodes, such as a root node (2), a terminal, Truth or True node (T) and a plurality of intervening nodes (3-16) arranged on a plurality of paths. Each path includes the root node (2), the true node (T) and some of the intervening nodes connected by "edges" or arrows. In terms of superconfigurations, a complete path from the root node (2) to the true node (T) defines a superconfiguration. The superconfiguration shown in the top row **1308** of the matrix **1000** in FIG. **10** (i.e., 100 100 110 100 10), is defined by a right path (2 3 4 5 6 T) in the MDD **1300**.

[0164] Each level of the MDD **1300** is associated with one family and the edges define the path for each feature. Each node will have at most one outgoing path for each feature, thus the features are mutually exclusive. Where there is a path between two nodes for more than one feature, a single edge is shown, but the edge label includes more than one ("1"). The edge labels correspond to a family's superconfiguration bits, where a "1" indicates the presence of a feature and a "0" indicates the absence of a feature. Where a feature is inactive, its edge points to the false terminal node, which is not shown in the diagram. Each complete path from the root node to the truth node is of the same length in nodes when the nodes are expanded, e.g., there are no long edges. An MDD that does not include any long edges may be referred to as a quasi-reduced MDD.

[0165] The MDD **1300** includes five different families **1302**: package (Pkg), radio (Radio), paint (Paint), trim (Trim) and moonroof (MoonRf). Each family includes multiple features. For example, the package (Pkg) family includes a 400 package, a 401 package and a 402 package; and the radio family includes: a Standard (Std) radio, a Deluxe (Dlx) radio and a Navigation (Nav) radio. The root node (2) is connected to intervening node 13 by edge label "001", which indicates the presence of the third package (402) and the absence of the first two packages (400 and 401). Again, some edge labels include more than one "1", which means that more than one feature is available. For example, intervening node (7) is connected to intervening node (8) by an edge with a label "011." This indicates that path 2, 7, 8 is a superconfiguration that includes the 401 package, and the Deluxe (Dlx) and/or the Navigation (Nav) radio.

[0166] FIG. **14** is a table **1400** that illustrates a comparison of the size of a matrix based product configuration (e.g., table **1000**, FIG. **10**) and an MDD based product configuration (e.g., MDD **1300**, FIG. **13**). For small product definitions (e.g., Vehicle A with 28 families) a matrix and an MDD have comparable size (20 KB). However, for medium product definitions (e.g., Vehicle B with 84 families) the matrix size (23,026 KB) is much larger than the MDD size (766 KB). For large product definitions (e.g., Vehicle C with 92 families) the matrix runs out of memory, but the MDD size (313 KB) is sufficient. Therefore table **1400** illustrates that an MDD is a useful tool when analyzing large product configurations. Using the MDD format, the minimum number of superconfigurations required to represent the buildable space of all possible valid configurations may be calculated. For complex products that number exceeds reasonably available memory.

[0167] With reference to FIG. **15**, the configuration engine **106** performs steps using reduced MDDs, such as reduced MDD **1500**, in one or more embodiments. FIG. **15** includes the MDD **1300** of FIG. **13** and a reduced MDD **1500**. In an MDD, a redundant node may be removed and its incoming and outgoing edges may be combined to form a "long edge." A redundant node is a node that corresponds to a family in which all features are active, i.e., its outgoing edge label includes only "1"s. For example, intervening node 16 in MDD **1300** corresponds to the moonroof (MoonRf) family, and its outgoing edge label "11" indicates that both the moonroof (Vista) and no moonroof (Less) features are active. Therefore node 16 is redundant and it can be removed, as depicted by the "X" disposed over it in FIG. **15**. Nodes 10 and 12 of MDD **1300** are also redundant and may be removed, as depicted by the X's over them in FIG. **15**. The reduced MDD **1500** shows the result of removing redundant nodes 16, 10 and 12 and collapsing their respective incoming and outgoing edges into long edges. The paths from 13-T, 9-T, and 10-T do not include a node for the moonroof (MoonRf) family. By collapsing the long edges, the reduced MDD **1300** has 3 fewer nodes, and some of its nodes have been renumbered. For example, since node 10 of MDD **1300** was removed, node 11 of MDD **1300** was renumbered as node 10 in reduced MDD **1500**. By reducing MDDs, the configuration system **100** reduces memory and processing usage.

[0168] In one or more embodiments, the configuration engine **106** performs steps using an expanded (not reduced) MDD, such as MDD **1300**, e.g., during a "Quick-Restrict" step, as described below. For a very large buildable space, the number of nodes added to expand long edges can be quite significant. As an example, an MDD may have 17,283 nodes when long edges are compressed, but grow to 47,799 nodes when long edges are expanded. Therefore good compression reduces the number of long edges significantly. However, the savings generated by the quick-restrict step are generally sufficient to justify the long edge expansion.

[0169] In the expanded MDD **1300**, nodes 10, 12, and 16 are not only redundant; they are also duplicated. Duplicate nodes are two nodes for a family (i.e., within a common row) that have identical outgoing edge labels. The reduced MDD **1500** is in canonical form, i.e., there are no duplicated nodes. The configuration engine **106** creates MDDs in canonical form. Canonical form helps to minimize the number of nodes required to represent the buildable space.

[0170] FIG. **16** includes a non-canonical MDD **1600** with duplicate nodes. Node 6 and node 8 include identical outgoing edge labels, and therefore are duplicates of each other, as referenced by numeral **1602**. Similarly, node 5 and node 7 are duplicates, as referenced by numeral **1604**. FIG. **16** also includes a canonical MDD **1610**. The duplicate nodes of MDD **1600** are merged in the canonical MDD **1610**. For example, the first duplicate nodes **1602** are merged to form a canonical node 6, as referenced by numeral **1612**. And the second duplicate nodes **1604** are merged to form a canonical node 5, as referenced by numeral **1614**. The MDDs, e.g., MDD **1600**, **1610**, are serialized as plain text for storage in a database, such as the DB **122** shown in FIG. **2**. Such text serialization ensures backwards compatibility across software versions and implementations.

[0171] With reference to FIG. **17**, when a family's values in the configuration space can be determined by the values from one or more other families, the family is said to be deterministic. Deterministic relationships are used to minimize the size of an MDD to allow for scaling to complex vehicle configurations.

[0172] FIG. **17** includes an MDD **1700** that is similar to the MDD **1300** of FIG. **13**, with the addition of two deterministic families: seat temperature control (Temp) and the presence of dice (Dice). The presence of dice is determined by the paint color. Thus, once the paint color (Paint) is known, there is just one choice for dice. If paint is White, then dice are not present (NoDice); however if paint is Red or Blue, then Fuzzy dice are present (FuzzyDice). The seat temperature control (Temp) is determined by the package (Pkg). If Pkg is 400, then the seat temperature control is not available (LessTemp); if Pkg is 401, then heated seat temperature control is included (Heat); and if Pkg is 402, then heat and cool temperature control (HeatCool) is included. In these examples Paint color and Pkg are the determinant families while Dice and Seat temp are deterministic. This is because their state is fully specified by the state of the determinant.

[0173] The presence of deterministic features has a negative impact on MDD compression. The MDD **1700** represents twenty-four configurations. However, the deterministic families cause the MDD **1700** to increase from sixteen nodes to twenty-four nodes. Generally, the number of nodes in an MDD is a good predictor of its performance. Thus, it is ideal to have an MDD with as few nodes as possible. To aid in MDD compression, the configuration system **100** extracts the relationship defining a deterministic family's values to form one or more external relationship MDDs, which allows for greater compression in the main MDD, i.e., MDD **1700**.

[0174] The configuration system **100** extracts the two deterministic families (Temp and Dice) from MDD **1700** to form reduced an MDD **1702**, a first external relationship MDD **1704** corresponding to the Dice family, and a second external relationship MDD **1706** corresponding to the Temp family. The deterministic families (Temp and Dice) remain in the MDD **1702**, but are trivialized—made all 1s—allowing the main MDD **1702** and the external relationship MDDs **1704**, **1706** to share the same structure.

[0175] The combination of the main MDD **1702** and all its external relationship MDDs **1704**, **1706** is referred to as a Global MDD. A Global MDD can be operated in the same way as an MDD; but, each step must account for both the main space and the relationships.

[0176] The configuration service **138** abstracts the implementation from any specific application and state management does not depend on the implementation being either "stateful" or "stateless", according to one or more embodiments. The configuration service **138** remembers a user's session history in a stateful implementation, and does not remember a user's session history in a stateless implementation. However, the front-end application **132** treats the configuration service **138** as stateless in so far as it does not need to handle session synchronization.

[0177] Each XML response to the front-end application **132** from the configuration service **138** includes a state object that is included in the next call. The content of the state object is not dictated by the configuration service **138**, but it contains any information necessary to maintain a user session. The state object is not modified by the front-end application **132**.

[0178] A configuration session begins with an XML request to load initial content (i.e., a "Load Request") to populate a starting point of initial screens on the front-end applications **132**. This is followed by additional XML requests to update the content ("Update Request") as user changes the configuration by selecting different options in the front-end application. The web service controller **136** responds with an XML configuration response that includes updated configuration content.

[0179] Each feature included in such configuration responses includes a selection or feature state attribute. In one embodiment, there are six different feature state values: Selected, Available, Included, Default, Excluded and Forbidden.

[0180] A Selected feature state is a feature that has been explicitly selected by the user and generated in response to a Load or Update XML request. The configuration service **138** cannot unselect this feature as a result of another selection without performing a conflict resolution procedure, except for features in a mutually exclusive feature group.

[0181] An Available feature state is a feature that is not currently selected by the user. But the user is free to select it without causing a conflict resolution procedure to be triggered. In a mutually exclusive group of features (e.g. exterior color), although only one feature can be selected at a time, the other features in that group will be marked as "Available"—they will only be marked as "Excluded" if they conflict with a user selected feature in another family.

[0182] An Included feature state is a feature which is the only available choice in the family. For example, this can occur when a selection of a feature requires another feature. This can be the result of either a package selection that includes this feature (e.g. "Climate Pack" includes "Air Conditioning"), or a "must"/"requires" relationship (e.g. "Heated Seats" requires "Leather Seats).

[0183] A Default feature state is a feature that was selected by the configuration service **138** as part of an automatic completion function, as described in detail below with reference to FIGS. **82-117**.

[0184] An Excluded feature state is a feature that conflicts with another user selection. Selecting this feature will prompt a conflict resolution procedure.

[0185] A Forbidden feature state is a feature that violates a constraint that cannot be resolved if selected. Attempting to select this feature will result in an error. This state has been introduced to support external constraints that restrict the valid buildable configurations.

[0186] The front-end application **132** alters the display of features based on their configuration state, in one or more embodiments. For example, in one embodiment, the front-end application **132** shows a symbol next to an excluded feature to indicate that it conflicts with a prior selection or it may choose not to display excluded or forbidden features.

[0187] In one embodiment, the configuration system **100** changes a feature from the Default feature state to the Selected feature in response to a user implicitly selecting the feature. For example, if Default features are presented as checkboxes or radio buttons, there is no action the user can take to check an already checked item. This means that while the user intended to select the feature, it is still marked as Default. Such an implicitly selected Default feature may complicate conflict resolution strategies. Thus, the configuration system **100** includes an option to change a feature from a Default feature state to the Selected feature state in response to a user viewing a default selection and not changing it.

[0188] The configuration service **138** will return a complete configuration to the front-end application **132** in response to any load or update request, according to one or more embodiments. This feature is referred to as "automatic completion." Default feature selections will be added to any Selected features or Included features to create a complete configuration with respect to displayable families. A feature is "displayable" if it can be displayed to the user, e.g. on the user's pc **112**; whereas a feature that cannot be displayed to the user is described as "no-display" or "not displayable." The front-end application **132** may choose to distinguish the automatic feature selections from those explicitly selected by the user, using each feature's state value returned in the configuration response.

[0189] Alternatively, in other embodiments the automatic completion feature is disabled. When the automatic completion feature is disabled, no default selections are made and the response will typically include an incomplete configuration. This enables different front-end application **132** designs where a user is prompted to actively select each feature in the configuration.

[0190] The configurator application **104** includes a conflict resolution procedure in which, in response to an update request to select a feature that leads to an invalid configuration; the configuration service **138** returns a new valid configuration with the newly selected feature and the minimum other changed features required to make the configuration valid. If the auto completion feature is enabled, the new configuration will include any necessary Default features. The web service controller **136** implements the conflict resolution feature to provide details on what features must be added and removed to resolve the conflict, as well as a "reject state" that is used if the user cancels the requested change.

[0191] With reference to FIG. **18**, the configurator application **104** includes a "single" conflict resolution strategy, according to one or more embodiments. The configuration service **138** resolves the conflict by finding a single valid configuration containing the new selection. FIG. **18** depicts a user interface **1800** that is displayed to the user on the user device (e.g., on the monitor of the PC **112**) by the front-end application **132** as part of the conflict resolution procedure. The user interface **1800** includes a message **1802** that alerts the user of a conflict (i.e., by selecting the adaptive cruise control feature, the Zetec package and the solar reflect windscreen features must be removed, and the titanium package must be added) and asks for confirmation that they still want to make the change. If the user cancels the change, e.g., by selecting the decline button **1804**, the subsequent view request includes a reject state to undo the prior selection (i.e., adaptive cruise control) in the configurator application **104**. The update request may unselect a feature. Since all families must have one feature selected to form a valid configuration, unselecting a feature is often equivalent to selecting the "less" feature in the family. Removing a feature from the configuration can also lead to a conflict. For example, if the user removes an included feature, the selected feature which includes it will also be removed.

[0192] Referring to FIG. **19**, the configurator application **104** includes a branched conflict resolution strategy, according to one or more embodiments. In branched conflict resolution, the configuration service **138** presents the user with a series of choices to help them select from a set of valid configurations. For example, FIG. **19** depicts a user interface **1900** that is displayed to the user on the user device (e.g., on the monitor of the PC **112**) by the front-end application **132**. The user interface **1900** includes a series of choices for a remote starter option (e.g., with or without (less) the remote starter), as referenced by numeral **1902**, and a series of choices for the color of the seats (e.g., Charcoal Black or Medium Light Stone), as referenced by numeral **1904**. In one embodiment, the branched conflict resolution strategy may be enabled by setting a return guided resolution flag (not shown), which is included in the communication between the front-end application **132** and the service API **134**.

[0193] With respect to state management, when a branched conflict resolution is returned in the response to the service API **134**, there will be no feature state because the new configuration isn't known until the user traverses the resolution tree, (i.e., selects from the options shown in the user interface **1900**). Once selections have been made, the front-end application **132** sends a second update request with all the changes made during resolution. At this time the response will include the new configuration state. Optionally, if there is only one target configuration, the response could include the new configuration state to save one call to the service.

[0194] In one or more embodiments, the conflict resolution strategy employed by the configurator application **104**, may add a feature or subtract a feature, which is referred to as "return delta" functionality. In one or more embodiments, the conflict resolution subtractions only contain Selected features removed from the configuration; and conflict resolution additions only contain Included features added to the configuration. If the new configuration caused a change in a Default feature, this is not included in the prompt to the user (e.g., not shown in the user interfaces **1800**, **1900**). If all changes are for default choices, there are no changes to report, and the response will not include conflict resolution.

[0195] Alternatively, in other embodiments, the response will include all additions and subtractions regardless of feature state when the request has set the return delta flag to true. This allows the front-end application **132** to inspect the resolution and apply some additional logic when deciding whether to prompt the user for a conflict or to silently make the changes.

[0196] The configuration engine **106** "validates" each configuration. A load request from the services API **134** may

include a full or partial configuration as a starting point. When a configuration is submitted to the configuration engine **106** with the load request, it is validated. By default, or when a validate flag is True, conflict resolution will be triggered and the response will include a valid configuration. However, if the request has set the validate flag to False, conflict resolution is not performed and an error message will be included in the response if the submitted configuration is not valid.

[0197] The configuration engine **106** performs steps on a buildable space in order to process a configuration request and generate data to build a response. The configuration engine **106** uses data structures and algorithms, including those based on multivalued decision diagrams (MDD) to perform the configuration steps. For comparison, some matrix based steps are also described.

[0198] In many cases the configuration engine **106** uses MDD steps to search the product space in the same way a structured query language (SQL) query searches a database. Both are preforming relational algebra. As appropriate, the SQL equivalent of each MDD step is described.

[0199] The configuration engine **106** checks if the buildable space defines a specific full configuration or a partial configuration, which is referred to as a "contains any" step.

[0200] In terms of SQL, this step is equivalent to performing a search and evaluating if there is at least one row in the result set. For example, consider the partial configuration (Dlx, Vista). If a database stored each configuration as a separate row, and each family choice as a string column, the SQL query would be SELECT*FROM mdd WHERE Radio='Dlx' AND Moonrf='Vista'.

[0201] For efficient storage, matrix and MDDs represent the product with superconfigurations. If each row in the database stored a superconfiguration with each feature as a Boolean column, the SQL would be SELECT*FROM mdd WHERE Dlx=TRUE AND Vista=TRUE.

[0202] For example, in one embodiment, the configuration engine **106** searches an MDD by stating the query as a feature mask. For example, to search for the partial configuration (Dlx, Vista) the mask would be 111 010 111 111 01. The radio family includes the following features: Standard (Std), Deluxe (Dlx) and Navigation (Nav). Since the search is limited to (Dlx), the only active bit corresponds to Dlx (i.e., 010) for the radio family. Additionally, the moonroof family includes: without a moonroof (Less) and with a moonroof (Vista). Since the search is limited to (Vista), the only active bit corresponds to Vista (i.e., 01). All other families are all 1s.

[0203] When the configuration engine **106** is performing a step to check for a partial configuration, one or more families will have all 1s. This means that the mask defines multiple configurations. The configuration engine **106** is querying the space to determine if any of the individual configurations defined in the feature mask superconfiguration are contained in the space. Thus the step is called "containsAny."

[0204] With reference to FIGS. **20** and **21**, the configuration engine **106** performs an MDD-based "containsAny" step using a depth-first search of the space to look for the first valid path defining at least one of the configurations from the mask. In a depth-first search, the configuration engine **106** starts at the root node, and traverses the edges in descending order of its features; an edge of 011 will be processed by first inspecting 001 and then 010.

[0205] FIG. **20** is an MDD **2000** that illustrates an example of the configuration engine **106** performing an MDD-based "containsAny" step for the partial configuration (Dlx, Vista). To determine if this partial configuration is valid, the configuration engine **106** performs a depth-first search using the feature mask 111 010 111 111 01. The search begins with the path 2-13. This path is aborted when the Radio feature Dlx is inactive on edge 13-14, as shown by the dashed edge **2002**. Next, the configuration engine **106** searches path 2-13-8-11-12-T. This path, highlighted by nodes in solid line, ends in True node **2004**, indicating a valid path has been found containing the partial configuration (Dlx, Vista). Note there are three additional paths containing (Dlx, Vista), 2-13-8-9-10-T, 2-7-8-11-12-T, 2-7-8-9-10-T; but, the configuration engine **106** stops the "containsAny" step after the first path is found.

[0206] FIG. **21** is an MDD **2100** that illustrates an example of the configuration engine **106** performing an MDD-based "containsAny" step for the partial configuration (Std, Vista). To determine if this partial configuration is valid, the configuration engine **106** performs a depth-first search using the feature mask 111 100 111 111 01. The search begins with path 2-13 which is aborted because neither edge 13-14, nor 13-8 is active for the standard radio feature (i.e., neither of the edge labels include a "1" in their first digit), as shown by dashed edges **2102**. Next the configuration engine **106** searches path 2-7, which is also aborted because Std is not active, as shown by dashed edges **2104**. Finally, the configuration engine **106** searches path 2-3-4-5-6 and aborts the search because 6-T is not valid for MoonRf.Vista, as shown by dashed edge **2106**. No paths are found containing both Std and Vista, thus this combination is found to be invalid.

[0207] The domain of a buildable space defines all the Available features—those features contained in one or more configurations. The domain can be represented as a bit set where each 1 (active feature) denotes a feature contained in the domain and each zero (inactive feature) denotes a feature absent from the domain. For any active bit in the domain, the space contains one or more configurations with that feature. If there is an inactive bit in the domain, the space contains no configurations with that feature. For a matrix, the domain is calculated by the OR of all superconfigurations in the space. The domain of the space shown in FIG. **10** is 111 111 111 111 11, because every feature is available in at least one configuration.

[0208] With an MDD, the configuration engine **106** calculates the domain by traversing the MDD in either a breadth-first or a depth-first manner, and using the active features on the edges to define the domain. In a breadth-first search, the configuration engine **106** starts with a root node, then explores neighbor nodes first before evaluating the next family. For example, the configuration engine **106** evaluates the MDD **2100** of FIG. **21** using a breadth-first strategy by starting with the root node 2 and evaluating path 2-13. Although path 2-13 is valid, the configuration engine evaluates neighbor nodes 7 and 3, i.e., paths: 2-7 and 2-3, before evaluating the next family, i.e., nodes: 14, 8 and 4. Once a path is determined to be invalid, the configuration engine **106** stops evaluating nodes farther down the path. For example, once path 13-14 is found to be invalid, the configuration engine **106** does not continue along the path to evaluate nodes 15 and 16. And as described above, in a depth-first search, the configuration engine **106** starts at the root node, and traverses the edges in descending order of its

features. In the depth-first search, levels (families) are not back-tracked until an invalid path, or the truth node, is encountered. In the configuration engine **106**, the full domain is not usually called, but rather domain is called on a restricted space. The domain of a restricted space is used in determining feature states.

[0209] The configuration engine **106** restricts a space by keeping only those configurations containing a specific feature or combination of features. With respect to a database query, the restrict step is equivalent to searching the table to find only those rows that match the query. The restricted features define the WHERE clause. Consider the partial configuration (Nav, Ruby, Vista). In terms of SQL, the query would be SELECT*FROM mdd WHERE Radio='Nav' AND Trim='Ruby' AND MoonRf='Vista'. In terms of superconfigurations, the step begins with creating a feature mask defining the query. This is the same feature mask that would be used for a containsAny step. For restrict, a space is created with the feature mask as its only superconfiguration. Then, the restricted space is created by the AND of the original space with the feature combination space.

[0210] FIGS. **22** and **23** show restrictions of the space defined in FIGS. **10** and **13**. In the table **1000** shown in FIG. **10**, the last two rows of superconfigurations contain the radio feature (Nav), the trim feature (Ruby) and both moonroof features (Vista and Less), which indicates that Nav and Ruby are available with the moonroof (Vista) or without the moonroof (Less). FIG. **22** is a table **2200** that depicts a restricted version of table **1000**, in which the five superconfigurations of table **1000** are restricted to two superconfigurations, and the Moonrf.Less bit is set to zero to remove the configurations for (Nav and Ruby and MoonRf.Less). FIG. **23** is a restricted MDD **2300** illustrating the restricted superconfigurations of table **2200**.

I. App. 3—Quick Restrict

[0211] For some algorithms, successive restricts will be performed on the same space. When an MDD is restricted, a new set of nodes is created and the edges are updated for the nodes to keep only the constrained features. When many restrict operations are performed, many node objects must be created as the MDD is replicated. For large MDDs, this can cause the memory usage or "footprint" to increase, and may also incur performance problems from "garbage collection", i.e., reclaiming memory occupied by objects that are no longer in use by the program. The configuration engine **106** addresses these issues using a "quick-restrict" strategy.

[0212] With reference to FIG. **24**, a method for evaluating an MDD using reversible restrictions (i.e., "quick-restrict") is illustrated in accordance with one or more embodiments and generally referenced by S**100**. The quick-restrict method S**100** is implemented as an algorithm within the configuration engine **106** using software code contained within the server **102**, according to one or more embodiments. In other embodiments the software code is shared between the server **102** and the user devices **112**, **114**. FIG. **25** illustrates an original MDD **2500**, and an MDD **2502** after it is "quick-restricted" to the partial configuration (Nav, Ruby, Vista) by the configuration engine **106** according to the quick-restrict method S**100** of FIG. **24**.

[0213] At S**102**, the configuration engine **106** saves a copy of the root node identity and a set of the node edges to the memory **108** (shown in FIG. **2**). The subroutine of S**102** is shown in the flowchart of FIG. **26**. At step S**104** the configuration engine identifies each node (y). Then at step S**106** the configuration engine copies or clones each outgoing edge of the node (y), and returns to step S**104** until each node (y) in the MDD **2500** is copied. Then at step S**108**, the configuration engine **106** returns the edge set and the identity of the root node to the main routine of FIG. **24**.

[0214] At step S**110**, the configuration engine **106** performs the quick-restrict subroutine for the given selected features. The subroutine of step S**110** is shown in the flowchart of FIG. **27**. At step S**112** the configuration engine **106** examines the cache memory for source (SRC) node (y) to determine if the quick-restrict subroutine has already been performed on node (y), (i.e., does cache(y) exist?) If cache (y) exists, then the configuration engine **106** proceeds to step S**114** and returns to the main routine. If cache(y) does not exist, the configuration engine **106** proceeds to step S**116**.

[0215] At step S**116**, the configuration engine **106** starts with array index zero, and sets the empty flag to true. By setting the empty flag to true, the configuration engine **106** assumes that there are no valid paths from the node, i.e., that all edges point to the false node. At step S**118**, the configuration engine **106** evaluates the array index (z) to determine if (z) is less than the number of node (y)'s children. A positive determination at step S**118** indicates that the configuration engine **106** has not analyzed all array indexes of node (y). If the determination is positive, the configuration engine **106** proceeds to step S**120**.

[0216] At step S**120**, the configuration engine **106** checks if y's child, or destination (DST) node, at array index (z) is false. If the configuration engine **106** determines that the child is false, then it proceeds to step S**122**, increments the array index (z) by one, and returns to step S**118**. With reference to FIG. **25**, the MDD **2500** does not show false nodes, but they are still present. For example, node 9 shows only one outgoing edge with the label "001." This indicates that array indexes zero and one are both false, but they are not shown extending to the false node on the MDD **2500** to avoid clutter. When analyzing node 9, the configuration engine **106** makes a positive determination at S**120** for array indexes zero and one, but makes a negative determination for array index two. If the determination at step S**120** is negative, the configuration engine **106** proceeds to step S**124**.

[0217] At step S**124**, the configuration engine **106** evaluates the quick-restricted features list for node (y) to determine if it contains feature (z). Otherwise, the configuration engine **106** sets y's child node along z to false at S**126** and then proceeds to step S**122** to increment the array index(z) by one.

[0218] At step S**128**, the configuration engine **106** check's if y's child (DST) at array index (z) is true. For example, with reference to MDD **2500**, node 16 connects to the true node (T) along array indexes zero and one. If the configuration engine **106** determines that the child node is the true node, then it proceeds to step S**130** and sets the empty flag to false (empty=false), which indicates that there is at least one edge that is connected to the true node (T), and then proceeds to step S**122**. If the determination at step S**128** is negative, the configuration engine proceeds to step S**132**.

[0219] For example, referring to MDD **2500**, node 3 is illustrated with one outgoing edge with the label "100", which indicates that node 3 includes the Standard radio, but

does not include the Deluxe radio or the Navigation radio. Since the Navigation radio was selected by the user, the configuration engine **106** determines that none of node 3's outgoing edges contain (z)'s corresponding feature. Therefore the configuration engine **106** sets node 3's children to false at S**126**, which is illustrated by disconnecting the outgoing edge to node 4 (as shown in MDD **2502**) and the edge label for path 3-4 is replaced with an edge label of "000." However, referring to MDD **2500**, node 7 includes two array indexes (011), which indicate that node 7 includes the Deluxe radio and the Navigation radio. Since the Navigation radio was selected by the user, the configuration engine **106** determines that one of node 7's outgoing edges contain (z)'s corresponding feature at S**124**, therefore the outgoing edge is not disconnected from node 8 (as shown in MDD **2502**) and the edge label for path 7-8 is replaced with an edge label of "001".

[0220] As shown in MDD **2502**, the configuration engine **106** removes the edge pointer for path 13-8 because Navigation is not an active feature for the Radio family (i.e., there was not a "1" in the third digit of the edge label); and removes the edge pointer for path 15-16 because Ruby is not an active feature for the Trim family at S**126**. Since only Vista is constrained for the moonroof family; the configuration engine **106** modifies the edge labels for paths 10-T and 12-T from "11" to "01". If the determination at step S**128** is negative, the configuration engine **106** proceeds to step S**132**.

[0221] At step S**132**, y's children, or (DST) nodes, are rewritten by the result of recursive invocation of the quick-restrict method on the child. All nodes of the MDD are processed in this manner.

[0222] At step S**134**, the configuration engine **106** checks y's child at array index (z) to determine if it's not a false node. If the determination is positive (e.g., if node (y) is connected to a valid node), then the configuration engine **106** proceeds to step S**136** and sets the empty flag to false, before incrementing the array index (z) at S**122**. However, if node (y) is connected to the false node along array index (z) then the configuration engine **106** proceeds directly to S**122** to increment the array index (z).

[0223] The quick restrict method S**110** operates on the nodes in a depth first search fashion. Steps S**118**-S**136** demonstrate an iterative process that operates on each array index (z) of a node (y) before proceeding to the next node. Once the configuration engine **106** has evaluated all array indexes (z) for a node (y), it will make a negative determination at step S**118** (i.e., z will be greater than or equal to the number of y's children), and proceeds to step S**138**.

[0224] At step S**138** the configuration engine **106** checks if all children (DST) of node (y) are false, i.e., it evaluates the empty flag to determine if any valid configurations were found. If all of y's children are false, the configuration engine **106** proceeds to step S**140**, sets node (y) to the false node, set cache(y) to y, and then returns the cache(y), i.e., saves the analysis of node (y) and returns to the main routine of FIG. **24**. Further, if the node does not contain any edges that conform to the constraint, then the edge pointer is disconnected from the child node in the MDD. If not all of the children nodes are false, then the configuration engine **106** proceeds to step S**142**, sets the cache(y) to (y), and then returns the cache(y), i.e., saves the analysis of node (y) and returns to the main routine of FIG. **24**.

[0225] For example, referring to FIG. **25**, the configuration engine **106** determines that node 3's features do not contain the Selected radio feature (Nav) at S**124**, and therefore sets node 3's child (node 4) to false at S**126**. Setting node 4 to false is represented by disconnecting the edge pointer between node 3 and node 4 in MDD **2502**. The configuration engine **106** determined that all of node 3's children were set to false at S**138**, and therefore set node 3 to false at S**140**. Setting node 3 to false is represented by disconnecting the incoming edge pointer to node 3 in the MDD **2502**.

[0226] Similarly, the configuration engine **106** disconnects the incoming edge pointer to node 15 at S**140**, because all of node 15's children were set to false at S**126**, which is depicted by the disconnected outgoing edge pointer of node 15 in MDD **2502**. Although the edge label for path 7-8 was revised at S**126**, the edge pointer was not disconnected. Therefore the configuration engine **106** determines that not all of node's 7 children are false at S**138**, and proceeds to S**142** without setting node 7 to false or disconnecting its incoming edge pointer in the MDD **2502**.

[0227] The quick-restrict subroutine S**110** is a recursive process. This process continues until all nodes are analyzed. The edges for complete paths from the root node (node 2) to the truth node (T) define the restricted configuration space.

[0228] At step S**144** the configuration engine **106** performs additional steps on the restricted MDD **2502**. In one embodiment, after completing a traversal of the MDD in a first direction (e.g., downward), the configuration engine **106** determines the domain for a restricted space by traversing the MDD again in the same direction (i.e., the configuration engine repeats S**110** for all nodes).

[0229] In another embodiment, the configuration engine **106** determines the domain at the same time as traversing the MDD in the first direction (i.e., during S**110**). Then at step S**144**, the configuration engine **106** changes direction (i.e., reverses) and traverses the MDD in a second direction, e.g., upwards from the truth node (T) to the root node (2). On the downward traversal, the configuration engine **106** trims the edges to conform to the constraint. On the upward traversal, the domain bit set is created from the remaining edges. Combining quick-restrict and domain in a single operation saves one traversal of the MDD. However, the operation modifies the MDD and the edges must be reset to undo the quick-restrict.

[0230] At S**146**, the configuration engine **106** restores the original set of node edges to the memory **108** (shown in FIG. **2**). The subroutine of S**146** is shown in the flowchart of FIG. **28**. At S**148** the configuration engine identifies each node (y). Then at step S**150** the configuration engine **106** copies each outgoing edge of the node (y), and returns to step S**148** until each node (y) in the MDD **2500** is copied. Then at step S**152**, the configuration engine **106** sets the MDD to the identity of the root node and returns to the main routine of FIG. **24**.

[0231] At step S**154** the configuration engine **106** determines if the user has selected different features. If the user has selected new features, the configuration engine **106** returns to S**110**. If the user has not selected new features, then the configuration engine **106** proceeds to step S**156** and deletes the copy of each node from S**102** to free memory.

[0232] As shown in MDD **2502**, paths 9-10-T and 11-12-T are duplicate paths, because they include the same features. As described above with reference to FIGS. **22-23**, the

restrict operation will reuse nodes to avoid duplicate nodes or sub-paths. Although, the quick-restricted MDD **2502** may contain more nodes than the restricted MDD **2300**, the configuration spaces defined by each are identical.

[0233] The quick-restrict method S**100** provides advantages over existing methods by performing successive restricts without creating a new MDD for every step. The configuration engine **106** saves the original edge pointers at S**102** and then quickly resets the MDD **2500** using the original edge pointers.

[0234] There are some cases, where a more efficient algorithm can perform the same operation without having to do the quick-restrict method, eliminating the time needed to reset the edge pointers. This time savings, while small, can be significant when working with extremely large configuration spaces. A "Restricted Domain" algorithm is one such algorithm.

[0235] In other embodiments, the configuration engine **106** determines a read-only restricted domain using an external set of edges (not shown). Instead of modifying the original MDD node edges the external edges are modified to reflect the restricted space. The configuration engine **106** restricts on the downward traversal and then updates the domain on the upward traversal. Such a method is a read-only, thread-safe operation, because the MDD is not modified.

[0236] The quick-restrict with domain operation method S**100** is slightly slower than this read-only approach for the same calculation; however, the time saved in the read-only operation is due to not having to reset the edges. FIG. **29** is a table **2900** illustrating a comparison of the performance of the quick-restrict with domain operation method S**100** to the read-only method. The larger and more complex the buildable space, the more nodes in the MDD, and the more time it requires to reset the edges after the quick-restrict method S**100**.

[0237] With reference to FIGS. **30-32**, the configuration engine **106** performs a "project" operation of an MDD to trim a space to a subset of the families while keeping all unique configurations, according to one or more embodiments. In terms of SQL, the projection operation is equivalent to specifying which columns of a table are returned in the query. To project the space to the package (Pkg) and the trim (Trim) families, the equivalent SQL would be: SELECT DISTINCT Pkg and Trim FROM mdd.

[0238] Selecting distinct configurations means that the resulting space should contain no duplicated configurations. During the MDD project operation, the configuration engine **106** removes any duplicated configurations (also called overlapping superconfigurations).

[0239] FIG. **30** is a table **3000** that shows the result of the configuration engine **106** reducing the buildable space in FIG. **10** to keep only the columns for the package and trim families. This space contains duplicate configurations. For example a configuration including the 401 package and Ruby trim is defined in Row 3 and in Row 4. Likewise a configuration including the 402 package and Ruby trim is defined in Row 3 and in Row 5. Therefore Row 4 and Row 5 are duplicates of Row 3 and can be removed. Further, Row 2 and Row 3 can be compressed to a single row. FIG. **31** is a table **3100** that shows the projected space after the overlap (duplicated configurations) has been removed and the space has been compressed. FIG. **32** is an MDD **3200** that represents table **3000**.

[0240] With reference to FIG. **33**, the configuration engine **106** lists, or enumerates all valid configurations that are utilized in conjunction with a constraint restricting the families to a subset of all families, according to one or more embodiments. Given a subset of families, enumeration will return all valid combinations of the features in those families. In terms of superconfigurations, enumeration is the opposite of compression.

[0241] The configuration engine **106** works with individual features which are added or removed from a current single configuration. While this suits the requirements of most ordering and build and price applications, in some cases, the configuration engine **106** enumerates the valid combinations of those features without working through all the possible paths.

[0242] The total number of possible permutations of features in a configuration model can be very large, so this configuration service is restricted to enumerating a reasonable subset of feature families. The configuration engine **106** can impose limits on the number of families that can be enumerated, however, it should be expected that a request resulting in more products than can be stored in a storage medium will not succeed.

[0243] FIG. **33** is a table **3300** that shows all valid combinations of paint and trim defined in FIG. **13**. The configuration engine **106** generates this list by first projecting the space to paint and trim. Next the configuration engine **106** traverses the MDD paths and expands each superconfiguration into individual configurations.

[0244] The configuration engine **106** determines if a new configuration is valid, in response to every configuration request. Where auto-completion is enabled, the configuration request will contain a full configuration, otherwise it will be a partial configuration. In either case, the configuration engine **106** validates the configuration request using the MDD "containsAny" operation. A configuration is valid if the containsAny operation returns true.

II. App. 2—Determining Feature States

[0245] Each feature included in the configuration response will have a feature state attribute, e.g. Selected, Available, Included, Default, Excluded or Forbidden. For a given set of selected features, the configuration engine **106** calculates the feature states for the remaining features.

[0246] There are multiple approaches for calculating feature states. In one embodiment, the configuration engine **106** calculates feature states using a basic algorithm that includes restricted domain steps which can be applied to both Matrices and MDDs. In another embodiment, the configuration engine **106** calculates feature states for MDDs using dynamic programming techniques.

[0247] With reference to FIG. **34**, a method for determining feature states using a restricted domain is illustrated in accordance with one or more embodiments and generally referenced by S**200**. The method S**200** is implemented as an algorithm within the configuration engine **106** using software code contained within the server **102**, according to one or more embodiments. In other embodiments the software code is shared between the server **102** and the user devices **112**, **114**.

[0248] First the configuration engine **106** identifies Forbidden features and Excluded features. At step S**202** the configuration engine **106** determines the restricted domain with respect to any initial locked features imposed on the

user. For example, the initial restriction can be timing point feature(s) which are features used to control the effectivity and visibility of configurations. For example, a new feature (e.g., new engine x) may be available only after the start of a new model year. Thus the first day of the new model year would be such a visible timing point. At step S**204**, the configuration engine **106** identifies features that are absent from the restricted domain, and classifies them as Forbidden features at S**206**. At step S**208**, the configuration engine **106** classifies any feature that is not Forbidden, as an Excluded feature unless it is assigned another feature state (i.e., Available, Included or Default) in the remaining steps of the algorithm.

[0249] At step S**210**, the configuration engine **106** first determines the restricted domain of all selections to identify an initial set of Available features. Then, for each selection F_j, the configuration engine **106** checks the restricted domain (selected—F_j) to identify Available features for Family j.

[0250] For example, FIG. **35** is a table **3500** illustrating a set of superconfigurations that define all valid configurations. FIG. **36** is a table **3600** that depicts the restricted domains used to determine the Available features and the Excluded features when the Selected features are Red paint and Ruby trim. A bit value of zero in table **3600** indicates that a feature is not Available, whereas a bit value of one indicates that the feature is Available.

[0251] First, the configuration engine **106** determines the restricted domain of all selections to identify an initial set of Available features at S**210**. For example, Red paint and Ruby trim are both available for the configurations listed in rows 3-5 of the table **3500**. Packages 401 and 402 are Available, but package 400 is not Available for the configuration listed in rows 3-5, as referenced by numeral **3502**. Therefore the restricted domain for the package family is "011", as referenced by numeral **3602**, which indicates that package 400 is not Available (i.e., "0"), and package 401 and 402 are Available (i.e., "11"). Any feature that is active in this domain is set as active in the availability bit set. Thus, the configuration engine **106** identifies package 401 and package 402 as being Available features, as referenced by numeral **3604**, and these features are set as active ("1") in the availability bit set, as referenced by numeral **3606**.

[0252] Next, the configuration engine **106** evaluates the restricted domain (selected—Ruby) to identify Available features for the trim family. To evaluate (selected—Ruby), the configuration evaluates configurations in which Red paint is Available. For example, Red paint is Available for the configurations listed in rows 1 and 3-5 of the table **3500**. Stone trim and Ruby trim are Available for the configurations listed in rows 1, and 3-5; but Charcoal trim is not Available, as referenced by numeral **3508**. Therefore the restricted domain for the trim family is "101", as referenced by numeral **3608**. The availability bit set for the trim family is revised to "101" based on this step, as referenced by numeral **3610**.

[0253] Then, the configuration engine **106** evaluates the restricted domain (selected—Red) to identify Available features for the paint family. To evaluate (selected—Red), the configuration evaluates configurations in which Ruby trim is Available. For example, Ruby trim is Available for the configurations listed in rows 3-5 of the table **3500**. Red paint and Blue paint are Available for the configurations listed in rows 3-5, but White paint is not Available, as referenced by numeral **3512**. Therefore the restricted domain for the trim family is "011", as referenced by numeral **3612**. The availability bit set for the paint family is revised to "011" based on this step, as referenced by numeral **3614**.

[0254] As shown in the availability bit set of table **3600**, the restricted domain for Red paint includes both Stone trim and Ruby trim. Both of these selections are Available without having to change the Red paint selection. The selection of Ruby trim excludes White paint, and shows that both Red and Blue paint are Available. Thus White paint would require the trim selection to be changed to something other than Ruby.

[0255] The resulting availability can be defined as bit set 011 011 011 101 11. The state of Red paint and Ruby trim will be Selected, all other active features will be Available and the inactive features, such as White paint and Charcoal trim, will be Excluded.

[0256] Referring back to FIG. **34**, the configuration engine **106** identifies any Included features at step S**212**. A feature is Included if there is only one Available feature for a non-Selected feature family. The non-Selected feature families listed in table **3600** are packaging (Pkg), radio (Radio) and moonroof (MoonRf). All of these feature families include more than one Available feature (i.e., each family includes more than one "1" in each cell). Thus, table **3600** shows no such Included features for a selection of Red paint and Ruby trim.

[0257] FIG. **37** is a table **3700** that depicts the restricted domains used to determine the Available features and the Excluded features when the Selected features are the 401 package (Pkg.401) and Red paint (Paint.Red). An Included feature can be the result of the interactions between multiple features. For example, table **3700** shows that if the 401 package and Red paint are Selected, then Ruby trim is an Included feature, as referenced by numeral **3720**, because it is the only possible trim choice that is compatible with the 401 package and Red paint.

[0258] Thus, the configuration engine **106** determines the feature states e.g. Selected, Available, Included, Excluded and Forbidden for a given set of selections using the method S**200**. This initial feature state determination is referred to as "Minimum Completion." FIG. **38** is a table **3800** that summarizes the results of the Minimum Completion determination when Red paint and Ruby trim have been selected from the product definition in FIG. **35**.

[0259] The configuration engine **106** determines the restricted domain by traversing the MDD. Performing the Minimum Completion operation using restricted domain means that for each additional selection, another restricted domain operation is performed. Thus, for N selections, N+2 restricted domain determinations are performed. These restricted domain operations include an initial restriction at S**202**, a restriction for the initial available set, followed by one for each feature at S**210**.

[0260] For MDDs, there is an alternate approach for determining the Available features using dynamic programming principles with a single operation that includes one downward traversal and one upward traversal of the MDD. This approach is more memory efficient and faster, especially for larger MDDs.

[0261] With reference to FIG. **39**, a method for determining feature states using dynamic programming is illustrated in accordance with one or more embodiments and generally referenced by S**300**. The method S**300** is implemented as an algorithm within the configuration engine **106** using soft-

ware code contained within the server **102**, according to one or more embodiments. In other embodiments the software code is shared between the server **102** and the user devices **112**, **114**.

[0262] At S**302**, the configuration engine **106** organizes the data into nodes by family and level. The subroutine of S**302** is shown in the flowchart of FIG. **40**. At step S**304**, the configuration engine **106** determines if a level node object exists, i.e., if the nodes are already organized by level. Otherwise, the configuration engine **106** proceeds to step S**306** and organizes the nodes into a level node array where the length of the array is equal to the number of families. The configuration engine **106** initializes each level array with an empty nodes list, i.e., sets the empty flag to true. At step S**308**, for each node (y) analyzed, the configuration engine **106** appends a node (e.g., adds a child node) to the analyzed node's level node list. Step S**308** is a recursive step, therefore the configuration engine **106** repeats S**308** until it finds the True node. After step S**308** the configuration engine **106** proceeds to step S**310** and returns to the main routine of FIG. **39** to analyze the nodes by level.

[0263] FIG. **45** is an MDD **4500** illustrating the data organized by level according to S**302** and a selection of Red paint and Ruby trim. The MDD 400 includes five levels. Level zero represents the package (Pkg) family, which includes three package features: 400, 401 and 402 that are depicted by edges (level arrays) 001, 010 and 100, respectively, that extend from node 2. Level one represents the Radio family, which includes three radio features: Std, Dlx and Nav that are depicted by edges (level arrays) 001, 010 and 100, respectively, that extend from nodes 3-5. Level two represents the Paint family, which includes three paint features: White, Red and Blue, that are depicted by edges (level arrays) 001, 010 and 100, respectively, that extend from nodes 6-8. Level three represents the trim family, which includes three trim features: Stone, Charcoal and Ruby, that are depicted by edges (level arrays) 001, 010 and 100, respectively, that extend from nodes 9-12. Level four represents the moonroof (MoonRf) family, which includes two moonroof features: Less and Vista, that are depicted by edges (level arrays) 01 and 10, respectively, that extend from nodes 13-16. Node 1 is the true node and node 0 is the false node (not shown).

[0264] At step S**312**, the configuration engine **106** initializes the state, or marking of each node, by creating a place for each value. For example, by default, all features are initially set to false (i.e., not marked) except the root node 2 and the true node 1. The root node 2 is set to true for the downward traversal (i.e., marked with a downward arrow); and the true node 1 is set to true for the upward traversal (i.e., marked with an upward arrow). Marking a node with a downward arrow indicates a valid partial configuration from the root node 2 down to the marked node and any intervening downward marked nodes along the path. Similarly, marking a node with an upward arrow indicates a valid partial configuration from the true node 1 up to the marked node and any intervening upward marked nodes along the path.

[0265] At S**314**, the configuration engine **106** creates a constraint object. The subroutine of S**314** is shown in the flowchart of FIG. **41**. The configuration engine **106** starts analyzing family level zero of the MDD **4500** (i.e., the package family) at step S**316**. At step S**318**, the configuration engine **106** determines if the family level (x) is less than the total number of families included in the MDD. If so, the configuration engine **106** proceeds to step S**320**.

[0266] At step S**320** the configuration engine **106** determines if the user has selected a feature for family level (x). If the user has selected a feature for family level (x), the configuration engine **106** proceeds to step S**322** and sets the Selected feature to the Allowed feature state and sets the non-selected features for family level (x) to not allowed. If no features are selected for family level (x), the configuration engine **106** proceeds to step S**324** and sets all features to the Allowed feature state. After steps S**322** and S**324**, the configuration engine **106** proceeds to step S**326** to evaluate the next family by incrementing family level (x) by one (i.e. x=x+1), then returns to step S**318**.

[0267] Once the configuration engine **106** has created a full constraint object using subroutine S**314**, the family level (x) will no longer be less than the total number of families, and the configuration engine **106** will make a negative determination at step S**318**. For example, after the configuration engine **106** evaluates the moonroof family (level 4), it will set x to five at step S**326**. Since there are five families in the MDD **4500**, the configuration engine **106** will determine that x (5) is not less than the number of families (5) at step S**318** and then proceed to step S**328** and then return to the main routine of FIG. **39**.

[0268] At step S**330** the configuration engine **106** initializes an availability bit set. The configuration engine **106** initializes the availability bit set by setting all bits to zero, which indicates that the features are Excluded. Whereas a bit set value of one indicates that a feature is Available.

[0269] At S**332**, the configuration engine **106** traverses the MDD **4500** in a downward direction. The subroutine of S**332** is shown in the flowchart of FIG. **42**. The configuration engine **106** starts analyzing the nodes included in family level zero of the MDD **4500** (i.e., the package family) at step S**334**. At step S**336**, the configuration engine **106** determines if the family level (x) is less than the total number of families included in the MDD. If so, the configuration engine **106** proceeds to step S**338**.

[0270] At step S**338**, the configuration engine **106** starts analyzing node zero. At step S**340** the configuration engine **106** compares the node number (y) to the number of nodes at the current level (x) to determine if y<Level (x) number of nodes. For example, initially, x is equal to zero and Level (0) has one node, therefore y is less than 1. If y is not less than the number of nodes at level (x), the configuration engine proceeds to step S**342** and increments the level (x) by one.

[0271] After a positive determination at step S**340**, the configuration engine **106** proceeds to step S**344** and sets the currently analyzed node or source node (SRC) to level(x) node (y); and sets the array index (z) extending from SRC to zero. The array index (z) corresponds to a feature of the family (x). At step S**346** the configuration engine **106** compares the array index (z) to the SRC's number of children. If z is not less than SRC's number of children, the configuration engine **106** proceeds to step S**348** and increments the node (y) by one. If the determination at step S**346** is positive, the configuration engine **106** proceeds to step S**350**.

[0272] At step S**350**, the configuration engine **106** sets the destination node (DST) to be the child node of node (y) along array index (z) (DST=SRC.child(z)). The configura-

tion engine **106** analyzes three conditions to determine whether or not to mark the destination node with a downward arrow:

[0273] 1) the destination node is not a false node;

[0274] 2) the source node was previously marked with a downward arrow; and

[0275] 3) the feature of array index (z) is allowed by constraint.

These three conditions are illustrated by steps S**352**-S**358** in FIG. **42**.

[0276] At step S**352**, the configuration engine **106** evaluates the DST node to determine if it is a false node. Otherwise, the configuration engine **106** proceeds to step S**354** to determine if the source node (i.e., the parent of the currently analyzed destination node) was previously marked with a downward arrow.

[0277] After a positive determination at S**354**, the configuration engine **106** determines if the feature of array index (z) is allowed by constraint at step S**356**, which was previously determined in steps S**320**-S**324** (FIG. **41**). If the conditions of steps S**352**, S**354** and S**356** are met, the configuration engine **106** proceeds to step S**358** and marks the destination node with a downward marker, by setting mark.down(DST) to true. If any of the conditions of steps S**352**, S**354** and S**356** are not met, the configuration engine **106** proceeds to step S**360** and increments the array index (z) by one.

[0278] Referring to FIG. **45**, the MDD **4500** illustrates the marked nodes after the downward traversal subroutine S**332** of FIG. **42** for a selection of Red paint and Ruby trim. The configuration engine **106** marks Node 2 with a downward marker at S**312** because the root node is a special case that always has a valid downward path. On the downward traversal of the MDD **4500**, a node is marked at S**358** if the conditions of steps S**352**-S**356** are met for the analyzed node. For example, the configuration engine **106** marks destination node 3 with a downward marker or arrow **4502** at S**356** because node 3 was determined to not be a false node at S**352**; node 3's source node (node 2) was previously marked with a downward arrow at S**354**; and the feature of the array index connecting node 2 and node 3 (i.e., Pkg.400) was determined to be allowed by constraint at S**356**. Since the user has not selected a packaging feature for this example, all packaging features are set to Allowed (see steps S**320**, S**324** of FIG. **45**.) Similarly, the configuration engine **106** determines that the remaining radio nodes and the paint family nodes (nodes 4, 5, 6, 7, and 8) have valid downward markers because the package (Pkg) family and the Radio family are not constrained. With respect to the partial configurations of Pkg and Radio, all features are Available because the user has not selected a feature from either family.

[0279] Regarding the trim level nodes (9-12), nodes 9 and 10 have downward markers because they were determined to not be false nodes at S**352**; their source nodes were marked with a downward arrow at S**354**; and they were determined to be on a valid path with Red paint at S**356**. However, nodes 11 and 12 do not have downward markers because they are not on a valid path with Red paint. Since the user has selected Red paint, the non-selected paint features (White and Blue) are set to not allowed at step S**322** (FIG. **41**) and thus the condition of step S**356** is not met for nodes 11 and 12.

[0280] Regarding the moonroof level nodes (13-16), node 14 has a downward marker because it is valid with Ruby trim; however node 13 does not have a downward marker because it is not on a valid path with Ruby trim. Since the user has selected Ruby trim, the non-selected trim features (Stone and Charcoal) are set to not allowed at step S**322** (FIG. **41**) and thus the condition of step S**356** is not met. Additionally, nodes 15 and 16 are not marked because their source nodes (11 and 12) were not marked, and thus nodes 15 and 16 do not satisfy the condition of step S**354**.

[0281] The downward traversal subroutine S**332** includes three for-loops. Once the configuration engine **106** has analyzed all paths it will make a negative determination at step S**336**, i.e., the level (x) will not be less than the number of families; and the configuration engine **106** will proceed to step S**362** and return to the main routine of FIG. **39**.

[0282] At S**364**, the configuration engine **106** traverses the MDD of FIG. **45** in an upward direction, which is illustrated by MDD **4600** in FIG. **46**. The subroutine of S**364** is shown in the flowchart of FIG. **43**. The configuration engine **106** starts analyzing the nodes included in the last family level of the MDD **4600** (i.e., the moonroof family) at step S**366**. At step S**368**, the configuration engine **106** determines if the family level (x) is greater than or equal to zero (i.e., not negative). If so, the configuration engine **106** proceeds to step S**370**.

[0283] At step S**370**, the configuration engine **106** starts analyzing node zero. At step S**372** the configuration engine **106** compares the node number (y) to the number of nodes at the current level (x) to determine if y<Level (x) number of nodes. For example, initially, x is equal to zero and Level (0) has one node, and therefore y is less than 1. If y is not less than the number of nodes at level (x), the configuration engine proceeds to step S**374** and decreases the level (x) by one.

[0284] After a positive determination at step S**372**, the configuration engine **106** proceeds to step S**376** and sets the currently analyzed node or source node (SRC) to Level(x) (y); and sets the array index (z) extending from SRC to zero. At step S**378** the configuration engine **106** compares the array index (z) to the SRC's number of children. If z is not less than SRC's number of children, the configuration engine **106** proceeds to step S**380** and increments the node (y) by one. If the determination at step S**378** is positive, the configuration engine **106** proceeds to step S**382**.

[0285] At step S**382**, the configuration engine **106** sets the destination node (DST) to be the child node of node (y) along array index (z) (DST=SRC.child(z)). The configuration engine **106** analyzes three conditions to determine whether or not to mark the destination node with an upward arrow:

[0286] 1) the destination node is not a false node;

[0287] 2) the destination node was previously marked with an upward arrow; and

[0288] 3) the feature of array index (z) is allowed by constraint.

These three conditions are illustrated by steps S**384**-S**388** in FIG. **43**.

[0289] At step S**384**, the configuration engine **106** evaluates the DST node to determine if it is a false node. Otherwise, the configuration engine **106** proceeds to step S**386** to determine if the destination node (i.e., the child of the currently analyzed source node) was previously marked with an upward arrow.

[0290] After a positive determination at S**386**, the configuration engine **106** determines if the feature of array index (z) is allowed by constraint at step S**388**, which was previously determined in steps S**320**-S**324** (FIG. **41**). If the conditions of steps S**384**, S**386** and S**388** are met, the configuration engine **106** proceeds to step S**390** and marks the source node with an upward marker, by setting mark. up(SRC) to true.

[0291] At steps S**392** and S**394**, the configuration engine **106** identifies Available features. The configuration engine **106** evaluates the source node to determine if it was previously marked with a downward arrow at S**392**; and if so it proceeds to step S**394** and sets the feature (z) of node (y) to Available. Thus, Available features are identified by inspecting each node on the upward traversal. If there is a valid downward path to the node, and a valid upward path from one of its destinations, the configuration engine **106** identifies this node as part of a valid path with respect to selections from other families. Any feature that is on the edge from the node to the destination is identified as Available, because it can be selected without requiring a change to the constrained features. If any of the conditions of steps S**384**-S**388** and S**392** are not met, the configuration engine **106** proceeds to step S**396** and increments the array index (z) by one.

[0292] Once the configuration engine **106** has analyzed all paths it will make a negative determination at step S**368**, i.e., the level (x) will not be greater than or equal to zero; and the configuration engine **106** will proceed to step S**398** and return to the main routine of FIG. **39**.

[0293] Referring to FIG. **39**, after determining the availability of all features at S**364**, the configuration engine **106** proceeds to step S**400** to determine the feature states. The subroutine of S**400** is shown in the flowchart of FIG. **44**. The configuration engine **106** starts analyzing the features included in family level zero of the MDD **4600** (e.g., the package family) at step S**402**. At step S**404**, the configuration engine **106** determines if the family level (x) is less than the total number of families included in the MDD. If so, the configuration engine **106** proceeds to step S**406**. At step S**406**, the configuration engine **106** starts analyzing node zero. At step S**408** the configuration engine **106** compares the node number (y) to the number of features at the current family level (x) to determine if y<family (x) number of features.

[0294] After a positive determination at step S**408**, the configuration engine **106** proceeds to step S**410** and sets the currently analyzed feature (FEAT) to Family(x).Feature(y). At step S**412**, the configuration engine **106** evaluates all of the features of a family, one at a time, to determine if a feature is selected. If a feature (FEAT) is selected, the configuration engine **106** sets the state of the feature to Selected at S**414**. If the feature is not selected, the configuration engine **106** proceeds to S**416** to determine if the feature was set to Available at S**394**. If the feature is Available, the configuration engine **106** proceeds to operation S**418** and sets the state of the feature to Available. If the feature is not Selected and not Available, the configuration engine **106** sets its state to Excluded at S**420**. After steps S**414**, S**418** and S**420** the configuration engine **106** proceeds to step S**422** to increment the analyzed feature (y) by one. After evaluating each feature of family (x) to determine if it is Available, Selected or Excluded, the configuration engine **106** will make a negative determination at S**408** and then proceed to step S**424**.

[0295] At steps S**424**-S**428** the configuration engine **106** determines if family (x) has Included features. First the configuration engine **106** determines if family (x) has any Selected features at S**424**. Otherwise, the configuration determines if the family has exactly one Available feature at S**426**. If only one feature of a given family is Available, then the sole Available feature is set to Included at S**428**. After steps S**424**, S**426** and S**428** the configuration engine proceeds to step S**430** and increments the family level (x) by one, then repeats steps S**404**-S**428** for the next family level. Once the configuration engine **106** has determined the feature states for all families of the MDD, it makes a negative determination at S**404** and then returns to the main routine at S**432**.

[0296] Just as the restricted domain method S**200** can be performed as a read only step, the dynamic programming method S**300** is also read-only and thread safe when the downward and upward markers are kept externally. Because feature state calculation operation is thread safe, multiple configuration requests could be handled simultaneously using the same MDD. Each request has its own copy of external downward and upward markers, and shares the MDD. Thread safety is a property that allows code to run in multi-threaded environments by re-establishing some of the correspondences between the actual flow of control and the text of the program, by means of synchronization. For N selections, the restricted domain calculation requires N+2 MDD traversals. An advantage of the dynamic programming method S**300** over the restricted domain method S**200** is that only two MDD traversals are required regardless of the number of selections. This provides the dynamic programming method S**300** with superior scalability for large and complex MDDs.

[0297] Referring to FIG. **46**, the MDD **4600** illustrates the marked nodes after the configuration engine **106** has performed the dynamic programming method S**300** including the downward traversal subroutine of FIG. **42** and the upward traversal subroutine of FIG. **43** for a selection of Red paint and Ruby trim. The configuration engine **106** marks the truth node (T) with an upward marker at S**312** because the truth node is a special case that always has a valid upward path.

[0298] On the upward traversal of the MDD **4600**, a node is marked at S**390** if the conditions of steps S**384**-S**388** are met for the analyzed node. Regarding the moonroof level of nodes, the configuration engine **106** marks node 13 with a upward marker or arrow at S**390** because its destination node (truth node) was determined to not be a false node at S**384**; node 13's destination node (truth node) was previously marked with an upward arrow at S**386**; and the feature of the array index connecting node 13 and the truth node (i.e., MoonRf.Less) was determined to be allowed by constraint at S**388**. Since the user has not selected a moonroof feature for this example, all moonroof features are set to Allowed (see steps S**320**, S**324** of FIG. **41**.) Similarly, the configuration engine **106** determines that the remaining moonroof level nodes (nodes 14, 15 and 16) have valid upward markers.

[0299] The configuration engine **106** determines the availability of the moonroof family features by evaluating the downward markers on the source moonroof nodes at S**392** after evaluating the upward markers on the destination truth node (T) at S**386**. Since all of the moonroof nodes (13-**16**) were marked with a downward arrow and the truth node was

marked with an upward arrow, the configuration engine **106** sets all of the moonroof features to Available at S**394** and determines the initial availability bit set to be 000 000 000 000 11.

[0300] Regarding the trim level nodes (9-12), the configuration engine **106** marks node 10 with a upward marker or arrow at S**390** because its destination node (node 14) was determined to not be a false node at S**384**; node 10's destination node (node 14) was previously marked with an upward arrow at S**386**; and the feature of the array index connecting node 10 and node 14 (i.e., Trim.Ruby) was determined to be allowed by constraint at S**388**, because it was selected by the user at S**322**. Similarly, the configuration engine **106** marks node 11 with an upward marker or arrow at S**390**. However, the configuration engine **106** does not mark nodes 9 and 12 with an upward arrow at S**390** because they are not on a valid path with Ruby trim. Since the user has selected Ruby trim, the non-selected trim features (Stone and Charcoal) are set to not allowed at step S**322** (FIG. **41**) and thus the condition of step S**388** is not met for nodes 9 and 12.

[0301] The configuration engine **106** determines the availability of the trim family features by evaluating the downward markers on the source trim nodes S**392**, after evaluating the upward markers on the destination moonroof nodes (13-16) at S**386**. Trim nodes 9 and 10 each have a downward marker and a destination with a valid upward marker (i.e., moonroof nodes 13 and 14). Therefore Ruby trim along path 10-14 and Stone trim along path 9-13 are marked as Available features at S**394** because either can be on a path (in a configuration) with Red paint. However, trim nodes 11 and 12 do not have a downward marker and therefore are not marked as Available. The Trim selection can change from Ruby to Stone without requiring a change to the paint selection (Red). However, the Trim selection cannot change from Ruby to Charcoal without requiring a change to the paint (Red). Therefore the configuration engine **106** determines the Charcoal trim to be Excluded at S**420**. The configuration engine **106** determines the updated availability bit set to be 000 000 000 101 11.

[0302] Regarding the paint level nodes (6-8), the configuration engine **106** marks node 7 with an upward marker because its destination node (10) was determined to not be the false node at S**384**; its destination node (10) was previously marked with an upward arrow (S**386**); and the features of the array indexes connecting node 7 and node 10 (i.e., Paint.Red) were determined to be allowed by constraint at S**388** because it was selected by the user at S**322**. Similarly, the configuration engine **106** marks node 8 with an upward marker or arrow at S**390**. However, the configuration engine **106** does not mark node 6 with an upward arrow at S**390** because its destination node (9) was not marked with an upward arrow, and thus node 9 does not satisfy the condition of step S**386**.

[0303] The configuration engine **106** determines the availability of the paint family features by evaluating the downward markers on the source paint nodes (6-8) at S**392**, and by evaluating the upward markers on the destination trim level nodes (9-12) at S**386**. Paint nodes 7 and 8 each have a downward marker and a destination with a valid upward marker (i.e., trim nodes 10 and 11). Therefore Red paint along path 8-10 and path 7-10, and Blue paint along path 7-11 are marked as Available features at S**394**. White paint along path 6-9 is not marked as an Available feature at S**394**, because node 9 was not marked with an upward marker at S**386**. Since Red paint and Blue paint are both Available, the paint selection can be changed between Red and Blue, without requiring a change to the trim selection (Ruby). The configuration engine **106** determines the updated availability bit set to be 000 000 011 101 11.

[0304] With respect to the partial configurations of the package and radio families, all features are allowed at S**324** because the user has not selected a feature from either family. Therefore nodes 2, 4 and 5 are marked with an upward arrow. But node 3 is not marked with an upward marker at S**390** because its destination node (6) was not marked with an upward arrow, and thus node 3 does not satisfy the condition of step S**386**.

[0305] The configuration engine **106** determines the availability of the radio family by evaluating the downward markers on the source radio nodes (3-5) at S**392**, and by evaluating the upward markers on the destination paint nodes (6-8) at S**386**. Radio nodes 4 and 5 each have a downward marker and a destination with a valid upward marker (i.e., paint nodes 7 and 8). Therefore Navigation radio along path 5-8 and path 4-7, and Deluxe radio along path 5-7 and path 4-7 are marked as Available features at S**394**. The Standard radio along path 3-6 is not marked as an Available feature at S**394**, because node 6 was not marked with an upward marker at S**386**. Since the Navigation radio and the Deluxe radio are both Available, the radio selection can be changed between Navigation and Deluxe, without requiring a change to the paint selection (Red) or the trim selection (Ruby). The configuration engine **106** determines the updated availability bit set to be 000 011 011 101 11.

[0306] The configuration engine **106** determines the availability of the package family by evaluating the downward markers on the source package node (2) at S**392**, and by evaluating the upward markers on the destination radio nodes (3-5) at S**386**. The package node 2 has a downward marker and destinations with valid upward markers (i.e., radio nodes 4 and 5). Therefore the 401 package along path 2-4 and the 402 package along path 2-5 are marked as Available features at S**394**. The 400 package along path 2-3 is not marked as an Available feature at S**394**, because node 3 was not marked with an upward marker at S**386**. Since the 401 package and the 402 package are both Available, the package selection can be changed between 401 and 402, without requiring a change to the paint selection (Red) or the trim selection (Ruby). The configuration engine **106** determines the full availability bit set to be 011 011 011 101 11 using the dynamic programming method S**300**, which is consistent with its determination using the restricted domain method S**200** as described above with reference to FIG. **34** and shown in FIG. **36**.

III. App. 1—Conflict Resolution

[0307] Each time the user selects a feature, the configuration engine **106** updates the configuration by adding the new feature and removing the sibling features of the same family. Then the configuration engine **106** performs a "containsAny" operation, as described above with reference to FIGS. **20-21**, to see if the MDD contains the new configuration. If the MDD does not contain the new configuration, then the new configuration is invalid and the configuration engine **106** performs a conflict resolution strategy.

[0308] Generally, there are two types of conflict resolution strategies invoked when a user selection, or change in selection, leads to a conflict: single conflict resolution and branched conflict resolution.

[0309] In single conflict resolution, the configuration engine **106** returns the single "next-closest" valid configuration and the feature additions and subtractions necessary to change the invalid configuration to a valid configuration in its response. The "closest" valid configuration would typically be determined by simply changing the newly requested feature back to its prior state. However, such a change would be inconsistent with the user's selection. Therefore the configuration engine **106** determines the next-closest valid configuration using a constraint that the newly requested feature is "locked" and not allowed to be changed, according to one or more embodiments.

[0310] In branched conflict resolution, the configuration engine **106** presents a set of configurations to the user in a resolution tree that are closest to the invalid configuration, and the user is prompted to make changes to the configuration to get to a valid configuration. When resolving conflicts there may be multiple valid configurations all at the same distance from the initial configuration. In this case there are a set of possible answers when finding the next-closest valid configuration. An option then is to use branched conflict resolution.

[0311] A strategy that is used to determine the closeness between configurations is referred to as the "minimum edit distance." In a configurator application **104**, the configuration engine **106** determines the minimum edit distance between an invalid configuration selected by the user and one or more valid configurations. The minimum edit distance refers to the number of features in the configuration that must be changed in order to transform the invalid configuration into a valid configuration. When comparing the invalid configuration to a valid configuration, the configuration engine **106** considers substitute operations to identify what features must change to create a valid configuration without changing the newly requested locked feature.

[0312] With reference to FIG. **47**, a method for resolving conflicts between a user selected invalid configuration and one or more valid configurations is illustrated in accordance with one or more embodiments and generally referenced by S**500**. The method S**500** is implemented as an algorithm within the configuration engine **106** using software code contained within the server **102**, according to one or more embodiments. In other embodiments the software code is shared between the server **102** and the user devices **112**, **114**.

[0313] At step S**502**, the configuration engine **106** saves a copy of the root node identity and a set of the node edges to the memory **108** (shown in FIG. **2**). S**502** is similar to subroutine S**102** described above with reference to FIG. **26**. The configuration engine identifies each node (y), copies or clones each outgoing edge of each node (y) in an MDD, and then returns the edge set and the identity of the root node.

[0314] At step S**504** the configuration engine **106** performs a minimum edit calculation of an MDD that is restricted to the configurations that contain the feature selection that triggered the conflict. In one embodiment the configuration engine **106** restricts the MDD using the restrict method as described above with reference to FIGS. **22-23**. In other embodiments, the configuration engine **106** restricts the MDD using the quick-restrict method S**100** described above with reference to FIGS. **24-28**. The quick-restrict based minimum edit calculation subroutine of S**504** is shown in the flowchart of FIG. **48**.

[0315] At step S**506** the configuration engine **106** examines the cache memory for source (SRC) node (y) to determine if the quick-restrict subroutine has already been performed on node (y). If cache (y) exists, then the configuration engine **106** proceeds to step S**508** and returns to the main routine of FIG. **47**. If cache (y) does not exist, the configuration engine **106** proceeds to step S**510**.

[0316] At step S**510**, the configuration engine **106** starts with array index zero, and sets the empty flag to true. By setting the empty flag to true, the configuration engine **106** assumes that there are no valid paths from the node, i.e., that all edges point to the false node. The configuration engine **106** also sets the minimum or best (Best) edit distance as the maximum allowable integer value (Integer Max Value).

[0317] At step S**512**, the configuration engine **106** evaluates the array index (z) to determine if (z) is less than the number of node (y)'s children. A positive determination at step S**512** indicates that the configuration engine **106** has not analyzed all array indexes of node (y). If the determination is positive, the configuration engine **106** proceeds to step S**514**.

[0318] At step S**514**, the configuration engine **106** checks if y's child, or destination (DST) node, at array index (z) is false. If the configuration engine **106** determines that the child is false, then it proceeds to step S**516**, increments the array index (z) by one, and returns to step S**512**. If the determination at step S**514** is negative, the configuration engine **106** proceeds to step S**518**.

[0319] At step S**518** the configuration engine **106** initializes the edits for feature (z) to the maximum integer value, e.g., by setting the total edit distance (cacheBot (z)) for the path from the truth node to the source node to the Integer Max Value.

[0320] At step S**520**, the configuration engine **106** evaluates the quick-restricted features list for node (y) to determine if it contains feature (z). Otherwise, the configuration engine **106** sets y's child node along z to false at S**522** and then proceeds to step S**516** to increment the array index (z) by one.

[0321] At step S**524**, the configuration engine **106** sets the edit value (EDIT) for the source node at feature (z). If the configuration includes feature (z), then the configuration engine sets EDIT to zero. If the configuration does not include feature (z), then the configuration engine sets EDIT to one.

[0322] At step S**528**, the configuration engine **106** check's if y's child (DST) at array index (z) is true. If DST is true, the configuration engine **106** proceeds to step S**530** and sets the empty flag to false. At S**532**, the configuration engine **106** calculates and stores the edit distance or cost (EDIT) for this feature. Then the configuration engine **106** calculates the total edit distance (cacheBot (z)) for the path from the truth node to the source node, as the sum of a previously calculated edit distance for the path (cacheMid (DST)) and EDIT. At step S**534**, the configuration engine **106** compares the total edit distance (cacheBot(z)) to the minimum edit distance (Best) for the given invalid configuration. If the total edit distance for the current feature (cacheBot (z)) is less than Best, the configuration engine **106** proceeds to step S**536** and sets cacheBot (z) as Best. Then the configuration engine **106** proceeds to step S**516** and increments feature (z)

by one. If the configuration engine **106** determines that DST is not the true node at S**528**, it proceeds to step S**538**.

[0323] At step S**538**, y's children, or (DST) nodes, are rewritten by the result of recursive invocation of the quick-restrict method on the child. All nodes of the MDD are processed in this manner. At step S**540**, the configuration engine **106** checks DST to determine if it's not a false node. If the determination is negative, i.e., DST is the false node, then the configuration engine **106** proceeds directly to S**516** to increment the array index (z). If the determination is positive (e.g., if node (y) is connected to a valid node), then the configuration engine **106** proceeds to steps S**530**-S**536** to update the EDIT distance and compare it to the Best minimum edit distance.

[0324] The quick restrict based minimum edit calculation subroutine S**504** operates on the nodes in a depth first search fashion. Steps S**512**-S**540** demonstrate an iterative process that operates on each array index (z) of a node (y) before proceeding to the next node. Once the configuration engine **106** has evaluated all array indexes (z) for a node (y), it will make a negative determination at step S**512** (i.e., z will be greater than or equal to the number of y's children), and proceeds to step S**542**.

[0325] At step S**542** the configuration engine **106** checks if all children (DST) of node (y) are false, i.e., it evaluates the empty flag to determine if any valid configurations were found. If all of y's children are false, the configuration engine **106** proceeds to step S**544**, sets node (y) to the false node, and then returns the cache (y), i.e., saves the analysis of node (y) and returns to the main routine of FIG. **47**. Further, if the node does not contain any edges that conform to the constraint, then the edge pointer is disconnected from the child node in the MDD. If not all of the children nodes are false, then the configuration engine **106** proceeds to step S**546**, sets the cacheMid (SRC Node) to Best, and then sets feature (z) to zero.

[0326] At step S**548**, the configuration engine **106** again evaluates the array index (z) to determine if (z) is less than the number of node (y)'s children. Otherwise, the configuration engine **106** proceeds to step S**550**, sets the cache (y) to (y), and then returns the cache (y), i.e., saves the analysis of node (y) and returns to the main routine of FIG. **47**. A positive determination at step S**548** indicates that the configuration engine **106** has not analyzed all array indexes of node (y). If the determination is positive, the configuration engine **106** proceeds to step S**552**.

[0327] At step S**552**, the configuration engine **106** compares the total edit distance (cacheBot (z)) to the minimum edit distance (Best) for the given invalid configuration. If the total edit distance for the current feature (cacheBot (z)) is greater than Best, the configuration engine **106** proceeds to step S**554** and sets DST node to false at step S**554**, and then increments feature (z) by one at step S**556** and then returns to S**548**.

[0328] The quick-restrict based minimum edit calculation subroutine S**504** is a recursive process. This process continues until all nodes are analyzed, then the configuration engine **106** returns to the main routine of FIG. **47**. The edges for complete paths from the root node (node 2) to the truth node (T) define the restricted configuration space.

[0329] Thus, the configuration engine **106** calculates the minimum edit distance of all paths, and identifies the minimum edit distance using subroutine S**504**. The configuration engine **106** calculates the minimum edit distance on the way back up the MDD, i.e., during an upward traversal. When a new minimum edit distance is found, prior minimum edit paths are trimmed from the MDD by setting the minimum edit distance (Best) to the currently analyzed path (cacheBot (z)) at S**536**. The minimum edit distance of a path refers to the sum of the edit distances of each edge. The edit distance for each feature change is 1. Where there is more than one active feature on an edge, the path for each feature is considered separately when calculating the minimum edit distance. Then the configuration engine **106** restricts the MDD to the paths that have the minimum edit distance by setting child or destination (DST) nodes to false if the edit distance for their path (cacheBot (z)) is greater than the minimum edit distance (Best) at S**554**.

[0330] FIG. **49** includes an original or unrestricted MDD **4900**, an intermediate MDD **4902** and a final restricted MDD **4904** illustrating the impact of various steps of subroutine S**504**.

[0331] In one example, the user selects the 400 package. Then the configuration engine **106** determines that the Standard (Std) radio, Stone trim and no moonroof (Less) are Included features; and that White paint is a Default feature. The states of these features are represented by underlined text, boxed text and circled text, respectively in FIG. **49**. Next the user selects Charcoal trim. The new selected set (400, Charcoal) is not a valid combination because the 400 package can only occur with Stone trim. So the new configuration (400, Std, White, Charcoal, Less) is invalid.

[0332] The intermediate MDD **4902** illustrates the original MDD **4900** after the configuration engine **106** has performed the minimum edit calculation S**504** on paths 2-13- . . . T with respect to a newly selected Charcoal trim feature. The configuration engine **106** first traverses path 2-13-14-15-16-T. No restrict is needed on the downward traversal because the only active trim feature on partial path 15-16 is Charcoal.

[0333] The configuration engine **106** calculates the minimum edit distance at steps S**530**-S**536** (FIG. **48**) during an upward traversal of the MDD. The configuration engine **106** considers two edges for the partial path T-16: one edge with the moonroof (Vista) and one edge without the moonroof (Less). The configuration engine **106** determines the edit distance for the partial path T-Vista-16 (i.e., with the moonroof (Vista)) to be one because it requires a change in the moonroof feature. For the partial path T-Less-16 without the moonroof (Less), no change is required, so the edit distance is zero. Because one path (T-Less-16) has a smaller edit distance than the other (T-Vista-16), the edge is restricted to keep the minimum edit path of T-Less-16. The edge from 16-T now shows "10", and the edit cost for Node 16 is 0, as referenced by numeral **4906**.

[0334] Continuing on the upward traversal, the configuration engine **106** calculates the edit distance of the remaining partial paths to be: zero for 16-15, because it contains Charcoal trim; one for 15-14, because it contains Blue paint, not the Default White paint; one for 14-13, because it contains the Navigation radio, not the Included Standard radio; and one for 2-13, because it contains the 402 package, not the Selected 400 package. Therefore the configuration engine **106** calculates the cumulative minimum edit distance for path 2-13-14-15-16-T to be three, as referenced by numeral **4908**.

[0335] Then the configuration engine **106** restricts the partial path 14-9-10-T because node 9 does not contain the selected Charcoal trim feature.

[0336] Next, the configuration engine **106** considers path 2-13-8-11-12-T. First, partial path 8-11-12-T is considered, and found to have edit distance of 1 because node 8 contains Blue paint and not the Default White paint. Then the configuration engine **106** restricts path 12-T to 12-Less-T; and restricts path 11-12 to 11-Charcoal-12 so that the edit distance for each is zero.

[0337] Then the configuration engine **106** considers partial path 8-9-10-T; but, because of its previous analysis of node 9, it knows that this partial path has been restricted and the 8-9 edge is removed. The cost of T-12-11-8-13 is 2. At this point the configuration engine **106** keeps both of the outgoing edges from 13 (i.e., 13-14 . . . -T, and 13-8- . . . -T) because they each require 2 edits. The edit distance for partial path 13-2 is 1. Thus, after traversing 2-13- . . . -T, the configuration engine **106** determines that the current minimum edit distance for Node 2 is 3.

[0338] The final restricted MDD **4904** illustrates the original MDD **4900** after the configuration engine **106** has quick-restricted it to Charcoal trim at S**504** and performed the minimum edit calculation for all paths.

[0339] Next the configuration engine **106** considers path 2-7-8-11-12-T. The configuration engine **106** previously determined that the minimum edit distance at node 8 is 1. The configuration engine **106** calculates the edit distance for partial path 8-7 to be one because it contains Deluxe and Navigation radio features and not the Included Standard radio feature; and calculates the edit distance for partial path 7-2 to be one, because it contains the 401 package and not the selected 400 package. The configuration engine **106** calculates the total edit distance for path 2-7-8-11-12-T to be three. Since this total edit distance is the same as that of paths 2-13- . . . -T, the configuration engine **106** keeps all three paths.

[0340] Next the configuration engine **106** considers path 2-3-4-5-6-T. This path is restricted once the configuration engine **106** determines that partial path 5-6 is not compatible with the Selected Charcoal trim constraint.

[0341] The final MDD **4904** illustrates the final minimum edit space for the invalid configuration (400, Std, White, Charcoal, Less) with the Charcoal trim feature locked, after the configuration engine **106** has completed the minimum edit calculation subroutine S**504**. As shown in FIG. **49**, the MDD **4904** contains three paths that each have a minimum edit distance of three: 2-1-14-15-16-T; 2-13-8-11-12; and 2-7-8-11-12-T.

[0342] With reference to FIG. **47**, at step S**558**, the configuration engine **106** determines if the conflict resolution strategy selected for the application is a guided strategy, i.e., a branched resolution strategy. If a guided conflict resolution is not selected, the configuration engine **106** proceeds to operation S**560**. At step S**560** the configuration engine **106** selects a single target configuration from the minimum edit space using an auto-completion technique such as sequence based or maximally standard. In one embodiment, the configuration engine **106** selects a single target configuration using a maximally standard auto-completion technique as described below with reference to FIGS. **93-117**. In another embodiment, the configuration engine **106** selects a single target configuration using a maximum weight quick-restrict technique.

[0343] The maximum weight quick-restrict based single target configuration calculation subroutine of S**560** is shown in the flowchart of FIG. **50**. The weights used in the weight calculation are based on priorities for features such that when there is more than one feature available, the highest priority, or feature having the maximum weight, is chosen as the default. S**560** is similar to the minimum edit quick restrict calculation S**504** of FIG. **48**. The only difference is that when the configuration engine **106** calculates the maximum weight, it 1) maximizes value instead of minimizing value (which changes the initial value of Best); and 2) calculates the weight for each path instead of the number of edits, where the weight is defined by a priority of the features.

[0344] Referring to FIG. **47**, the configuration engine **106** resets or restores the original set of node edges to the memory **108** (shown in FIG. **2**) at step S**574**. S**574** is similar to S**146** described above with reference to FIG. **28**. At S**574** the configuration engine **106** identifies each node (y), copies the outgoing edges of each node in the MDD. Then the configuration engine **106** sets the MDD to the identity of the root node.

[0345] At step S**576**, the configuration engine **106** determines the feature states for the prior configuration. The prior configuration refers to the prior valid configuration before the configuration engine **106** changed a feature based on a user selection; where the addition of the new feature led to an invalid configuration. S**576** is similar to subroutine S**300** described above with reference to FIGS. **39-44**. At step S**578**, the configuration engine **106** determines the feature states for the new target configuration. S**578** is also similar to S**300**.

[0346] With reference to FIG. **49**, in one embodiment the configuration engine **106** selects a single target configuration (401, Dlx, Blue, Charcoal, Less) from the minimum edit space of MDD **4904** as S**560**. Once the target is identified, the configuration engine **106** determines feature states for the prior configuration at S**576** and for the new target configuration at S**578**. All features start as Default features. Next, Selected features are determined by adding the previous selections that are still valid to the newly selected feature. Then, the Available, Excluded and Available calculations are determined using the new selections. In this example the new target configuration states are: Selected (Charcoal), Included (Blue) and Default (401, Dlx, Less).

[0347] At step S**580** of FIG. **47**, the configuration engine **106** generates a response that includes the feature additions and subtractions to resolve the conflict. These features include feature state information—the new state for additions and the prior state for subtractions. The configuration engine **106** returns raw data including the new and prior feature state maps, the new configuration state (ConfigState), and the list of additions and subtractions. The author decorator **154** will use this data to compose the response. The derivation of additions and subtractions subroutine S**580** is shown in the flowchart of FIG. **51**.

[0348] At S**582**, the configuration engine **106** starts analyzing feature (z). At step S**584**, the configuration engine **106** determines if the feature (z) is less than the total number of features for the source node. If so, the configuration engine **106** proceeds to step S**586**. At step S**586**, the configuration engine **106** defines the term "Prior" as the prior feature state of feature (z) and the term "State" as the new feature state of feature (z).

[0349] At step S**588**, the configuration engine **106** evaluates the feature state term Prior (determined at S**576**) to determine if feature (z) was present in the prior configura-

tion. S588 is illustrated in the flow chart of FIG. 52. If the Prior term is equal to Selected, Included or Default, then the configuration engine 106 determines that the Prior term defines a State of a feature that is present in the Prior configuration and returns true at step S592. Otherwise, the configuration engine 106 returns false at S594. Then the configuration engine 106 sets a boolean before value (BEF) equal to the determination at steps S590-S592, i.e., true or false.

[0350] Also at step S588, the configuration engine 106 evaluates the feature state term State to determine if it is present in the New configuration. If the State term is equal to Selected, Included or Default, then the configuration engine 106 returns TRUE at step S592. Otherwise, the configuration engine 106 returns false at S594. Then the configuration engine 106 sets a boolean after value (AFT) equal to the determination at steps S590-S592, i.e., true or false.

[0351] At step S594, the configuration engine 106 compares BEF to AFT for feature (z) to see if it has changed. Otherwise, i.e. if BEF equals AFT, the configuration engine 106 proceeds to step S596 and increments the family level (x) by one. If the determination is negative, then the feature was either added or subtracted, and the configuration engine 106 proceeds to step S598 to evaluate "Return Delta."

[0352] "Return Delta" refers to the amount of information provided to the user in response to a conflict. The front-end application 132 determines this level of information by setting the return delta feature to true or false. Enabling the return delta feature (i.e., setting return delta to true) results in more information being provided to the user. Conversely, setting return delta to false results in less information being provided to the user.

[0353] When the Return Delta flag is set to true, the response will include all additions and subtractions regardless of feature state. This could allow the front-end application 132 to inspect the resolution and apply some additional logic when deciding whether to prompt the user in response to a conflict or to silently make the changes. When the Return Delta flag is set to false, the conflict resolution subtractions will only contain Selected features removed from the configuration and additions will only contain Included features added to the configuration. If the new configuration caused a change in a Default feature, this is not included in the prompt to the user. If all changes are for Default choices, there are no changes to report, and the response will not include conflict resolution. The Return Delta flag is set to false by default, in one or more embodiments.

[0354] At S598, if Return Delta is set to true, the configuration engine 106 proceeds to step S600 to evaluate BEF. If BEF is set to false at S588 (which indicates that feature (z) was contained in AFT but not in BEF), the configuration engine 106 adds feature (z) to the list of additions (additions.add(feat(x))) at step S602. If BEF is set to true (which indicates that feature (z) was contained in AFT but not in BEF), the configuration engine 106 adds feature (z) to the list of subtractions (subtractions.add(feat(x))) at step S604.

[0355] If Return Delta is set to false, the configuration engine 106 proceeds to step S606. At S606, the configuration engine 106 evaluates BEF and the state of (z). If BEF is false and the state of (z) is Default, the configuration engine 106 returns to S596 to increment the family level (x) by one. If the determination at S606 is negative, the configuration engine 106 proceeds to step S608. At S608, if BEF is true and the state of (z) is Selected, the configuration engine 106 returns to S596. Otherwise, the configuration engine 106 returns to S600.

[0356] FIG. 53 is a table 5300 that illustrates the additions and subtractions for the MDD 4900 of FIG. 49 when Return Delta is true and when Return Delta is false. The MDD 4900 had a Prior configuration of Selected (400), Included (Std, Stone, Less) and Default (White), as referenced by numeral 5302. The configuration engine 106 selected a new target configuration of Selected (Charcoal), Included (Blue) and Default (401, Dlx, Less) at step S578, as referenced by numeral 5304. When Return Delta is false, the configuration engine 106 does not show changes (additions or subtractions) to Default Features, but when return delta is true, all changes are shown, as referenced by numeral 5306.

[0357] FIG. 54 is a window 5400 that is displayed to the user to convey the additions and subtractions of features to their prior configuration to accommodate the new configuration triggered by their selection of Charcoal trim, using a single target configuration when Return Delta is false. Again, when Return Delta is false, the configuration engine 106 does not show changes (additions or subtractions) to Default features.

[0358] FIG. 55 is a window 5500 that is displayed to the user to convey the additions and subtractions of features to their prior configuration when Return Delta is true. Since Return Delta is true, the configuration engine 106 shows all changes, including changes to the Default features.

[0359] Referring to FIG. 47, the configuration engine 106 determines if guided resolution (i.e., branched conflict resolution) is selected at step S558. If so, it proceeds to step S610.

[0360] In contrast to single conflict resolution, which only presents a single next-closest configuration to the user, branched conflict resolution can present the entire minimum edit space to the user, in tree format. This allows the front-end application 132 to present the user with a series of prompts to drive towards a valid configuration.

[0361] In the minimum edit example described above with reference to FIGS. 49 and 53-55, the minimum edit space contains three superconfigurations defining four configurations. This space can be compressed to two superconfigurations as shown in FIG. 56.

[0362] FIG. 56 is a table 5600 that illustrates an example in which the paint family has just one choice (Blue) and the package and radio families each have two choices (401, 402 and Dlx, Nav), but, the choices are not dependent one each other. The choice of package (401 or 402) does not change the choice of radio (Dlx, Nav). When all the family choices are independent, the configuration engine 106 can resolve the conflicts in a single step.

[0363] The configuration engine 106 derives a resolution object that describes the actions to change the invalid configuration to a valid configuration, which is transformed by the author decorators 154 into a SOAP xml response (not shown) which can be presented to a user in a single pop up window 5700, as shown in FIG. 57.

[0364] This example illustrates the advantages offered by branched conflict resolution, as compared to single conflict resolution. The configuration engine 106 presents the user with more information about how to resolve a conflict and

asks them to choose the valid package and radio features, rather than the application silently selecting a valid configuration.

[0365] When there are no dependent choices, it is a relatively simple process to transform the target matrix into the conflict resolution response. However, the resolution object is more complicated when one family's choice is dependent on another family.

[0366] The configuration engine **106** makes a series of prompts or inquiries to the user when the guided choices are dependent on a prior choice. However, each prompt can include more than one independent choice. When there is a nested or dependent choice, the configuration engine **106** lists the divergent branch as the last feature group in the list.

[0367] In branched conflict resolution, the minimum edit space may include some partial matches, where a choice is present in some, but not all of the target configurations. Partial matches in the configuration space can cause the target space to be too large, and it can lead to awkward user prompts. In one or more embodiments, the configuration engine **106** trims the minimum edit space to remove partial matches.

[0368] With reference to FIG. **47**, at step S**610**, the configuration engine **106** determines if removing partial matches functionality is enabled. If the determination at S**610** is positive, the configuration engine **106** proceeds to S**612**. The algorithm for removing partial matches S**612** is shown in the flowchart of FIG. **58**.

[0369] With reference to FIG. **58**, at step S**614** the configuration engine **106** creates a weight object, with positive weights for features that are contained within the configuration, and zero weight for features that are not contained within the configuration.

[0370] At step S**616**, the configuration engine **106** creates cache objects. The configuration engine **106** returns data from cache for node (Y) for weightFeature and for weightNode, if it already exists. WeightFeature refers to the weight from the node to the bottom of the MDD (i.e., the true node)—for a specified feature. WeightNode is a Node-based lookup instead of feature based lookup. Thus, for each node, the configuration engine **106** determines the maximum weight path feature for a given node, stores the weight for each feature. Once the configuration engine **106** knows the maximum feature weight for a node and trims its edges, then it stores the maximum node weight. The maximum node weight is used as a cache for when the configuration engine **106** analyzes subsequent nodes. It can use the previously calculated maximum weight for that node and trim the outgoing edges from the node to only those edges with the maximum weight.

[0371] At step S**618**, the configuration engine **106** performs the maximum weight quick-restrict operation. The maximum weight quick-restrict operation of S**618** is the same as the maximum weight quick-restrict subroutine S**560** described above with reference to FIG. **50**. After the quick restrict operation has removed partial matches, the configuration engine **106** proceeds to S**620** (FIG. **47**) to derive a resolution object.

[0372] Referring back to FIG. **47**, if the configuration engine **106** determines that removing partial matches functionality is not enabled at S**610**, it proceeds to S**620** to derive a resolution object. The algorithm for deriving a resolution object S**620** is shown in the flowcharts of FIGS. **59-63**.

[0373] Referring to FIG. **59**, the configuration engine **106** identifies families to change at step S**622**. The algorithm for identifying families to change S**622** is shown in the flowchart of FIG. **60**. Referring to FIG. **60**, the configuration engine creates a bitset of the invalid configuration at step S**624**. Next, the configuration engine calculates the domain bitset of the minimum edit space at step S**626**. Then at step S**628**, the configuration engine ANDs the configuration bitset and the domain bitset.

[0374] FIGS. **64-74** are examples illustrating the impact of various steps of the derive resolution object subroutine S**620** of the branched conflict resolution method. Referring to FIG. **64**, in one example, a configuration of Default (400, Std, White, Stone, Less) is modified when the user selects the 401 package. The configuration engine **106** determines the new invalid configuration to be (401, Std, White, Stone, Less) with a minimum edit space depicted by MDD **6400**.

[0375] FIG. **65** is a table **6500** listing the invalid configuration (401, Std, White, Stone, Less) as a bitset (010 100 100 100 10). FIG. **66** is a table **6600** that depicts the minimum edit space of the configuration converted to a matrix, as determined in step S**626** (FIG. **60**).

[0376] Referring back to FIG. **60**, the configuration engine **106** starts analyzing the nodes included in family level zero (x) of the MDD at step S**630**. At step S**632**, the configuration engine **106** determines if the family level (x) is less than the total number of families included in the MDD. If so, the configuration engine **106** proceeds to step S**634**.

[0377] At S**634**, the configuration engine **106** evaluates the number of active features in the intersection, or bitwise conjunction, or "ANDed" bitset for family (x). If there is at least one active feature in the ANDed bitset for family(x), the configuration engine **106** proceeds to step S**636** and increments the family level (x) by one. However, if there are no active features in the ANDed bitset for family(x), the configuration engine **106** proceeds to step S**638** and adds family(x) to the list of families to change, and then proceeds to step S**636**.

[0378] FIG. **67** is a table **6700** illustrating the bitwise conjunction (AND) of the domain of the edit space in the example from table **6600** with the invalid configuration from table **6500**. The configuration engine **106** identifies any family with no active bits (i.e., only "0"s) in the sum, as referenced by numeral **6702**, as a family to change to transform the invalid configuration into a valid one. Thus, the configuration engine **106** determines that the radio, paint and trim families are families to change.

[0379] With reference to FIG. **60**, once the configuration engine **106** has analyzed all families, it will make a negative determination at step S**632**, i.e., the level (x) will not be less than the number of families. Then the configuration engine **106** proceeds to step S**640** and returns to the routine of FIG. **59**.

[0380] Referring to FIG. **59**, at step S**642** the configuration engine **106** trims the edit space to the families identified in S**622**. Then at step S**644**, the configuration engine **106** converts the trimmed space to a matrix, which is represented by table **6600** (FIG. **66**).

[0381] FIG. **68** is a table **6800** that illustrates the minimum edit space from table **6600** after it is trimmed to the families to change (i.e., radio, paint and trim) from table **6700**.

[0382] At step S**646** (FIG. **59**), the configuration engine **106** creates a resolution object that describes the actions to

change the invalid configuration to a valid configuration. The algorithm for creating a resolution object S**646** is shown in the flowchart of FIG. **61**.

[0383] In creating a resolution object S**646**, the configuration engine takes as input the invalid configuration, a target matrix (i.e., the edit space), the list of families to change and the list of families that have been processed so far. The list of families to change is sorted in priority order, according to one or more embodiments. This sort can be provided by an alternate sequence, or default to the minimum edit space family structure shown in FIG. **68**.

[0384] The configuration engine **106** divides the families to change into No-Branch and Branch lists. The configuration engine **106** identifies any family that is not identical in all rows of the target matrix as a Branch family. Each resolution contains a list of actions for any number of No-Branch families and a single Branch family.

[0385] An action specifies the family to change, its prior choice, and its possible choice(s). To simplify parsing of a resolution, the configuration engine **106** organizes the Branch action to be the last action. The Branch items lead to a divergent choice and the choices for the remaining families are unknown until the Branch family choice is made. Each choice of the Branch family will have an associated nested resolution object defined in a feature choice that results in resolution mapping. The nested resolution objects are derived by with a new target matrix that is restricted to the feature choice of the branching family, the original families list, and the updated processed families list.

[0386] Referring to FIG. **61**, step S**648** is a recursive call in which the configuration engine **106** sets the families to consider changing (Families to Consider) equal to the families list minus the families that were already processed (Families Processed).

[0387] At step S**650**, the configuration engine **106** divides the families to consider changing (Families to Consider) into Branch and No-Branch lists. A Branch family is a family whose bit pattern is not identical in all rows of the target matrix, whereas a No-Branch family's bit pattern is identical in all rows. The algorithm for dividing families into Branch and No-Branch S**650** is shown in the flowchart of FIG. **62**.

[0388] With reference to FIG. **62**, the configuration engine **106** derives a unique bit pattern for each family based on the target matrix at S**652**. The configuration engine **106** starts analyzing the nodes included in family level zero of the MDD at step S**654**. At step S**655**, the configuration engine **106** determines if the family level (x) is less than the total number of families to consider changing in the MDD. If so, the configuration engine **106** proceeds to step S**656**.

[0389] At step S**656** the configuration engine **106** evaluates the number of unique patterns for family (x). If there is more than one unique pattern, the configuration engine **106** proceeds to step S**658** and adds the family to the Branch category. If there is only one unique pattern, the configuration engine **106** proceeds to step S**660** and adds family level (x) to the No-Branch list. After steps S**658** and S**660**, the configuration engine **106** increments the family level (x) by one and then returns to S**655**. Once the configuration engine **106** has organized all families into the Branch and No-Branch lists, it makes a negative determination at S**655**, and then sorts the families within each list by priority at step S**662**, with the highest priority family listed first. The priority of each family is based on its family weight and is determined by a separate system, according to one or more embodiments. After S**662**, the configuration engine **106** returns to the create resolution object subroutine S**646** of FIG. **61**.

[0390] Referring back to FIG. **68**, the first call to derive the resolution method object S**620** will use the invalid configuration from FIG. **65**, the target matrix from FIG. **68**, and will pass an empty list for the processed families list—deriveResolution (Cfg65, Matrix68, [Radio, Paint, Trim], [ ]). As shown in table **6800**, the bit pattern for radio (i.e., "011") is the same in both rows, but is different for both paint and trim. Therefore the configuration engine **106** identifies radio as a No-Branch family at S**656** and S**660**, because it only has one unique pattern. And the configuration engine **106** identifies paint and trim as Branch families at S**656** and S**658**, because they each have more than one unique pattern.

[0391] Referring back to FIG. **61**, the configuration engine **106** initializes a resolution object at S**664** by creating an empty object. The configuration engine **106** starts analyzing the first No-Branch family (b=0) at step S**666**. At step S**668**, the configuration engine **106** determines if the No-Branch family index (b) is less than the total number of No-Branch families. If so, the configuration engine **106** proceeds to S**670**. At step S**670**, the configuration engine **106** creates an action for No-Branch family index (b) and adds the action to the Resolution Object. The algorithm for S**670** is shown in the flowchart of FIG. **63**.

[0392] With reference to FIG. **63**, the configuration engine **106** initializes an empty Action object at step S**672**. Next the configuration engine **106** sets a current feature (z) equal to the family feature (Family) in the invalid configuration at step S**674**.

[0393] At step S**676**, the configuration engine **106** sets the group of valid features for the user to choose from (Choices) equal to the active features for the No-Branch family index (b) in the target matrix domain. Then at step S**678**, the configuration engine **106** returns the Action object to the subroutine S**646** of FIG. **61**. With reference to FIG. **61**, the configuration engine **106** adds the No-Branch family index (b) to the families processed list at step S**679**, then it increments the No-Branch family index (b) by one and then returns to step S**668**.

[0394] Referring to FIGS. **65-71**, the configuration engine **106** initializes an empty resolution and adds an action for all No-Branch families, which in this case is radio—the only No-Branch family identified from table **6800**. As shown in table **6500**, the invalid configuration includes the Standard radio (i.e., "100") and the trimmed minimum edit space of table **6800** shows the possible valid choices for radio include Deluxe and Navigation (i.e., "011"). The Action for radio in this example specifies the family to change (i.e., radio), its prior choice (Standard), and its possible choices (Deluxe and Navigation), which may be represented by: Action {Radio, [Std], [Dlx, Nav]} as shown in FIG. **71**.

[0395] Referring back to FIG. **61**, once the configuration engine **106** has analyzed all of the No-Branch families, it will make a negative determination at step S**668** (i.e., the No-Branch family index (b) is greater than or equal to the number of No-Branch families) and then proceed to step S**680**. At S**680** the configuration engine **106** evaluates the number of Branch families. If there are zero Branch families, the configuration engine **106** returns to the derive resolution subroutine S**620** of FIG. **59**. If there are Branch families (i.e., Branch families >0), the configuration engine **106** proceeds to step S**682**.

[0396] At S**682** the configuration engine **106** sets Family equal to the first family (i.e., the highest priority family) in the Branch list. At step S**684** the configuration engine **106** creates Action for the Family. The configuration engine **106** initializes an empty Action Object and sets the current feature equal to the Family configuration feature. Next the configuration engine updates the Action to set Choices equal to the active features for Family in the target matrix domain. At S**688**, the configuration engine **106** creates a new Families Processed list by appending Family to the Families Processed list.

[0397] At step S**690**, the configuration engine **106** starts analyzing Choice (c=0). At step S**692**, the configuration engine **106** compares Choice (c) to the number of choices. If Choice (c) is less than the number of choices, then the configuration engine **106** proceeds to step S**694** and creates a new target matrix that is restricted to Choice (c).

[0398] At step S**696**, the configuration engine **106** creates a Resolution Object for Choice (c) using the new target matrix and the new Families Processed list by the result of recursive invocation. At step S**698**, the configuration engine **106** updates the Action to set Branch for Choice (c) equal to the nested Resolution Object. At step S**700**, the configuration engine **106** increments Choice (c) by one and returns to step S**692**. Once the configuration engine **106** has analyzed all Choices, it determines that Choice (c) is equal to the total number of choices at S**692** (i.e., C is not less than # Choices). In response to a negative determination at S**692**, the configuration engine **106** proceeds to step S**702** and adds the Action object to the Resolution. Then it returns to subroutine S**620** (FIG. **59**) and then back to the main conflict resolution routine S**500** of FIG. **47**.

[0399] With reference to FIGS. **65-71**, the configuration engine **106** generates the Action for the highest priority Branch family, which in this case is the Paint family because it defaulted to using MDD/Matrix structure/family ordering where the right to left ordering defines priority values. As shown in table **6700**, paint oriented to the left of trim, so paint is higher priority. For each paint Choice, the configuration engine derives a nested resolution by first creating a new target matrix by restricting to the paint choice, and making a call to derive resolution.

[0400] As shown in the trimmed minimum edit space of table **6800**, there are two possible Choices for the paint family—Red (i.e., "010") and Blue ("001"). There will be two additional calls to derive resolution: one for Red paint—deriveResolution(Cfg65, Matrix.Red, [Radio, Paint, Trim], [Radio, Paint]); and one for Blue paint—deriveResolution (Cfg65, Matrix.Blue, [Radio, Paint, Trim], [Radio, Paint]). As shown in FIG. **71**, the Action for paint in this example specifies the family to change (i.e., paint), its prior choice (White, i.e., "100" in table **6500**), and its possible choices (Red and Blue), which may be represented by: Action {Paint, [White], [Red, Blue]}.

[0401] In both cases (Red paint and Blue paint), the configuration engine **106** uses a trimmed target matrix and the families list contains only one family (trim) because there is only one other Branch family in this example. FIG. **69** is a target matrix **6900** for the Red paint choice. The target matrix **6900** shows that there is just one choice for trim (i.e., Ruby, "001"). FIG. **70** is a target matrix **7000** for the Blue paint choice. The target matrix **7000** shows that there are two choices for trim (i.e., Charcoal and Ruby, "011"). The configuration engine **106** determines the resulting resolution of each target matrix **6900** and **7000** and adds the nested resolutions for paint to its action to determine a final resolution object **7100** (FIG. **71**).

[0402] As shown in FIG. **71**, the Action for the nested resolution for Red paint in this example specifies the family to change (i.e., trim), its prior choice (Stone, i.e., "100" in table **6500**), and its possible choice (Ruby), which is represented by: Action {Trim, [Stone], [Ruby] }. The Action for the nested resolution for Blue paint in this example specifies the family to change (i.e., trim), its prior choice (Stone, i.e., "100" in table **6500**), and its possible choices (Charcoal and Ruby), which is represented by: Action {Trim, [Stone], [Charcoal, Ruby]}.

[0403] Referring back to FIG. **47**, at S**704**, the configuration engine **106** restores the original set of node edges to the memory **108** (shown in FIG. **2**). S**704** is similar to subroutine S**146** of FIG. **28**. The configuration engine identifies each node (y), copies each outgoing edge of each node. Then, the configuration engine **106** sets the MDD to the identity of the root node. At step S**706**, the configuration engine **106** returns a response by providing the resolution object **7100** to the author decorator **154**; who in turn transforms the resolution object **7100** into a SOAP xml response (FIG. **2**) which is presented to the user in a series of windows, as shown in FIGS. **72-74**.

[0404] As described above with reference to resolution object **7100** (FIG. **71**), the configuration engine **106** determined a guided resolution that includes a nested choice for trim. The Available choices for trim depend on the user's selection for paint. FIG. **72** depicts a window **7200** with a first prompt that is displayed to the user. FIG. **73** and FIG. **74** show the next prompt displayed as determined by the choice of Red or Blue paint.

[0405] If the user selects Red paint in response to the prompt shown in window **7200**, the configuration engine **106** will guide them along partial path 8-9-10 (FIG. **64**) and then display window **7300** to instruct them to change the trim from Stone to Ruby.

[0406] However, if the user selects Blue paint in response to the prompt shown in window **7200**, the configuration engine **106** will guide them along partial path 8-11-12 (FIG. **64**) and then display window **7400** to instruct them to change the trim from Stone to one of Charcoal and Ruby.

[0407] As described above with reference to FIG. **58**, in branched conflict resolution the minimum edit space may include some partial matches, where a choice is present in some, but not all of the target configurations. Partial matches in the configuration space can cause the target space to be too large, and it can lead to awkward user prompts. The configuration engine **106** may trim the minimum edit space using the removing partial matches subroutine S**612**. This is done using the maximum weight quick-restrict operation, according to one or more embodiments. However, only the partial match features are provided with relative non-zero weights; all other features are given an equal weight of zero at S**614**.

[0408] FIGS. **75-76** are examples illustrating the impact of various steps of the remove partial matches subroutine S**612**. In the illustrated embodiment, a selection of D2 leads to the invalid configuration (A1, B2, C3, D2, E3, F1) with the minimum edit space shown in MDD **7500** of FIG. **75**. There are two partial matches—B2 (i.e., the outgoing edge "010" from node 12) and E3 (i.e., the outgoing edge "001" from node 11)—where a user selection could remain the same or

be changed. If B2 remains unchanged, then E3 must change to E2, because E3 is not located along the same path as B2 (i.e., partial path 11-12-13-14). But, E3 can remain unchanged if B2 changes to B3, because E3 is located on the same path as B3 (i.e., partial path 11-15-16-14).

[0409] With an alternate sequence {A1 A2 B1 B2 B3 D1 D2 C1 C2 C3 C4 E1 E2 E3 F1 F2}, and an invalid configuration of {D2 A1 E3 B2 C3 F1}, the structure used for path weights is {A1 B2 C3 D2 E3 F1}. The maximum weight operation will ignore any edges with negative weights (A2 B1 C1 C2 C4 D1 E1 E2 F2). The weights for the two paths in the minimum edit space are shown in Table **7600** of FIG. **76**. The first row shows path 2-10-11-12-13-14-1. This path defines the configuration of {D2 A1 E2 B2 C2 F2} and a path weight bit set of 010100. There are two active bits in the path weight which correspond to the features B2 and D2—the two features on this path with non-negative weights. There are no active bits for the other features (A1, C2, E2, F2) because they have negative weights and are ignored. The second row shows path 2-10-11-15-16-13-1. This path defines the configuration of {D2 A1 E3 B3 C2 F2} and a path weight bit set of 000110. There are two active bits in the path weight which correspond to the features D2 and E3. Based on these weights, the configuration engine **106** trims the space to a single path of 2-10-11-12-13-14-T. The higher priority family B will remain unchanged, and the resolution will include a change for family E.

[0410] When an update request is made, the configuration engine **106** modifies the prior configuration by adding the newly Selected feature (while also removing the prior choice for the feature's family). As described above with reference to FIGS. **47** and **48**, if the updated configuration is invalid, conflict resolution is triggered and the configuration engine **106** calculates the minimum edit space at S**504**. The service API **134** does not dictate how the minimum edit space is calculated. The configuration engine **106** determines the level of precedence to be given to prior Selected features.

[0411] In one embodiment, the configuration engine **106** treats all features in the invalid configuration equally, regardless of feature state. In this instance the minimum edit calculation S**504** is performed using the full invalid configuration.

[0412] In another embodiment, the configuration engine **106** gives precedence to keeping the maximum number of previously selected features. The configuration engine **106** performs this strategy by performing the minimum edit calculation S**504** using a partial configuration that ignores any feature whose prior state is Included or Default. Only the new and previous Selected features are kept when making the minimum edit space calculation. In one embodiment the configuration engine **106** performs the minimum edit calculation S**504** after determining whether or not the resolution is guided at S**558**. If the resolution is not guided, the configuration engine **106** performs the minimum edit space calculation S**504** using only the Selected features. However, if the resolution is guided, the configuration engine **106** performs the minimum edit space calculation S**504** using the full invalid configuration.

[0413] FIGS. **77-81** are examples illustrating the impact of various steps of the minimum edit space calculation S**504** using a full configuration and using a partial configuration. In the illustrated embodiments, the configuration engine **106** analyzes the full buildable space shown in Table **7700** of FIG. **77**, where the features are numerically named—e.g. Family E has two features E1 and E2. If the previous configuration is Selected (F1, E2) and Included (A2, T1, R1) and Default (S5, P1, Y3) and an update request is received to select T2. The configuration engine **106** determines that the updated configuration is invalid because (F1, E2, T2) is not a valid combination. As shown in Table **7700**, row 1 is the only configuration that includes both F1 and E2, but it includes T1 not T2.

[0414] The two possible minimum edit spaces are shown in Table **7800** of FIG. **78** and Table **7900** of FIG. **79**. The first minimum edit space (**7800**) considers the partial configuration where only the new set of selected features is used in the minimum edit space calculation (F1, E2, T2) and the second minimum edit space (**7900**) considers the full configuration where the complete invalid configuration is used in the minimum edit space calculation (F1, E2, A2, T2, R1, S5, P1, Y3).

[0415] Using the matrix structure as the priority sequence, the configuration engine **106** identifies a single target configuration from each minimum edit space. The priority sequence is based on the matrix as is with left most bit being highest priority. So in this case, the configuration engine **106** selects a configuration by choosing features for each family in a left-to right fashion, and choosing the left-most available feature for each family. With reference to FIG. **78**, since the E and F families are the same for all configurations; and the configuration in the bottom row has the highest priority feature for family Y, the first space will result in a target of E1, F2, Y1, T2, R3, P1, S1, A2 which corresponds to the bottom row of Table **7800**. The second space will result in a target of E1, F2, Y3, T2, R1, P1, S5, A2. The decision on how to calculate the minimum edit space will affect the target configuration, and thus affects the number of feature edits required.

[0416] Table **8000** of FIG. **80** shows the changes required (i.e., the shaded features in target 1) when the configuration engine **106** makes the minimum edit space calculation with only the selected features, based on the minimum edit space in Table **7800**. As shown in Table **8000**, the features to change are: E1, F2, Y1, R3 and S1.

[0417] Table **8100** of FIG. **81** shows the changes required (i.e., the shaded features in target 2) when the minimum edit space calculation is made with the full invalid configuration, based on the minimum edit space in Table **7900**. As shown in Table **8000**, the features to change are: E1 and F2. In both cases the changes to families E and F are the same because T2 is only available with E1 and F2. But, if the minimum edit space calculation considers only the Selected features, there are three other required changes (i.e., Y1, R3 and S1, as shown in Table **8000**).

[0418] When the configuration engine **106** performs the calculation with the full configuration, it minimizes the total edits, as shown in Table **8100**. It gives no special precedence to previous Selected features.

[0419] When the configuration engine **106** performs the calculation with only the new selected set, it is attempting to minimize the changes to prior selections by giving precedence to Selected features. The side effect is that this increases the total number of edits required, as shown in Table **8000**.

[0420] The other motivation to perform the minimum edit space calculation on only Selected features is to ensure that the maximally standard default is always driven by the Selected set.

[0421] The configuration engine **106** results include a Boolean flag to indicate if there is a conflict. When there is a conflict, the configuration engine **106** determines either a single conflict resolution object or a branched conflict resolution object, depending on the guided resolution flag.

[0422] Because the services API **134** dictates that a conflict resolution is returned only if there are changes required to the previous selected features, it is possible that the same configuration request will return conflict=true when guided=true, but will return conflict=false when guided=false.

[0423] Using the product definition from FIG. **10**, in one example the configuration engine **106** considers a configuration where the user has selected Red paint, and autocomplete is true. One such configuration is Selected (Red) and Default (400, Std, Stone, Less). If the user updates the configuration by selecting the navigation (Nav) radio, the new configuration (Nav, Red, 400, Stone, Less) is invalid.

[0424] When guided resolution is true, the configuration engine **106** returns a conflict of false and a returns resolution such as {Actions [Action {Pkg, [400], [401,402] }, Action {Trim, [Stone], [Ruby]}]}.

[0425] However, when guided resolution is false, the API dictates that there is no conflict because the new Selected features (Nav, Red) are compatible. Even though changes are required for Pkg and Trim, because these were default choices, no conflict is reported. The configuration engine **106** will return conflict of false even though the new configuration and associated feature states show that there is a necessary change to prior default choices for the 400 package, and Stone trim—{Std=AVAILABLE, Nav=SELECTED, Stone=EXCLUDED, White=EXCLUDED, Charcoal=EXCLUDED, Vista=AVAILABLE, Dlx=AVAILABLE, Red=SELECTED, 400=EXCLUDED, 401=DEFAULT, 402=AVAILABLE, Less=DEFAULT, Blue=AVAILABLE, Ruby=INCLUDED}.

[0426] The service API **134** specifies that a request can select or unselect a feature. Selecting a feature adds it to the configuration. When a feature is unselected, the API dictates only that the feature state is no longer Selected. It does not require that the feature be removed from the configuration.

[0427] There are at least two embodiments. In a first embodiment, the configuration engine **106** removes the unselected feature from the Selected set and proceeds normally. In a second embodiment, the configuration engine **106** removes the unselected feature from the configuration entirely.

[0428] Removing the unselected feature from the Selected set follows the API specification that the feature is no longer Selected. However, the feature may not actually be removed from the configuration. In a configurator application, this could mean that the user unchecks the box to remove the feature only to have the checkbox checked again because it is not removed from the configuration. This behavior can be difficult because no matter what the user does to try and remove the feature, the feature keeps getting added back.

[0429] In this first embodiment, if a Default or Included feature is unselected, there will be no change in the configuration or feature states. The Selected set does not change because the feature being unselected wasn't in the Selected set. As such, an Included feature will remain Included and a default will remain default. Included state depends on the Selected set which did not change and auto completion is designed to give repeatable results for the same minimally complete configuration which did not change.

[0430] If a Selected feature is unselected, the feature state may change to Included or Default. If the feature is Included by the remaining Selected features, its state will change from Selected to Included. Otherwise, it is possible that the removed feature is added back during auto completion.

[0431] Depending on the front-end application **132**, the user is most likely unaware of the feature states of Selected, Included and Default. If the user is unaware of feature states, it can be quite perplexing to unselect a feature only to have it remain in the configuration.

[0432] To avoid this confusion, in the second embodiment, the configuration engine **106** removes the feature from the configuration regardless of its state. This implementation will trigger conflict resolution when a feature from the configuration is unselected. To ensure the unselected feature remains absent from the new configuration, the configuration engine **106** restricts the buildable space to only those configurations that do not contain the unselected feature prior to the minimum edit calculation. This approach ensures that the unselected feature is removed from the configuration.

IV. App. 4—Maximum Standard

[0433] When auto completion is enabled, the configuration engine **106** makes Default choices until the configuration is complete with respect to displayable families. This is done by making determinations for each incomplete family that is consistent with prior Selected and Included feature states.

[0434] In one embodiment, the configuration engine **106** makes Default feature state determinations based on a priority sequence for each family and feature. Incomplete families are processed in priority order, and where more than one feature is valid with prior Selected, Included and Default feature states, the feature priority is used to make the Default determination.

[0435] With reference to FIG. **82**, a method for automatically completing a configuration using a sequence-based approach is illustrated in accordance with one or more embodiments and generally referenced by S**750**. The method S**750** is implemented as an algorithm within the configuration engine **106** using software code contained within the server **102**, according to one or more embodiments. In other embodiments the software code is shared between the server **102** and the user devices **112**, **114**. The method S**750** uses quick-restrict and domain to make default selections.

[0436] At S**752**, the configuration engine **106** starts with an MDD restricted to Selected features, and a configuration defined by the Selected and Included features. At S**754**, the configuration engine **106** sorts families by priority. Then, at S**756**, for each family without Selected or Included features, that are sorted by priority, the configuration engine **106** calculates the domain of the current restricted space. At S**758**, the configuration engine determines the highest priority feature for the family, marks it as a Default feature, and further restricts the restricted space to this feature.

[0437] Alternatively, in another embodiment, after S**752**, the configuration engine **106** proceeds to S**760** and uses sequences to determine the maximally weighted path in the MDD that contains the Selected and Included features, which identifies default choices for families to complete.

[0438] Path weight can be represented by a feature Bitset, where the families are ordered left to right according to priority (with the leftmost family being the highest priority), and features are also ordered left to right according to priority within each family (with the feature corresponding to the leftmost bit being the highest priority). This is analogous to sorting the features in descending priority and assigning weights by powers of 2, -1 2 4 8 16 . . . , and ensures that each path will be uniquely weighted, and that there will be exactly one maximally weighted path when each family and feature is assigned a sequence. To compare two paths, the Bitsets are sorted in descending bit order.

[0439] Initially the path weight is 0, and as an edge is added during MDD traversal, edge weight is added to the path by simply setting a corresponding bit associated with the feature on that edge.

[0440] Ideally, the MDD structure would reflect the priority sequence for features and families. However, compression of the MDD requires the family ordering to be determined by the compression algorithm, and the feature order is arbitrarily determined by feature conditions of the MDD algorithm. While the MDD structure can't be used to store priority sequence, a secondary structure can be created to store this sequencing. The MDD structure is used to determine the feature for each edge, and the alternate/secondary structure is used to set the feature's bits in the path weight Bitset.

[0441] FIG. **83** shows an MDD **8300** that defines the same buildable space as MDD **1300** (FIG. **13**), except that the features within each family are sorted alphabetically and the family order is based on MDD compression. For example, the radio features are sorted as STD, DLX and NAV in MDD **1300** and as DLX, NAV and STD in MDD **8300**. FIG. **84** is a table **8400** that defines the alternate sequence structure defining path weights, which is the same as the structure of MDD **1300**.

[0442] Referring back to FIG. **82**, the configuration engine **106** begins the weighted operation with a minimally completed configuration, and starts with a quick-restrict operation (S**752**). In another embodiment, the configuration engine **106** combines the quick-restrict operation with the weighted operation (S**760**), as is done in Minimum Edit Distance subroutine described with reference to FIG. **48** regarding resolving configuration conflicts. On the downward traversal, no additional action is taken. On the upward traversal, the configuration engine **106** calculates the path weight with the aid of the alternate sequence structure **8400**. Where a node has more than one outgoing edge, the weight is calculated separately for each feature that has a valid outgoing edge. When a new maximally weighted path is found, lesser weighted paths are trimmed from the MDD.

[0443] FIG. **85** illustrates an MDD **8500** after the configuration engine **106** restricts the MDD **8300** (FIG. **83**) to a selection of Blue paint and no Included or Default features, i.e., a minimally complete configuration: Selected{Blue}+Included{ }+Default{ }, as described with reference to S**752**. Then configuration engine **106** starts the weighted operation, i.e., S**760**, with the quick-restricted space shown in MDD **8500**. Using depth-first search and traversing child edges in descending bit order, the first path traversed is 2-8-9-10-6-T.

[0444] The configuration engine **106** calculates path weight on the upward traversal beginning with partial path T-6-10-9. This path weight, along with the weight of its partial paths, is shown in Table **8600** of FIG. **86**.

[0445] The configuration engine **106** continues the downward traversal from Node 9 with the 401 package (Pkg.401). There are two partial paths to consider: 9-5-7-T and 9-5-6-T. The two partial paths from Node 5 to the truth node are compared to find the maximum weight between Ruby and Charcoal Trim, as shown in Table **8700** (FIG. **87**). The configuration engine **106** determines that Charcoal Trim (path 9-5-6-T) is the maximum, because 000 000 001 010 00 is greater than 000 000 001 001 00, as referenced by numeral **8702**, and the edge 5-7 (Ruby Trim) is removed (shown in FIG. **89**).

[0446] Next, the configuration engine **106** compares the paths from Node 9 to the Truth node, as shown in Table **8800** (FIG. **88**). The configuration engine **106** determines that the partial path for the 401 Package is the maximum because the corresponding bits (010) are greater than the corresponding bits of the 402 package (i.e., 001), as referenced by numeral **8802**; and trims edge 9-10 from the MDD **8500**. At this point the MDD will look like MDD **8900**, as shown in FIG. **89**.

[0447] The maximum weight path for edge 8-9 is MoonRf.Less, because the corresponding bits for Less (10) are greater than the corresponding bits for Vista (01). The configuration engine **106** determines that the configuration {Dlx,Less,401,Charcoal,Blue} is the maximum weight for path 2-8- . . . -T and that the maximum weight configuration for path 2-3- . . . -T is {Std,Less,401,Charcoal,Blue}. The edge weights for these paths are shown in Table **9000** in FIG. **90**, and the maximum weight outgoing edge from node 2 is Dlx, because 010 is greater than 001 (Nav), as referenced by numeral **9002**.

[0448] The configuration engine **106** trims edges 2-8-9-5, 10-6 and 7-T as shown in MDD **8900** (FIG. **89**). This leaves the maximum weight path as 2-3-4-5-6-T, and the configuration engine **106** marks Radio.Dlx, MoonRf.Less, Pkg.401 and Trim.Charcoal as Default features at S**760** to generate an autocompleted configuration of: Selected {Blue}+Included{ }+Default{Dlx,Less,401,Charcoal}.

[0449] A problem with sequence based auto completion is that it requires additional data to define the priority sequence. This requires manual setup for every vehicle in the catalog. Furthermore, it is not guaranteed to provide the maximally standard configuration.

[0450] Sequence-based auto completion is attempting to supplement the MDD with data that will allow the autocompleted configuration to be maximally standard. A "maximally standard" configuration is one where a configuration contains the most possibly standard content, where standard content is determined by the product definition. A standard feature is generally a feature that is included in the base price for a product, such as a vehicle. Whereas an optional feature is typically an upgrade and will increase the price of the product.

[0451] For simpler product definition, it is may be possible to ensure maximally standard configurations using the alternate sequence approach. However, for more complex product definition, this approach will not work.

[0452] The product definition defines a set of feature conditions that determine when a feature is available. There are three types of availability—as a standard feature, as an included feature, and as an optional feature. A feature could have different availability depending on the current configuration. A standard feature is a feature that is included in the base product (vehicle) configuration and an optional feature is a feature that is not included in the base product. An optional feature is typically an upgrade that will add cost to the base price, but this is not always the case as a feature could be a zero-cost option or even a less expensive option than the standard feature. An included feature is generally a feature that is associated with another feature (e.g., a package) and was added to the product by selecting the other feature. For example, leather seats and fuzzy dice may be included in the 401 package. When the user selects the 401 package, they see one change in price for the package, but the change includes both features (i.e., leather seats and fuzzy dice).

[0453] The configuration engine **106** uses the maximally standard algorithm to distinguish feature availability based on standard or optional feature conditions. A feature included in a package is a special case of a standard feature. In addition to the valid buildable space, the product definition also defines the standard feature conditions for each family. When the configuration engine **106** selects Default features using the maximally standard algorithm, it is configured to select as many standard features as possible to avoid or limit adding optional content and potentially increasing the price when the user has not explicitly selected the feature.

[0454] FIG. **91** is a Table **9100** that shows a matrix that defines the same buildable space as MDD **1700** (FIG. **17**). FIG. **92** is a Table **9200** that shows the standard feature conditions for the product definition defining the buildable space.

[0455] For the package (Pkg) family, there is a single feature condition defining Pkg.400 as the standard package, as referenced by numeral **9202**. For the Radio family, there are two standard feature conditions, as referenced by numeral **9204**. The first (upper row) defines Radio.Std as the standard Radio for Pkg.400. The second (lower row) defines Radio.Dlx as the standard Radio for Pkg.401 and Pkg.402. The buildable space shows that Radio.Nav is also available. Because this feature condition is not included in the Standard feature conditions (i.e., Radio.Nav is not active (1) in condition **9204**), the navigation radio (Nav) is defined as an optional choice that can be added in place of the deluxe radio (Dlx) for the 401 or 402 package. There is a single feature condition for the moonroof family defining no moonroof (Less) as the standard package, as referenced by numeral **9206**. There are two standard feature conditions for the dice family, as referenced by numeral **9208**; the first defines no dice as the standard feature for white paint; and the second defines fuzzy dice as the standard feature for red and blue paint. There are three standard features for the seat temperature (Temp) family, as referenced by numeral **9210**; the first defines no seat temperature feature (LessTemp) as the standard feature for the 400 package; the second defines heated seat control (Heat) as the standard feature for the 401 package; and the third defines heated and cooled seat control (HeatCool) as the standard feature for the 402 package.

[0456] With reference to FIG. **93**, a method for automatically completing a configuration using a maximally standard algorithm is illustrated as software code in accordance with one or more embodiments and generally referenced by **9300**.

[0457] To automatically complete the configuration using standard feature conditions, the configuration engine **106** first restricts the buildable space to the minimally complete configuration at operation **9301**. Families absent from the configuration are processed in priority order, and the space is restricted for each successive choice. To make the default choice for a family, its standard feature conditions are inspected to see what standard choice(s) are still possible at operation **9304**. If no standard choices are possible, the domain of the restricted space will define possible optional choices at operation **9306**. Where more than one possible choice exists for a family, alternate sequencing is used to choose the highest priority feature as the default at operation **9308**.

[0458] It is uncommon, but valid, for a family to have to no standard feature conditions. This is handled as if no standard features are available, and the choice will be made from the optional content.

[0459] It is also valid for the standard feature conditions to define more than one standard feature for the same partial configuration. Where more than one standard feature is available, the highest priority feature will be chosen.

[0460] Where all feature conditions for a family are stored in a single MDD, the standard features that are still allowed with prior choices can be identified by an AND operation of this MDD with the current restricted buildable space. An AND operation generates a new MDD for each inspection of a feature condition MDD. As discussed in the Quick-Restrict section, this will incur performance problems from garbage collection. The alternate approach, as shown in FIG. **93**, is to divide the standard feature conditions by feature (and not just family) and use the containsAny operation. The containsAny operation is the same logic as an AND operation, however the new space is not created.

[0461] With reference to FIG. **94**, in one embodiment, the configuration engine **106** considers a scenario where a user has selected the Vista moonroof and Ruby trim. With these two selections, Fuzy Dice is included. Given a priority order of [Pkg, Radio, Moonrf, Temp, Paint, Dice, Trim], the families to autocomplete, in order are [Pkg, Radio, Temp, Paint]. The configuration engine **106** does not auto-complete the Dice, Trim and Moonroof families because they are already "complete", i.e., they have features with Selected or Included feature states.

[0462] The configuration engine **106** restricts the MDD to the minimally complete configuration Vista, Ruby, and Fuzzy Dice as shown by MDD **9400** in FIG. **94**.

[0463] First, the configuration engine **106** processes the package family (Pkg) because it has the highest priority. As described with reference to FIG. **92**, there is one standard feature condition for the package family (i.e., Pkg 400, **9202**). Because the 400 package (i.e., the left-most feature of Pkg) is not contained in the restricted space illustrated by MDD **9400**, the standard feature is not available. MDD **9400** shows that the domain of the package family is 011 showing that both the 401 and 402 packages are available in the restricted space. The configuration engine **106** chooses the 401 package based on priority and the space is further restricted to package 401 (i.e., nodes 9 and 10 are removed) as shown by MDD **9402**. The restricted space now contains a single superconfiguration.

[0464] Next, the configuration engine processes the radio family. As described with reference to FIG. **92**, there are two standard feature conditions for the radio family: one for Radio.Std and one for Radio.Dlx (**9204**, FIG. **92**). At this point the deluxe (Dlx) radio and the navigation (Nav) radio are available in the restricted space illustrated by MDD **9402**. The configuration engine **106** selects Dlx as a Default feature, because it is the only standard feature remaining in the restricted space, and further restricts the buildable space to Dlx, as indicated by the modified edge label from 011 to 010 and reference by numeral **9404**. However, this restriction will not change availability for other families since the space is already a single superconfiguration.

[0465] Next, the configuration engine **106** processes the seat temperature (Temp) family. There are three standard feature conditions for Temp (**9210**, FIG. **92**). But, since Pkg 401 has already been selected, only Temp.Heat will be available, as indicated by edge label 010 in MDD **9402**. Therefore the configuration engine **106** adds heated seats (Heat) to the configuration as a Default feature.

[0466] Next, the configuration engine **106** processes the paint family. There are no standard feature conditions for paint (Table **9200**, FIG. **92**). The domain of the restricted space has two choices—Red and Blue, as indicated by edge label 011 in MDD **9402**. The configuration engine **106** adds Red as the Default choice, and further restricts the MDD **9402** to Red (010) as referenced by numeral **9406**, because Red (010) has more weight than Blue (001).

[0467] Finally, the configuration engine **106** processes the dice family. There are two feature conditions for Dice (**9208**, FIG. **92**). Because Red paint has been added to the configuration, only Fuzzy Dice is available, and fuzzy dice is already an Included feature. Therefore the configuration engine **106** does not change its feature state.

[0468] With reference to FIGS. **95-99**, another method for automatically completing a configuration using a maximally standard algorithm is illustrated in accordance with one or more embodiments and generally referenced by S**800**. The maximally standard auto-completion method S**800** is implemented as an algorithm within the configuration engine **106** using software code contained within the server **102**, according to one or more embodiments. In other embodiments the software code is shared between the server **102** and the user devices **112**, **114**.

[0469] At S**802**, the configuration engine **106** generates a maximally standard data space (MsDataSpace) based on a total or global buildable space and a relationships work object (relWork). The total buildable space includes a main space that defines all possible valid configurations of non-deterministic features and relationships spaces that define the availability of deterministic features in terms of non-deterministic features. The intersection of the main space and the relationships spaces define all possible valid configurations. The main space and relationship spaces are specified by MDDs, according to one or more embodiments. The total buildable space includes the main MDD and a relationships MDD, and is quick-restricted to any Selected and Included features. RelWork is a temporary object that is used to process a specific condition that may occur based on the order families are processed. As discussed below, relWork is nontrivial if after selecting a Default feature from a deterministic family, there is more than one defining feature condition such that the configuration engine **106** cannot quick restrict the main MDD. Then at S**804**, the configuration engine **106** identifies any family without any Selected or Included features as a family to complete.

[0470] With reference to steps S**806-811**, the configuration engine **106** analyzes each family of the MDD in priority order. The configuration engine **106** starts analyzing the first family in the sorted families to complete list at step S**806**. At step S**808**, the configuration engine **106** determines if the index (x) is less than the total number of families included in the families to complete list. If not, the configuration engine **106** proceeds to S**810** (Done). If so, the configuration engine **106** proceeds to step S**812** to determine the possible available standard features for family (x) within the restricted buildable space (MsDataSpace). Once the configuration engine **106** has completed its analysis of family (x), it returns to S**811** and starts analyzing the next family by incrementing x by 1. The subroutine of S**812** is shown in the flowchart of FIG. **96**.

[0471] With reference to FIG. **96**, at S**814** the configuration engine **106** initializes the list of possible standard features by setting it to empty. With reference to steps S**816**, S**818** and S**842**, the configuration engine **106** analyzes each feature of family(x). The configuration engine **106** starts analyzing feature zero (z=0) at S**816**. At step S**818** the configuration engine **106** compares feature (z) to the number of features at the current family (x) to determine if z<the number of features in family (x). If the determination is positive, the configuration engine **106** proceeds to S**820** to determine the essential sets for feature availability. Once the configuration engine **106** has completed its analysis of feature (z) of family (x), it returns to S**842** and starts analyzing the next feature of family (x) by incrementing z by 1. The subroutine of S**820** is shown in the flowchart of FIG. **97**.

[0472] Referring to FIG. **97**, the configuration engine **106** determines a setlist of "essential sets" for the availability of feature (z) at S**820**-S**848**. At S**824** the configuration engine **106** initializes the setlist by adding the main MDD of the current buildable space (buildableSpace). Then at S**826**, the configuration engine **106** evaluates relWork to determine if it is not trivial (i.e., not all 1). If relWork is not all trivial, (i.e., a previous Default selection was made from a deterministic family such that the relationship defined more than one determinant condition) then the configuration engine **106** proceeds to S**828** and adds relWork to the setlist. After S**828**, or in response to a negative determination at S**826**, the configuration engine **106** proceeds to S**830** to determine if there are not any standard feature conditions defined for this feature. If there are no standard feature conditions defined for feature (z), then its standard space is null. If there are standard feature conditions defined for feature (z), then a negative determination is made at S**830**, and the configuration engine **106** proceeds to S**832** and adds the standard space to the setlist. After S**832**, or in response to a positive determination at S**830**, the configuration engine **106** proceeds to S**834** to determine if the family of feature (z) is deterministic. If family (x) is deterministic, the configuration engine **106** adds the relationship space for family (x) to the setlist at S**836**. After S**836**, or in response to a negative determination at S**834**, the configuration engine **106** proceeds to S**838** and returns the setlist to the subroutine of FIG. **96**.

[0473] Referring to FIG. **96**, at S**840** the configuration engine **106** determines if the intersection of all spaces in the setlist for feature (z) is empty (i.e., if feature (z) is not a

standard feature or it is a standard feature that is not available with the current configuration). If the determination at **S840** is negative, the configuration engine **106** proceeds to **S844** and adds feature (z) to the list of possible available standard features (POSSIBLE). After **S844**, or after a positive determination at **S840**, the configuration engine **106** proceeds to **S842** to analyze the next feature (z) of family (x). Once the configuration engine **106** has analyzed all features of family (x), it makes a negative determination at **S818** and proceeds to **S846** to return the possible available standard features for family (x) to the main maximally standard routine of FIG. **95**.

[0474] With reference to FIG. **95**, at **S848** the configuration engine determines if there are any standard features for family (x). If there are no standard features, i.e., if POSSIBLE is empty, then the configuration engine **106** proceeds to **S850** to find the domain of family (x) in the maximally standard data space (MsDataSpace). The subroutine of **S850** is shown in the flowchart of FIG. **98**.

[0475] With reference to FIG. **98**, the configuration engine **106** determines the domain of family (x) at **S850**-**S858**. At **S852**, the configuration engine **106** calculates the domain space of family (x) as the intersection of the main space and the relWork. If the configuration engine **106** determines that family (x) is deterministic at **S854** based on the product definition, then it proceeds to **S856**. At **S856** the configuration engine **106** sets the domain space equal to the intersection of the domain space and relationship space for that family. After **S856**, or in response to a determination that family (x) is not deterministic at **S854**, the configuration engine **106** proceeds to **S858** and sets the Domain to the domain of the domain space, adds any feature from family (x) that is present to the Domain to the list of possible features and returns to the main maximally standard routine of FIG. **95**.

[0476] Referring to FIG. **95**, after determining the domain of family (x) at **S850**, or in response to a negative determination at **S848**, the configuration engine **106** proceeds to **S860** and sorts POSSIBLE by an alternate sequence that defines the priority of the features. At **S862** the configuration engine **106** selects the first feature in the sorted list as the Default feature for Family (x). Then the configuration engine **106** proceeds to **S864** to restrict the maximally standard data space to the new Default feature. The subroutine of **S864** is shown in the flowchart of FIG. **99**.

[0477] With reference to FIG. **99**, at **S864**-**S880**, the configuration engine **106** further restricts the maximally standard data space to the new Default feature. At **S866** the configuration engine **106** restricts the main space to the new Default feature. Then if family (x) is deterministic, the configuration engine **106** proceeds to **S870** and defines a temporary relationship (relTemp) by restricting the family relationship space to the new Default feature choice. Then at **S872**, the configuration engine **106** sets relWork to the intersection of relWork and relTemp. At **S874**, the configuration engine **106** evaluates relWork to determine if it has a single path, i.e., a "singleton." If the standard features (relWork) is a singleton, the configuration engine **106** proceeds to **S876** and uses quick restrict to restrict the main space using the single bitmask from relWork; and then resets relWork to be trivial (all 1s) at **S878**. After **S878**, or in response to a negative determination at **S868** or **S874**, the configuration engine **106** proceeds to **S880** and returns to the main maximally standard routine of FIG. **95**.

[0478] With reference to FIGS. **100**-**101**, deterministic relationships are used to enable efficient creation and storage of the full buildable space. As described previously, the global buildable space can be stored in a Global MDD that has a main space (e.g., an MDD) and a set of relationship spaces (e.g., MDDs). Maximally standard auto completion requires standard feature conditions. The buildable space (FIG. **100**) and Standard Feature Conditions (FIG. **101**) are shown as Tables **10000** and **10100**, respectfully, and combined in a single Buildable object.

[0479] There is no concept of displayable and non-displayable families during MDD generation. Displayable families refer to content this is displayed to the user, for example paint color is displayed to a user in a build and price application. Whereas non-displayable families refer to content that is not displayed to the user, such as an electrical harness in a build and price application. A Superconfiguration Generator (SCG) library (not shown) is a component of the ETL **128** (FIG. **2**). The SCG library simply finds all relationships in order to generate the smallest possible main space.

[0480] Some algorithms that target displayable content (e.g., validation and feature state mapper) have been optimized to work on a single MDD, and others require all displayable content to be in a single MDD (e.g., minimum edit). Valid input for the configurator is a global space where no displayable families have been pulled to an external relationship, even if it is deterministic.

[0481] One possible way to build a global space that conforms to the configurator input is to control which families can be deterministic. The SCG algorithm includes an argument called relignore which can be used to specify which families are not allowed to be deterministic. This argument can be used to specify that displayable families are not allowed to be deterministic.

[0482] Another approach is to allow SCG to build the MDD by pulling out whatever relationships it finds. Then, an extra step is used before the space can be used by the configurator. Any relationship that defines a display family dependence on the main MDD is flattened back into the main MDD. To do this efficiently, the MDD is recompressed as the relationships are flattened.

[0483] Generally both approaches will take the same amount of processing time. In the second approach, the MDD may be generated much faster, but that any time savings is used in the extra flattening step.

[0484] The configuration engine **106** will be validating two types of configurations—a configuration of displayable features only or an expanded configuration of displayable and no display features.

[0485] To validate a configuration of displayable features the relationships can be ignored because the configuration engine **106** does not allow any displayable family to be deterministic. This means that the main MDD contains all of the information necessary to define feature combinations from the displayable families. There is no information in the relationships that will shrink the space of displayable families that is defined in the main MDD. Thus, only the main MDD is used to validate a configuration of display family features. To validate a configuration of displayable families, simply call contains operation on the main MDD.

[0486] To validate a configuration that contains displayable and no-display features, both the main MDD and the relationships are inspected. Just as the MDD containsAny

operation is used to validate a configuration against an MDD, the Global MDD also has a containsAny operation that can validate a configuration against a Global MDD. The Global MDD operation utilizes the MDD containsAnyParallel operation, to validate a fully expanded configuration, the parallel operation will turn the mask into its own MDD and inspect that space along with the main space and all relationship MDDs. To validate a partial configuration of both display and no display features, the parallel operation inspects the mask MDD, the main MDD and the relationships associated with each deterministic feature in the configuration. When processing a partial configuration, relationships for families that have no feature in the configuration can be ignored.

[0487] FIG. **102** is a table **10200** that shows several example configurations, the relevant MDDs, and the containsAny call that the configuration engine **106** uses to validate the configuration.

[0488] Conflict resolution is also limited to displayable content. As with the "contains" operation, when dealing with configurations that do not contain deterministic features, the main MDD contains sufficient information for the conflict resolution without including the relationships.

[0489] As described with reference to FIG. **47**, conflict resolution begins with a minimum edit distance calculation that is performed on the main MDD.

[0490] For Single Conflict Resolution, the target configuration is identified by performing auto completion in the minimum edit space. Because the main MDD includes no display content, the target configuration will also include no display content. The no-display content is stripped from the configuration engine response. Alternatively, the author decorators can be modified to ignore no-display features when adding conflict resolution to the xml response.

[0491] For Branched Conflict Resolution, the entire minimum edit space is used to build the response. The minimum edit space is projected to displayable content before building the response.

[0492] There is no concept of displayable and non-displayable families during the first stage of authoring or determining the feature conditions. As such, the feature conditions for displayable families may be determined with a dependency on no-display families. This means that when the maximally standard auto completion algorithm uses the full configuration space; it accounts for the external relationship MDDs in addition to the main MDD.

[0493] When using a Global Space (main space+relationships spaces) for maximally standard auto completion, the basic logic is the same as using an MDD. Instead of checking a single MDD, the algorithm checks the global space—both the main MDD and the relationship MDDs. The basic operations of restrict, containsAny and domain accounts for both the main space and external relationships.

[0494] During auto completion, the configuration engine **106** restricts the main space for each feature choice (S**864**). When the choice is deterministic, the external relationship defines the determinant conditions for each feature. The deterministic relationship space is restricted to the feature choice to determine the determinant conditions and then the main space is restricted to the corresponding determinant conditions.

[0495] When a deterministic feature has a single determinant condition, the configuration engine **106** quick-restricts the main space using the determinant condition, according to one or more embodiments (S**874**-S**876**). The buildable space as shown in Table **10000** (FIG. **100**) includes a main space **10002** and a relationships space **10004**. The relationships space **10004** shows that features Y2, Y5 and Y6 map to B1 as referenced by numeral **10006**, and that features Y1, Y3, and Y4 map to B2 as referenced by numeral **10008**. Table **10300** of FIG. **103** shows only one row remains after the relationship is restricted to the choice B1 (S**870**). The configuration engine **106** quick-restricts the main space using this single superconfiguration as the bitmask. After B1 is chosen, the main space is restricted to B1 (S**866**) and is also restricted to Y2, Y5, and Y6 (S**876**), as shown in Table **10400** of FIG. **104**, and referenced by numeral **10402**.

[0496] The quick-restrict operation accepts a single bit mask. This means that the main space cannot be quick-restricted to reflect a deterministic choice whenever a deterministic feature has multiple determinant conditions. Table **10000** (FIG. **100**) shows that family K is determined by two families [A,S] as referenced by numeral **10010**. Table **10500** of FIG. **105** shows the two rows defining the determinant conditions for feature K2. The first row of Table **10500** shows that A2,S3; A2,S5; A2,S6 all map to K2 and the second row shows that A1,S3 also maps to K2. When K2 is chosen, the main space is restricted to K2, but the configuration engine **106** cannot quick-restrict it with the K2 determinants because the restricted relationship space defines two superconfigurations. Instead, the configuration engine **106** adds a special relWork space, as referenced by numeral **10602**, to store the determinant conditions as shown FIG. **106**. After restricting to K2, relWork contains the two determinant conditions for K2.

[0497] Just as the global space requires both the main MDD and all its relationships to fully define the space, relWork is necessary to fully define the restricted space. However, if relWork is trivial (i.e., all 1s) then the configuration engine **106** can ignore it because it isn't further restricting the space.

[0498] The configuration engine **106** initializes the relWork space as a trivial space, with a single superconfiguration of all 1s, according to one or more embodiments. When a deterministic choice is made that has multiple determinant conditions, relWork is AND'ed with the restricted relationship space (S**872**). If the result of the AND operation is a single superconfiguration (S**874**), the main space is restricted with that superconfiguration (S**876**) and relWork is trivialized (S**878**).

[0499] Referring to FIG. **107**, if the configuration engine **106** further restricts the space to M2, then there is just one row remaining in relationship M as shown in Table **10700**. When this restricted relationship space is ANDed with relWork (from FIG. **106**), just one row remains as shown in Table **10800** of FIG. **108**. This is used to restrict the main space, and then relWork is reset, with the final result shown in Table **10900** of FIG. **109**.

[0500] The configuration engine **106** uses the containsAny operation of the maximally standard auto completion algorithm to determine if a Standard feature condition space is still valid in the restricted space, according to one or more embodiments.

[0501] For two MDDs, the operation mdd1.containsAny (mdd2) is equivalent to not(isEmpty(mdd1.and(mdd2)). The ContainsAny operation can be extended to operate on two or more MDDs and is called containsAnyParallel. For three

MDDs, the operation mdd1.containsAnyParallel([mdd2, mdd3]) is equivalent to not(isEmpty(mdd1.and(mdd2).and (mdd3)).

[0502] When dealing with deterministic relationships, this contains any operation may need to include the relWork MDD.

[0503] In order for the configuration engine **106** to determine if a standard feature is available in the restricted space (S**812**), the containsAny operation must always operate on the standard space and the main space and may need to account for the relWork space and an external relationship space. When the family is deterministic and relWork is not trivial the operation will be standardSpace.containsAny (mainSpace, rel, relWork). The operation can skip relWork if it is trivial and will only include a relationship if the feature is deterministic (S**822**).

[0504] The configuration engine **106** determines if any standard features for A are still valid in the space from FIG. **103**, by accounting for the A standard space and the main space in the containsAny operation. There is no relationship space because A is not deterministic and the relWork is ignored because it is trivial (i.e., A is all is in Table **10300**).

[0505] In order for the configuration engine **106** to determine if any standard features for B are still valid in the space from FIG. **103**, the containsAny operation must account for the B standard space, the main space and relationship B space. The configuration engine **106** ignores relWork because it is trivial.

[0506] The configuration engine **106** determines if any standard features for M are still valid in the space from FIG. **106**, by accounting for the M standard Space, main Space, Relationship M space, and relWork in the containsAny operation.

[0507] The maximally standard auto completion algorithm uses domain calculation when the standard feature conditions do not identify a possible choice. Because only the domain of a single family is needed, not all of the relationships must be included. The domain calculation must consider the main space, the relWork space, and, if the family is deterministic, the external relationship space. This is done by first ANDing the spaces, and then calculating the domain on the resulting space (S**850**).

[0508] The algorithm for maximally standard auto completion without deterministic relationships was shown previously in FIG. **93**.

[0509] The modifications to account for deterministic relationships are:

[0510] 1) Change mdd.quickRestrict to SPACE.RESTRICT;

[0511] 2) Change mdd.containsAny with SPACE.containsAny, where space defines the main MDD, the relationship MDDs, and the relWork MDD discussed above in the Restrict Global Space section; and

[0512] 3) Change mdd.findDomain.toListActive(Fa) to SPACE.findDomain(Fa).

The new algorithm is shown as flowcharts in FIGS. **95-99** and as software code in FIG. **110**.

[0513] The following example illustrates the modified algorithm and references steps in the flow charts and operations in the software code where applicable. This example will use the global space and standard feature conditions defined previously in Table **10000** (FIG. **100**) and Table **10100** (FIG. **101**). Table **10100** also lists the family priority order, as generally referenced by numeral **10102**.

[0514] With reference to FIG. **111**, the configuration engine **106** restricts the space starting with a minimally complete configuration of Selected {Y3,E1,P1}+Included {F2} (**11001**, FIG. **110**; S**802**, FIG. **95**). The configuration engine **106** makes Default choices for the remaining families in priority order as defined in Table **10100** (FIG. **101**), i.e.: V, R, K, B, A, M, T, I, S (S**804**, FIG. **95**).

[0515] Referring to FIG. **112**, the configuration engine **106** determines that Family V is deterministic and includes a single standard feature condition. To determine if standard feature V3 is available, the configuration engine **106** checks the main space, standard space and deterministic relationship (S**820**, FIG. **97**). Operation **11004** (FIG. **110**) stdV3. containsAnyParallel(mainSpace, relV) returns false because V3 is not available with the current choice Y3 (see also S**840**, FIG. **96**). The domain of V, from operation **1106** (FIG. **110**) AND(relV,mainSpace), will identify only one possible choice (see also S**852**-S**858**, FIG. **98**). V2 is added and the newly restricted space, as determined at S**864**, FIG. **96** and operation **11008**, FIG. **110**, is shown in Table **11200** of FIG. **112** and referenced by numeral **11202**.

[0516] With reference to FIGS. **112-113**, the configuration engine **106** determines that Family R has no standard feature conditions (S**830**-S**836**, FIG. **97**; operation **11004**, FIG. **110**). The domain of the restricted space identifies two possible choices—R1, R4 as referenced by numeral **11204** (S**852**-S**858**, FIG. **98**; operation **1106**, FIG. **110**). Without alternate sequencing, R1 is picked as the Default choice and the space is further restricted (S**864**, FIG. **99**; operation **11008**, FIG. **110**) as shown in Table **11300** of FIG. **113**, and referenced by numeral **11302**.

[0517] Referring back to FIG. **101**, the configuration engine **106** determines that Family K has two standard feature conditions (S**830**-S**836**, FIG. **97**; operation **11004**, FIG. **110**). StdK1 defines K1 as standard for V1 and StdK2 defines K2 as standard for V2 or V3. Because V2 has been previously chosen, K2 is the only available standard choice and is added to the configuration. The configuration engine **106** further restricts the space to reflect this choice (S**864**, FIG. **99**; operation **11008**, FIG. **110**). Family K is deterministic. When RelK is restricted to K2, there are two rows remaining. The main space cannot be quick-restricted and relWork is updated as shown in Table **11400** of FIG. **114**.

[0518] With reference to FIG. **115**, the configuration engine **106** adds B2 based on its Standard space and chooses A2 because it is standard with {R1, V2}. The restricted space after these choices is shown in Table **11500** of FIG. **115**. Table **11500** shows that the configuration engine **106** could restrict relWork to A2, minimize it to one row, use it to restrict the main space and then trivialize it; however, the containsAny optimizations (operation **11004**, FIG. **110**) dictate that it is actually better to wait until relWork is minimized by another deterministic feature. It is actually counterproductive to restrict relWork for every choice.

[0519] Referring to FIG. **116**, the configuration engine **106** determines that Family M has standard feature conditions (S**830**-S**836**, FIG. **97**; operation **11004**, FIG. **110**). M1 is the only standard feature that is still available in the restricted space. After relWork is updated for M1, only one row remains, i.e., the second row of Table **11500** of FIG. **115**. This row is used to restrict the main space and relWork is trivialized, as shown by Table **11600** of FIG. **116** (S**864**, FIG. **99**; operation **11008**, FIG. **110**). The restricted space after processing family M is shown in Table **11600**.

[0520] With reference to FIG. **117**, the configuration engine **106** adds T1 as the Default choice from its Standard Feature Condition and I1 is chosen as the Default from the possible choices I1 and I2 (S**830**-S**836**, FIG. **97**; operation **11004**, FIG. **110**). The deterministic relationship for I shows

that I1 maps to S1 or S5. After T1 and I1 are chosen, S5 remains the only choice for family S, as shown in Table **11700** of FIG. **117**.

[0521] The configuration engine's use of relWork to account for deterministic relationships when quick-restricting the main space, along with the containsAnyParallel operation, allows for a very efficient maximally standard auto completion algorithm. This allows the configuration engine **106** to support maximally standard configurations without requiring feature condition authoring to be modified in order to account for display and no display families.

[0522] Computing devices described herein, generally include computer-executable instructions where the instructions may be executable by one or more computing devices such as those listed above. Computer-executable instructions may be compiled or interpreted from computer programs created using a variety of programming languages and/or technologies, including, without limitation, and either alone or in combination, Java™, C, C++, C#, Visual Basic, Java Script, Perl, etc. In general, a processor (e.g., a microprocessor) receives instructions, e.g., from a memory, a computer-readable medium, etc., and executes these instructions, thereby performing one or more processes, including one or more of the processes described herein. Such instructions and other data may be stored and transmitted using a variety of computer-readable media.

[0523] While exemplary embodiments are described above, it is not intended that these embodiments describe all possible forms of the invention. Rather, the words used in the specification are words of description rather than limitation, and it is understood that various changes may be made without departing from the spirit and scope of the invention. Additionally, the features of various implementing embodiments may be combined to form further embodiments of the invention.

What is claimed is:

1. A system comprising:

a memory device adapted to store data representative of a multi-valued decision diagram (MDD) specifying a buildable space of all possible valid configurations of a vehicle, the MDD including a root node, a truth node, and at least one level of intervening nodes, each level of the MDD corresponding to a family of mutually-exclusive features represented by at least one node, each intervening node of a level connecting to nodes of a next adjacent level by outgoing edges having labels indicating valid features of the family and to nodes of a prior adjacent level by incoming edges that are outgoing edges of the prior adjacent level, such that a complete path from the root node through the outgoing edges to the truth node defines at least one of the valid configurations; and

a processor in communication with the memory, programmed to

identify an invalid configuration,

generate a restricted buildable space, including to

determine an edit distance of each complete path indicative of a number of features to change the invalid configuration of that path to one of the valid configurations,

identify a minimum of the edit distances, and

remove configurations having edit distances larger than the minimum; and

identify at least one feature to change the invalid configuration to at least one valid configuration based on the restricted buildable space; and

generate output indicative of the at least one feature to change.

2. The system of claim **1**, wherein the processor is further programmed to provide a message to a user indicative of the at least one feature to change.

3. The system of claim **1**, wherein the processor is further programmed to:

receive input indicative of a non-guided resolution; and

generate a further restricted buildable space having a single target configuration, including to

determine a weight of each path in the restricted buildable space, indicative of a priority of its features,

identify the maximum weight of all paths, and

remove configurations having weights that are less than the maximum weight.

4. The system of claim **3**, wherein each feature is assigned a weight such that each path is uniquely weighted.

5. The system of claim **1**, wherein the processor is further programmed to:

receive input indicative of a non-guided resolution; and

generate a further restricted buildable space having a single target configuration, including to

determine a number of standard features of each path in the restricted buildable space,

identify a maximum standard number of features of all paths, and

remove configurations having less than the maximum standard number of features.

6. The system of claim **1**, wherein the processor is further programmed to:

receive input indicative of a non-guided resolution;

determine a first configuration corresponding to one of the valid configurations;

determine a second configuration corresponding to the invalid configuration in response to receiving a selection of at least one feature;

determine a first feature state for each feature of the first configuration, including at least one of Selected, Included and Default feature states;

generate a further restricted buildable space having a single target configuration, including to

determine a second feature state for each feature of the second configuration, including at least one of Selected, Included and Default feature states;

identify additions in the second configuration indicative of features present in the second configuration and not present in the first configuration, and the corresponding second feature state of each feature; and

identify subtractions in the second configuration indicative of features present in the first configuration and not present in the second configuration, and the corresponding first feature state of each feature.

7. The system of claim **6**, wherein the processor is further programmed to:

receive a return delta indicative of an amount of information provided to a user in response to an invalid configuration; and

provide a message to the user indicative of the at least one feature to change to at least one valid configuration based on the further restricted buildable space and the return delta.

8. The system of claim **7**, wherein the message is further indicative of all additions and subtractions in response to a return delta of true; and

wherein the message is further indicative of no additions or subtractions having Default feature states in response to a return delta of false.

9. The system of claim **7**, wherein the message is further indicative of only additions having Included feature states and only subtractions having Selected feature states in response to a return delta of false.

10. The system of claim **1**, wherein the processor is further programmed to:

receive input indicative of a guided resolution; and

generate a further restricted buildable space having at least one target configuration, including to

determine a node weight indicative of a weight of the node to the truth node along each path in the restricted buildable space, wherein a positive weight is assigned to each feature included in the configuration and a weight of zero is assigned to each feature that is not included in the configuration,

identify a maximum node weight of all paths, and

remove configurations having weights that are less than the maximum node weight, and thereby remove partial matches.

11. The system of claim **1**, wherein the processor is further programmed to:

receive input indicative of a guided resolution;

determine a first configuration corresponding to one of the valid configurations;

determine a second configuration corresponding to the invalid configuration in response to receiving a second selection of at least one feature;

generate a further restricted buildable space having multiple target configurations, including to

determine a bitset of the second configuration,

determine a domain bitset of the restricted space,

calculate a bitwise conjunction of the second configuration bitset and the restricted space bitset, and

identify at least one family to change as a family having no active bits in the bitwise conjunction; and

provide a message to a user that displays the at least one feature to change from each family to change, to resolve the at least one feature.

12. The system of claim **11**, wherein the processor is further programmed to:

determine a resolution object, including nested resolution objects, indicative of actions required to change the invalid configuration to one of the valid configurations in the buildable space, wherein every family to change will have at least one action, and wherein there is more than one choice, including to

identify a branch family as any family that leads to a divergent choice and the choices for remaining families are unknown until the branch family choice is identified,

identify a no-branch family as any family that does not lead to divergent choices for remaining families, and

provide the message to the user that displays the features from the no-branch families and no more than one branch family, and prompt the user to select from the displayed features.

13. A method comprising:

storing, in a memory, data representative of a multi-valued decision diagram (MDD) specifying a buildable space of all possible valid configurations of a vehicle, the MDD including a root node, a truth node, and at least one level of intervening nodes, each level of the MDD corresponding to a family of mutually-exclusive features represented by at least one node, each intervening node of a level connecting to nodes of a next adjacent level by outgoing edges having labels indicating valid features of the family and to nodes of a prior adjacent level by incoming edges that are outgoing edges of the prior adjacent level, such that a complete path from the root node through the outgoing edges to the truth node defines at least one of the valid configurations;

receiving input indicative of a non-guided resolution;

identifying an invalid configuration;

generating a restricted buildable space, including

determining an edit distance of each complete path indicative of a number of features to change the invalid configuration of that path to one of the valid configurations,

identifying a minimum of the edit distances, and

removing configurations having edit distances larger than the minimum,

generating a further restricted buildable space having a single target configuration, including

determining a weight of each path in the restricted buildable space, indicative of a priority of its features,

identifying the maximum weight of all paths, and

removing configurations having weights that are less than the maximum weight;

identifying at least one feature to change the invalid configuration to at least one valid configuration based on the further restricted buildable space; and

generating output indicative of the at least one feature to change.

14. The method of claim **13**, further comprising:

determining a first configuration corresponding to one of the invalid configurations;

determining a second configuration corresponding to the single target configuration;

determining a first feature state for each feature of the first configuration, including at least one of Selected, Included and Default feature states;

determining a second feature state for each feature of the second configuration, including at least one of Selected, Included and Default feature states;

identifying additions in the second configuration indicative of features present in the second configuration and not present in the first configuration, and the corresponding second feature state of each feature;

identifying subtractions in the second configuration indicative of features present in the first configuration and not present in the second configuration, and the corresponding first feature state of each feature;

receiving a return delta indicative of an amount of information provided to a user in response to an invalid configuration; and

providing a message to the user that displays the at least one feature to change from each family to change to resolve the at least one feature.

15. The method of claim **14**, wherein the message is indicative of all additions and subtractions in response to a return delta of true; and

wherein the message is further indicative of no additions or subtractions having Default feature states in response to a return delta of false.

16. The method of claim **14**, wherein the message is indicative of only additions having Included feature states and only subtractions having Selected feature states in response to a return delta of false.

17. A method comprising:

storing, in a memory, data representative of a multi-valued decision diagram (MDD) specifying a buildable space of all possible valid configurations of a vehicle, the MDD including a root node, a truth node, and at least one level of intervening nodes, each level of the MDD corresponding to a family of mutually-exclusive features represented by at least one node, each intervening node of a level connecting to nodes of a next adjacent level by outgoing edges having labels indicating valid features of the family and to nodes of a prior adjacent level by incoming edges that are outgoing edges of the prior adjacent level, such that a complete path from the root node through the outgoing edges to the truth node defines at least one of the valid configurations;

receiving input indicative of a guided resolution;

identifying an invalid configuration;

generating a restricted buildable space having multiple target configurations, including

determining an edit distance of each complete path indicative of a number of features to change the invalid configuration of that path to one of the valid configurations,

identifying a minimum of the edit distances, and

removing configurations having edit distances larger than the minimum,

identifying at least one family to change, including

determining a bitset of the invalid configuration,

determining a domain bitset of the restricted space,

calculating a bitwise conjunction of the invalid configuration bitset and the restricted space bitset, and

identifying the at least one family to change as a family having no active bits in the bitwise conjunction; and

providing a message to a user that displays at least one feature to change from each family to change, to resolve the at least one feature.

18. The method of claim **17**, further comprising:

determining a resolution object, including nested resolution objects, indicative of actions required to change the invalid configuration to one of the valid configurations in the buildable space, wherein every family to change will have at least one action, and wherein there is more than one choice, including

identifying a branch family as any family that leads to a divergent choice and the choices for remaining families are unknown until the branch family choice is identified,

identifying a no-branch family as any family as any family that does not lead to divergent choices for remaining families, and

providing the message to the user that displays the features from the no-branch families and no more than one branch family, and prompt the user to select from the displayed features.

19. The method of claim **17**, further comprising:

determining a node weight indicative of a weight of the node to the truth node along each path in the restricted buildable space, wherein a positive weight is assigned to each feature included in the configuration and a weight of zero is assigned to each feature that is not included in the configuration;

identifying a maximum node weight of all paths; and

removing configurations having weights that are less than the maximum node weight, and thereby remove partial matches.

20. The method of claim **17**, further comprising receiving a user selection of a feature that resulted in the invalid configuration.

* * * * *