Hierarchical Decomposition Tool: Scripting Language and Implementation

(draft, private use only)

Blaž Zupan

January 8, 1997

Contents

1 Script language: Description and usage

1.1 Variables, descriptions, and structure

add var ID
del var ID
del vars { IDLIST }
Commands used to add and delete variables. ID is a string starting with a letter and may include digits and characters "." and "_". `IDLIST` is a list of variables ID's, separated with commas.

ID depends on { IDLIST }
Asserts a dependency of variable ID on variables listed in IDLIST. In case of undefined variables it asserts them (like using `add var`).

list var[**s**]
list struc[**ture**]
Lists known variables and structure.

Script: (`ex0.s`)

```
a depends on {b,c}
b depends on {d,e}
c depends on {f,g}
echo variables:
list vars
echo structure:
list struct
```

Output: (`% sds -c ex0.s`)

```
variables:
g f e d c b a
structure:
a
  b
    d
    e
  c
    f
    g
>>
```

Example 1: Definition of variables and dependencies.

IDLIST in { VALLIST }
Qualitative values of variables in IDLIST are those mentioned in the VALLIST. Syntax for

VALLIST is the same as for IDLIST, except when `prefer QUALITATIVE` is set (default), it allows items in IDLIST to start with a digit.

Script: (`ex1.s`)

```
y depends on {a, b, c, d}
set print 4
a in {small,med,big,extra_big}
b,c in {0,1,2,3}
list desc
```

Output: (`% sds -c ex1.s`)

```
   d   :
   c   : 0    1    2    3
   b   : 0    1    2    3
   a   : smal med  big  extr
   y   :
>>
```

Example 2: Defining qualitative values of variables.

1.2 Decision tables

For each non-leaf variables, the decision table with instances in the form of attribute-valued vectors may be given.

sel ID
This command should precede any assignment of rules to a variable. Variable ID should be an internal node of the structure. After this commands, all the rule assignment and other rule operations will refer to this node.

rule (VALLIST)
Defines the rules for the selected node variable. VALLIST gives first the values of the input variables (variables that selected variable is directly dependent on) and the value of selected variable (last in the list).

rule table { VALLIST }
Like `rule (VALLIST)`, except several rules can be specified. Interpreter requires "}" to be in the same line as the last value of `VALLIST`.

[**list**|**ls**]
Lists the rules of selected variable.

del rules
del rule (VALLIST)

del rule NUM
Delete all rules or a specified rule.

1.3 Decomposition and operations on structure

decompose ID
Decomposes structure rooted at variable ID.

one decompose ID
Does a single step of decomposition of structure at variable ID.

1.4 Decomposition preferences

set color {opt|heuristic}
Sets the type of coloring to be used with decomposition. Default is `heuristic`.

set decomposition debug NUM
Sets a debugging level for decomposition. Default is 0.

set decomposition NUM1 [**to NUM2** [**and NUM3 to NUM4**]]
Sets the size of a bounding set to be either NUM1, or be within a range NUM1 to NUM2. Default is NUM1=NUM2=2. If NUM3 and NUM4 are used, then they specify a range of number of atributes present in both bound and free set.

set dm {yes|no}
Specifies if decomposition should use a decomposition matrix explicitly, or construct the incompatibility matrix directly withouth construction decomposition matrix.

1.5 Testing the effects of decomposition

copy ID1 to table ID2
Copies rules for variable ID1 to table ID2.

list table ID
Lists the rules of rule table ID.

copy NUM % from table ID2 to ID1
After this command rules for variable ID1 are a copy of NUM % of rules from table ID2.

compare ID1 to table ID2
Compares rules of variable ID1 to those stored in table ID2.

Script: (`ex2.s`)

```
a,b,c,d,e in {0,1}
a depends on {b,c}
c depends on {d,e}

sel a
rule (0,1,1)
rule 3=1
echo rules for a:
list

sel c
rule table {
  0 0 0
  1 0 1
}
echo rules for c:
list
```

Output: (`% sds -c ex2.s`)

```
rules for a:
nrules 4
(list)
    b c = a
    0 0   ?
    0 1   1
    1 0   ?
    1 1   1
rules for c:
nrules 4
(list)
    d e = c
    0 0   0
    0 1   ?
    1 0   1
    1 1   ?
>>
```

Example 3: Several ways to define the decision table. Usually `rule table` would be used.

Script: (`ex3.s`)

```
y depends on {a, b, c}
a,b,c,y in {0,1}
sel y
rule table {
  0 0 0 0
  0 0 1 0
  0 1 0 0
  0 1 1 0
  1 0 0 0
  1 0 1 1
  1 1 0 1
  1 1 1 0}
copy y to table tmp
copy 30 % of table tmp to y
decompose y
compare y to table tmp
```

Output: (`% sds -c ex3.s`)

```
Copy from table tmp to y (30%) (8->2)
of 8 different 2 ( 25.00%)
>>
```

Example 4: This examples illustrates the use and verification of generalization capabilities of decomposition. Suppose that `y=a AND (b XOR c)`. We first set y to a complete list of this function, copy the rules to a temporary table, and then select 30% of rules from this table to be those for `y`. Then we do a decomposition and test the resulting decomposed structure with the rules of a temporary table.

set repeat test NUM
test decompose ID1 for NUM1 to NUM2 step NUM3 rules of ID2
Last three lines of the script in the last examples can be replaced by more elaborate test. This would test the decomposition of ID1 and would compare the rules obtained to those from the table ID2. Before every decomposition, a certain percentage of randomly selected rules from table ID2 would be included for ID2. The percentage would be from NUM1 to NUM2. Number of tests for each distinct percentage is set by `set repeat test NUM` directive.

1.6 Other operations on structure

compose ID
Flattens the structure below variable ID. If this command is preceded with command `decompose ID`, the structure remains the same but the rules for ID might change due to rule generalization of decomposition process.

join IDLIST FOR ID1 TO ID2
Combines variables in IDLIST that ID1 directly depends on and constructs a new variable ID2.

decompose table ID on IDLIST1 and IDLIST2
Constructs partition table for decomposition of variables ID directly depends on to partitions IDLIST1 and IDLIST2.

1.7 Other

 # COMMENT
A comment in a script.

quit
Exits the program.

load ID
Loads the script named ID.

log TEXT
Writes `TEXT` to a log file.

echo TEXT
Writes `TEXT` on stdout.

prefer QUALITATIVE
prefer QUANTITATIVE
Tells interpreter how to interpret integer numbers. For example, in case for `prefer`

`QUALITATIVE`, where suitable, 123 would be interpreted as a string "123" rather than a number 123. By default, string interpretation is preferred.

set print INT
Use first INT characters when printing variable names and their values. Default is `set print 1`.

2 Implementation

2.1 Organization of the Sds program

sds source code is structured into several modules which perform different tasks. The script language parsing is done through the use of LEX and YACC. Modules and files used are:

`sds.h`	header file with global variables and typedefs
`cmd.y`	yacc description
`lex`	lexical analyzer
`color.c`	coloring algorithms
`decomp.c`	automatic and semi-automatic structure decomposition, uses graph coloring defined in color.c
`desc.c`	manages descriptors of variables and their interval or fuzzy representations
`eval.c`	evaluation of the model, performs simple crisp and more complex interval and fuzzy evaluation
`fuzzy.c`	automatic fuzzy identification
`heuristics.c`	decomposition heuristics, computes informativity measure, gain ratio, gini index, relieff measure
`io.c`	handling of save instruction, saves the model and existing settings to file
`learn.c`	learns fuzzy and interval representations, uses genetic algorithm (pgapack)
`misc.c`	utilities, memory handling, etc.
`opt.c`	handles the definition of options, for which it can then call evaluation routines
`rules.c`	definition of rules, deletion, rule tables
`sds.c`	main routine, calls initialization routines and invokes script processor
`struct.c`	adding, deleting variables, manual definition of dependencies

2.2 Sds Invocation

```
Usage: sds [options]
 -l    lexical debugging
 -g    ga debugging
```

```
 -e    evaluation debugging
 -d    check descriptor consistency
 -c f  read cmds from command file f
```

Useful is also a Perl script `gen`, whcih is invoked by

```
gen -fFUNCT
```

where `FUNCT` is the name of the function it has to output. The function is output as a `sds` script. Try, for example, the following (both programs are located in `~blaz/bin`):

```
% gen -fb2 > tmp.s
% sds -c tmp.s
```