AMERICAN UNIVERSITY OF BEIRUT

Specification Construction Using Equivalence Relations and SMT Solvers

by

Rabeeh Ghaleb Abou Ismail

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
September 2015

AMERICAN UNIVERSITY OF BEIRUT

Specification Construction Using Equivalence Relations and SMT Solvers

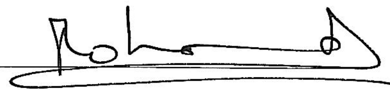
by

Rabeeh Ghaleb Abou Ismail

Approved by:

Dr. Paul Attie, Associate Professor — Advisor
Computer Science

Dr. Fadi Zaraket, Assistant Professor — Member of Committee
Electrical and Computer Engineering



Dr. Mohammad Jaber, Assistant Professor — Member of Committee
Computer Science

Date of thesis defense: September 14, 2015

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Abou Ismail Rabeeh Ghaleb

Last First Middle

● Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☑ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, **three years after the date of submitting my thesis, dissertation, or project,** to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

[signature] 28-Sept-2015

Signature Date

An Abstract of the Thesis of

Rabeeh Ghaleb Abou Ismail for Master of Computer Science
Major: Computer Science

Title: Specification Construction Using Equivalence Relations and SMT Solvers

We propose an approach to write formal specifications. Our approach partitions the (possibly infinite) state-space of the specification into a finite number of equivalence classes. The partition is defined by the equivalence relation induced by the valuations of a finite set of first-order logic formulae. Our work builds on existing work, which presents a method for writing specifications, along with a preliminary text-based implementation. In this thesis, we extend the current implementation with a graphical-user interface, and use this implementation to conduct experiments with the goal of demonstrating the value of the method by using it to write difficult and intricate specifications, and also using the experimental results as feedback for further improvements to the method.

Contents

Abstract 4

1 Introduction 1

1.1 Introduction and motivation 1

1.2 The specification construction problem 3

2 Specification Construction 4

2.1 The specification construction algorithm 4

2.1.1 Partial assignments using variables 4

2.1.2 Partial assignments using vocabulary 5

2.2 Figure 2.1shows the constructFor algorithm taken from Atttie et al [1] . 5

3 Implementation 7

3.1 Software architecture 7

3.2 External software 7

3.2.1 Z3 7

3.2.2 ANTLR 8

3.2.3 QT 8

3.2.4 ABC 8

3.2.5 EXPRESSO 8

3.2.6 GraphVIZ 9

4 User Manual 10

4.1 Tool and user interface 10

4.2 Command line run 11

4.2.1 Running a file 11
4.2.2 Building a vocab from the theory 11
4.2.3 Injecting an existential quantifier 11
4.2.4 Quantifier free vocab 12
4.2.5 Checking the file 12
4.2.6 Running the tool with the GUI 12
4.2.7 Help 12
4.2.8 Sample in use 13
4.2.9 Disadvantages of the CLI 16
4.3 The graphical user interface (GUI) 17
4.3.1 Theory name text box (figure 4.2) 17
4.3.2 The variables table (figure 4.3) 18
4.3.3 Add variable button (figure 4.10) 19
4.3.4 Delete variable button (figure 4.8) 20
4.3.5 The grammar table (figure 4.9) 20
4.3.6 Add grammar button (figure 4.10) 21
4.3.7 Delete grammar button (figure 4.15) 23
4.3.8 The constants table (figure 4.16) 24
4.3.9 Add constant button (figure 4.17) 24
4.3.10 Delete constant button (figure 4.19) 24
4.3.11 The vocab table (figure 4.20) 25
4.3.12 Add vocabulary button (figure 4.22) 26
4.3.13 Delete vocabulary formula button (figure 4.25) 27
4.3.14 The accept and reject buttons (figure 4.26) 27
4.3.15 Inject universal/existential quantifier radio button (figure 4.27) . . 28
4.3.16 Number of operations per clause (figure 4.28) 28
4.3.17 Undo drop down list (figure 4.29) 28
4.3.18 Formula result box (figure 4.30) 28
4.3.19 Construct theory (figure 4.31) 29
4.3.20 New file button (figure 4.32) 29
4.3.21 Save file button (figure 4.33) 29

4.3.22 Load file button (figure 4.34) 30
4.3.23 Undo feature (figure 4.35) 30
4.3.24 Karnaugh map (figure 4.36) 31

5 Case Studies **32**

5.1 Sample use case scenario 32
5.2 "inorder.th" theory file 33
5.3 "eina.th" theory file 41

6 Proposed Work **47**

6.1 Future work 47
6.2 Conclusion 47

List of Figures

2.1 $\mathsf{ConstructFormula}(\nu, vtt, vff)$ 6

4.1 a snapshot of the general interface 17

4.2 the theory name box at the top left corner 17

4.3 the variables table is the top left table 18

4.4 the variables table while a theory is running 19

4.5 the variable add button having a plus sign located right under the variables table 19

4.6 the window for adding a new variable 19

4.7 the variable type dropdown list 20

4.8 the variable table delete button having an x sign located under the variables table 20

4.9 the Grammar table is the bottom left table 20

4.10 the grammar add button having a plus sign located right under the grammar table 21

4.11 the window for adding a new grammar 21

4.12 the left variable dropdown list while adding a grammar 21

4.13 the relation dropdown list 22

4.14 the right variable or literal dropdown list while adding a grammar 23

4.15 the grammar table delete button having an x sign located under the grammar table 23

4.16 the Constants table is the bottom table to the right of the grammar table 24

4.17 the constant add button having a plus sign located right under the constants table 24

4.18 the window for adding a new constant 24

4.19 the constant table delete button having an x sign located under the constants table 24

4.20 the Vocabulary table at the top right 25
4.21 the Vocabulary table while a theory is running 26
4.22 the vocab add button having a plus sign located right under the vocab table 26
4.23 the window for deciding the method to add the vocab in 26
4.24 the add new predicate window 27
4.25 the vocabulary table delete button having an x sign located under the vocabulary table 27
4.26 the accept and reject button on the bottom of the window 27
4.27 the radio buttons to inject universal or existential quantifiers 28
4.28 counter for the number of operations per clause 28
4.29 the undo drop down list located to the right of the theory name 28
4.30 the result box of the theory 28
4.31 the button on the top of the window to construct the theory from whatever variables, grammars and vocabs that exist. 29
4.32 the button on the top of the window to generate a new theory file 29
4.33 the button on the top of the window to save the existing theory constructed in the GUI 29
4.34 the button on the top of the window to load a theory file either previously saved from the GUI or written by the user from scratch 30
4.35 the undo button with the drop down list of the states passed 30
4.36 Karnaugh map on the bottom left representing all the state space 31

Chapter 1

Introduction

1.1 Introduction and motivation

Writing a (formal) specification has long been recognized as crucial part of the development of software. Formal specifications help in proving correctness of a program. In addition, some program synthesis methods require formal specification as their input. Attie et. al. [1] presented a method of writing a formal specification for a terminating program P which takes one input I and produces one output O. We hold as a basis that the user knows informally the purpose of P and how it should act, and can determine whether it is behaving normally or not according to his notion of a pre-condition and post-condition. So based on this assumption we follow that:

1. if the user was given an input, he can judge whether it is relevant or irrelevant. If it is relevant, it should be processed properly, and if not then it can be ignored.
2. if the user was given an output consistent with that input, he can judge if the output is correct with respect to the input or not.

Code synthesis techniques that do not start from a specification offer the user no notion of correctness except the own judgment of the user. Attie et. al. [1] propose a method that creates accurate specifications in first order logic, *including quantifiers* (To our knowledge this is the first technique to produce specification with a quantified formulae) and that only requires the user to:

1. Provide a set of variable declarations that form a type theory.
2. Describe the variables as index, bound, or data variables w.r.t. array variables in a simple grammar.
3. Judge a sequence of variable valuations generated using an SMT solver.

This method guarantees accurate specifications provided that the user makes all judgments correctly.

The rest of this report is as follows.
Chapter 2 Presents the specification construction algorithm. Chapter 3 Discusses our implementation as well as third party software used. Chapter 4 Presents the use of the CLI and how it works as well as the GUI and its features. Chapter 5 Presents several example applications of the method. Chapter 6 Discusses proposed work.

The proposed approach to write formal specifications partitions the (possibly infinite) state-space of the specification into a finite number of equivalence classes. The partition is defined by the equivalence relation induced by the valuations of a finite set of first-order logic formulae.

We define a behavior as a single input-output pair. The users intuition is modeled as a possibly infinite set of judgments over inputs and behaviors. This intuition gets formalized as a specification S which is a precondition-postcondition pair *(P,Q)* expressed as a first order logic formula

A precondition and a postcondition are evaluated over an input and a behavior respectively and we produce an accurate specification this way. The precondition holds for input iff the user judges this input relevant The postcondition holds for a behavior if the user judges the input irrelevant as in (dont care) or the output as correct with respect to the input.

We need simpler methods of generating specification because the developer usually commits errors and this approach reduces this. for example Sorting two integers x and y where the final permutation of x and y is ordered where we have $x < y$. The precondition is *true* since all valuations of x and y are acceptable Assume the post condition Q

$$Q \triangleq (x_i = x_o \wedge y_i = y_o) \wedge x_o \leq y_o.$$

For $x_i = 1$, $y_i = 2$, $x_o = 1$, $y_o = 2$, here $Q = tt$, where tt and $f\!f$ denote *true* and *false* respectively. Subscripts of i, o indicate the initial and final values, respectively. For some initial values of x and y this postcondition would hold but not for all such as $x_i = 2$, $y_i = 1$, $x_o = 1$, $y_o = 2$ Now consider *Q'*

$$Q' \triangleq [(x_i = x_o \wedge y_i = y_o) \vee (x_i = y_o \wedge y_i = x_o)] \wedge x_o \leq y_o,$$

The final values of x, y are an ordered permutation of the initial values. we have $Q' = tt$ and $Q = f\!f$.

This would be an accurate post condition But if we try arbitrary valuation we might get Q being sufficient So we conclude that to check accuracy of a *(P,Q)* pair we need to check all cases But we cannot check an infinite number of cases.

The main idea is that we can partition S in to a finite set of equivalence classes by producing a combination of relations between our variables. In the previous example we are only interested in the $<$ and $=$ relationship between the variables so generating all combination would result in 12 formulas. But this would still generate a relatively large number of equivalence classes 2^{12}.

To make this a useful procedure we need to:

1. Generate a representative behavior from each valuation formula.
2. Classify each behavior as correct or incorrect.

We deal with this by submitting the valuation formula to an SMT solver. Which will find a satisfying assignment if the valuation formula is satisfiable. We use the user as an oracle and he will interact with the algorithm and judge whether the behavior is correct or incorrect.

This method that creates accurate specifications in first order logic, *including quantifiers* and only requires the user to:

1. Provide a set of variable declarations that form a type theory.
2. Describe the variables as index, bound, or data variables w.r.t. array variables in a simple grammar, or provide his own set of vocabulary formulas.
3. Judge a sequence of variable valuations generated using an SMT solver.

This method guarantees accurate specifications provided that the user makes all judgments correctly.

1.2 The specification construction problem

Attie et. al. [1] reduced the specification construction problem in to a formula construction problem. To do that we need to define a type theory and vocabulary formulas.

The type theory is the set of our variables and their types and whether they are free or bound.

From our type theory we define a vocabulary based on equivalence relations between our variables, or based on user defined vocabulary clauses.

The tool developed generates a set of vocabulary formulas from these equivalence relations or the user can provide a set of vocab formulas if he has a better idea of how the specification is going to be constructed. In the first case where the vocabulary formulas are automatically generate the search space is usually large, but in the second case where the user provides his vocab it would make the search space smaller and therefore would make the user answer less queries before getting the final result.

Chapter 2

Specification Construction

2.1 The specification construction algorithm

The following is an informal outline of the algorithm presented.

1. Partition our theory in a finite number of equivalence classes
2. For each equivalence class find a valuation on it.
3. Submit this valuation to the SMT solver
   (a) If the solver fails then we return failure
   (b) If the solver returns un satisfiable remove it from equivalence classes
   (c) If it is satisfiable propose the current assignment to the user and let him decide whether it is a part of tt or ff.
      i. If the user places it in vtt, we do the conjunction of the current vocab based on their assignment and disjunction with the formula $F(EX : F \vee (f1 \wedge \neg f2 \wedge f3 \wedge f4))$
      ii. If the user rejects it, its partial assignments are removed from the search space
4. Simplify the formula
5. Return the formula

2.1.1 Partial assignments using variables

By picking variables as having bad values the algorithm checks what vocabulary valuations these values are causing and prunes out any equivalence classes that lead to a similar value for the vocab.

2.1.2 Partial assignments using vocabulary

If we decide that a vocabulary formula or a set of formulas is bad then we will do the conjunction of them and prune out any equivalence classes that have a similar assignment.

2.2 Figure 2.1shows the constructFor algorithm taken from Atttie et al [1]

ν : Represents our vocabulary formulas.
vtt : Represents the set of accepted valuations.
vff : Represents the set of rejected valuations.
Σ : Represents our set of behaviors.
Σ/ν : Represents the partition or our search space or behaviors based on the vocab.
$\mathcal{F}$: Represents our formula being constructed.
V_ν : Represents the vocabulary valuations.
φ : Represents the search space to be covered.
$fm(V_\nu))$: Represents the formula of the vocab valuation V_ν.
(σ_v) : Represents a query or a current behavior.

$\mathsf{ConstructFormula}(\nu, vtt, v\!f\!f)$
1. { Precondition: $\{vtt, v\!f\!f\}$ partitions Σ and $\Sigma/\nu \leq \{vtt, v\!f\!f\}$ }
2. $\mathcal{F}$:= false; $\varphi := \nu \mapsto \{tt, f\!f\}$
3. { Invariant: $\mathcal{F} \equiv (\bigvee V_\nu : V_\nu \in (\nu \mapsto \{tt, f\!f\}) - \varphi \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$ }
4. **while** $\varphi \neq \emptyset$
5. select some valuation $V_\nu \in \varphi$;
6. $\varphi := \varphi - \{V_\nu\}$;
7. submit $fm(V_\nu)$ to an SMT solver;
8. **if** $thesolverfails$**then return** ("$failure''$); return with failure
9. **if** $fm(V_\nu)$ is satisfiable **then**
10. let σ_v be the returned satisfying assignment;
11. query the developer: is σ_v in vtt or in $v\!f\!f$?
12. **if** developer answers $\sigma_v \in vtt$ **then** $\mathcal{F} := \mathcal{F} \frown$ "$\vee$" $\frown fm(V_\nu)$;
13. **else** $\varphi := \varphi - \mathsf{partialAssignment}(\sigma_v)$;
14. **else** solver returned unsat
15. $\varphi := \varphi - unsat$ where $unsat$ is the unsat core valuations;
16. **endwhile** ;
17. { Postcondition: $\mathcal{F} \equiv (\bigvee V_\nu : V_\nu \in \nu \mapsto \{tt, f\!f\} \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$ }
18. simplify $\mathcal{F}$ using ABC [3] and ESPRESSO [2];
19. **return** ($\mathcal{F}$);

Figure 2.1: $\mathsf{ConstructFormula}(\nu, vtt, v\!f\!f)$

Chapter 3

Implementation

3.1 Software architecture

The tool takes as its primary input a theory file consisting of variables and their types along with a set of constants. it also needs a set o vocabulary formulas in which case the input either has equivalence relations defined over the variables as grammar rules, or a set of vocabulary formulas which their combination is going to represent our search space. In the former case the vocabulary clauses are generated from the grammar rules along a line that states how many operations to be done on each equivalence relation defined. Lastly the user decides how many quantifiers to inject in a line because the result can have specifications with quantifiers.

This would be the primary input to construct our theory however during construction the tool takes the users feedback as input to start building the final formula as displayed in the algorithm above.

3.2 External software

Some external software was used in this project for different purposes:

3.2.1 Z3

Z3 is the main SMT solver we are using, Z3 is a high-performance open source theorem prover being developed at Microsoft Research. it is being used to evaluate the solvability of the formulae we need for the tool to when we find a solvable solution to our vocab formulae , that's when we ask the user to decide whether such an assignment for the variables is plausible or not if the answer is yes the solver continues to the next solvable assignment if not it prunes out all assignment that would lead to this unwanted result and continues to the next variable assignment.

3.2.2 ANTLR

The ANTLR (ANother Tool for Language Recognition) parser generator is used for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, the ANTLR parser generator can build and walk parse trees.

3.2.3 QT

QT is an open source cross-platform application and UI framework for developers using C++, it is being used for creating the main GUI(Graphical User Interface) for our tool.

3.2.4 ABC

ABC is a growing software system for synthesis and verification of binary sequential logic circuits appearing in synchronous hardware designs. ABC combines scalable logic optimization based on And-Inverter Graphs (AIGs), optimal-delay DAG-based technology mapping for look-up tables and standard cells, and innovative algorithms for sequential synthesis and verification.

ABC provides an experimental implementation of these algorithms and a programming environment for building similar applications. Future development will focus on improving the algorithms and making most of the packages stand-alone. This will allow the user to customize ABC for their needs as if it were a tool-box rather than a complete tool.

3.2.5 EXPRESSO

The Espresso logic minimizer is a computer program using heuristic and specific algorithms for efficiently reducing the complexity of digital electronic gate circuits. Espresso was developed at IBM by Robert Brayton. Richard Rudell later published the variant Espresso-MV in 1986 under the title "Multiple-Valued Logic Minimization for PLA Synthesis". Espresso has inspired many derivatives.

A radically different approach to this issue is followed in the ESPRESSO algorithm, developed by Brayton e.a. at the University of California, Berkeley. Rather than expanding a logic function into minterms, the program manipulates "cubes", representing the product terms in the ON-, DC- and OFF-covers iteratively. Although the minimization result is not guaranteed to be the global minimum, in practice this is very closely approximated, while the solution is always free from redundancy.

3.2.6 GraphVIZ

Graphviz is open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. we are using this to better visualize our search space for our equivalence classes and for a better visualization of the Pruning process done by the Z3 SMT solver.

Chapter 4

User Manual

4.1 Tool and user interface

The tool provides an interface for the user. The user either loads an existing theory file or creates a new one. If the user was to load a theory file containing a set of variables and grammar for those variables (optional) and a vocabulary formulae (also optional) he simply clicks the load button and selects his file and the file will be loaded in to the GUI with the variable placed in to the variables table and grammar in to the grammar table etc. if no vocabulary formulae were provided the tool generates a vocabulary from the grammar of the theory.
On the other hand if the user wants to create the Theory he adds variables one by one and he can add the grammar statements one by one as with the constants and finally the vocab can be generated or added manually adding vocab gives you the option of adding a new vocab clause by writing it manually or adding it as a first order logic wff statement from a previous theory result. You can then choose whether to generate a quantifier free equivalent of the vocab or a vocab with Existential quantifiers.
After setting the theory file you can run it buy using the construct button. When the theory is ran, the tool generates possible queries for your variables assignments and the $truth/false$ values of the vocab based on the value of those variables at which point the user can either accept or reject the current variable assignment.
If the user accepts the variable assignment then the tool will generate the next query however if he decides to reject he can either select some vocab clauses to be the source of the rejection or variables, he does so by checking the check box in the reason column next to every variable or vocab clause.
On the case of rejection the tool rejects this assignment and all assignments leading to the conjunction of the truth/false values of the vocab clauses. EX. (V1 = F AND V2 = T AND V3 = T) and then generates a new query pruning out the unaccepted values so far.
This process goes on until the search space is exhausted and the result shows down the bottom of the window in a result box where it can be save for future work.
During this process the user and undo any decision previously made in addition to that the user has a karnaugh map displayed down the bottom of the window which gives his

a better idea of the search space and how it is getting pruned.

4.2 Command line run

Some sample examples on how the command line interface runs.

4.2.1 Running a file

```
-t | --theory filename*: pass a theory file name.
if this option is not used, a simple build in vocab is used as a demo

./runSC -t [location of theory file]
ex:
./runSC -t TheoryFiles/eina.th
```

This runs the theory file from the command line and the tool will start asking queries that the user rejects or accepts to generate the result.

4.2.2 Building a vocab from the theory

```
-b | --build-vocab: build vocab from the rules and declarations in the theory file.

./runSC -t [location of theory file] -b
ex:
./runSC -t TheoryFiles/eina.th -b
```

This flag builds the vocab from the variables constants and grammar rules stated in the file -t must be present for this to work.

4.2.3 Injecting an existential quantifier

```
-e | --inject-exists: if passed the injected quantified variables are existential.

./runSC -t [location of theory file] -e
ex:
./runSC -t TheoryFiles/eina.th -e
```

Injects an existential quantifier to the result and makes the injected quantified variables existential.

4.2.4 Quantifier free vocab

```
-f | --quantifier-free: if passed a decidable quantifier free vocab

./runSC -t [location of theory file] -f
ex:
./runSC -t TheoryFiles/eina.th -f
```

Makes the result quantifier free when the vocab is decidable.

4.2.5 Checking the file

```
-p | --proper: checks if the current .th file is valid or not

./runSC -t [location of theory file] -p
ex:
./runSC -t TheoryFiles/eina.th -p
```

Checks if the file is a proper .th file with no errors or not.

4.2.6 Running the tool with the GUI

```
-g | --GUI: use the GUI interface

./runSC -g
this runs the GUI with no theory file and one can be loaded from the GUI

./runSC -t [location of theory file] -g
this runs the GUI and loads the file directly to it
ex:
./runSC -t TheoryFiles/eina.th -g
```

Runs the theory file using the graphical user interface of the tool.

4.2.7 Help

```
-h | --help: prints this help

./runSC -h
```

The help for the tool.

4.2.8 Sample in use

```
(Note: everything between parentheses is not actually shown on the command line
but merely serves here as feedback for explanation)
./runSC -t TheoryFiles/eina.th

(this will run the theory and the user will get asked a bunch of queries)

SPCHK: checking choice...[1 1 1 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: Adding unsat core to eliminated patterns...[- - - ]
SPCHK: Adding unsat core to eliminated patterns...[- - - ]
SPCHK: choice is satisfiable.
       Is the satisfying assignment below a good model for your property?
       Notice that a specification accepts a don't care assignment.
(this notice will be issued only twice).
  a[0:int] = 4:int, a[otherwise]= 4:int
  a.size_minus_1 = 0:int
  e = 4:int
  left = 0:int
  right = 0:int
SPCHK: Answer by 'Y' to accept and 'N' to reject the assignment.
'M' to view more details from the solver about the model.
USER > y

(here the user decides to either accept the vocab or reject it so we will accept
by typing y because the query presented doesn't have any issues and should be
accepted at this point we can type m and get information about the model)

SPCHK: checking choice...[1 1 0 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: Adding unsat core to eliminated patterns...[- - - ]
SPCHK: Adding unsat core to eliminated patterns...[- - - ]
SPCHK: choice is satisfiable.
       Is the satisfying assignment below a good model for your property?
       Notice that a specification accepts a don't care assignment.
(this notice will be issued only twice).
  a[0:int] = 5:int, a[otherwise]= 5:int
  a.size_minus_1 = 0:int
  e = 4:int
  left = 0:int
  right = 0:int
SPCHK: Answer by 'Y' to accept and 'N' to reject the assignment.
```

```
'M' to view more details from the solver about the model.
USER > n

(here we decide to reject the presented query because he value of e is not in array a

SPCHK: If the rejection is due to part of the assignment,
       press 'V' to select the bad variables,
       press 'B' to select the bad vocabulary values, or
       press 'C' to continue.
USER > c

(here we can state the reason for the rejection
V: is to select which variable or variables if any made the query a bad one
B: is to select which vocab formula or formulas if any made the query a bad one
C: is to just reject the current assignment
we are going to use C now
we get this)

SPCHK: checking choice...[1 0 1 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: Adding unsat core to eliminated patterns...[- - - ]
SPCHK: Adding unsat core to eliminated patterns...[- - - ]
SPCHK: choice is satisfiable.
       Is the satisfying assignment below a good model for your property?
  a[-1:int] = 5:int, a[otherwise]= 5:int
  a.size_minus_1 = 0:int
  e = 5:int
  left = 0:int
  right = 0:int
SPCHK: Answer by 'Y' to accept and 'N' to reject the assignment.
'M' to view more details from the solver about the model.
USER > n
SPCHK: If the rejection is due to part of the assignment,
       press 'V' to select the bad variables,
       press 'B' to select the bad vocabulary values, or
       press 'C' to continue.
USER > b

(here we used B and the we get a listing of all vocab formulas to state which one or
ones is causing the problem by typing their number and -1 when we are done -2 restart
the picking and -3 ignores the choice of picking bad vocab formulas in the first plac

  + 0+ (and (and (<= 0 left) (<= left right)) (<= right a.size_minus_1)) is true
  + 1+ (and (<= left i) (<= i right)) is false
  + 2+ (= e (select a i)) is true
SPCHK: Type the ids of the vocab formulae that are the reason for rejecting the
```

```
satisfying example.Type '-1'  to finish, '-2' to restart, and '-3' to exit bad vocab
formulae option.
USER > 1
USER > -1

SPCHK: The reason for rejecting the model is the partial assignment
0
+ 1+ (and (<= left i) (<= i right)) is false
SPCHK: please confirm by typing 'Y'. Restart by typing 'R'. Ignore and continue by
typing any other key.
USER > y

(then we are asked to confirm the choice by typing Y for yes or R to restart
in the case of restart we will just get the list of vocab formulas again and we
just state their numbers again. now we get the new query.)

SPCHK: ignore bad partial assignment: [- 0 - ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 0 0 ]
SPCHK: checking choice...[0 1 1 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: Adding unsat core to eliminated patterns...[- - - ]
SPCHK: Adding unsat core to eliminated patterns...[- - - ]
SPCHK: choice is satisfiable.
       Is the satisfying assignment below a good model for your property?
  a[-1:int] = 3:int, a[otherwise]= 3:int
  a.size_minus_1 = (define a.size_minus_1 Int)
  e = 3:int
  left = -1:int
  right = 0:int
SPCHK: Answer by 'Y' to accept and 'N' to reject the assignment. 'M' to view more
details from the solver about the model.
USER > n

(notice that left is an index in the array therefor it shouldn't be -1 so we want to
reject this assignment for this variable this time we will use V and get)

SPCHK: If the rejection is due to part of the assignment,
       press 'V' to select the bad variables,
       press 'B' to select the bad vocabulary values, or
       press 'C' to continue.
USER > v

(now we can list the bad variables for example we don want the size to be )

Please indicate the variables that present unsatisfying values by name.
Enter the names one at a time and when you're done, enter '0'.
```

```
left
0

(the zero is to say we are done with listing variables. and we then get the final
result after the tool has exhausted all possible classes of queries and does a bunch
of computations. and we get this)

SPCHK: formula after esprersso simplify.
  INORDER = (((0 <= left) and (left <= right)) and (right <= a.size_minus_1)) ((left
and (i <= right)) (e = a[i]);
OUTORDER = Spec;
Spec = exists i.  (((0 <= left) and (left <= right)) and (right <= a.size_minus_1)) *
((left <= i) and (i <= right)) * (e = a[i]);

(now to check just if the file will properly run we do the following)

./runSC -t TheoryFiles/eina.th -p

(here we will see some computation but the last line should state)

The File TheoryFiles/eina.th is a Proper file

(if the file is not we will get some error as to where the problem might be)
```

4.2.9 Disadvantages of the CLI

After some testing we saw a set of disadvantages with the CLI.

1. No constant feedback around the values of the vocabulary formulas.
2. Generating a new theory requires the user to write a theory file outside the software using some third party text editor.
3. The user needs to know the syntax of the theory text file.
4. If a mistake were done on some step the run needs to be repeated from scratch.
5. Little indication over which equivalence classes have been pruned and whats left from the search space except user intuition.
6. Not seeing the results of the vocab at all times which slows down the users decision of accepting or rejecting.

4.3 The graphical user interface (GUI)

The user interface is as follows seen in figure 4.1

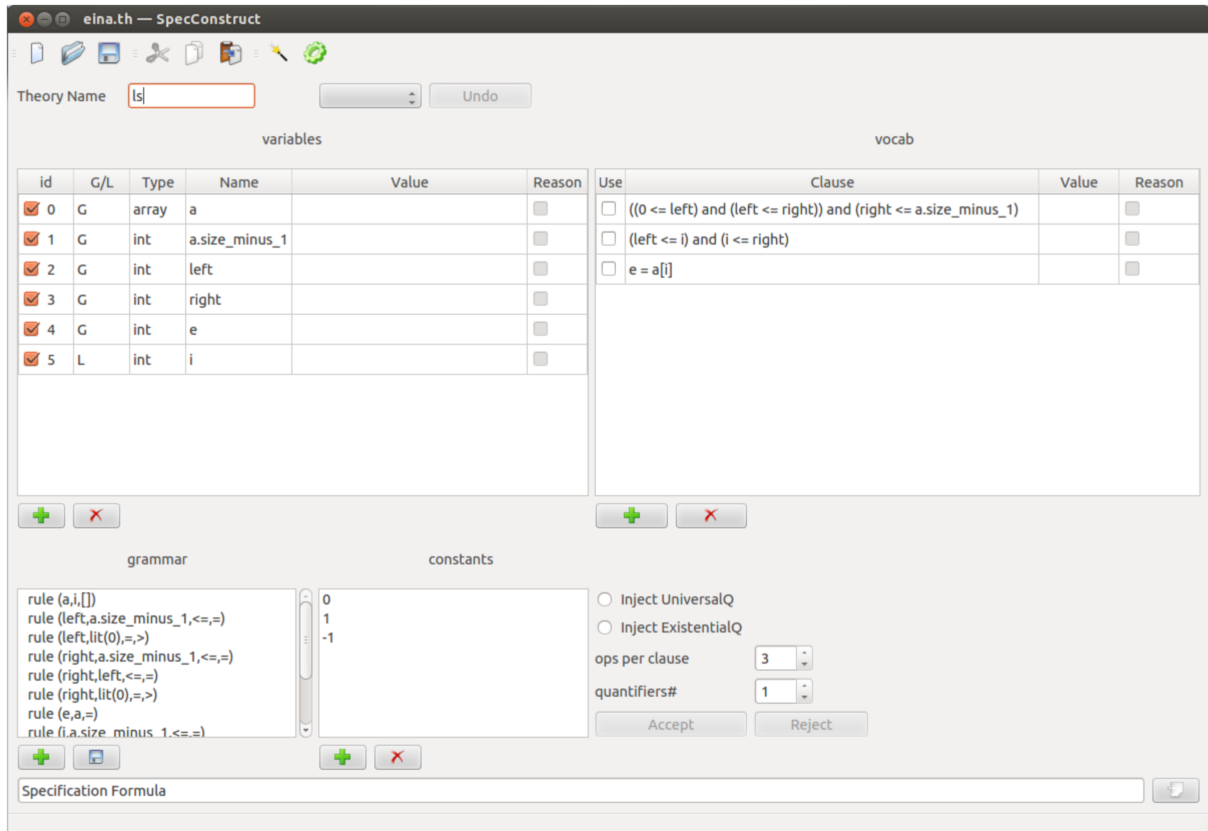


Figure 4.1: a snapshot of the general interface

4.3.1 Theory name text box (figure 4.2)

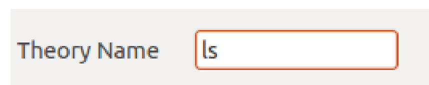


Figure 4.2: the theory name box at the top left corner

The theory name box at the top left corner has the name of the theory currently being run or built and if the theory build were to be saved it is the name it will take.

4.3.2 The variables table (figure 4.3)

variables

id	G/L	Type	Name	Value	Reason
0	G	array	a		
1	G	int	a.size_minus_1		
2	G	int	left		
3	G	int	right		
4	G	int	e		
5	L	int	i		

Figure 4.3: the variables table is the top left table

The variables table contains all the variables for the theory file being generated or run and it's divided in to:

ID
: First column has The ID is basically the ID of the current variable.

Global/Local
: The G/L in the second column is a feedback on whether the variable at hand is a global variable (G) or a local variable (L).

Type
: The Type field in the third column is the type of variable wheather it is an integer (int), boolean (bool), integer array (array), or a boolean array (barray).

Name
: Fourth column is the field for the variable's name.

Value
: The fifth column has the value of the variable it will remain empty while the tool is in build mode, values will be assigned to variable after constructing and running the theory where the user will start being asked for acceptance or rejection of queries based on the values given to each variable (a small example of the value when the tool is running can be seen in figure 4.4).

G/L	Type	Name	Value	Reason
	array	a	a[0]=5, a[otherwise]=5	☐
	int	a.size_minus_1	-1	☐
	int	left	-1	☐
	int	right	0	☐
	int	e	4	☐

Figure 4.4: the variables table while a theory is running

Reason

The sixths and final column is also only used when the theory is being run. it is simply a checkbox where the user checks which variables are the reason for rejection the current query if desired and if the query is to be rejected.

4.3.3 Add variable button (figure 4.10)

Figure 4.5: the variable add button having a plus sign located right under the variables table

This button adds a new variable to the theory by opening a new window where we can specify the name and type of variable with specifying whether its local or global as can be seen in figure 4.6.

Add New Variable
Name var3
Type int
Global
Local
Add

Figure 4.6: the window for adding a new variable

The Name Field

The name field box is where the variable name will be written.

Variable Type Drop Down List

The variable name drop down list is where we pick the type of the variable whether

it is an integer (int), boolean (bool), integer array (array), or a boolean array (barray) as can be seen in figure 4.7.

Global/Local Radio Button
Radio buttons to determine whether the variable is global or local (Global by Default).

Add button
The add button inside the add new variable window directly adds the variable to the end of the variables table.

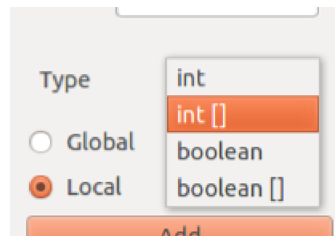


Figure 4.7: the variable type dropdown list

4.3.4 Delete variable button (figure 4.8)

Figure 4.8: the variable table delete button having an x sign located under the variables table

The variable delete button deletes the selected variable in the variable table, to select a variable simply click on any item in its row in the variables table and it will be selected (when a cell is selected it will be highlighted in orange).

4.3.5 The grammar table (figure 4.9)

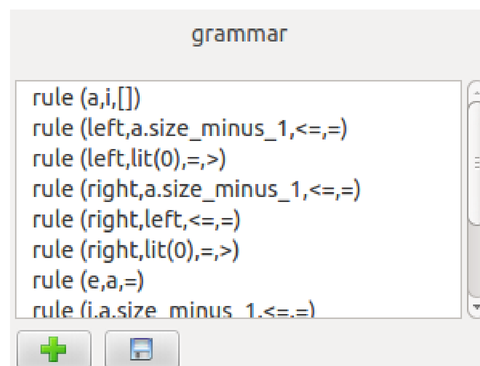


Figure 4.9: the Grammar table is the bottom left table

The grammar table contains all the grammar rules for the theory file to help generate a vocabulary if needed.

4.3.6 Add grammar button (figure 4.10)

Figure 4.10: the grammar add button having a plus sign located right under the grammar table

This button adds a new grammar to the theory by opening a new window where we can specify the name of a variable with specifying a relation to another variable or literal as can be seen in figure 4.11 (types of relations are mentioned below).

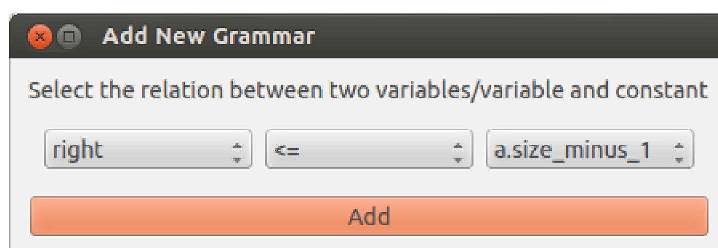


Figure 4.11: the window for adding a new grammar

Left Variable Drop Down List

The dropdown list on the left is for specifying the first variable, it will display all the variables of the theory that we have in the variables table shown in figure 4.3.

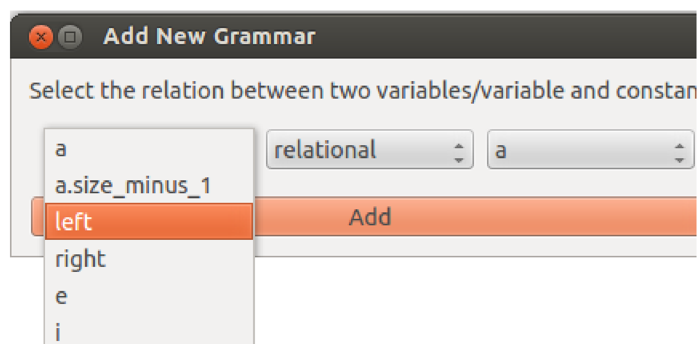


Figure 4.12: the left variable dropdown list while adding a grammar

Relation Drop Down List

The Middle dropdown list is to specify the type of relation between the first and second variable or literal, it will display all possible relations between the two left and right operands (different types of relations are mentioned below).

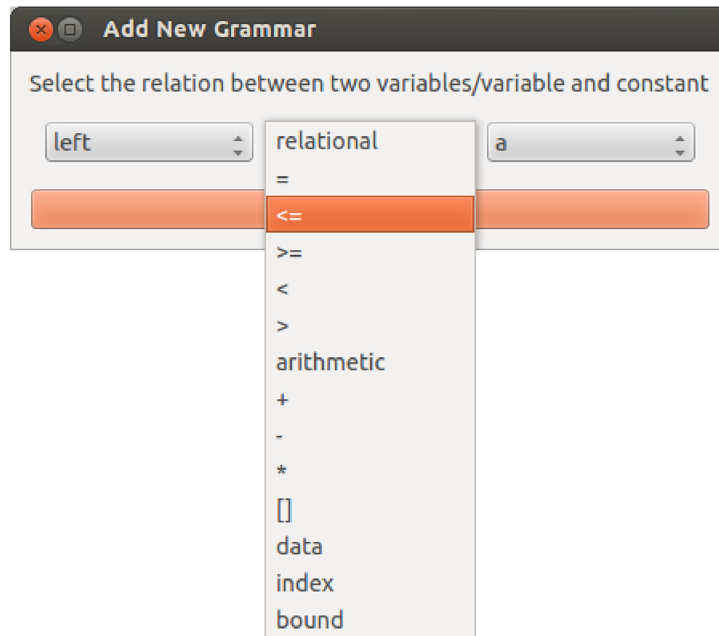


Figure 4.13: the relation dropdown list

Relational

Relational is basically all the comparison operators (=,<=,>=,<,>). it basically creates grammar for all possible comparison operators between the two operands

=

Equals is the normal equality operator.

<=

Less Than or equal is the normal less than or equal comparison operator.

>=

Greater Than or equal is the normal greater than or equal comparison operator.

<

Less Than is the normal less than comparison operator.

>

Greater Than is the normal greater than comparison operator.

arithmetic

Arithmetic is the math operators plus (+), minus (-), and multiplication (*).

+

The plus sign is the normal addition operator.

-

The minus sign is the normal subtraction operator.

*****

The multiplication sign is the normal product operator.

index

Is a relation to specify that an integer is an index in the array.

bound
Is a relation to dictate a variables as being within the bounds of an array,

Right Variable Drop Down List
The dropdown list on the right is for specifying the second variable or literal, it will display all the variables we have added to the theory in the variables table seen in figure4.3 and all the constants we have added to the theory in the constants table seen later in figure 4.16 .

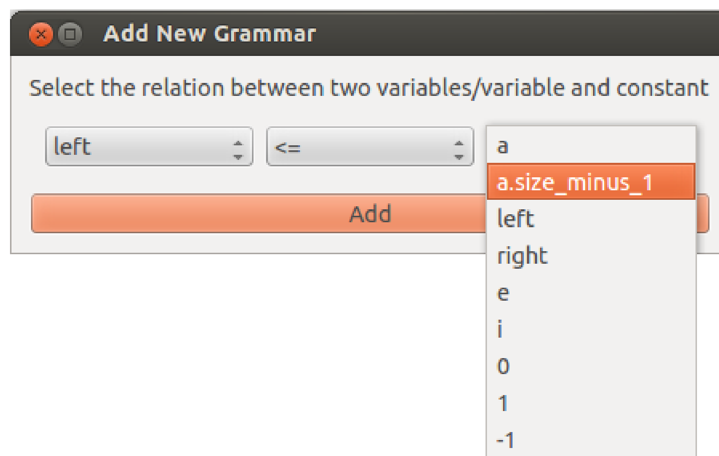


Figure 4.14: the right variable or literal dropdown list while adding a grammar

4.3.7 Delete grammar button (figure 4.15)

Figure 4.15: the grammar table delete button having an x sign located under the grammar table

The grammar delete button deletes the selected grammar in the grammar table, to select a grammar simply click on any grammar formula in the grammar table and it will be selected (when a row is selected it will be highlighted in orange).

4.3.8 The constants table (figure 4.16)

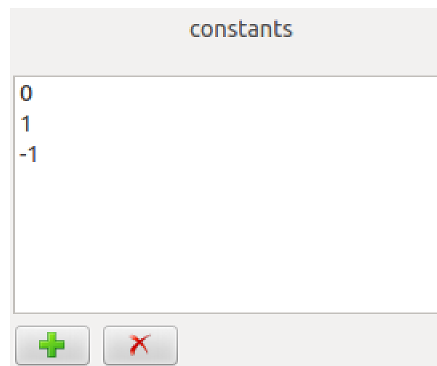


Figure 4.16: the Constants table is the bottom table to the right of the grammar table

The constants table contains the constants needed to use with the theory to check for bounds among other things.

4.3.9 Add constant button (figure 4.17)

Figure 4.17: the constant add button having a plus sign located right under the constants table

This button adds a new constant to the theory by opening a new window where we can specify the constant to be added as can be seen in figure 4.18.

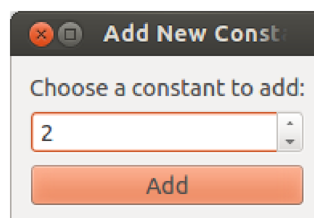


Figure 4.18: the window for adding a new constant

4.3.10 Delete constant button (figure 4.19)

Figure 4.19: the constant table delete button having an x sign located under the constants table

The constant delete button deletes the selected constant in the constants table, to select a constant simply click on any constant in the constants table and it will be selected (when a row is selected it will be highlighted in orange).

4.3.11 The vocab table (figure 4.20)

vocab

Use	Clause	Value	Reason
☐	((0 <= left) and (left <= right)) and (right <= a.size_minus_1)		☐
☐	(left <= i) and (i <= right)		☐
☐	e = a[i]		☐

Figure 4.20: the Vocabulary table at the top right

The vocabulary table has all the vocabulary formulas generated by the tool or stated by the user it is on the top right and its divided into:

Use
: The first column of the table which is basically a check box to decide whether to include the clause in the theory or not.

Clause
: The second column, the clause column which contains the vocabulary formula.

Value
: The third column which has a value that displays true or false while a theory file is running based on and evaluation of the formula in column two according to the values of its variables in the variables table fifth column an example of how the results are displayed are shown in figure 4.21.

Reason
: The Fourth and final column in the vocabulary table has a checkbox for each row as

in each vocabulary formula, the user checks the boxed while the tool is running in case a query was suggested and he decides to reject it based on specific vocabulary evaluation or combination of evaluations, the checkboxes can be checked before rejecting the query to prune out any further queries resulting in the same picked combination of vocabulary evaluations (can be seen running in figure 4.21).

Use	Clause	Value	Reason
☐	((0 <= left) and (left <= right)) and (right <= a.size_minus_1)	true	☐
☐	(left <= i) and (i <= right)	true	☐
☐	e = a[i]	false	☑

Figure 4.21: the Vocabulary table while a theory is running

4.3.12 Add vocabulary button (figure 4.22)

Figure 4.22: the vocab add button having a plus sign located right under the vocab table

This button adds a new vocabulary formula to the theory by opening a new window where we can specify the method we want to use for adding the new vocabulary formula as can be seen in figure 4.23

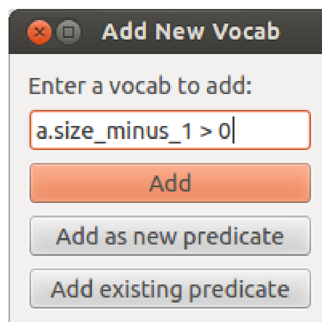


Figure 4.23: the window for deciding the method to add the vocab in

Text Box

The text box seen in figure 4.23 is where we write the formula we want to add to the vocabulary table.

Add Button

The add button in the window seen in figure 4.23 adds the formula typed in to the textbox above it to the vocabulary table.

Add as new predicate Button

The add as new predicate button opens a window to add a vocabulary formula as a functional predicates as can be seen in figure 4.24.

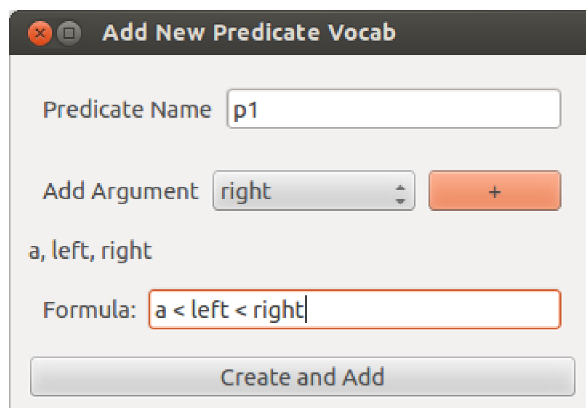


Figure 4.24: the add new predicate window

Add existing predicate Button

The add as existing predicate button opens a window to load a vocabulary formula from another file.

4.3.13 Delete vocabulary formula button (figure 4.25)

Figure 4.25: the vocabulary table delete button having an x sign located under the vocabulary table

The vocabulary delete button deletes the selected vocabulary formula from the vocabulary table, to select a formula simply click on any item in its row in the vocabulary table and it will be selected (when a cell is selected it will be highlighted in orange).

4.3.14 The accept and reject buttons (figure 4.26)

Figure 4.26: the accept and reject button on the bottom of the window

The Accept and Reject buttons are inactive while building or loading a theory file and are only active when a theory is running their purpose is for the user to decide whether to accept the current query suggestion or to reject it based on the desired result.

4.3.15 Inject universal/existential quantifier radio button (figure 4.27)

Figure 4.27: the radio buttons to inject universal or existential quantifiers

Above the Accept and reject button ...

4.3.16 Number of operations per clause (figure 4.28)

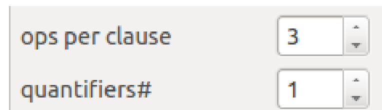


Figure 4.28: counter for the number of operations per clause

Right above the accept and reject buttons there is the counter for the number of operation per clause bound and number of quantifiers for the theory.

4.3.17 Undo drop down list (figure 4.29)

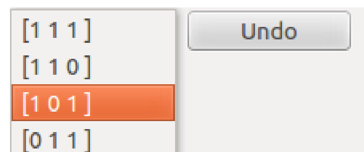


Figure 4.29: the undo drop down list located to the right of the theory name

Right to the next of the theory file name there's a drop down list that has the previous vocabulary states that the user either accepted or rejected and by clicking on any of the states and then the undo button the theory will reset to that state deleting the pruned equivalence classes after that state.

4.3.18 Formula result box (figure 4.30)

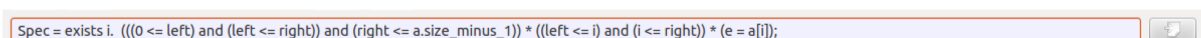


Figure 4.30: the result box of the theory

At the lower part of the interface lies the result box that shows all intermediate results as the user accepts and rejects formulas and after all the pruning is done and a final answer is reached displays the final answer.

To the right of the formula box lies a button to save the result to a file, this result can be saved to a file where it can be later on loaded as a separate vocabulary and placed in to some new theory, this can be done through the add existing predicate button in section 4.3.12 as shown in figure 4.24.

4.3.19 Construct theory (figure 4.31)

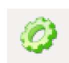


Figure 4.31: the button on the top of the window to construct the theory from whatever variables, grammars and vocabs that exist.

The construct theory button basically the run button is the button that generates the theory internally from the variables, grammar, constants and vocabulary formulas given by the user, if the vocabulary is non existent it is generated from the grammar, after this process the queries are given in the tables where the user will start accepting and rejecting results and thereby pruning the search space.

4.3.20 New file button (figure 4.32)

Figure 4.32: the button on the top of the window to generate a new theory file

The new file button basically neglects whatever is in the GUI and clears everything so that the user can build a new theory.

4.3.21 Save file button (figure 4.33)

Figure 4.33: the button on the top of the window to save the existing theory constructed in the GUI

The save theory button Saves the variables, grammar, vocabulary formulas, constants, and number of operation per clause, and number of quantifiers in the theory file format.

4.3.22 Load file button (figure 4.34)

Figure 4.34: the button on the top of the window to load a theory file either previously saved from the GUI or written by the user from scratch

The load button loads the theory file to the GUI by placing the variables, grammars, constants, number of operations per clause, number of quantifiers, and vocabulary values in the GUI.

4.3.23 Undo feature (figure 4.35)

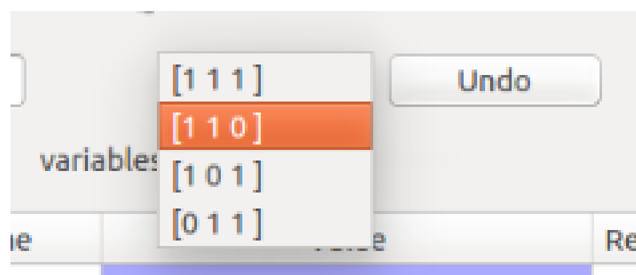


Figure 4.35: the undo button with the drop down list of the states passed

The undo button displays a set of boolean numbers each representing a valuation on the vocab previously queried to the user. The user picks any of them and then clicks the undo button at which point the tool will regenerate a valuation for that vocabulary assignment and all states after that would be deleted, this can be seen in figure 4.35.

4.3.24 Karnaugh map (figure 4.36)

Figure 4.36: Karnaugh map on the bottom left representing all the state space

Karnaugh map displays the state space that needs to be searched where it has color coding with green representing accepted equivalence classes, red representing the rejected ones and gray representing the ones not yet covered. The Letters on the left and top represent the names of the vocabulary clauses, where each vocabulary clause is given a name and its name is represented by the boolean index in the karnaugh map.

Chapter 5

Case Studies

5.1 Sample use case scenario

1. We can start by stating the theory name.
2. We then add the list of variables with their types names and scopes.
3. We then insert the constants.
4. We then insert the grammar rules.
5. Optionally we insert vocabulary formulas.
6. One can specify a number using a counter to state how many quantifiers should be in the theory.
7. We can save the theory generated where the tool will generate a .th file.
8. We can skip all the previous steps and just load a previously existing theory file.
9. We can construct the theory and the tool will start generating the queries.
10. The vocab is used for the queries if a vocabulary doesn't exist it is generated from the grammar formulas.
11. On generation we can see the values given to all the variables next to the respective variables name.
12. There are the vocabulary formulas in which feedback about which of them are now with a false value and which of them is of a true value and that speeds up the user's decision a lot since we don't have to manually think about it while looking at them, the user can pick one or multiple vocabulary formulas to be the reason for rejection.
13. A name to the vocabulary formula is given for use in the karnaugh map.
14. We can accept the following suggestion or reject it.

15. If he accepts it will be added to the list of accepted equivalence classes.

16. In the case of rejection there are check boxes next to all the variables and vocab formulas that can help us decide whether it is the reason we rejected this query. It Will be included in to the rejected equivalence classes, when this is done it prunes out or rejects all equivalence classes that have the same assignment for the selected variable or variables or vocabulary formula or formulas.

17. Then the next query is asked and so on until all equivalence classes are covered.Then a final result can be presented.

18. During this process the user can undo a step or number of steps and go back to a previous query. Where he will continue from there. This helps due to the error prone nature of the user, and one might need to undo after noticing that he had done an error.

19. This can be noticed in two way using two methods off feedback firstly at all times during the run a karnaugh map is shown at the bottom of the widow that contains a Boolean name for all vocabulary clauses to give the user a partial idea of what has been pruned so far and how many queries are left, the karnaugh map basically has each one of its grids representing one equivalence class in our search space.

20. The karnaugh map displays a red green or gray circle in each grid implying that this equivalence class is accepted for green, is rejected for red and still not checked for gray.

21. The second feedback method is an intermediate final result of our theory and from that we can read it and know if we are on the right track or not.

22. After that is all over we should view the final result at the bottom of the screen and next to a button that allows us to save the result to a file.

In this chapter, we present case studies of specification construction.

1. ”inorder.th” theory file.

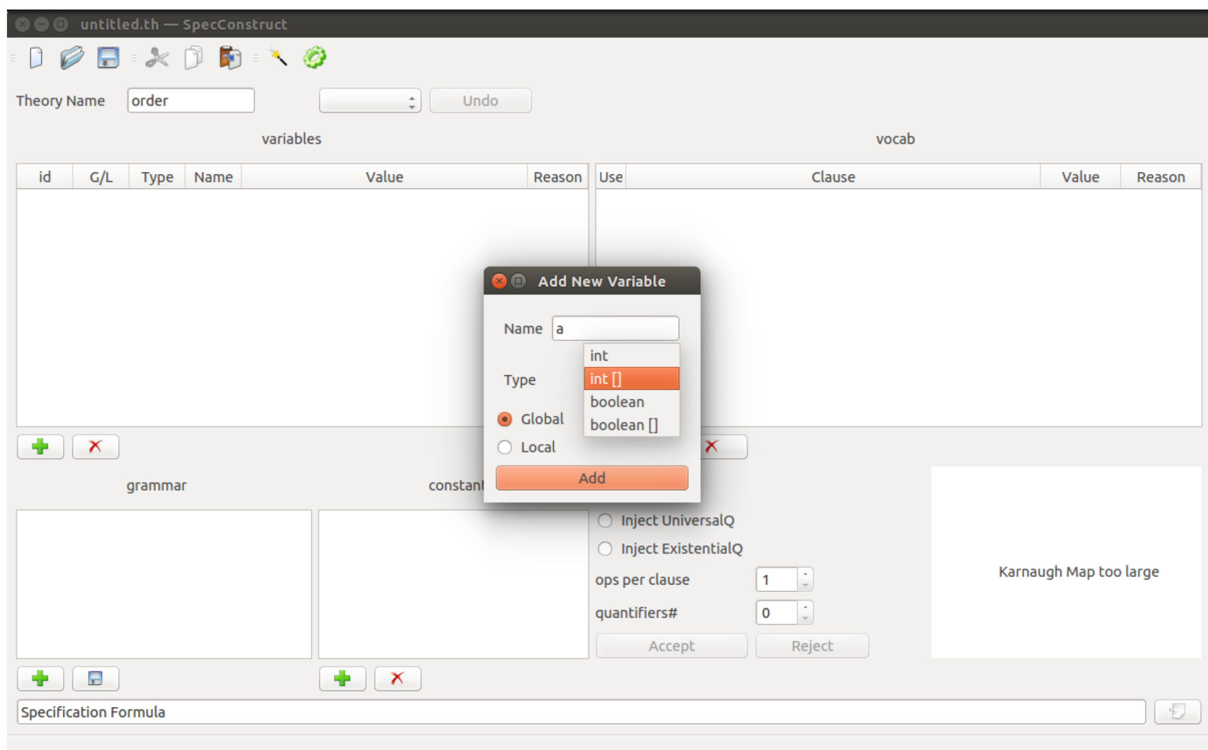
2. ”eina.th” theory file.

For each case study we generate a theory file using the GUI save it as a .th file and then run it using the construct button.

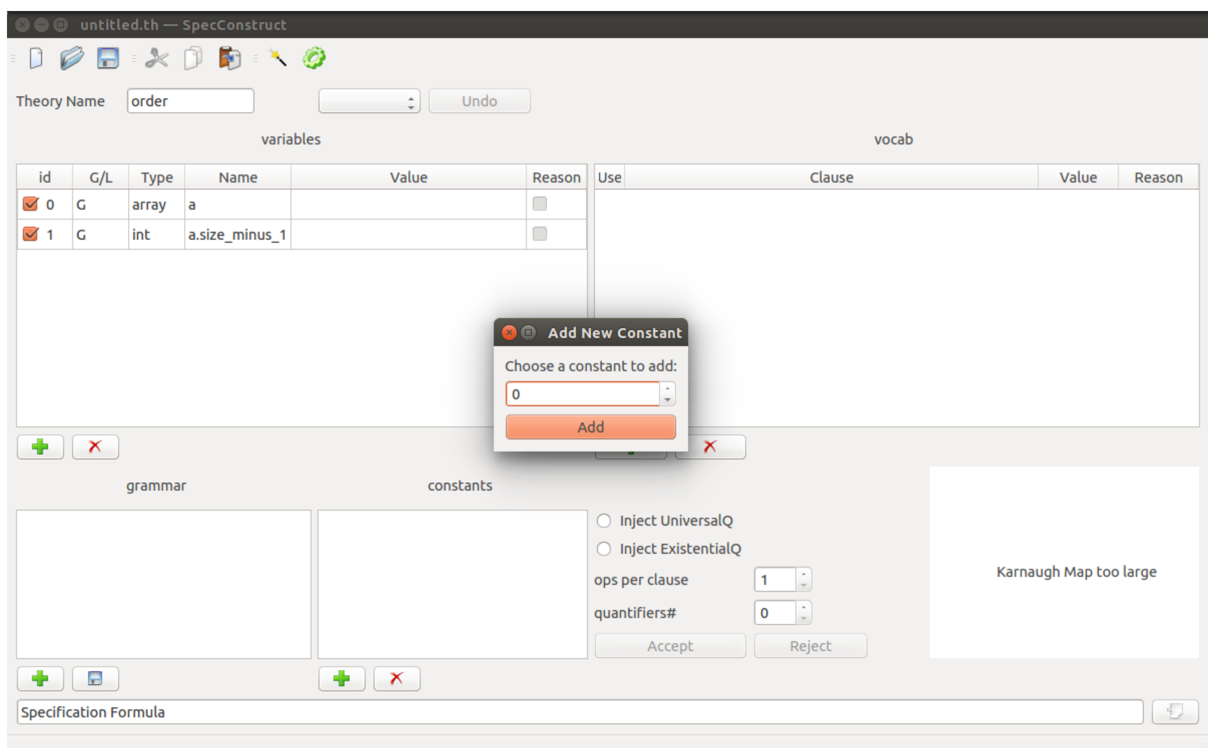
5.2 ”inorder.th” theory file

For the array being in order example we use the automatic vocabulary generation and run it from that.

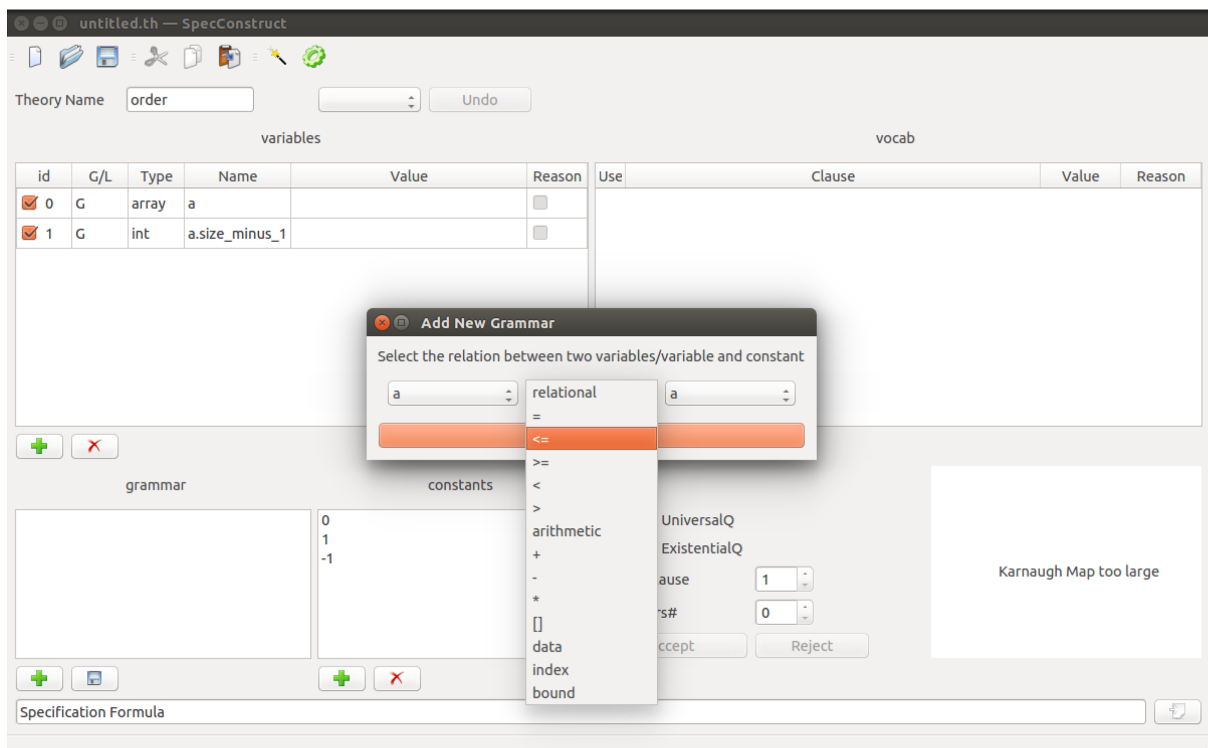
We begin by giving it a name then listing our type theory.



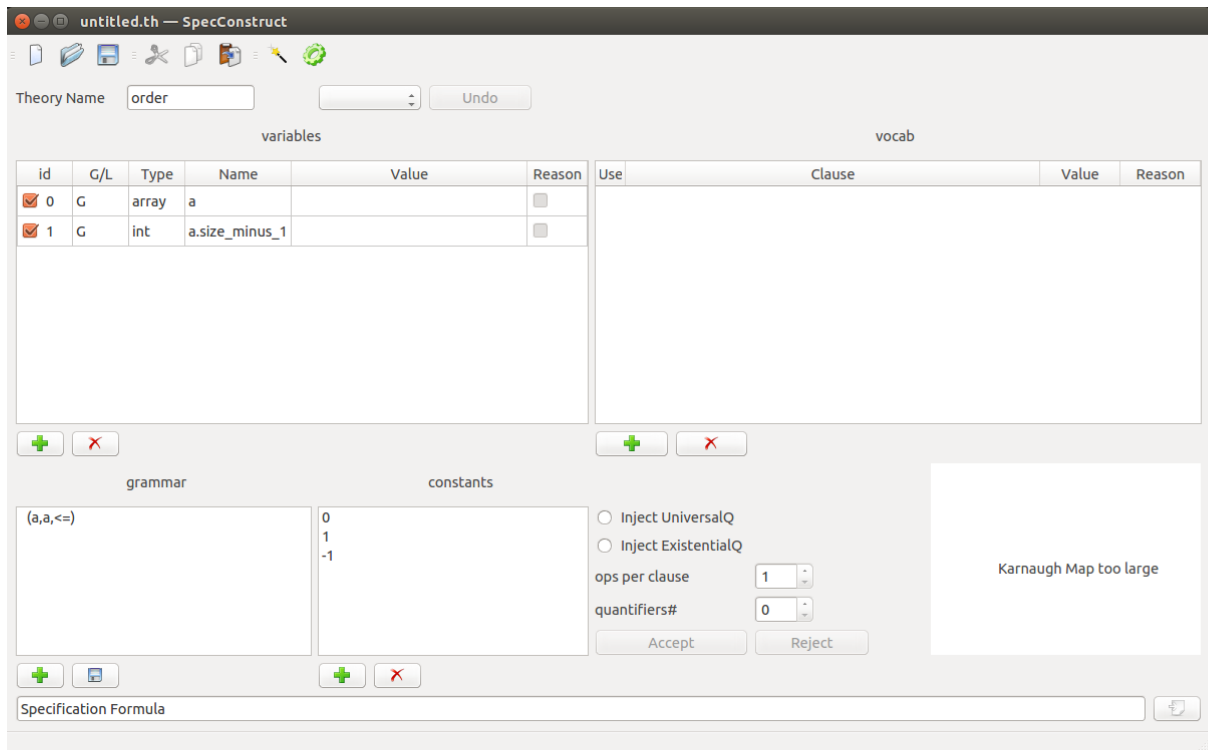
We name the variable then pick the type and check whether the variable is global or local. We then proceed to add our constants.



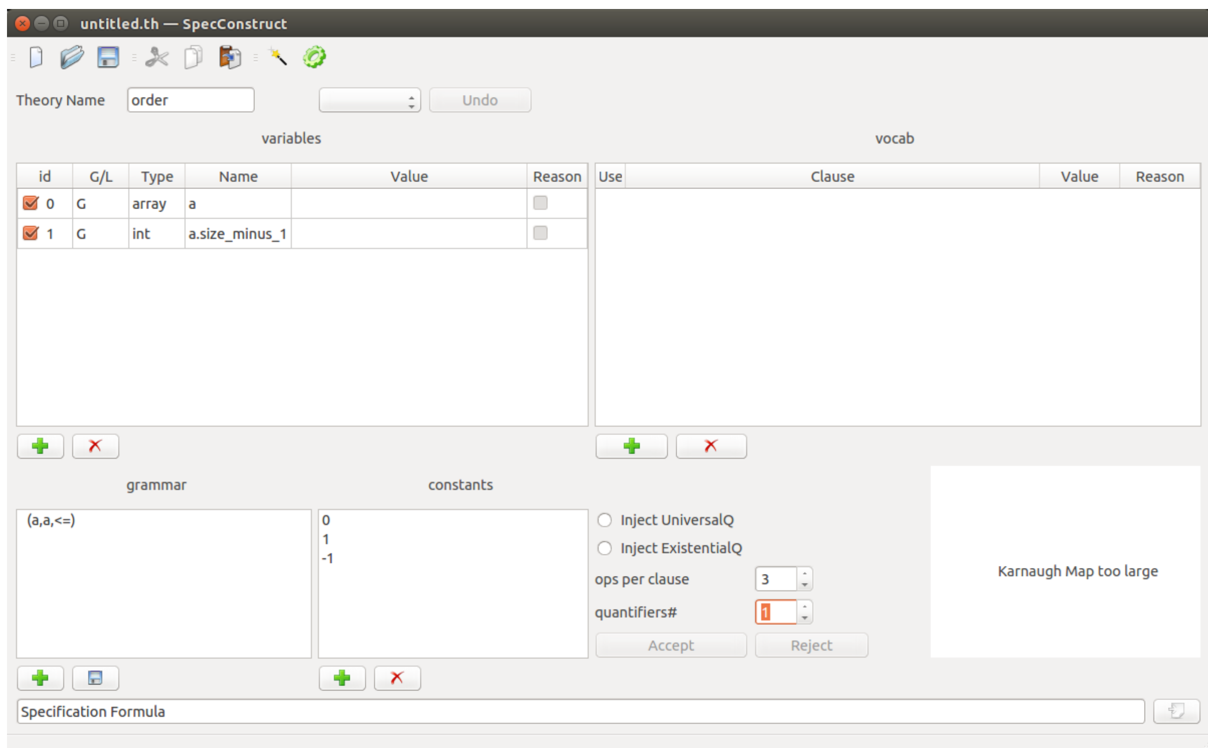
Since we are only interested in the relation of values in the array that are less than others we define a grammar relation as such by picking the array from the first list and the relation from the second and again the array in the first, and this would define our equivalence classes.



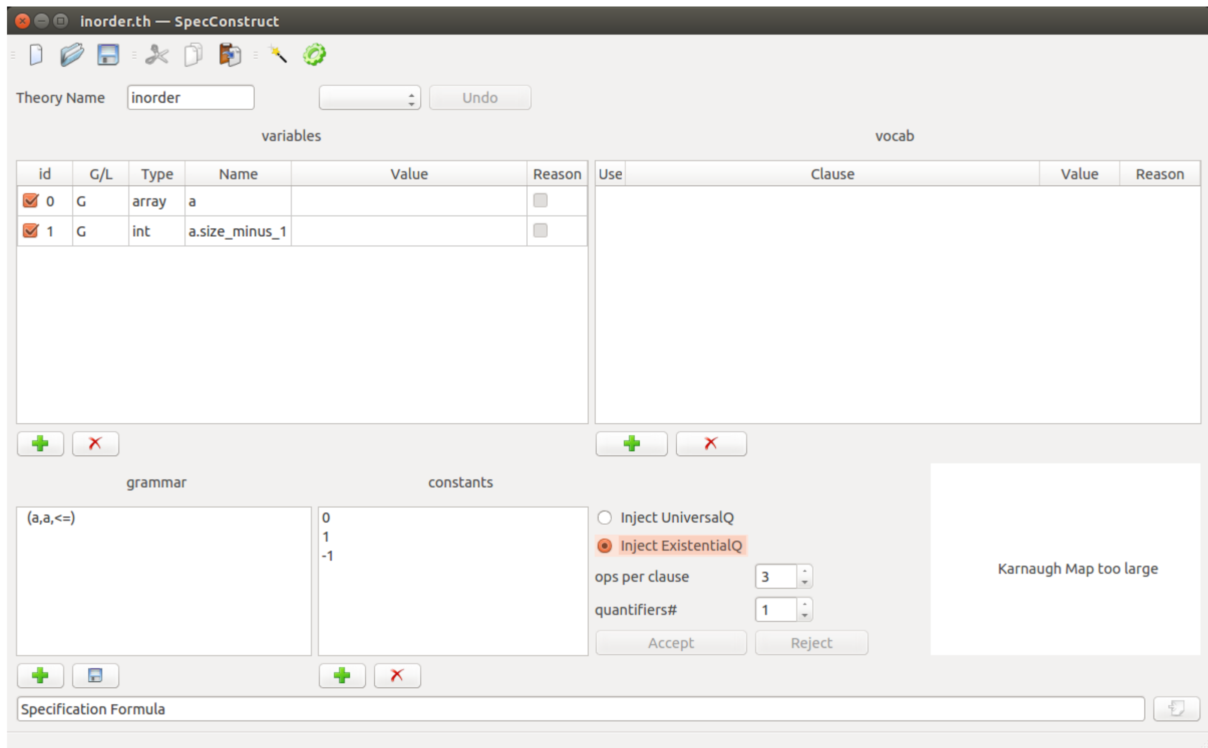
Now all that is left is to decide how many operations to do on each clause and how many quantifiers to inject so we do that.



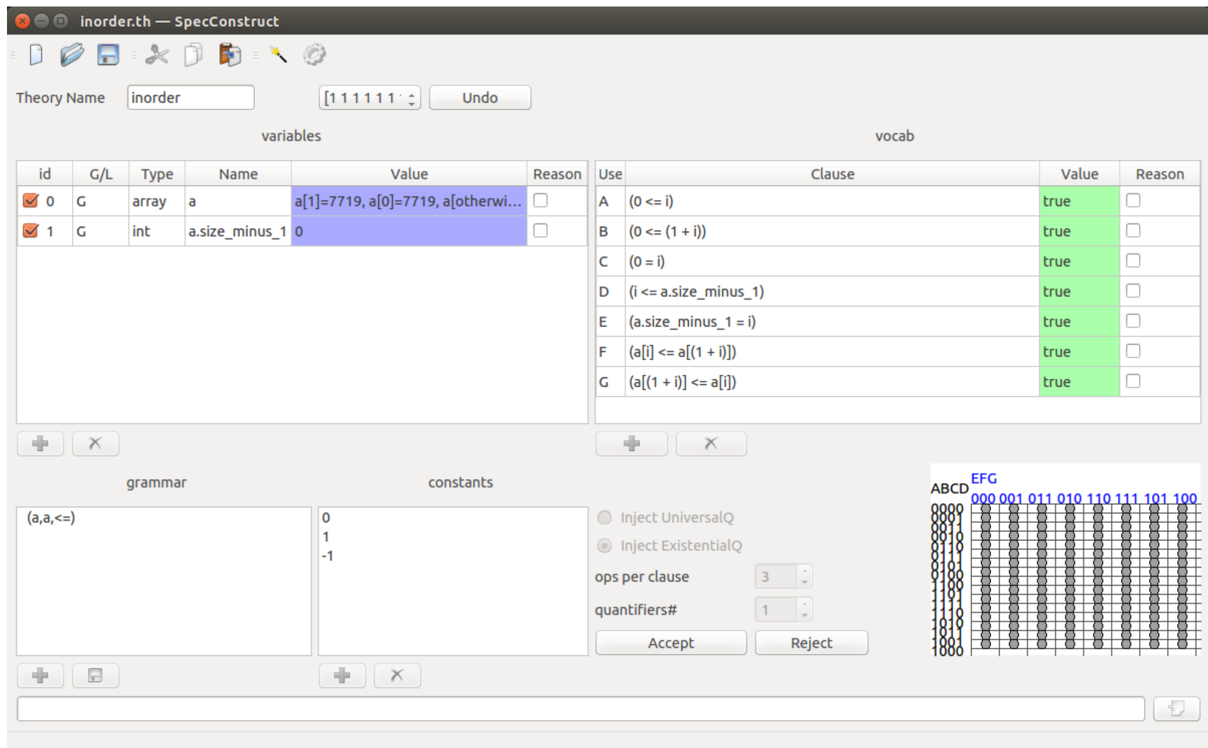
Now that our theory is over we want to construct it to get a result.



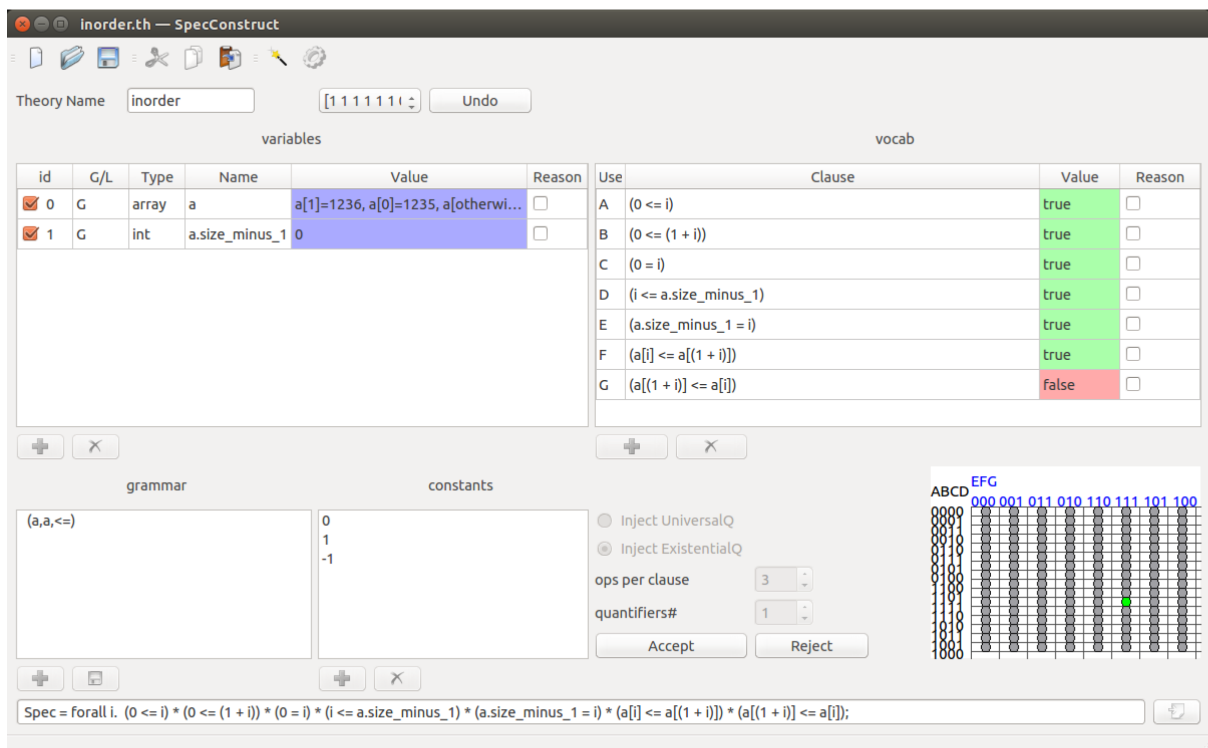
We do that by clicking on the construct button (the green gear).



Now we can see that the tool generated vocabulary clauses which will have different valuations representing our equivalence classes. In addition to that now we can see or variables assignments in blue in the variables table and our vocabulary values in the vocab table are in Green and red displaying whether they have a false or a true value on this query. We can notice as well the karnaugh map generated which is all gray at this point since we still have not accepted nor rejected any equivalence class.



We accept the current valuation here since all the vocab values are true and it happens to be satisfying to our intentions.



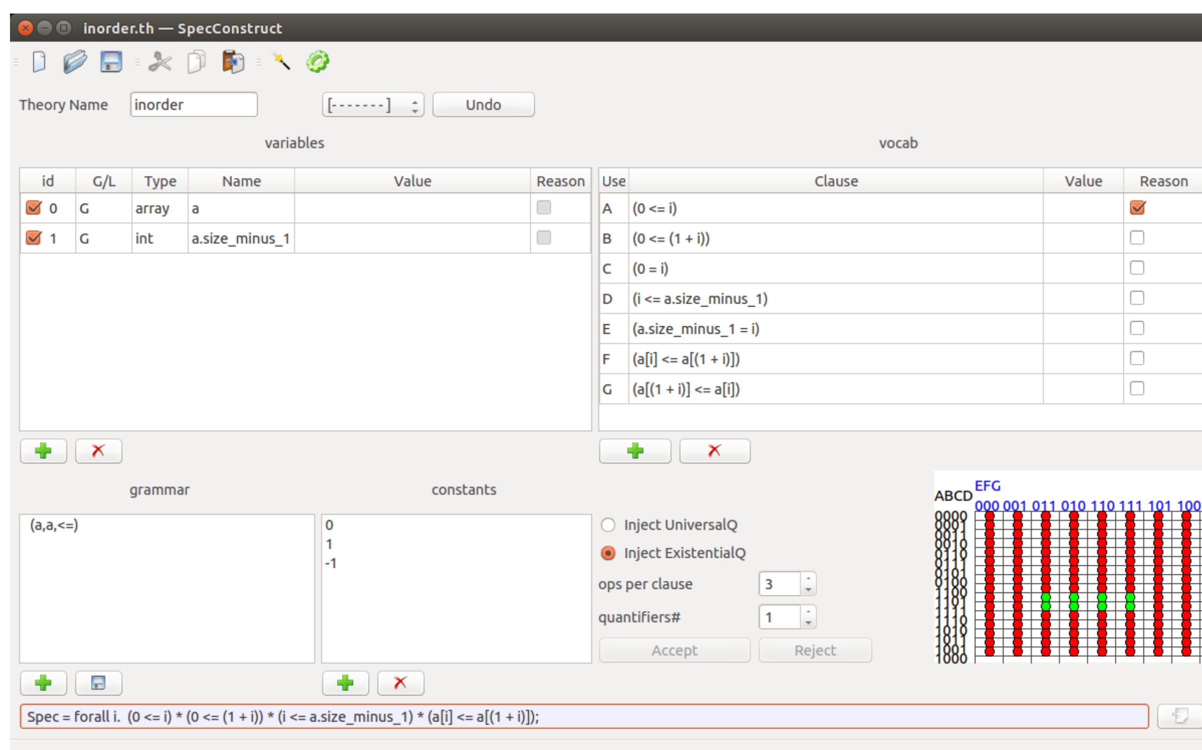
Notice we have a green dot in the karnaugh map which represents the accepted equivalence class.

Now we get the last vocab value as a false which is acceptable considering the variable assignments.

In this case we cannot have the F clause be false at any point so we want to reject the current assignment, and for pruning purposes we base our rejection on the F clause being false so we check the checkbox in reason and reject.

We can see that a huge part of the search space has been pruned out by having a red dot in their corresponding cell.

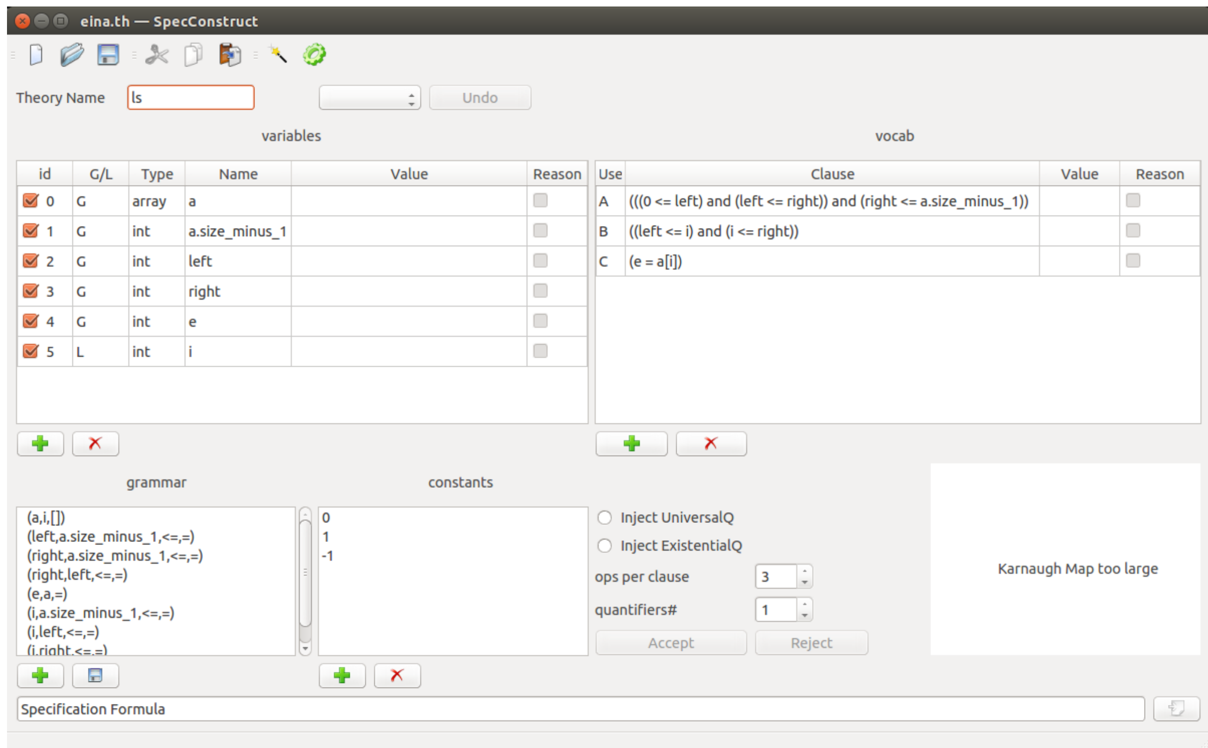
We continue with or method until everything has been pruned or accepted or rejected.



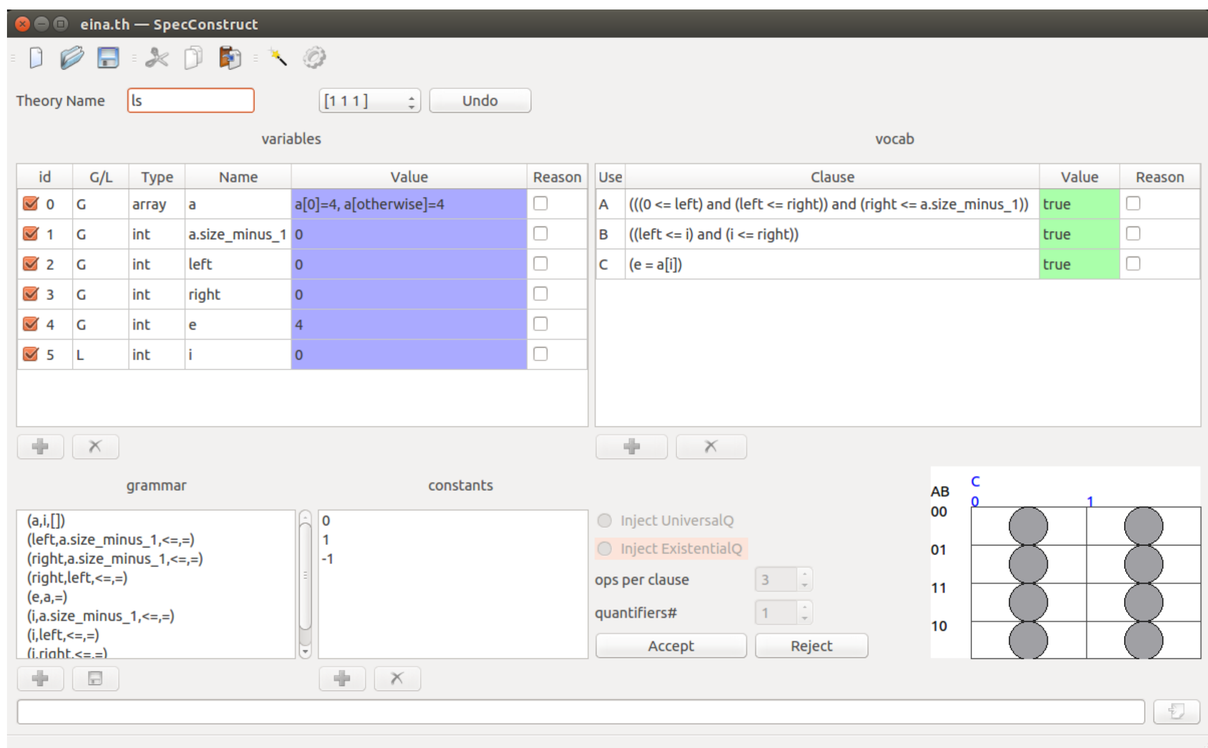
We can see our final result displayed in the bottom of the window were we can save it and use it in another theory.

5.3 "eina.th" theory file

The "eina.th" theory file is a smaller example where we provide our type theory and state the vocab clauses manually. Since the user states them manually we can speculate that we will not get a huge search space.



When we run it we can see from the karnaugh map that our search space is relatively small.



We accept the first assignment since the assignment is satisfying.

We need to reject this assignment based on the false vocab clause.

Similar to "inorder.th" theory file example we pick the false clause as the reason for the rejection.

eina.th — SpecConstruct

Theory Name: ls   [1 0 1]   Undo

variables

id	G/L	Type	Name	Value	Reason
0	G	array	a	a[-1]=5, a[otherwise]=5	
1	G	int	a.size_minus_1	0	
2	G	int	left	0	
3	G	int	right	0	
4	G	int	e	5	
5	L	int	i	-1	

vocab

Use	Clause	Value	Reason
A	(((0 <= left) and (left <= right)) and (right <= a.size_minus_1))	true	
B	((left <= i) and (i <= right))	false	✓
C	(e = a[i])	true	

grammar

(a,i,[])
(left,a.size_minus_1,<=,=)
(right,a.size_minus_1,<=,=)
(right,left,<=,=)
(e,a,=)
(i,a.size_minus_1,<=,=)
(i,left,<=,=)
(i,right,<=,=)

constants

0
1
-1

Inject UniversalQ
Inject ExistentialQ
ops per clause 3
quantifiers# 1
Accept Reject

AB C
0 1
00
01
11
10

Spec = exists i. (((0 <= left) and (left <= right)) and (right <= a.size_minus_1)) * ((left <= i) and (i <= right)) * (e = a[i]);

The same as above.

More pruning...

We can see our final result displayed in the bottom result box after all the search space

has been covered. This run can be completed with only 4 queries and takes less than a minute to solve using the GUI.

Chapter 6

Proposed Work

6.1 Future work

More work can be done to improve on this current method, We can provide a better view of the undo choices other than a boolean string or also include the variable assignments that were given at that state. Coloring only the cells whose value have changed is better than entire values columns because the changes are what the user is interested in. In addition to that further automatic simplification of the specification formula generated (getting rid or redundant parts) should be done for a simpler more readable result. Karnaugh map should display squared grouping of the different grids for better feedback on generated formula (show visually how the karnaugh map is grouping out equivalence classes). The karnaugh map can also be interactive in a way where the user clicks on a grid element and he can get an assignment for that current vocab valuation which would further speed up the pruning process assuming the user has a good idea of his end result. Also more tests should be run, to thoroughly debug the implementation. Finally conduct more case studies, in particular for specifications involving recursive functions.

6.2 Conclusion

We have shown the importance of accurate specification, and we have shown how to generate one based on the users intuition in an easier interactive way where the user generates it incrementally. Along with this we have seen the command line interface (CLI) which provides an applicable implementation of this method of writing a specification with *quantified variables* and we have seen where it is lacking and its inconvenient parts. We saw how a graphical user interface (GUI) could solve a lot of the issues involving inconveniences with the command line interface as well as provide better feedback and more error tolerance.

Bibliography

[1] Paul C. Attie, Fadi A. Zaraket, Mohamad Noureddine, and Farah El-Hariri. Specification construction using behaviours, equivalences, and SMT solvers. *CoRR*, abs/1307.6901, 2013.

[2] R.K. Brayton, A.L. Sangiovanni-Vincentelli, C.T. McMullen, and G.D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[3] A. Mishchenko, N. Eén, R.K. Brayton, M.L. Case, P. Chauhan, and N. Sharma. A semi-canonical form for sequential AIGs. In *DATE*, pages 797–802, 2013.