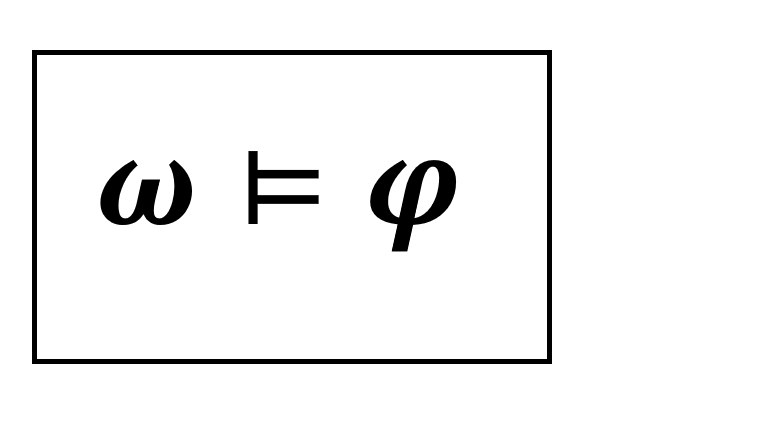
****

Software Engineering Department

Ort Braude College

Course 61771: Extended Project in Software Engineering

**Fast LTL Satisfiability Checking by SAT Solvers**

****

In Partial Fulfillment of the Requirements for

Final Project in Software Engineering (Course 61771)

Karmiel – June 2020

Tzach Sheratzky

**Supervisor**:

Dr. Sarai Sheinvald

Contents

1. Introduction 1

2. Theory 1

2.1. Background and related work 1

2.2. Linear Temporal Logic (LTL) 2

2.3. SAT Problem 3

2.4. Conventions 4

2.5 Theorems 5

2.6 Running Example 6

2.7 Microsoft Z3 Theorem Prover 7

2.8 Spot 7

2.9 Software Architecture 7

3. Software Engineering Documents 8

3.1 Use Case 8

3.2 GUI 10

3.3 Design 17

3.4 Testing 17

4. Results and conclusions 20

4.1 Results 20

4.2 Conclusions 30

References 30

1. **INTRODUCTION**

The question of the satisfiability of a linear temporal logic (LTL) plays an important role in validating linear temporal specifications, to detect such errors in a system’s pre-design stage [1], [2]. Therefore, an efficient satisfiability check of large LTL formulas is a necessary tool. The purpose of those formulas is to specify system behavior, e.g. a simple LTL formula can define that during the run, two threads cannot enter the same critical section at the same time, is written as , where the is a Globally operator meaning that the formula inside his scope holds all of the time. Another example is that when a process requests a resource, the operating system allocates it during the run, which could be written as: , where is a Future operator meaning that the formula inside his scope will hold sometime in the future. Those two examples demonstrate satisfiable formulas, but on the other hand, there are formulas that are unsatisfiable as the following one: , which means that globally holds and in the future it will not hold anymore, which is a conflict.

The main goal of this project is to implement an LTL solver, a program that checks the satisfiability of an LTL formula [3]. We will do so by implementing a new approach to this problem that is based on [4], which reduces the satisfiability problem for LTL to the satisfiability problem of a set of SAT [5] formulas. Our program is implemented by integrating existing SAT solver tools [6]. Then, experiments were tested to analyze the efficiency of this method.

The procedure that we present in our project is implemented by two-phase checks. The first one, which uses an SAT solver, is implemented by finding the Obligation Sets of an LTL formula. Each Obligation Set is a group of atoms that is created by grouping the significant atoms of an LTL formula that determine whether the original formula is satisfiable or not. This phase assumes that the SAT solver receives every atom as a clause, and then if the logical conjunction of all the clauses evaluates to , we could say that we have found a consistent Obligation Set [4], and so the formula is satisfiable. In case where this check does not succeed, which means that we have not found a consistent Obligation Set, then we must normalize the LTL formula by using Normal Form [4] , which is a set of formulas that describe how the original formula behaves within a continuous time frame. With that information, we search for the existence of a time frame, from which, the formula is infinitely satisfiable by some of their related atoms. By checking if the atoms of an Obligation Formula [4], which extends the Obligation Set, are a subset of ’s atoms that we already found, it conditionally provides the original formula’s satisfiability result.

Eventually, by following that approach, the expected result is an accelerated algorithm for the complete decision procedure of LTL formulas satisfiability solving.

1. **THEORY**
   1. **Background and related work**

In past years, there were many approaches to deal with the LTL problem, in order to check the consistency of such specifications. One of the approaches that have been researched and developed was the model-checking approach, that reduces LTL satisfiability to LTL model checking by model checking [6] the negation of given formula against a universal model [1]. Another approach, that our implementation is based on [7], is an automata-based approach that reduces satisfiability checking to checking an automaton’s transition system by some techniques, and especially the obligations set method which will be explored in detail.

* 1. **Linear Temporal Logic (LTL)**

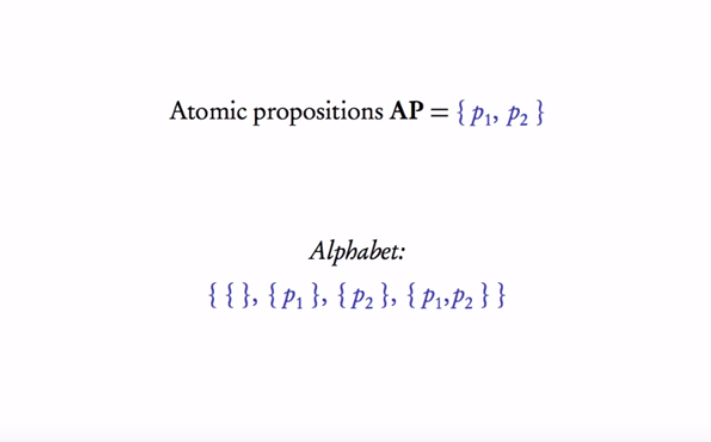
We focus our project on linear temporal logic (LTL) which is a model that allows us to check and examine the behavior of a formula over time. This formula can express the specification and properties of a system by using temporal operators. We define property as a set of infinite words over the power set of atomic proposition’s set. Let AP be a set of atomic propositions. Let be an LTL formula which is defined by the following syntax:

, where .

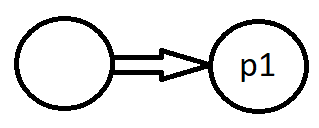
We define the following as abbreviations: (sometime is true) and (everywhere is true). The temporal operators are a complete set of temporal logic connectives. is defined as a literal if it is an atomic proposition or its negation. If does not contain any temporal logic operators it defined as propositional formula. Let be a specified property, which can be written as an infinite trace of letters . The following sequence when we use to denote the prefix of until its letter, and to denote the suffix, the semantics as follow by given a trace and properties :

; ;

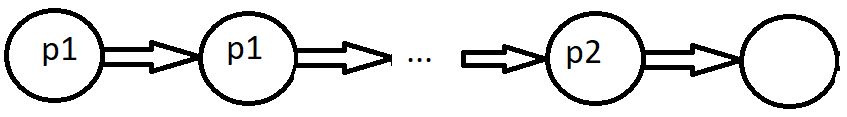
We define that is **satisfiable** if there exists an infinite trace such that . The satisfiability problem and the model checking problem are both PSPACE-complete for linear temporal logic. The following is a visual example for the definitions:



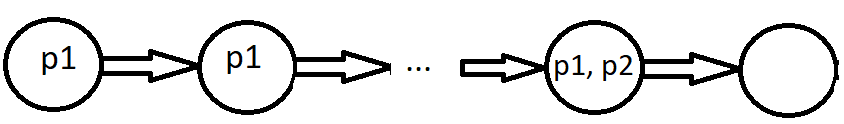
*Fig.1 Atomic propositions and their power set*



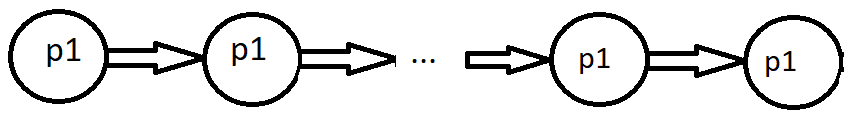
*Fig.2 () illustration*



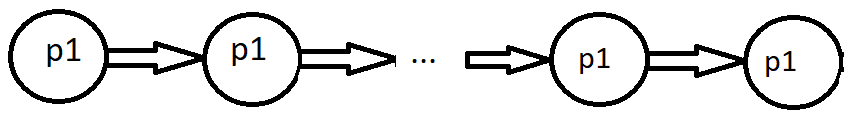
*Fig.3 () illustration*



*Fig.4 (), when holds illustration*



*Fig.5 (), when does not holds illustration*



*Fig.6 () illustration*

* 1. **SAT Problem**

Another important problem in the computer science field is to determine whether a Boolean formula is satisfiable. An instance of the satisfiability (SAT) problem is a Boolean formula with three components; a set of N variables , a set of literals such that a literal is a variable () or its negation (, and a set of distinct clauses: . Each clause consists of only literals combined by just logical-or ( connectives. The goal of the satisfiability problem is to determine whether there exists an assignment of truth values to variables that make the following Conjunctive Normal Form (CNF) formula satisfiable: . Where is a logical-and connective. By the above definition, finding an interpretation of variables that satisfies a CNF is a difficult problem. Moreover, the SAT problem is a core of a large family of computationally intractable NP-complete problems. In our project, a SAT-solver is integrated to accelerate the approach for LTL solving. For example: let . SAT problem for this Boolean expression described as the following: is there an assignment of or values for the variables , such that the given Boolean expression holds? In this case, the answer for the SAT problem is **yes** because we could choose an assignment such that which satisfies the formula and evaluates the expression to .

* 1. **Conventions**

Every LTL formula in our implementation should be in a negation normal form (NNF), which means that negation operators are applied to literals only. is a propositional formula if it does not contain temporal operators In addition, we convert the Eventually and Globally operators to Until and Release operators by the following semantics: and , in order to use them in the related theory.

Terms and Definitions

For a formula , we define its **Obligation Set** by as:

,

is a literal, then

or , then

is a Consistent Obligation if and only if , where , and is the number of elements in .

The **Normal Form** of an LTL formula defined by as follows:

, if is a propositional formula

, if

Here such that the root operator of is not a disjunction, then is the set of disjuncts of .

The labeled **Transition System** [4] generated from the formula is a tuple where:

* is the initial state.
* Act is the set of CNFs over , which is the set of the transition system’s alphabet.
* The transition relation Act is defined by: iff there exists:

.

* is the smallest set of formulas such that , and implies . Intuitively it is the set of states in the transition system.

Given an LTL formula , the corresponding **Obligation Formula** is recursively defined as:

and

where is a literal, then

, then

or then

, then

, then

The **Weak Satisfaction Relation** [4] is defined by:

Let be a set of literals of , and is a propositional formula in NNF.

* if is a literal, or then iff .
* iff and .
* iff or .
  1. **Theorems**

Our satisfiability checking method is based on the following theorems:

* **Obligation Acceleration (Theorem 1)** [4]: Assume is a consistent obligation, Then
* **SAT-Based Generalized Satisfiability Checking (Theorem 2)** [4]: The LTL formula is satisfiable if and only if there exists an SCC and a state in such that L(.
  1. **Running Example**

To explain how the method works, let us demonstrate by an example. Given the following LTL formula: , which represents the question: does holds sometimes while always holds, then, we ask whether is satisfiable or not. First use the LTL semantics to eliminate the Eventually and Globally operators, then we get in result . By theorem 1 we find that:

{

={ =

Because, then there is not a consistent obligation related to . So, by theorem 1 we cannot say that is satisfiable or unsatisfiable. In this case, we must use theorem 2 by constructing the transitions system . We start by calculating the normal form of :

{}

Which could be understood as a self-loop automaton that is constructed by a state with a self-transition labeled by :

Start

*Fig.7 Automaton illustration of LTL formula’s normal form*

By the normal form of , we found that there is an SCC such that L(. In order to check the statement of in theorem 2, we calculate , since is the state related to the achieved :

(

Then, by the weak satisfaction relation’s definition: L( and and , but so in this case the weak satisfaction relation definition does not holds, and therefore the condition of theorem 2 is not met, therefore is unsatisfiable. In this example the first theorem does not provides a solution, As a result, in our implementation we must implement an algorithm to calculate the Normal Form, create the transition system of and an algorithm to find SCC in , then, we implement the checks which are demonstrated above.

* 1. **Microsoft Z3 Theorem Prover**

To accelerate calculations, we integrate an external SAT solver [8] to our project. Once we find the obligation set of a formula , by using the accelerated tool, the satisfiability checking of theorem 1 achieves high performance. The Z3 API contains the functionality of creating clauses of literals and find whether there is an assignment of truth values to their variables. The following code’s snapshot demonstrates how to create three variables, attach them to three clauses as literals and run a satisfiability check on them:

// Create Z3 context

z3::context context;

// Create instance of z3 Solver

z3::solver solver { context };

// Create variables

z3::expr a = context.bool\_const("a");

z3::expr b = context.bool\_const("b");

z3::expr c = context.bool\_const("c");

// Create clauses

solver.add(!a && b && c);

solver.add(a && !b && c);

solver.add(a && b && !c);

// Check for solution

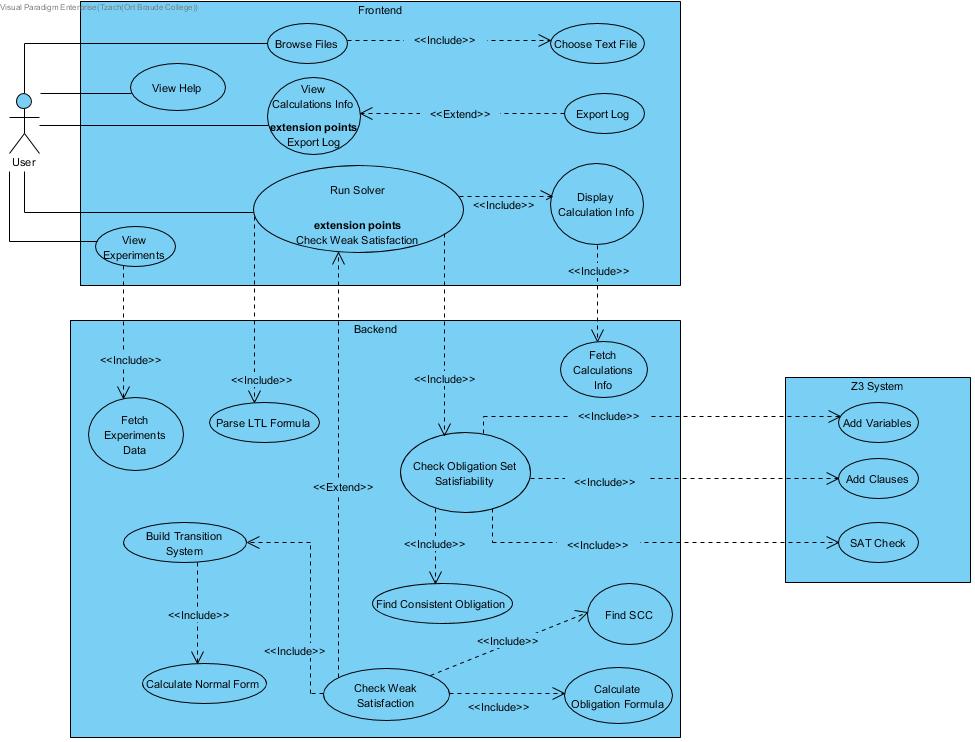
bool sat = (solver.check() == z3::sat);

* 1. **Spot**

After some research, we found that the optimal way for parsing and manipulating LTL formulas is by using an external library. The one we have found that meets our requirements is Spot [9], which is a high-performance C++ platform for LTL manipulation. We used this platform to parse, traverse, and manipulate formulas in our implementation.

* 1. **Software Architecture**

During the algorithm implementation, which we did in C++ over Unix, we thought about how to implement the GUI. The options were to use QT, or to write a cross-platform code so we could write the GUI in C# with the WPF framework and use the algorithm C++ code as a library. After some thinking, we decided to go for a web client-server architecture, so our project could be accessible from everywhere. As a result, we wrapped our C++ algorithm implementation by Crow [10] library as a restful API, and the client side implemented with React web frontend library, where the interaction between them is by web sockets, GET and POST http requests.

1. **SOFTWARE ENGINEERING DECUMENTS**
   1. **Requirements**

UC1: Browse Files

* Goal: Define input formulas for the algorithm to run on.
* Preconditions: file with the formulas exists.
* Possible user errors: select irrelevant file or unsupported format.
* Limitations: a “.txt.” file with formulas written with only letters from the English alphabet and logical operators.
* Pseudo code Flow:

|  |  |
| --- | --- |
| Actor | System |
| 1) Click ‘Browse’ button | 2) Display “Folder browse dialog form” |
| 3) Choose text file within the browse dialog | 4) Display formulas from the file |

UC2: Run Solver

* Goal: Start the LTL formulas processing
* Preconditions: UC1 completed successfully.
* Possible user errors: None.
* Limitations: None.
* Pseudo code Flow:

|  |  |
| --- | --- |
| Actor | System |
| 1) Click ‘Solve’ button | 2) Change “Solve” button to “Restart”  3) Run the algorithm checking process.  4) Show calculations log. |

UC3: Export Log

* Goal: Export the calculations log and save it locally.
* Preconditions: UC2 completed successfully.
* Possible user errors: None.
* Limitations: None.
* Pseudo code Flow:

|  |  |
| --- | --- |
| Actor | System |
| 1) Click ‘Export button | 2) Creates download of the log as a text file. |

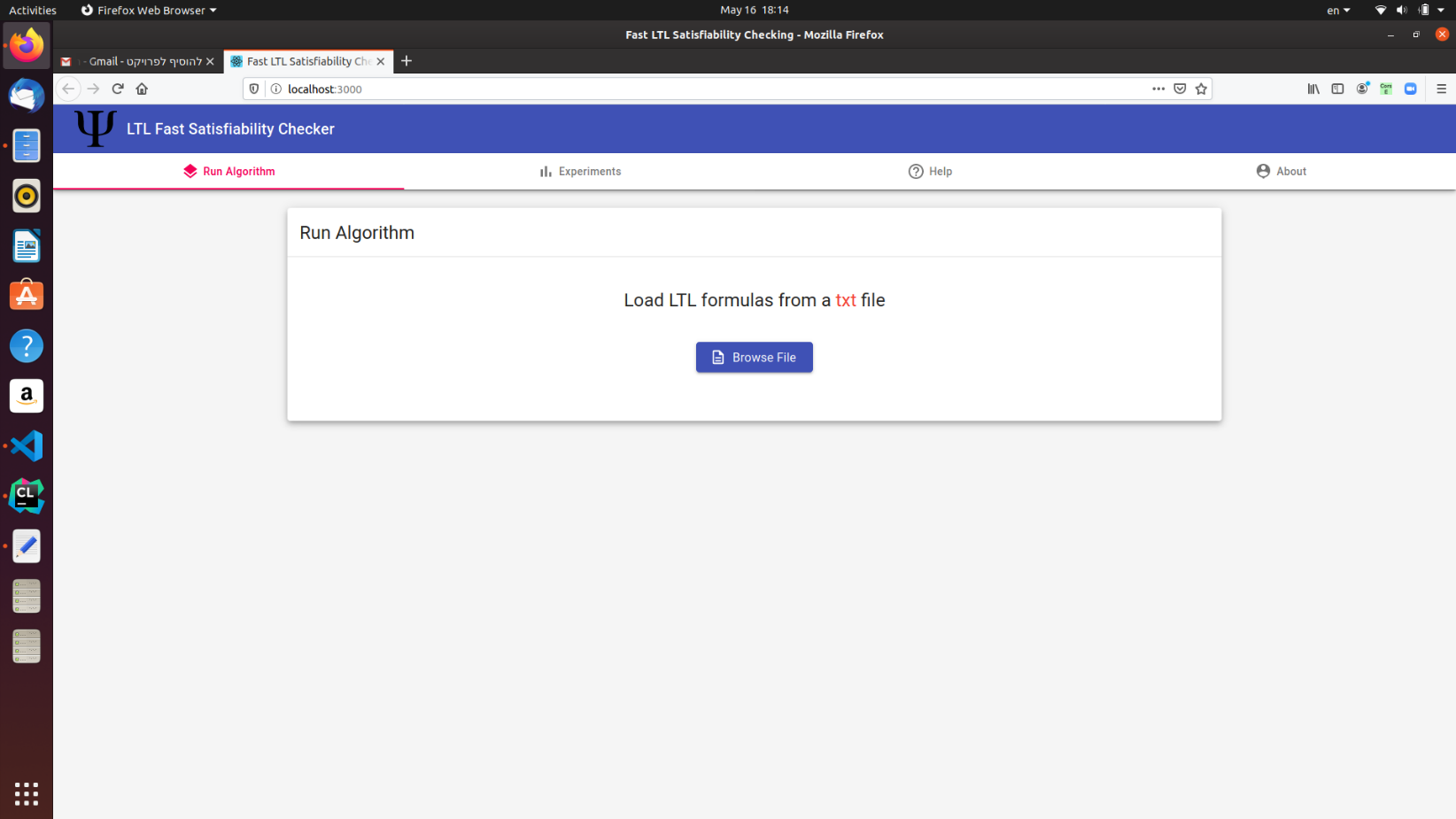
UC4: Display Results

* Goal: Result of formulas analysis presented after the algorithm check completes.
* Preconditions: UC2 completes successfully.
* Possible user errors: Displaying results before running the algorithm.
* Limitations: None.
* Pseudo code Flow:

|  |  |
| --- | --- |
| Actor | System |
| 1) Click “Experiments” button | 2) Fetch execution times and satisfiability information about the last run. |
|  | 3) Calculates and displays charts with the relevant information |

* 1. **GUI**

The GUI section will present the full use case of loading formulas, running the algorithm and view results and how to set up the system.



Main screen: This screen has a menu on top where the user can navigate between the options:

1. Run Algorithm – First view by default.
2. View Experiments.
3. Help section.
4. View information about the project.

Fig. 8: Main screen

Once the user clicks on the “Browse File”, an option to select text files from the local storage opens.

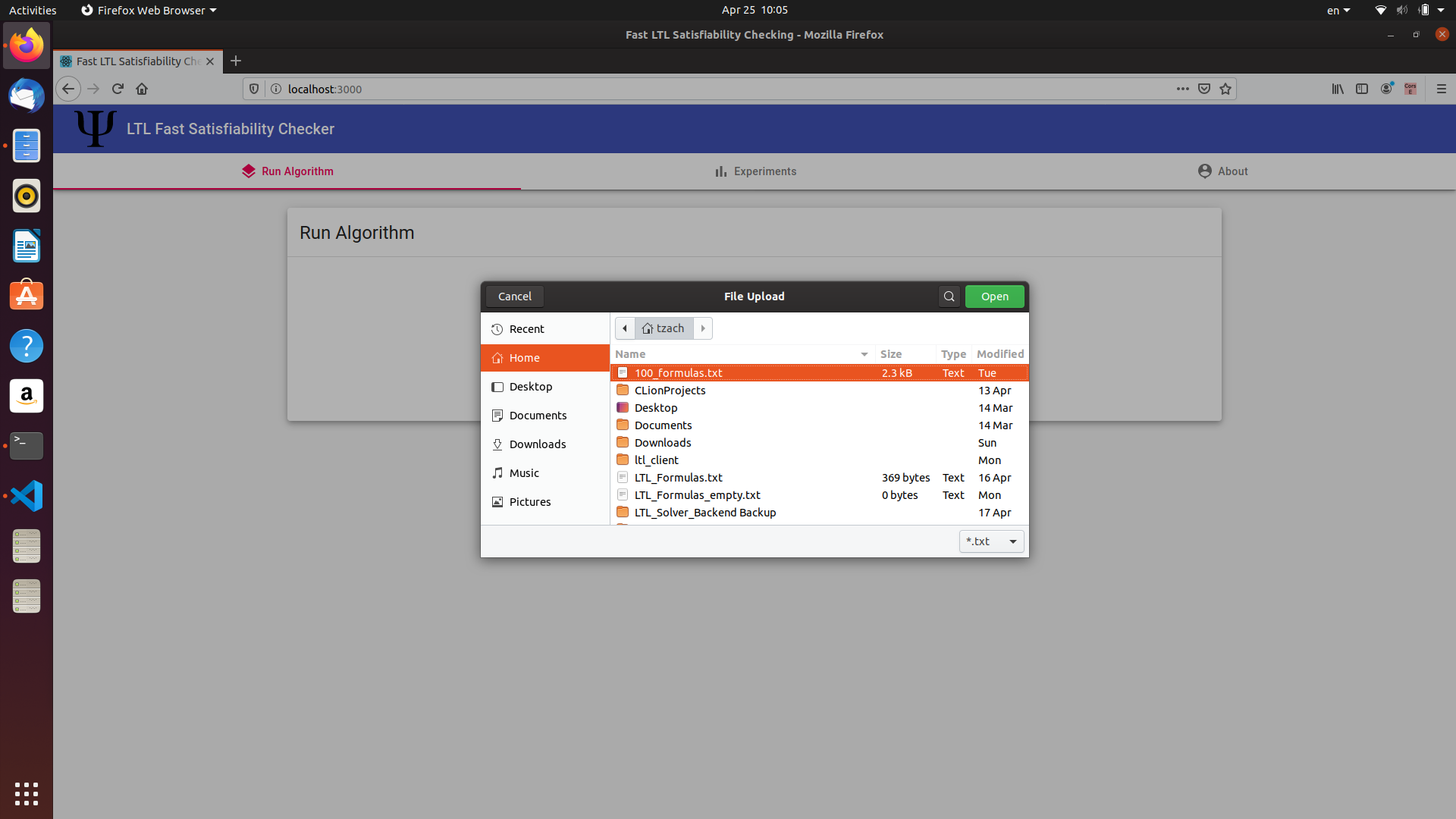


Fig. 9: File upload

After choosing a text file with LTL formulas, they will be displayed inside the “Loaded Formulas” View.

By Pressing “START” the algorithm will begin to run.

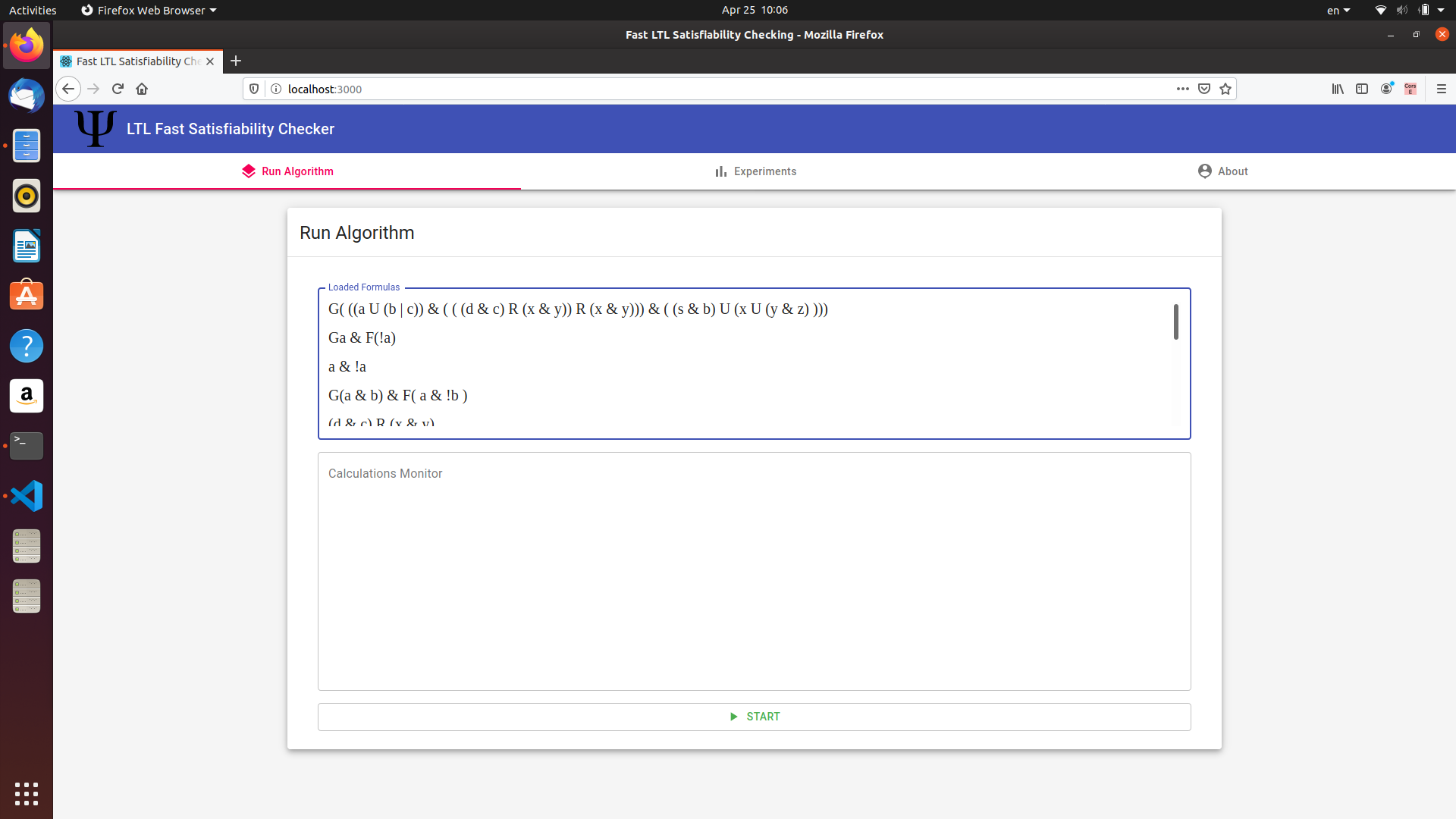
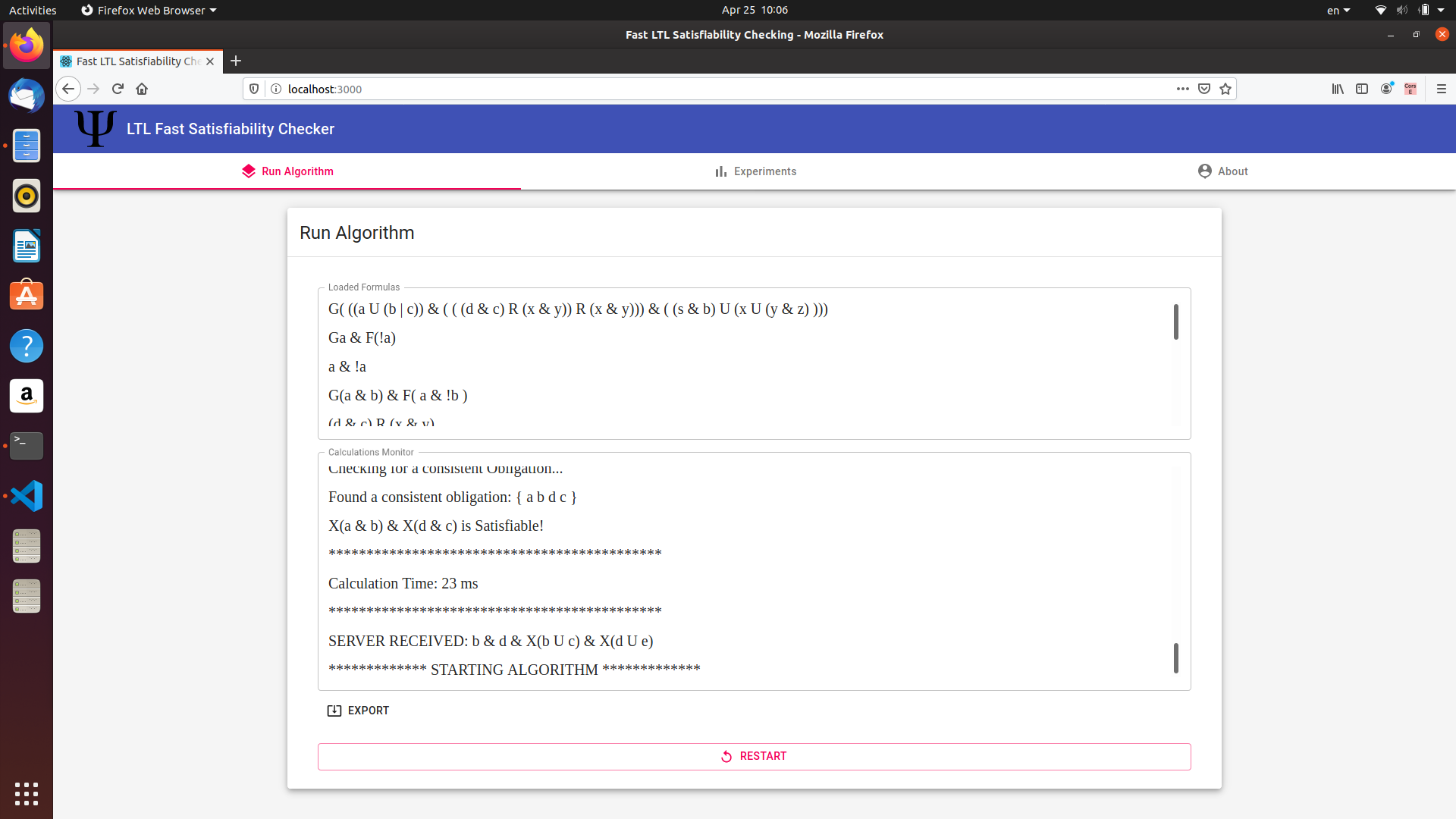


Fig. 10: Loaded Formulas

m



After click on “START”, the calculations monitor logs the algorithm progress, execution time per formula and whether the formula is satisfiable or unsatisfiable.

Note that to restart the process and begin with a new formulas file, the “RESTART” button can be clicked.

Fig. 11: Calculations Monitor

m

The backend works in the background and logs every formula that is received, and displays any http request from the client application. The server must be running to let the client application process formulas.

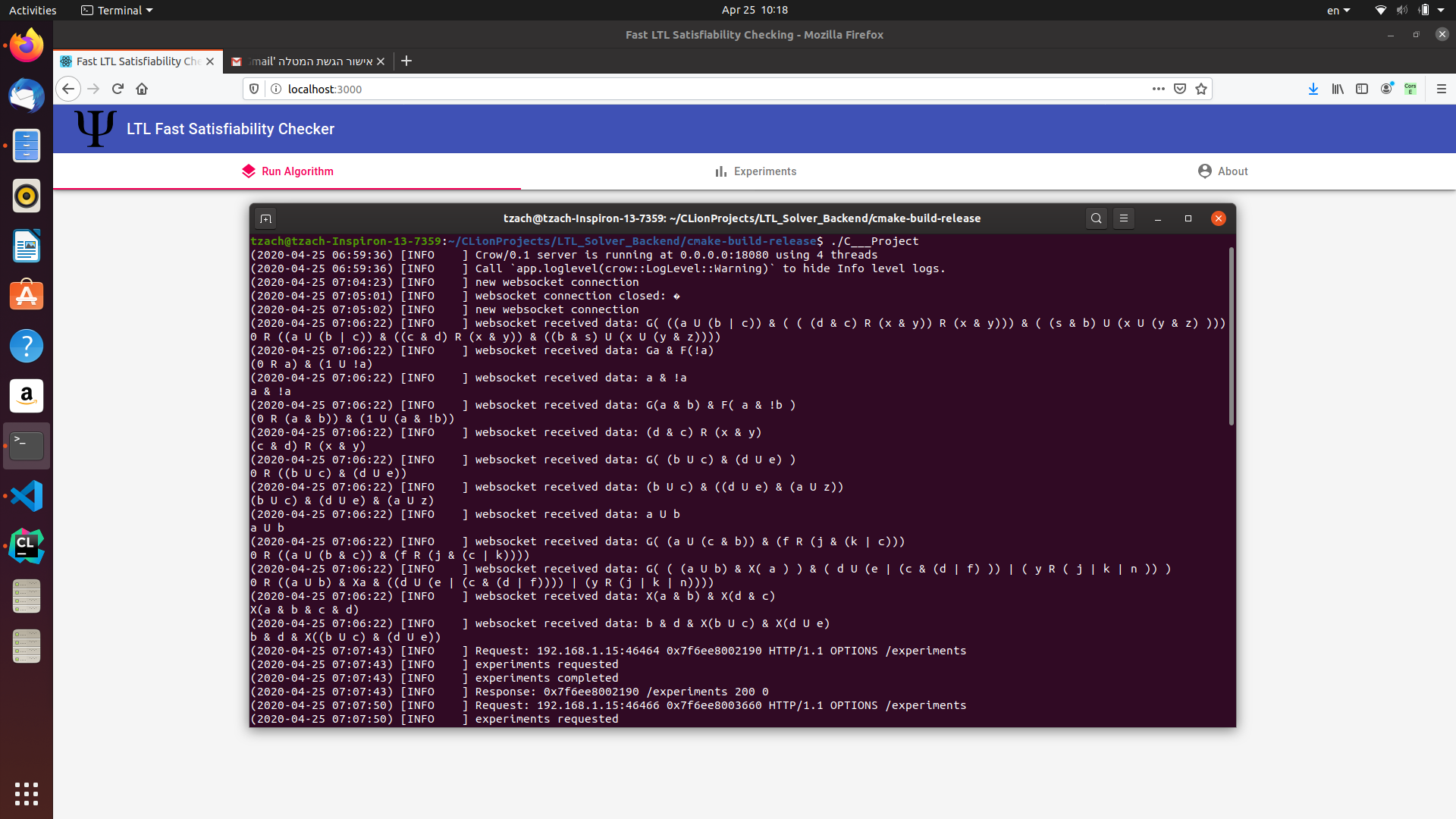
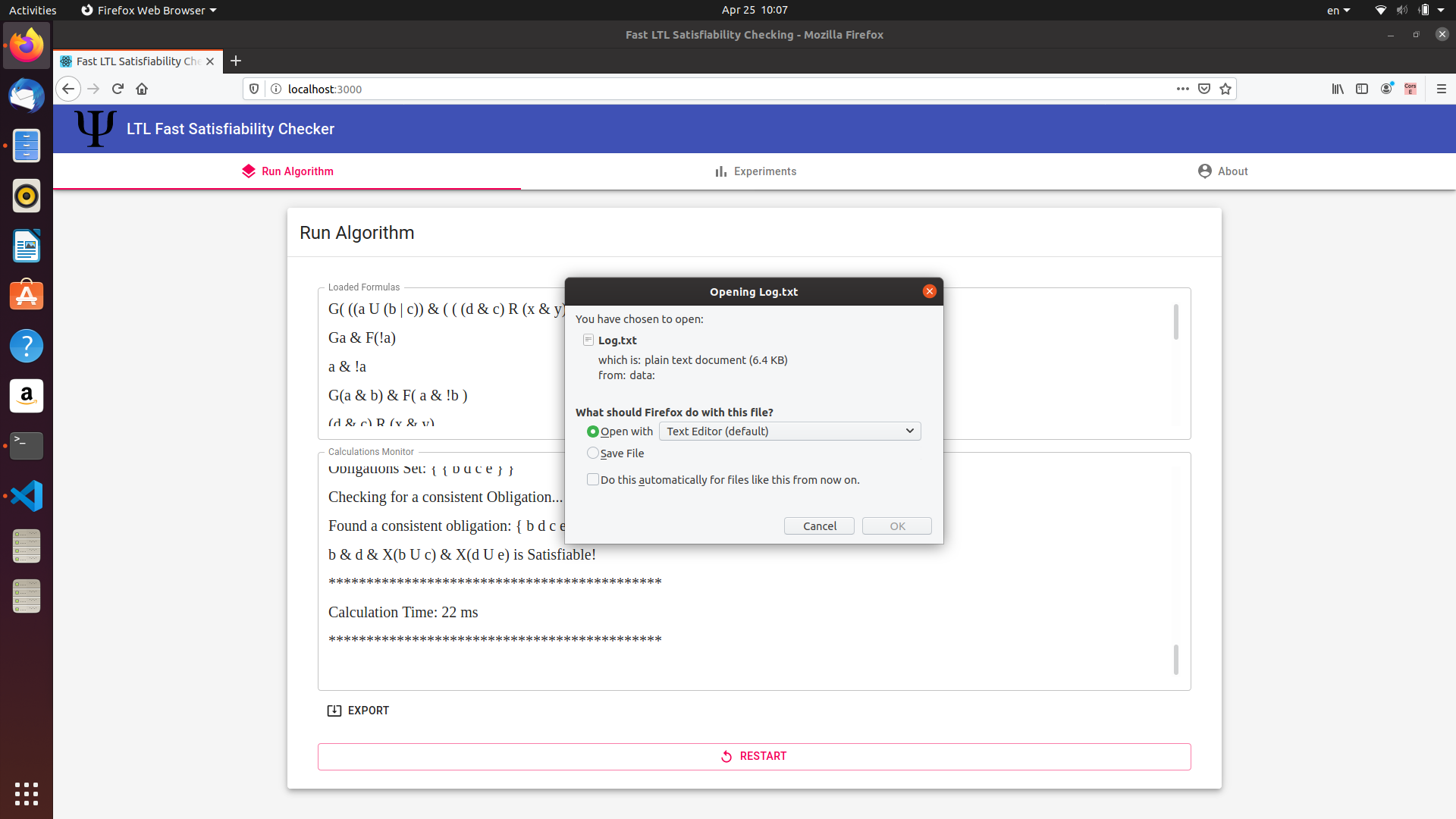


Fig. 12: Backend in runtime

m



**`**

Click on the “EXPORT” button to download the log

By clicking the “EXPORT” button on the bottom-left of the view, the monitor log content can be downloaded via a text file.

Fig. 13: Export calculations monitor

m

Then, if we open the downloaded log file, we can follow the algorithm output via the text file.

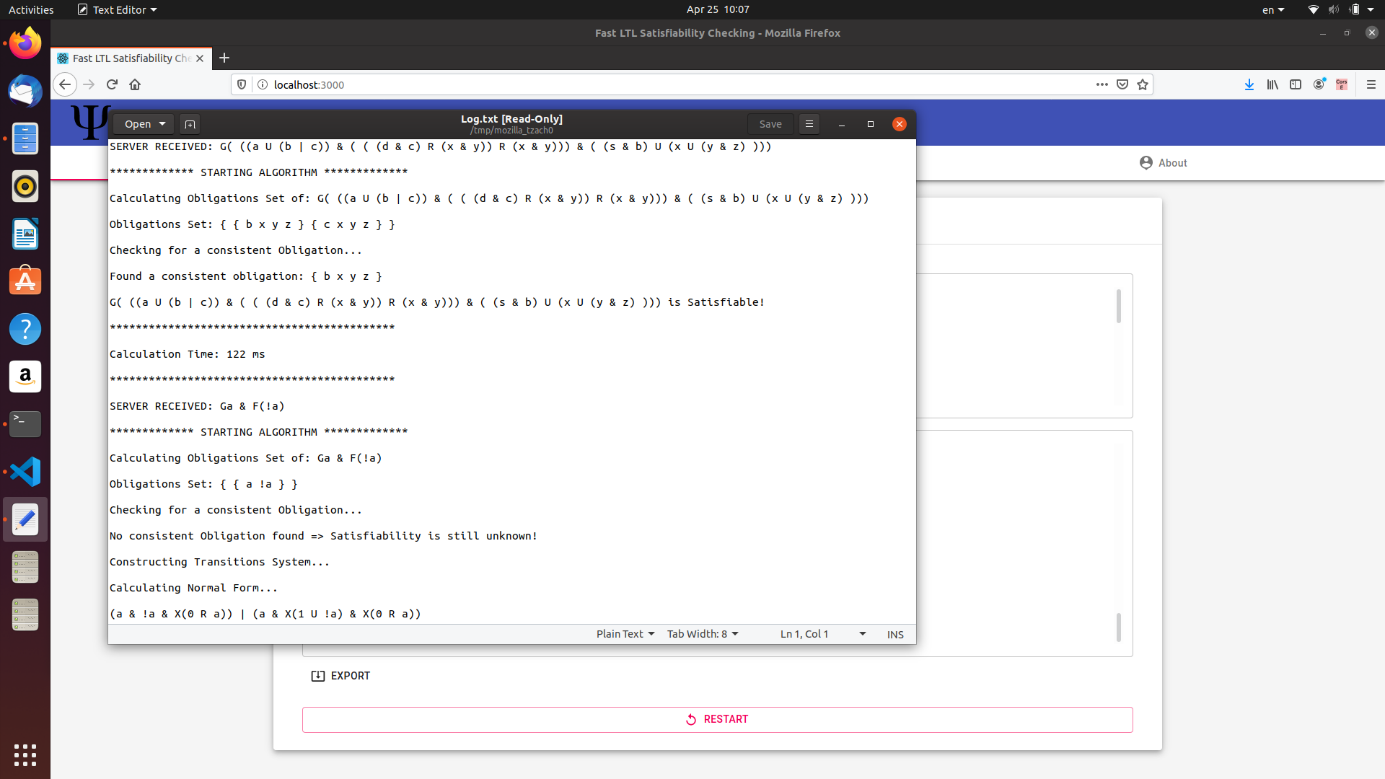


Fig. 14: Display exported file

m

By changing to the “Experiments” screen, by using the menu on top, we can explore the algorithm results.

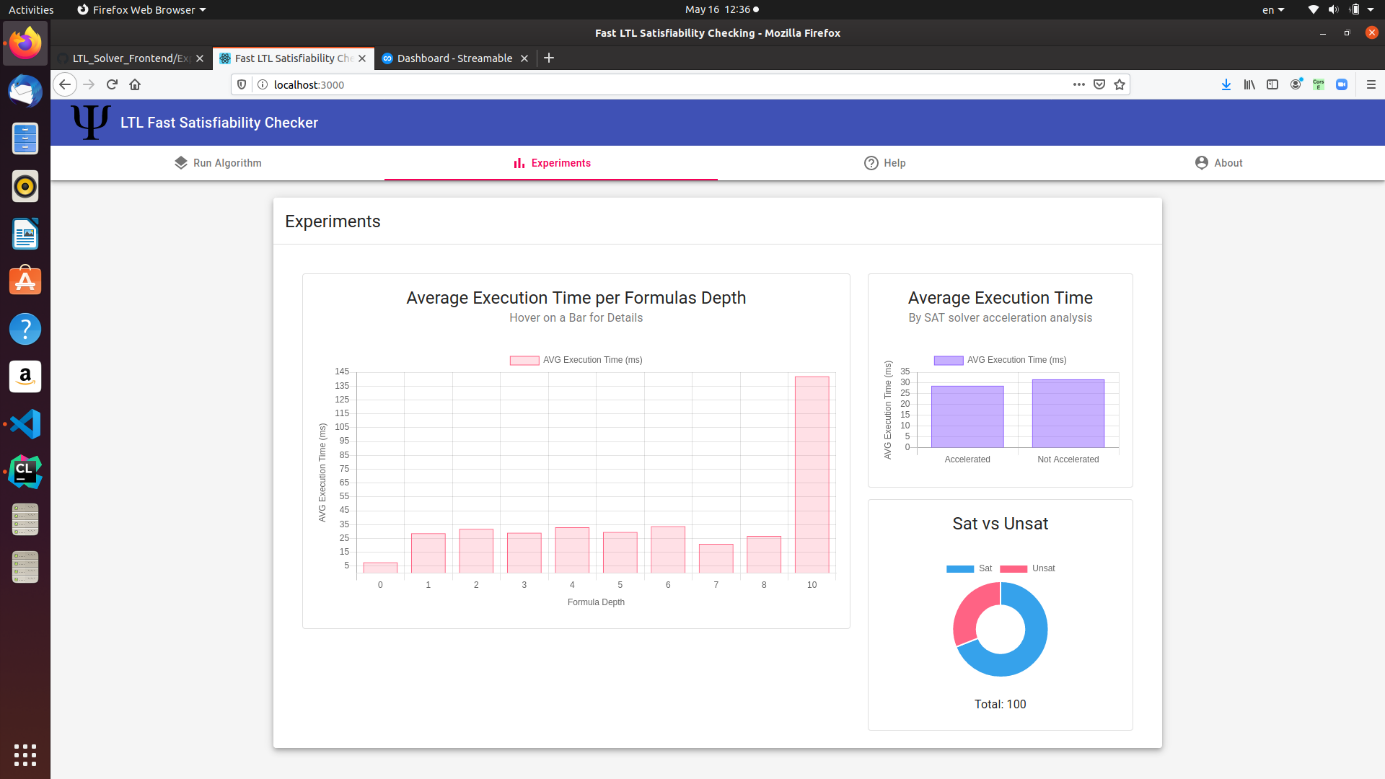


Fig. 15: Experiments screen

m

While moving with the mouse on the bars, we can see the average execution time (milliseconds) as a function of the formula’s depth

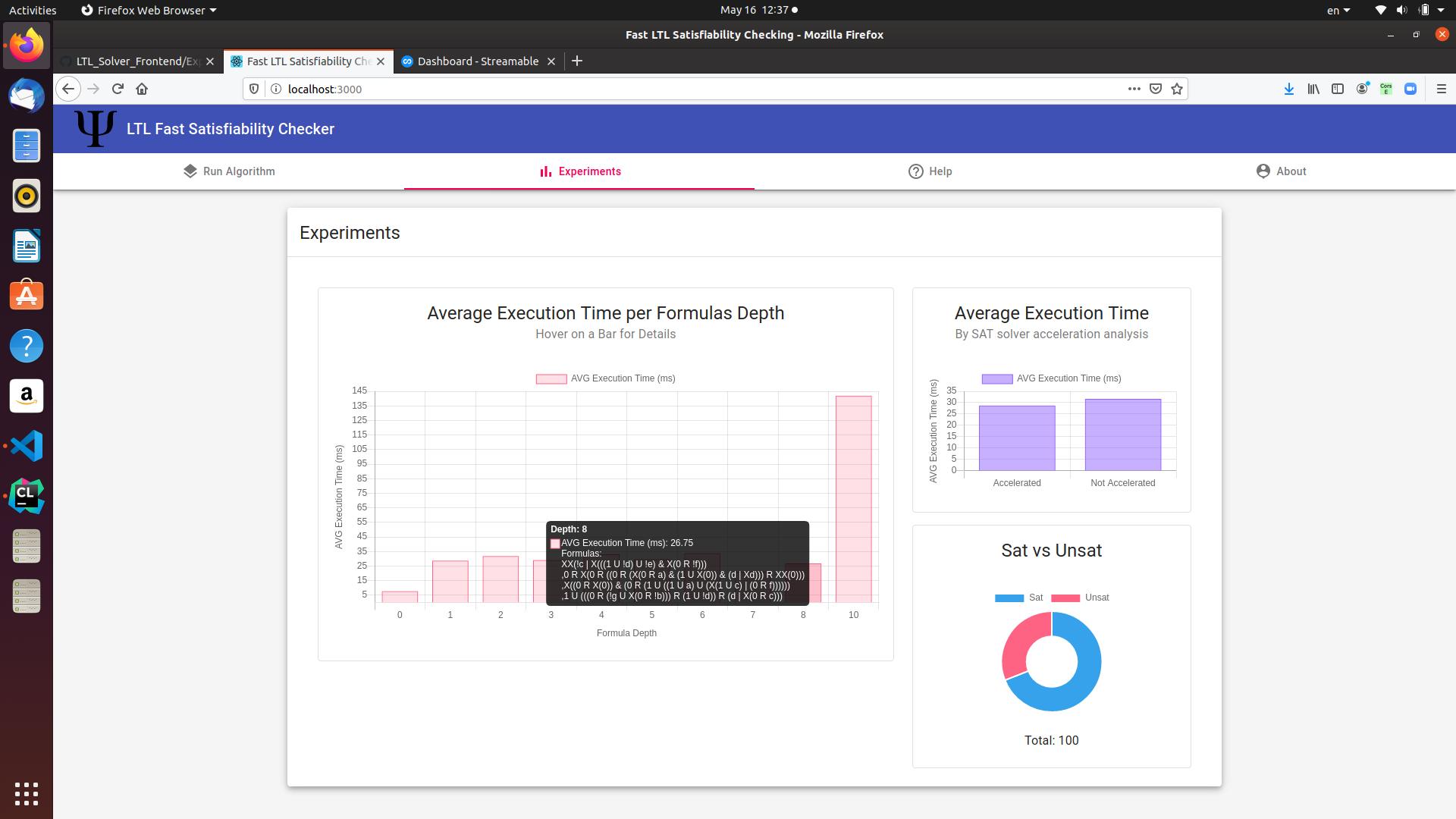


Fig. 16: Display average execution time result for each formulas depth group

m

To see the numbers in the pie chart, we must stand with the mouse over a part of the chart.

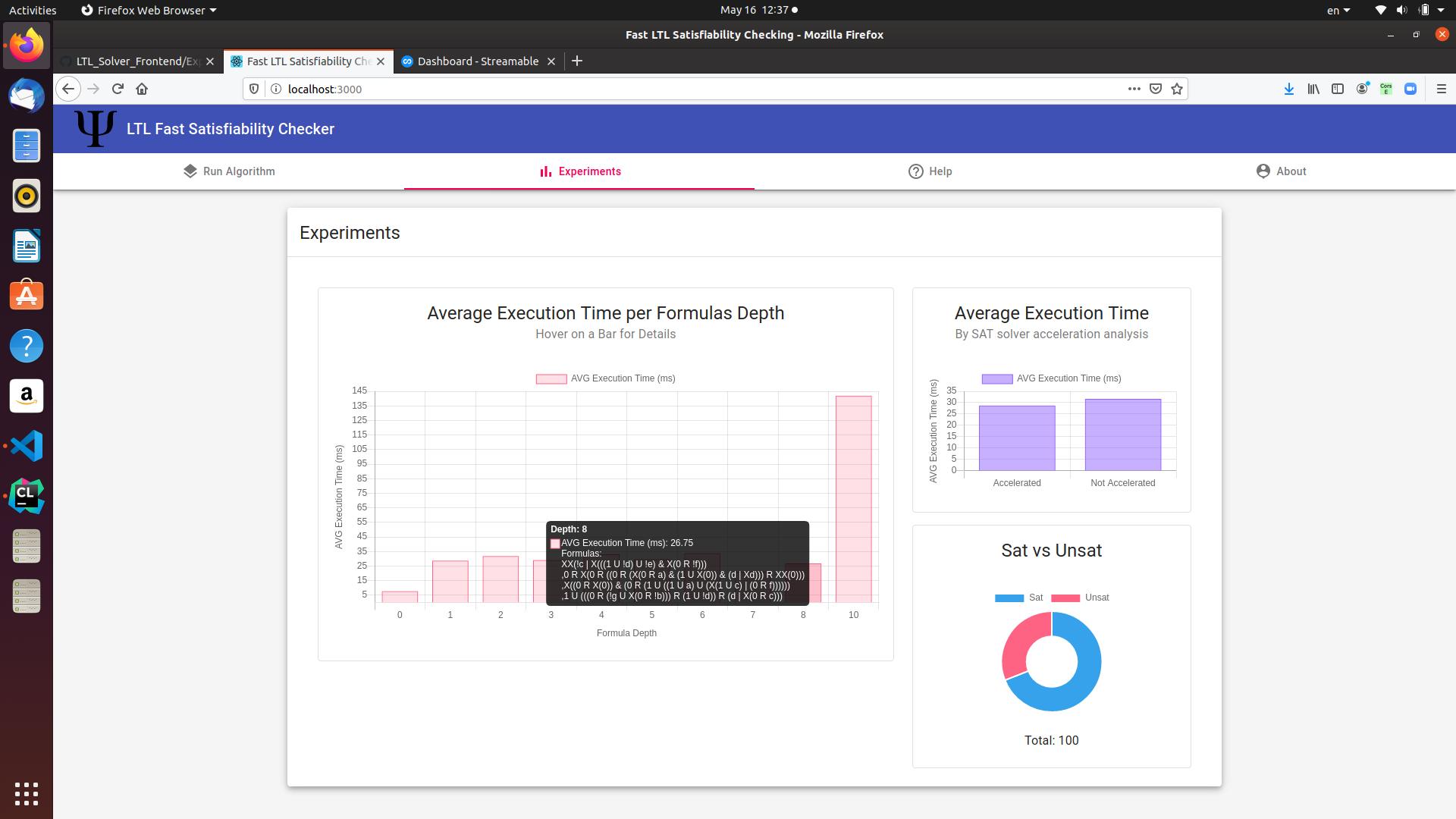
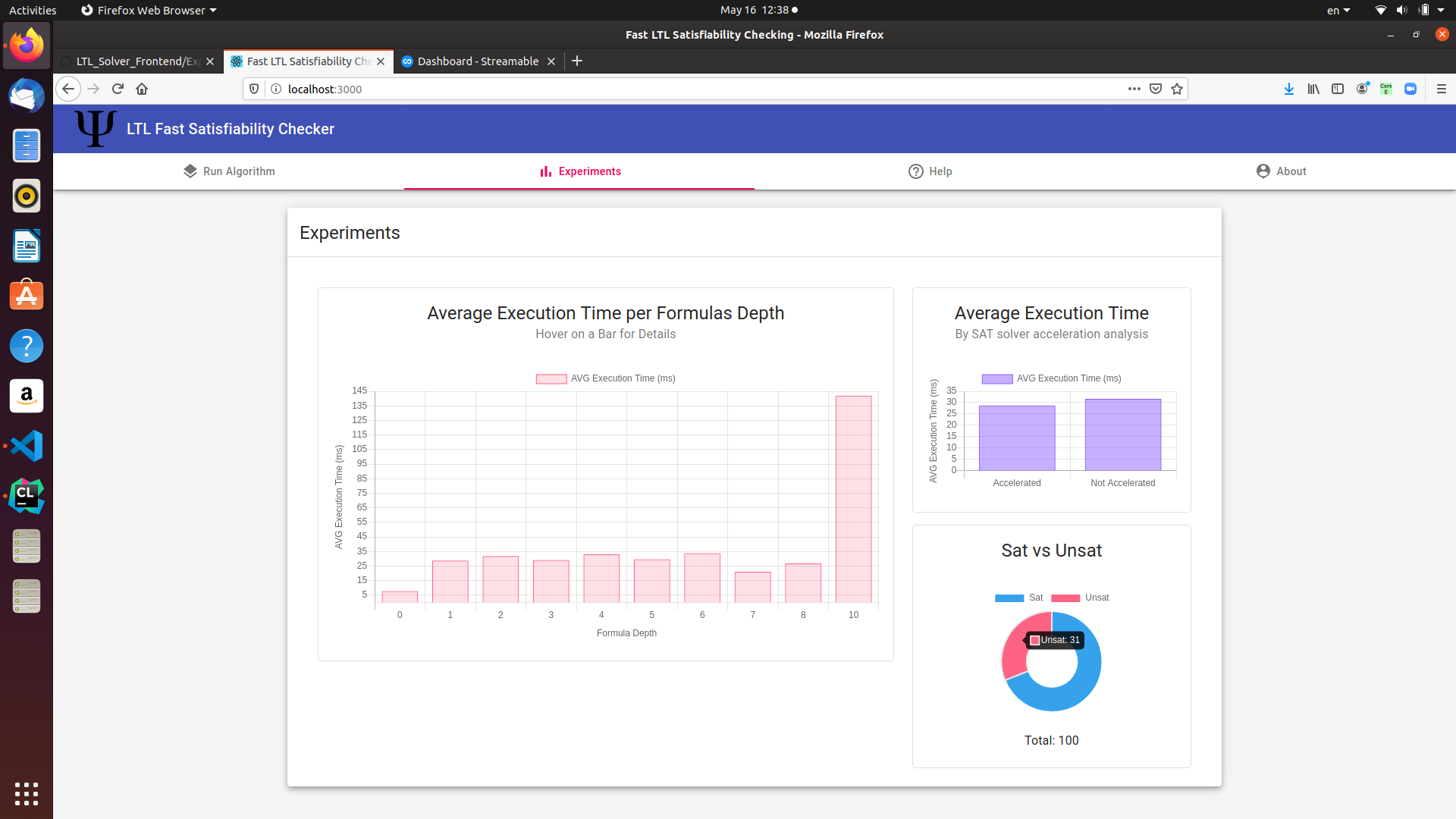


Fig. 17: Display Satisfiable vs Unsatisfiable formulas number

To see the numbers in the bar chart, we must stand with the mouse over a part of the chart.

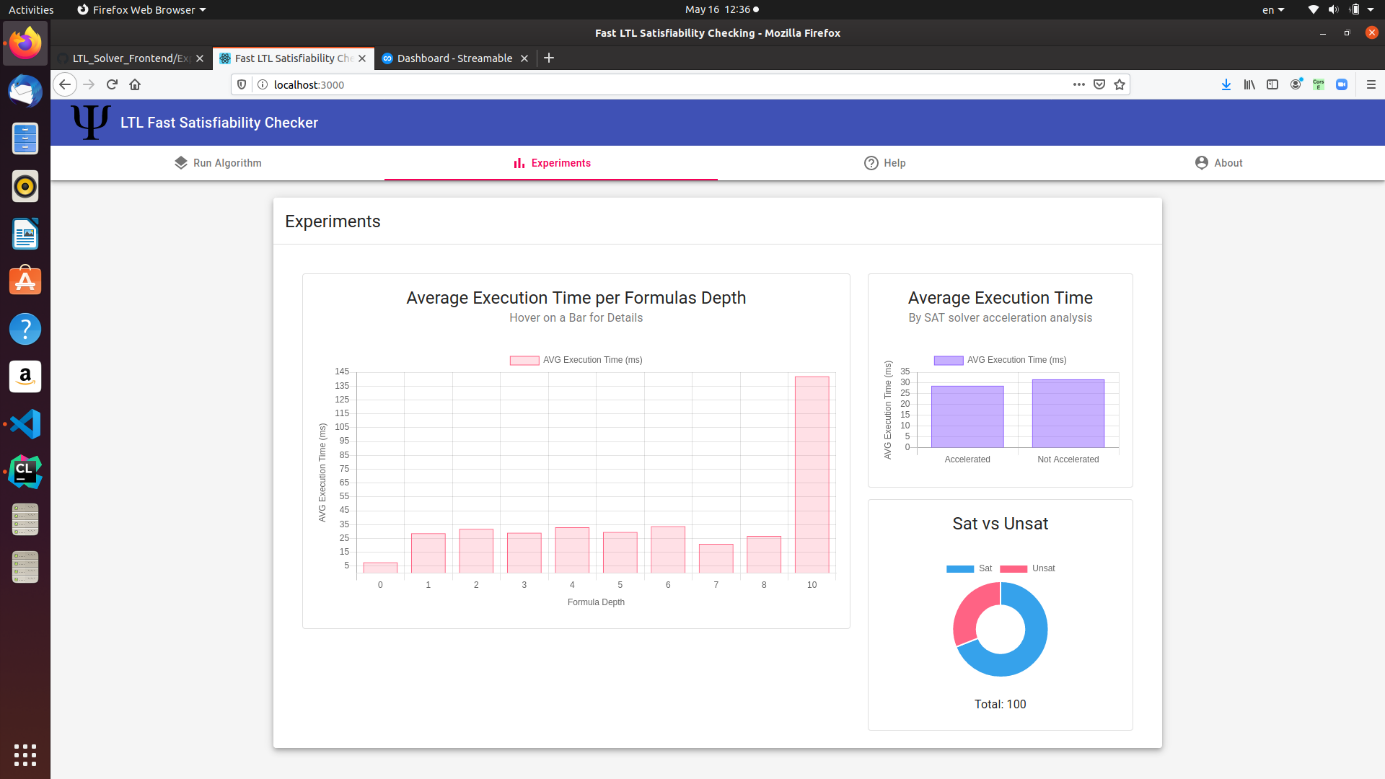
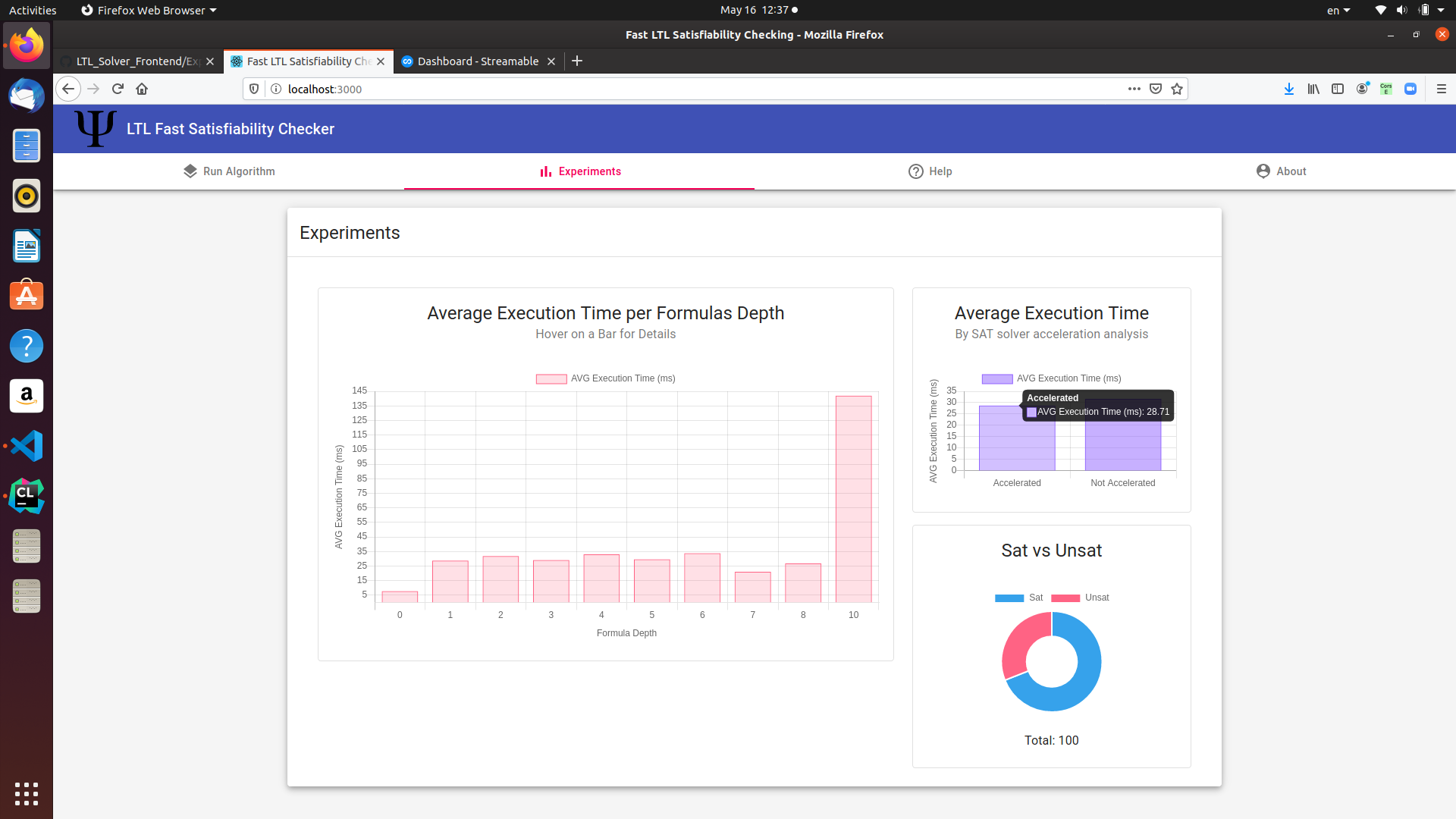


Fig. 18: Display the average execution time for each case of when the SAT acceleration step success and when it is not

Whenever a guidance is necessary, we created a “Help” section which can be accessible by the top navigation menu. The section contains a categorized description for all the system capabilities.

Press on any arrow at the right side to open/close a section

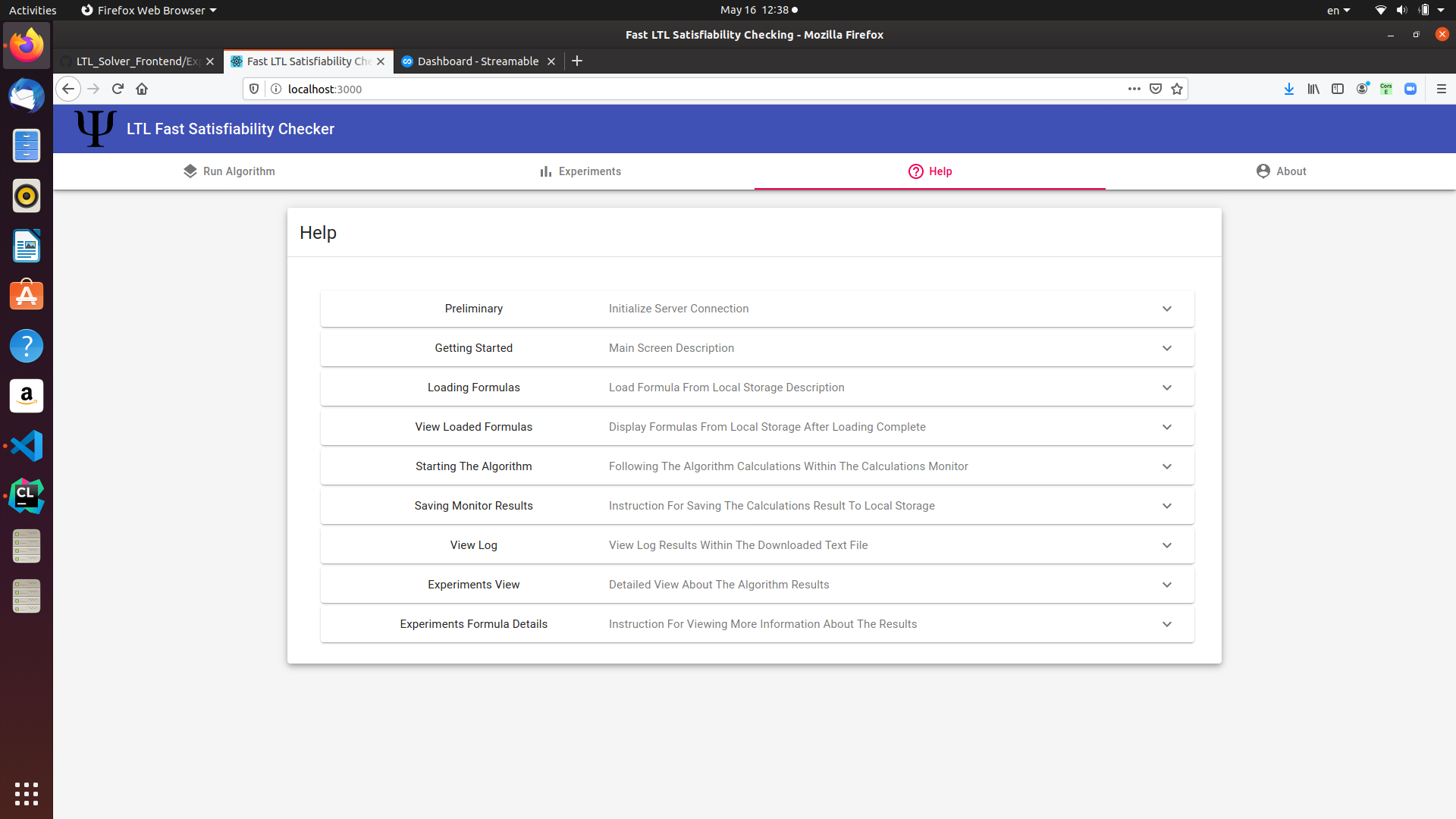


Fig. 19: Help screen

Setup the Server (on Unix environment)

1. Open a terminal and navigate to the directory where the server’s executable file is located in.
2. Run the executable by the command ./Filename (without extension), i.e:

./C\_\_\_Project

Setup the Client

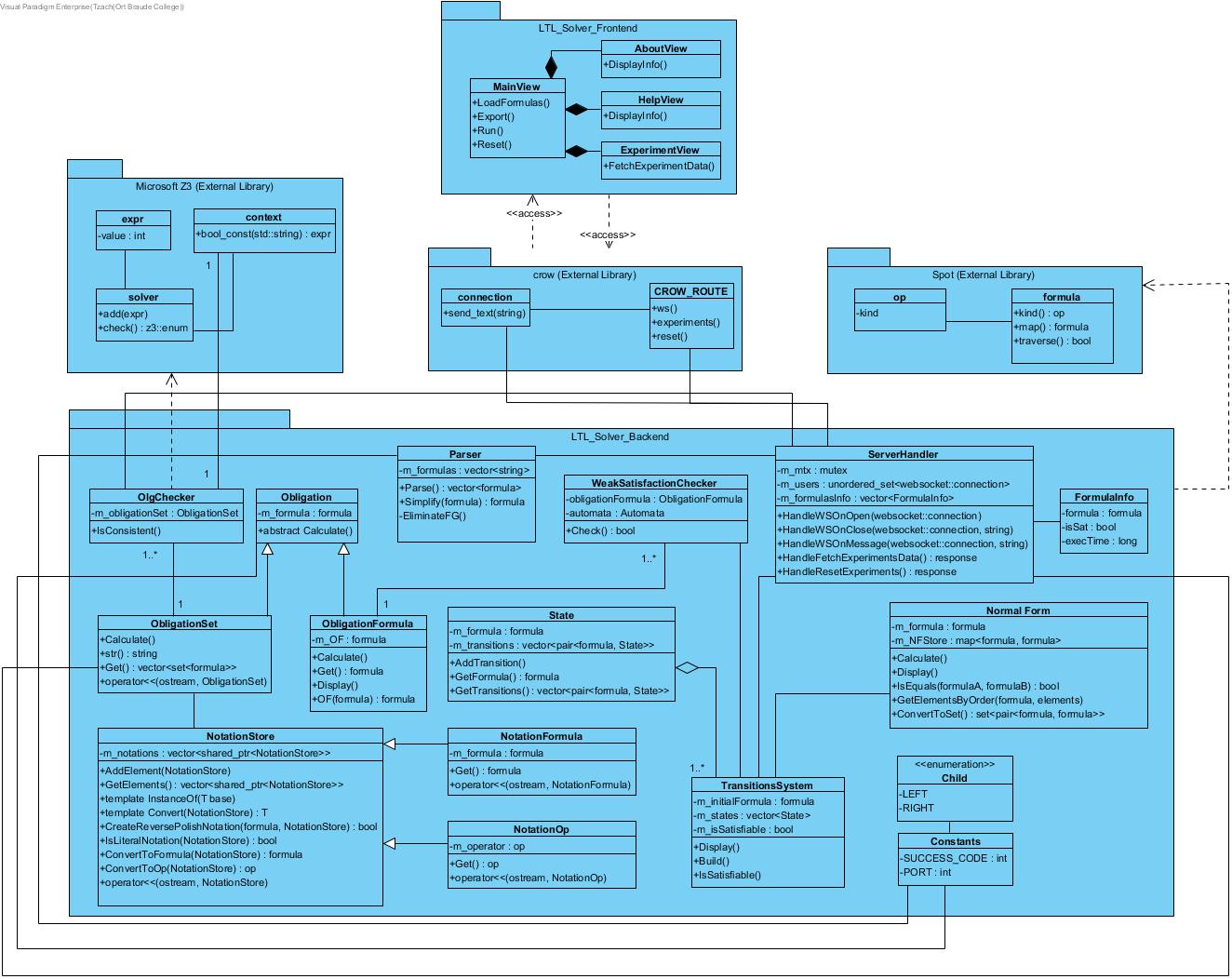
1. Download and install nodeJS: <https://nodejs.org/en/download/>
2. Download and install Visual Studio Code: <https://code.visualstudio.com/>
3. Open the Frontend project’s directory in Visual Studio Code
4. Open a terminal in the VS Code with the keys: CTRL+Shift+` or on the top menu, terminal option.
5. In the terminal run the command: npm install (it will install the project’s dependencies)
6. Open the file /src/constants.js in the VS Code and change the IP to be as yours, so it could connect to the backend:

const API = 'http://<IP>:18080/';

1. Install CORS handler extension to your browser such as:

<https://chrome.google.com/webstore/detail/moesif-orign-cors-changer/digfbfaphojjndkpccljibejjbppifbc>

So, the server will be able to get http requests from the client when they are both run on the same localhost machine, as in our case. Be sure to activate this extension on browser.

1. In the VS Code terminal write: npm start (it should start the project in a browser)
   1.  **Design**
   2. **Testing**

**Browse file**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected results** | **Actual results** | **Comments** |
| BrowseFile1 | Click “Browse File” button  Number of formulas: 0  Click: “Open” | System throws out an error. | Pass | **Precondition:**  There is an empty text file. |
| BrowseFile2 | Click “Browse File” button  Number of formulas: 100 and click “Open” | Display the formulas into the text field. | Pass | **Precondition:**  There is a text file with 100 formulas. |

**Experiments**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected results** | **Actual results** | **Comments** |
| ExpCheck1 | Click “Experiments” on menu before running the algorithm. | System displays information message: “Please run the algorithm first” | pass | **Precondition:**  None |
| ExpCheck2 | Enter: “Main View” screen.  Click: “Browse file”  Load a file with 5 satisfiable and 5 unsatisfiable formulas  Click: “Run”  Click “Experiments View” screen | System displays charts with information about the formulas the algorithm ran on. | Pass | **Precondition:**  There is a text file that contains 5 satisfiable and 5 unsatisfiable formulas. |
| ExpCheck3 | Enter: “Main View” screen.  Click: “Browse file”  Load a file with 5 satisfiable and 5 incorrect format formulas  Click: “Run”  Click “Experiments View” screen | System displays charts with information for only the correct format formulas | Pass | **Precondition:**  There is a text file that contains 5 satisfiable and 5 incorrect format formulas. |

**Solver Check**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected results** | **Actual results** | **Comments** |
| SolverCheck1 | Enter: “Main View” screen.  Click: “Browse file”  Load a file with 5 unsatisfiable and 5 incorrect format formulas  Click: “Run” | Show calculations within the monitor text field, where the results are for correct formula formats. | Pass | **Precondition:**  There is a text file that contains 5 unsatisfiable and 5 incorrect formats formulas |
| SolverCheck2 | Enter: “Main View” screen.  Click: “Browse file”  Load a file with 100 satisfiable formulas  Click: “Run” | Show calculations within the monitor text field, where the result is “Sat” for each formula. | Pass | **Precondition:**  There is a text file that contains 100 satisfiable formulas |
| SolverCheck3 | Enter: “Main View” screen.  Click: “Browse file”  Load a file with 100 unsatisfiable formulas  Click: “Run | Show calculations within the monitor text field, where the result is “Unsat” for each formula. | Pass | **Precondition:**  There is a text file that contains 100 Unsatisfiable formulas |
| SolverCheck4 | Enter: “Main View” screen.  Click: “Browse file”  Load a file with 100 incorrect formulas format  Click: “Run | The monitor text field will not display information about any formula. | Pass | **Precondition:**  There is a text file that contains 100 incorrect format formulas |
| SolverCheck5 | Enter: “Main View" screen.  Click: “Browse file”  Load a file with formulas.  Click “START”  Click “RESTERT” before the algorithm finish. | Stop calculations and back to the main screen with the option to load a new text file. | pass | **Precondition:**  There is a text file that contains formulas |

|  |  |  |  |
| --- | --- | --- | --- |
| **Test No.** | **Test subject** | **Expected result** | **Actual results** |
| 1. | Run the algorithm multiple times | The execution time after the first calculation should be improved | Pass |
| 2. | Load formulas from a chosen file | Retrieve formulas to the formulas text area. | Pass |
| 3. | Run the algorithm on 100 satisfiable formulas | The experiment screen will display in result that there are 100 satisfiable formulas | Pass |
| 4. | Run the algorithm on 100 unsatisfiable formulas | The experiment screen will display in result that there are 100 unsatisfiable formulas | Pass |
| 5. | Check that the Normal Form calculation is correct for 100 formulas. | The calculations monitor will display the text “NF Calculated Successfully” after every Normal Form calculation. | Pass |

Table1. Testing plan

1. **RESULTS AND CONCLUSIONS**
   1. **Results**

First, we must mention that we generated the formulas for the tests with the Spot’s command-line tool with the following parameters: maximum number of literals set to 7 and the formula maximum size to 10, where the size is the number of temporal operators. Such that we retrieved completely random formulas matching those parameters.

In this section we review the system’s behavior by running our application on many variants of formulas. We present the average execution time of formulas as a function of their tree’s depth. Then, we explore the execution times when the SAT solver acceleration method succeeds, and when it does not.

***Experiment 1***

In this experiment we load 100 formulas, then, the average execution time for each formula’s depth group will be presented. We expect to get higher calculation time for larger formula’s depth, but it is not a must, because it is possible that the SAT solver acceleration method will succeed for most of the larger formulas and not for the small ones. The two cases will be presented in the following tests. But first, here is an example for depth detection:

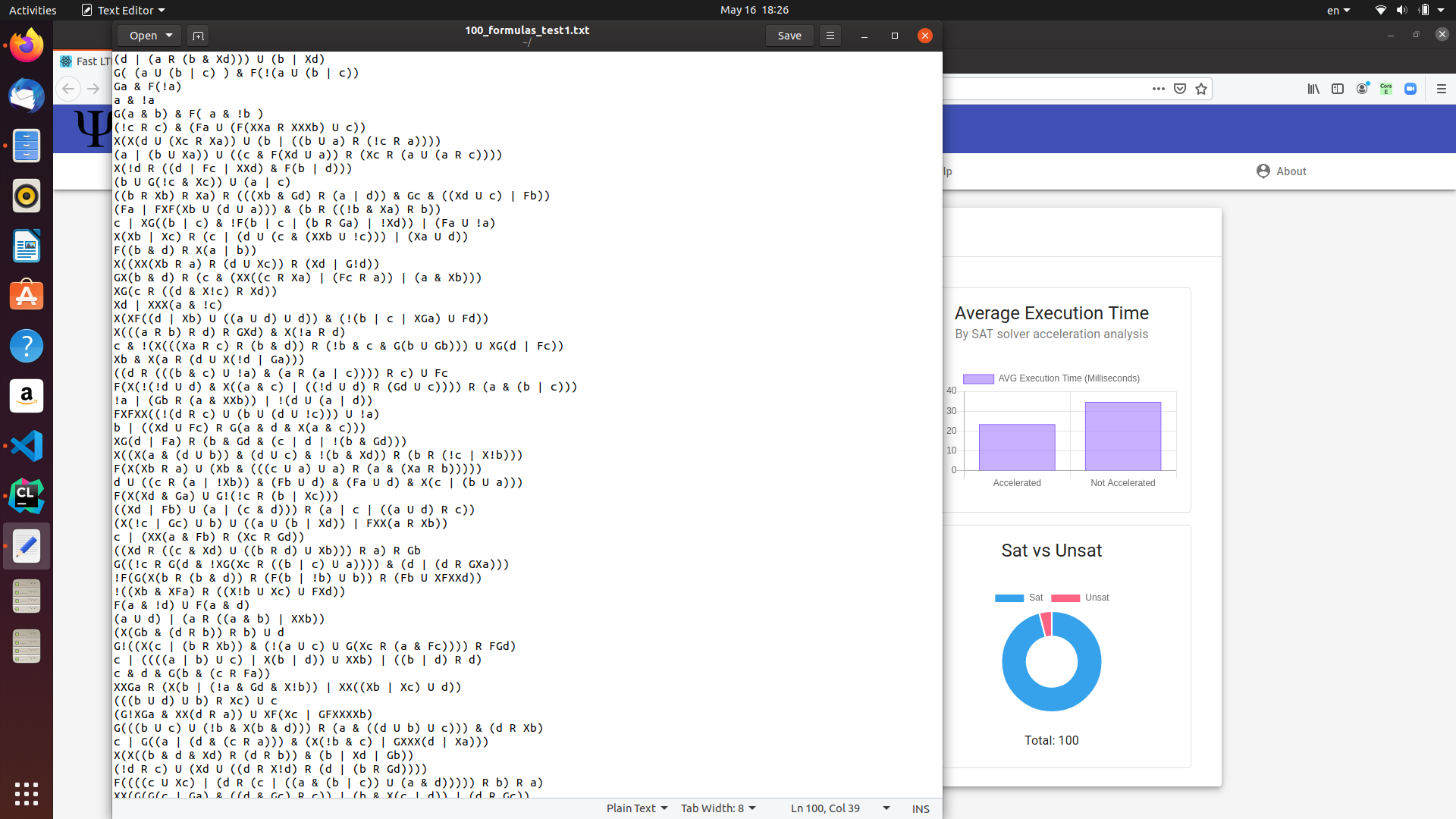
Depth = 0

Depth = 5

Fig. 20: Example of formula’s depth tree presentation

Test 1:

In this test, 96/100 of the formulas are satisfiable, so we assume that the SAT acceleration method will work for most of the formulas, therefore the execution times expected to be relatively low.



The parameter we insert is the related text file, where it contains:

96 SAT formulas,

4 unSAT formulas.

Maximum number of literals: 7

Maximum formula’s size: 10

Fig. 21: Input file for test 1

Fig. 20: Example of formula’s depth tree presentation

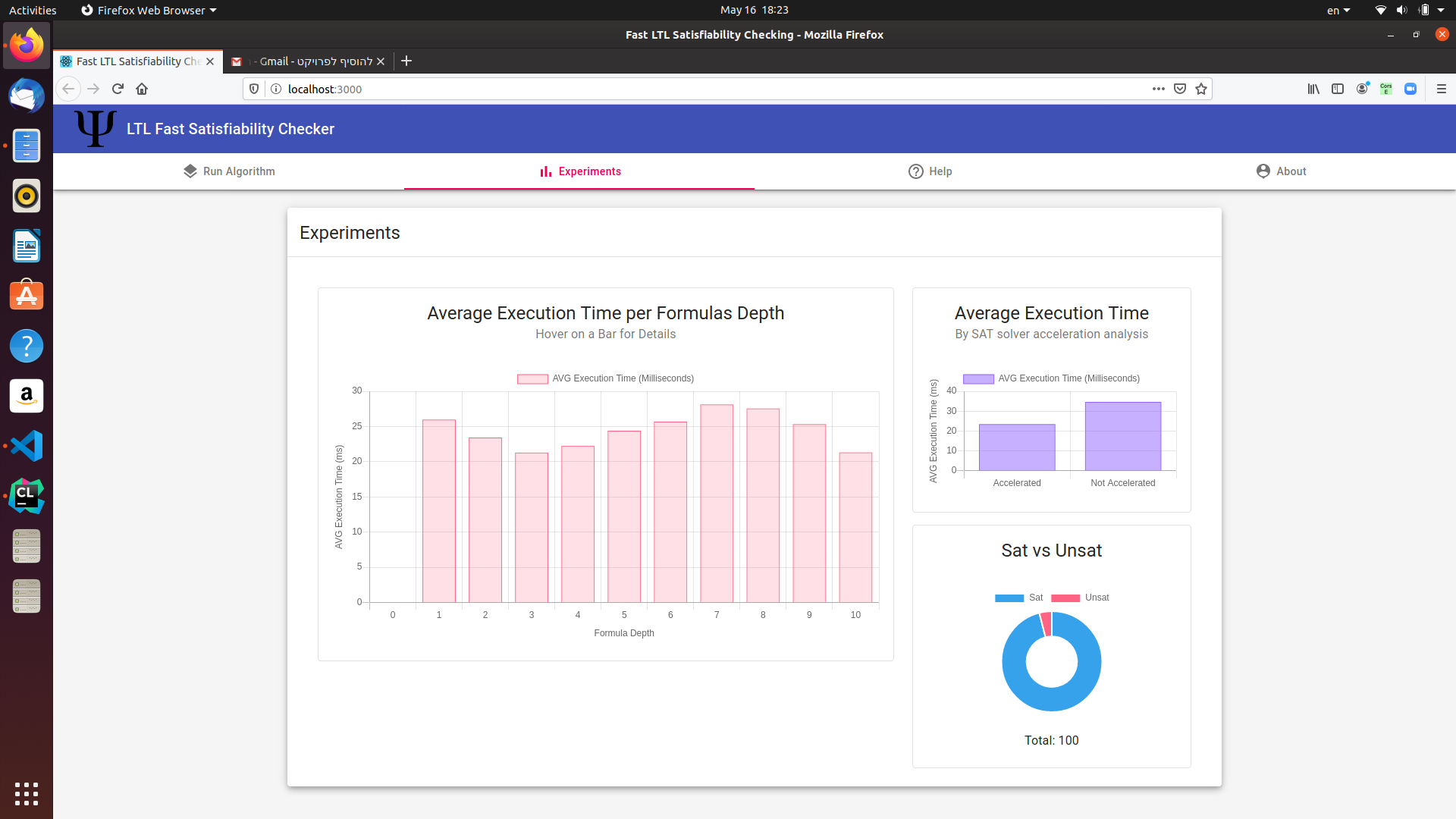


Fig. 22: Formulas distribution

Note that the calculations applied on the formulas after simplifications, therefore, there are cases where the formulas transformed to true/false, so it takes less than 1 millisecond to return an answer.

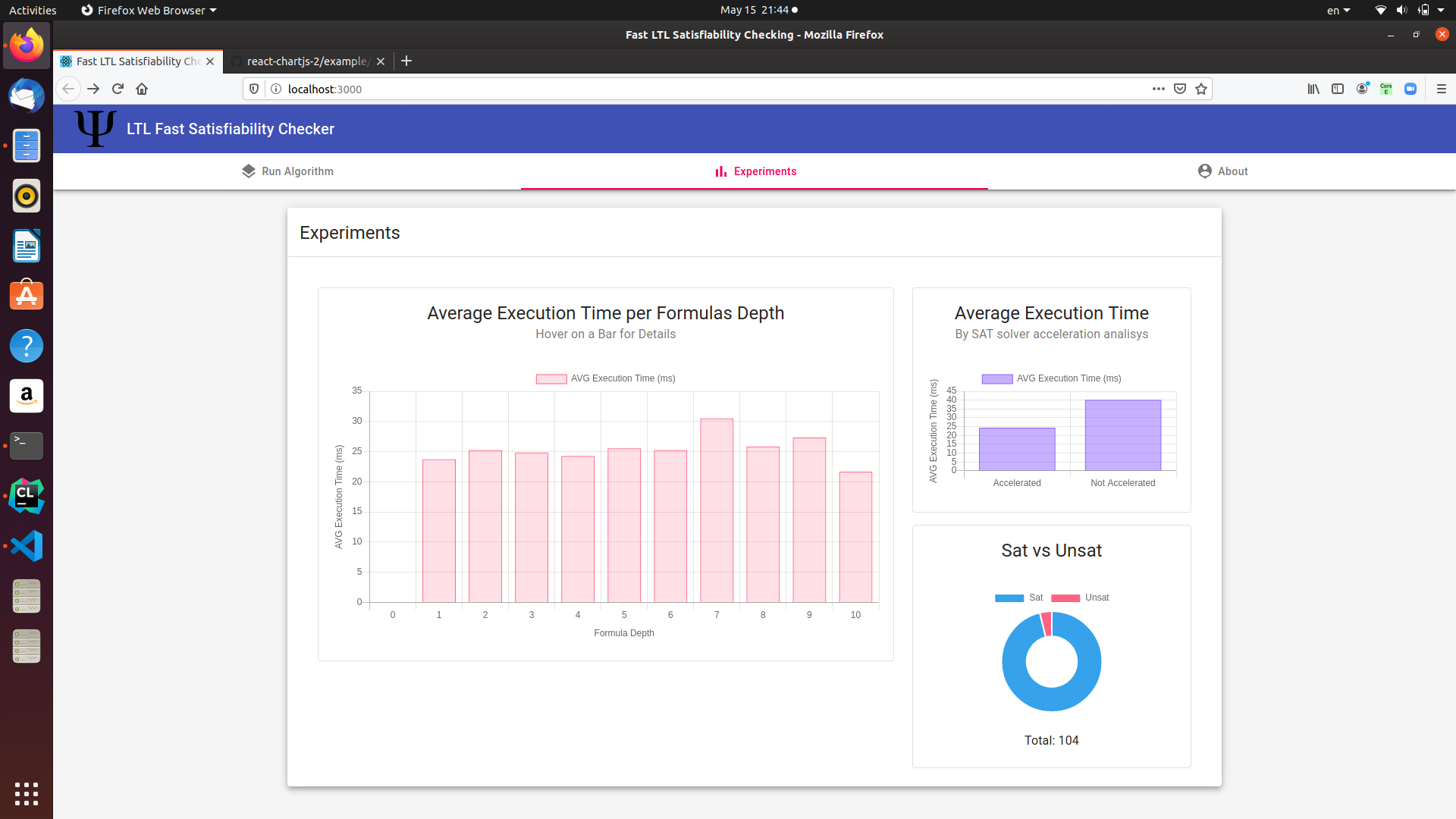
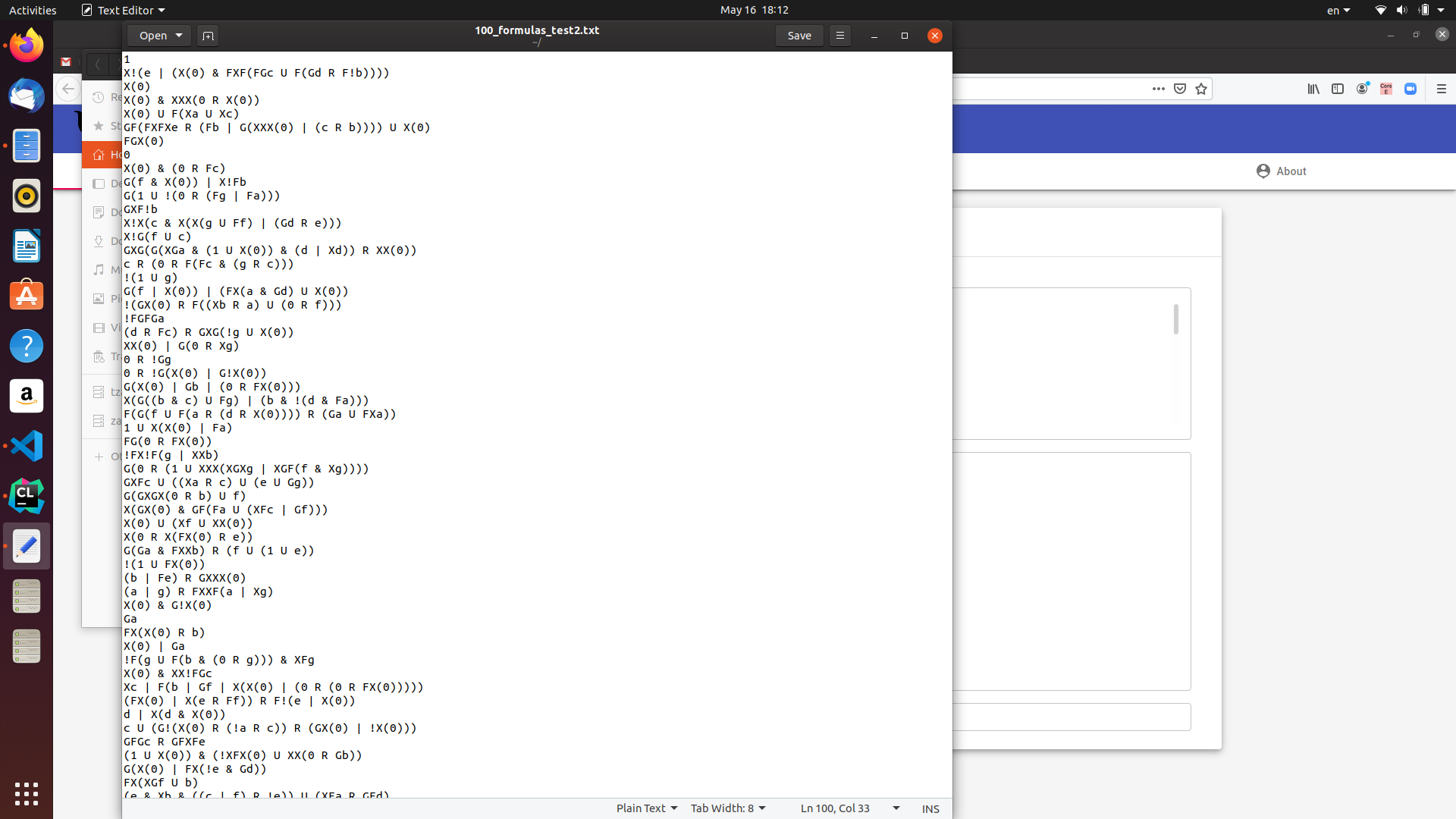


Fig. 23: Depth execution time experiment results

Test 2:

In this test, there are 31/100 unsatisfiable formulas. In this case the SAT acceleration method will not work for at least 31 formulas, therefore, the execution times expected to be higher.



The parameter we insert is the related text file, where it contains:

69 SAT formulas,

31 unSAT formulas.

Maximum number of literals: 7

Maximum formula’s size: 10

Fig. 24: Input file for test 2

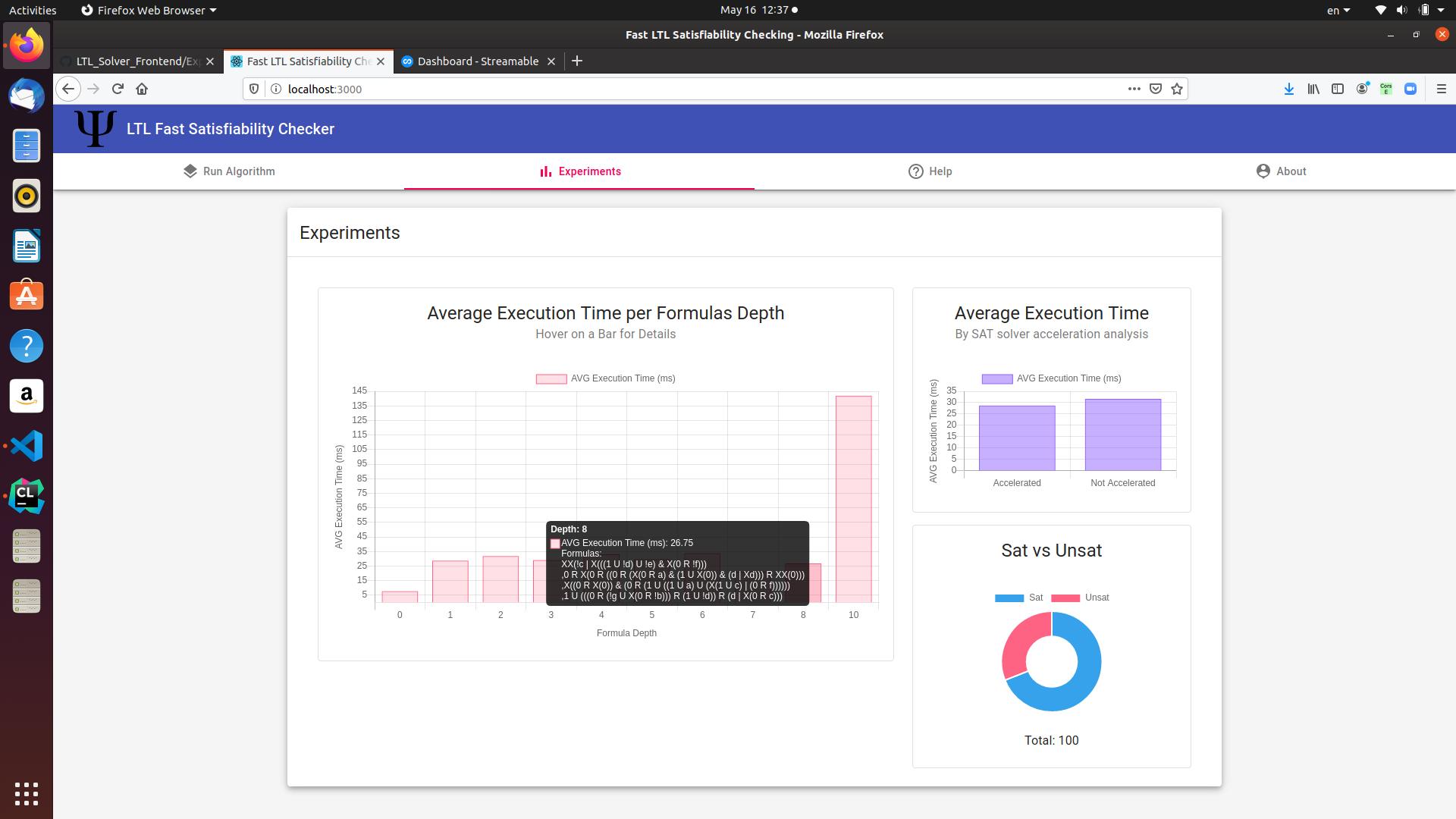


Fig. 25: Formulas distribution

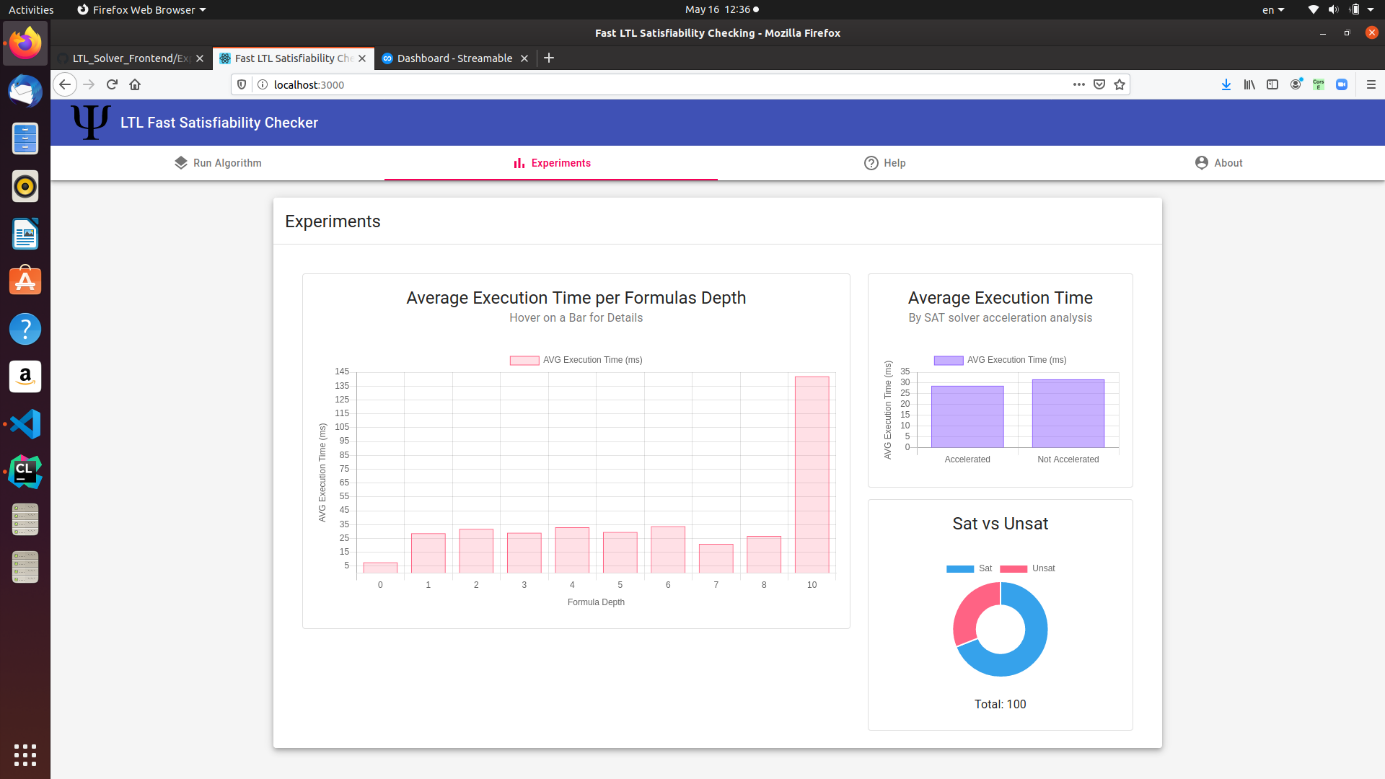
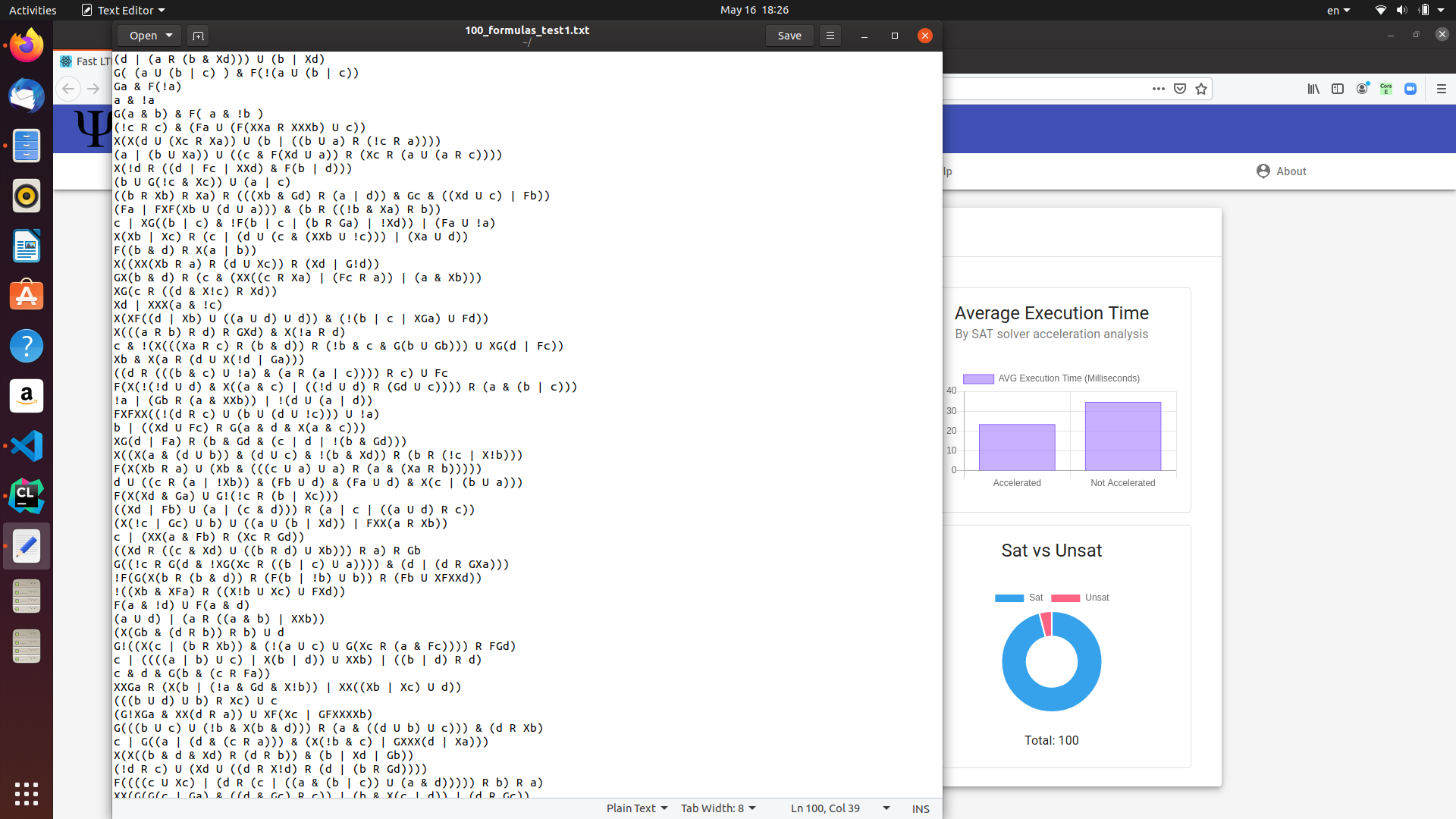


Fig. 26: Depth execution time experiment results

***Experiment 2***

In this experiment we will load 100 formulas and compare the average execution time for the cases when the SAT solver acceleration method success versus the cases when it does not. We expect to get much lower average execution time for the cases when the accelerated method succeeds.

Test 1:

The parameter we insert is the related text file, where it contains:

96 SAT formulas,

4 unSAT formulas.

Maximum number of literals: 7

Maximum formula’s size: 10

Fig. 27: Input file for test 1

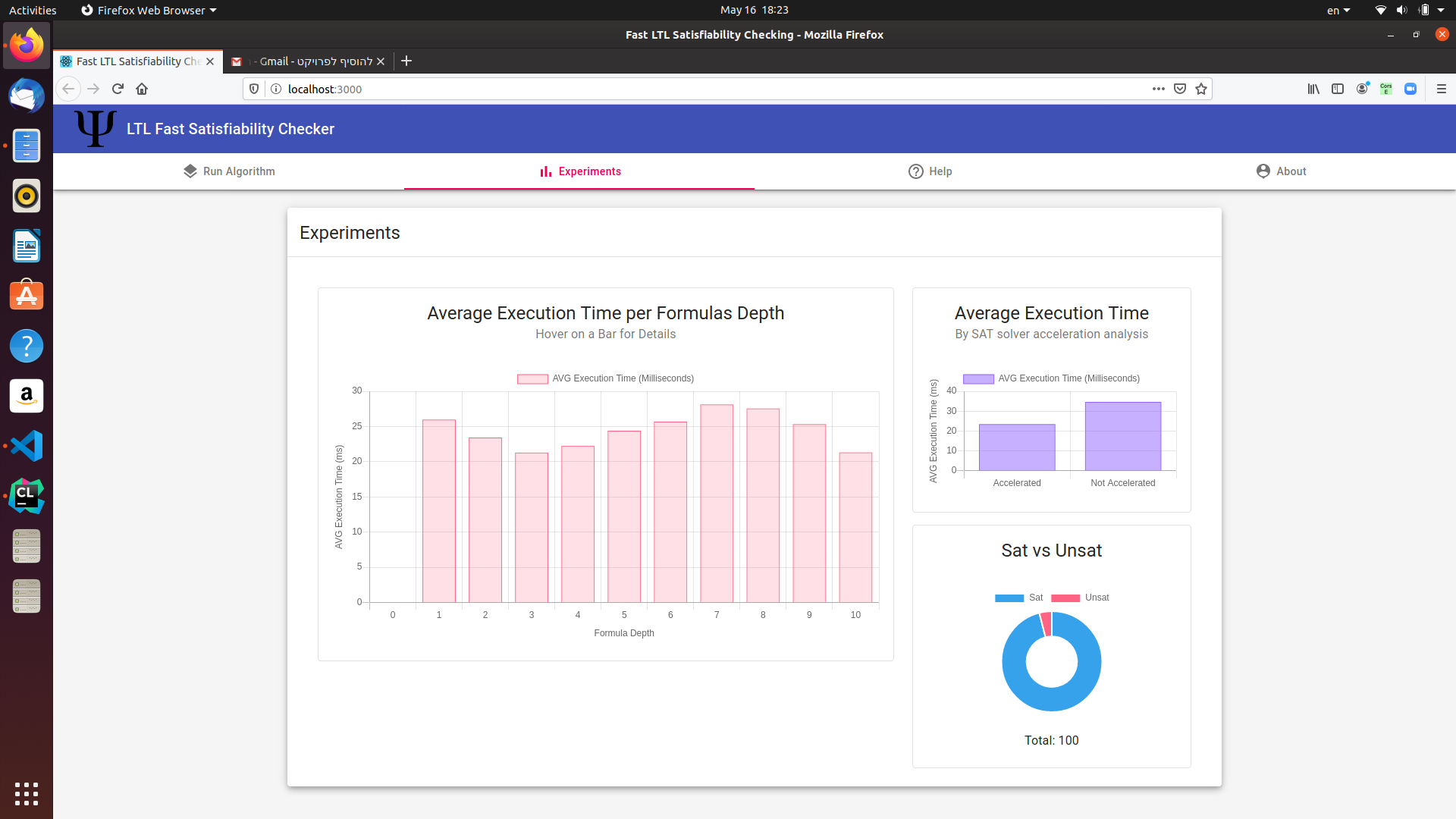


Fig. 28: Formulas distribution

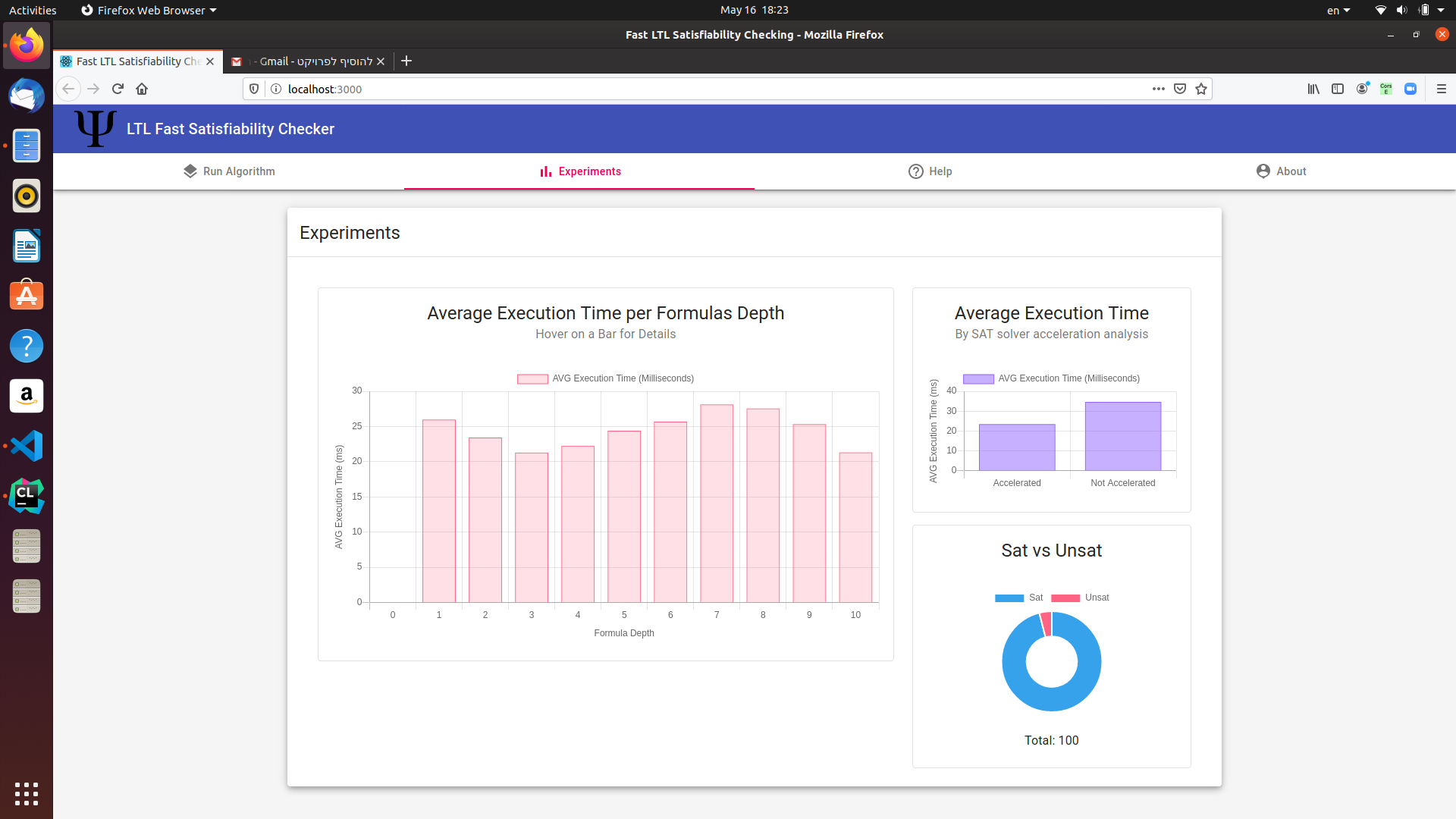
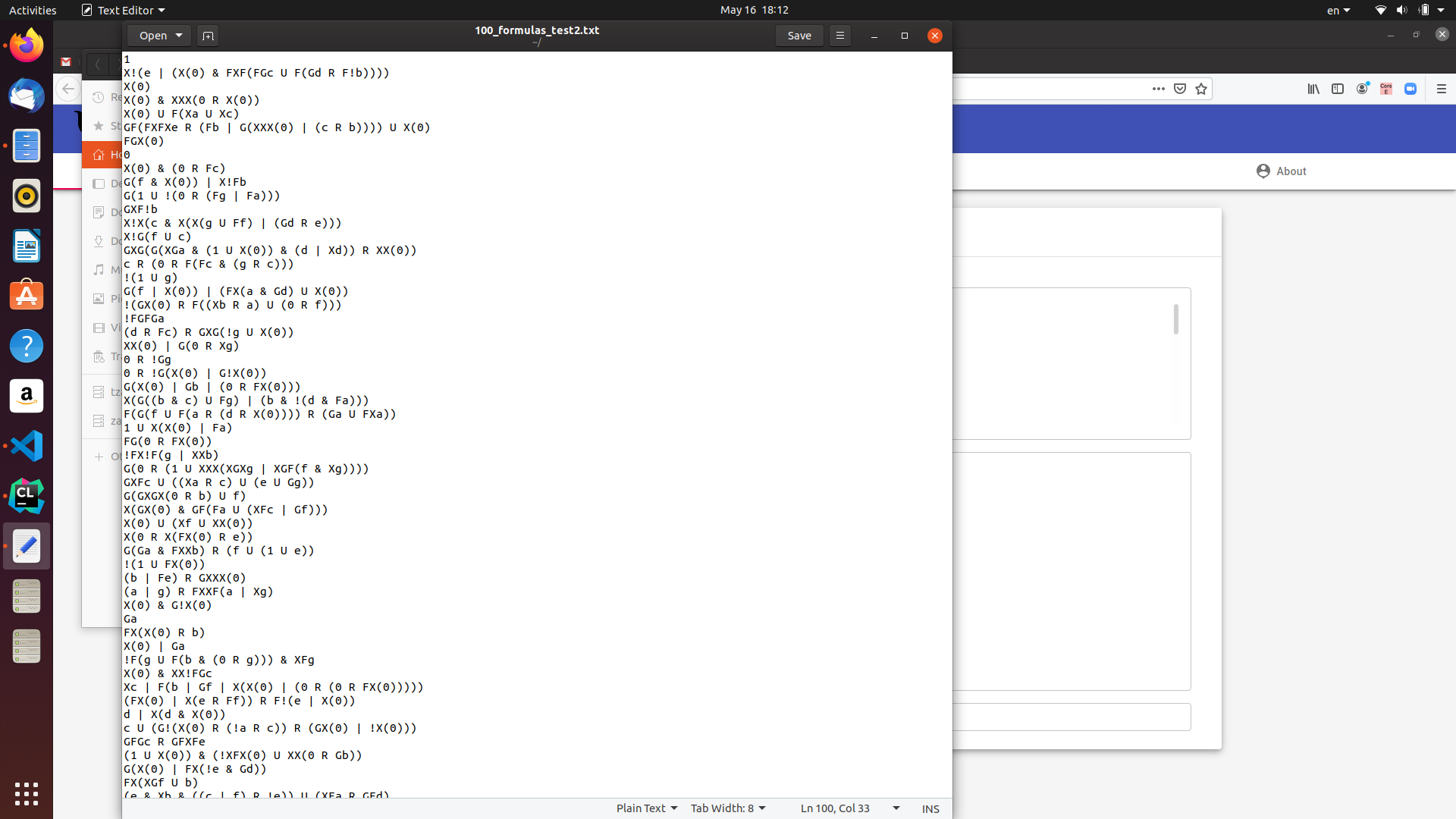


Fig. 29: Experiment result of SAT solver usage method

Test 2:



The parameter we insert is the related text file, where it contains:

69 SAT formulas,

31 unSAT formulas.

Maximum number of literals: 7

Maximum formula’s size: 10

Fig. 30: Input file for test 2

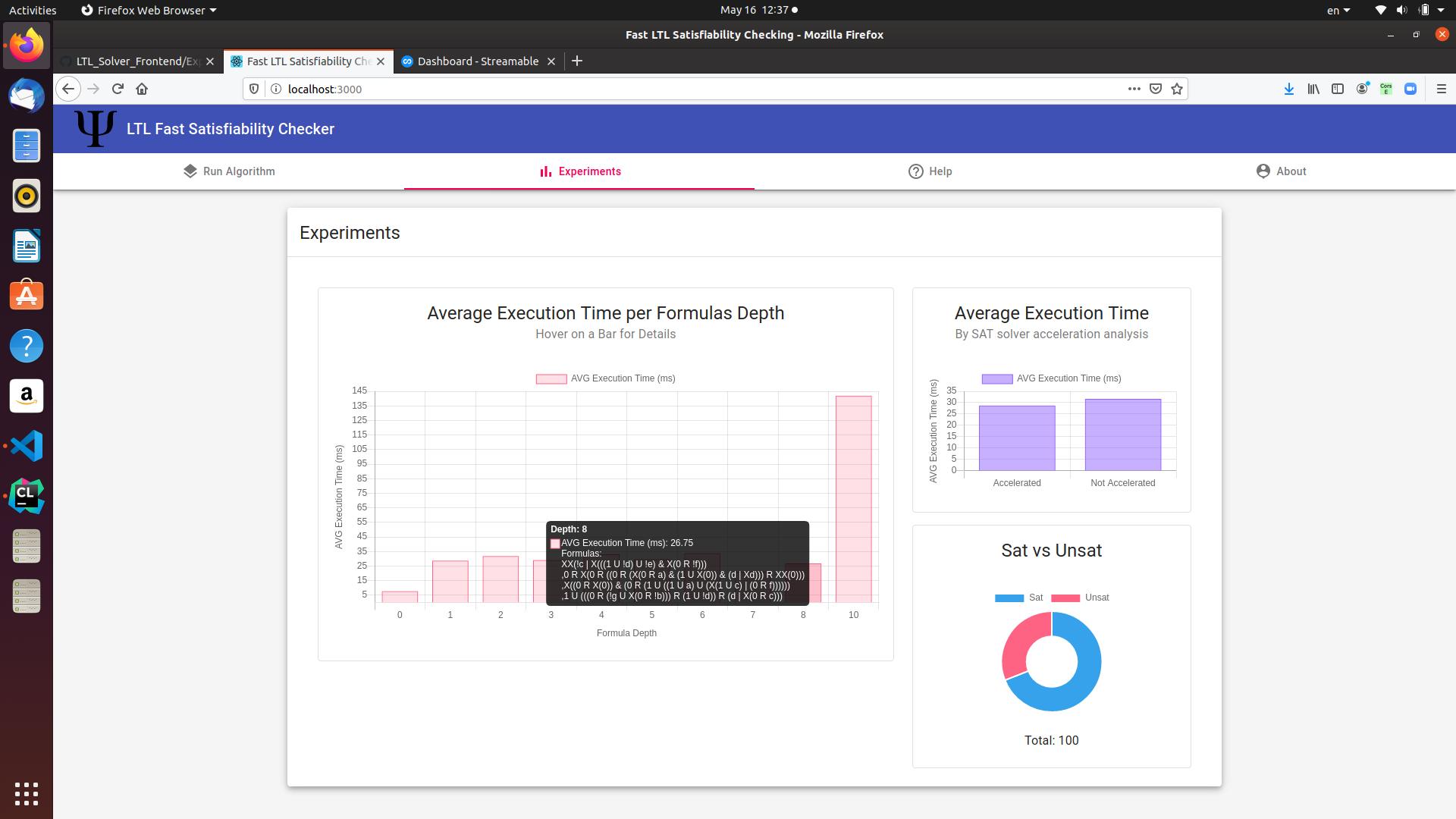
****

Fig. 31: Formulas distribution

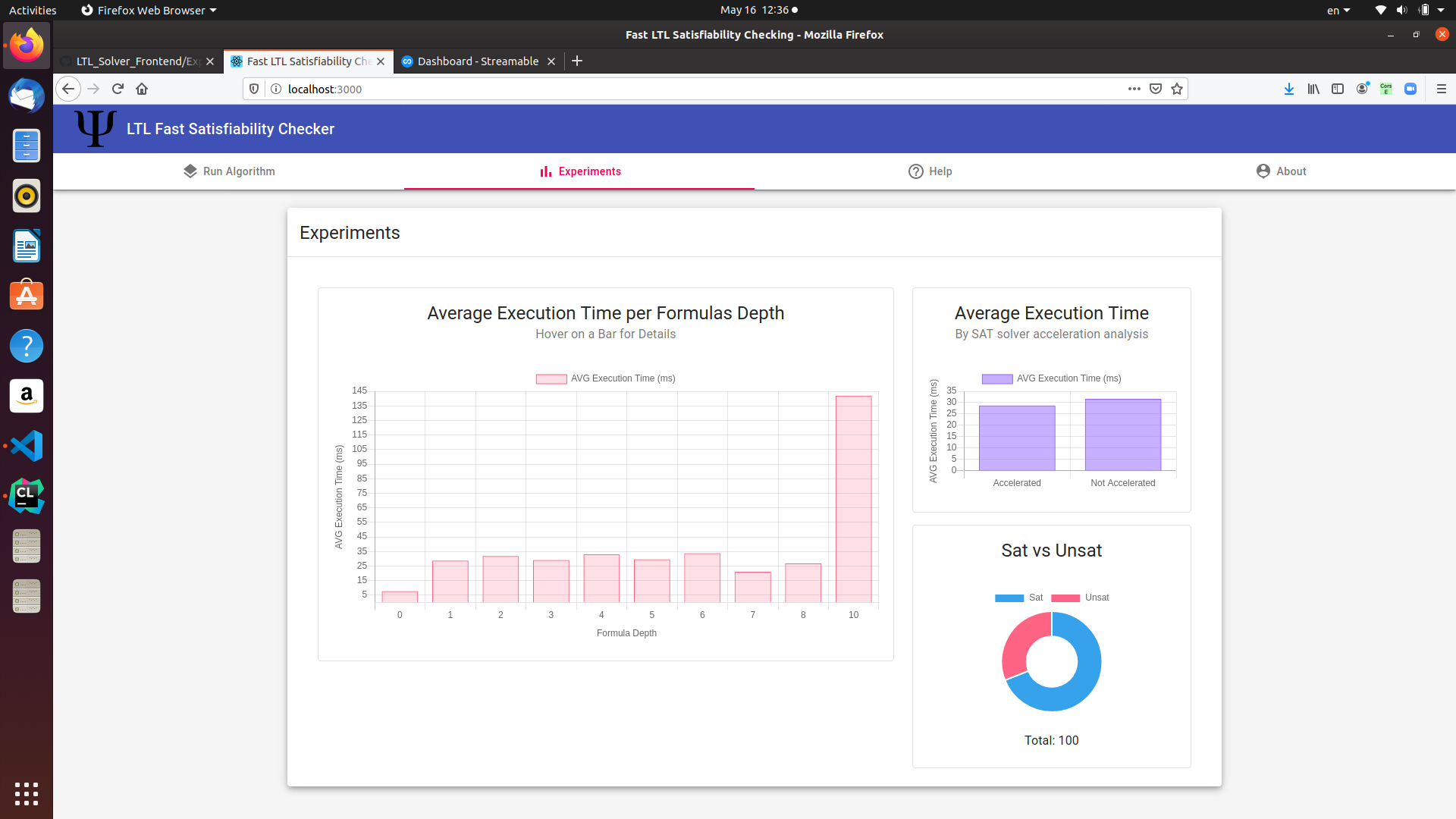


Fig. 32: Experiment result of SAT solver usage method

* 1. **Conclusions**

At the experiment stage, we have made many executions on different types of formulas where some of them are satisfiable the others are not, when sometimes the satisfiable formulas cannot be analyzed by the SAT solver, trying to understand better how the algorithm responds to the various cases. We learned that the SAT acceleration method is clearly improves the calculations time performance whenever it can be used. On the other hand, by a given formula’s depth, we cannot say that its calculation time will be higher or lower due to the lack of information whether it is can be solved by the SAT solver tool. Another important criterion is the formula’s size, which increases the calculations complexity because the transitions system’s states number would be increased, so it directly affects the tests where most of the formulas are unsatisfiable.

According to first experiment we can see that for the first test, the largest formulas requires a short calculation time. That is because most of the input formulas in this case are satisfiable, which means that there are better chances for the SAT solver to determine their satisfiability. On the other hand, when we executed the algorithm on different formulas, where the number of unsatisfiable formulas is larger than their amount in the first test, then we can say by the results, that the SAT acceleration method does not work for the major cases and larger depth formulas actually gets high execution time.

From the second experiment we can say that, the more formulas that can be analyzed by the SAT solver-the better calculation performance we achieve. For example, for the first test, the major formulas are satisfiable, which means that the SAT solver will detect the satisfiability of them most. As result, we can see that the average execution time on them is about half of the execution time for the formulas that the SAT solver did not classify as satisfiable, or that they are unsatisfiable. From the second test case, it is seen that we run the algorithm on more unsatisfiable formulas then we did for the first test, and it is still appears that the execution time of the accelerated method is lower, not by far but it is because the general execution time on these formulas input is pretty low.

In summary, we successfully implemented an accelerated algorithm for the complete decision procedure of LTL formulas satisfiability checking, just as we expected in the introduction.

**REFERENCES**

[1] K.Y Rozier and M.Y. Vardi. “LTL satisfiability checking”, *Int’l J. Software Tools Technology Transfer*, 123-137, 2010

[2] K.Y Rozier and M.Y. Vardi. “A multi-encoding approach for LTL symbolic satisfiability checking”, *Int’l Symp. Formal Methods*, 417, 2011

[3] A.P. Sistla and E.M. Clarke. “The complexity of propositional linear temporal logic”, *Journal of the ACM*, 32(3), 733-749, 1985

[4]J. Li, G. Pu, L. Zhang, M.Y. Vardi and J. He "Fast LTL Satisfiability Checking by SAT Solvers", *State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences*, 1-10, 2014*.*

[5]J. Gu, P.W. Purdom, J. Franco and B.W. Wah "Algorithms for the Satisfiability (SAT) Problem: A Survey",*DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1-3,1997*.*

[6] E. M. Clarke, A.Bierea, R. Raimi, and Y. Zhu. “Bounded model checking using satisfiability solving”, *Formal Methods in System Design*, 7-34 2001

[7] J. Li, L. Zhang, G. Pu, M. Vardi, and J. He. “Ltl satisfiability checking revisited”, *State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences*, 2013

[8] Leonardo de Moura and Nikolaj Bjorner “Z3: An Efficient SMT Solver”, *Tools and Algorithms for the Construction and Analysis of Systems 337-340*, 2008

[9] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu “Spot 2.0 – a framework for LTL and w-automata manipulation”, *In Proc of ATVA’16, LNCS 9938,* 122-129*,* 2016

[10] Jaeseung Ha “Crow C++ micro web framework”, <https://github.com/ipkn/crow>