Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Workflow interface for tracking and editting a program

Author:
Xiao Luo

Supervisor:
Fariba Sadri

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science of Imperial College London

September 2018

Abstract

LPS(Logic-Based Production System and Agent Language) is a logic-based programming language. It can be abstract and hard to comprehend for a novice. An interactive visible graph interface may be more descriptive, therefore can help beginners to learn LPS. The outcome of running LPS is actions, events, and fluents occur in sequential order in a time line, so LPS share some similarities with workflow. Thus, A way to implement the interactive interface is to create a workflow graph. In this report we describe the architecture and implementation of a workflow interface for LPS. It is hoped that this graph interface can provide a way to express LPS by graphs and make it easier for people to understand and create LPS programme.

Keywords: logic programming systems, workflow graph, web interface

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. Fariba Sadri, who gives me constructive comments and warm encouragement throughout the whole process. Without her guidance this paper would not be materialized.

Contents

Chapter 1

Introduction

1.1 Motivations and Objectives

Coding in pure text can be complicated and abstract to a novice. Users frequently have to remember almost all the keywords and syntaxes, and make sure that the program is grammatically correct, in order to get the final output. Even a tiny mistake can block the running of a huge program. Even if we get all problem fixed, the process of program running might be confused, especially when the running outcome is relatively simple. Although nearly all programs provide debug mode, the debug commands can often be more complex than coding itself in the beginning. The above facts all make teaching and learning programming difficult.

With the development of modern information technology, graphic user interfaces (GUIs) and web based interface has come out to make programming easier, either by automatic cueing or by rendering the running outcome to be more descriptive. Nevertheless, even with those advanced tools, users still have to input the code themselves, and it is still complicated to a novice. Therefore, a better way need to be found to make programming easier for beginners.

When illustrating the process of program running, one of the best ways is to draw a workflow graph. Workflow graphs are really descriptive, and they demonstrate the things happened sequentially. Some programming tools such as GUIs have already implemented that kind of functionality to generate a graph as the outcome, yet the reverse process to generate code out of graph is seldom implemented. This gives us the idea to start from drawing workflow graphs and translate them into programs.

The advantages of programming in workflow graphs are as follows: The elements of workflow graphs are presented, so users do not have to remember the keywords. Users do not need to write the code directly, lowering the possibility to make grammatical mistakes. In addition, Workflow graphs are a straight forward way to demonstrate the intrinsic logic of programming.

This report discusses easyLPS, an online workflow interface which can be translated

into LPS(Logic-Based Production System and Agent Language). EasyLPS consists of a client application written in html, CSS, and javascript, and a web application server written in Java. It is deployed on Microsoft Azure, and can be accessed by https://easylpsazure.azurewebsites.net/.

The objectives of the project are:

1. To build a workflow interface on which users can draw graphs to express LPS clauses and logics.
2. To develop a translator from workflow interface to LPS programs.
3. To allow editting LPS programs via workflow interface.

1.2 Contributions

The contributions of the report are to:

1. Provide a detailed description and guidance of a novel workflow interface which can be translated to LPS.
2. Evaluate the workflow interface and present future directions.
3. Demonstrate a way to express LPS program by graphs.

1.3 Statement of Originality

I declare that this report is completed by myself, and has not been submitted before to fulfill another degree or diploma. The report and the codes contain no material written by others except where references were made.

Chapter 2

Background

2.1 LPS

LPS (Logic-based Production System) is an extension of logic programming.[8] It inherited most of the features in Prolog. Computation in LPS generates a timeline of events, fluents, and actions.[8]

Actions, fluents, and events are three basic elements of LPS program. Each element in the program is represented as a predicate symbol potentially followed by a set of constant or variable symbols in a bracket. In order for LPS to distinguish, constant symbols start with a lower-case letter and variable symbols start with an upper-case letter. The underscore character "_" is a wild match character which can represent everything. The elements are reflections and abbreviations of real life meaning. For example, arrive(london) means the action to arrive in london(a constant) in LPS, while arrive(X) means arrive in variable X. In the same way, locates(bill, london) can represent that bill locates in london. The number of Constants and/or Variables in the bracket is called arity: locates(bill, london) is a binary predicate(a predicate with arity of 2). In addition, constants or variables can also be lists surrounded by square brackets. There can also be elements represented as a nullary predicate(proposition), for instance, predicate "switch" may mean the action to switch on or switch off the light.

One of the unique feature of LPS is that it is highly related to time. At the beginning of the program there can be a unary predicate maxTime(_) to define the maxmimum length of timeline in program output. Actions and events happen at a time point, while fluents occur in a period of time. For instance, in the timeline, switch from 4 to 5 means the action switch happens at the time point between time period 4 and period 5, while light_on at 5 means the fluent light_on holds during time period 5. There is no major difference between actions and events, except that actions is observed in the initialization part of LPS, that is to say, actions are intensional predicate at a time point while events sometime can be extensional. Therefore, in most cases, events can be viewed as a subset of actions and it is not incorrect to put events into actions.

From my perspective, similar to other commonly used programming language, the structure of a typically LPS program can be divided into three parts: declaration, initialization, and program body. Declaration part includes all wild match of actions, fluents, and events appeared in the program: the predicate symbols and if arity greater or equal to one, followed by underscores splitted by comma in a bracket with equal number to arity. Initialization part states the observed facts, including which action happens at which time, and which fluent holds initially. The program body mainly describes the logic rules of the following kinds(syntax)[2][5][6]:

1. Reactive Rules. Reactive Rule is a type of if condition. In logic semantics, it is accompanied by time stamp, indicating that if antecedents happen, then consequents will be true in the time afterwards.

2. Macro Actions. Macro Action is a kind of logic program, and has similar format to action. It can be considered as a bundle of a series of actions and fluents. For instance, if we make bread, make cheese, and put cheese on bread sequentially, the whole process can be considered as a macro action make sandwich.

3. Causal Theories. In LPS, fluents show the status of objects, and are extensional predicates, that is, we cannot modify it directly apart from initialization process. For example, we cannot directly control whether the light is on or off, but we can control it by the switch action. an action initiates or terminates a fluent, sometimes under certain conditions is the majority form of causal theories.

4. Preconditions. Preconditions is also called False Conditions, meaning that some actions and/or some fluents cannot happen together under certain conditions. For example, we have an action cut_bread and a fluent bread, and cut_bread cannot happen if fluent bread does not hold.

2.2 SWISH

SWISH[1] is an online SWI-Prolog environment. It is a subset and an extension of SWI-Prolog[9]. Users can input the program of prolog or LPS in the textarea of left half page, and run commands in the right bottom area. The final results will be shown in the right top region.

To help users understand the LPS program, LPS specify syntaxes in the text area. Those syntaxes includes actions, fluents, and events, as well as keywords such as if, then, initiates, terminates, and updates. The pre-processor of SWISH also conduct a process similar to compile, indicating whether there are any syntax errors in the program before running.[8]

The results of LPS in SWISH will be calculated on each time interval, and if actions, fluents, and events occur at certain time or during certain periods, they will be demonstrated on the time line graph. Macro actions will also be calculated, yet

not shown on the graph. If several circumstances satisfy the LPS program, the first one will be demonstrated, while if program run into some ”runtime error”, ’Program failed’ message will be printed, but actions, events, and fluents happen before will still be shown in the time line.

2.3 Workflow

Workflow describes a time series of events, actions, and the relationship between them. It demonstrates the idea of natural parallelism and fork-join patterns.[3] It was introduced into computing area to resolve the problem that early computer applications are designed to solve specific problem only and find it difficult to handle changed situation. [4] As a programming model, workflow contains the thought of multi-processing and helps programmers identify and exploit parallelism. [3]

Workflow graph is a way to model control flow using workflow, and different workflow graph can actually model same kinds of behavior[7]. The basic composition of workflow graphs are the elements and the relationship between elements. All workflow graph consists of a start point, and at least one end point. The relationship between elements indicates the time sequence of occurring. Therefore, workflow graphs are sometimes used as the graphic demonstration of logic program.

Chapter 3

Design

According to the purpose of the project, to convert a workflow interface to LPS code, it is quite natural to adopt the concept of Object Oriented Programming, to design several classes to hold the workflow graphs themselves as well as the elements in the graph.

3.1 Front End

The front end is implemented by HTML, CSS, and javascript. HTML declares the content of web page, CSS describes the style, while javascript implements the function related to all mouse and keyboard events. Data from the front end are packed in Json strings and sent to back end via Ajax.

Since it is the first time I do front end with HTML, CSS, and javascript, I decide to make use of raw javascript as much as possible, and avoiding importing too many unnecessary javascript libraries. Although the length of code may increase, raw javascript can be supported by nearly all web browsers, and can show the implementation details directly, fulfilling my software requirement accurately. In terms of complicated data processing, simply hand over to the server side via Ajax. The reason why I choose Ajax is that I want to only update parts of the web page rather than reloading the whole web page. It is probably more user friendly to let all user inputs remain on the web page.

3.2 Back End

The back end, which is the server side, is implemented by Java. Both Java and C++ are object oriented programming language, but C++ might be closer to a machine language, which means, may take more effort to do coding. Regarding the web page, speed performance does not really matter as long as the result can be demonstrated within a second. Therefore, java is preferred, given that it is both powerful and elegant.

Many popular Java web application frameworks are available, and the majority of modern frameworks are based on servlet. Considering that the workflow interface is only a web page while nearly all the modern frameworks are too large and too powerful, I choose to implement the servlet myself to avoiding using those complex frameworks. Therefore the functionality of back end (server side) mainly consists of three parts:

- servlet part, to accept http request with raw data and send back response with processed data.
- domain object part, which is created to store the detailed information about the web page, graphs, and elements in the graphs from the front end
- data processing part, to convert data into LPS code

3.3 Graph Expression of LPS

In the project, we represent the observations and initial states by Initialization Graph, the reactive rules by Reactive Rule Graph, and the macro actions by Macro Action Graph. Considering that Causal theories and Preconditions are not related to time, they are represented by pure text.

3.3.1 Initialization Graph

Below shows an instance of Initialization Graph. Ten lines are drawn evenly to demonstrate the time from 1 to 10. The green rectangles mean atomic actions, and the position indicates the time of observation. For example, the green rectangle in the middle means:

```
observe ignite(bed) from 4 to 5.
```

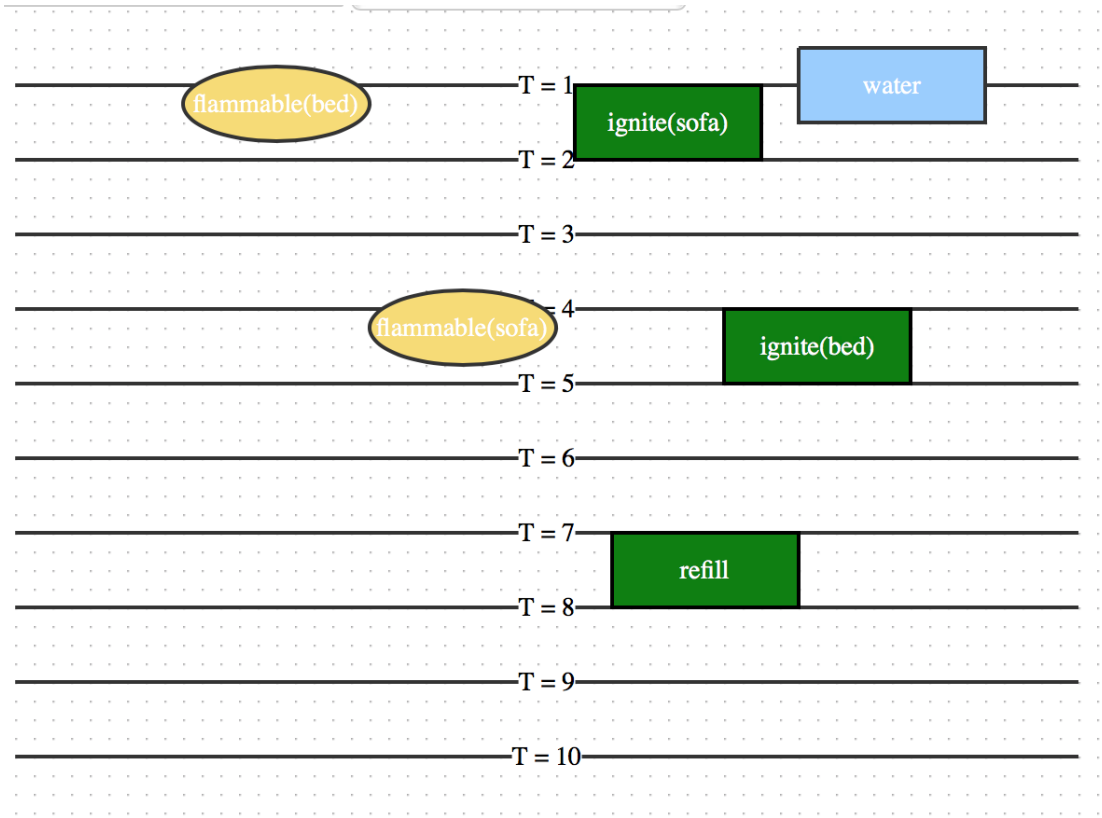

Figure 3.1: Initialization Graph

The blue rectangles express the initial states of fluents. For instance, the only blue rectangle means:

```
initially water.
```

Although the initial states are supposed to be expressed at or before time 1(line T = 1), there might no be enough space if the number of initial states of a initialization graph is too large.
The yellow ellipses declare the timeless facts, and in the same way, the positions of them make no difference. The two ellipses in LPS represent:

```
flammable(bed).
flammable(sofa).
```

3.3.2 Macro Action Graph

Below demonstrates an instance of Macro Action Graph. Similar to Initialization Graph, ten time lines are drawn evenly, but these lines just indicate the time sequence, and adjacent time lines sometimes may or may not represent consecutive time points.

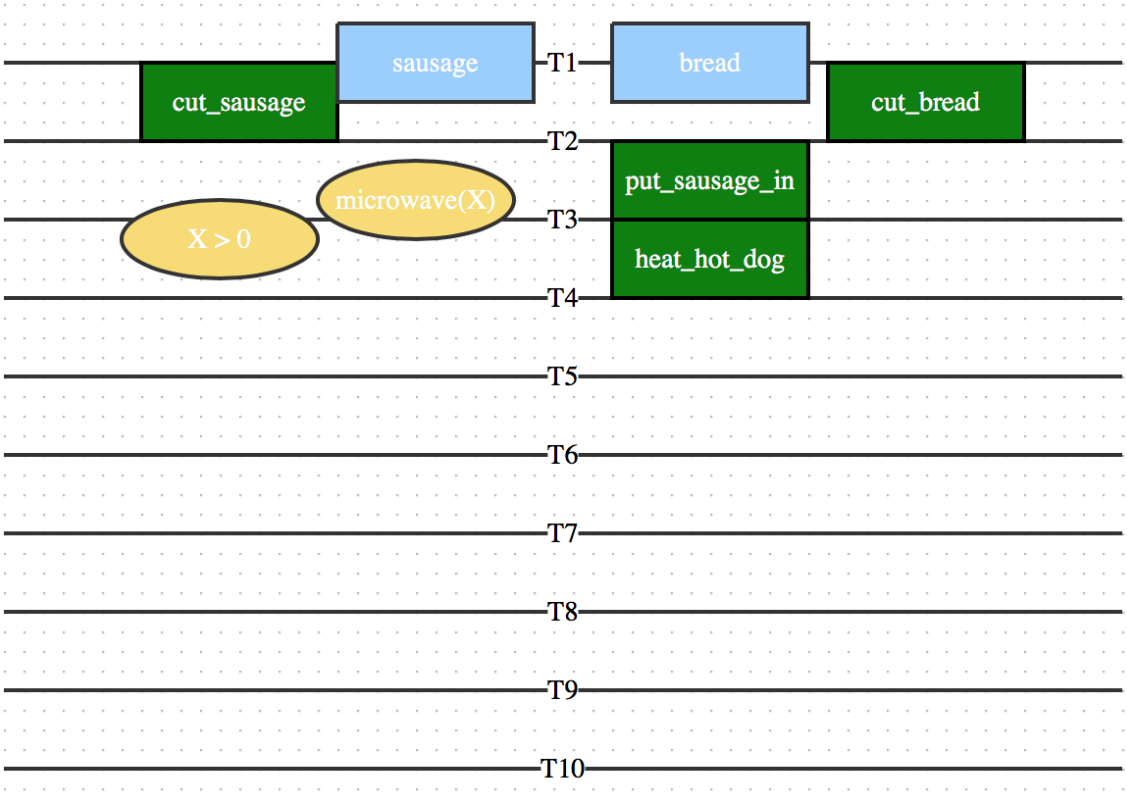

Figure 3.2: Macro Action Graph

In Macro Action Graph, the green rectangles represent atomic actions or macro actions, the blue rectangles fluents, and yellow ellipses timeless facts or conditions. The higher the cell locates, the earlier it happens in the macro action. In addition, the location of fluent represent the time the fluent shall be valid. For instance, the two blue rectangles means:

```
bread at T1
sausage at T1
```

Macro Action Graph is saved with a name, and the name will be the predicates for the macro action. So if we save the graph with name make_hot_dog, the LPS clause for the graph will be:

```
make_hot_dog from T1 to T4 if
    bread at T1,
    sausage at T1,
    cut_bread from T1 to T2,
    cut_sausage from T1 to T2,
    put_sausage_in from T2 to T3,
    microwave(X),
    X > 0,
    heat_hot_dog from T3 to T4.
```

3.3.3 Reactive Rule Graph

In Reactive Rule Graph, same as macro action, the green rectangles represent atomic actions or macro actions, the blue rectangles fluents, and yellow ellipses timeless facts or conditions. The higher the cell locates, the earlier it happens in the reactive rule.

The unique things are as follows: each cell is distinguished by its font-colour, the white font colour represents that the cell is an antecedent, and the red font colour represents that the cell is a consequent. The reactive rule graph must contain at least one antecedent and at least one consequent. Due to causality, consequents should not appear before antecedents.

In addition, since fluents cannot be modified directly and can only be modified in causal theory, they cannot appear as part of the conclusion.

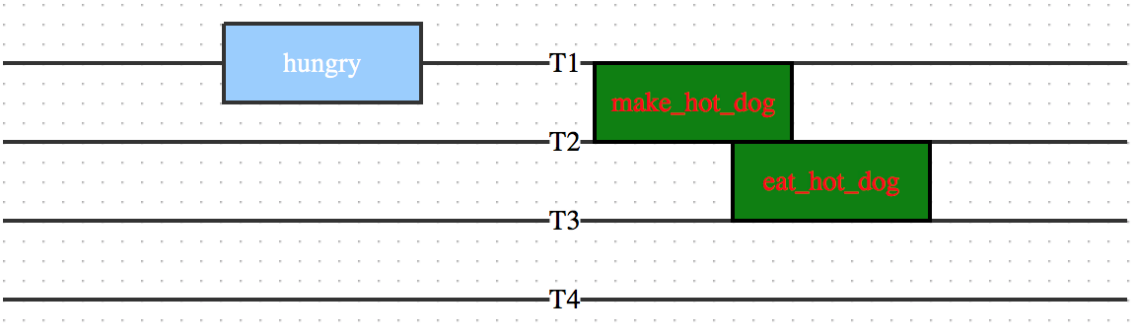

Figure 3.3: Reactive Rule Graph

For instance, the above graph means:

```
if hungry at T1
then make_hot_dog from T1 to T2, eat_hot_dog from T2 to T3.
```

Chapter 4

Product

4.1 Overview

The workflow interface easyLPS is implemented to be a web page. EasyLPS is currently deployed on https://easylpsazure.azurewebsites.net website. The web page consists of four parts: the menu bar on top of the page, the workflow graph on the left half, the code area at right top, and the clause panel at right bottom of the page.

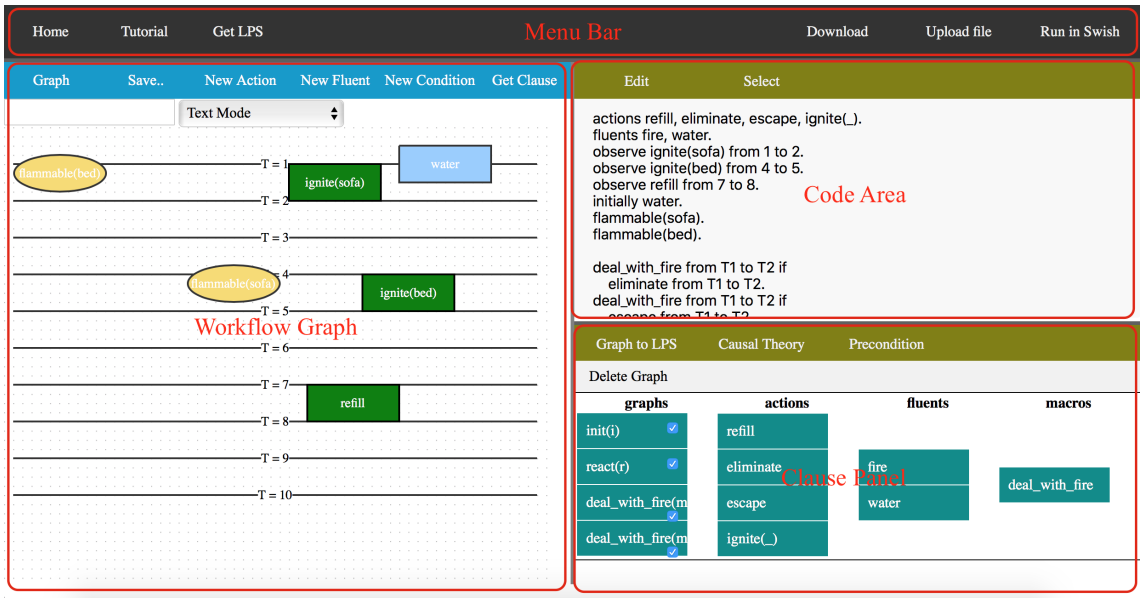

Figure 4.1: Page

The key steps of creating a complete LPS code can be summarized as follows.

1. Create several graphs in the workflow graph to describe the initialization states, observations, causal theories, and macro actions, and save them respectively to graph buttons in the ”graph to LPS” section in the clause panel.

2. Select the graph buttons in the ”graph to LPS” section, click the ”to LPS” button in the menu bar, and check the generated code in the code area as well

as the format of actions, fluents and macro actions generated in the table of ”graph to LPS” section. If error is found or modification is needed, click the corresponding graph button, modify the graph and save it again, and repeat the step until satisfied with the result.

3. Input the contents of causal theories and preconditions in the clause panel with the help of select menu with information about actions, fluents, and conditions coming from the graphs, and generate the clauses.

4. Select the generated graph buttons, causal theories, and preconditions, and click the ”to LPS” button the generate the final LPS code.

5. Click the Download button in the menu bar to save all selected graphs, causal theories, and preconditions as a json string in a text file to the computer if needed. If wanting to load a saved file on the page, click the ”Upload File” button to upload the file.

6. Finally, click the ”Run in SWISH” button to open SWISH web page in a new tab, and copy the generated LPS code to run in SWISH.

Each function in easyLPS will be explained in detail in the following section. The complete user guide is in Appendix C.

4.2 Workflow Graph

One of the most significant and essential part of the project is the workflow graph. Workflow graph locates in the left half of the page, on top of which locates a drop-down menu. The graph itself is implemented with jointjs, a workflow javascript library. Three types of graphs are designed to be converted into LPS code: Initialization graph describe all the initial states of fluents and observations of actions, each ReactiveRule graph generates one reactive rule clause, and each Macroaction graph generates one macro action clause.

4.2.1 Create a new graph

To start with, users need to create a graph by hovering the mouse over graph button and clicking the type of the graph. An empty graph with time line from 1 to 10 will be shown on the left screen.

Figure 4.2: New graph

4.2.2 Add elements to the graph

Users are then able to add elements including actions, fluents, and conditions by clicking the ”New Action”, ”New Fluent”, and ”New Condition” button respectively. After elements is added to the top of the graph, users can drag them to the position they like, and elements will adjust automatically to the nearest correct position. Actions shall always locate between time lines, indicating actions observed or occurred from the upper-bound time to lower-bound time. Fluents shall always locate on the time lines, indicating that the fluent holds in the initialization graph or in other situation, that the fluent holds at the time on the time line.

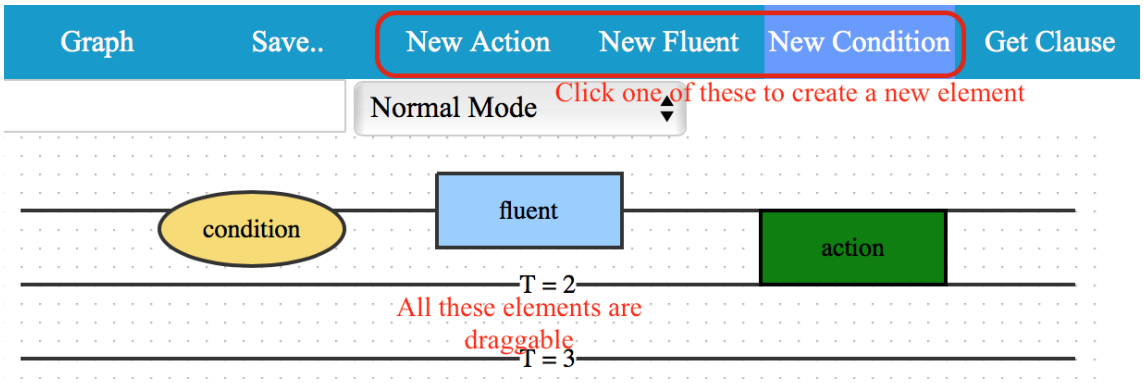

Figure 4.3: New element

4.2.3 Insert or modify the text in the element

Newly added elements will not be recognized by the graph, unless text is inserted or modified afterwards. In order to insert text into the elements, users need to select the ”Text Mode” and type or paste text into the input area. Only letters, numbers, and certain symbols are allowed in the input area.

When in Text Mode, right click the element in the graph to insert text. If the element is an action or a fluent, javascript will check whether the input text violates the corresponding syntaxes. If yes, an alert information will be shown in front of the screen and the insert operation will be invalid.

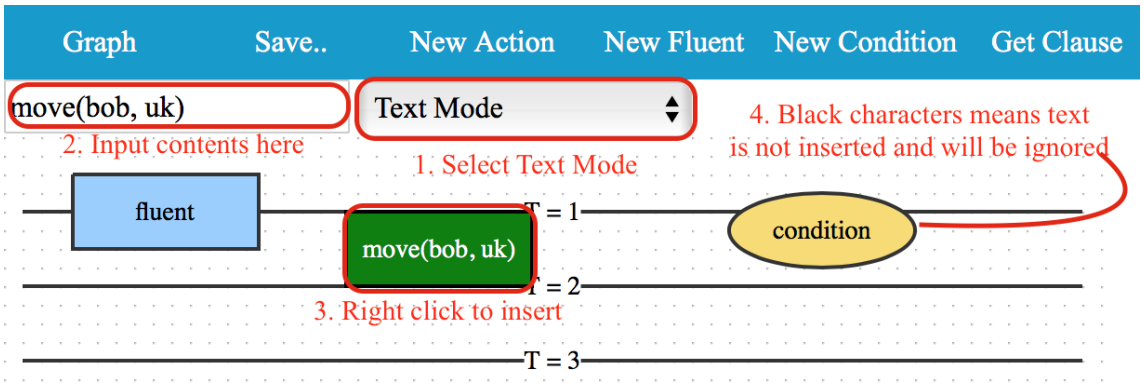

Figure 4.4: Insert Text

Otherwise, the input text will be successfully inserted into the element, and a button will be created in the dropdown menu. For example, after inserting text to an action, a button will be generated in the dropdown menu of ”New Action”, and clicking the button will create a new action at the bottom of the graph. The button is valid when creating other graph until refreshing the page or closing the whole web page session.

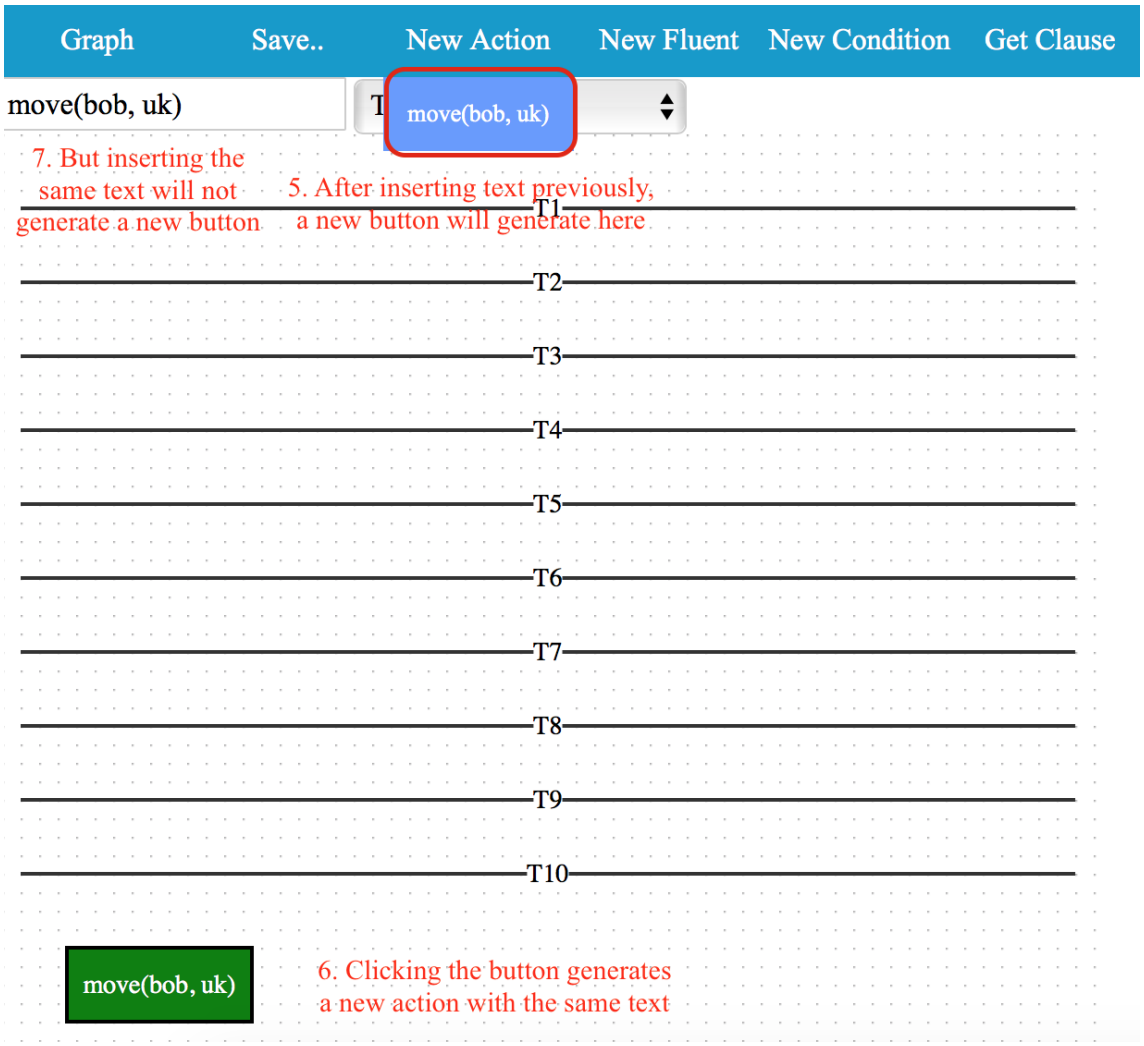

Figure 4.5: New Action Button

4.2.4 Save the graph

When users are satisfied with the graph, they can hover the mouse over ”Save..” menu and click the ”Save” button. A window will pop up for users to input the name for the graph. For Initialization and ReactiveRule graph, the graph name is only aimed for users to distinguish between different graphs, while for Macroaction graph the graph name is the predicate expression of the macro action, thus the system will check whether the name violates the action syntax. If yes, an error message will be printed and the save process is halted. If no, the button for the graph will be shown in the ”graph to LPS” section in the clause panel. The button for the graph is associated with a checkbox indicating whether the graph is selected to generate the final code.

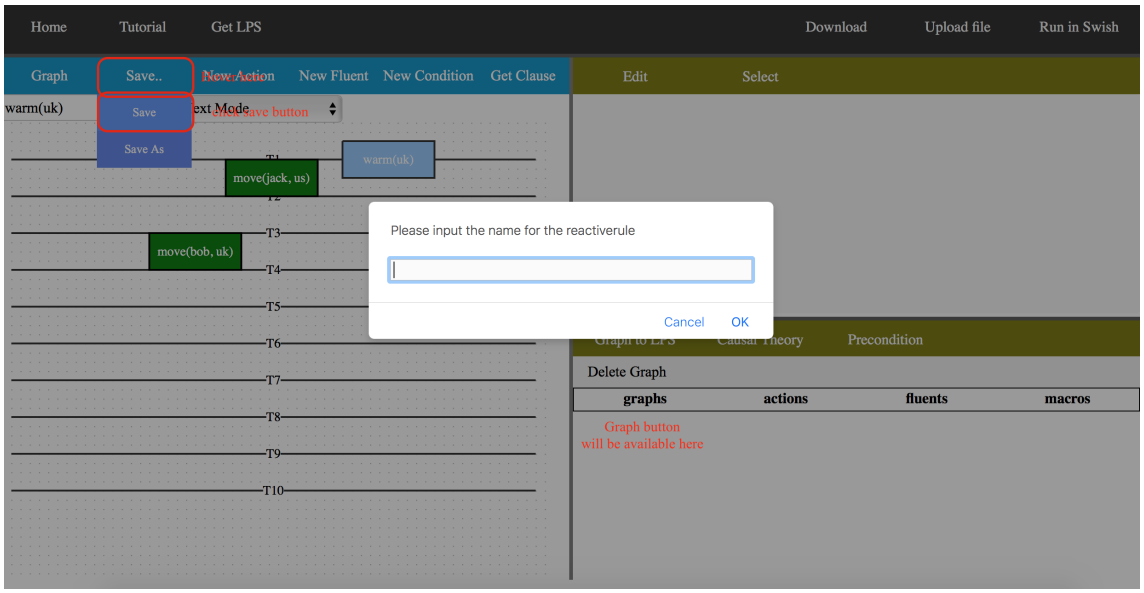

Figure 4.6: Save the graph

The letter in the bracket after the graph name indicates the type of the graph: 'i' stands for Initialization Graph, 'r' for Reactive Rule Graph, and 'm' for Macro Action Graph.

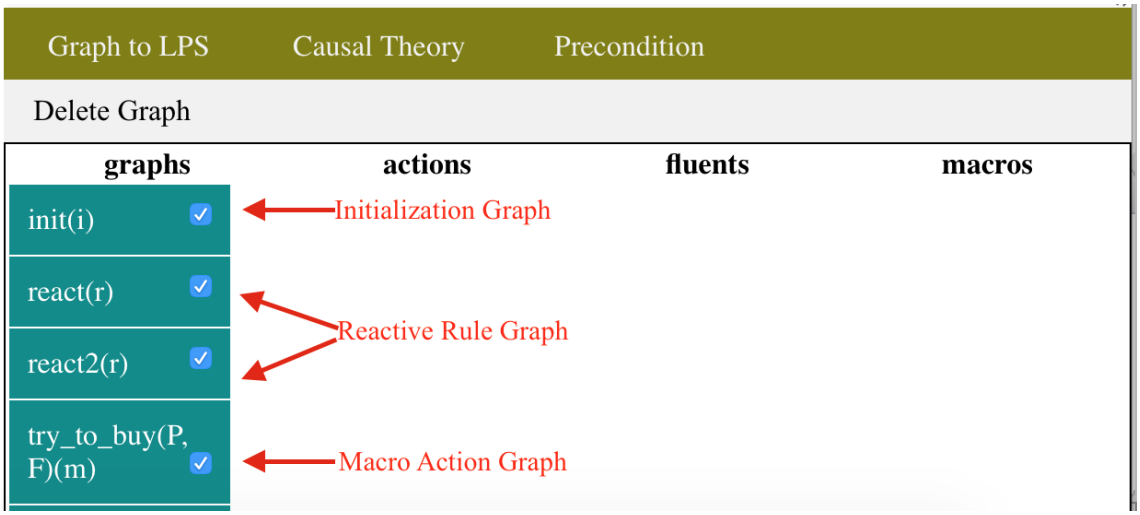

Figure 4.7: Graph symbol

After saving the graph, users can work on a new graph by repeating the first step "Create a new graph", and if they want to go back to the graph and do some modification, they can do it simply by clicking the button for the graph. After the modification, users can either click the "Save" button to save the graph to the original button, or click the "Save As" button to give the graph a new name and save the graph to a new graph button. In latter situation, instead of modifying the contents of the original graph, these contents are duplicated, modified, and saved.

4.2.5 Get clause from the graph

After the graph is saved, users can click the ”Get Clause” button to generate the corresponding LPS code to the graph currently appeared on the screen. If there is any error occurring during the process, the error message will be alerted and LPS code will not be generated.

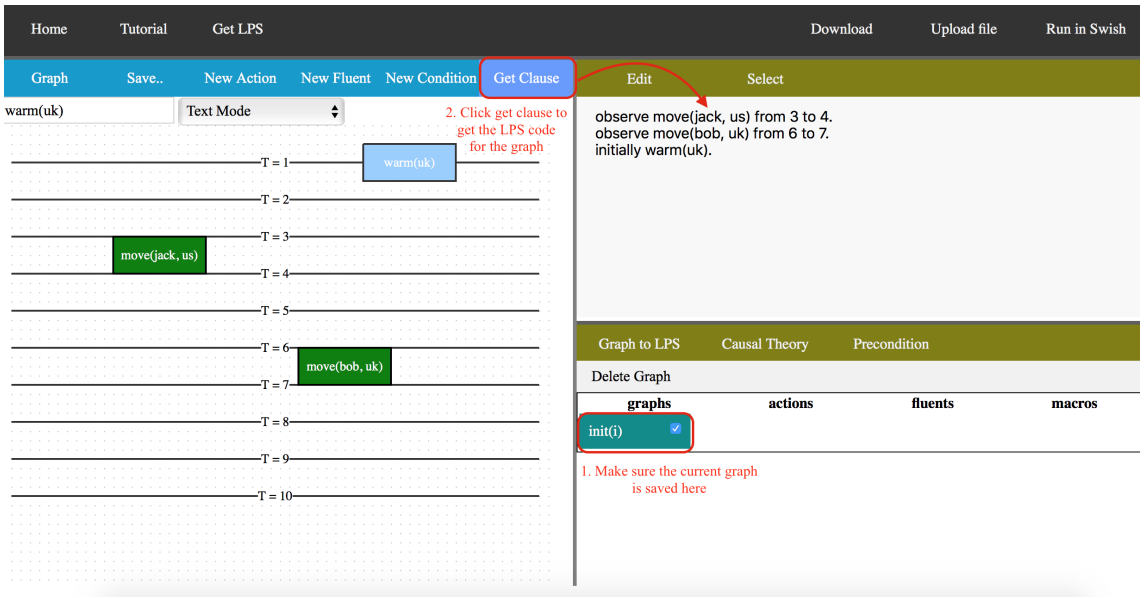

Figure 4.8: Get Clause

4.3 Clause Panel

4.3.1 Graph to LPS Panel

The Graph to LPS Panel holds all graph buttons and demonstrates the actions, fluents, and macro actions in a previous generated LPS code. After drawing all the graphs and before inputting the causal theories and preconditions, the graph buttons are demonstrated in the first column of the table with a checkbox, storing the graph json, graph type, graph name, and graph id. Clicking the graph button will show the graph it stores in the workflow graph. A ”Delete Graph” button is presented to delete all selected graph buttons, and users can also delete a single graph by right clicking the graph button.

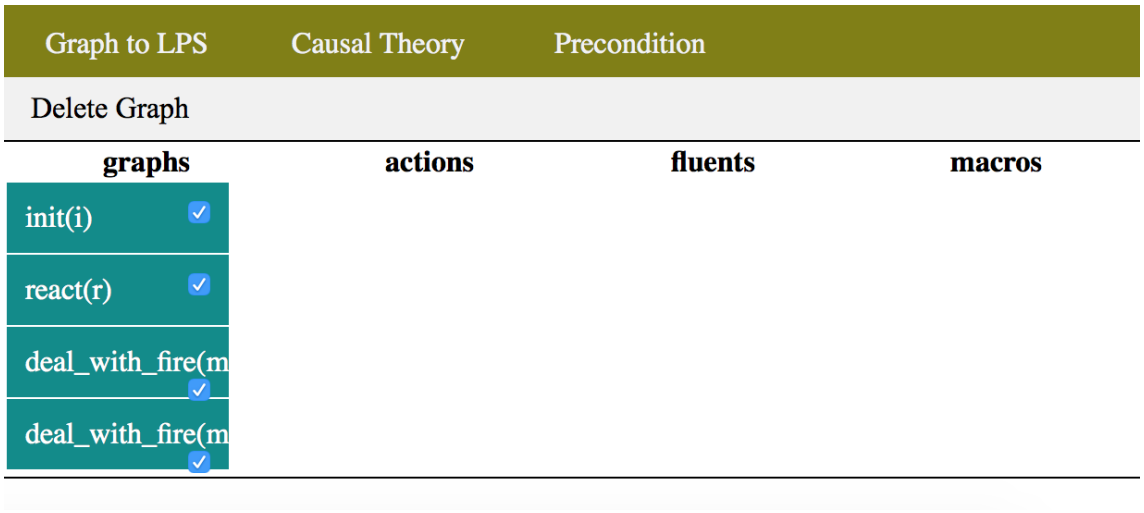

Figure 4.9: Graph to LPS

After drawing all the graphs, users can continue to process by clicking the ”to LPS” button in the menu bar. The required information about all selected graphs will be packed into a Json string and sent to back-end to process via Ajax. The returned data will also be a Json string containing not only the LPS code for all graphs but also the information about all actions, fluents, conditions, and macro action predicates which will be stored in a global variable called global.collection to ease the way to do causal theory and preconditions input afterwards. Part of those information will be shown in the table for users to check whether all actions presented is primitive. Macro action cannot appear in the causal theories and preconditions.

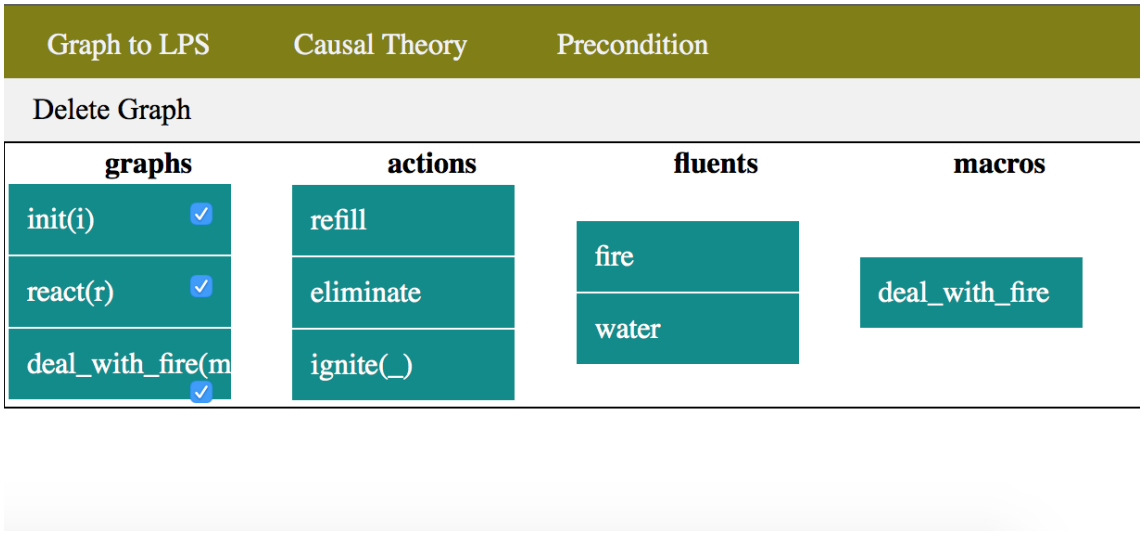

Figure 4.10: Graph to LPS

4.3.2 Causal Theory Panel

The causal theory panel is a user-friendly interface to generate the causal theories. After generating the LPS code from the graphs, all names of actions, fluents, and

conditions and the wild match of actions and fluents have been stored in a global variable global.collection in javascript, therefore, the select area can get the value of selection from the global variable.

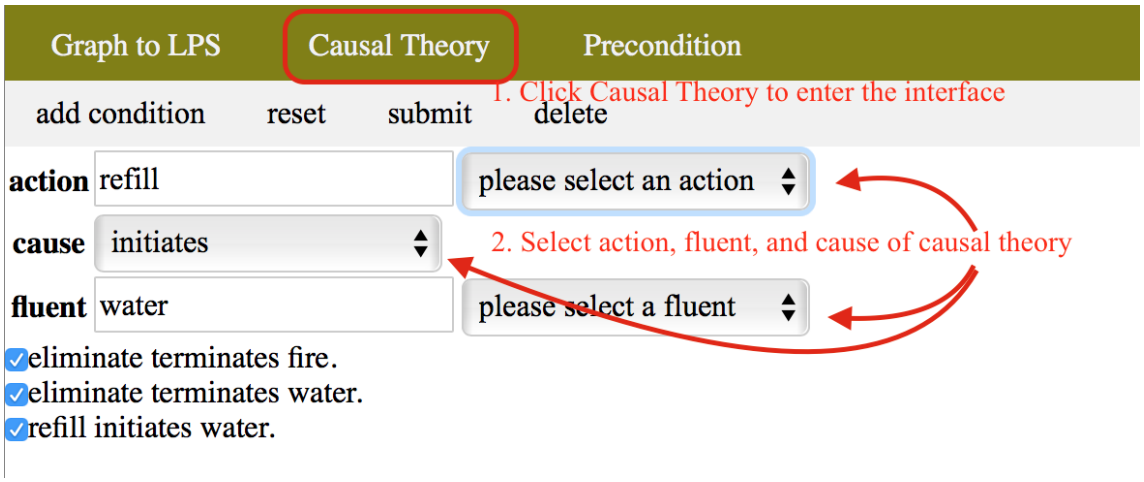

Figure 4.11: Causal Theory Panel

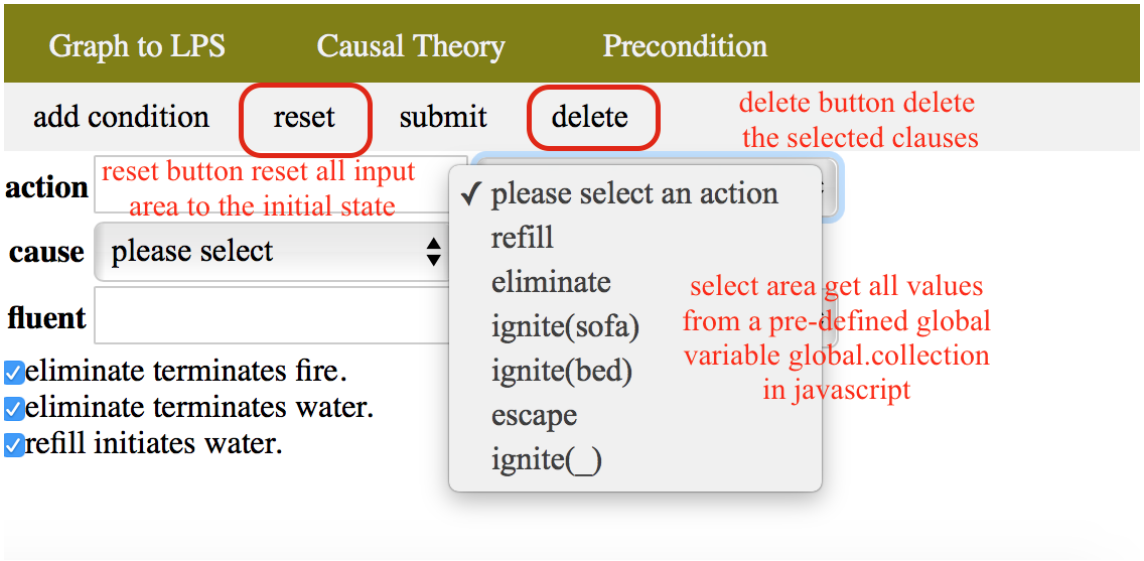

Figure 4.12: Causal Theory Panel

If users select ”updates” as the relation between action and fluent, a new line will be generated enabling users to input the before and after state of a variable in the fluent.

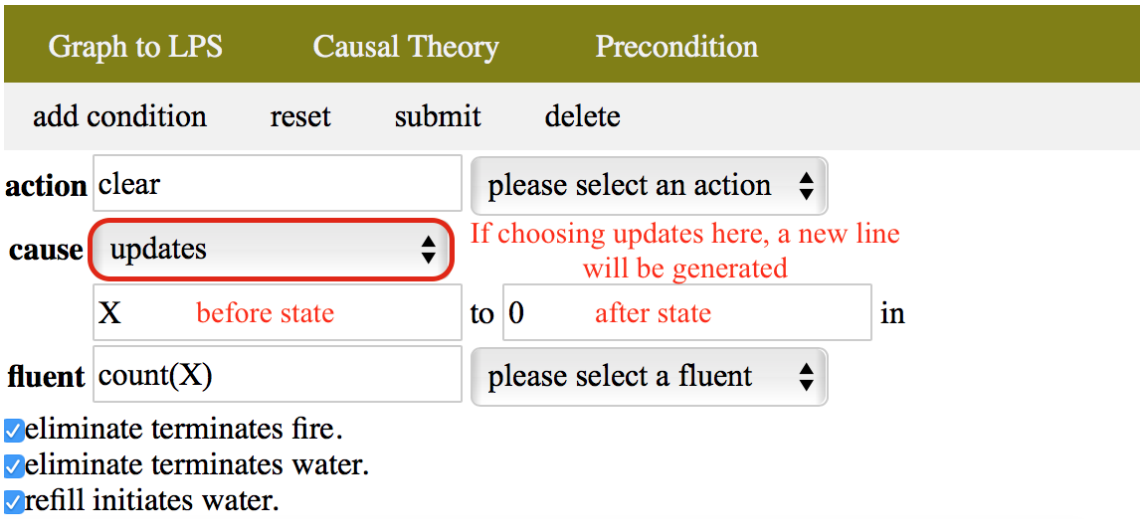

Figure 4.13: Causal Theory Panel

If the input value is not shown in the select area or users prefer to type in the value themselves, autocomplete function is also provided in the input area.

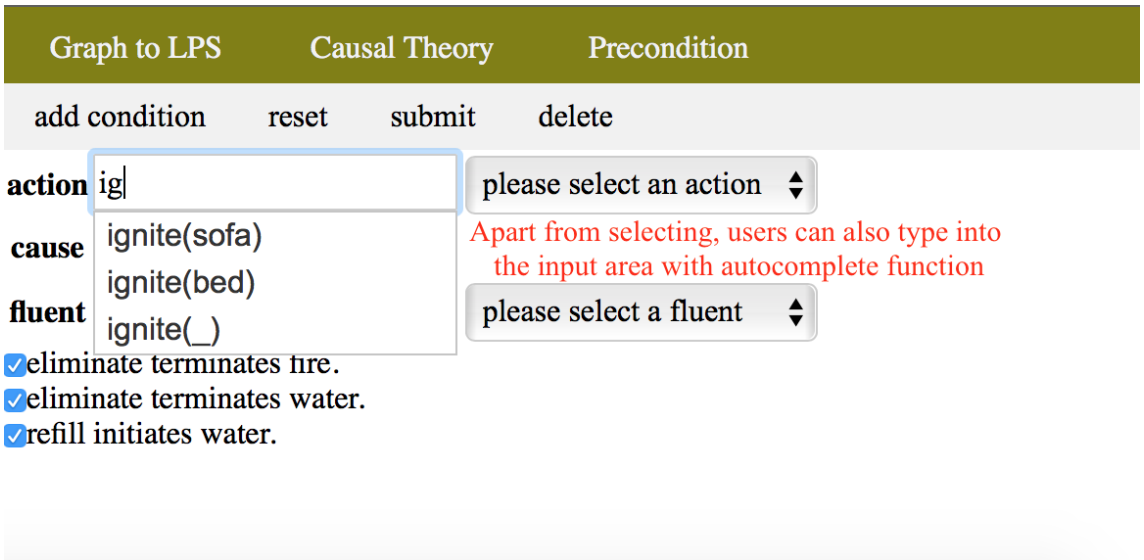

Figure 4.14: Causal Theory Panel

Some clausal theories come with one or more conditions, and uses can add those conditions by clicking the ”add condition” button in button bar and do the input afterwards. newly added conditions can be deleted by the delete button at the rear of each condition line.

After all the above process has been done, one causal theory can be generated by clicking the submit button in button bar. The reset button next to it help clear all the input before and set the whole area of input to the initial state, and the delete button in button bar delete all causal theory selected.

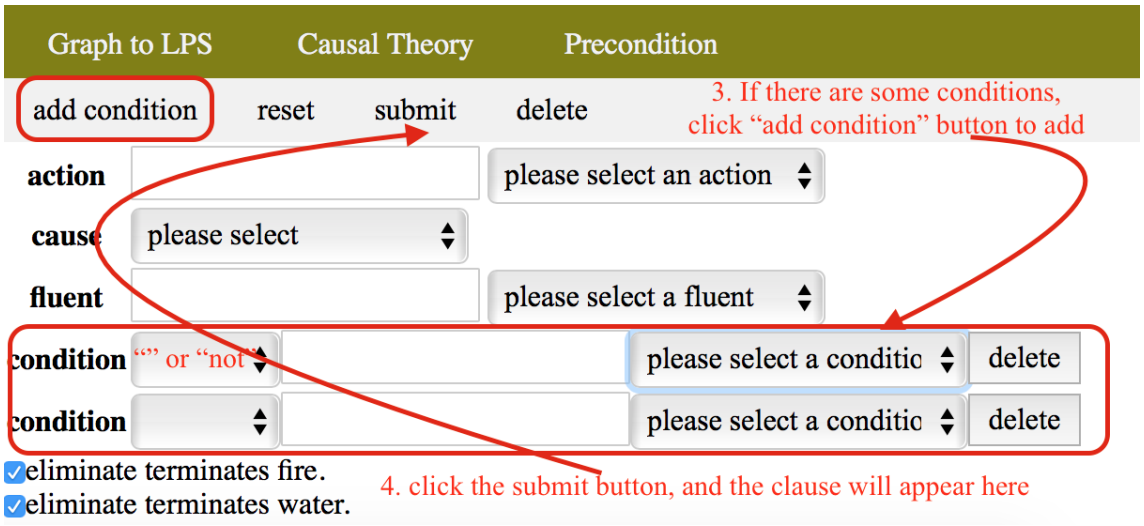

Figure 4.15: Causal Theory Panel

4.3.3 Precondition Panel

The precondition panel has similar structure to the causal theory panel. Users input actions, fluents, and conditions which cannot occur simultaneously. The preconditions will be generated according to the sequence of adding the predicates or condition clauses.

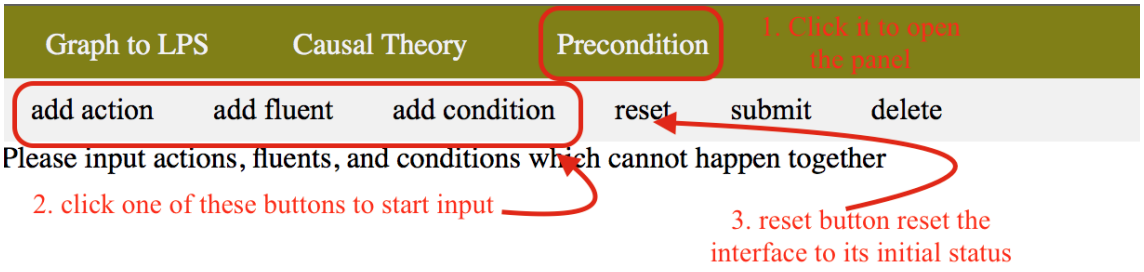

Figure 4.16: Precondition Panel

The select areas, input areas, and delete buttons have almost the same function as those in the Causal Theory Panel.

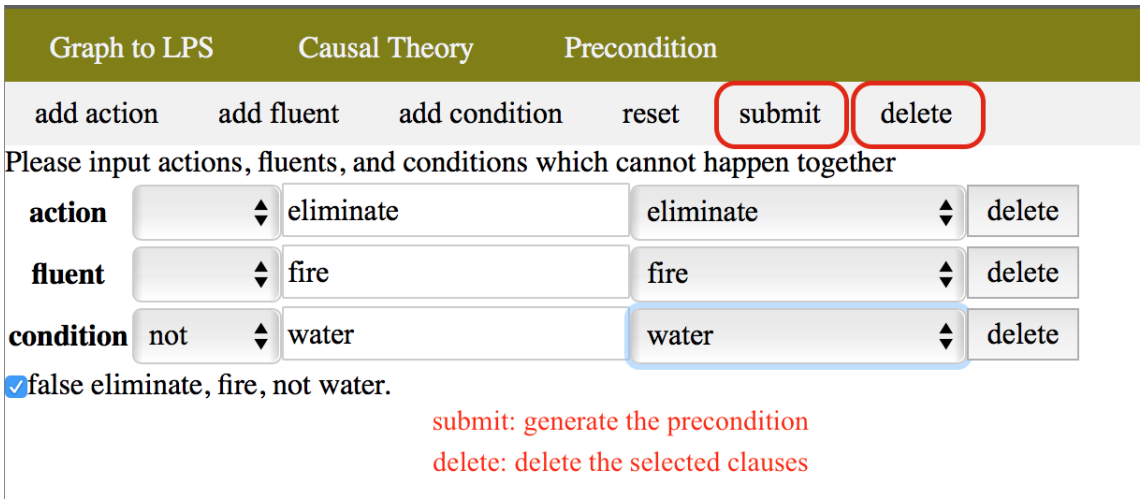

Figure 4.17: Precondition Panel

4.4 Code Area

The code area consists of two panels: one for edition, the other for selection. The edit panel is the default panel shown on the page which is designed for users to do some modification to the final code directly, but the modification cannot directly change the graphs, reactive rule clauses, and precondition clauses. If the user click the ”to LPS” or ”Get Clause” button again, the changes to the code will be lost.
If users want to do modification permanently, they can apparently find the corresponding graphs or clauses to change the contents, but to ease their way, the select panel is provided to make the search more promptly. Users can switch the panel simply by clicking the select button on the button bar, and the same code as the edit panel will be demonstrated in the code area. The code can be divided into the following parts:

1. Header. The header part declares the wild match of actions and fluents occurred in graphs and clauses. This part is generated automatically and therefore, does not have a counterpart.

2. Code from graphs. Initialization graphs can be converted to several clauses about initial states and observations, and other graphs are converted to a single clause. When double clicking those clauses, the corresponding graph will be demonstrated on the workflow graph region and are ready to be modified.

3. Causal theories. Causal Theories comes from all selected clauses in the ”Causal Theory” section of the Clause Panel, and double click the causal theory sentences will show the interface of ”Causal Theory” section in Clause Panel.

4. Preconditions. Same as causal theories, double click the precondition clauses show the interface of ”Precondition” section in Clause Panel.

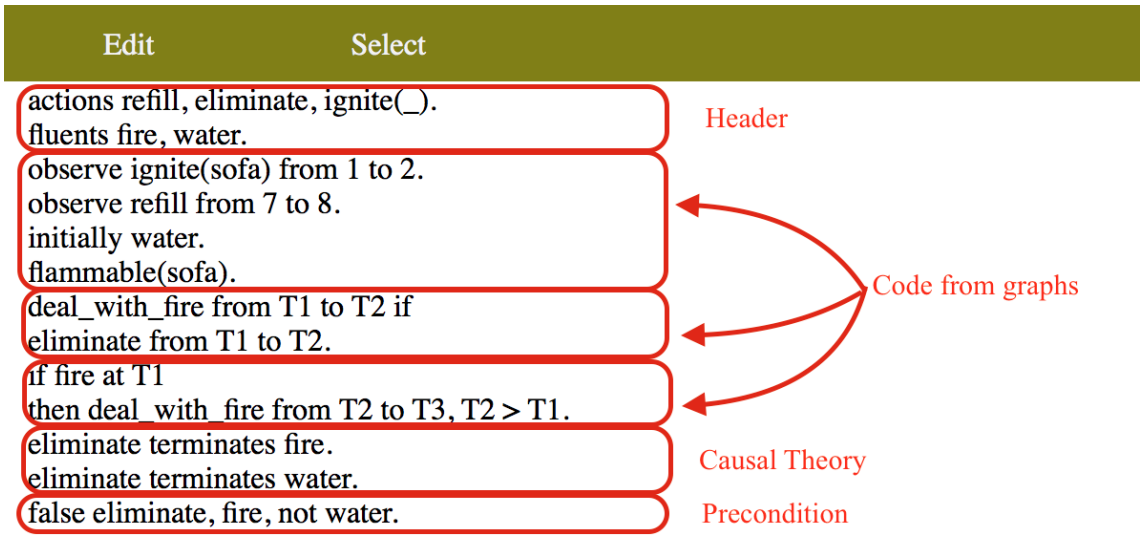

Figure 4.18: Text Area

4.5 Menu Bar

4.5.1 Home Button

The Home button locating on left top of the page refresh the web page.

4.5.2 Tutorial

Tutorial button on left top of the page open the github repository of this project in a new tab. The tutorial is in the README file demonstrated at the bottom of the github web page.

4.5.3 To LPS

”to LPS” button, as described before, translate the information in the web page into a Json string, and send to application server in the back-end. After back-end processing of the Json string, the final LPS code is shown in the Code Area, wild match of atomic actions, fluents, and macro actions is demonstrated in the ”Graph to LPS” panel, and the contents of all select area and the autocomplete contents of all input area in the Clause Panel are updated.

4.5.4 Upload and Download

Upload and Download functions are purely implemented by front-end javascript. The most significant part of the function is to translate the whole page from and to a Json string.

Download button write the Json string into a txt file and enable users to name the file with letters, numbers, and underscore. Upload button in the menu bar allow users to upload only the txt files, and convert the Json string in the file into the graph buttons, causal theory clauses, and precondition clauses. After the loading, users can regenerate the LPS code by clicking the ”to LPS” button. If the process of ciphering and loading the txt file throw an error, an error message will be alerted.

4.5.5 Run in SWISH

The Run in SWISH button locating on right top of the page open the SWISH web page in a new tab of the browser. Users can copy and paste the LPS code in the code area and to get the final result of the LPS code.

Figure 4.19: Menu Bar

Chapter 5

Implementation

In this chapter, we mainly discuss about how the information in the front-end web page is packed, how the packed information is sent to back-end and processed, and how the processed information is sent back and parsed to be demonstrated.

In the Front-End section we discuss the implementation about how the page is divided, how to do input restriction, and some details about the workflow graph. The other part of the page is implemented by generic javascript and some fundamental html and CSS, and therefore those codes are self-explained.

The next three sections describe the core function implementation of the web page. All useful information on the page can be converted to and from a page Json, and the stringified Json object is the content of download and upload files. In the ”to LPS” function, the page Json string will be sent to a servlet via Ajax, and the return data is also packed into a Json string. This process will be discussed in detail in section 5.4 and 5.5. ”Get Clause” function can be considered as the simplified ”to LPS” function, and will be discussed in section 5.7.

The web page involves a large amount of input, and it is inevitable that users may make mistakes. Thus, section 5.6 is about error handling.

5.1 Front-End Overview

5.1.1 Page Divider

The Front End web page is divided into four parts. It is implemented by pure CSS. The upper 60 pixels is for menu bar; the left half and the right half are equally divided according to the size of window. the right half is divided into two regions evenly, Code Area and Clause Panel.

5.1.2 Input Restriction

In LPS, only letters, numbers, and certain symbol characters are valid in the program, therefore, to make the system more user-friendly and avoid vicious input doing damage to the web server, we adopt the following regular expression to do the input restriction in front-end javascript.

```
obj.value.replace(/[^\a-\z\A-\Z0-9\(\)\,\_\ \@\<\>\=\+\-\*\/\\]/g,'');
```

In addtion, for inputs of actions and fluents, extra checks are conducted in the front-end javascript to verify whether they match the following regular expression.

```
/^(\s)*[a-z](\w)*(\s)*(\((\s)*(\w)+(\s)*(,(\s)*(\w)+(\s)*)*\))?$/
```

5.1.3 Workflow Graph

Workflow Graph is implemented by the javascript library JointJS. The Workflow Graph is actualized by creating a joint.dia.Paper on a div element in the Workflow Graph region to hold a new joint.dia.Graph. By default setting elements in the graph are draggable. Ten joint.shapes.standard.Link object is created to be the time lines with interval of 40 pixels in each graph, and the attribute ”interactive/linkMove” is set to be false to precent those lines from moving.

The functionality of creating elements in the graph is implemented by creating a shape in joint.shapes.standard, and set their attributes to be default. The javascript function paper.on('cell:contextmenu', function(cellView)) enables users to change the content of certain elements or delete certain elements in the graph according to the mode in the select area by right clicking the element, and the javascipt function paper.on('element:pointerup', function(elementView, evt, x, y)) automatically adjust the positions of elements to the nearest reasonable places by directly modifying the value of y.

JointJS library also provides two functions to convert the graph from and to Json objects: graph.toJson() and graph.fromJson(JsonObject). With the help of these two functions, we save the graph to a button by setting the jsonstring attribute of the button as the stringified graph.toJson(), and load the graph when clicking the button by extracting and parsing the Json string from jsonstring attribute and passing it as parameter to graph.fromJson().

5.2 Back-End Architecture

The Java Back-End consists of four packages:

1. servlet package. Servlet package has two classes: LpsServlet accept the Ajax request coming from ”to LPS” button, create a Page object, and response with a Json string; ClauseServlet accept the Ajax request coming from ”Get Clause” button, create a Graph object, and response with a string of LPS code translated from the graph.

2. domain package. Domain package holds all entity classes designed to store useful information from the front end.

3. exception package. Exception package is essential in back-end error handling. When an exception is generated in a class, it will generate a error message and call the constructor of the corresponding exception, and the exception will eventually be thrown to the servlet.

4. utils package. Utils package consists of classes with all static methods to be reused in the back-end application.

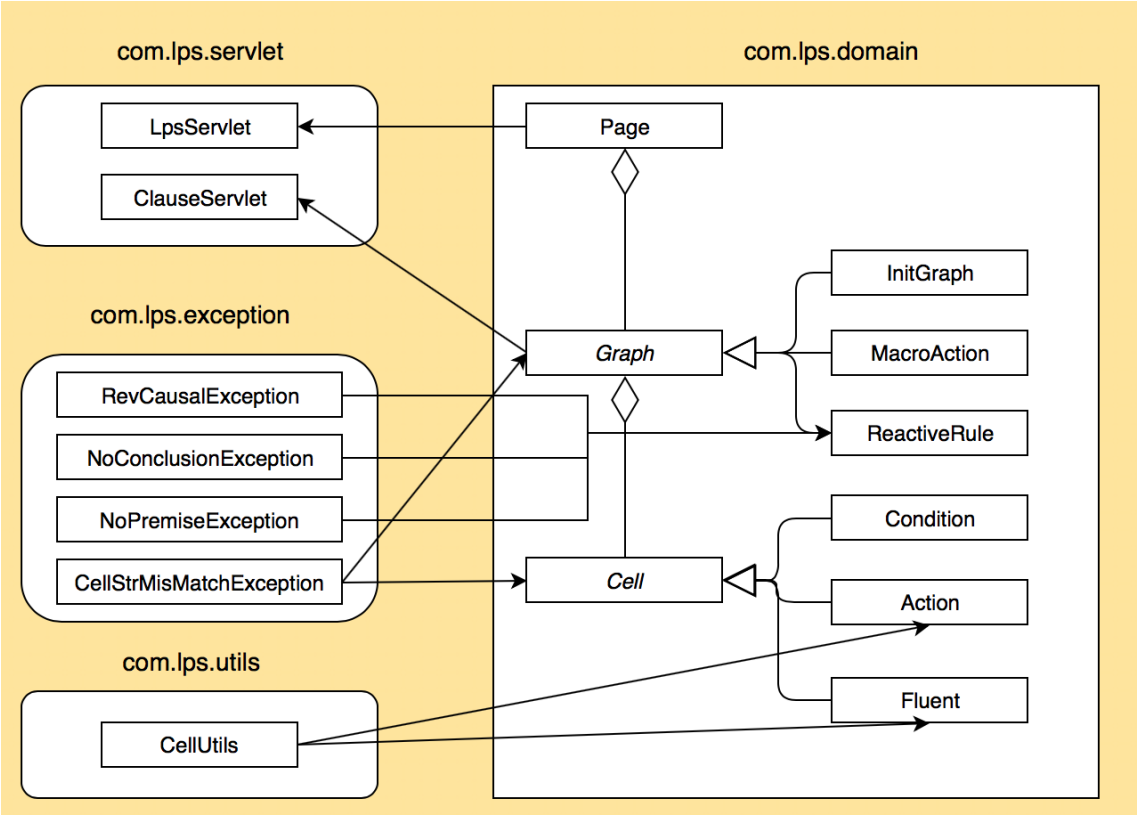

Figure 5.1: Back-End Architecture

5.3 Page Json

To pack the information about the whole web page, we need to collect all information coming from graphs, causal theories, and preconditions. Therefore, by divide strategy, we collect them separately from the clause panel.

```
var pageJson = {
    graphJson: getGraphJson(),
    causalJson: getCausalJson(),
    precondJson: getprecondJson()
}
```

5.3.1 Graph Json

Given by JointJS javascript library, the graph is implemented with a toJson() method to convert the graph into a json object. Below is a typical json object for a rectangle.

```
var element = {
    type: standard.Rectangle,
    // Rectangle-action, Polyline-fluent, and Ellipse-condition
    position: {x: 390,y: 160},
    // the position of element in pixel
    size: {width:100,height:40},
    angle: 0,
    id: "4f0d5b88-a6bd-4676-aaa0-f1cac63f376a",
    z: 14,
    attrs: {body: {fill: green},label: {fill: white, text: ignite(bed)}}
    /* body:fill green-action, blue-fluent, yellow-conditions
       label:fill black-text not inserted, white-text inserted,
            red-is part of conclusion */
}
```

The type, position, and attributes of each element in the graph will then be extracted for processing in the Java back-end.

During the saving process, a unique id is generated for each graph by concating the system time in milliseconds with a random three digit number, and a new LI element will be created in the left most table cell in ”Graph to LPS” panel, and the json string, graph type, name and id of the graph will be stored as attribute in the A element appended to the LI element.

```
<ul id="subgraphs">
    <li> /* newly created */
        <a>
         <span /><checkbox />
        </a>
    </li>
</ul>
```

When collecting the graph json, javascript will go through each graph button(the a element above) with a checked checkbox and form the subGraphJson below.

```
var subGraphJson = {
    jsonStr: button.getAttribute("jsonstring"),
    graphType: button.getAttribute("graphtype"),
    graphName: button.getAttribute("graphname"),
    graphId: button.getAttribute("graphid")
}
```

Those subGraphJson will then be added to the lists in graphJson according to the graph type.

```
var graphJson = {
    initGraph : [],
    reactGraph : [],
    macroGraph : []
}
```

5.3.2 Causal Theory Json

We get the data of causal theories from the causal theory panel. To demonstrate and hold each causal theory after users click the submit button, we append a new child to the div element at the bottom of the panel.

```
<div id="causalStr">
    <p> /* newly created */
        <checkbox /><span />
    </p>
</div>
```

On creating the new p element, we standardize the text in the text input area of actions, fluents, and possibly conditions and store them as the attributes of the span element which is the second child of the p element. Conditions are stored in an array and therefore need to be converted into json string to be stored in the attribute of the span element. The innerHTML of span element is the text of causal theory.

```
var causalJson = {
    text: span.innerHTML,
    action: span.getAttribute("action"),
    fluent: span.getAttribute("fluent"),
    condition: JSON.parse(span.getAttribute("condition"))
}
```

The above javascript shows how we pack the json object of a single causal theory. For each causal theory with the checkbox in front of it checked, we add the above causalJson to an array to get the final json object for the whole causal theory panel.

5.3.3 Precondition Json

The generation process of precondition json is very much similar to that of causal theory json.

```
<div id="preCondStr">
    <p> /* newly created */
        <checkbox /><span />
    </p>
</div>
```

The only difference from causal theory json is that the number of actions, fluents, and conditions in preconditions are all ranged from 0 to n, respectively, so the text of these elements are all stored in json arrays. Those arrays are stringified and stored in the attributes of the span elements.

```
var precondJson = {
    text: span.innerHTML,
    action: JSON.parse(span.getAttribute("action")),
    fluent: JSON.parse(span.getAttribute("fluent")),
    condition: JSON.parse(span.getAttribute("condition"))
}
```

Same as causal theory json, the precondition json is packed in a json array to get the final json object for the whole precondition panel.

5.4 Back-End Processing of Page Json

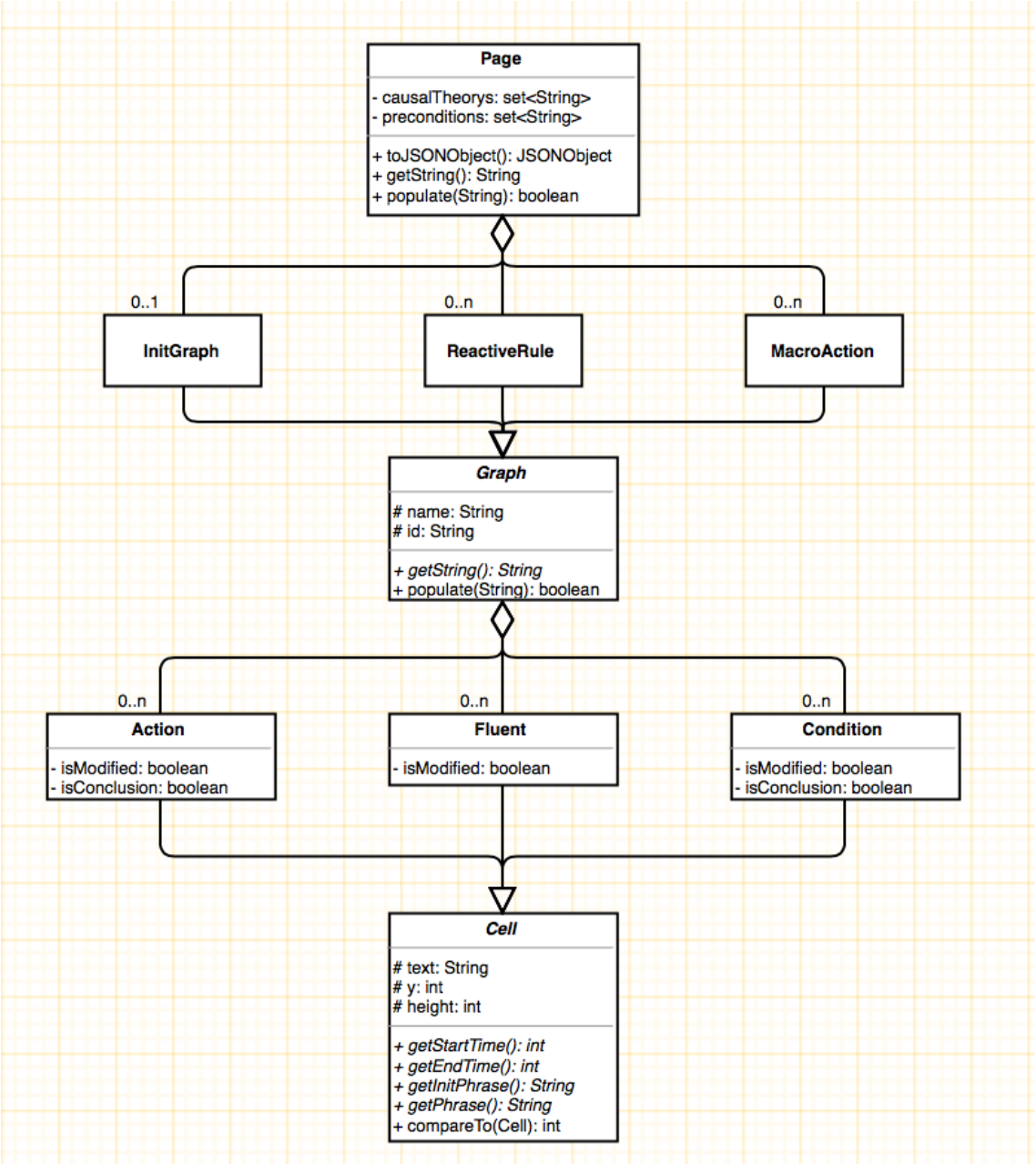

Figure 5.2: UML Diagram

The Java Back-End mainly aims to process the Graph Json string and Page Json string sent via Ajax in ”Get Clause” and ”to LPS” functions. The result data will also be a Json Object sent back via Ajax. A domain package is designed to hold all information in the json object.

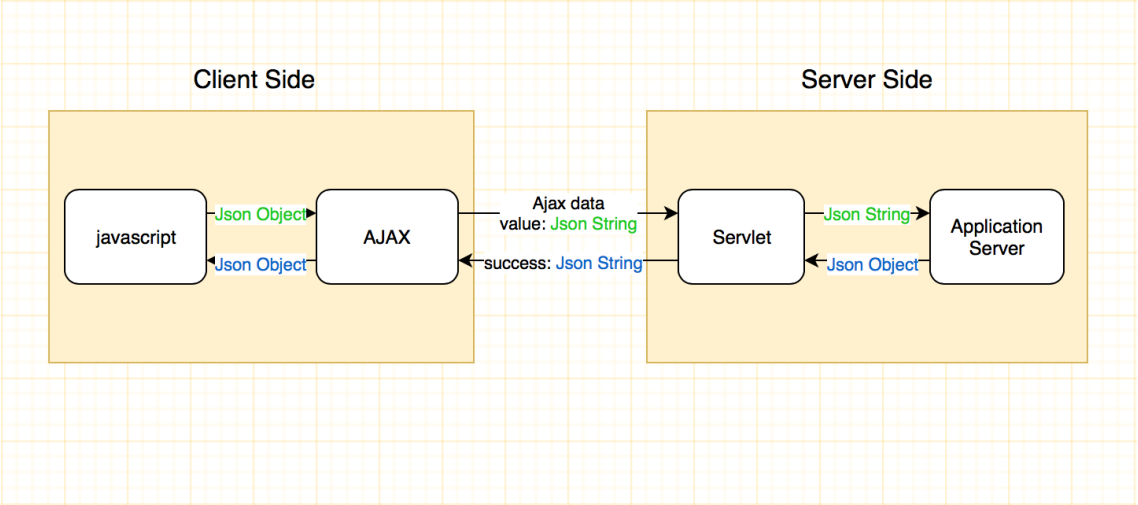

Figure 5.3: Ajax

As mentioned in section 5.1, json object about all useful information in the web page is packed in the front end, and stringified in Ajax to send to the servlet in the back-end to process. After receiving the Ajax request, the servlet creates and initiates a Page object and pass the Json String in as the parameter.

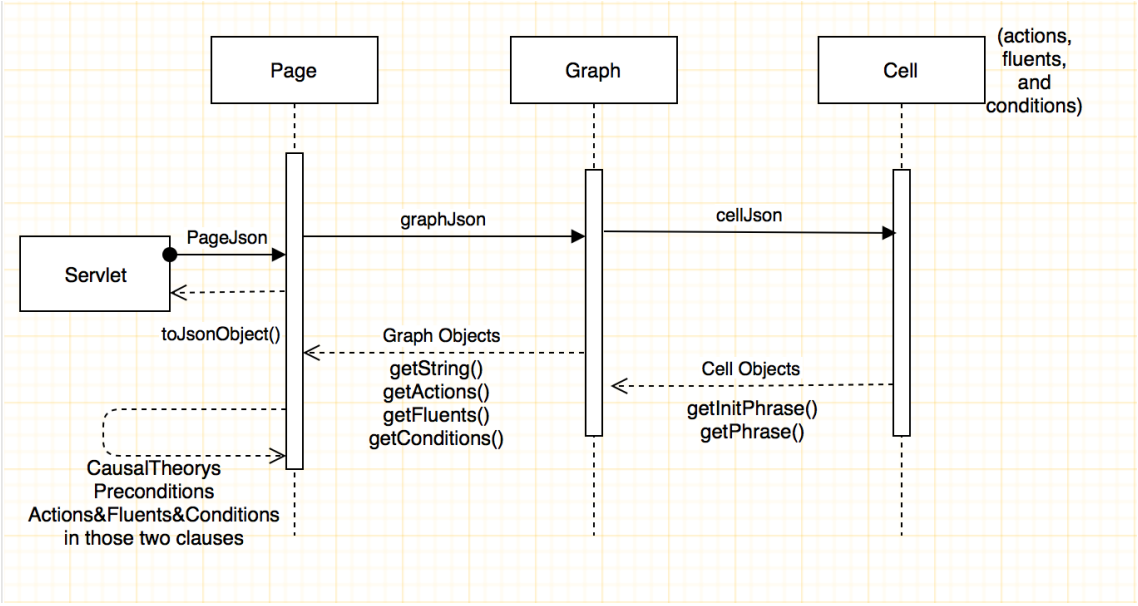

Figure 5.4: Message Path

In the Page Object, the json string is parsed and divided into several parts, the graph json part is used to create and initiate Graph objects, and the other part is used to initiate the text of the causal theories and preconditions and the actions, fluents, and conditions appeared in them. The Graph objects, after receiving the graph json, create and initiate the Cell objects according to the json about elements. Therefore, all useful information has then been stored in all domain objects.

As is depicted in Figure 5.4, the Cell class provides getInitPhrase() and getPhrase() method to translate the object into LPS code in the Initialization Graph and other kinds of graphs, respectively. The Graph class provides getString() method to translate the Graph object into LPS clauses with the help of getInitPhrase() and getPhrase() method. Most importantly, the Page object, after created and initiated, provide a public toJsonObject() method to the servlet, using the returned JsonObject as the final output data which will be sent back via Ajax to the front end.
The jsonObject is of the following form:

```
var jsonObject = {
    header: String, //getHeader()
    graph: jsonArray, //elements: {id: String, text: String}
    causalTheory: String, //getCausalTheory()
    precondition: String, //getPrecondition()
    code: String, //getString()
    actions: jsonArray,
    fluents: jsonArray,
    macros: jsonArray,
    conditions: jsonArray,
    actionsWild: jsonArray,
    fluentsWild: jsonArray,
    macrosWild: jsonArray,
}
```

The Page class provides getHeader(), getCausalTheory(), getPrecondition() methods to return each part of the final code separately, and getString() method to return the final LPS code. Information about graphs is packed in the graph element of the jsonObject. Four set of String is created and initiated in the class to store the predicates of actions, fluents, and conditions collecting from each graph stored in the Page object, and the names of MacroGraphs. The actionsWild, fluentsWild and macrosWild elements in the jsonObject derive from the wild match of action, fluent and macro String set, which is the results of converting the variable of predicates into underscores.

5.5 Front-End interaction with processed json

If the Page Json is successfully processed in the back-end, the Json string will be sent back and the following operation will be performed in the front-end.

The result Json string is parsed to reduct the jsonObject created by the toJSONObject() function of Page class in the back-end.

The code element obviously updates the value of the ”Edit” panel of Code Area.

The header, graph, causalTheory, and precondition element help recreate the ”Select” panel of code area, enabling users to double click the clauses to show the

corresponding graph, ”Causal Theory” panel , or ”Precondition” panel. This is implemented by the ondblclick() function binded to the newly created paragraph element.

Actions, fluents, macros, conditions, actionsWild, fluentsWild, and macrosWild are stored in the global variable global.collection, which will then be used to demonstrate to users in the ”Graph to LPS” panel, and to help users with the input of causal theory and preconditions if they have not done so or want to do some modification because the select area and autocomplete function of text input area will consult the global.collection for select choices and autocomplete contents.

5.6 Error Handling

Four types of error may be generated in the back-end process, so we implement four classes all extending from java.lang.Exception and pack them into the exception package. If any exception occurs during the process, an error message will be printed on the screen and the remaining process will be terminated.

5.6.1 Cell String Mismatch Exception

For all actions and fluents occurred in a graph, we have a process to check whether their predicate expression match with a regular expression. If not, a new CellStrMismatchException will be thrown to the Servlet. This exception is not likely to occur if users do the input in the front-end because front-end involves some input restrictions and also do regular expression matching. However, if users upload a txt file and try to process it to get the LPS code, this kind of checking is by all means necessary.

5.6.2 No Premise Exception

All Reactive Rules and Macro Actions should have at least one precedent. If not, the corresponding clause cannot be generated, and a new NoPremiseException will be thrown to the Servlet.

5.6.3 No Conclusion Exception

All Reactive Rules should have at least one consequent. If not, the corresponding reactive rule cannot be generated, and a new NoConclusionException will be thrown to the Servlet.

5.6.4 Reverse Causal Exception

For each reactive rule, all precedents shall happen earlier than all consequents. To judge the sequence of elements, each Cell has a getStartTime() function to calculate the start time of the Cell object and Cell class implements the compareTo() function

to enable Cell comparison. If the above rule is violated, a new RevCausalException will be thrown to the Servlet.

5.6.5 Error Demonstration

The Servlet implements a try-catch clause to catch the above three exceptions during the processing. If any of the exception is caught, the status of response will be set to 400 and the error message in the caught exception will be sent back to the Ajax function in the front-end and printed in front of the screen.

5.7 Get Clause

The functionality of Get Clause enable users to check the clause immediately after saving the graph, and the implementation of the function can be viewed as a simplified version of ”to LPS” process.

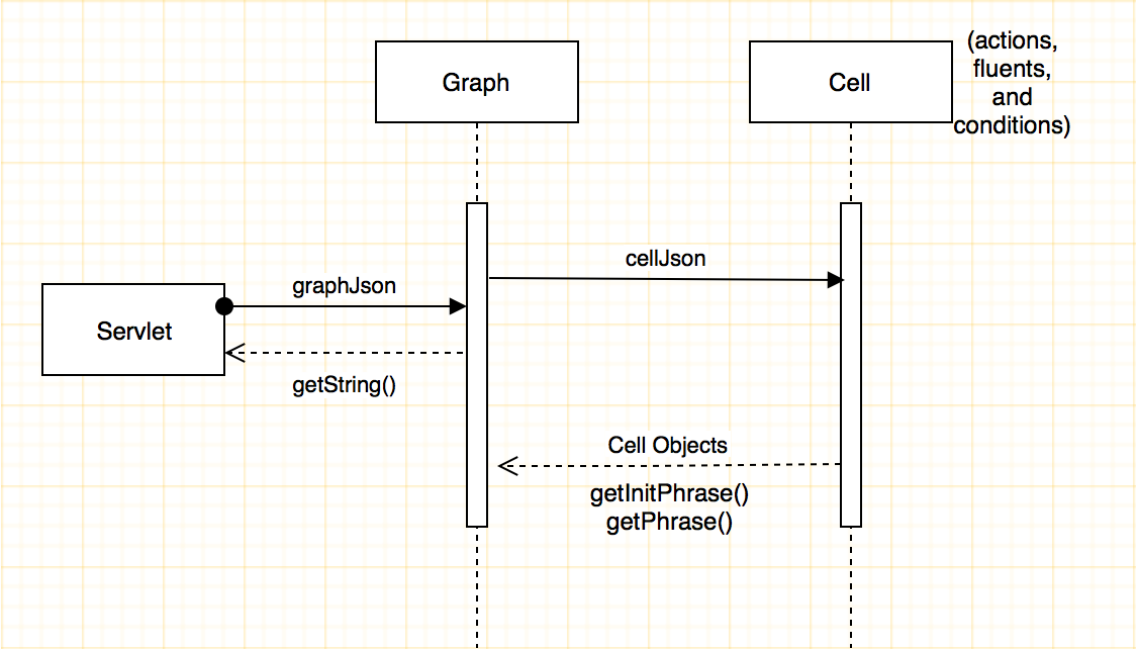

Figure 5.5: Get Clause

The graph json is packed and send to the back-end application server via Ajax, and the servlet creates a new graph according to the graph type indicated in the graph json, and creates objects of Action, Fluent, and Condition class. After the graph is created and initiated by the json string, the servlet calls the getString() function to get the LPS code generated for the graph, and send it back to Ajax. The clause is demonstrated in the Edit panel of Code Area.

Chapter 6

Testing

6.1 Unit Testing

Unit Testing in the Front-End is conducted by javascript library Qunit. We do testing in folder /WebContent/testing on every javascript function which is not related to the web page element or the java back-end. The result html below indicates that all Qunit testing has passed. The other part will be tested in the integrated testing.

Figure 6.1: Qunit Testing

Unit Testing in the Back-End is conducted by Java Junit4 library. The aim of the Back-End unit testing is to cover every path of tested function and return all kinds of results.

Class	Function	Test Cases
CellUtils.java	toStandardForm(String)	12
CellUtils.java	toWildMatch(String)	18
Action.java	getInitPhrase(), getPhrase()	12
Fluent.java	getInitPhrase(), getPhrase()	12

Condition.java	getInitPhrase(), getPhrase()	3
Graph.java	populate(String)	6
InitGraph.java	getString()	8
ReactiveRule.java	getString()	8
MacroAction.java	getString()	8

6.2 Integration Testing

We deploy the project on localhost by Apache Tomcat 8.5 and open the web page localhost:8080/easyLPS/index.html to do integrated testing on every function and every button. The results are as follows:
Almost all of the testing was actually performed repeatedly with mutiple test cases, but for simplicity, we just show one example in each of the following rows. For testing described in section 2 and section 3, we prepare a test case recurrent_fire_graph.txt and upload the file before the testing. The file locates in folder https://github.com/luoxiao0123/easyLPS/tree/master/WebContent/testing.

Description	**Expected**	**Actual**
Section 1: WORKFLOW GRAPH		
INPUT AREA: type in or paste Non-word characters	only comma, at, space, forward and backward slash, four operators, and comparison operator preserve, the other disappear immediately	✓
BUTTON BAR: hover on Graph and click the Initialize/ReactiveRule/MacroAction button	A new graph with ten time lines appear below the input area	✓
BUTTON BAR: click the New Action/New Fluent/New Condition button	A new action/fluent/condition of size(100, 40) appears at height 40/20/30 pixels and a random width	✓
GRAPH REGION: select Text Mode, type in a valid action/fluent/condition predicate in the input area, and right click a(an) action/fluent/condition in the graph	action/fluent/condition element in the graph is modified to be the same as text in input area and is white; A new button is generated under New Action/New Fluent/New Condition with the same text	✓

GRAPH REGION: select Remove Mode, and right click a(an) action/fluent/condition in the graph	action/fluent/condition element in the graph is deleted	✓
GRAPH REGION: select Conclusion Mode, and right click a(an) action/condition with font-colour white in a ReactiveRule graph	font-colour of the action/condition element become red	✓
BUTTON BAR: hover on the New Action/New Fluent/New Condition and click a newly generated button below	A new action/fluent/condition of size(100, 40) appears at height 440 pixels and a random width with the same text in the button	✓
BUTTON BAR: hover on the ”Save..” button and click the Save/”Save As” button below when no graph is drawn in the graph region	alert ”Please draw a graph first!”	✓
BUTTON BAR: not type in the name in the input window when first saving a graph	the graph will not be saved	✓
BUTTON BAR: save a macro action graph to invalid names including ”*”, ”Macro”, ”macro(X,)”, ”macro(X, Y”	alert ”#invalid name# is not a valid name for a macroaction”	✓
BUTTON BAR: click the ”Get Clause” button when no graph is drawn in the graph region or the graph is not saved	alert ”Please save the graph first!”	✓
Section 2: CLAUSE PANEL		
type in or paste Non-word characters in all text input area	only comma, at, space, forward and backward slash, four operators, and comparison operator preserve, the other disappear immediately	✓
CAUSAL THEORY: type in or paste Non-word characters in the two text input area generated by selecting ”updates” as the clause	all Non-word characters disappear immediately	✓
GRAPH TO LPS: after uploading recurrent_fire_graph.txt	Four graph buttons appear in the column graphs	✓
GRAPH TO LPS: click the above four buttons respectively	Four corresponding graphs appear respectively in the graph area	✓

GRAPH TO LPS: after uploading recurrent_fire_graph.txt, click the ”to LPS” button	refill, eliminate, escape, and ignite(_) appear in the actions column, fire and water in the fluents column, and deal_with_fire in the macros column	✓
CAUSAL THEORY: After uploading recurrent_fire_graph.txt and clicking the ”to LPS” button, click the select area of action	refill, eliminate, escape, ignite(_), ignite(bed), ignite(sofa) appear in the select dropdown menu	✓
CAUSAL THEORY: After uploading recurrent_fire_graph.txt and clicking the ”to LPS” button, input ”ig” in action input area	ignite(_), ignite(bed), ignite(sofa) appear in the input dropdown menu to autocomplete	✓
Section 3: CODE AREA		
EDIT: upload recurrent_fire.txt and clicking the ”to LPS” button	text in appendix appear in the textarea and can be modified directly	✓
SELECT: upload recurrent_fire.txt and clicking the ”to LPS” button	text in appendix appear in the region and cannot be modified	✓
SELECT: double click line 1-2 of the above code	nothing happens	✓
SELECT: double click line 3-8 of the above code	initialization graph init is displayed in graph region	✓
SELECT: double click line 10-11 of the above code	the first macro action graph is displayed in graph region	✓
SELECT: double click line 12-13 of the above code	the second macro action graph is displayed in graph region	✓
SELECT: double click line 14-15 of the above code	reactive rule graph react is displayed in graph region	✓
SELECT: double click line 16-18 of the above code	causal theory panel is displayed in clause panel	✓
SELECT: double click line 19 of the above code	precondition panel is displayed in clause panel	✓
Section 4: MENU BAR		
GET LPS: click the button when no graphs, causal theories, or preconditions are submitted	no error thrown, no actions, fluents, or macros demonstrated and no input help in clause panel	✓

GET LPS: select more than one initialization graph when generating the final LPS code	no error thrown	✓
DOWNLOAD: type in Non-word characters for the file name	fail to download, alert error message ”only letters and numbers are allowed in the file name, please input again”	✓
DOWNLOAD: click the button when no graphs, causal theories, or preconditions are submitted	no error thrown, download successfully, all key withs value empty list	✓
UPLOAD FILE: click the button	only txt files are allowed to be uploaded	✓
UPLOAD FILE: upload a txt file containing no json string	alert ”failed to parse the file! Please upload the correct file”	✓
UPLOAD FILE: upload the same valid txt file consecutively	nothing happens since the second upload	✓
UPLOAD FILE: upload a valid txt file in which at least one of graphs with id already presented in ”Graph to LPS” panel	those graphs will not be loaded again, the others will be loaded normally	✓
RUN IN SWISH: click the button	SWISH opens in a new tab	✓

6.3 Performance Testing

We deploy the easyLPS workflow interface on Azure App Service, which can be accessed via https://easylpsazure.azurewebsites.net/. Then we run the following test, each for 1 minutes. Detailed graph information is attached in appendix D.

6.3.1 Azure Load Test with 20 concurrent users

The workflow interface web page provides perfect support to 20 concurrent users. The average response time is only 0.05 seconds, the average request per second is 242.82, and the successful request rate is 100 percent.

6.3.2 Azure Load Test with 200 concurrent users

Then we increase the number of virtual user to 200. We did not set a really large number, for fear the insufficiency of the azure credit. The successful rate request rate is still 100 percent, but the average request per second falls down to 53.33, and the response time goes up dramatically to 14.12 second per request. Since the peak of CPU Percentage and Memory Percentage in the testing are 74.5% and 76.67%,

respectively, it may be concluded that the rising in response time probably due to we choose a really cheap app service plan in azure for developing and testing, and the CPU and memory of the running environment is relatively low. Above all, the 100 percent successful request rate proves that the easyLPS web page is robust and stable.

Chapter 7

Evaluation

7.1 Key Feature Implemented

- The workflow interface for specifying LPS rules and clauses
- The clause panel for generating LPS clauses which cannot be represented by workflow interface
- The code area for editting and tracking LPS
- The translator from a single graph to LPS
- The translator from graphs, causal theories, and preconditions to LPS
- Allow debuging LPS by showing the original source
- Allow editing LPS via workflow interface
- Download selected graphs and clauses
- Upload text file to load graphs and clauses

7.2 Features Not Implemented

Due to time limitation, the login function and database feature has yet to be implemented. These are all self-proposed features, and will be detailedly discussed in the future work chapter. The potential of implementing those two functions actually demonstrate the advantage and huge extension possibility of Java Web project. The workflow interface right now is quite similar to a normal Java application, yet the establishment of database will enable users to store their work remotely. Furthermore, Java provides perfect support to SQL database, and the classes in domain package in Java can be easily transformed into SQL entities because it is previously considered during the design process.

7.3 Unresolvable Problems

A potential problem may be the insufficient check of the upload files. At present, only txt files are allowed to upload, and the file contents shall be a valid Json string. The Json string will be parsed and loaded as different page elements, including graphs, causal theories, and preconditions. The front-end javascript only checks whether the Json string has specific keys, yet does not check the validity of the values. Those value will then be sent to the back-end via Ajax if users do translation to LPS. If the file was modified directly, it might not throw an error on the front-end, yet might result in unexpected runtime exception in the back-end. Moreover, it cannot prevent vicious files from damaging the application server.

A possible solution might be to encrypt the file when downloading, and decrypt the file when uploading. However, this will hide the details of Json string to users, and may not be good for education purpose. Thus, the encryption was not implemented.

7.4 Evaluation

The product fulfills almost all the requirements of the project description. It provides with a workflow interface and an automatic translator to translate the workflow graphs into LPS code. Users can also track the graph by double clicking the generated LPS code.

The architecture of the code is well-designed. The four packages implements complete different functions, and the code is relatively easy to comprehend because of no overlapping in function implementation. The thought of object-oriented programming is well adopted in the coding, each domain class is encapsulated to avoid unauthorized access, and inheritance and abstraction largely reduce the redundancy of the code. All exceptions are caught and thrown to higher hierarchy with clear error messages, making the system more robust.

The translation accuracy is fully guaranteed. Unit testing and integration testing are cautiously performed to make sure that the generated LPS code are grammatically correct and can represent accurately all information demonstrated in the workflow graph. Each path of all significant javascript functions and Java functions is carefully covered in the unit testing. It also performs quite well in performance testing.

The web page also involves a number of small features such as error messages and input helping to make it more user-friendly. The front-end input restriction avoid users mistyping in characters which never appear in LPS. Input helping in Clause Panel largely reduces the possibility of users' making mistakes, and significantly promotes the efficiency of inputting. Moreover, the syntax checking of atomic actions, fluents, and macro actions presents the error immediately. The error messages state clearly where the error locates, reducing the amount of effort of correcting the error. Furthermore, users can download the contents they submitted and upload the file to

load those contents and continue working.

It is also highly-extensible, having lots of aspects to expand its functions. The developing process follows a spiral model, where extensibility is carefully considered at the beginning of architecture designing process, and features can be added flexibly. The web application server is written in Java, and provide support to various kinds of platform. A possible extension would be to establish a database and implement the log in function to preserve all the data users submitted in the web page. This is discussed in detail in chapter 8.

The greatest challenge in the project is that the workflow interface has to satisfy all kinds of situation in LPS code, and the boundary situations have to be carefully considered. This is solved by listing all the boundary conditions in advance, find out a back-end processing logic to accommodate all the situations, and conduct testing on them afterwards.

Nevertheless, in the process to make it more user-friendly and cover more situations in LPS language, even a small step may take huge effort, so there are still quite a few features are yet to improve, and these improvement will be discussed in the future work chapter.

Chapter 8

Future Work

8.1 Improving User Friendliness

The interface can be made more user-friendly: currently users have to select the Text Mode, Delete Mode, and Conclusion Mode, and then right click the elements in the graph to insert text, delete, and set the elements as part of conclusion, respectively. The process can be simplified to the situation that clicking the elements in the graph will show tools around the elements and the above functionalities can be achieved by clicking different tools. Another improvement can be to double click the element and directly modify the code. However, due to limitation of the current JointJS library, these functionalities may be hard to achieve, therefore we may import more powerful javascript library such as mxGraph to solve the problem if time permits.

8.2 Extending Acceptable Predicate

Currently the predicates of actions, fluents do not accept dash and square bracket, i.e., actions and fluents with the form of X-Y or with variables in form of lists such as A([X, Y], Z) cannot be expressed in the web page. In future work, actions and fluents should have their form expanded in order to hold more complex situations.

8.3 Singleton Variables and Safety Conditions

In LPS, most variables have to be grounded. It would be better if singleton variables are marked bold in the final LPS code. When singleton variables appear in the fluent of a causal theory, an error message shall be thrown out.

8.4 Adopting More Flexible Layout

Right now the layout of the page is fixed, and it is implemented by pure CSS. However, the format may change in different types of browsers. More care may need to be taken to adapt to different browsers of different sizes. In addition, the graph

width and height is fixed to be both 600 pixels, and time line number is set to be 10. Thus, three parameters can be added to adjust the graph size and the number of time lines.

8.5 Accommodating More Complicated Situations

In reactive rule graphs, there might be an atomic action and a macro action which start simultaneously, and in the graph representation, they will end at the same time, but this might not be the case in reality, and requires manually modification on the final LPS code. A possible solution might be adding a selectable attribute called ”time sequence” to the reactive rule graph: the default setting shall be ”on”, and it acts just like the current situation. But in above situation, users can set the attribute to be ”off”, and add the time relation themselves.

8.6 Log in Function and Database Establishment

Right now users can save their work by downloading the Json string of the page, yet it is more convenient if we build a database to store those information. For each user, we record the unique username and the password; For each graph, we record the username, the unique graph id, graph name, graph type, and graph Json string; For each causal theory, we record the username, the text, action, fluent, Json string of conditions; For each precondition, we record the username, the text, Json string of actions, Json string of fluents, and Json string of conditions. By the method above, we are able to establish a SQL database to realize the functionality to demonstrate all graphs, causal theories, and preconditions the user has previously saved or submitted when they log in.

Appendix A

Ethics Checklist

	Yes	No
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		✓
Does your project involve the use of human embryos?		✓
Does your project involve the use of human foetal tissues / cells?		✓
Section 2: HUMANS		
Does your project involve human participants?		✓
Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from "Human Embryos/Foetuses" i.e. Section 1)?		✓
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?		✓
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		✓
Does it involve processing of genetic information?		✓
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		✓
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		✓
Section 5: ANIMALS		
Does your project involve animals?		✓
Section 6: DEVELOPING COUNTRIES		
Does your project involve developing countries?		✓

If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		✓
Could the situation in the country put the individuals taking part in the project at risk?		✓
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		
Does your project involve the use of elements that may cause harm to the environment, animals or plants?		✓
Does your project deal with endangered fauna and/or flora /protected areas?		✓
Does your project involve the use of elements that may cause harm to humans, including project staff?		✓
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		✓
Section 8: DUAL USE		
Does your project have the potential for military applications?		✓
Does your project have an exclusive civilian application focus?		✓
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		✓
Does your project affect current standards in military ethics - e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?		✓
Section 9: MISUSE		
Does your project have the potential for malevolent/criminal/terrorist abuse?		✓
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?		✓
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?		✓
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?		✓
Section 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?	✓	

Will your project use or produce goods or information for which there are data protection, or other legal implications?		✓
Section 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?		✓

My project uses the following javascript libraries:

```
jQuery 3.1.1
jquery-ui 1.12.1
lodash 3.10.1
backbone 1.3.3
jointjs 2.1.3
Qunit 2.6.2
```

and the following Java libraries:

```
jakarta commons-lang 2.6
jakarta commons-beanutils 1.9.3
jakarta commons-collections 3.2.2
jakarta commons-logging 1.2
ezmorph 1.0.6
json-lib 2.4
```

The related javascript and CSS files are all stored in folder /WebContent/js and /WebContent/css, respectively, and all files include a copyright licensing description in the beginning. For all above libraries, commercial use, modification, distribution, patent use, and private use are all allowed, under the condition to disclose source, license and copyright notice, and same license (file). Those conditions are fully fulfilled in the project.
All other codes which are not originated have been marked in the source code.

Appendix B

Launch Guide

1. Download a Eclipse Installer on https://www.eclipse.org/downloads/
2. Open Eclipse Installer and select Eclipse IDE for Java EE Developers to download
3. Download Apache Tomcat 8.5 on https://tomcat.apache.org/download-80.cgi
4. Open Eclipse IDE for Java EE Developers and select File-New-New Dynamic Web Project
5. Type in easyLPS as the project name, select 3.1 as the Dynamic web module version, and click on ”New Runtime...”
6. Select Apache-Apache Tomcat v8.5-Next
7. Browse the Tomcat Installation Directory and click Finish, then click Finish to create the new web project
8. Download WebContent and src folders in https://github.com/luoxiao0123/easyLPS, and copy everything under these two folders to the newly created web project
9. Download json-lib Java library on https://sourceforge.net/projects/json-lib/files/ and its dependencies

   ```
   jakarta commons-lang 2.5
   jakarta commons-beanutils 1.8.0
   jakarta commons-collections 3.2.1
   jakarta commons-logging 1.1.1
   ezmorph 1.0.6
   ```

10. Right click the project, select Build Path-Configure Build Path
11. Select Libraries-Classpath, and click Add External JARs to add the above six Java libraries
12. Install a new server, and run the program successfully on http://localhost:8080/easyLPS/index.html

Appendix C

User Guide

C.1 Common Functions

- click Home button to refresh the page
- click Tutorial button to open the github repository of the web page in a new tab, and the tutorial is demonstrated in the README file
- click Run in Swish to open SWISH in a new tab

C.2 Start Programming on Workflow Graph

In this section, we will illustrate with an example to demonstrate what to do if we want to express with the workflow interface what will happen after a person feels hungry.

C.2.1 Initialize

First of all, we would like to show the observations and initial states: bob feels hungry from time 3 to time 4, and he intends to buy some food to eat. Then we find that available food includes burger, whose price is 10 pounds. Bob does not have any cash with him, but his account balance is 100 pounds.
The above facts can be expressed with a Initialization Graph. The steps are as follows:

1. hover over Graph button and click on Initialize to generate the graph paper

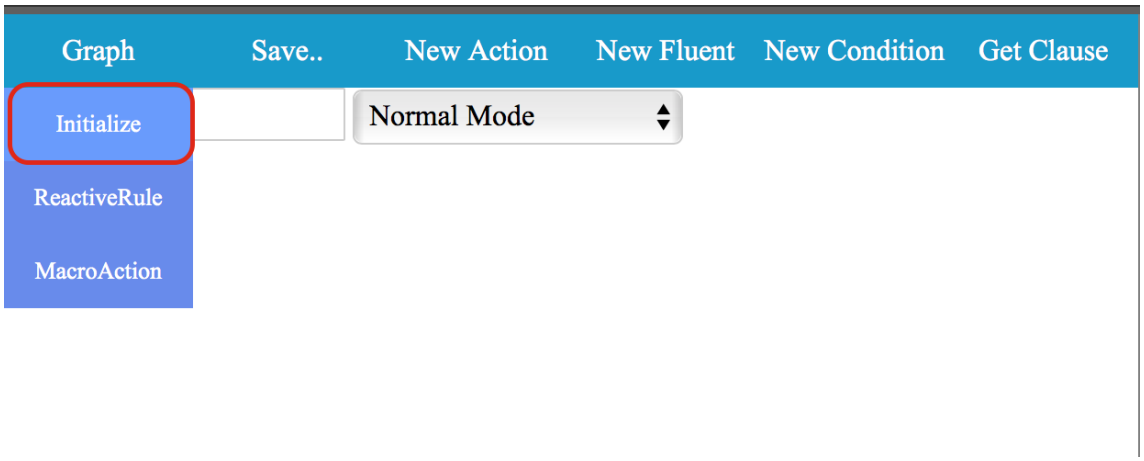

Figure C.1: Initialize

2. add all observations: click on New Action button, a green box for action will be generated. Select Text Mode and insert feel_hungry(bob) to the input area and right click the green box to insert text.

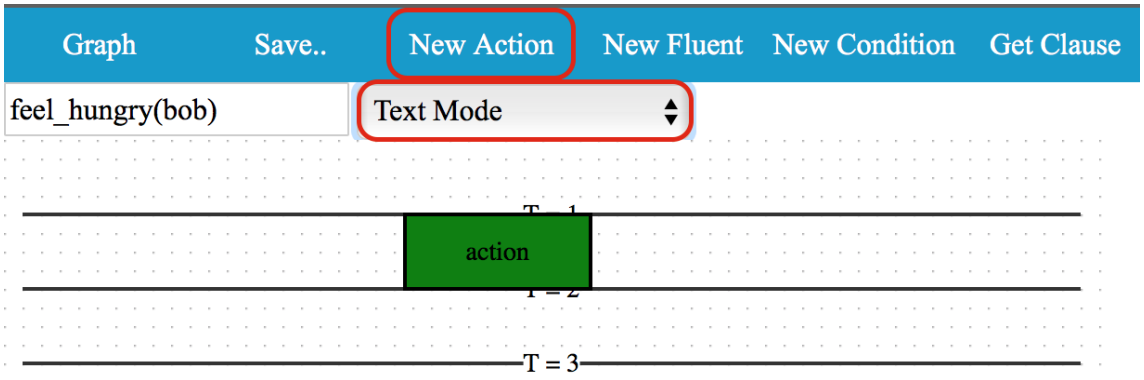

Figure C.2: Observation1

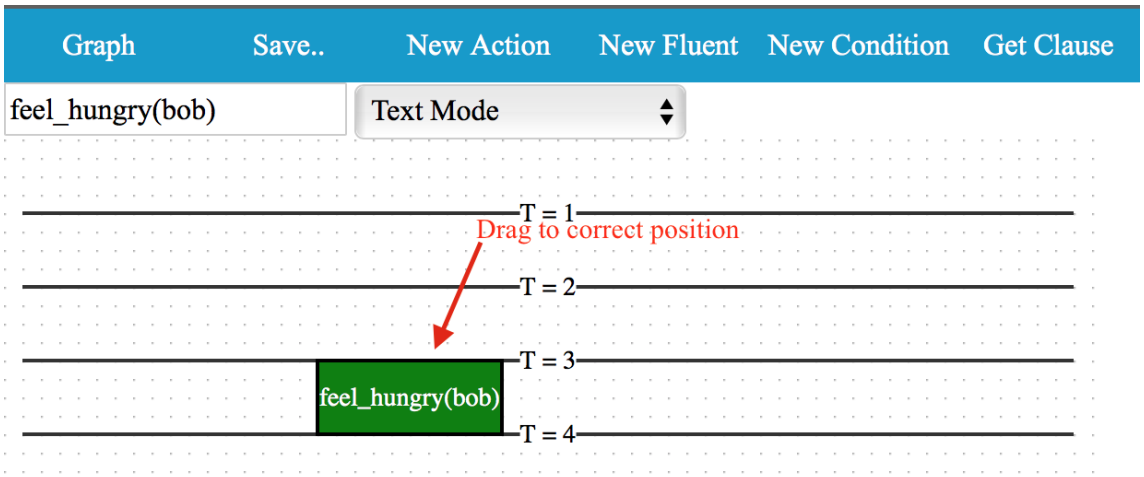

Figure C.3: Observation2

Drag the box to a position between T = 3 and T = 4. Right now the below graph means that bob feels hungry from time 3 to time 4.

3. add all initial states: click New Fluent button twice to generate two blue rectangles for fluent. Type in balance(bob, 100), cash(bob, 0) respectively and right click those tow blue rectangles to insert the text.

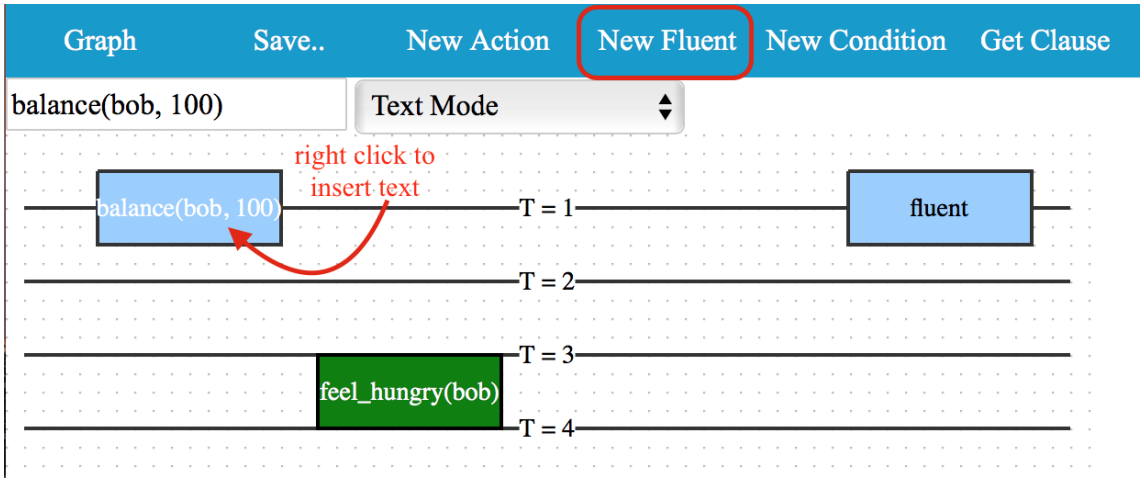

Figure C.4: Initial states

The two blue rectangles express that Bob does not have any cash with him, but his account balance is 100 pounds.

4. add all timeless facts: click on New Condition twice and insert the text in the same way to illustrate that available food includes burger, whose price is 10 pounds.

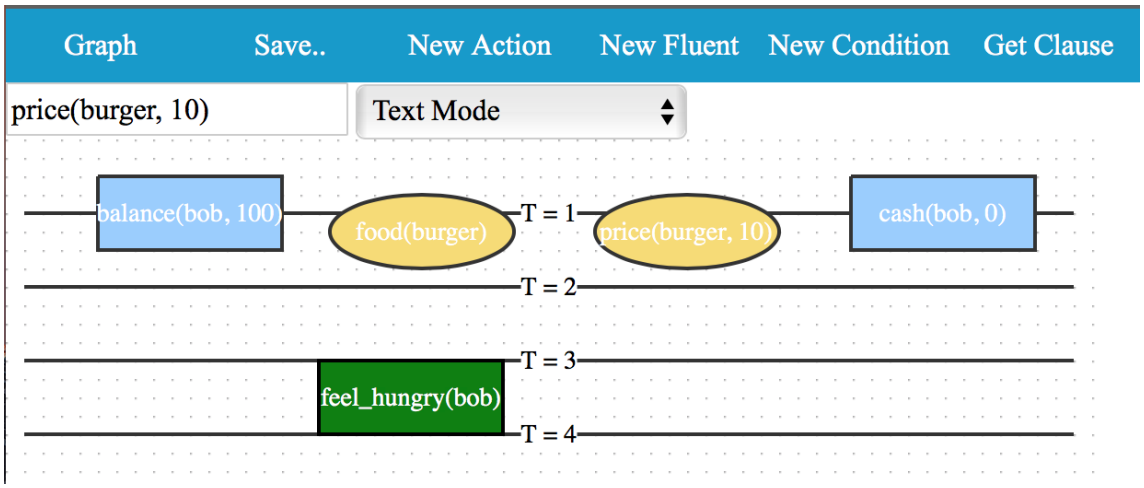

Figure C.5: Timeless Facts

5. hover over Save.. button and click on save

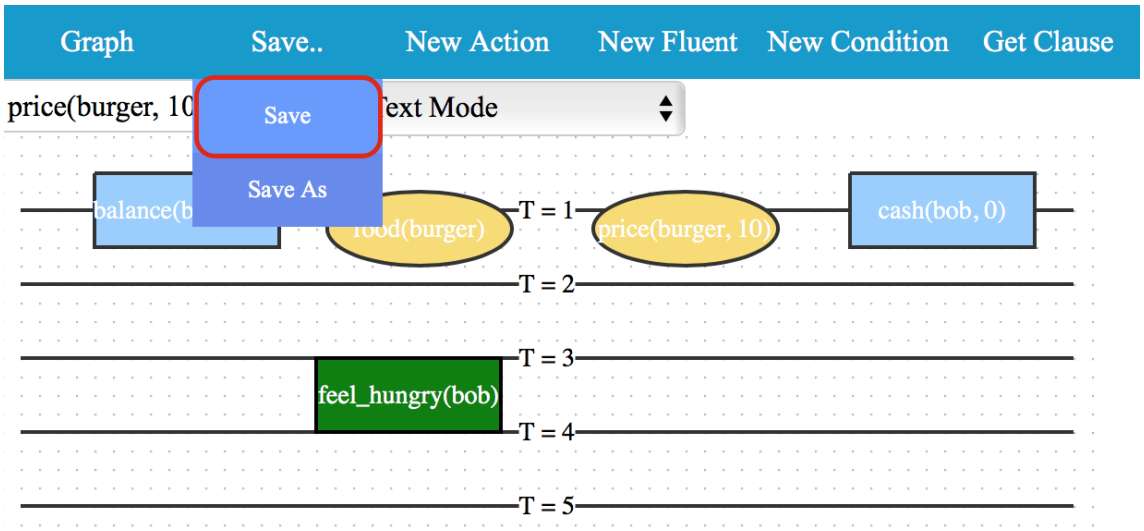

Figure C.6: Save1

6. input a random name for the initialization graph and save

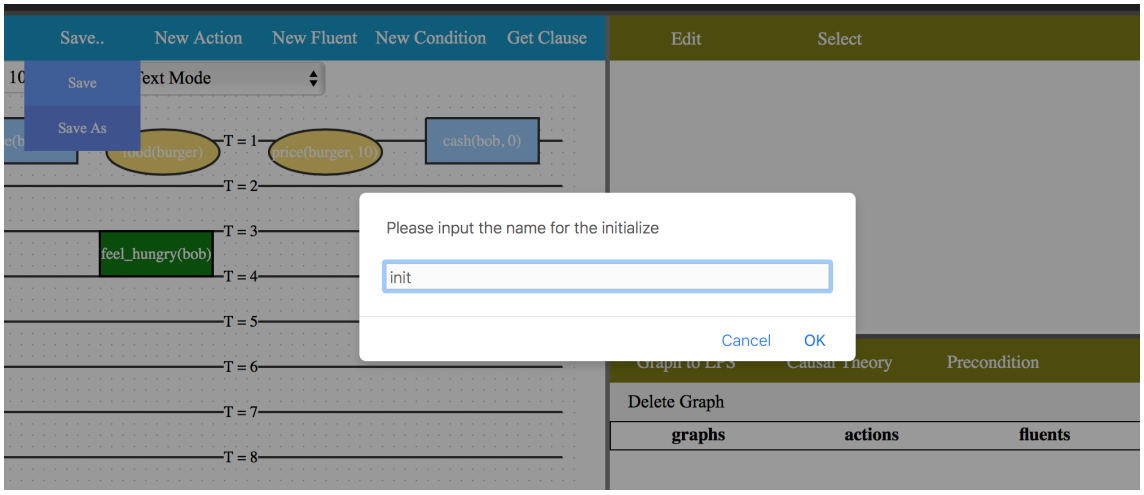

Figure C.7: Save2

7. click Get Clause button to get LPS code for the graph. The LPS code will be demonstrate in the code area. The button in the clause panel will store all the information about the graph.

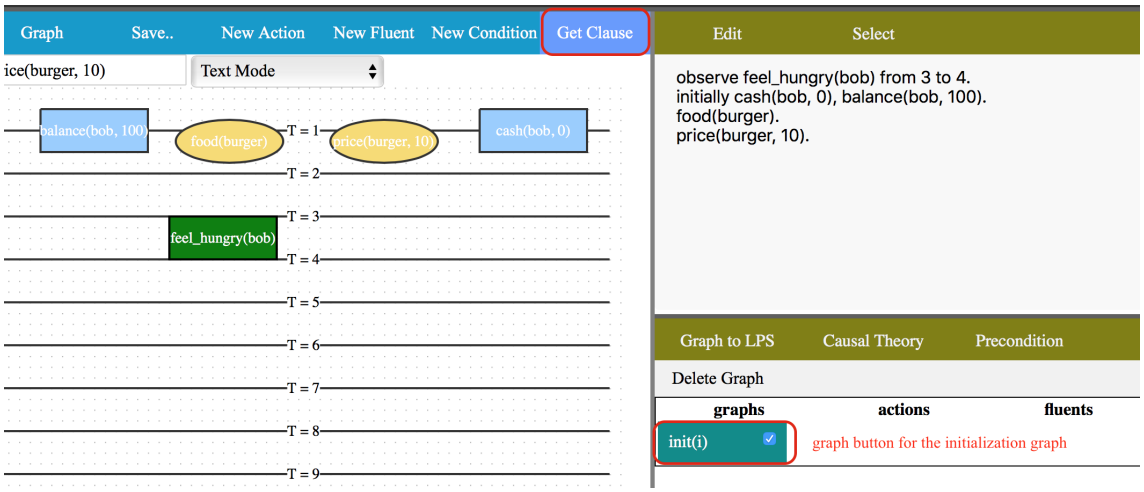

Figure C.8: Get Clause

C.2.2 Reactive Rule

After setting the initial states, we would like to move on to state some rules: if a person feels hungry, he or she will try to buy some food. If a person pays for food, and he or she is hungry, he or she will eat the food immediately. Each rule will be expressed by a Reactive Rule Graph. The steps are as follows:

1. hover over Graph button and click on ReactiveRule

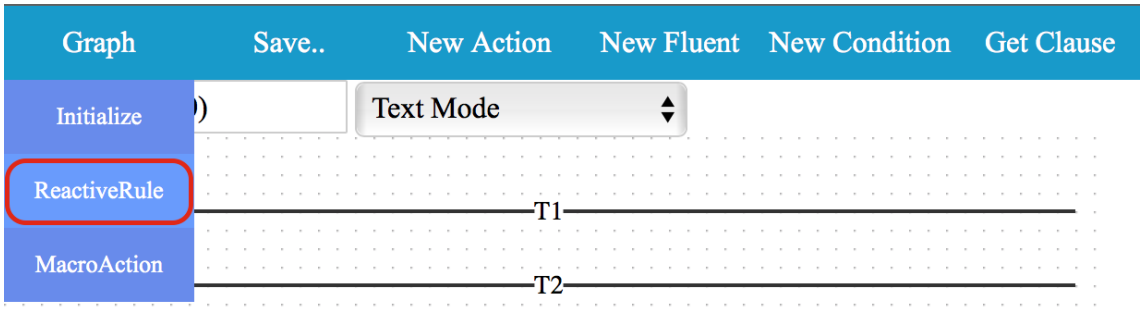

Figure C.9: Reactive Rule

2. add all antecedent: feel_hungry(P) here means variable P (a person) feels hungry.

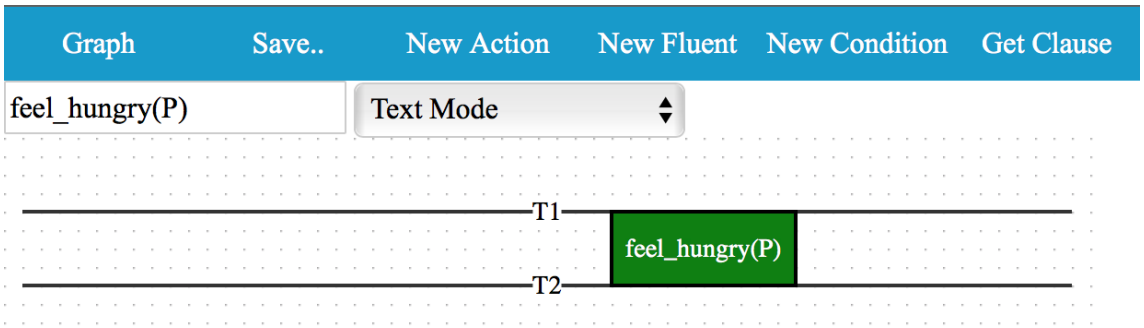

Figure C.10: Antecedent

3. add all consequent: after creating element and inserting text, select Conclusion Mode and right click to set the cell to be consequent. If we want to reset a consequent to be an antecedent, right click the red font colour again to make it white.

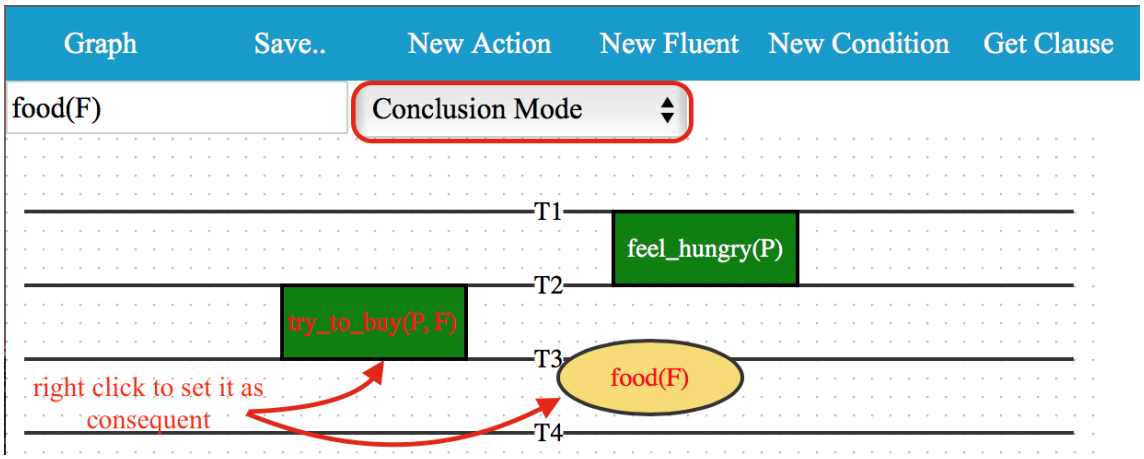

Figure C.11: Consequent

4. hover over Save.. button and click on save

5. input a random name for the reactive rule graph and save. This graph means that if a person feels hungry, he or she will try to buy some food. Please note that a Reactive Rule graph must include at least one antecedent and at least one consequent, and all antecedents should appear earlier (higher in position in graph) than all consequents. The sequence of each cell determines the sequence of predicates in the LPS clause. For instance, try_to_buy(P, F) appears before food(F) in the LPS clause only because try_to_buy(P, F) is higher in position in graph than food(F).

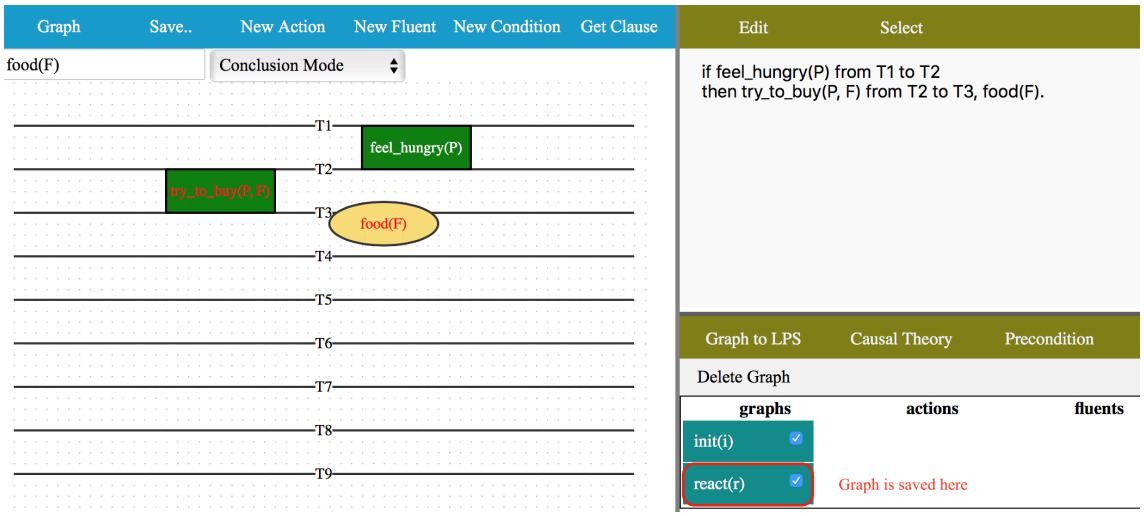

Figure C.12: Save3

6. repeat step 1-5 above to generate more reactive rules.

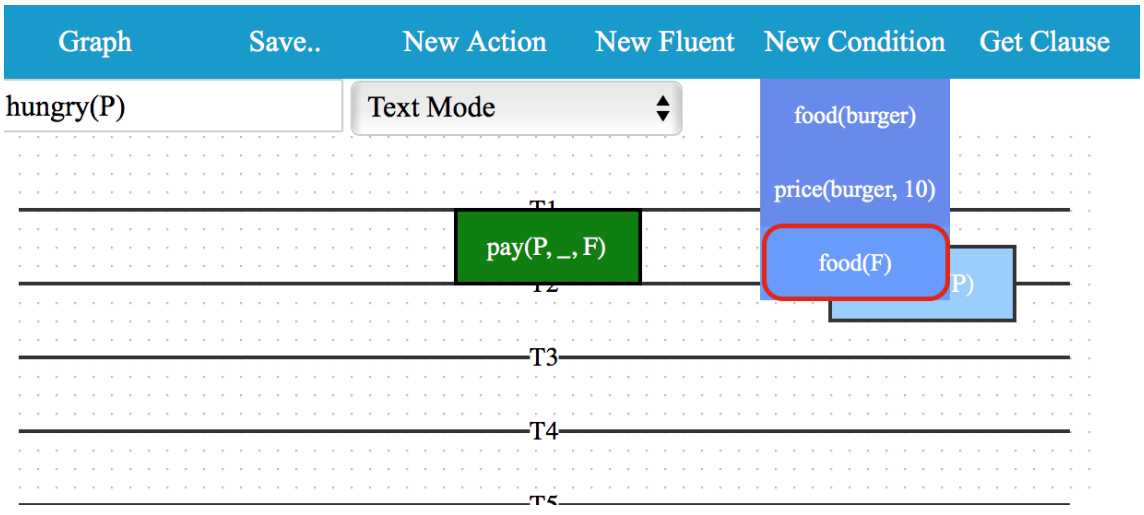

Figure C.13: Quick Generation of Cells

Since we have input before a condition that variable F belongs to food, we can hover over New Condition and click the ”Food(F)” directly to generate the condition already with the same text. This is the same with New Action and New Fluent.

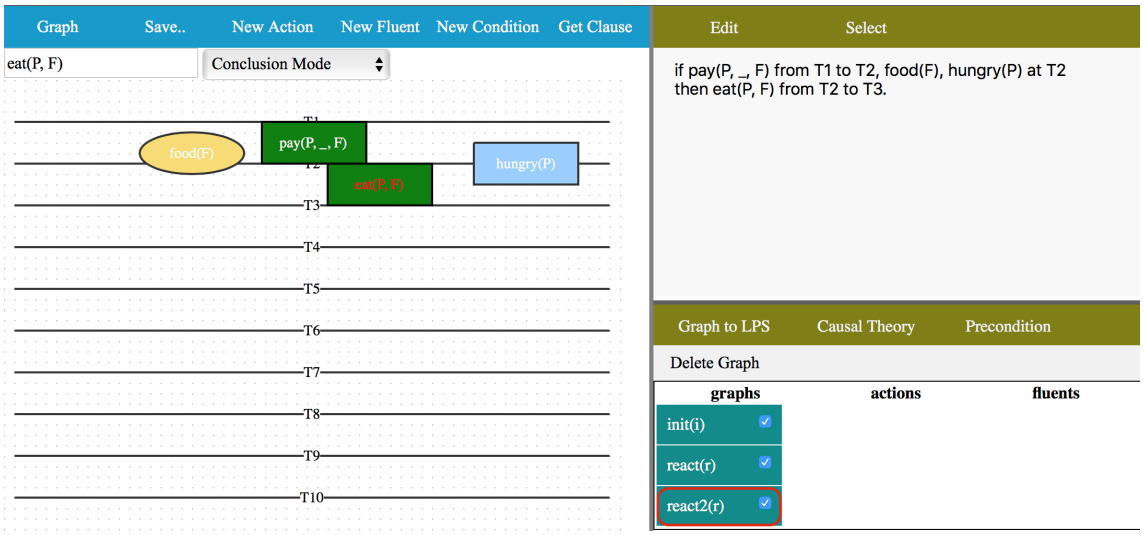

Figure C.14: Save4

The above graph illustrates that if a person pays for food, and he or she is hungry, he or she will eat the food immediately.

C.2.3 Macro Action

Some of the actions might consist of a series of actions, fluents, and conditions, and those are called macro actions. For instance, the try_to_buy(P, F) indicating the a person P tries to buy something F is actually a macro action. It includes the following three situations:

- the food F is of price Pr, and the person P pay with cash
- the food F is of price Pr, and the person P has to withdraw some money M to pay the bill
- the food F is of price Pr, and Pr is greater than person P's cash plus deposit, so P does nothing.

The steps are as follows:

1. hover over Graph button and click on MacroAction

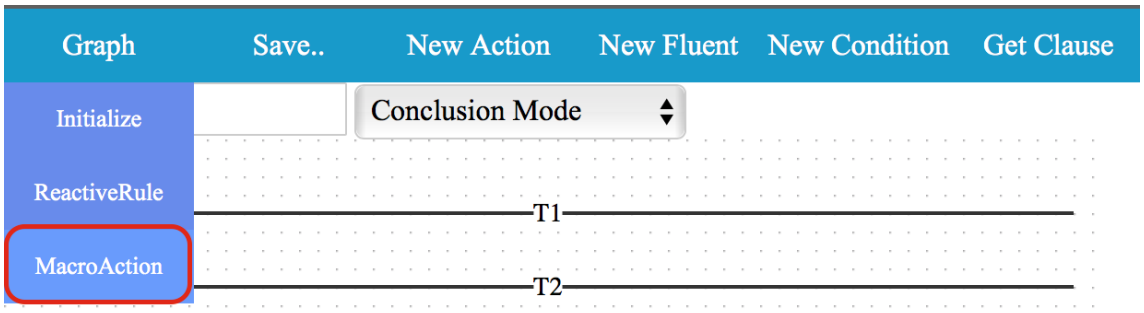

Figure C.15: Macro Action Graph

2. add the antecedents sequentially

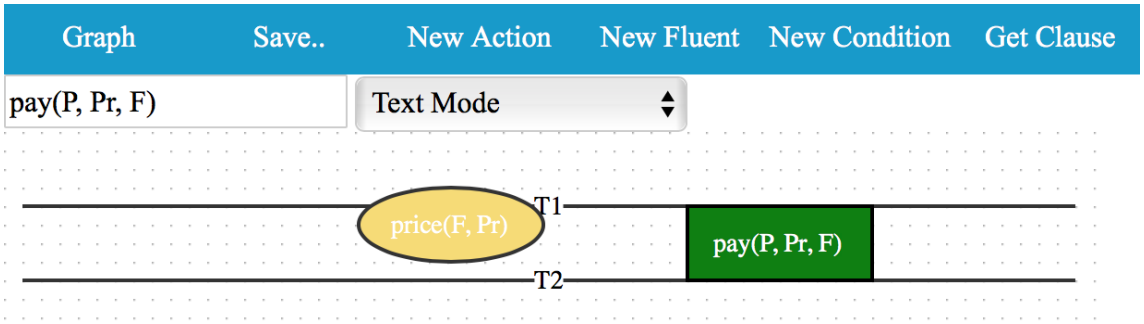

Figure C.16: Macro Action antecedent

3. save the graph with name of the macro action predicate(in this case same with name ”try_to_buy(P, F)”) and click ”Get Clause” to check

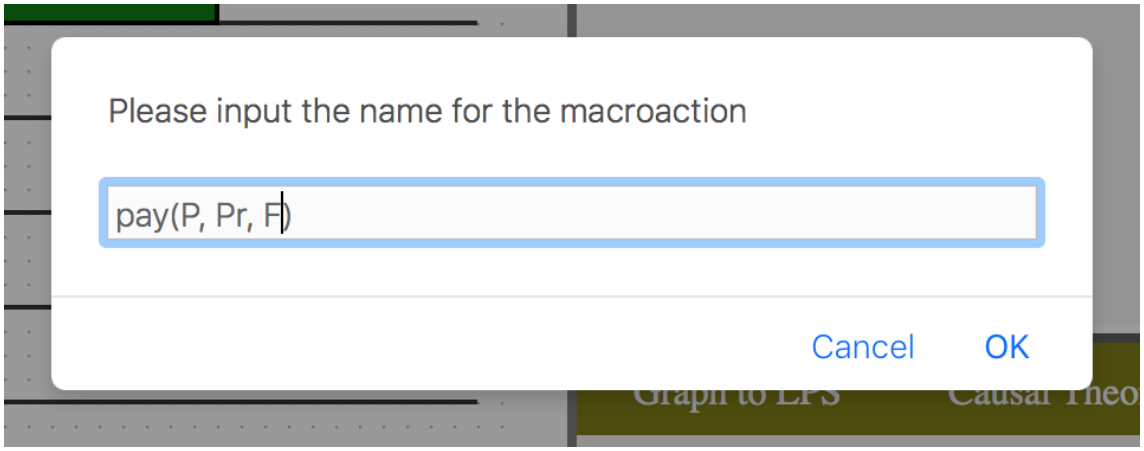

Figure C.17: Save5

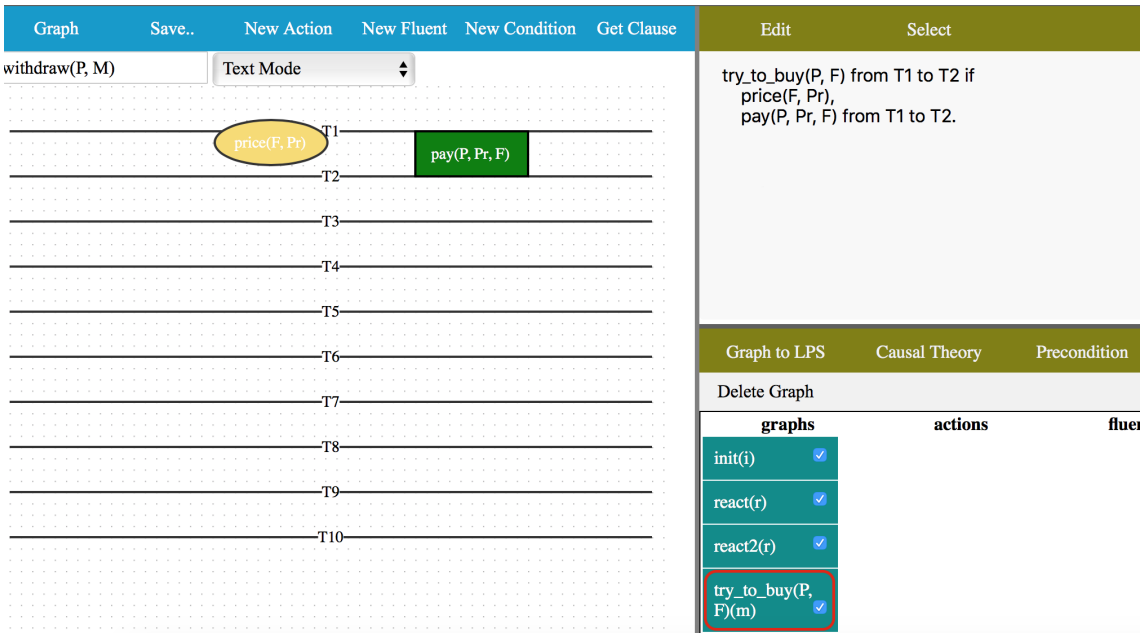

Figure C.18: Save6

4. repeat step 1-3 to generate more graphs

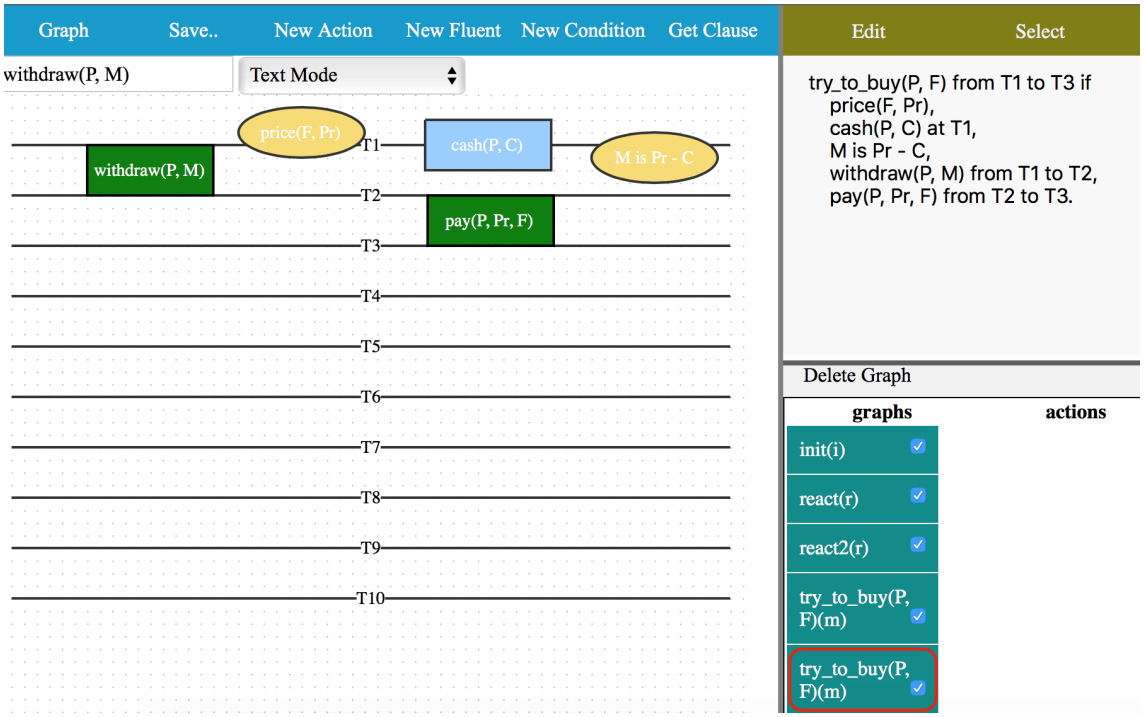

Figure C.19: Save7

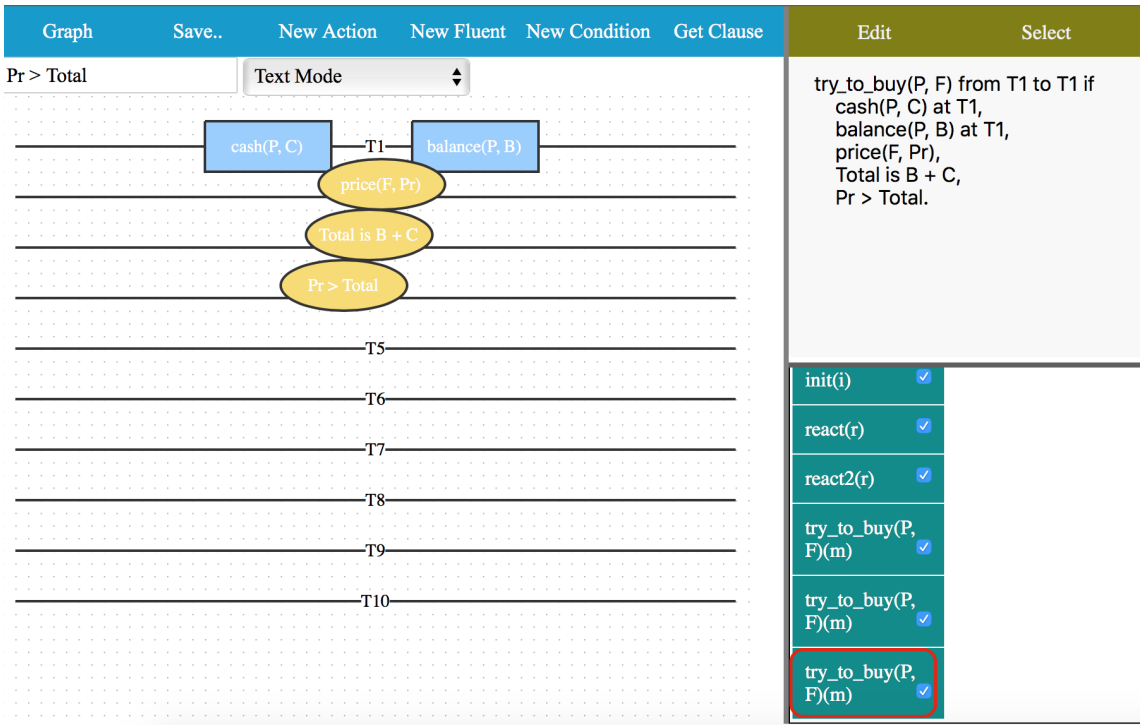

Figure C.20: Save8

C.3 Generating Time-independent Clause

C.3.1 Checking

select the graph buttons we want to translate to final LPS code and click ”to LPS” button. Confirm that in ”Graph to LPS” panel predicates under actions column are all atomic.

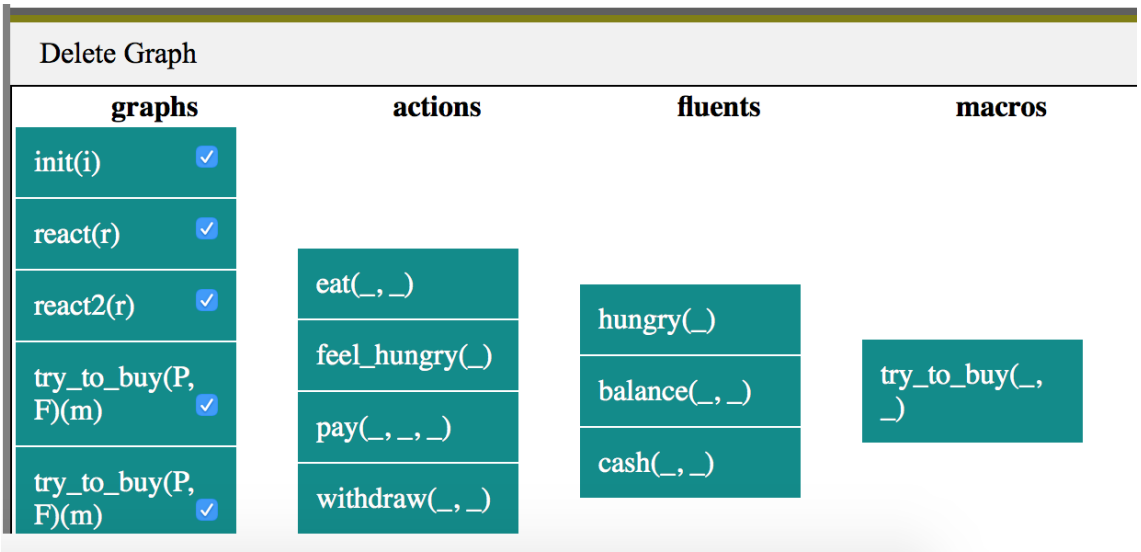

Figure C.21: Checking

The ”delete graph” button delete all selected graphs in the graphs column.

C.3.2 Causal Theory

Fluents cannot be directly modified, they can only be instantiated in the initial states, or be affected by actions in causal theory. For example, if a person P starts to feel hungry, fluent hungry(P) will be initiated. If the person eat something, hungry(P) will no longer be valid. In addition, if the person P withdraw some money M, his or her balance B will be updated to B - M, and cash C will be updated to C + M.
The steps are as follows.

1. click the Causal Theory button to enter the panel
2. for normal causal theory, select or input the action, fluent, and select the cause

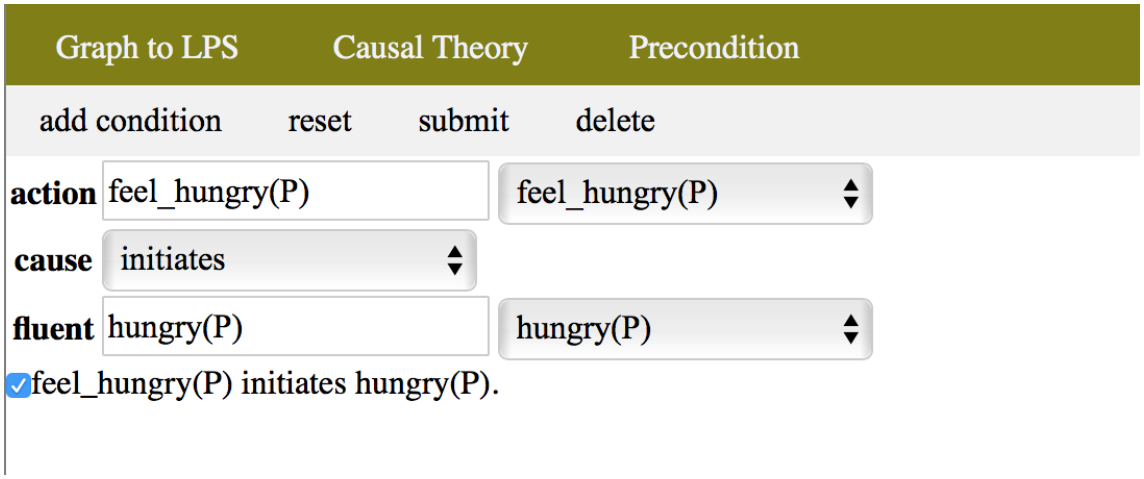

Figure C.22: Causal Theory 1

3. for causal theory with condition, select or input the action, fluent, and select the cause, then click the add condition button and input or select the condition.

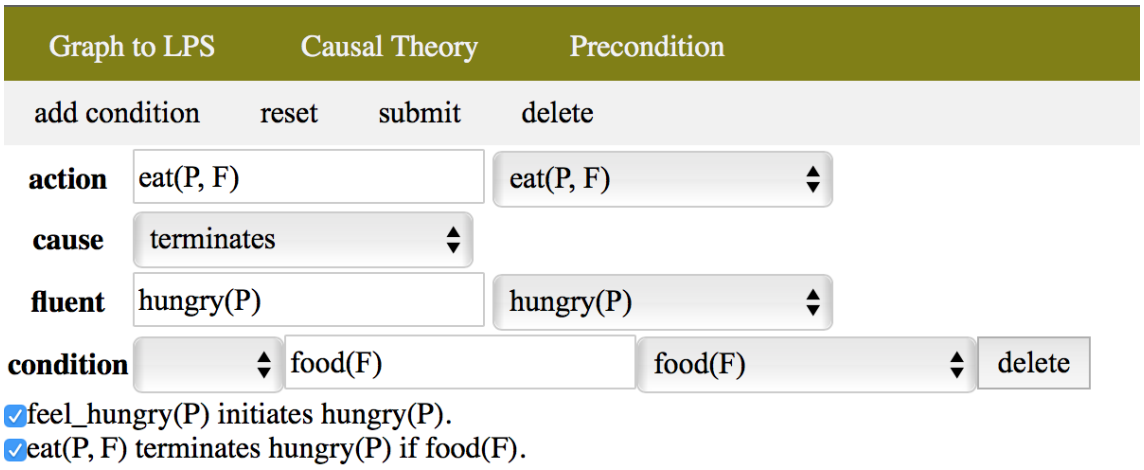

Figure C.23: Causal Theory 2

4. if the cause is selected to be update, a new line will appear in the middle for users to input the original state and the new state of a variable in the fluent.

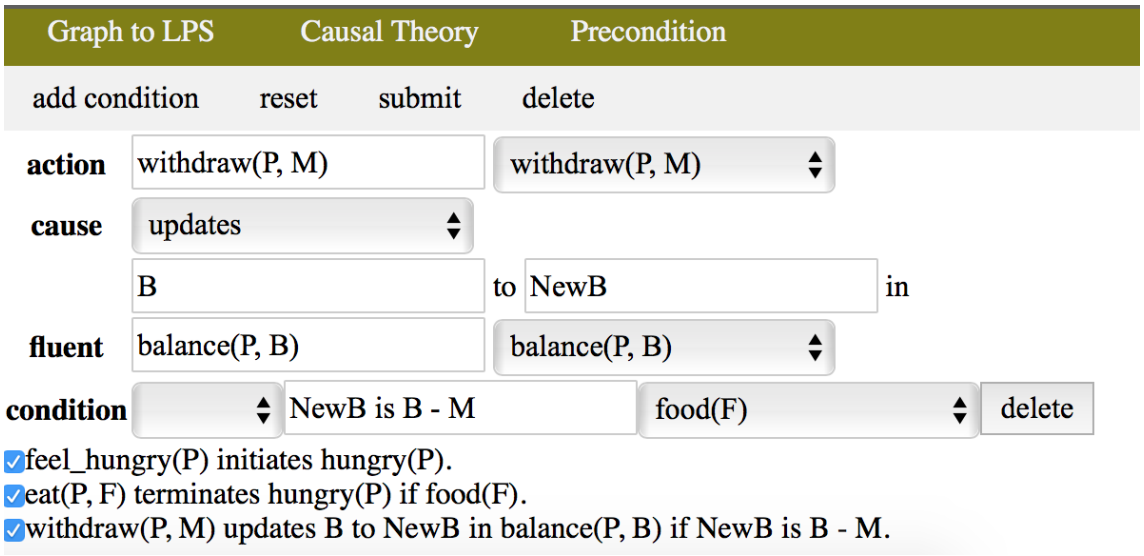

Figure C.24: Causal Theory 3

5. finally, click the submit button to generate the clause.

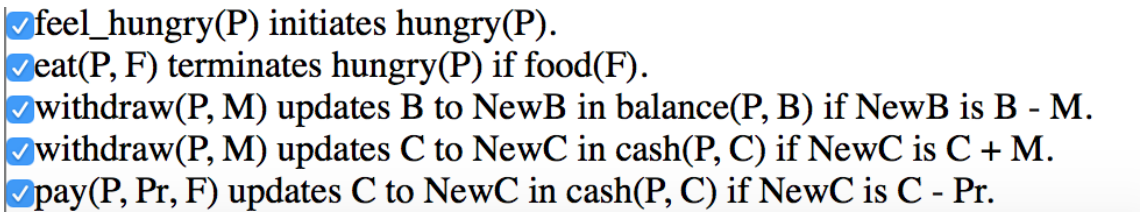

Figure C.25: Causal Theory 4

Reset button reset the panel to its original state, the delete button in the button bar delete selected clauses at the bottom, and the delete button at the end of each row of condition delete the entire row. The contents of the select area and the autocomplete function of the input area is presented only if we generate the LPS code for the graphs previously.

C.3.3 Precondition

Apparently there are some predicates that cannot happen together: if person P holds less amount of cash than the price Pr of product F, P cannot pay for the product; Person P cannot withdraw more money than the balance of the account.
The steps for the generating preconditions are:

1. click the Precondition button to enter the panel
2. click add action, add fluent, or add condition several times to create the input area
3. select or directly input the action, fluent, or condition
4. click submit button and the clause will be generated at the bottom

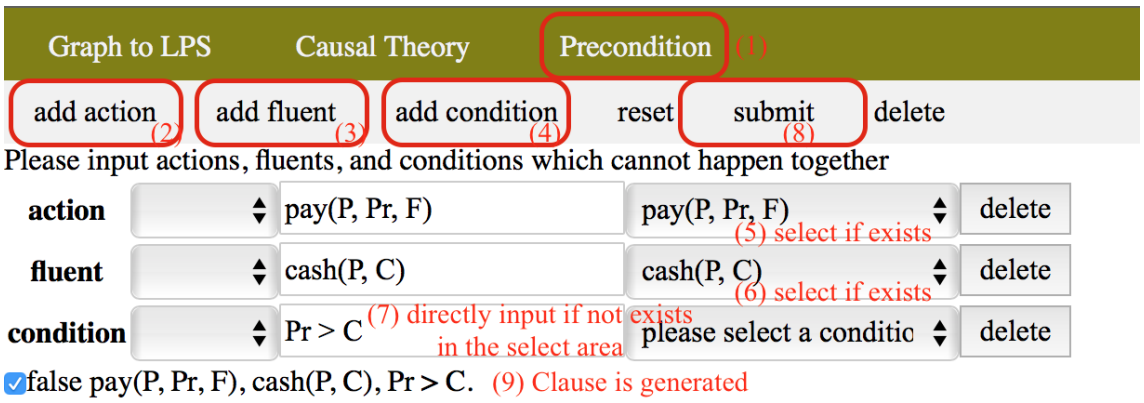

Figure C.26: Precondition1

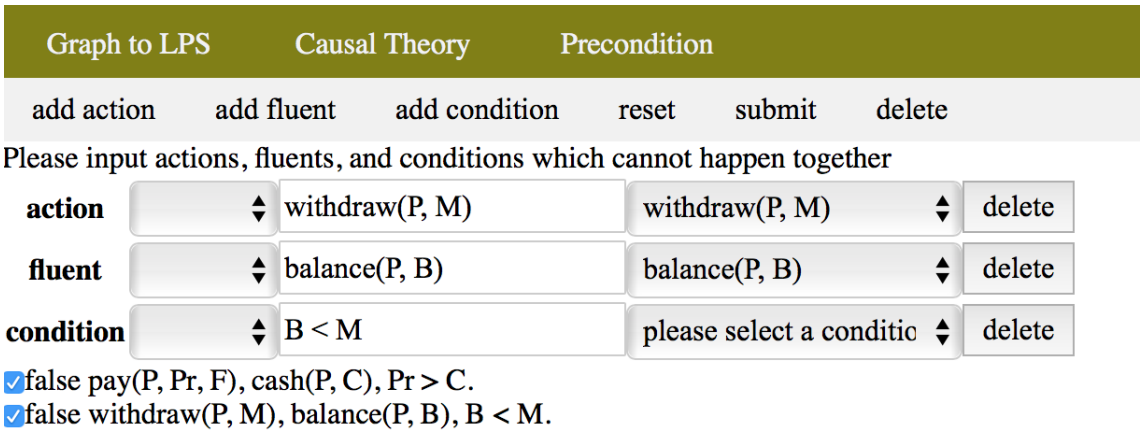

Figure C.27: Precondition2

The reset button clear the input field and set the Precondition panel to its initial state, the delete button in the button bar delete selected clauses at the bottom, and the delete button at the end of the row of input field delete the entire row. The contents of the select area and the autocomplete function of the input area is presented only if we generate the LPS code for the graphs previously.

C.4 Modification

C.4.1 Locate the Graph

There are two ways to locate the graph, one way is to click the graph button in ”Graph to LPS” panel directly, and the other is to double click the code in the select panel of code area, and the corresponding graph will be shown in the graph region.

C.4.2 Modify the Graph

click the corresponding graph button in ”Graph to LPS” panel or code in select panel, and the graph will be demonstrated in the graph region. After the modification, hover over ”Save..” and click the ”Save” button, and the modification will be saved.

C.4.3 Duplicate the Graph

click the corresponding graph button in ”Graph to LPS” panel or code in select panel , and the graph will be demonstrated in the graph region. After the modification, hover over ”Save..” and click the ”Save As” button to save the graph to a new name, and the new graph will be saved separately, and the original graph is not modified.

C.5 Download and Upload

When we get all graphs, causal theories, and preconditions ready, we can select those we want, and click ”Download” button to save the json string of the page into a text file.
The next time we open easyLPS, we can upload the saved file, and all graphs, causal theories, and preconditions are preserved. (Clicking to LPS button this time can regenerate the LPS code).

C.6 Run in Swish

When we get all graphs, causal theories, and preconditions ready, we can select those we want, and click ”to LPS” button again to generate the final LPS code.

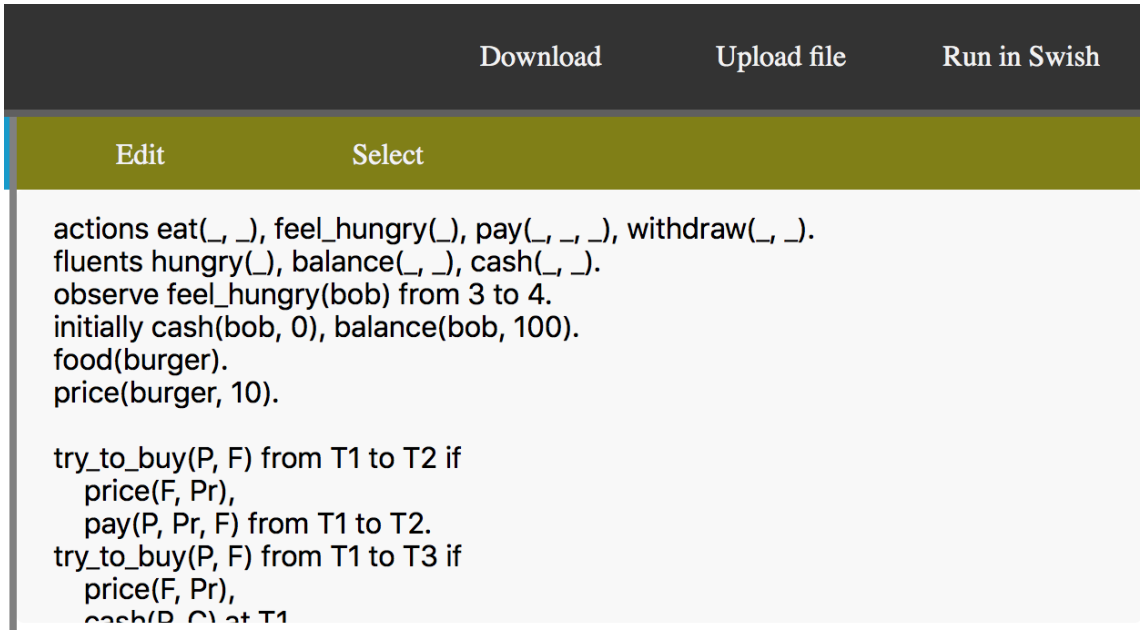

Figure C.28: Final Code

After the code is generated, we copy the code, click ”Run in Swish” button, create a new program in Swish, and paste the code. Then type in ”go(Timeline)” in the command line, and click ”Run!” to get the running result of LPS.

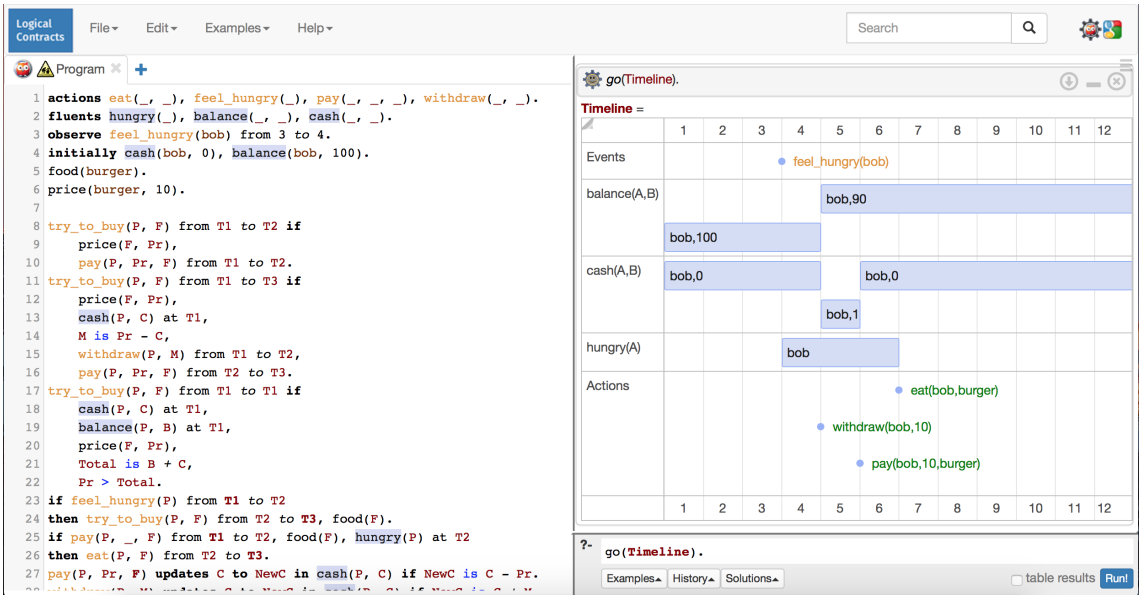

Figure C.29: Run in Swish

Appendix D

Azure Performance Testing Results

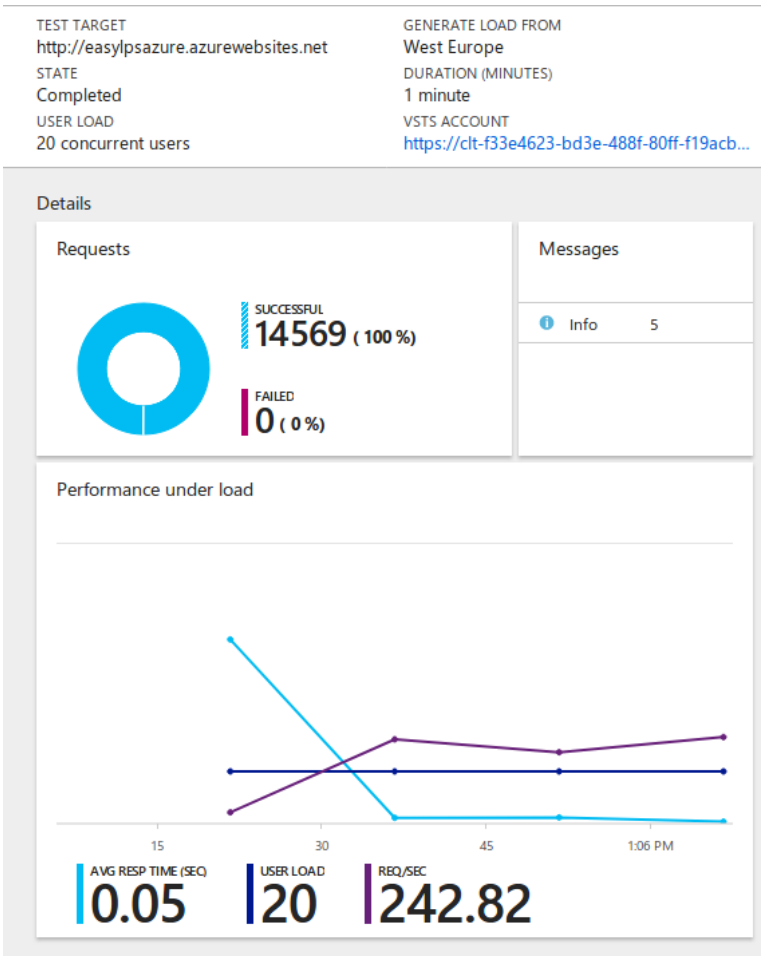

Figure D.1: 20 concurrent users

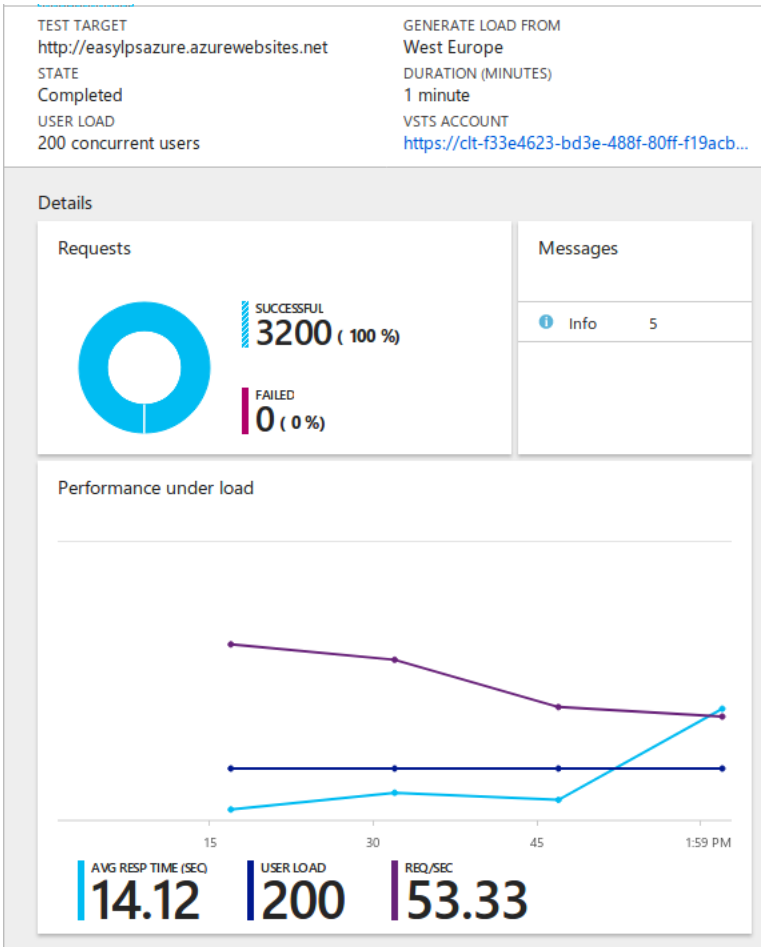

Figure D.2: 200 concurrent users

Bibliography

[1] Logical contracts.
urlhttp://http://demo.logicalcontracts.com. Accessed: 2018-09-07. pages 4

[2] LPS—logic production systems.
urlhttp://lps.doc.ic.ac.uk. Accessed: 2018-09-07. pages 4

[3] *WOOL: A Workflow Programming Language*, 2008. pages 5

[4] Ahmed Elmagarmid and Weimin Du. Workflow management: State of the art versus state of the products. In *Workflow management systems and interoperability*, pages 1–17. Springer, 1998. pages 5

[5] Robert Kowalski and Fariba Sadri. A logic-based framework for reactive systems. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 1–15. Springer, 2012. pages 4

[6] Robert Kowalski and Fariba Sadri. Programming in logic without logic programming. *Theory and Practice of Logic Programming*, 16(3):269–295, 2016. pages 4

[7] Jussi Vanhatalo, Hagen Völzer, Frank Leymann, and Simon Moser. Automatic workflow graph refactoring and completion. In *International Conference on Service-Oriented Computing*, pages 100–115. Springer, 2008. pages 5

[8] Jan Wielemaker, Fabrizio Riguzzi, Robert Kowalski, Torbjorn Lager, Fariba Sadri, and Miguel Calejo. Using SWISH to realise interactive web based tutorials for logic based languages. pages 3, 4

[9] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. pages 4