Hubble Spacer Telescope

by

Aishwarya Ramanathan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Abstract

Visualizing a model checker's run on a model can be useful when trying to gain a deeper understanding of the verification of the particular model. However, it can be difficult to formalize the problem that visualization solves as it varies from person to person. Having a visualized form of a model checker's run allows a user to pinpoint sections of the run without having to look through the entire log multiple times or having to know what to look for. This thesis presents the Hubble Spacer Telescope (HST), a visualizer for Spacer, an SMT horn clause based solver. HST combines multiple exploration graph views along with customizable lemma transformations. HST offers a variety of ways to transform lemmas so that a user can pick and choose how they want lemmas to be presented. HST's lemma transformations allow a user to change variable names, rearrange terms in a literal, and rearrange the placement of literals within the lemma through programming by example. HST allows users to not only visually depict a Spacer exploration log but it allows users to transform lemmas produced, in a way that the user hopes, will make understanding a Spacer model checking run, easier.

Given a Spacer exploration log, HST creates a raw exploration graph where clicking on a node produces the state of the model as well as the lemmas learned from said state. In addition, there is a second graph view which summarizes the exploration into its proof obligations. HST uses programming by example to simplify lemma transformations so that users only have to modify a few lemmas to transform all lemmas in an exploration log. Users can also choose between multiple transformations to better suit their needs.

This thesis presents an evaluation of HST through a case study. The case study is used to demonstrate the extent of the grammar created for lemma transformations. Users have the opportunity to transform disjunctions of literals produced by Spacer into a conditional statement, customized by the contents of the predicate. Since lemma transformations are completely customizable, HST can be viewed as per each individual user's preferences.

Acknowledgements

I would like to thank my supervisors, Professor Richard Trefler and Professor Arie Gurfinkel for providing their time, knowledge, and support throughout the creation of this thesis.

I would like to thank Professor Grant Weddell and Professor Jian Zhao for reading my thesis, attending my presentation, and providing thought-provoking questions and feedback.

I would like to the PROSE team at Microsoft for answering my queries about PROSE which ultimately led to the completion of the PROSE section of this thesis.

Finally, I would like to thank my peers, many of whom have helped by providing feedback on the visualizer, especially Nham Le, who has been and continues to be valuable collaborator on the visualizer.

Dedication

This is dedicated to all those who have ever wanted something dedicated to them.

Table of Contents

List of Figures

Chapter 1

Introduction

Model checkers are not the easiest to understand and gain useful information from. It is often the case that those that interact with model checkers are practiced in what to look for and can therefore understand what is presented to them. Even still, there are cases where model checker runs do not produce an expected output and then a seasoned user cannot rely on familiar indicators to be able to figure out what was different in the run. One way that has proven to be useful in improving the understanding of model checker runs is visualizing the run with annotations [5].

A visualization can be useful in multiple ways. A model checker like Spacer, whose solving method is IC3 based, can be visualized as a directed graph because Spacer's exploration follows a tree like structure with the same starting point for each exploration tree. Having the exploration as a graph allows a user to be able to click on each node for further details on the state of the exploration. While this makes the model checker run more accessible, each of the states can still be hard to decipher. A user may want to change the representation of a selected node. However, an exploration tree produced by Spacer can have hundreds, thousands, even hundreds of thousands of nodes which means a user would have to make the same changes over and over again to each node. This is where program synthesis can be useful to synthesize programs based on how the user decides to make changes.

Program synthesis through programming by example has played an interesting role in the world of artificial intelligence. While traditional machine learning relies on many training examples in order to create a very basic program, programming by example is able to produce favourable results with a fraction of the sample space [19]. This is achieved by using a combination of a reduction of the problem space, smart backtracking of possible programs, as well as a ranking system to score possible programs. Programming by examples has been applied to a number of problem spaces, resulting in generally improved functionality of the aforementioned problem space [19, 3].

In HST, program synthesis is used to determine changes that a user has made to a set of lemmas. To implement this synthesis system, we use PROSE, a program synthesis framework created by Microsoft. The system works by producing an input-output example or examples. The inputs come from the lemmas that SPACER produces while the output comes from the user's changes to the lemma. These examples are fed into PROSE whose problem space comes from a domain-specific language. For HST, we have provided a domain-specific language based on lemmas that Spacer produces. The system is discussed in detail in Chapter 4

This thesis consists of 6 consequent chapters. We first discuss the background information of the project Chapter 2. Chapters 3 and 4 detail the key tools of the project, HST and Prose respectively. Chapter 5 discusses a case study that was used to evaluate HST. The thesis concludes with Chapter 6 and Chapter 7 where we feature some related work as well as talk about some of the limitations and future plans for HST.

Chapter 2

Background

2.1 Model Checking With Constrained Horn Clauses

Constrained Horn Clauses (CHCs) are a fragment of a First-Order Logic. A single Horn Clause is a sentence of the form

$$\forall V \cdot (\varphi \wedge p_1[X_1] \wedge \cdots \wedge p_n[X_n]) \to h[X]$$

where φ is a constraint in some background theory (e.g., Linear Real Arithmetic (LRA), Arrays, bit-vectors, or a combinations of these theories), V is the set of all free variables in the clause, each X_i is a term over V, $p_1, \ldots, p_n$ and h are n-ary predicates, and each $p_i[X_i]$ is an application of a predicate, p_i to first order terms X_i. A set $\mathcal{C}$ of CHCs is *satisfiable* (modulo theory $\mathcal{T}$) iff there is an FOL model of $\mathcal{T}$ that satisfies every clause in $\mathcal{C}$.

Many questions in program verification and model checking naturally correspond to CHC satisfiability [6]. In particular, a weakest liberal precondition (WLP) of an imperative program can be expressed by a set of CHCs [6]. Under this interpretation, it is convenient to interpret a model for CHCs as an inductive invariant, and a resolution proof showing that CHCs are unsatisfiable as a counterexample. We rely on this duality when presenting results of CHC analysis to the users by visualizing the model as a collection of lemmas, where each lemma is a part of an inductive invariant, and visualizing the search for an inductive invariant as a tree of potential counterexamples.

$$\begin{aligned} \forall x \cdot x \leq 0 &\implies P(x) \\ \forall x, x' \cdot P(x) \wedge x < 5 \wedge x' = x + 1 &\implies P(x') \\ \forall x \cdot P(x) \wedge x \geq 10 &\implies false \end{aligned}$$

Figure 2.1: Example of Satisfiable Model in CHC from [6]

$$\begin{aligned} P(x) &\equiv \{x \mid x \leq 5\} \\ &\equiv \{5, 4, 3, 2, ...\} \end{aligned}$$

Figure 2.2: Representation for Figure 2.1 using the standard model of arithmetic [6]

CHC Example

Figure 2.1 depicts a set of CHCs that are satisfiable. This set of clauses is satisfiable by any x value that is less than or equal to 5. Figure 2.4 depicts a set of CHCs are unsatisfiable. The set of clauses in Figure 2.4 is unsatisfiable because if x begins at 0, after two iterations, x will have a value of 2. While this satisfies the recurrence condition (line 2 of Figure 2.4) the resulting value enters the bad state of the model (line 3 of Figure 2.4) which tells us that the model can enter a bad state from at least one path that begins at the initial state. Therefore, the model defined in Figure 2.4 is unsatisfiable.

Spacer

SPACER is an SMT based horn clause solver that uses an exploration of the model to determine inductive invariants for a model[1]

2.2 Programming By Example

Programming by example (PBE) is a type of program synthesis that allows a user to create a script (or a program) from a set of input/output examples. PBE works as follows: (a) the user presents a collection of input/output examples $E = \{(i_k, o_k)\}$, (b) PBE generates

[1]https://github.com/Z3Prover/z3/tree/master/src/muz/spacer.

$$\begin{aligned}
\vdash \forall x \cdot x \leq 0 &\implies x \leq 5 \\
\vdash \forall x, x' \cdot x \leq 5 \land x < 5 \land x' = x + 1 &\implies x' \leq 5 \\
\vdash \forall x \cdot x \leq 5 \land x \geq 10 &\implies false
\end{aligned}$$

Figure 2.3: Validation of Example in Figure 2.1 [6]

$$\begin{aligned}
\forall x \cdot x \leq 0 &\implies Q(x) \\
\forall x, x' \cdot Q(x) \land x < 5 \land x' = x + 1 &\implies Q(x') \\
\forall x \cdot Q(x) \land x \geq 2 &\implies false
\end{aligned}$$

Figure 2.4: Example of Unsatisfiable Model in CHC form [6]

a program P such that $P(i_k) = o_k$ for every input/output pair $(i_k, o_k) \in E$, if possible; (c) the user either accepts P, or, produces a new input/output pair (i_n, o_n) such that $P(i_n) \neq o_n$, and the synthesis process repeats. An example is demonstrated in Figure 2.6 and will be explained further in the next paragraph.

The unique strength of PBE is that it is easy to present in a format that allows non-programmers to create complex scripts. One of the most common and effective uses of PBE is in data wrangling, the manipulation of data to a format that can be more easily digested by the appropriate downstream format [19]. For example, given a column of first names and an adjacent column of corresponding last names in an Excel spreadsheet, this data may not be in an ideal format for other utilities that a user may have. A user may want to have the aforementioned names in the format of last name followed by first initial. FlashFill would allow a user to change the name format by inputting this new format of name into another column. FlashFill would then auto-fill the corresponding format for all last names, saving the user from tedious, repetitive work of changing the format for every single name. The aforementioned example is demonstrated in Figure 2.6 and Figure 2.7.

$$
\begin{array}{c}
(x=0)\dfrac{\forall x \cdot x \leq 0 \implies Q(x)}{Q(0)} \qquad \forall x \cdot Q(x) \wedge x < 5 \implies Q(x+1) \\
\hline
Q(1) \\
\forall x \cdot Q(x) \wedge x < 5 \implies Q(x+1) \\
\hline
Q(2) \\
\forall x \cdot Q(x) \wedge x \geq 2 \implies \mathit{false} \\
\hline
\mathit{false}
\end{array}
$$

Figure 2.5: Refutation of Example 2.4 [6]

2.2.1 Programming By Example with Microsoft Prose

In this thesis, we use a PBE framework called PROSE from Microsoft Research. [2] The framework consists of a series of APIs for program synthesis. The key component of PROSE used in this thesis is its program synthesis frame work for custom domain-specific languages (DSLs). Synthesizing programs with PROSE involves a program specification.

Every specification has a set of inputs and a set of constraints for the program's output based on the program's expected output. Every PROSE DSL consists of three main components: (a) the grammar/semantics, (b) witness functions, and (c) features/ranking scores. The following breakdown of the Prose framework is based on [45] and is illustrated by Figure 2.8.

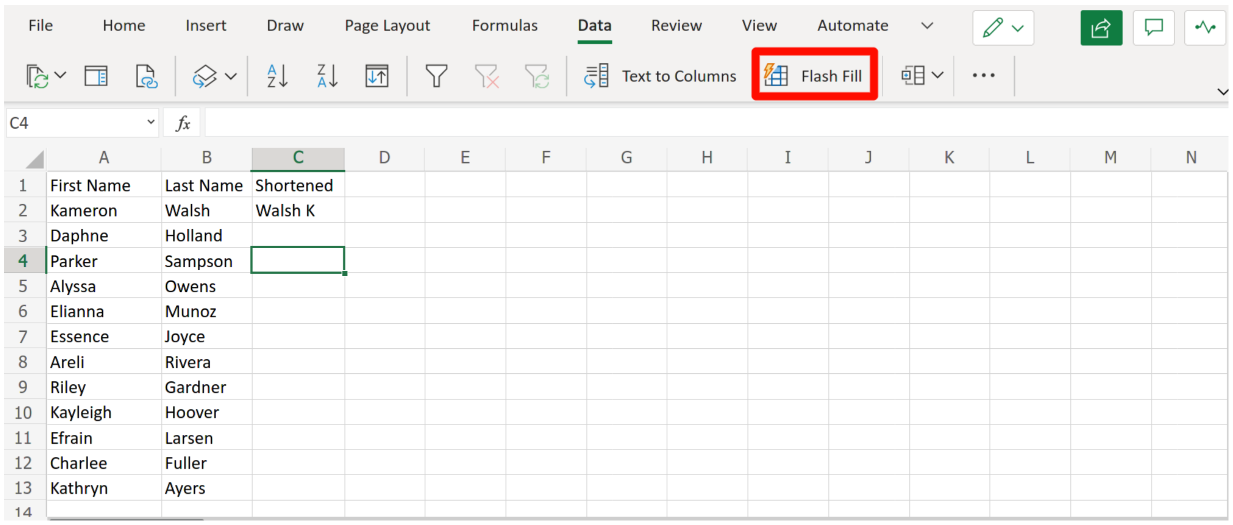
Synthesis Algorithm

The synthesized program or programs is expected to be consistent with the spec and is synthesized through a combination of deduction, search, and ranking.

- Deduction is a top-down exploration of the DSL grammar, which reduces the synthesis problem to smaller synthesis sub-problems through the divide-and-conquer dynamic programming algorithm.

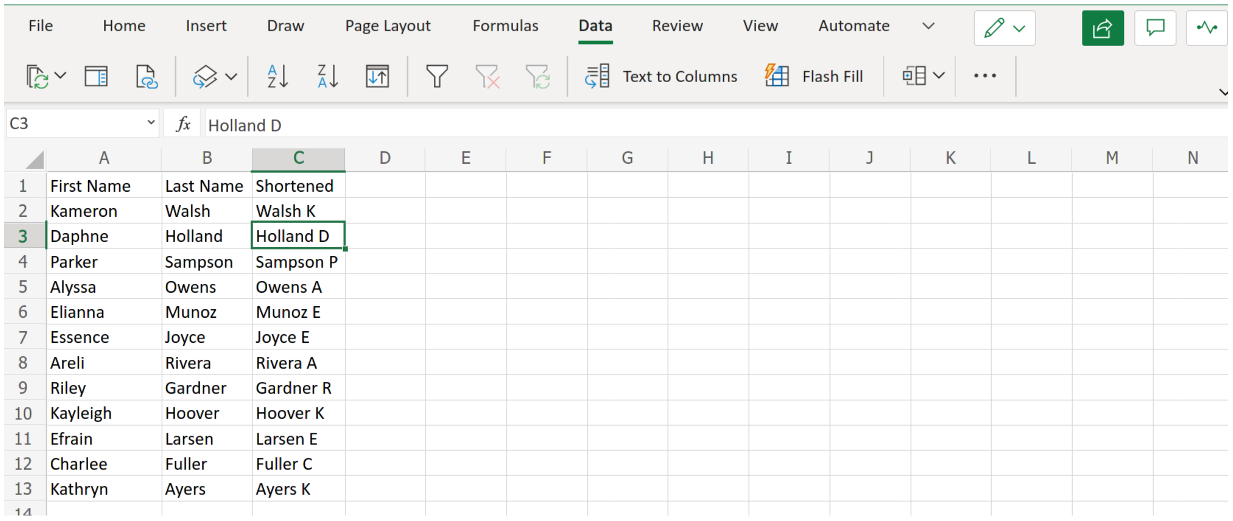
- Search is an enumerating algorithm, which constructs possible sub-expressions from the grammar and checks if they are consistent with the spec.

[2] https://github.com/microsoft/prose.



	A	B	C
1	First Name	Last Name	Shortened
2	Kameron	Walsh	Walsh K
3	Daphne	Holland	
4	Parker	Sampson	
5	Alyssa	Owens	
6	Elianna	Munoz	
7	Essence	Joyce	
8	Areli	Rivera	
9	Riley	Gardner	
10	Kayleigh	Hoover	
11	Efrain	Larsen	
12	Charlee	Fuller	
13	Kathryn	Ayers	

Figure 2.6: FLASHFILL Example of Initial Data and User Input



Holland D

	A	B	C
1	First Name	Last Name	Shortened
2	Kameron	Walsh	Walsh K
3	Daphne	Holland	Holland D
4	Parker	Sampson	Sampson P
5	Alyssa	Owens	Owens A
6	Elianna	Munoz	Munoz E
7	Essence	Joyce	Joyce E
8	Areli	Rivera	Rivera A
9	Riley	Gardner	Gardner R
10	Kayleigh	Hoover	Hoover K
11	Efrain	Larsen	Larsen E
12	Charlee	Fuller	Fuller C
13	Kathryn	Ayers	Ayers K

Figure 2.7: FLASHFILL Example after FLASHFILL has been applied

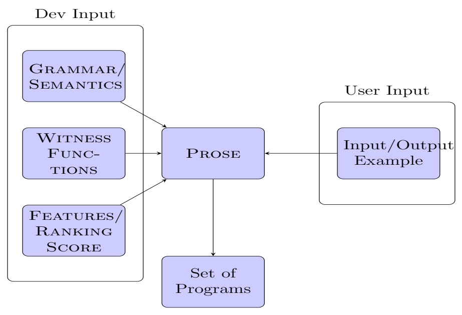


Figure 2.8: Inputs required for Prose to produce a set of programs.

- Ranking consists of picking the most effective program based on consistency with the spec. While this may produce many programs, programs can be filtered through the description, further down, in Section 2.2.1, Feature/Ranking Score.

Grammar & Semantics

The syntax or grammar of the prose synthesis framework sets the stage for the entire set of allowed functions that can be used in the synthesized program. This grammar is known as a domain-specific language or a DSL. A DSL is a context-free language that consists of functions for a specific purpose. Each semantics function corresponds to a function description defined in the grammar. There are no type specifications for a semantics function as per the PROSE framework but the user should ensure that input and output types of each function match the types in the corresponding function definition in the Grammar.

Witness Functions

Witness functions are used to perform back-propogation learning on given input and output examples. Each Witness Function corresponds to a parameter of a semantics function. Using a witness function, the synthesis program can deduce a specification for the parameter when given an input and output for the related semantics function. A witness function takes in a specification on the output of the entire semantics functions and returns an output spec for one parameter of the semantics function. A very simple example of a witness function would be, given a semantics function that takes in two parameters, that are numbers, and returns the sum of the two numbers, the witness function for the second parameter would be given a first parameter and a sum. Given the number 2 and a resulting sum of 5, the witness function would produce a mapping of the input/output example (2,5) paired with 3 as $2 + 3 = 5$.

Feature/Ranking Score

Every program is assigned a score based on a metric in its recursive structure. Generally that filters down into its parameters where each variable parameter used is assigned a score.

Chapter 3

Hubble Spacer Telescope

The Hubble SPACER Telescope (HST) is a tool that users can use to visualize SPACER's exploration on a given model while also manipulating the lemmas produced in order to make it easier for the user to read and understand. The tool is built using a ReactJS front end along with a Flask and .NET Core back end. The .NET Core back-end houses the PROSE program synthesis framework while the Flask back end contains methods to support the visualization part of HST.

HST derives from the base code for the Vampire Saturation Visualizer, SatVis [18]. Spacer produces a log of the exploration that can be visualized in the form of a directed graph or tree. Each node in the graph pertains to a program state of the exploration tree.

3.1 Overview

In this section, we demonstrate HST by showing a detailed user interaction on an example from our case study. The case study itself is described later in Chapter 5.

When you first load HST, the landing page contains:

1. a large input box (left side of Figure 3.1),
2. a text box with a drop-down for a selection of SPACER options (top of blue box in Figure 3.1),
3. a text box to input a manual string of SPACER options (bottom of blue box in Figure 3.1), and

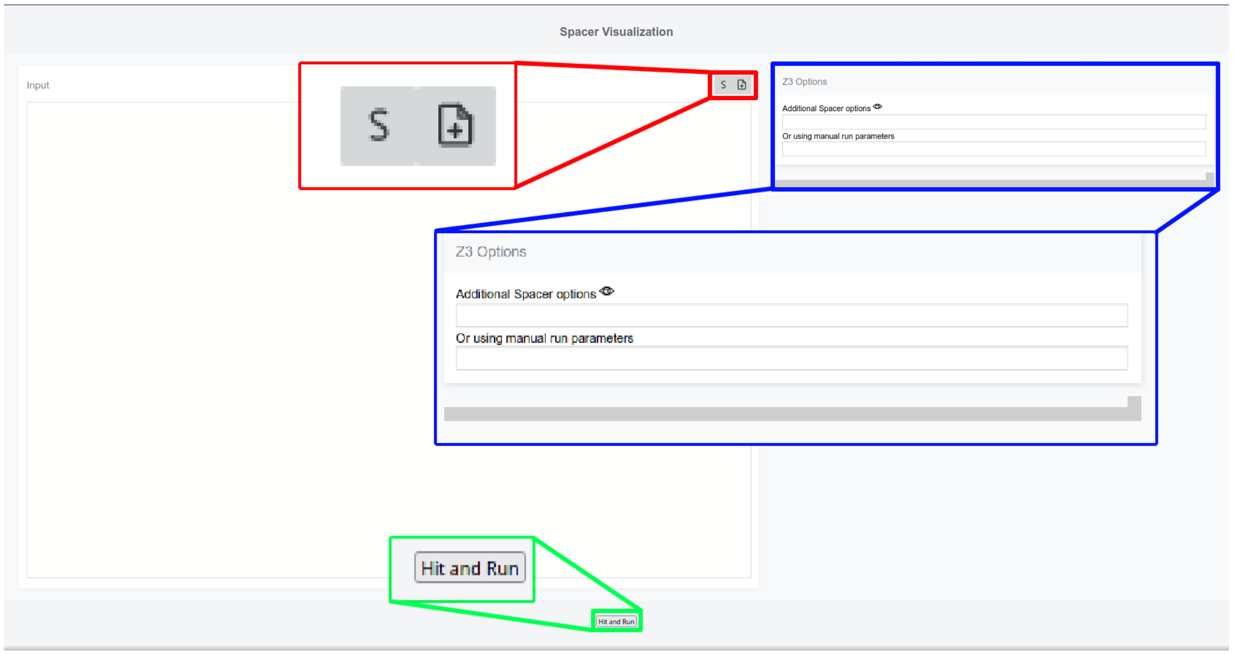


Figure 3.1: HST landing page.

4. a `Hit and Run` button (green box in Figure 3.1).

A user can click on the file icon (right icon of red box in 3.1) to choose a file from a local hard drive.

If you click on the `S` (left icon of red box in Figure 3.1) button, the user is presented multiple options for inputting a model, shown in Figure 3.2. The user can choose to upload an SMT-LIB file or a Spacer.log file of an existing Spacer run. In addition, the user can choose to give a custom name to the particular run. In our case, we would like to upload the `simple-bakery.smt2` file and since the file already has a recognizable name, we will not be adding a custom name for the run.

Once a file has been selected, a user can either add additional options or proceed by clicking the `Hit And Run` button, at which point they will be redirected to the Visualization Page, shown in Figure 3.3. For this example, we will not be using any extra options.

The Visualization Page has a number of different things for the user to interact with. The bulk of the page is taken up by the graph view. By default, the view shown is the PobVis view. When first encountering the Visualization Page, the graph view is populated by the contents of the intermediate Spacer log. The user can see the status of Spacer

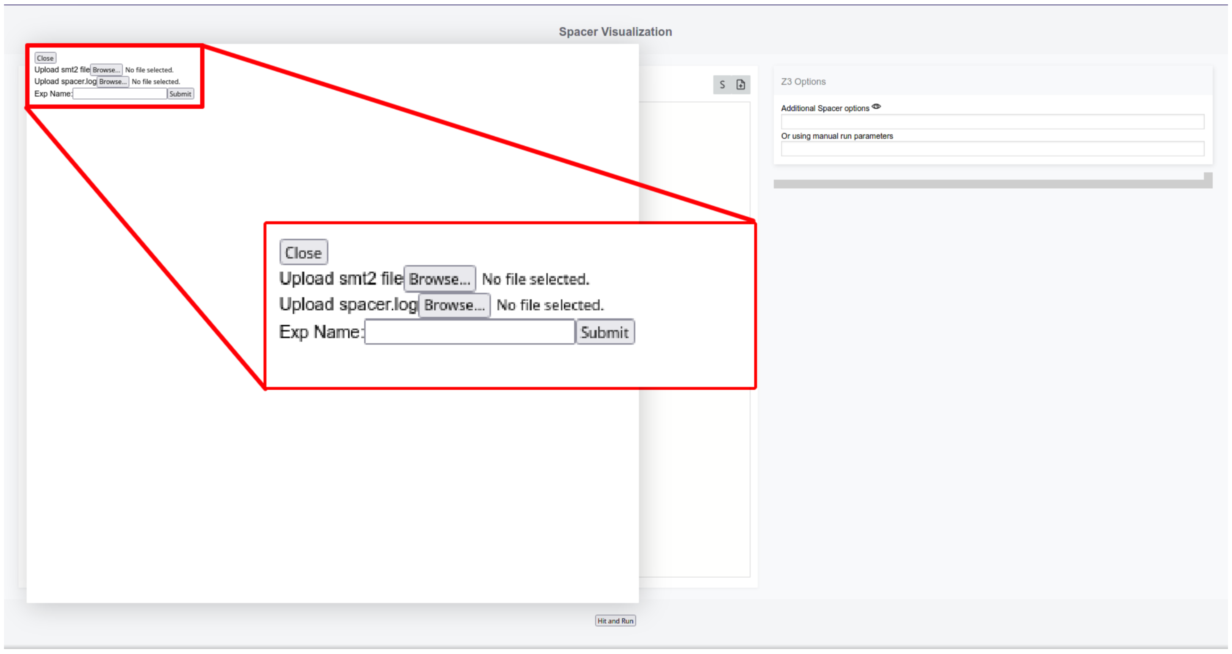
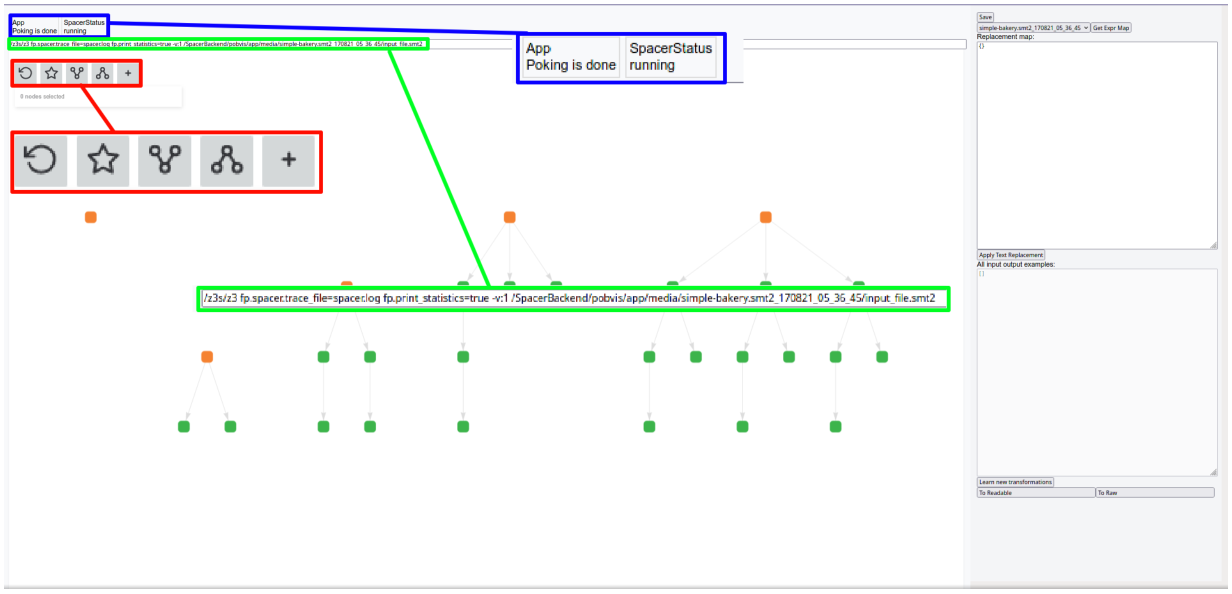

Spacer Visualization
Close
Upload smt2 file Browse... No file selected.
Upload spacer.log Browse... No file selected.
Exp Name:
Submit
Z3 Options
Additional Spacer options
Or using manual run parameters
Hit and Run

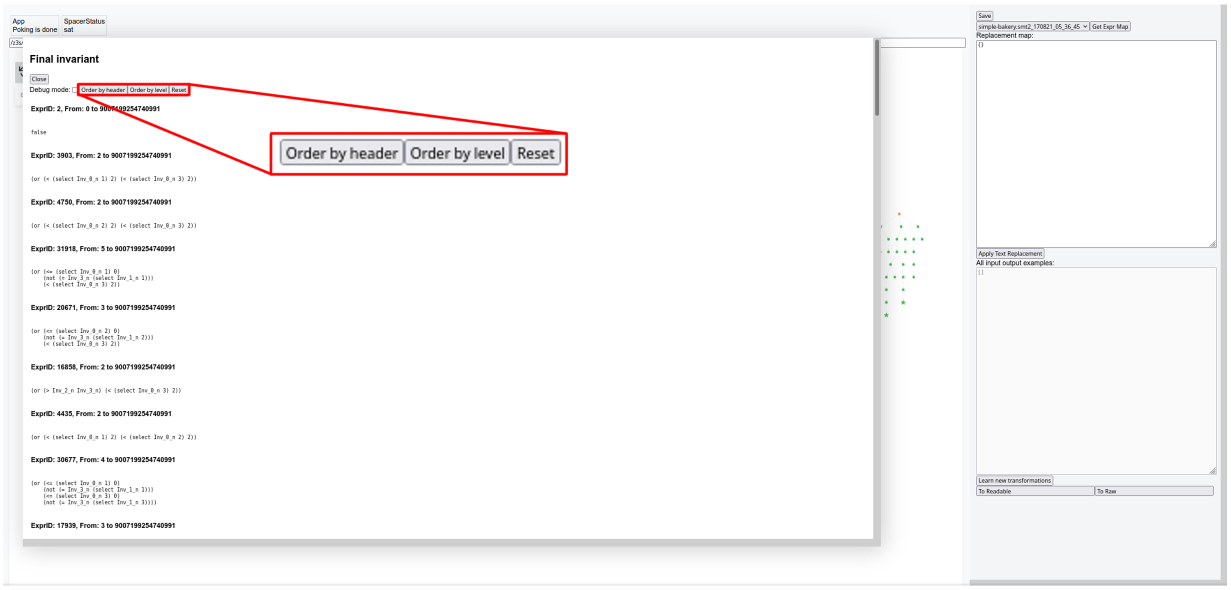
Figure 3.2: HST Spacer Input Options



App	SpacerStatus
Poking is done	running

```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_170821_05_36_45/input_file.smt2
```

Figure 3.3: HST Visualization Page



App
Poking is done

SpacerStatus
sat

Final invariant

Debug mode:

```
ExprID: 2, From: 0 to 9007199254740991

false

ExprID: 3903, From: 2 to 9007199254740991

(or (< (select Inv_0_n 1) 2) (< (select Inv_0_n 3) 2))

ExprID: 4750, From: 2 to 9007199254740991

(or (< (select Inv_0_n 2) 2) (< (select Inv_0_n 3) 2))

ExprID: 31918, From: 5 to 9007199254740991

(or (<= (select Inv_0_n 1) 0)
    (not (= Inv_3_n (select Inv_1_n 1)))
    (< (select Inv_0_n 3) 2))

ExprID: 20671, From: 3 to 9007199254740991

(or (<= (select Inv_0_n 2) 0)
    (not (= Inv_3_n (select Inv_1_n 2)))
    (< (select Inv_0_n 3) 2))

ExprID: 16858, From: 2 to 9007199254740991

(or (> Inv_2_n Inv_3_n) (< (select Inv_0_n 3) 2))

ExprID: 4435, From: 2 to 9007199254740991

(or (< (select Inv_0_n 1) 2) (< (select Inv_0_n 2) 2))

ExprID: 30677, From: 4 to 9007199254740991

(or (<= (select Inv_0_n 1) 0)
    (not (= Inv_3_n (select Inv_1_n 1)))
    (<= (select Inv_0_n 3) 0)
    (not (= Inv_3_n (select Inv_1_n 3))))

ExprID: 17939, From: 3 to 9007199254740991
```

simple-bakery.smt2_170821_05_36_45

Replacement map:
{}

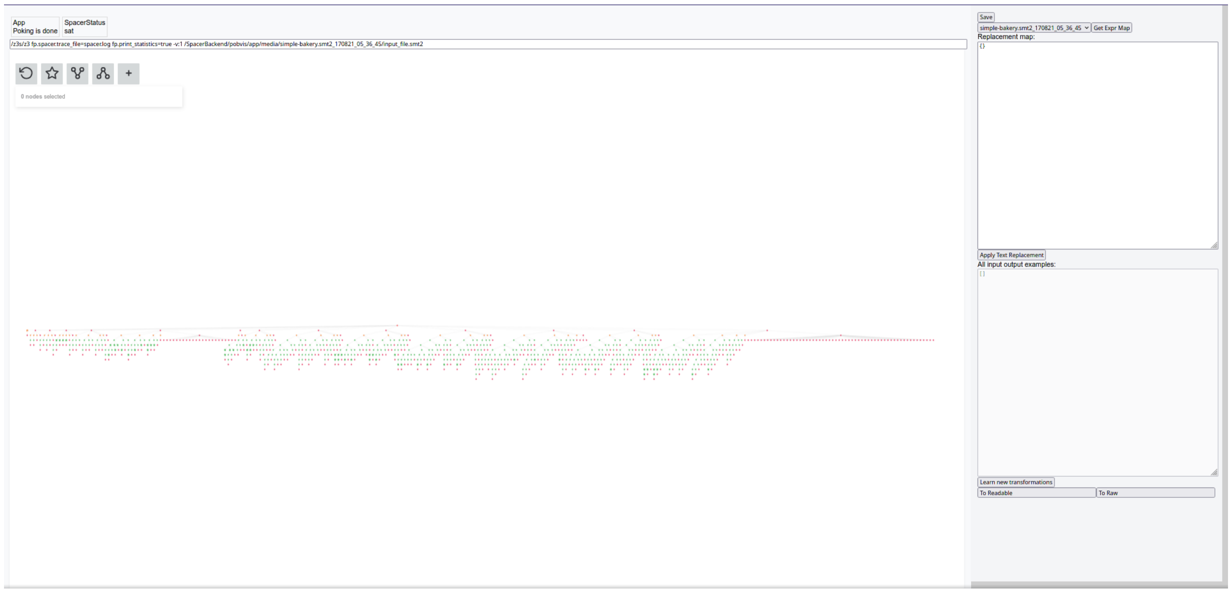
All input output examples:
[]

Figure 3.4: Star View

in the top left boxes, as shown by the blue highlighted box in Figure 3.3. The box on the left contains that status of the Poke and the box on the right contains the status of Spacer. As you can see, in Figure 3.3, the status of the Poke is `Poking is done` while the Spacer status still says `running`. Below those boxes, you can see the Spacer command, highlighted by the green box in Figure 3.3, that was run to produce the Spacer log that HST is interpreting. Below the Spacer command, there are a series of five buttons, Poke, Star, SatVis, PobVis, and Multi Select, highlighted by the red box in Figure 3.3.

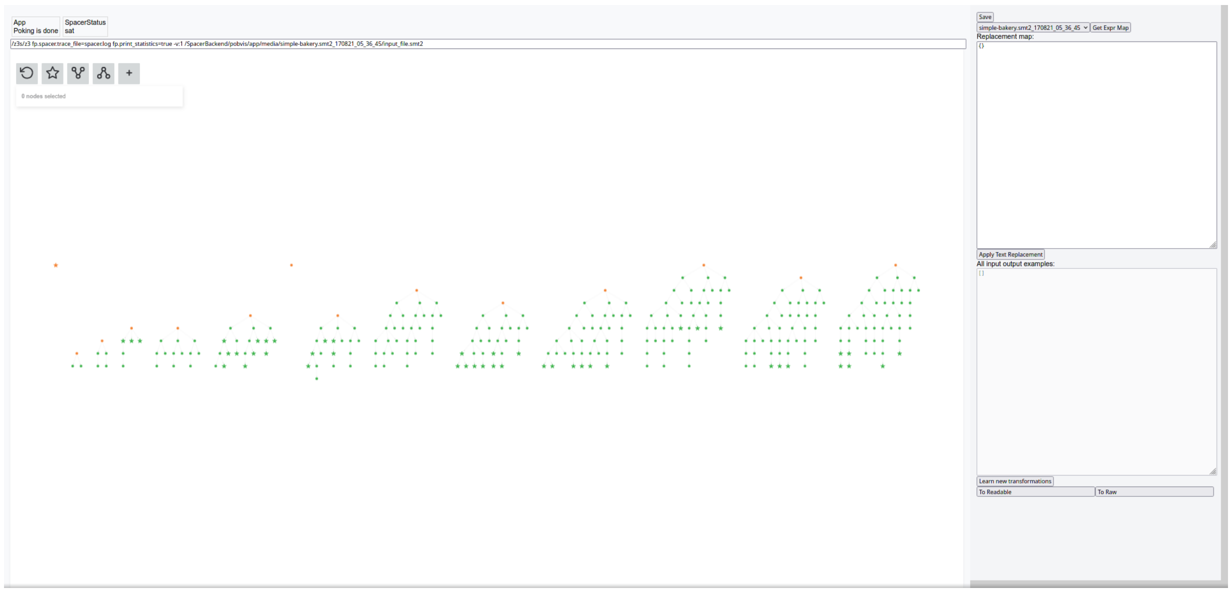
A user can use the Poke button to check on the status of Spacer. Once Poke is clicked, the Poke status box will say Poking Spacer... until the Poke is complete, at which point the box will say Poking is done. In most cases, a user will want to Poke until Spacer has completed its exploration and has made a conclusion on the model. Once the Spacer status box has a value other than Running, Spacer has completed its run and Spacer will not update the Spacer log file.

The next three buttons have to do directly with the visualization graph shown in HST, Star, SatVis, and PobVis buttons. Clicking on the Star button will allow a user to see all lemmas that have been identified as an inductive invariant for the model (Figure 3.4). Each of these inductive invariants' nodes in the graph can be identified by a star symbol. As you can see, there are a number of inductive invariants identified for this example. The SatVis button allows a user to view the full exploration graph from Spacer (Figure 3.5).



```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_170821_05_36_45/input_file.smt2
```

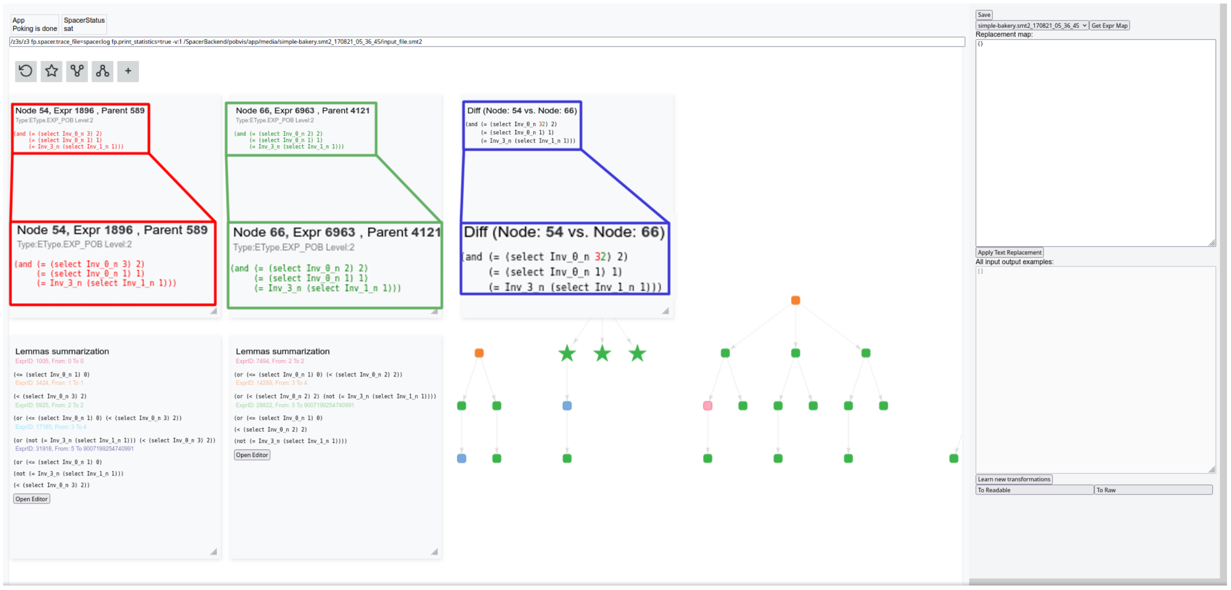
Figure 3.5: SatVis View



```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_170821_05_36_45/input_file.smt2
```

Figure 3.6: PobVis View

The PobVis button allows a user to see a summarized version of the exploration in the



```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_170821_05_36_45/input_file.smt2
```

Node 54, Expr 1896 , Parent 589
Type:EType.EXP_POB Level:2

```
(and (= (select Inv_0_n 3) 2)
     (= (select Inv_0_n 1) 1)
     (= Inv_3_n (select Inv_1_n 1)))
```

Node 66, Expr 6963 , Parent 4121
Type:EType.EXP_POB Level:2

```
(and (= (select Inv_0_n 2) 2)
     (= (select Inv_0_n 1) 1)
     (= Inv_3_n (select Inv_1_n 1)))
```

Diff (Node: 54 vs. Node: 66)

```
(and (= (select Inv_0_n 32) 2)
     (= (select Inv_0_n 1) 1)
     (= Inv_3_n (select Inv_1_n 1)))
```

Lemmas summarization

```
ExprID: 1005, From: 0 To 0
(<= (select Inv_0_n 1) 0)
ExprID: 3424, From: 1 To 1
(< (select Inv_0_n 3) 2)
ExprID: 5925, From: 2 To 2
(or (<= (select Inv_0_n 1) 0) (< (select Inv_0_n 3) 2))
ExprID: 17185, From: 3 To 4
(or (not (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
ExprID: 31918, From: 5 To 9007199254740991
(or (<= (select Inv_0_n 1) 0)
(not (= Inv_3_n (select Inv_1_n 1)))
(< (select Inv_0_n 3) 2))
```

Lemmas summarization

```
ExprID: 7494, From: 2 To 2
(or (<= (select Inv_0_n 1) 0) (< (select Inv_0_n 2) 2))
ExprID: 14289, From: 3 To 4
(or (< (select Inv_0_n 2) 2) (not (= Inv_3_n (select Inv_1_n 1))))
ExprID: 28822, From: 5 To 9007199254740991
(or (<= (select Inv_0_n 1) 0)
(< (select Inv_0_n 2) 2)
(not (= Inv_3_n (select Inv_1_n 1))))
```

simple-bakery.smt2_170821_05_36_45
Replacement map:
{}
All input output examples:
[]

Figure 3.7: Multi Select Mode

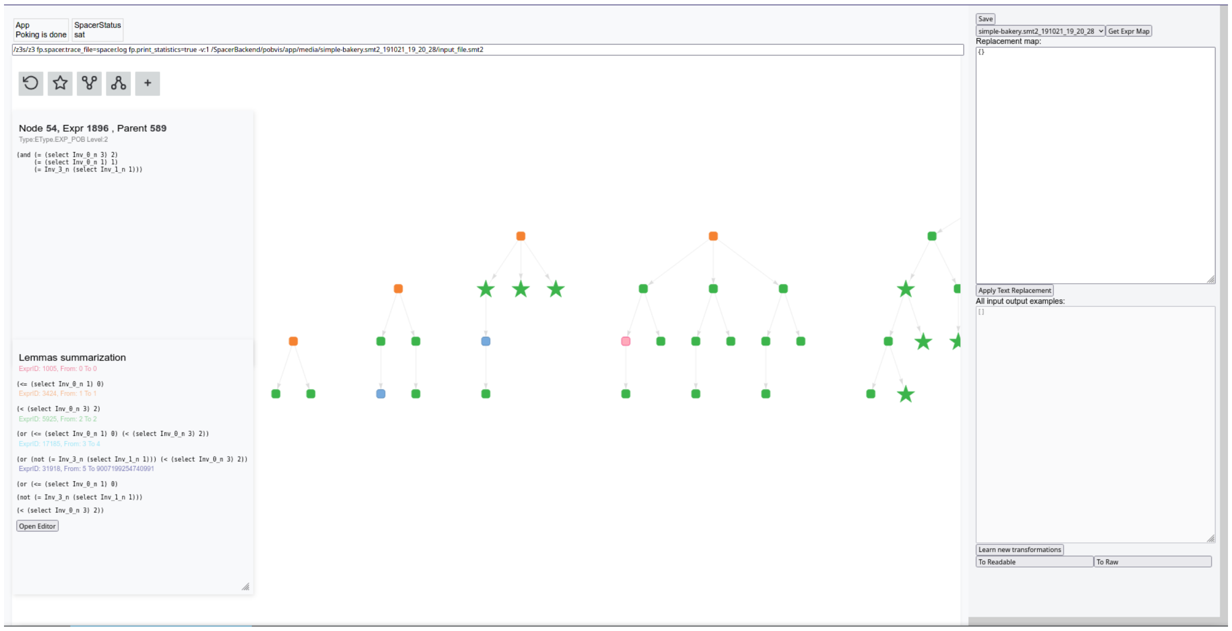
form of separate trees for each level of exploration (Figure 3.6).

Tha last button, the Multi Select button allows a user to compare two different nodes in the graph. The user can do this by clicking on the Multi Select button and then clicking on the two nodes they would like to compare. Another box will appear with the diff (blue box in Figure 3.7) between the two nodes selected. The nodes are coloured according to the diff in order for the user to easily see which node is being referenced in the diff. An example of Multi Select mode can be seen in Figure 3.7.

The user can interact with the graph by clicking on nodes. Clicking on a node will open up the two information boxes for the node on the left side of the page and change the colour of the node to red. The top box is the state of the problem that the node represents. The bottom box shows potential and confirmed invariants for the problem. The user has multiple options on changing the appearance of the lemmas in the lemma summarization.

Firstly, a user can customize the variable names in the lemmas by utilising the variable map on the top left of the graph page. An example of this is shown in Figure 3.9.

Secondly, a user can toggle between infix and prefix notation for the lemmas to make it easier to read. By default, lemmas are presented with prefix notation. The user can use the To Readable button to switch to infix notation and the To Raw button to switch back to prefix notation. This is shown in Figure 3.10.



App
Poking is done

SpacerStatus
sat

```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_191021_19_20_28/input_file.smt2
```

Node 54, Expr 1896 , Parent 589
Type:EType.EXP_POB Level:2

```
(and (= (select Inv_0_n 3) 2)
     (= (select Inv_0_n 1) 1)
     (= Inv_3_n (select Inv_1_n 1)))
```

Lemmas summarization

ExprID: 1005, From: 0 To 0

```
(<= (select Inv_0_n 1) 0)
```

ExprID: 3424, From: 1 To 1

```
(< (select Inv_0_n 3) 2)
```

ExprID: 5925, From: 2 To 2

```
(or (<= (select Inv_0_n 1) 0) (< (select Inv_0_n 3) 2))
```

ExprID: 17185, From: 3 To 4

```
(or (not (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
```

ExprID: 31918, From: 5 To 9007199254740991

```
(or (<= (select Inv_0_n 1) 0)
(not (= Inv_3_n (select Inv_1_n 1)))
(< (select Inv_0_n 3) 2))
```

simple-bakery.smt2_191021_19_20_28

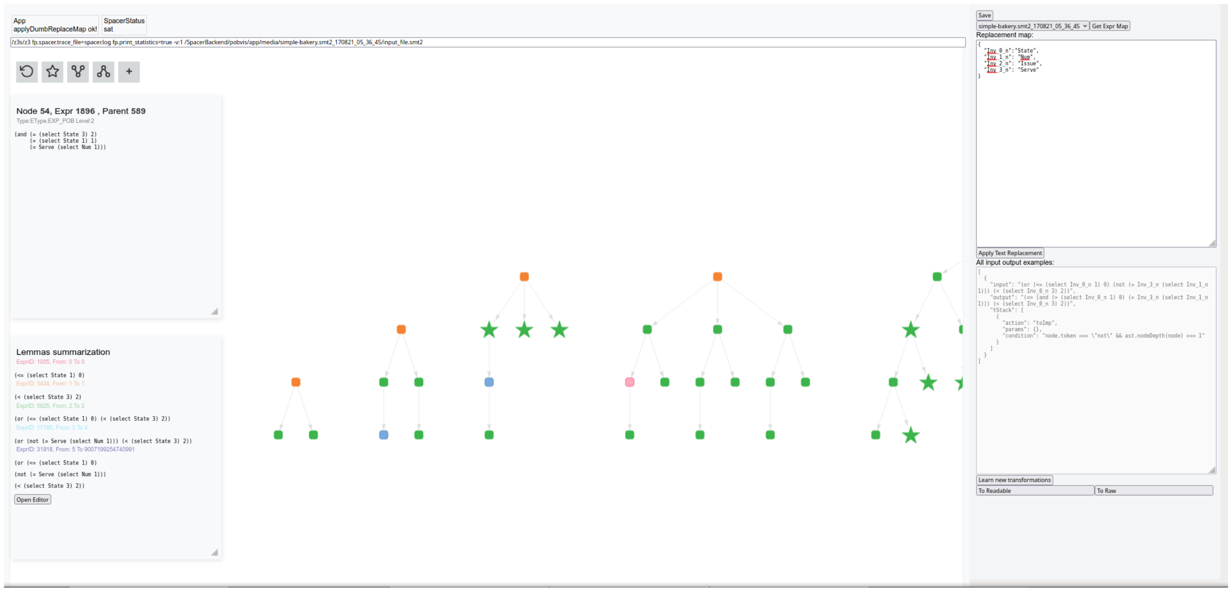
Replacement map:

```
{}
```

All input output examples:

```
[]
```

Figure 3.8: Selecting a Node



App
applyDumbReplaceMap ok!

SpacerStatus
sat

```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_170821_05_36_45/input_file.smt2
```

Node 54, Expr 1896 , Parent 589
Type:EType.EXP_POB Level:2

```
(and (= (select State 3) 2)
     (= (select State 1) 1)
     (= Serve (select Num 1)))
```

Lemmas summarization

ExprID: 1005, From: 0 To 0

```
(<= (select State 1) 0)
```

ExprID: 3424, From: 1 To 1

```
(< (select State 3) 2)
```

ExprID: 5925, From: 2 To 2

```
(or (<= (select State 1) 0) (< (select State 3) 2))
```

ExprID: 17185, From: 3 To 4

```
(or (not (= Serve (select Num 1))) (< (select State 3) 2))
```

ExprID: 31918, From: 5 To 9007199254740991

```
(or (<= (select State 1) 0)
(not (= Serve (select Num 1)))
(< (select State 3) 2))
```

simple-bakery.smt2_170821_05_36_45

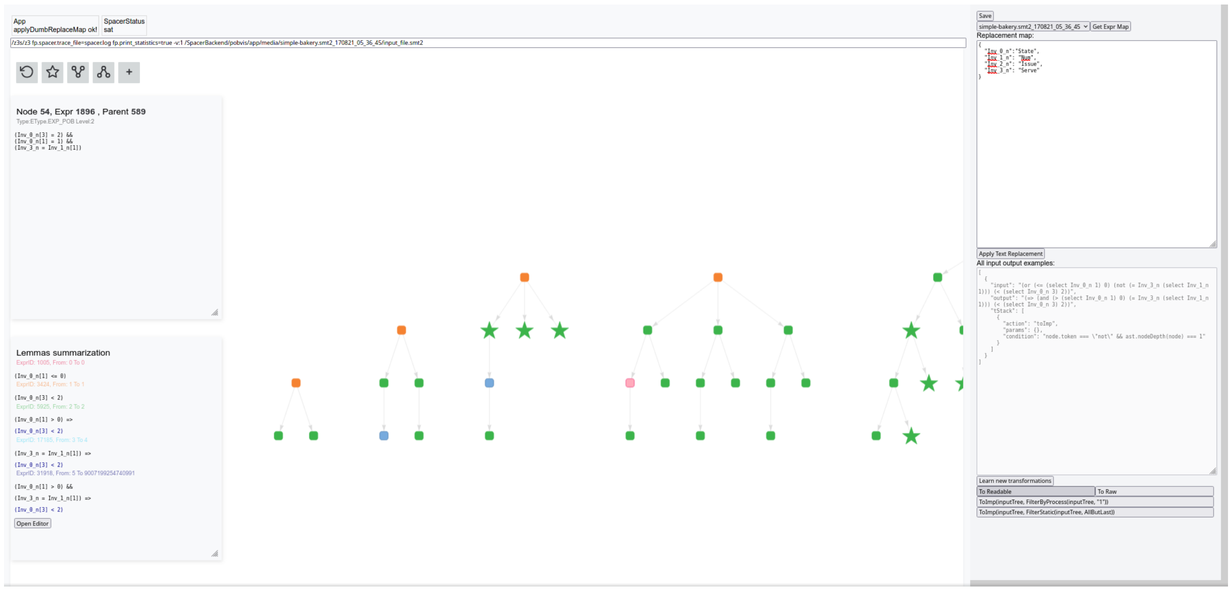
Replacement map:

```
{
  "Inv_0_n":"State",
  "Inv_1_n": "Num",
  "Inv_2_n": "Issue",
  "Inv_3_n": "Serve"
}
```

All input output examples:

```
[
  {
    "input": "(or (<= (select Inv_0_n 1) 0) (not (= Inv_3_n (select Inv_1_n
1))) (< (select Inv_0_n 3) 2))",
    "output": "(=> (and (> (select Inv_0_n 1) 0) (= Inv_3_n (select Inv_1_n
1))) (< (select Inv_0_n 3) 2))",
    "tStack": [
      {
        "action": "toImp",
        "params": {},
        "condition": "node.token === \"not\" && ast.nodeDepth(node) === 1"
      }
    ]
  }
]
```

Figure 3.9: Replacing Variables with Custom Values


App
applyDumbReplaceMap ok!
SpacerStatus
sat
+
Save
Get Expr Map
Apply Text Replacement
Learn new transformations
To Readable
To Raw
ToImp(inputTree, FilterByProcess(inputTree, "1"))
ToImp(inputTree, FilterStatic(inputTree, AllButLast))
Open Editor

```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_170821_05_36_45/input_file.smt2
```

Node 54, Expr 1896 , Parent 589
Type:EType.EXP_POB Level:2

```
(Inv_0_n[3] = 2) &&
(Inv_0_n[1] = 1) &&
(Inv_3_n = Inv_1_n[1])
```

Lemmas summarization
ExprID: 1005, From: 0 To 0

```
(Inv_0_n[1] <= 0)
```

ExprID: 3424, From: 1 To 1

```
(Inv_0_n[3] < 2)
```

ExprID: 5925, From: 2 To 2

```
(Inv_0_n[1] > 0) =>
(Inv_0_n[3] < 2)
```

ExprID: 17185, From: 3 To 4

```
(Inv_3_n = Inv_1_n[1]) =>
(Inv_0_n[3] < 2)
```

ExprID: 31918, From: 5 To 9007199254740991

```
(Inv_0_n[1] > 0) &&
(Inv_3_n = Inv_1_n[1]) =>
(Inv_0_n[3] < 2)
```

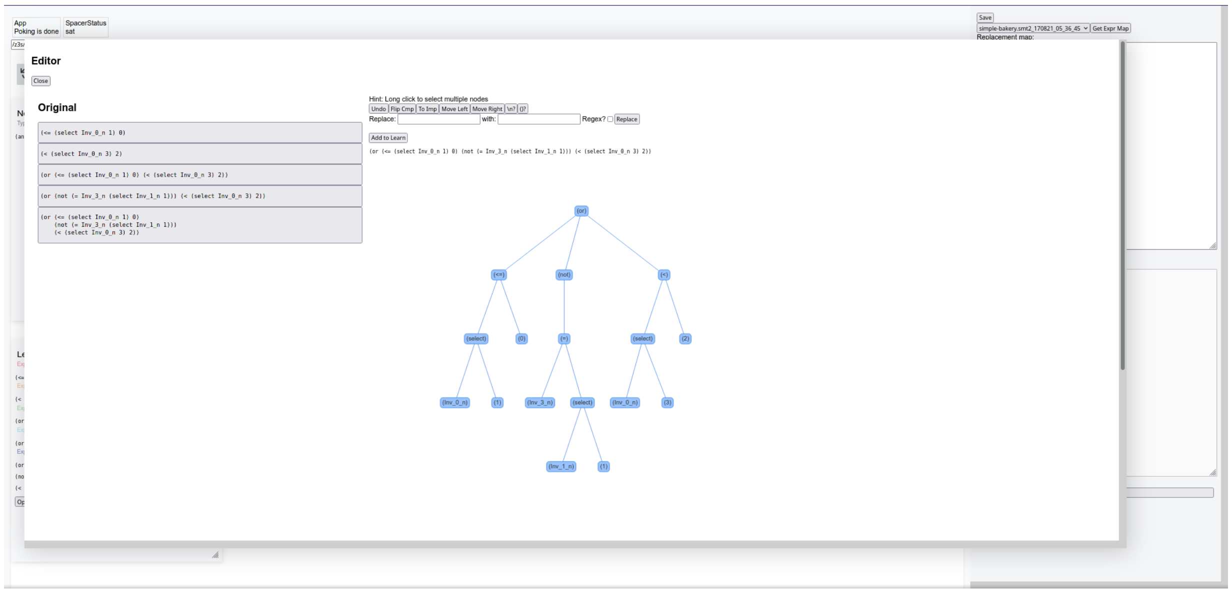
simple-bakery.smt2_170821_05_36_45
Replacement map:

```
{
  "Inv_0_n":"State",
  "Inv_1_n": "Num",
  "Inv_2_n": "Issue",
  "Inv_3_n": "Serve"
}
```

All input output examples:

```
[
  {
    "input": "(or (<= (select Inv_0_n 1) 0) (not (= Inv_3_n (select Inv_1_n
1))) (< (select Inv_0_n 3) 2))",
    "output": "(=> (and (> (select Inv_0_n 1) 0) (= Inv_3_n (select Inv_1_n
1))) (< (select Inv_0_n 3) 2))",
    "tStack": [
      {
        "action": "toImp",
        "params": {},
        "condition": "node.token === \"not\" && ast.nodeDepth(node) === 1"
      }
    ]
  }
]
```

Figure 3.10: Lemmas with Infix Notation



simple-bakery.smt2_170821_05_36_45
Replacement map:

Original

```
(<= (select Inv_0_n 1) 0)
```

```
(< (select Inv_0_n 3) 2)
```

```
(or (<= (select Inv_0_n 1) 0) (< (select Inv_0_n 3) 2))
```

```
(or (not (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
```

```
(or (<= (select Inv_0_n 1) 0)
    (not (= Inv_3_n (select Inv_1_n 1)))
    (< (select Inv_0_n 3) 2))
```

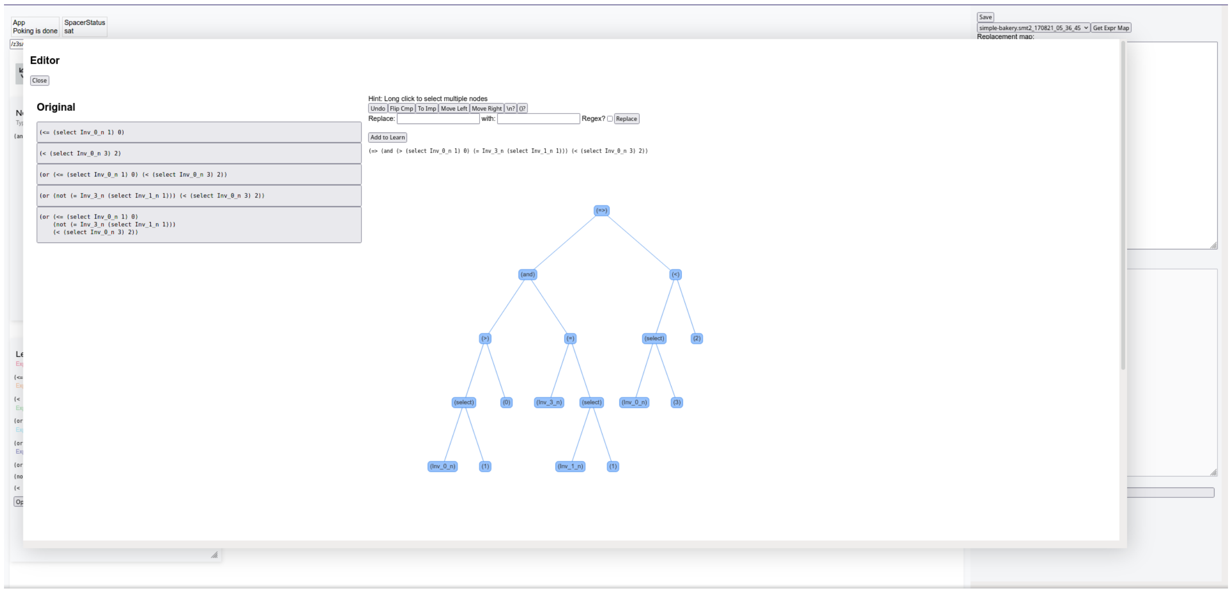
Hint: Long click to select multiple nodes

Replace: with: Regex? ☐

```
(or (<= (select Inv_0_n 1) 0) (not (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
```

Figure 3.11: Open Editor Box

Once Spacer has completed its run (the Spacer status box says `sat`, `unsat`, or



Original

```
(<= (select Inv_0_n 1) 0)
```

```
(< (select Inv_0_n 3) 2)
```

```
(or (<= (select Inv_0_n 1) 0) (< (select Inv_0_n 3) 2))
```

```
(or (not (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
```

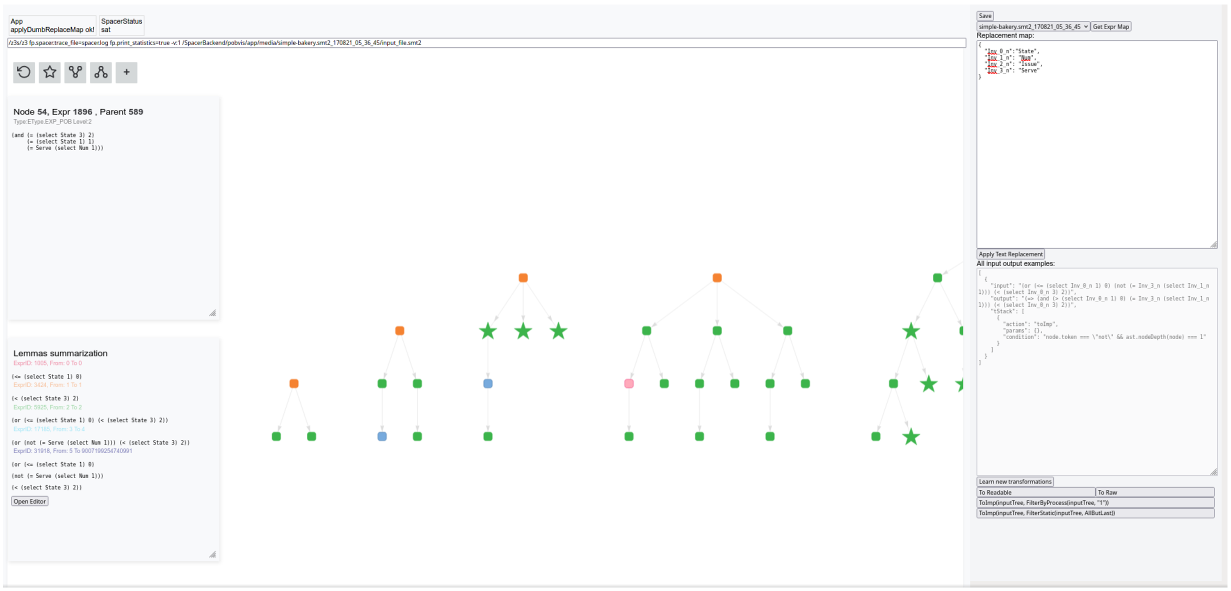
```
(or (<= (select Inv_0_n 1) 0)
    (not (= Inv_3_n (select Inv_1_n 1)))
    (< (select Inv_0_n 3) 2))
```

```
(=> (and (> (select Inv_0_n 1) 0) (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
```

Figure 3.12: Edited Lemma in OPEN EDITOR Box

`unknown`), the user will see a button on the bottom of the box titled OPEN EDITOR. Clicking on OPEN EDITOR will open a box in the center of the page that allows a user to modify lemmas learned on the selected node. The OPEN EDITOR box contains buttons on the left side that correspond to each of the lemmas in the selected node. When a user clicks on one of these buttons. The corresponding lemma is presented on the right side in its abstract syntax tree form. This tree can then be modified by selecting nodes and using the buttons on the top to modify the tree. In Figure 3.11, the fourth lemma in the node was selected and the AST for that lemma is displayed. After selecting the first and second nodes in the second level of the tree, the user can click on the TOIMP button and transform their tree to describe an equivalent conditional statement. This tree is displayed in Figure 3.12.

Once the node is modified to a user's liking, the user can click on the ADD TO LEARN button and both the initial form as well as the modified form of the lemma get stored as an input/output example. All stored input/output examples are visible to the user on the bottom right side of the main graph space and can be seen in Figure 3.13. Once a user is satisfied with the amount of input/output examples, the user can click on the LEARN NEW TRANSFORMATIONS button under the input/output examples display box to trigger the Custom Prose API, discussed in Chapter 4, which will process the input/output examples to produce a function that can be applied to every other node. When complete, clicking


App
applyDumbReplaceMap ok!

SpacerStatus
sat

```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_170821_05_36_45/input_file.smt2
```

Node 54, Expr 1896 , Parent 589
Type:EType.EXP_POB Level:2

```
(and (= (select State 3) 2)
     (= (select State 1) 1)
     (= Serve (select Num 1)))
```

Lemmas summarization

ExprID: 1005, From: 0 To 0
```
(<= (select State 1) 0)
```
ExprID: 3424, From: 1 To 1
```
(< (select State 3) 2)
```
ExprID: 5925, From: 2 To 2
```
(or (<= (select State 1) 0) (< (select State 3) 2))
```
ExprID: 17185, From: 3 To 4
```
(or (not (= Serve (select Num 1))) (< (select State 3) 2))
```
ExprID: 31918, From: 5 To 9007199254740991
```
(or (<= (select State 1) 0)
(not (= Serve (select Num 1)))
(< (select State 3) 2))
```

simple-bakery.smt2_170821_05_36_45

Replacement map:
```
{
  "Inv_0_n":"State",
  "Inv_1_n": "Num",
  "Inv_2_n": "Issue",
  "Inv_3_n": "Serve"
}
```

All input output examples:
```
[
  {
    "input": "(or (<= (select Inv_0_n 1) 0) (not (= Inv_3_n (select Inv_1_n
1))) (< (select Inv_0_n 3) 2))",
    "output": "(=> (and (> (select Inv_0_n 1) 0) (= Inv_3_n (select Inv_1_n
1))) (< (select Inv_0_n 3) 2))",
    "tStack": [
      {
        "action": "toImp",
        "params": {},
        "condition": "node.token === \"not\" && ast.nodeDepth(node) === 1"
      }
    ]
  }
]
```

Open Editor
Save
Get Expr Map
Apply Text Replacement
Learn new transformations
To Readable
To Raw
ToImp(inputTree, FilterByProcess(inputTree, "1"))
ToImp(inputTree, FilterStatic(inputTree, AllButLast))

Figure 3.13: Example of Transformations Learned

on Learn new transformations, will display buttons that can be clicked to apply the corresponding transformation to all lemmas in every exploration node. In Figure 3.13, there are two possible transformations that are learned.

Overall, HST allows a user to see the exploration in the form of a graph and click on individual nodes to learn more about them. Additionally, a user can modify specific lemmas however they wish and use that as an example for the HST to learn how to transform every other lemma according to the user's changes. This allows the visualization to be catered to each individual user.

3.2 Summary

3.2.1 Inputting a Model

The first page you'll see on the HST is the model input page or the landing page, as shown in Figure 3.1. As a user, one has a number of options when inputting a model or model run. Firstly, a user can type in a model definition using SMTLib syntax. Another option is to upload an SMTLib file. A user can also upload a Spacer log file rather than having

HST run Spacer and produce a Spacer log. Users also have the option of custom naming for a run to help with organization.

In addition to choosing a method to interpret a model, the user can also specify options and flags that Spacer has. The user can use the available drop down menu which provides a number of options and their types whether it true/false, string, etc. There is also the option to paste in a Spacer delimited line of the options for those that have a larger set of options. Once a model is chosen, the user can use the Hit and Run button at the bottom of the page.

Lastly, this first page also features a list of previously run models in the user's HST session.

3.2.2 Overview of Visualization Page

Once a user clicks on Hit and Run, they are redirected to another page while HST begins the processing of the inputted model.

In the top left, the user will see two informational boxes, a line of text, and a series of five different buttons. The first box tells the user the status of HST, whether it is waiting for information from the backend or if the frontend is fully up to date with the status of the backend at the time of the poke, Poking Spacer... and Poking is done respectively. The second box is the status of Spacer whether that be running, `sat`, `unsat`, or `unknown`. The line of text is the exact Spacer command that was run, if applicable.

The five buttons below are Poke, Star, SatVis, PobVis, and Multi Select. The Poke button allows the user to check the current status of the Spacer run. Clicking on the Poke button will update the previously mentioned status boxes for HST and Spacer. The Star button opens a modal that shows the user all inductive invariant lemmas when applicable. The SatVis and PobVis buttons toggle between two possible views that the user can use to explore the run. The Multi Select button allows a user to toggle between looking at a single node and looking at two nodes.

3.2.3 HST Graph Views

When the user clicks on the Poke button, the Spacer log of the current run is pulled from the Spacer process running in the backend and interprets the log into a node graph. The Poke button allows the user to visualize the log without waiting for Spacer to complete its evaluation. The user can choose to view the log in one of two views, SatVis and PobVis, both of which are selected by clicking on its respective button in the top left corner of the screen. The SatVis view shows the full chronological exploration graph of each of the events included in Spacer's log. While this view is useful for visualizing Spacer's log in its entirety, it can be difficult to use for larger problems. The PobVis view includes a separate tree for each level of exploration and has a summary of all lemmas discovered through that particular depth of exploration. This allows for a more concise yet comprehensive understanding of what makes up the proof or the counterexample for the verification of a problem.

Clicking on a node in either graph will display boxes on the left side of the screen. The top box contains the node id, expression id, and the parent id. Below the ids, there is the state of the model for that node. If applicable, the second box provides a lemma summarization for the node selected. The lemma summarization includes lemmas identified before and after the exploration of the node selected.

3.2.4 User Edits to Lemmas

As a user, you can apply edits to the lemmas in the lemma summarization. Note, edits made will only allow a user to rearrange the lemma and therefore does not affect the meaning of the lemma. A user can do this by clicking on the Open Editor button on the bottom. Note, this button will only appear when Spacer's run on the problem is complete. Once the editor box is open, a user can choose one of the lemmas from the selected node to modify. Clicking on the lemma will generate a syntax tree of the lemma where a user can click on nodes of the tree to select sub statements of the lemma. The user can then choose from the buttons on the top to transform the lemmas as they see fit. Once the lemma is in a form you would like, you can click the Add to Learn button and both the original and the transformed versions of the lemma should be saved. You can see your saved, transformed lemmas in the box on the bottom right of the graph page. Once the user is satisfied with the amount of transformed lemmas saved (this number must be at least 1), the user can click Learn new transformations. Learn new transformations will ping the the Custom Prose API, which will be discussed in detail in Chapter 4 and return any programs found. Found programs can be applied to the entire problem's lemmas by clicking on the program. Keep in mind, newly transformed lemmas can be saved at any time but they will not affect the learned list of programs unless the user clicks on Learn new transformations after saving newly transformed lemmas.

3.2.5 Star Modal

In the PobVis view, nodes where inductive invariant lemmas are identified are denoted by using a star shaped node. The user can also click on the Star button in the top left to view all Star Nodes. This allows the user to view all inductive invariants in one place. The user can also choose between three ordering views. The user can also choose to order star lemmas in by grouping together similar lemmas or by grouping together lemmas by which level of exploration they were discovered at.

3.2.6 Variable Replacement

Spacer often produces its own variable names for the sake of simplicity, these variable names are often difficult to either decipher or keep track of when reading through a Spacer run in HST. HST also features a variable replacement map on the right hand side where users can input a custom dictionary mapping Spacer variables to whatever names they wish.

3.2.7 Prefix Notation to Infix Notation

The HST displayed the states as well as the lemmas in the form of s-expressions. While this may be easily readable to some, others might prefer these lemmas to use infix notation for better understand ability.

The conversion occurs in the backend using a custom parsing function along with and SMTLib parser library for .NET. The SMTLib parser converts each of the lemmas from prefix notation into a syntax tree. The parsing function will then traverse through the tree recursively to build out the equivalent lemma in infix notation form. This function can be applied to all lemmas in a given run by using a To Readable button on the HST run post processing page. The user also has to the option to revert back to prefix notation, if they wish, by clicking on the To Raw button on the same page, beside the To Readable button.

3.2.8 Diff Between Nodes

As mentioned previously, one of the five utility buttons at the top left is a Multi Select button which allows a user to simultaneously view the information of two different nodes.

To make a user's life easier, there is also a diff that is run between the two selected nodes, allowing a user to view possible discrepancies in the states of two similar nodes.

3.2.9 Ease of Use Features

HST allows a user to traverse either graph view with arrow keys to provide a more seamless experiencing while moving through nodes of the SPACER exploration graph. Any lemma that is in the form of a conditional expression has a difference in colour between the precedent and antecedent to make it easier to differentiate between the two.

3.3 Choosing Features

The features discussed above were chosen based on two main sources; related work and interaction with current users. The precursor to HST consisted of a graph where a user could click on a node and see the state as well as the lemma summarization of the node. This was a base feature that made sense to replicate in HST. From there, users expressed a number of feature requests including an ability to manipulate lemmas so that they may be presented differently as well as being able to see differences between two nodes. Based on user feedback, a number of features were added to HST throughout its development. From there, each feature was put through trial and error until it was decided that the feature either needed to be developed further or it was not needed.

Chapter 4

HST Programming By Example

4.1 Framework for Understanding Lemmas

In order to simplify the multiple types of transformations users can apply to lemmas, the system uses a three part transformation suite. Those three parts are denoted as variable transformations, literal transformations, and lemma transformations.

4.2 Transforming Variables

Transforming variables corresponds to renaming variables based on input/output examples of the user. For example, the user might want to rename all variables of the form `Inv_i_n` into `Var_i` by providing the following single input/output example: `Inv_1_n` $\mapsto$ `Var_1`. Since these transformations are done on the string representation of variable names, no special DSL is required. Instead, we reuse the DSL of FlashFill [19]. This transformation can be generalized into $\alpha\_x\_y- > \beta\_x\_\texttt{const}$.

These transformations allow the user to apply a string manipulation-esque transformation across all eligible variables in a Spacer run. Of course, what can be learned is limited to what is expected of FlashFill. For example, the learned transformation assumes that variables have some pattern in common and the transformation is a function of that pattern. In particular, not much can be learned and generalized from input/output examples that completely rename a variable. For example, if the user renames `Inv_0_n` into `State`, not much can be learned in addition to this single replacement because the renaming does not exhibit any particular pattern. On the other hand, given an example `Inv_1_n` $\mapsto$ `Var_1`, FlashFill can learn a general transformation corresponding to renaming `Inv` to `Var` and deleting the suffix `_n`. The learned transformation, will, for example, map `Inv_2_n` to `Var_2`.

4.3 Transforming Literals

The ideas presented in this section are not included in HST as of yet but is theoretical work. Similar to the transformation of lemmas 4.4, the transformation of literals involves the movement of terms within the literal. The key difference in transforming literals lies in the equal sign. The grammar would be similar to the grammar for lemma transformation with the addition of keeping track of the position of the literal segment in relation to the equal sign. The change in grammar would lead to a slightly different execution. This ensures that any literal segment that is moved to opposite side of the equal sign is made negative as per the algebraic manipulation of equivalent equations.

4.4 Transforming Lemmas

4.4.1 DSL & Semantics

```
Node outputTree := ToImp(inputTree, leftSide)
List<int> leftSide := filter | JoinFilters(filter, filter)
List<int> filter := FilterByName(inputTree, name)
                  | FilterByArrayIndex(inputTree, index)
                  | FilterStatic(inputTree, type)
```

The grammar above describes the transformation from a disjunction of literals to a conditional statement. The current grammar only supports the disjunction of literals as input. The input, `inputTree`, is a custom tree structure that represents SMTLib statements as a recursive structure. Each literal in the input is assigned an integer index value as per the order the literals appear in `inputTree`. Based on what numbers make up the `leftSide`, the corresponding literals are negated and become the antecedent of the conditional statement while the remaining literals become the consequent. Users have the option of filtering by variable name, filtering by an index in an array, filtering by negative literals, or filtering by every literal except the final one. Filtered literals will always become the antecedent of the conditional statement. A user also has the option to combine up to two of the filters mentioned previously. Note that the transformation does not modify the meaning of any given lemma.

LeftSide

LeftSide is a list comprised of integers that each correspond to a literal in a given lemma. It is calculated based on three possible filters that can be combined or used individually.

- **FilterByName:** Every literal with the given variable name becomes a part of the consquent. Possible variables to filter by are restricted to the variables that are in the lemma. If FilterByName is given a variable that is not found in the lemma, it has no effect.
- **FilterByArrayByIndex:** The function is given an index of an array that is present in the one or more literals. The indexes of corresponding literals become a part of leftSide.
- **FilterStatic:** Consists of two possible static functions
  - **FilterByNot:** Indexes of negative literals (contains a `not` at the beginning of the literal) become a part of leftSide.
  - **FilterAllButLast:** Every index except the last that of the last child becomes a part of leftSide

4.4.2 Witness Functions

Each semantic function can be paired with an optional witness function. In order to verify that a program is correct, witness functions are used to back-propogate a function using an input-output example. Each function is written for a variable parameter of a function. Given the input-output example, the function should return viable parameters that when applied to the function with the input, produce the output.

ToImp

ToImp has one corresponding witness function for the second parameter of the function, `leftSide`. For this function, the witness function would be given and `inputTree` and an `outputTree` as spec and we would expect it to produce valid input for `leftSide`, that is we expect it to return a list of integers that when given to the ToImp function along with `inputTree`, `outputTree` is returned. This is achieved by generating the possible lists that could satisfy this particular witness function based on how many literals make up the precedent in `outputTree`. Each of those lists of integers are then inputted as the `leftSide` parameter of the ToImp function along with the `inputTree` available in the spec. The resulting tree is compared against the `outputTree` in the spec to check if they are equivalent. Those that are equivalent are stored in a global example spec for the function. Once every possible list of integers is checked, the local example spec is returned.

JoinFilters

JoinFilters has two witness functions, one for each of the filter parameters. For each filter, we want to ensure that the combined result of both filters produces the resulting `outputTree`. Therefore, both witness functions work together in the sense that one must fill in the parts of the lemma that the other accounts for. For example, given the lemma below:

$$(State[2] <= 0) \ \vee \ (\neg(Serve = Num[2])) \vee$$
$$(State[3] <= 0) \ \vee \ (\neg(Serve = Num[3]))$$

Where each literal in the lemma has a corresponding indices based on its position in the lemma. The witness functions for both filters should ensure that the second filter is the difference between the resulting list and the first filter.

FilterByName, FilterByArrayByIndex, FilterStatic

Each of the filter functions return a list of integers where each integer corresponds to the index of the literal in the original lemma. The witness function for each filter function produces a filtered list by each of the possible identifiers in the lemma and checks it against the expected list of integers. For example, given a lemma that has four different variable names, the witness function for FilterByName would generate that list of four variables, compute the resulting list of integers from FilterByName with each of the variables and return those that match the expected list of integers.

$$(State[2] <= 0) \ \vee \ (\neg(Serve = Num[2])) \vee$$
$$(State[3] <= 0) \ \vee \ (\neg(Serve = Num[3]))$$

Given the above equation, FilterByName has three possible identifiers, `State`, `Serve`, and `Num`. Say our expected list is `[0,2]`. The witness function would then generate that list using each of those identifiers on FilterByName. The respective lists for `State`, `Serve`, and `Num` would be `[0,2]`, `[1,3]`, and `[1,3]`. As we can see, the only identifier that matches our expected output is the list produced by `State` so the witness function would return a mapping with `State`.

4.4.3 Ranking Score

We chose to tune the Ranking Score for the above grammar by putting a preference on longer programs. This was done by compounding the score for every function in the final program.

Chapter 5

Case Studies

To create a useful grammar, the following case studies are presented for which the grammar provides a comprehensive set of transformations that can be applied to the lemmas created by Spacer. In this chapter, we demonstrate that the visualizer and grammar work well on a realistic example.

5.1 *Simple Bakery*

Simple bakery [36] is one implementation of the Mutual Exclusion algorithm [10]. Mutual exclusion is a property of a system that is used to prevent race conditions and deadlock by controlling concurrency.

Simple bakery was chosen as a case study because as a basic form of mutual exclusion, it is simple to model in SMT-based model checking software yet it is a problem that is not extremely well researched in terms of how model checkers find the solution and what that looks like visually. In an attempt to gain a better understanding of *simple bakery* and its proof as presented by Spacer, an implementation of *simple bakery* was created to be fed into Spacer. The implementation in question, is restricted to three processes. This decision was made in order for Spacer to come to a conclusion on the satisfiability of the problem. Spacer treats the latter as a problem with an unknown outcome. In other words, Spacer will return a `sat` value of `unknown` because it is unable to identify the symmetry between arbitrary processes. In practice, *simple bakery* is a confirmed satisfiable problem.

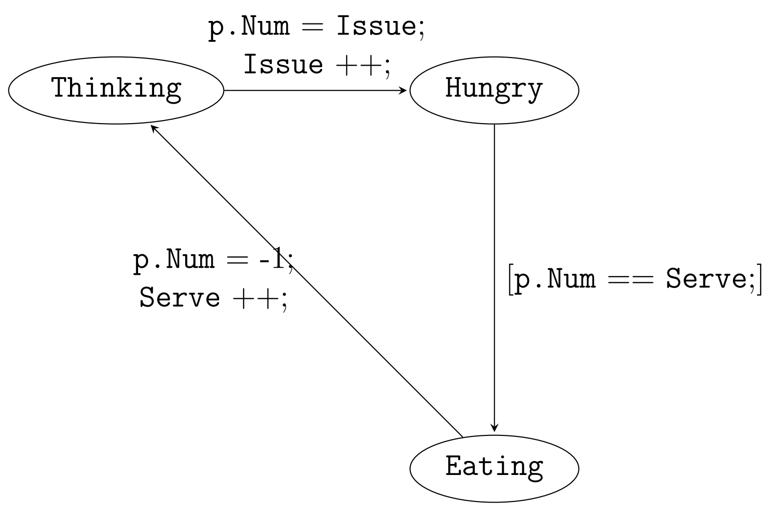
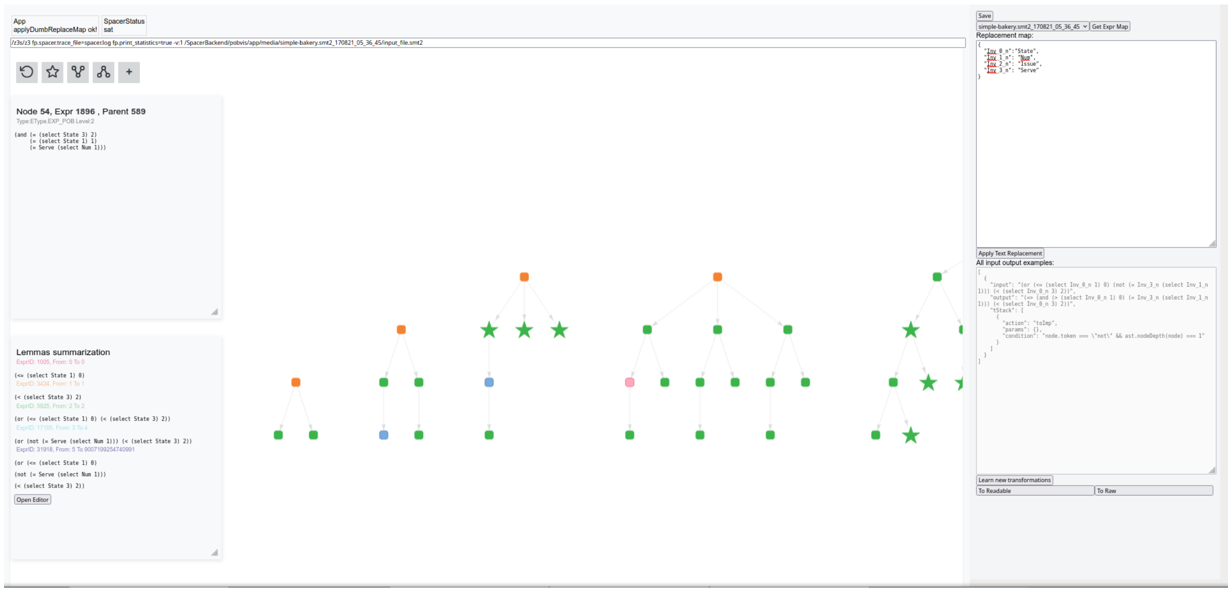


Figure 5.1: *Simple Bakery* state machine for a process `p`. Square brackets indicate a guard. Every state has an implicit self loop (not shown).

5.1.1 Description of *Simple Bakery*

Lamport's bakery algorithm [36], also known as *simple bakery*, is a well known implementation of a mutual exclusion protocol. Intuitively, it simulates a line in a bakery: each customer gets a number and only one number is served at any given time. In this section, we will explain the implementation that is inputted into HST. The implementation works as follows: there are two global counters: `Issuer` and `Served`. Both counters are initially 0. Processes that participate in the implementation can be in one of three states: `Thinking`, `Hungry`, or `Eating`. All processes begin in the `Thinking` state. When a process is `Thinking`, it has a `Num` value of -1. A `Thinking` process can non-deterministically transition to `Hungry`. Once a process is `Hungry`, it is given a `Num` value equivalent to the current value of `Issuer` and `Issuer` increments by 1. This operation happens atomically through the operating system. When a process is `Hungry`, it continues to be `Hungry` until its `Num` matches the current value of `Served`, at which point it can transition to `Eating`. In theory, a process that is `Eating` can non-deterministically transition back to `Thinking`, at which point, its `Num` would also be reset to -1 and `Served` would increment by 1. This operation is done atomically. This is illustrated in Figure 5.1. In practice, however, the non-determism of a process completing its `Eating` can cause a hang in the system and can be solved by introducing a timeout.



```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_170821_05_36_45/input_file.smt2
```

Node 54, Expr 1896 , Parent 589
Type:EType.EXP_POB Level:2

```
(and (= (select State 3) 2)
     (= (select State 1) 1)
     (= Serve (select Num 1)))
```

Lemmas summarization
ExprID: 1005, From: 0 To 0

```
(<= (select State 1) 0)
```

ExprID: 3424, From: 1 To 1

```
(< (select State 3) 2)
```

ExprID: 5925, From: 2 To 2

```
(or (<= (select State 1) 0) (< (select State 3) 2))
```

ExprID: 17185, From: 3 To 4

```
(or (not (= Serve (select Num 1))) (< (select State 3) 2))
```

ExprID: 31918, From: 5 To 9007199254740991

```
(or (<= (select State 1) 0)
(not (= Serve (select Num 1)))
(< (select State 3) 2))
```

simple-bakery.smt2_170821_05_36_45
Replacement map:

```
{
  "Inv_0_n":"State",
  "Inv_1_n": "Num",
  "Inv_2_n": "Issue",
  "Inv_3_n": "Serve"
}
```

All input output examples:

```
[
  {
    "input": "(or (<= (select Inv_0_n 1) 0) (not (= Inv_3_n (select Inv_1_n
1))) (< (select Inv_0_n 3) 2))",
    "output": "(=> (and (> (select Inv_0_n 1) 0) (= Inv_3_n (select Inv_1_n
1))) (< (select Inv_0_n 3) 2))",
    "tStack": [
      {
        "action": "toImp",
        "params": {},
        "condition": "node.token === \"not\" && ast.nodeDepth(node) === 1"
      }
    ]
  }
]
```

Figure 5.2: Replacing Variables with Custom Values

5.1.2 Running *Simple Bakery* on HST

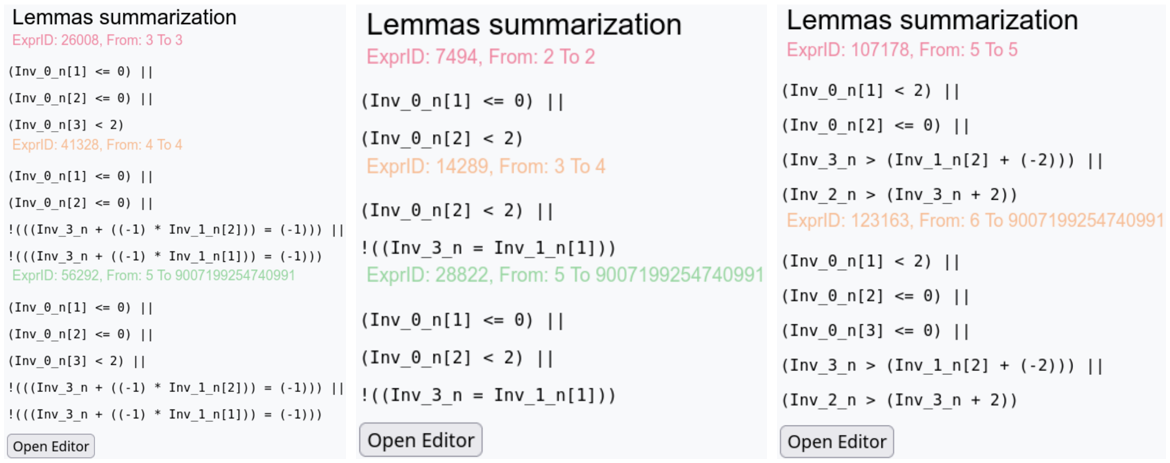
Through the evaluation *simple bakery* in HST, we are checking to see that the model satisfies the mutual exclusion property. For this particular problem, that property can be described as, for any distinct processes p and q, if p is `Eating` then q is not `Eating`

$$\forall p, q \cdot p \neq q \implies (\texttt{p.State} = 2 \implies \texttt{q.State} \neq 2)$$

The *simple bakery* implementation mentioned in 5.1.1 was entered into the visualizer with no additional options. For this particular problem, SPACER is able to show that the property is satisfied (i.e., returns `sat`). For this reason, we focus on the POBVIS view to identify inductive invariants found by SPACER.

Based on the ordering of the lemmas, we first apply the variable replacement to all of the lemmas. For this particular model, we have four different variables: `p.State`, `p.Num`, `Issue`, and `Serve`. We can do this by creating a dictionary mapping in the REPLACEMENT MAP.

Next, we want to get a basic idea of what kind of lemmas are learned by SPACER through the exploration. We can do this by clicking on a few nodes and looking at resulting lemmas. By default, lemmas are presented with prefix notation. We can toggle between prefix notation and infix notation for better readability.


Lemmas summarization

ExprID: 26008, From: 3 To 3

```
(Inv_0_n[1] <= 0) ||
(Inv_0_n[2] <= 0) ||
(Inv_0_n[3] < 2)
```

ExprID: 41328, From: 4 To 4

```
(Inv_0_n[1] <= 0) ||
(Inv_0_n[2] <= 0) ||
!(((Inv_3_n + ((-1) * Inv_1_n[2])) = (-1))) ||
!(((Inv_3_n + ((-1) * Inv_1_n[1])) = (-1)))
```

ExprID: 56292, From: 5 To 9007199254740991

```
(Inv_0_n[1] <= 0) ||
(Inv_0_n[2] <= 0) ||
(Inv_0_n[3] < 2) ||
!(((Inv_3_n + ((-1) * Inv_1_n[2])) = (-1))) ||
!(((Inv_3_n + ((-1) * Inv_1_n[1])) = (-1)))
```

Lemmas summarization

ExprID: 7494, From: 2 To 2

```
(Inv_0_n[1] <= 0) ||
(Inv_0_n[2] < 2)
```

ExprID: 14289, From: 3 To 4

```
(Inv_0_n[2] < 2) ||
!((Inv_3_n = Inv_1_n[1]))
```

ExprID: 28822, From: 5 To 9007199254740991

```
(Inv_0_n[1] <= 0) ||
(Inv_0_n[2] < 2) ||
!((Inv_3_n = Inv_1_n[1]))
```

Lemmas summarization

ExprID: 107178, From: 5 To 5

```
(Inv_0_n[1] < 2) ||
(Inv_0_n[2] <= 0) ||
(Inv_3_n > (Inv_1_n[2] + (-2))) ||
(Inv_2_n > (Inv_3_n + 2))
```

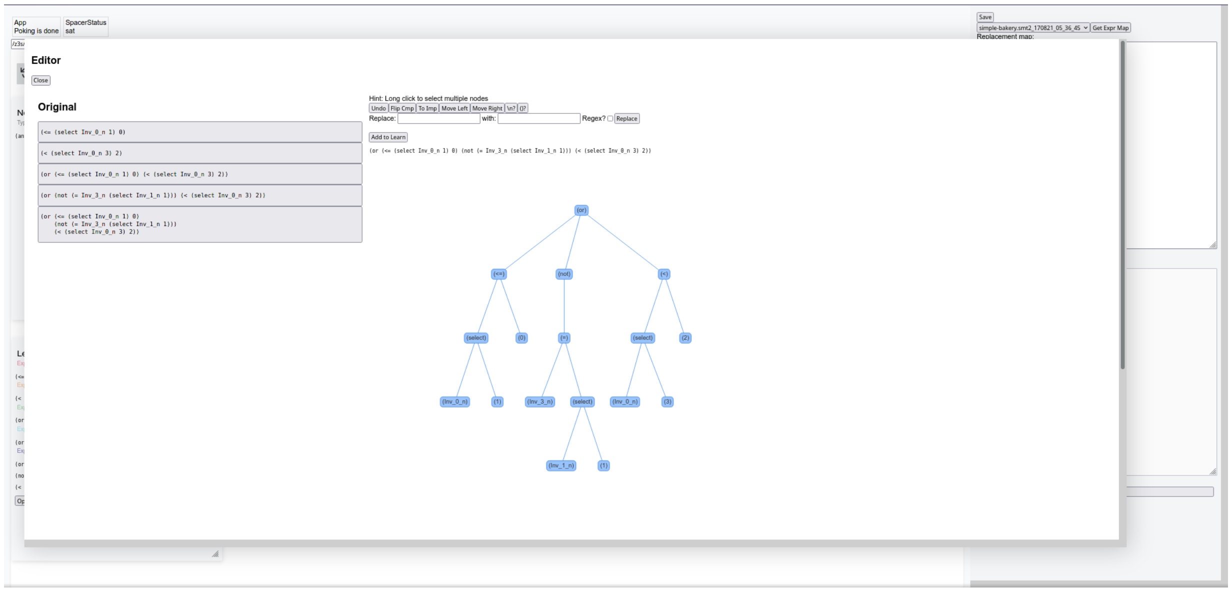
ExprID: 123163, From: 6 To 9007199254740991

```
(Inv_0_n[1] < 2) ||
(Inv_0_n[2] <= 0) ||
(Inv_0_n[3] <= 0) ||
(Inv_3_n > (Inv_1_n[2] + (-2))) ||
(Inv_2_n > (Inv_3_n + 2))
```

Figure 5.3: *Simple Bakery* Transformation in HST

SPACER produces lemmas of the problem as a disjunction of literals. For someone who is looking to interpret the model, this is not the easiest form to understand nor work with. The goal is to convert the disjunction of literals into a conditional statement where literals become part of either the antecedent or the consequent. The transformed format allows the user to more easily draw conclusions about the variables and literals in each lemma.

The next thing we want to do is to manipulate at least one lemma so that a transformation program can be synthesized and applied to all other lemmas, making them look like what we'd expect to see in the solution of this problem. For this particular problem, I found that utilising the FilterStatic method with the AllButLast parameter, gives the most useful form of each lemma. To do this, I can click on any given node and click on the OPEN EDITOR button. This opens a window that allows the user to choose a lemma and generate the syntax tree for the lemma. By clicking on top level nodes of each literal in the lemma, the user can select everything but the last literal and click on the TOIMP button to convert this lemma into a conditional statement. Once the user is satisfied with the format of the lemma, they can click on ADD TO LEARN and I now have an input/output example to feed to the back-end. The resulting transformation options can be seen in Figure 5.6. As mentioned previously, we want to apply the `FilterStatic(inputTree, AllButLast)` transformation. Once applied, we can click into the STAR MODE to see all inductive invariants according to the transformation applied.



Original

```
(<= (select Inv_0_n 1) 0)
```

```
(< (select Inv_0_n 3) 2)
```

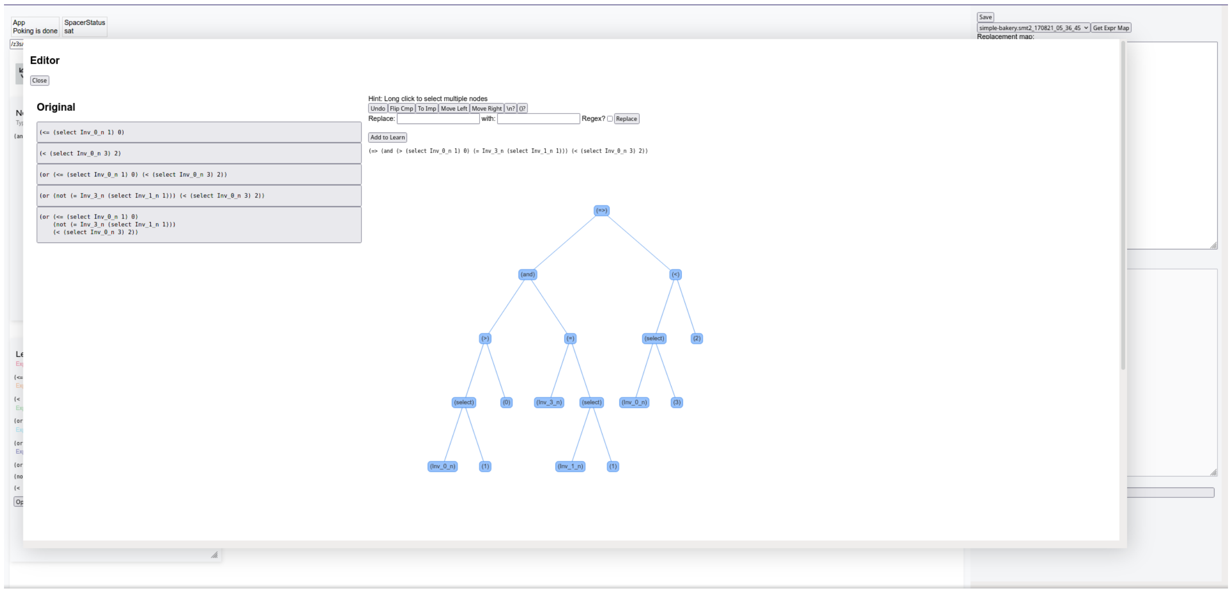
```
(or (<= (select Inv_0_n 1) 0) (< (select Inv_0_n 3) 2))
```

```
(or (not (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
```

```
(or (<= (select Inv_0_n 1) 0)
    (not (= Inv_3_n (select Inv_1_n 1)))
    (< (select Inv_0_n 3) 2))
```

```
(or (<= (select Inv_0_n 1) 0) (not (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
```

Figure 5.4: Open Editor Box



Original

```
(<= (select Inv_0_n 1) 0)
```

```
(< (select Inv_0_n 3) 2)
```

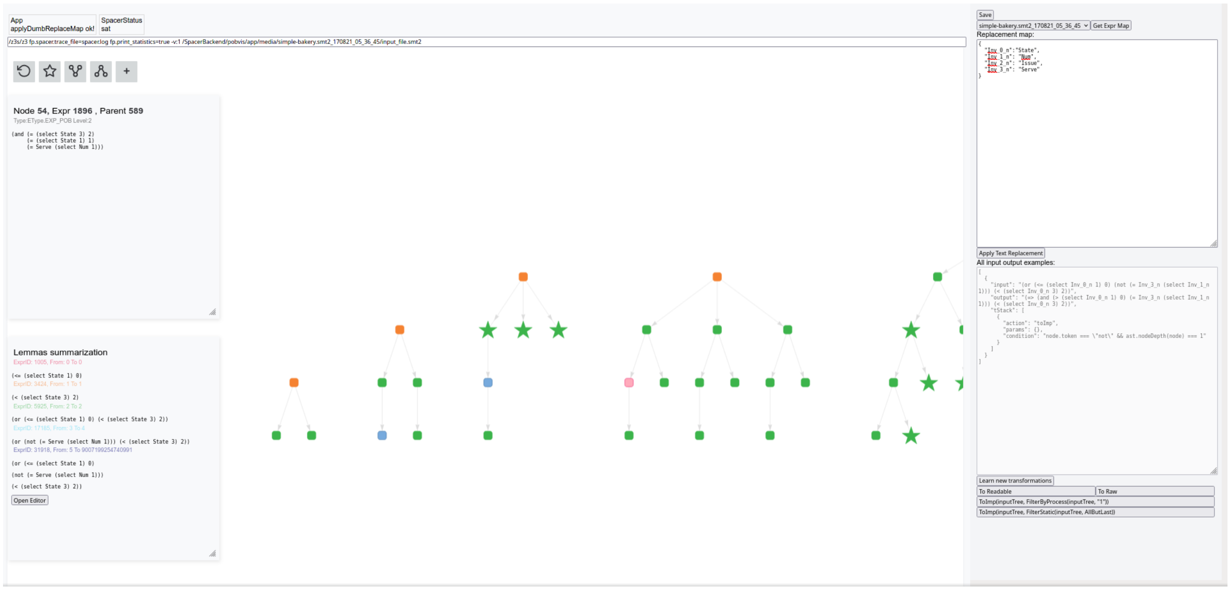
```
(or (<= (select Inv_0_n 1) 0) (< (select Inv_0_n 3) 2))
```

```
(or (not (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
```

```
(or (<= (select Inv_0_n 1) 0)
    (not (= Inv_3_n (select Inv_1_n 1)))
    (< (select Inv_0_n 3) 2))
```

```
(=> (and (> (select Inv_0_n 1) 0) (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))
```

Figure 5.5: Edited Lemma in Open Editor Box



```
/z3s/z3 fp.spacer.trace_file=spacer.log fp.print_statistics=true -v:1 /SpacerBackend/pobvis/app/media/simple-bakery.smt2_170821_05_36_45/input_file.smt2
```

Node 54, Expr 1896 , Parent 589

Type:EType.EXP_POB Level:2

```
(and (= (select State 3) 2)
     (= (select State 1) 1)
     (= Serve (select Num 1)))
```

Lemmas summarization

ExprID: 1005, From: 0 To 0

```
(<= (select State 1) 0)
```

ExprID: 3424, From: 1 To 1

```
(< (select State 3) 2)
```

ExprID: 5925, From: 2 To 2

```
(or (<= (select State 1) 0) (< (select State 3) 2))
```

ExprID: 17185, From: 3 To 4

```
(or (not (= Serve (select Num 1))) (< (select State 3) 2))
```

ExprID: 31918, From: 5 To 9007199254740991

```
(or (<= (select State 1) 0)
(not (= Serve (select Num 1)))
(< (select State 3) 2))
```

simple-bakery.smt2_170821_05_36_45

Replacement map:

```
{
  "Inv_0_n":"State",
  "Inv_1_n": "Num",
  "Inv_2_n": "Issue",
  "Inv_3_n": "Serve"
}
```

All input output examples:

```
[
  {
    "input": "(or (<= (select Inv_0_n 1) 0) (not (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))",
    "output": "(=> (and (> (select Inv_0_n 1) 0) (= Inv_3_n (select Inv_1_n 1))) (< (select Inv_0_n 3) 2))",
    "tStack": [
      {
        "action": "toImp",
        "params": {},
        "condition": "node.token === \"not\" && ast.nodeDepth(node) === 1"
      }
    ]
  }
]
```

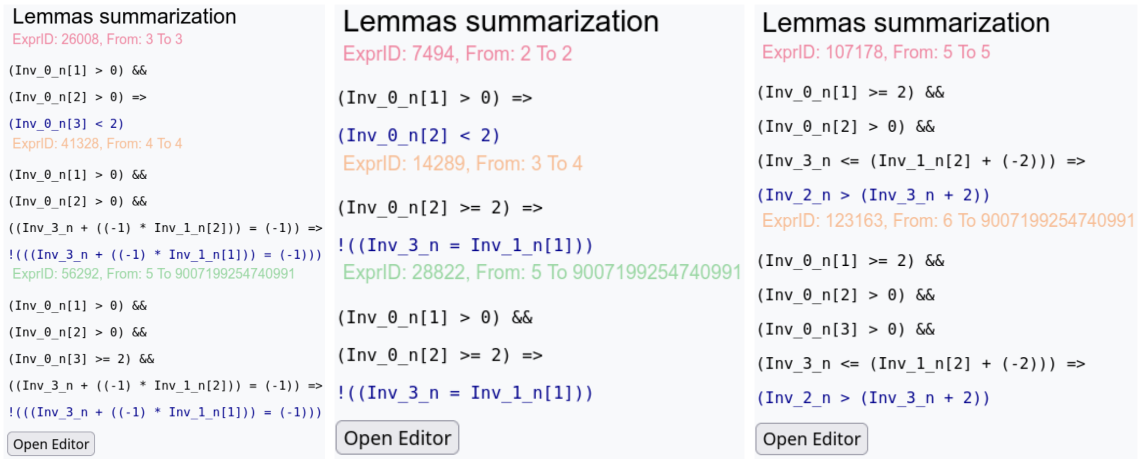
```
ToImp(inputTree, FilterByProcess(inputTree, "1"))
ToImp(inputTree, FilterStatic(inputTree, AllButLast))
```

Figure 5.6: Example of Transformations Learned

5.1.3 Expected versus Actual Results

According to the book by Chandy and Misra [8], we're looking for lemmas that fall into the following solution conditions. The following invariants have been modified slightly from [8] according to the implementation being used with Spacer. The problem remains the same through these modifications:

1. If a process, p, is thinking (i.e., $\texttt{p.State} = 0$) then $\texttt{p.Num} = -1$;
2. If a process, p, is hungry (i.e., $\texttt{p.State} = 1$) then either $\texttt{p.Num} = -1$ or $(\texttt{Serve} \leq \texttt{p.Num} < \texttt{Issue})$;
3. If a process, p, is eating (i.e., $\texttt{p.State} = 2$) then $\texttt{p.Num} = \texttt{Serve}$ and any other process, q, is not eating;
4. If two distinct processes, p and q, have the same Num then $\texttt{p.Num} = \texttt{q.Num} = -1$;
5. For any x with $\texttt{Serve} \leq x < \texttt{Issue}$, there exists a process s.t. $\texttt{p.Num} = x$ for that process.



Lemmas summarization

```
ExprID: 26008, From: 3 To 3
(Inv_0_n[1] > 0) &&
(Inv_0_n[2] > 0) =>
(Inv_0_n[3] < 2)
ExprID: 41328, From: 4 To 4
(Inv_0_n[1] > 0) &&
(Inv_0_n[2] > 0) &&
((Inv_3_n + ((-1) * Inv_1_n[2])) = (-1)) =>
!(((Inv_3_n + ((-1) * Inv_1_n[1])) = (-1)))
ExprID: 56292, From: 5 To 9007199254740991
(Inv_0_n[1] > 0) &&
(Inv_0_n[2] > 0) &&
(Inv_0_n[3] >= 2) &&
((Inv_3_n + ((-1) * Inv_1_n[2])) = (-1)) =>
!(((Inv_3_n + ((-1) * Inv_1_n[1])) = (-1)))
```

Open Editor

Lemmas summarization

```
ExprID: 7494, From: 2 To 2
(Inv_0_n[1] > 0) =>
(Inv_0_n[2] < 2)
ExprID: 14289, From: 3 To 4
(Inv_0_n[2] >= 2) =>
!((Inv_3_n = Inv_1_n[1]))
ExprID: 28822, From: 5 To 9007199254740991
(Inv_0_n[1] > 0) &&
(Inv_0_n[2] >= 2) =>
!((Inv_3_n = Inv_1_n[1]))
```

Open Editor

Lemmas summarization

```
ExprID: 107178, From: 5 To 5
(Inv_0_n[1] >= 2) &&
(Inv_0_n[2] > 0) &&
(Inv_3_n <= (Inv_1_n[2] + (-2))) =>
(Inv_2_n > (Inv_3_n + 2))
ExprID: 123163, From: 6 To 9007199254740991
(Inv_0_n[1] >= 2) &&
(Inv_0_n[2] > 0) &&
(Inv_0_n[3] > 0) &&
(Inv_3_n <= (Inv_1_n[2] + (-2))) =>
(Inv_2_n > (Inv_3_n + 2))
```

Open Editor

Figure 5.7: *Simple Bakery* Transformation in HST

Based on the transformation described in section 5.1.1, we expect that transformed lemmas that are identified as inductive invariants, will describe one of the above solution conditions. Note that non transformed lemmas still describe one of the above solution conditions since the transformation does produce an equivalent lemma in a different format. However, we expect that the user chooses to transform lemmas to make it easier to identify which solution condition a lemma describes. An example lemma transformation would be the following

$$\begin{aligned}&(Inv\_0\_n[2] <= 0) \ \vee \ (\neg(Inv\_3\_n = Inv\_1\_n[2])) \ \vee \\ &(Inv\_0\_n[3] <= 0) \ \vee \ (\neg(Inv\_3\_n = Inv\_1\_n[3]))\end{aligned}$$

to

$$\begin{aligned}&(State[2] > 0) \ \wedge (Serve = Num[2]) \ \wedge (State[3] > 0) \ \rightarrow \\ &\neg((Serve = Num[3]))\end{aligned}$$

Though the above two lemmas are equivalent in meaning, it is easier to see that the second lemma is in the form of solution condition #3, stating that if one process is `Eating` then any other process must not be `Eating` for the user who chose to use this transformation. Similarly, the user can then use the transformation learned from this particular example and apply it to all other lemmas to be able to see that every inductive invariant produced by SPACER, falls into one of the above solution conditions. The user doesn't necessarily have to use the above input/output example to be able to better understand *simple bakery*. The user can manipulate the lemmas however they see fit in order to cater their HST experience to themselves.

Chapter 6

Related Work

While there have been a number of tools that describe the visualization of various solvers as well as a number of tools that simplify text through program synthesis, HST introduces the idea of visualizing Spacer while also catering the usage of the visualizer to each individual user through program synthesis.

6.1 Visualizing Model Checkers

The first tool relevant to our work attempted to improve the understanding of parallel constraint solvers that are SMT-based. In this paper, Budakovic et al. [7] created a tool that allows a user to retrieve information about the execution of the partitioning tree parallelization approach at any given point of time. Both the tool in [7] as well as HST attempt to tackle the problem of visualizing tree-like structures that are often found in SMT solvers, IC3/PDR algorithms, SAT solving, etc due to the use of parallel computing. The tool comprises six different views, each modifiable by various selections chosen by the user. The user would then be able to visualize this tree and determine the satisfiability or unsatisfiability of the model. Parallelization trees are constructed such that the root of the tree is an AND-node which represents the input instance. Every AND-node represents a constraint problem and potentially multiple constraint solvers. Given the instances in the AND-node, if one is satisfiable then the input instance is satisfiable. Alternatively, if one of the AND-node's sub-trees has a child which is unsatisfiable or one of the constraint solvers on the instance of the AND-node is unsatisfiable then the input instance is unsatisfiable. While their tool was fully proposed in this paper, Budakovic et al. do not evaluate the effectiveness of their tool. They merely propose that the tool will "help users to be more confident with parallel computing techniques, encouraging the community to develop new efficient decision procedures for constraint solving".

The second relevant tool is an extension of an existing z3 visualizer, created by Microsoft Research, called the VCC Axiom Profiler [4], and was used to help understand the quantifier instantiations that occur in an SMT run. The paper most significantly contributed an implementation that identifies and explains previously unknown matching loops which was confirmed by experimenting on 34 159 SMT files. While this does not perfectly match up with what we are doing in creating HST, both tools are similar since they both utilise visualization to identify patterns in SMT solvers. Part of finding these matching loops was a visualizer that takes in a z3 log file. From the file, the visualizer allows the user to visualize the raw data of the quantifier instantiation, (inherited from the VCC Axiom Profiler), an instantiation graph that displays the causal relationships between quantifier instantiations, making high-branching and long-sequence scenarios identifiable. Each node represents a quantifier instantiation and each quantifier has its own colour. The visualizer has various options (like maximum depth control, expand or collapse children of a node, node with the highest number of children, etc.) in order to help the user identify quantifiers that could cause problems. Through many different input tests, the tool "allows a to effectively explore and understand the quantifier instantiations by an SMT run, their connections and potentially-problematic patterns which arise (e.g. due to matching loops)". Their visualizer has also shown to help users analyze more complicated SMT models.

6.2 Programming By Example with Prose

There are a number of papers that outline the development of Prose [19, 3, 20, 21, 45, 23, 22, 24, 26, 28] and the applications of Prose [27, 39, 25, 51, 33, 9, 55, 2, 46, 38, 49, 12, 54, 32, 42], to reference a few. Of the papers, we found that the work done by Pan et al. [44] and Rolim et al. [49] was most relevant to the work we have done.

Rolim et al. focuses on an application of Prose called Refazer. Refazer is a technique that produces program transformations automatically based on user input. Similar to HST, Refazer produces program transformations by using a custom domain-specific language, domain-specific deductive algorithms for synthesizing transformations in the DSL, and functions for ranking the synthesized transformations. The custom domain-specific language for both HST and Refazer are based on making changes to an abstract syntax tree where Refazer applies changes to code written in Python and C# and HST applies changes to lemmas in disjunction of literals form. However, Refazer's goal is to apply changes to the code while HST makes changes that produce a logically equivalent statement. Refazer was evaluated by applying it on two different domains. Refazer was first given code edits made by students to fix errors in their assignment submissions. Based on those code edits, Refazer synthesized program transformations that would allow other students with the same errors in their assignments to have the correct changes be automatically applied. Based on 4 programming tasks done by 720 students, Refazer helped correctly modify 87% of student submissions. The second domain Refazer was applied to was 3 large open-source projects that were in need of repetitive edits throughout the code base. Normally, developers would need to make many of the same code edits in multiple places in the code base. With Refazer, 83% of code edits on 59 different scenarios were learned correctly using only 2.8 examples on average.

Another tool that was relevant to our work was the Merge Conflict Resolution tool created by Pan et al. [44]. The merge between the code base for Microsoft Edge and its fork source, Chromium has conflicts that are difficult to merge automatically. Upon analysis, it is noted that most conflicts can be resolved through 1-2 line changes and many are similar. Thus, similar changes can be learned through program synthesis with the use of a custom DSL. Through this custom DSL, 11.4% of conflicts (41% of 1-2 line changes) can be resolved with 93.2% accuracy. Similar to Refazer as well as HST, the Merge Conflict Resolution tool consists of a custom domain-specific language, domain-specific deductive algorithms for program synthesis and ranking functions. What is unique about the Merge Conflict Resolution tool is the use of recursion in the domain-specific language. In Merge Conflict Resolution, they use recursion in multiple parts of the DSL including to concatenate two transformations to make one larger transformation that can then be applied to the merge condition. Similarly, we use recursion to combine two filters to make one large filter to be applied to the conversion between a disjunction of literals and an equivalent conditional statement.

The work presented in this thesis combines different aspects of each of the related works discussed above to create a completely different tool in terms of uses as well as techniques.

Chapter 7

Conclusion

In this thesis, we study the problem of developing visual aids to help understand model checking results. This is an extremely difficult task since it is difficult to define the problem formally. Specifically, we develop the first visualizer, called Hubble Spacer Telescope (HST), for an SMT-based model checker SPACER. The visualizer address two major issue. First, it presents an intermediate run of the model checker as a tree of explored configurations. The user is able to examine the trees of different runs, examine the details of the formulas corresponding to each node, and the lemmas that are learned by the model checker to block each node. In our evaluation, we find that the view is very useful at quickly assessing the quality of the model checker run. For example, whether the model checker is converging, diverging, potential reasons for divergence, and potential symmetry in the proof being constructed. We believe that this visual aids will be useful both in diagnosing hard problems and in future improvements to the model checking algorithm. The second issue addressed by HST is to help the user of HST automate presentation of complex formulas. This is a common problem to many logic-based, and more generally, symbolic techniques. The tools often present information in a form convenient to the algorithm, and usually that form might be very difficult for humans to understand. Automatically making formulas "easy" to understand seems like a daunting task. Instead, we focus by learning from examples, what a user considers "easy" and transforming the rest of the presented formulas to match that style. Our approach is based on the Prose framework that has been successfully used in similar tasks in Microsoft Excel FLASHFILL and in program repair tasks [19]. By developing a specialized DSL for formula-transformations, we are able to use Prose to learn meaningful simplification rules based on user examples. In our case study, we show that this can significantly reduce the necessary manual user interactions to convert an automatically generated proof of Lamport's Bakery Protocol into an easy-to-understand form.

7.1 Limitations & Future Work

While Prose is a very powerful framework, it is limited by the way the code is setup. It also takes a significant amount of knowledge on the framework in order to use it to its full potential. The key lies in the witness functions and how they are defined. Throughout the process of building HST, a large amount of time was spent on fine tuning the grammar into what it is now. I would expect that any changes to the grammar, no matter how small, would require some time into how the changes alter the effectiveness of the grammar, specifically its witness functions. Additionally, grammars with deeply nested programs are difficult to maintain and optimize. Shallow programs may suffice for our current program needs but it may be wise to move away from Prose to either another program synthesis framework or make our own. This is something that could be explored in future work.

Another limitation that boxes HST is the amount of users and case studies that have contributed to its development. The grammar that has been created has been heavily influenced by the case study mentioned in Chapter 5. While we have kept the grammar as general as possible, it would benefit future users of HST to use more case studies and examples to improve the grammar so that it can be effectively used on a variety of Spacer runs. Additionally, HST would benefit from more feedback. Throughout the process of creating HST, we have received feedback on it from a few peers and professors. The feedback has been essential to creating HST and building into what it has become. Therefore, more feedback would help develop the tool.

As of right now, HST is not very scalable. The graph views can only effectively produce smaller to medium sized graphs. Anything with a larger number of exploration trees would be very small on the graph views and would require a user to zoom in to gain any information. Additionally, it would take very long to load in a larger Spacer log file. Currently, we allow a user to visualize a partially complete Spacer run based on the intermediate log. However, load time tends to increase with the size of the log file, which means a user can be waiting a while before they see anything of use. Potential future work on this could be to visualize portions of the run instead of the whole thing. This could include more views as well as partitioning of data according to what the user deems important.

References

[1] Umair Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: For the student programs, from the student programs. In *40th International Conference on Software Engineering (ICSE), Software Engineering Education and Training Track*, February 2018.

[2] Shaon Barman, Sarah Chasins, Ras Bodik, and Sumit Gulwani. Ringer: Web automation by demonstration. In *OOPSLA 2016*, November 2016.

[3] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Ben Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *PLDI '15 Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Microsoft Research Technical Report, April 2014. Distinguished Artifact Award.

[4] Nils Becker, Peter Müller, and Alexander J. Summers. The axiom profiler: Understanding and debugging smt quantifier instantiations. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–116, Cham, 2019. Springer International Publishing.

[5] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. Explaining counterexamples using causality. *Formal Methods Syst. Des.*, 40(1):20–40, 2012.

[6] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.

[7] Jelena Budakovic, Matteo Marescotti, Antti E. J. Hyvarinen, and Natasha Sharygina. Visualizing smt-based parallel constraint solving. 2017.

[8] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1st edition, 1988.

[9] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *ICSE '16 Proceedings of the 38th International Conference on Software Engineering*, pages 345–356. Association for Computing Machinery, May 2016.

[10] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, September 1965.

[11] Ian Drosos, Titus Barik, Philip J. Guo, Rob DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *ACM CHI Conference on Human Factors in Computing Systems*, April 2020. Best Paper Award.

[12] Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. In *IJCAI 2017*, May 2017.

[13] Anna Fariha, Ashish Tiwari, Alexandra Meliou, Arjun Radhakrishna, and Sumit Gulwani. Coco: Interactive exploration of conformance constraints for data understanding and data cleaning. In *2021 International Conference on Management of Data*, June 2021.

[14] Anna Fariha, Ashish Tiwari, Arjun Radhakrishna, and Sumit Gulwani. Extune: Explaining tuple non-conformance. In *2020 International Conference on Management of Data*, pages 2741–2744. ACM, June 2020.

[15] Anna Fariha, Ashish Tiwari, Arjun Radhakrishna, Sumit Gulwani, and Alexandra Meliou. Conformance constraint discovery: Measuring trust in data-driven systems. In *2021 International Conference on Management of Data*, June 2021.

[16] Molly Q Feldman, Ji Yong Cho, Monica Ong, Sumit Gulwani, Zoran Popovic, and Erik Andersen. Automatic diagnosis of students' misconceptions in k-8 mathematics. In *2018 ACM Conference on Human Factors in Computing Systems (CHI)*, January 2018.

[17] Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachi Nagappan, and Ashish Tiwari. Feedback-driven semi-supervised synthesis of program transformations. In *OOPSLA*. ACM, October 2020.

[18] Bernhard Gleiss. Saturation visualization for vampire, Jun 2020.

[19] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*, January 2011.

[20] Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *PLDI '14, June 09 - 11 2014, Edinburgh, United Kingdom*, June 2014.

[21] Sumit Gulwani. Programming by examples (and its applications in data wrangling). In *Lectures at Marktoberdorf Summer School, Aug 2015*, August 2015.

[22] Sumit Gulwani. Program synthesis using stochastic techniques. *Communications of the ACM, Technical Perspective*, 59(2):113, February 2016.

[23] Sumit Gulwani. Programming by examples (and its applications in data wrangling). In *Verification and Synthesis of Correct and Secure Systems*. IOS Press, January 2016.

[24] Sumit Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Invited talk at IJCAR 2016*, June 2016.

[25] Sumit Gulwani. Spreadsheet programming using examples. In *Keynote at SEMS 2016*, January 2016.

[26] Sumit Gulwani. Research for practice: Programming by examples. *Research for Practice, CACM*, 60:46–49, July 2017.

[27] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. volume 55, pages 97–105, January 2012. Invited to CACM Research Highlights.

[28] Sumit Gulwani and Prateek Jain. Programming by examples: Pl meets ml. In *APLAS 2017*. Springer, October 2017.

[29] Sumit Gulwani and Prateek Jain. *Programming by Examples: PL meets ML*. IOS Press, February 2019.

[30] Sumit Gulwani, Vu Le, Arjun Radhakrishna, Ivan Radicek, and Mohammad Raza. Structure interpretation of text formats. In *OOPSLA*. ACM, October 2020.

[31] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. In *PLDI*. ACM, May 2018.

[32] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Bjoern Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Learning at Scale (L@S)*, pages 89–98. ACM, ACM, August 2017.

[33] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. Semi-supervised verified feedback generation. March 2016.

[34] Ashwin Kalyan, Abhishek Mohta, Alex Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *6th International Conference on Learning Representations (ICLR)*, January 2018.

[35] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. Automatic abstraction in smt-based unbounded software model checking. *CoRR*, abs/1306.1945, 2013.

[36] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974.

[37] Nham Le. Spacer visualization, Mar 2021.

[38] Vu Le, Daniel Perelman, Alex Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. Interactive program synthesis. *arXiv preprint*, arXiv:1703.03539, March 2017.

[39] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Alex Polozov, Rishabh Singh, Ben Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In *ACM Symposium on User Interface Software and Technology (UIST) 2015*, November 2015.

[40] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. On the fly synthesis of edit suggestions. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. ACM, October 2019.

[41] Nagarajan Natarajan, Danny Simmons, Naren Datha, Prateek Jain, and Sumit Gulwani. Learning natural programs from a few examples in real-time. In *AIStats*, January 2019.

[42] Saswat Padhi, Prateek Jain, Daniel Perelman, Alex Polozov, Sumit Gulwani, and Todd Milstein. Flashprofile: Interactive synthesis of syntactic profiles. *arXiv preprint*, arXiv:1709.05725, September 2017.

[43] Saswat Padhi, Daniel Perelman, Prateek Jain, Alex Polozov, Sumit Gulwani, and Todd Millstein. Flashprofile: A framework for synthesizing data profiles. In *ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH - OOPSLA)*. ACM, November 2018.

[44] Rangeet Pan, Vu Le, Nachi Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. Can program synthesis be used to learn merge conflict resolutions? an empirical analysis. In *2021 IEEE/ACM 43nd International Conference on Software Engineering (ICSE '21)*. IEEE, May 2021.

[45] Alex Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH) 2015*, October 2015.

[46] Mohammad Raza and Sumit Gulwani. Automated data extraction using predictive program synthesis. In *AAAI 2017*. Association for the Advancement of Artificial Intelligence, February 2017.

[47] Mohammad Raza and Sumit Gulwani. Disjunctive program synthesis: A robust approach to programming by example. In *AAAI 2018*. Association for the Advancement of Artificial Intelligence, February 2018.

[48] Mohammad Raza and Sumit Gulwani. Web data extraction using hybrid program synthesis: a combination of top-down and bottom-up inference. In *SIGMOD (Special Interest Group on Management of Data)*. ACM, June 2020.

[49] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *ICSE 2017*, April 2017.

[50] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. Here we go again: Why is it difficult for developers to learn another programming language? In *42nd International Conference on Software Engineering (ICSE)*, July 2020.

[51] Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. In *43rd Symposium on Principles of Programming Languages (POPL 2016)*. ACM - Association for Computing Machinery, January 2016.

[52] Carsten Sinz. Visualizing sat instances and runs of the dpll algorithm. *J. Autom. Reasoning*, 39:219–243, 08 2007.

[53] Matt Soos. Crystalball: Sat solving, data gathering, and machine learning, Jun 2019.

[54] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Björn Hartmann. Tracediff: Debugging unexpected code behavior using trace divergences. In *VL/HCC*, pages 107–115, August 2017.

[55] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. Fidex: Filtering spreadsheet data using examples. In *OOPSLA'16 October 25-30, 2016, The Netherlands*, October 2016.

[56] Mengshi Zhang, Daniel Perelman, Vu Le, and Sumit Gulwani. An integrated approach of deep learning and symbolic analysis for digital pdf table extraction. In *ICPR 2020*, January 2021.

APPENDICES

Appendix A

Models Used in the Grammar

A.1 Node

```
public class Node
{
    private static int _idCounter;
    public Node(Z3_decl_kind type, List<Node> children, Expr expr,
    Context ctx)
    {
        _id = _idCounter;
        _idCounter++;
        Expr = expr;
        Children = children;
        Type = type;
        Ctx = ctx;
    }

    public List<Node> Children { get; set; }

    private Node Parent { get; set; }
    private int _id;
    public Expr Expr { get; }
    public Context Ctx { get; }
    public Z3_decl_kind Type { get; }

    public void AddChild(Node child, int position = -1)
```

```
{
    if (child.Parent == null)
    {
        child.Parent = this;
    }
    if (position != -1)
    {
        Children.Insert(position, child);
    }
    else
    {
        Children.Add(child);
    }
}

public override string ToString()
{
    return Expr.ToString();
}

public bool IsEqualTo(Node tree)
{
    if (tree is null)
    {
        return false;
    }
    if (Children?.Count == 0 && tree.Children.Count == 0)
    {
        return Expr.ToString() == tree.Expr.ToString();
    }
    if (Children?.Count != tree.Children.Count)
    {
        return false;
    }
    var result = true;
    for (var i = 0; i < Children.Count; i++)
    {
        result = result && Children[i]
                 .Equals(tree.Children[i]);
    }
    return Type == tree.Type && result;
}

public override bool Equals(object obj)
{
    return IsEqualTo(obj as Node);
}

public override int GetHashCode()
{
    return HashCode.Combine((int) Type);
}

public IEnumerable<string> GetIdentifiers()
{
    var result = new List<string>();
    if (Children.Count == 0)
    {
        if (Expr.FuncDecl.DeclKind ==
            Z3_decl_kind.Z3_OP_UNINTERPRETED)
        {
            result.Add(Expr.ToString());
        }
    }
    foreach (var child in Children)
    {
        result = result.Concat(child.GetIdentifiers())
                       .ToList();
    }
    return result;
}

public bool HasIdentifier(string id)
{
    var result = false;
    if (Children.Count != 0)
    {
        foreach (var child in Children)
        {
            result = result || child.HasIdentifier(id);
        }
    }
    return result || Expr.ToString() == id;
}

public bool IsNot()
{
    return Expr.FuncDecl.DeclKind == Z3_decl_kind.Z3_OP_NOT;
}

public IList<Expr> FlattenTree()
{
    var exprs = new List<Expr>();
    if (Expr.NumArgs == 0
    || Expr.FuncDecl.DeclKind == Z3_decl_kind.Z3_OP_SELECT)
    {
        exprs.Add(Expr);
    }
    foreach (var child in Children)
    {
        exprs = exprs.Concat(child.FlattenTree()).ToList();
    }

    return exprs;
}


public IEnumerable<string> GetProcesses()
{
    var children = new List<string>();

    foreach (var child in Children)
    {
        var exprs = child.FlattenTree();
        var indices = Utils.FindSelect(exprs.ToList());

        foreach (int index in indices)
        {
            var selectExpr = exprs[index];
            var process = selectExpr.Args[1].ToString();

            if (!children.Contains(process))
            {
                children.Add(process);
            }
        }
    }

    return children;
}
}
```

Appendix B

SMT Files for Simple Bakery

B.1 Simple Bakery Three Processes

```
(set-logic HORN)
(declare-fun Inv ((Array Int Int) (Array Int Int) Int Int) Bool)
(assert (forall ((State (Array Int Int))
         (Num (Array Int Int))
         (Issue Int)
         (Serve Int)
         (|State'| (Array Int Int))
         (|Num'| (Array Int Int))
         (|Issue'| Int)
         (|Serve'| Int)
         (i Int)
         (j Int))
  (=> (and (= Issue 0)
           (= Serve 0)
           (= (select State 1) 0)
           (= (select State 2) 0)
           (= (select State 3) 0)
           (= (select Num 1) (- 1))
           (= (select Num 2) (- 1))
           (= (select Num 3) (- 1)))
      (Inv State Num Issue Serve))))
```

```
(assert (forall ((State (Array Int Int))
         (Num (Array Int Int))
         (Issue Int)
         (Serve Int)
         (|State'| (Array Int Int))
         (|Num'| (Array Int Int))
         (|Issue'| Int)
         (|Serve'| Int)
         (i Int)
         (j Int))
  (let ((a!1 (or (and (= (select State i) 0)
                      (<= i 3)
                      (>= i 1)
                      (= |Serve'| Serve)
                      (= |Issue'| (+ Issue 1))
                      (= |State'| (store State i 1))
                      (= |Num'| (store Num i Issue)))
                 (and (= (select State i) 1)
                      (<= i 3)
                      (>= i 1)
                      (= (select Num i) Serve)
                      (= |Serve'| Serve)
                      (= |Issue'| Issue)
                      (= |Num'| Num)
                      (= |State'| (store State i 2)))
                 (and (= (select State i) 2)
                      (<= i 3)
                      (>= i 1)
                      (= |Issue'| Issue)
                      (= |Serve'| (+ Serve 1))
                      (= |Num'| (store Num i (- 1)))
                      (= |State'| (store State i 0))))))
    (=> (and (Inv State Num Issue Serve) a!1)
        (Inv |State'| |Num'| |Issue'| |Serve'|)))))
(assert (forall ((State (Array Int Int))
         (Num (Array Int Int))
         (Issue Int)
         (Serve Int)
         (|State'| (Array Int Int))
         (|Num'| (Array Int Int))
         (|Issue'| Int)
         (|Serve'| Int)
         (i Int)
         (j Int))
  (=> (and (Inv State Num Issue Serve)
           (= (select State i) 2)
           (= (select State j) 2)
           (distinct i j)
           (<= i 3)
           (>= i 1)
           (<= j 3)
           (>= j 1))
      false)))

(check-sat)
```