Knowledge Representation And Reasoning

Project 3 : Deterministic Actions and Agents

Created By
ADEYEMI ADEDAYO TOLULOPE

September 13, 2021

Contents

Chapter 1

Problem Statement

1.1 Problem Statement

Let DS_3 be a class of dynamic systems satisfying the following assumptions:

A1. Inertia law.
A2. Compete information about all actions and all fluents.
A3. Branching model of time.
A4. Only deterministic actions are allowed.
A5. Only sequential actions are admitted.
A6. Characterizations of actions:

- Precondition represented by a set of literals (a fluent f or its negation ¬f); if a precondition does not hold, the action is executed with empty effect;
- Postcondition (i.e., effect of an action) represented by a set of literals;
- An agent who performs an action; if effects of an action are not specified for a given agent, then we assume that the effect of the action is empty.

A7. Effects of an action depend on a state where the action starts and an agent who performs that action.
A8. All actions are performed in all states.
A9. Partial descriptions of any state of the system are allowed.
A10. No constraints are defined.

A program is a sequence P = $((A_1, ag_1), \ldots ,(A_n, ag_n))$, n > 0, such that A_i is an action and ag_i is an agent performing A_i , i= 1, . . . , n. An agent ag is involved in executing a program P whenever he/she performs some action A_i and the effect of this action is non-empty.

1.2 TASK

Define an action description language ADL(DS_3) for representing dynamic systems of the class specified above, and define the corresponding query language which allows

us to get answers for the following queries:

Q1. Does a given goal condition Υ (a set of literals) hold after performing a given program P in an initial state?

Q2. Is an agent ag involved in executing a given program P? Implement the action language and the query language specified above.

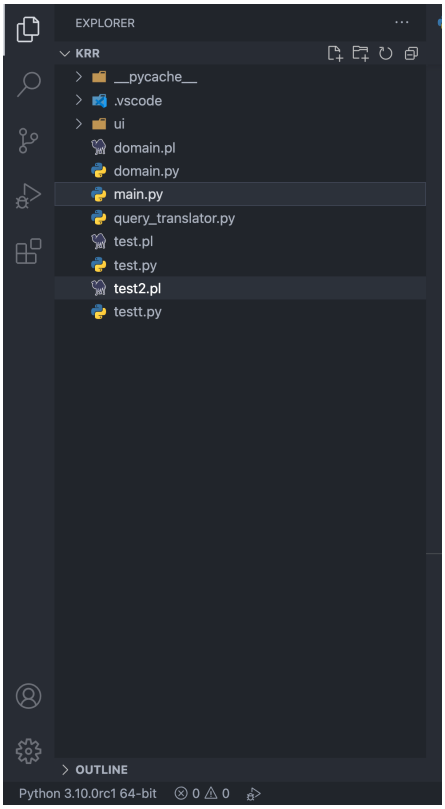
Remark: According to the specification given above, ADL(DS_3) is an extension of the language A where agents are involved.

Chapter 2

Project Implementation

2.1 PROJECT STRUCTURE

This project is a Desktop application that was designed using Python and Prolog. The project was developed on Visual Studio code and the figure below shows the project structure:



Below is the existing classes and the descriptions of the operations they perform:

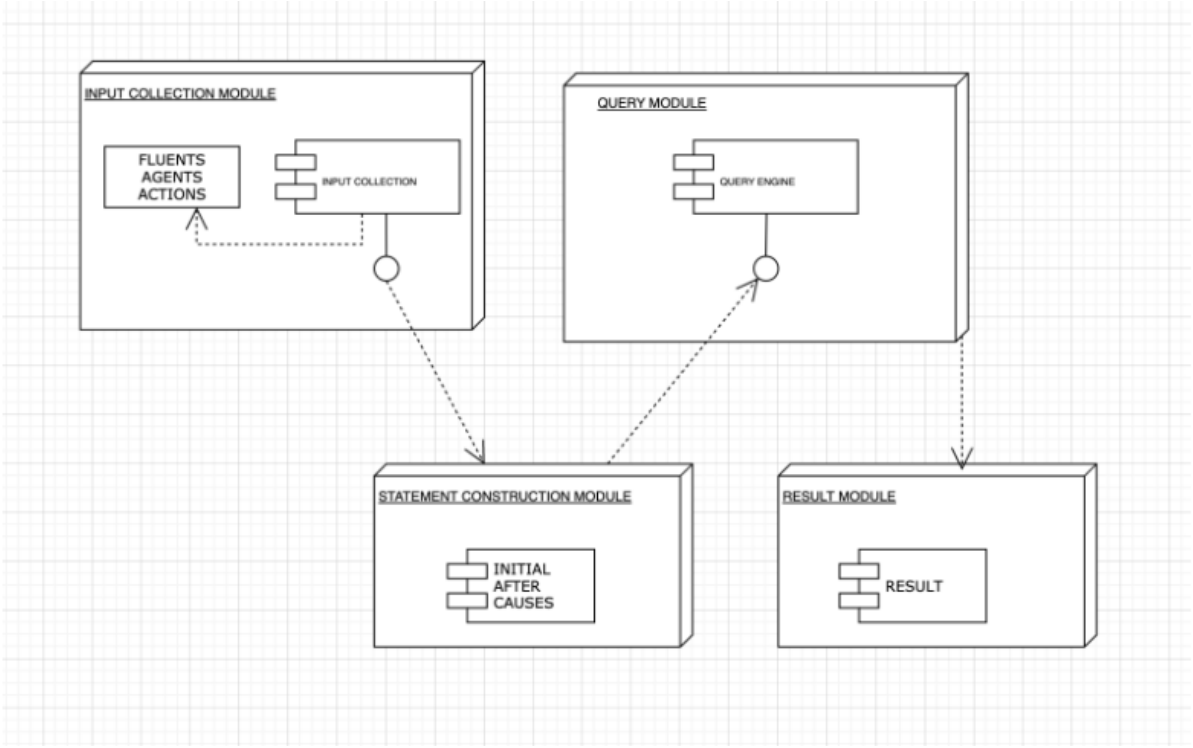
No	Class Name	Description
1	main.py	The entry point of the application. It is a class that connects the program to the user interface
2	domain.py	This method accepts the fluents, actions, agents, initial states and domain given by the user. This method also uses the accepted inputs to create a domain.pl file that is used as the main engine of the program.
3	domain.pl	This file holds the engine of the program and is created by the domain.py method. It is the knowledge base containing the facts and rules for this program. This file is what returns the results of a query whenever it is executed
4	query$_{t}ranslator.py$	This is a class that takes in the query for the program, translates it to the form understood by the program and outputs the result of the query
5	test.pl/test2.pl	These are python files for testing the program.
6	test.py/testt.py	These are prolog files for testing the program.
7	UI folder	This folder contains the Main.ui file which is an XMLfile for the user interface. it has all the information on how the UI operates.

2.2 ARCHITECTURE: MODULES

This project is divided into 4 modules, these modules include:

- **INPUT COLLECTION MODULE:** This module is the starting point of the software. In this module, the system accepts all fluents, agents and actions and stores them in the system. This module is implemented to have three stages. The first step request for each fluents from the user, the second step accepts all the actions that exists in the domain from the users and lastly all the agents that perform actions in the domain are accepted in the third step.
- **STATEMENT CONSTRUCTION MODULE:** Firstly, this module receives the data from the from the input collection module. It then accepts the initial states and provide sub systems that allow users to construct CAUSES and AFTER statements. This statements are created using drop-down buttons and check boxes.
- **QUERY MODULE:** This module is the engine of the system, as it performs the computation that provides result to the queries. In this module, the user provides the query to be executed by the program in the format specified in the user guide. This engine accepts two different query formats as required by the project. The module translate each query to an instruction understood by the program. The module also gives feedback on the consistency of the domains.
- **RESULT MODULE**: This module basically presents the conclusion from the computation by the QUERY MODULE. This module is the endpoint of the software.

The figure below is the architectural design of the system:



2.3 PROCESS FLOW (ALGORITHM)

The process flow is described in the following step:

1. Start
2. System accepts fluent(s) and stores in List
3. system accepts action(s) and stores in List
4. system accepts agent(s) and stores in List
5. system accepts INITIALLY value of fluents, if its multiple we separate with commas.
6. system accepts CAUSES statements with optional conditions
7. if user clicks ADD, statement constructed is added to List of Statement Objects
8. system accepts AFTER statements with optional conditions
9. if user clicks ADD, statement constructed is added to List of Statement Objects
10. System accept query
11. Query is being sent to query engine
12. Engine decide if query checks query format if its fluent holds or if its for agent involvement

13. System returns result to the user.

2.4 SOFTWARE DEVELOPMENT LIFE CYCLE: Iterative Enhancement Software Development Model)

For the purpose of this project, the Iterative Enhancement Model was used. The iterative-enhancement model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. In this model, the software is divided into several modules, which are incrementally developed and delivered. First, the development team developed the core module of the system and then it is later refined into increasing levels of capability of adding new functionalities in successive versions based on recommendation from Dr. Anna. Each linear sequence produces a deliverable increment of the software. This model was chosen for the following reasons:

1. its capability to provide user feedback on the developed aspect of the development.
2. it provides encouragement which enhances subsequent development.
3. it divides the project into smaller projects which is easier to manage and control.
4. early delivery of the core module.

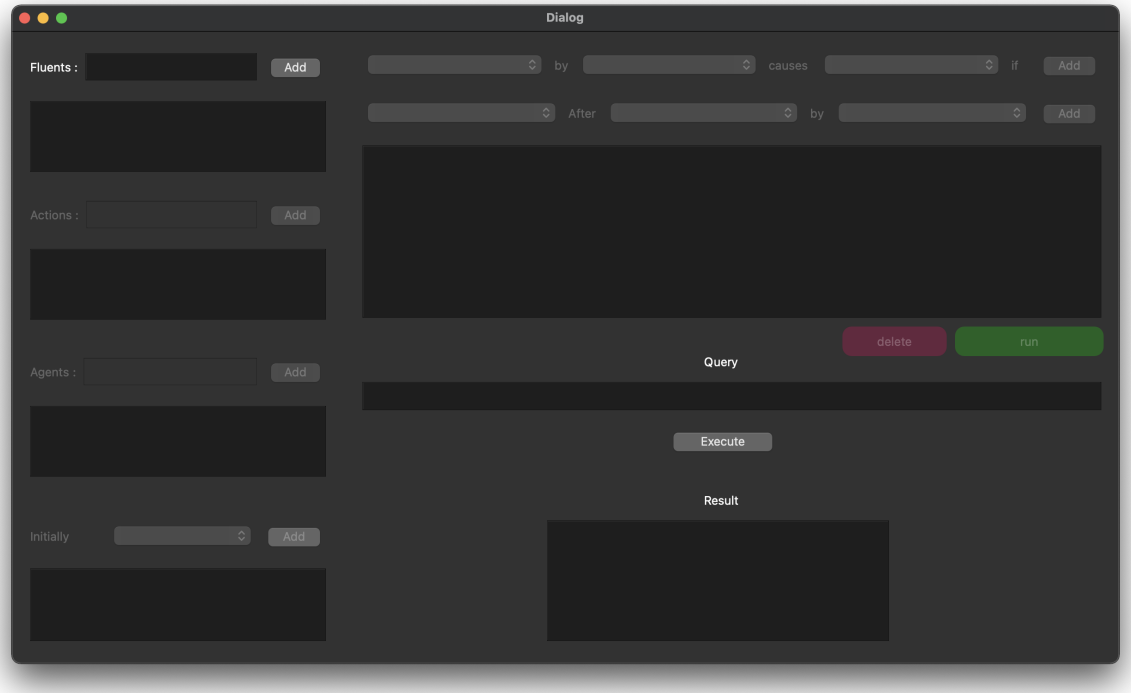
2.5 DATA STRUCTURE

For this project, Prolog terms, Python List and Dictionary Data Structure were mainly used. Prolog is a logic programming language which as its roots in predicate logic,it is associated with artificial intelligence and computational linguistics. As a predicate logic, it allows the use of quantified variables over non logical objects using sentences that contains variables. For example: "There exists X such that X is a James and X is a man". Prolog is also a declarative language therefore the program is written using relations such as facts and rules and computations are initiated using queries over these rules and facts. It enables Backtracking. The data structure used are prolog terms like compound terms, variables, strings and list. These data structures are what make up the facts and rules that is queried by the python program. The data structures used in python are mostly lists, dictionaries, sets and tuples. These data types were used at different occasions based on their properties. The List Data type defines a sequential set of elements to which you can add new elements and remove or change existing ones. The Dictionary data structure is a key, value pair. It pairs a key to its values which can be a list, set or tuple. It is ordered, mutable but does not allow duplicates. A Set datatype accepts unique values, they are not ordered, immutable and can take in new items. While a Tuple is a datatype that stores data in an ordered and immutable way. List, dictionaries and tuples are data types that can be to searched, and accessed via indexes.

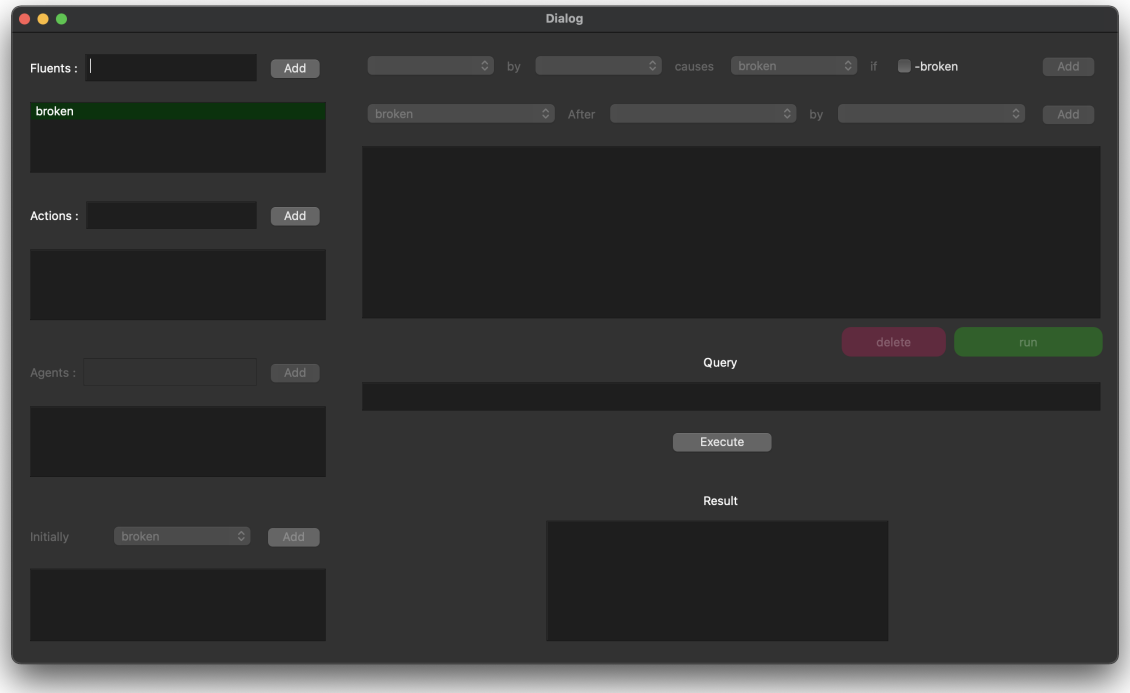
Chapter 3

User Guide

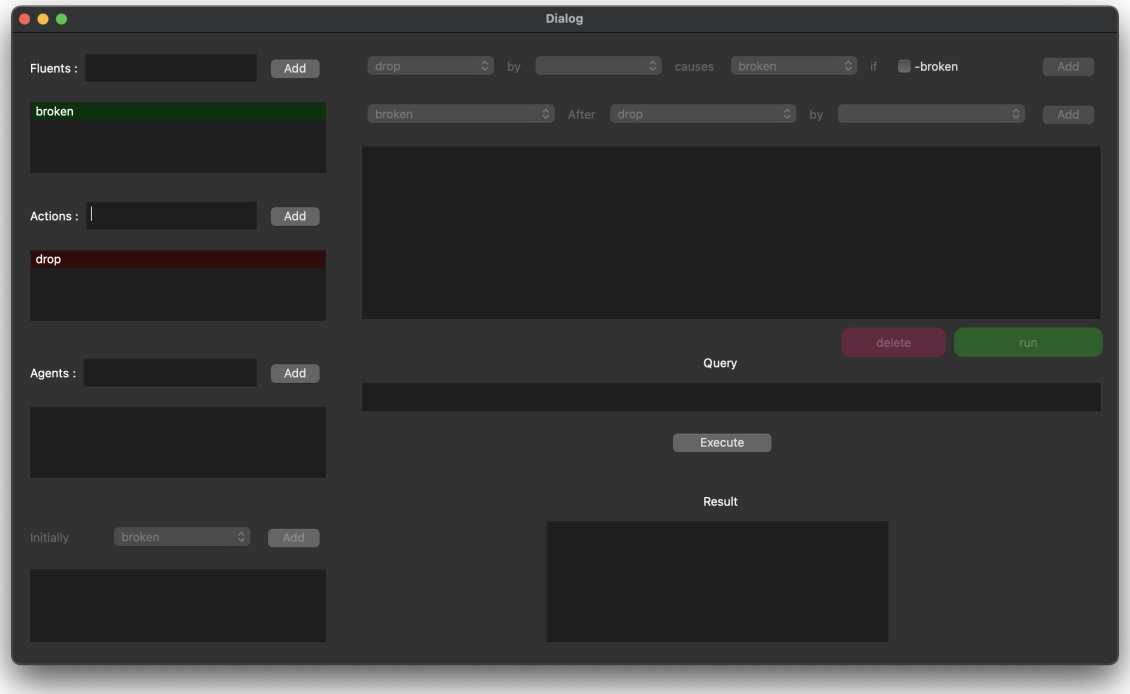
Initially it looks like this:

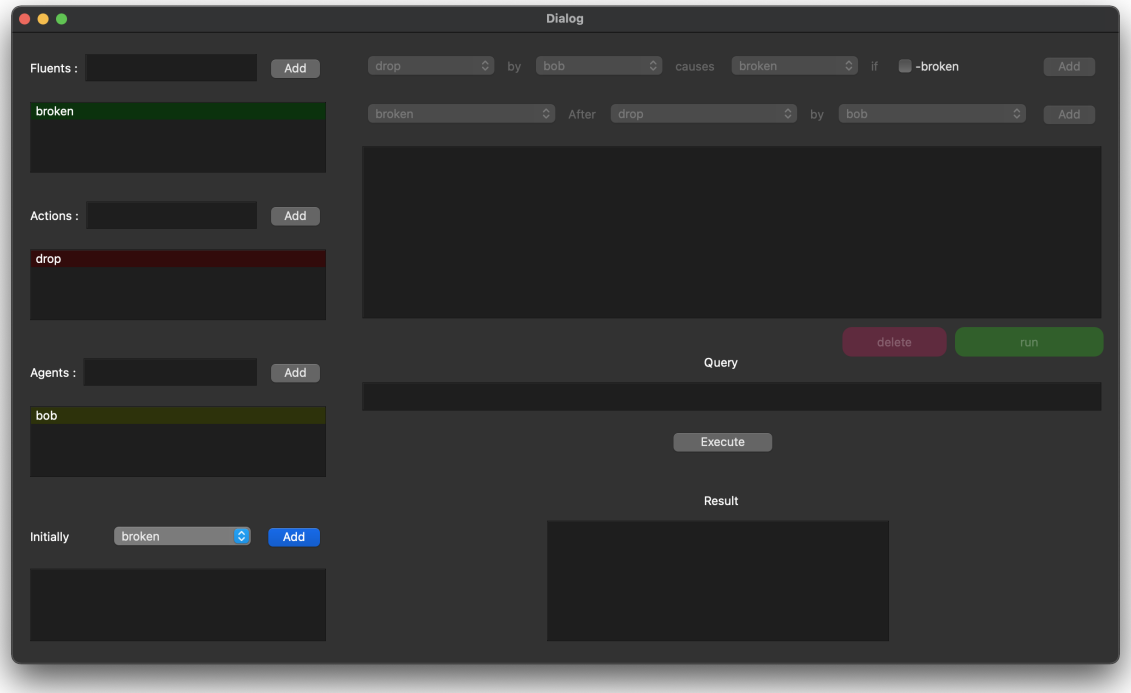


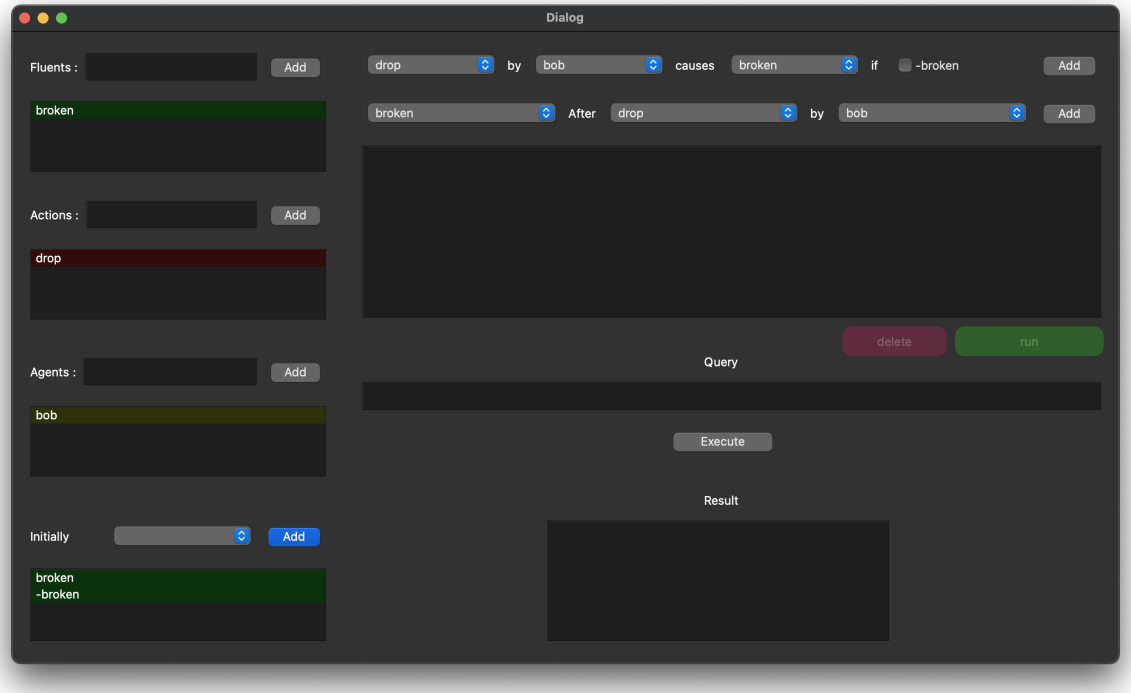
This program only has one window. All parts of the program are disabled except the option for adding fluents. After the fluents are added the actions window is enabled and actions are accepted as input into the system.



The input of actions enables the agents window. Now the agents window accepts input. After the input of agents, the initially state is enabled. Here a drop down of all the possible fluent(s) are available, and the fluents in the initial states are added.



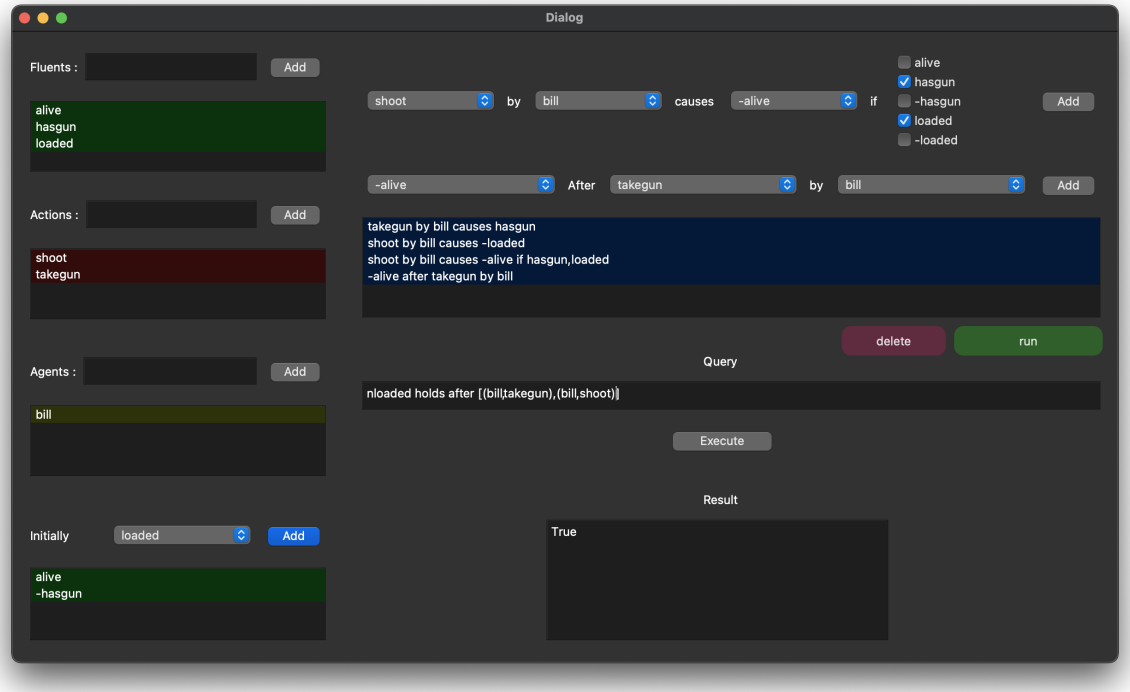




After the initial states are added, the statement construction module which consists the drop down of all the possible fluent(s), action(s), agent(s) and causes to enable a seamless statement generation is enabled as seen in the image above. After constructing the statements, the run button creates the domain, the facts, and rules used in the Prolog program. This Prolog program is queried to get the results
Below is the format of the query accepted by the program

- Fluent holds after [(agent,action),(agent2,action)]
- nFluent holds after (n represent not) [(agent,action),(agent2,action)]
- nf holds after [(jim,action),(bill,action)]
- AGENT is involved in [(agent,action),(agent2,action)]
- Bill is involved in [(jim,action),(bill,action)]

The image is an example of the final state having results for the queries.



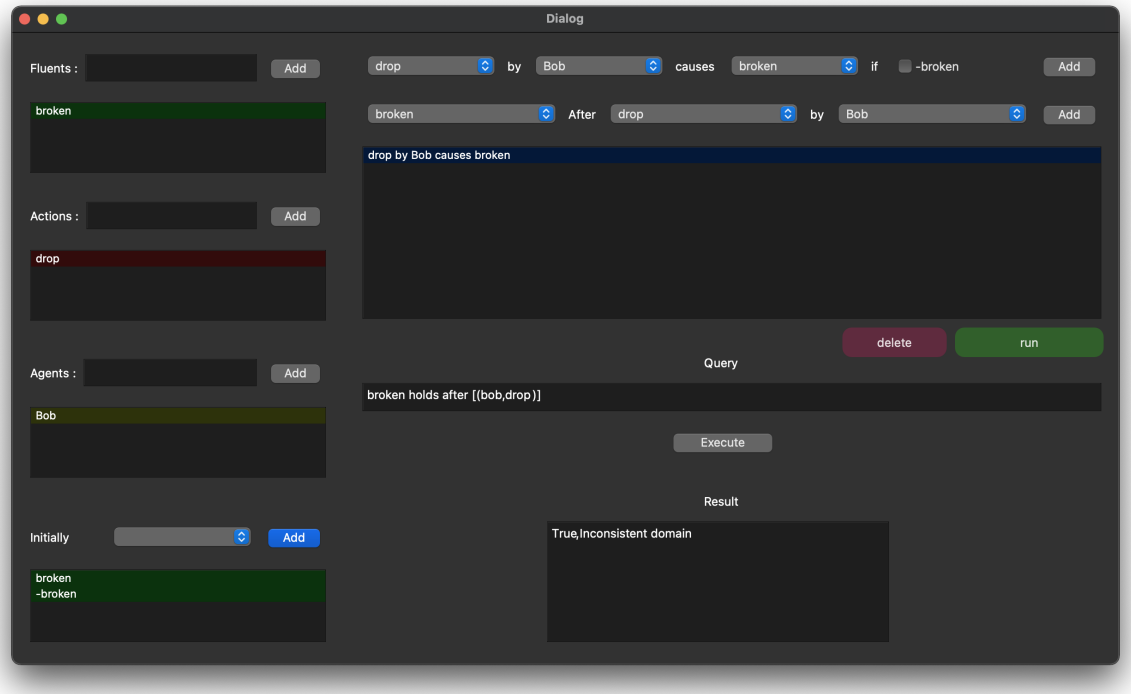
Chapter 4

Testing

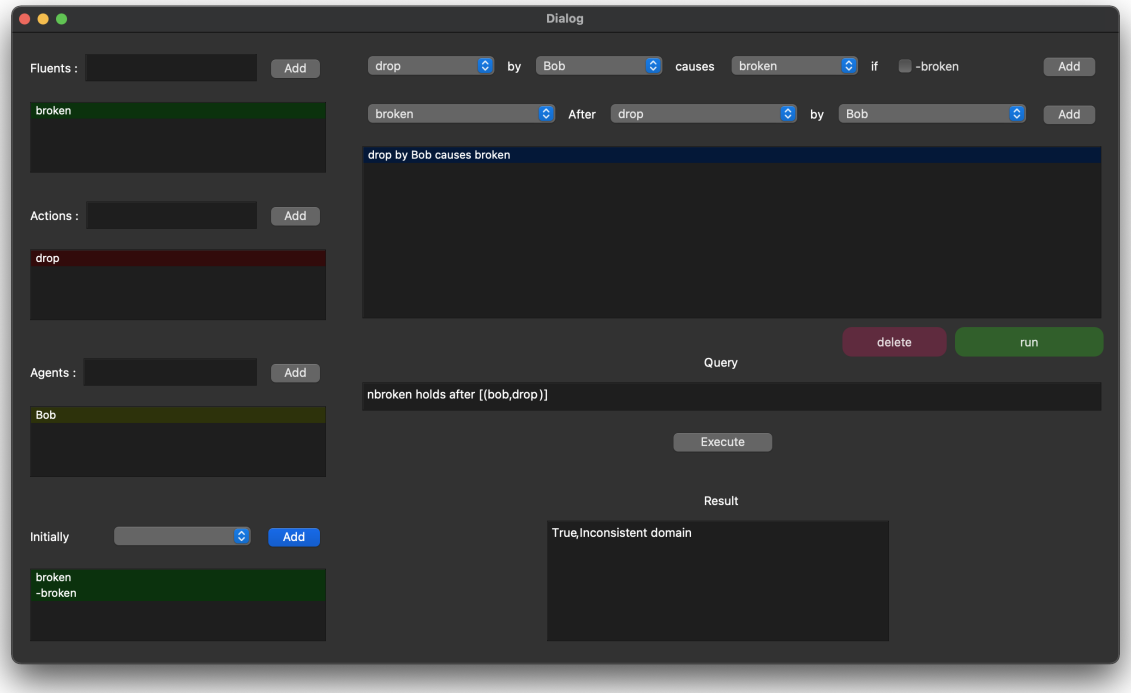
TEST CASE 1

Initially broken
Initially -broken
DROP by Bob causes broken

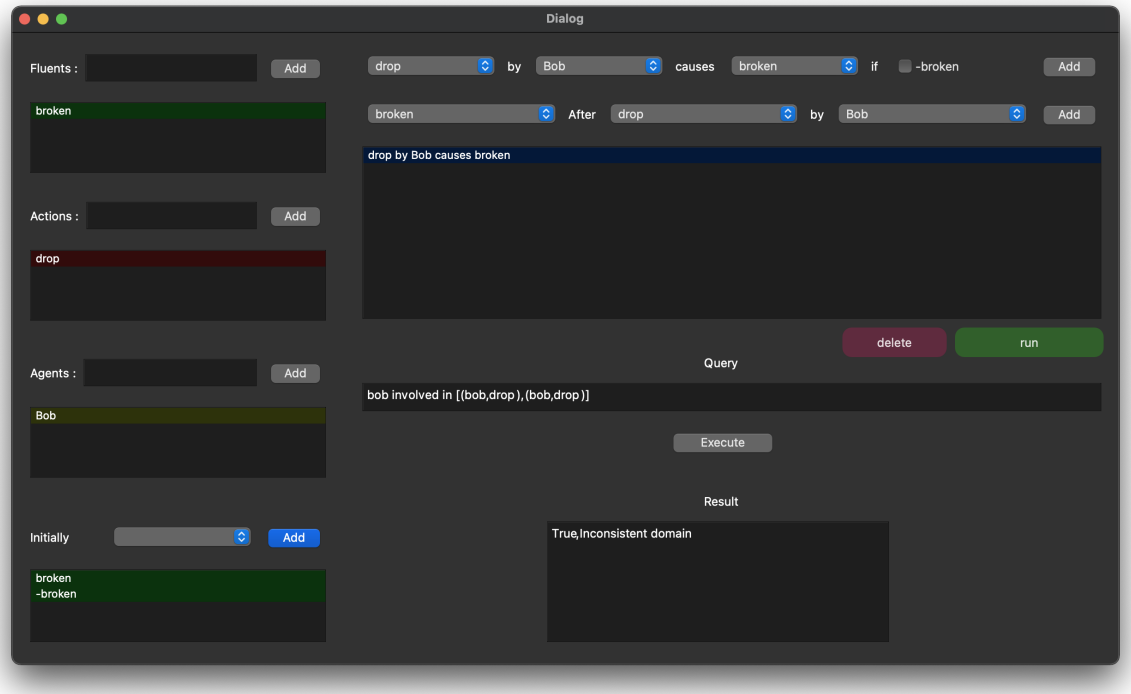
1. Query: broken holds after (DROP, Bob). Answer: TRUE.



2. Query: -broken holds after (DROP, Bob). Answer: TRUE.



3. Query: Bob involved in ((Drop, Bob), (Drop, Bob)). Answer: TRUE.

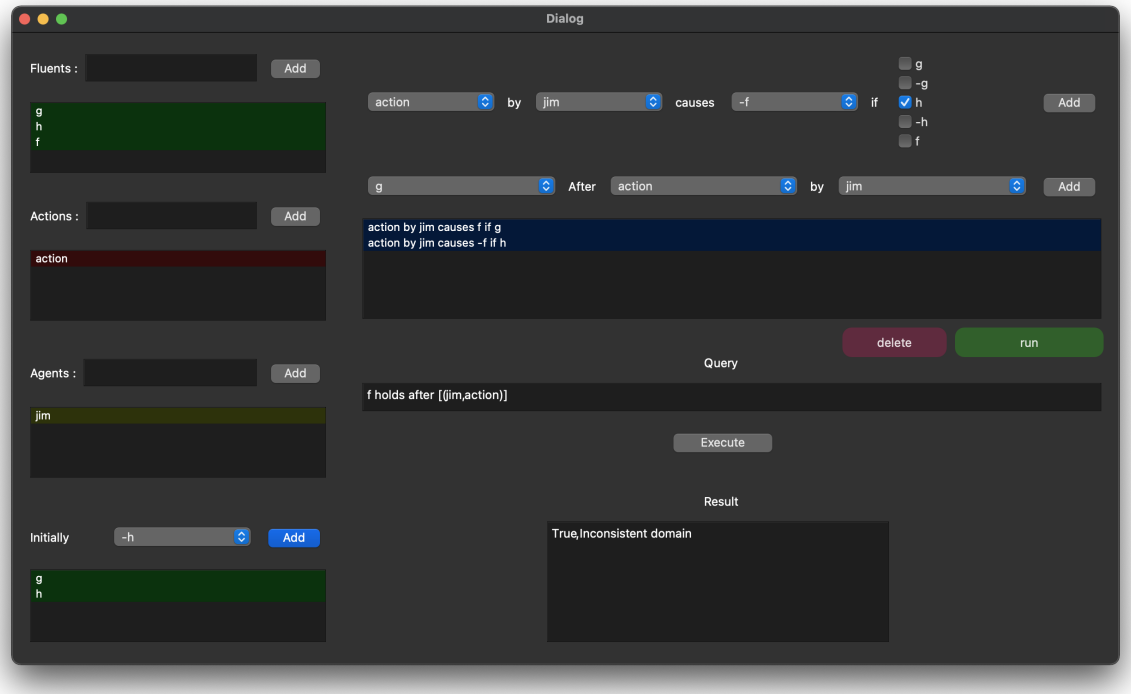


Conclusion
Note that the above domain is inconsistent since There is no state satisfying both broken and -broken.

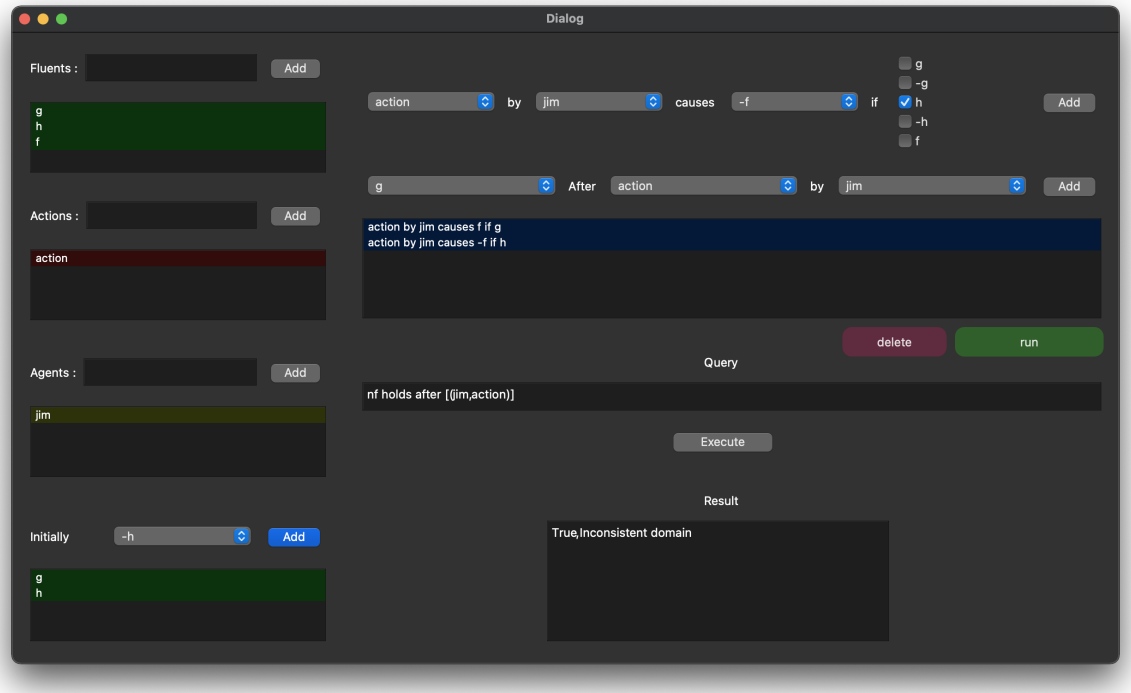
TEST CASE 2

Initially g
Initially h
ACTION by Jim causes f if g
ACTION by Jim causes -f if h
The above domain is inconsistent. There is no transition function defined for ACTION, Jim, and the initial state is satisfying f and g.

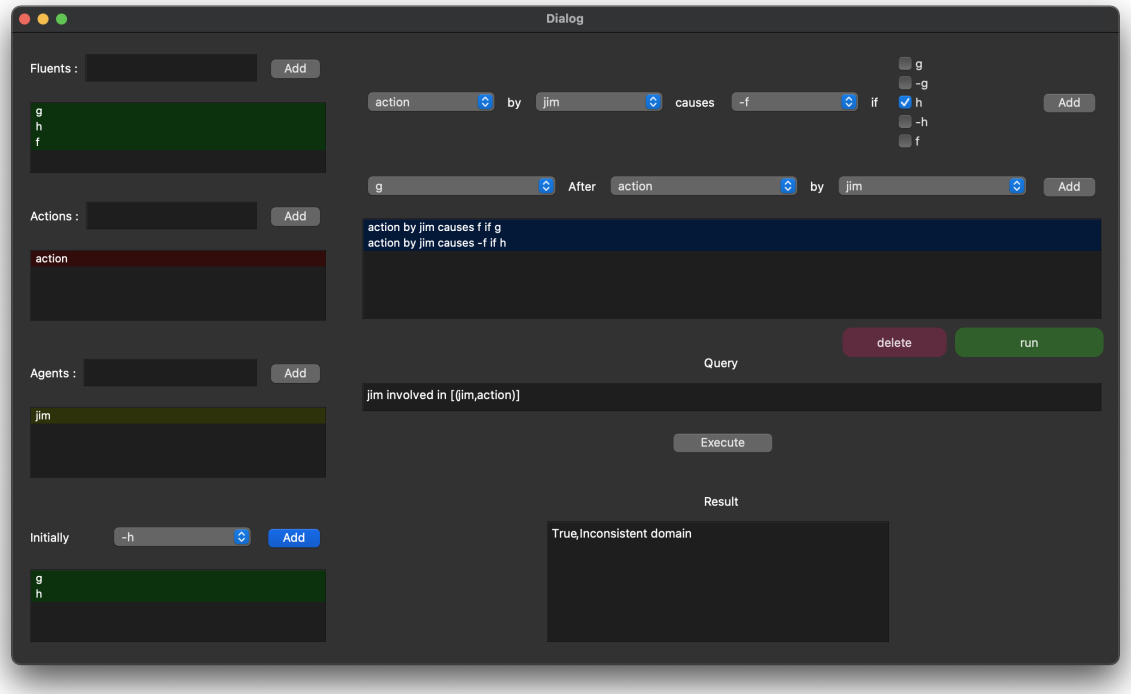
1. Query: f holds after (ACTION, Jim). Answer: TRUE.



2. Query: -f holds after (ACTION, Jim). Answer: TRUE.



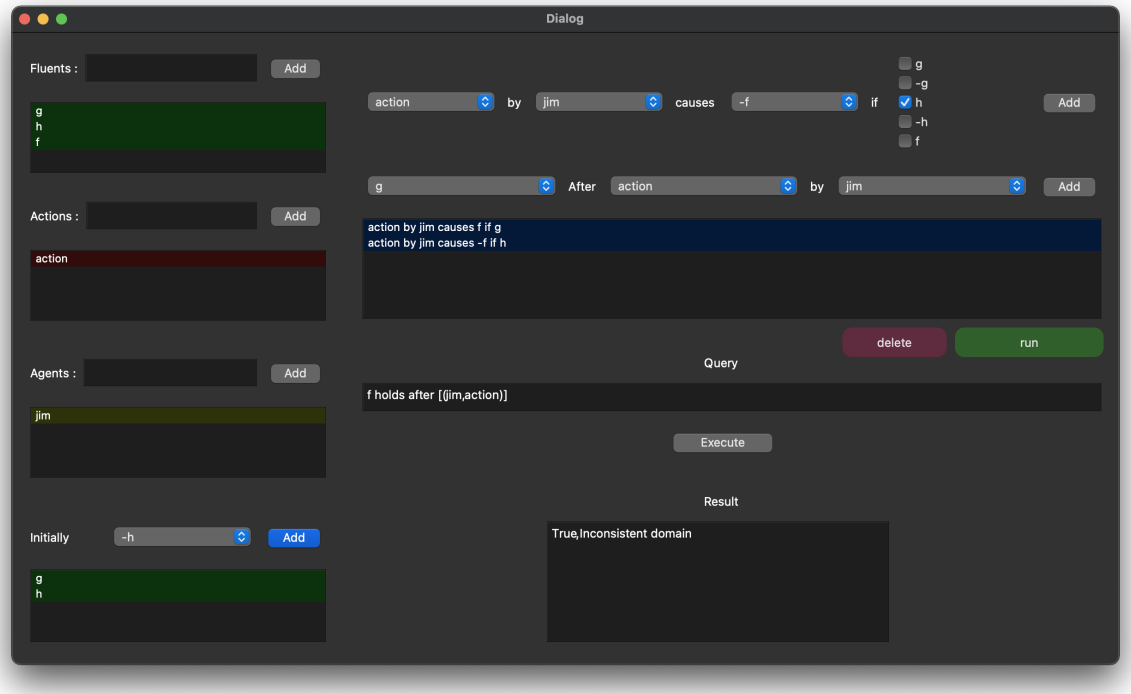
3. Query: Jim involved in (ACTION, Jim). Answer: TRUE.



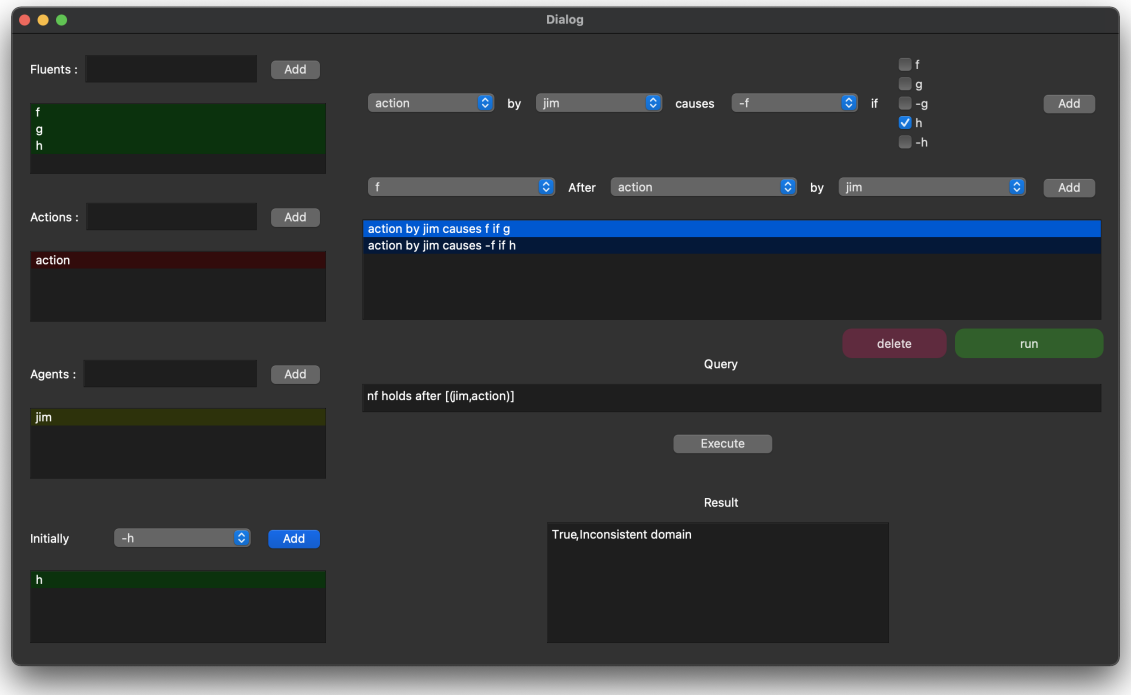
TEST CASE 3
Initially h
ACTION by Jim causes f if g

ACTION by Jim causes -f if h

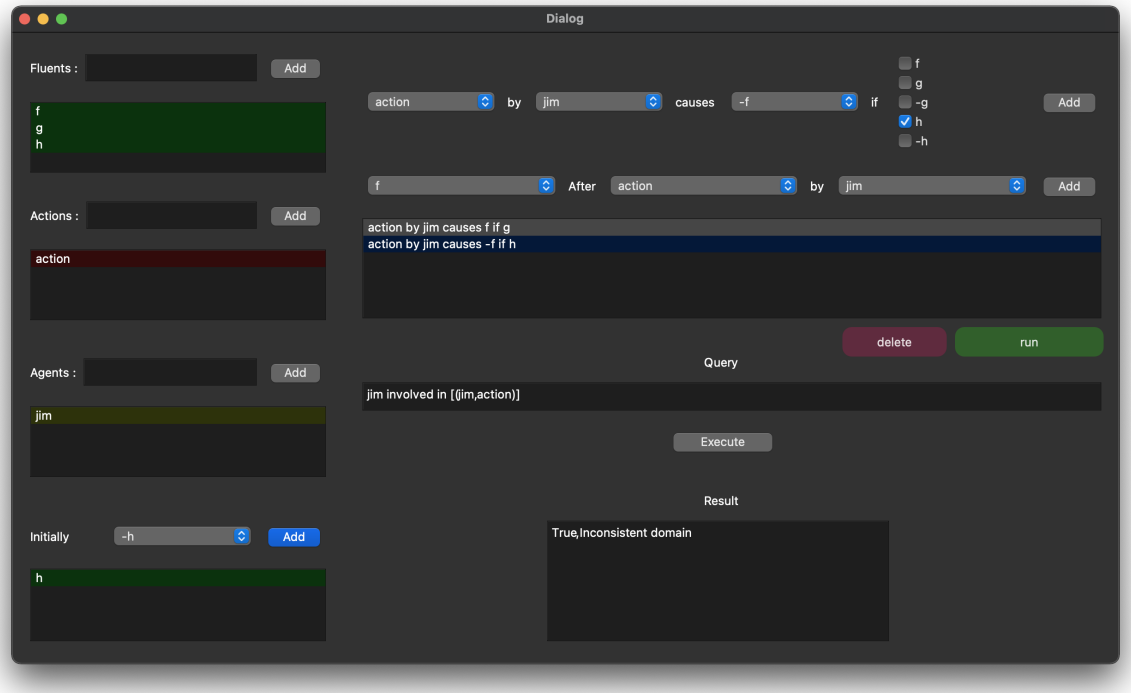
1. Query: f holds after (ACTION, Jim). Answer: YES.



2. Query: -f holds after (ACTION, Jim). Answer: TRUE.



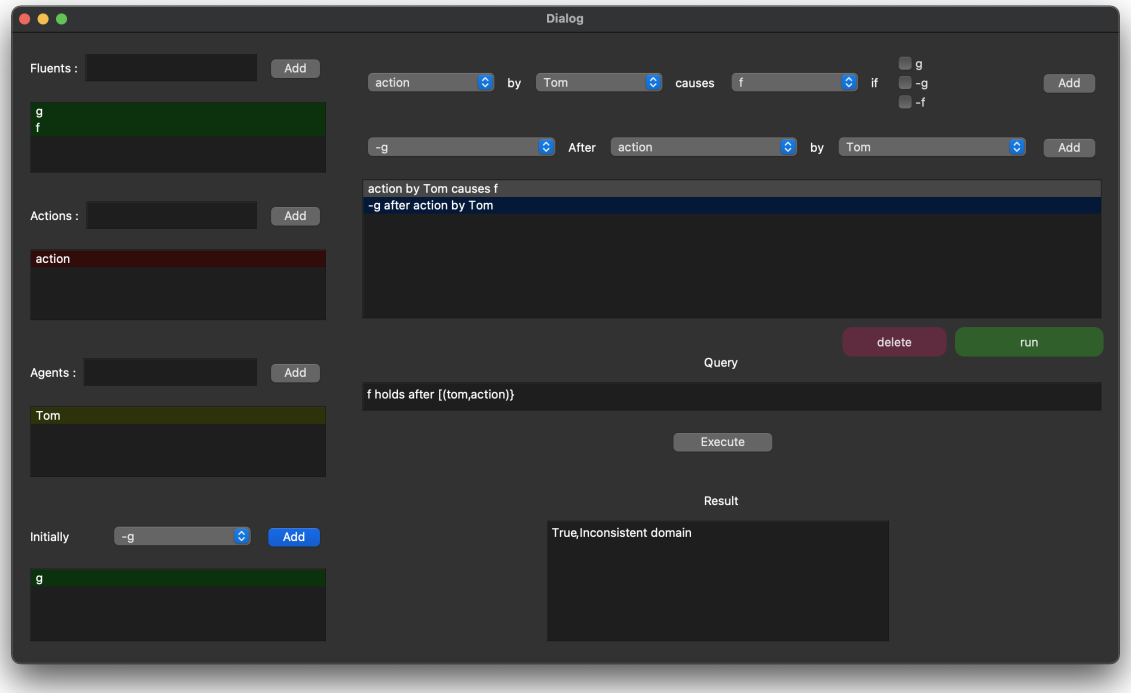
3. Query: Jim involved in (ACTION, Jim). Answer: TRUE.



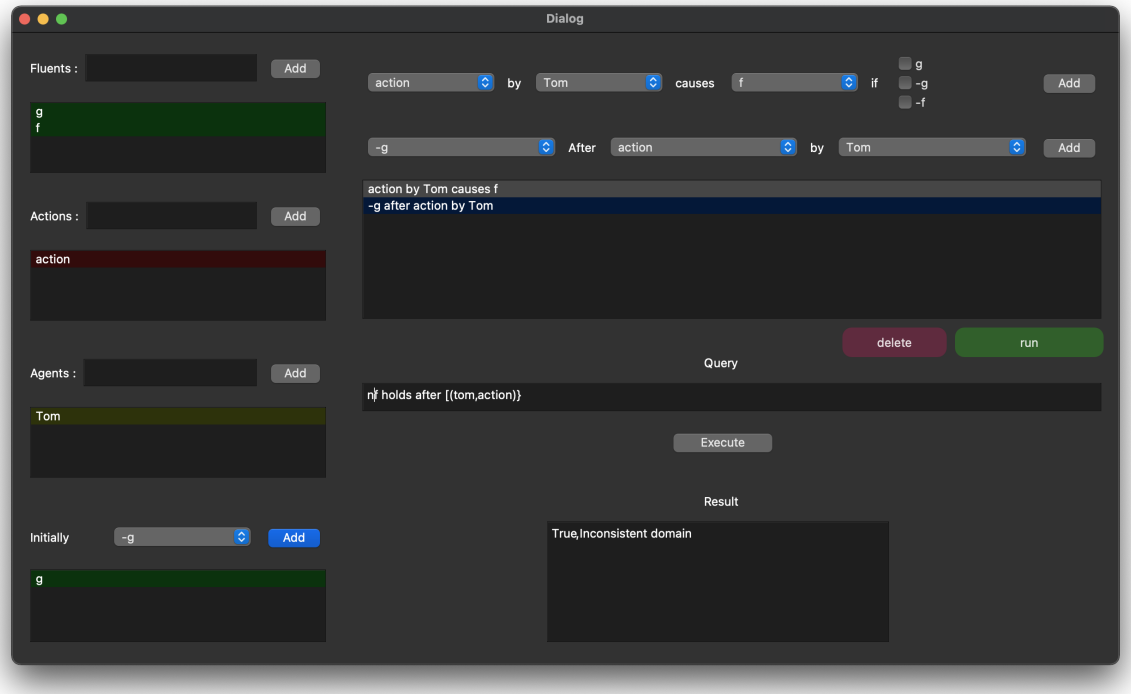
Conclusion
The above domain is inconsistent. There is no transition function defined for ACTION, Jim, and the state satisfying h, f and g.

TEST CASE 4
Initially g
ACTION by Tom causes f;
-g after ACTION by Tom

1. Query: f holds after (ACTION, Tom). Answer: TRUE.



2. Query: -f holds after (ACTION, Tom). Answer: TRUE.

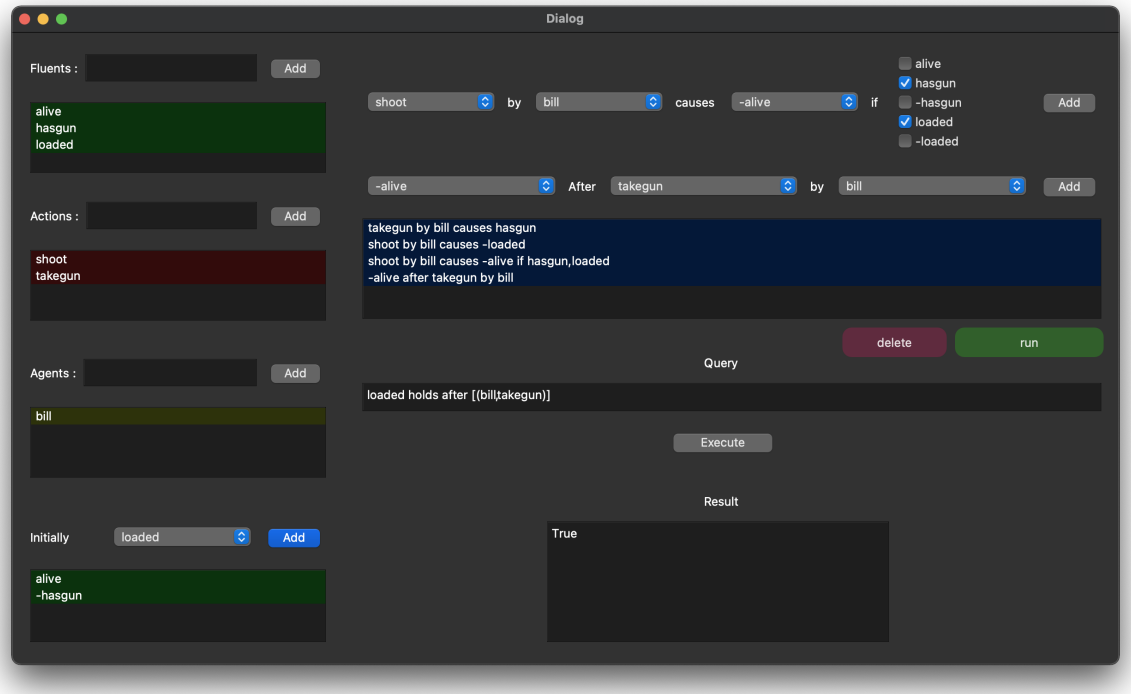


Conclusion
The above domain is inconsistent. Due to inertia law. There is no transition function defined for ACTION, tom, and the change of state from g to -g.

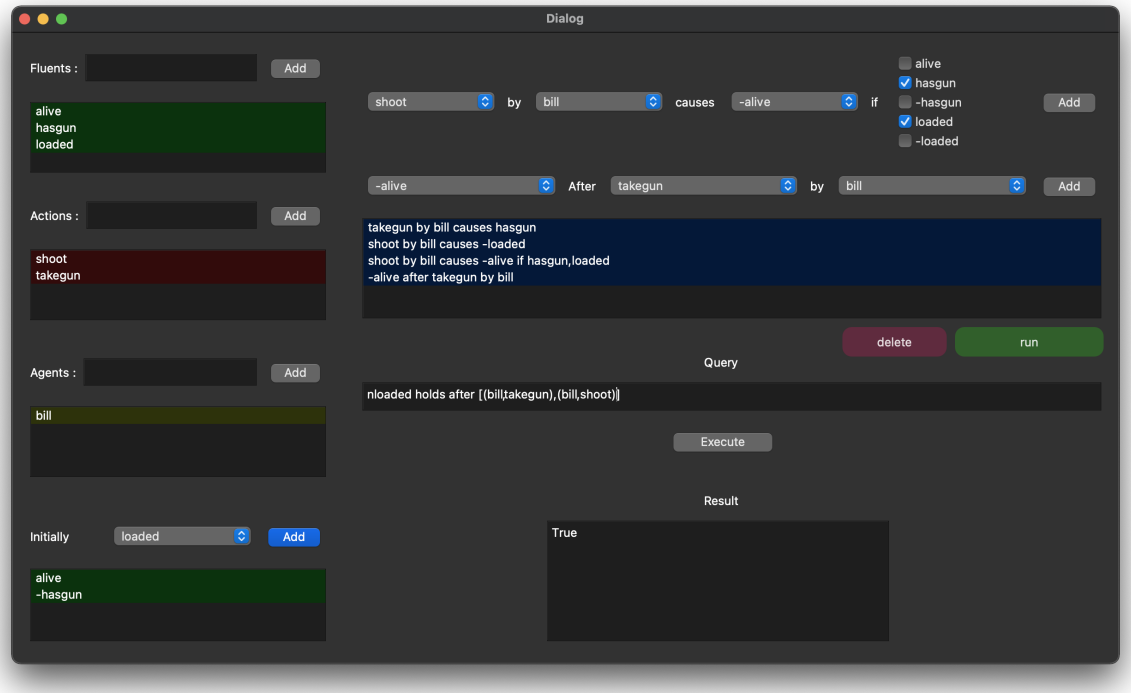
TEST CASE 5

Initially alive;
Initially -hasGun;
TAKEGUN by Bill causes hasGun;
SHOOT by Bill causes -loaded;
SHOOT by Bill causes -alive if loaded ∧hasGun;
-alive after TAKEGUN by Bill;

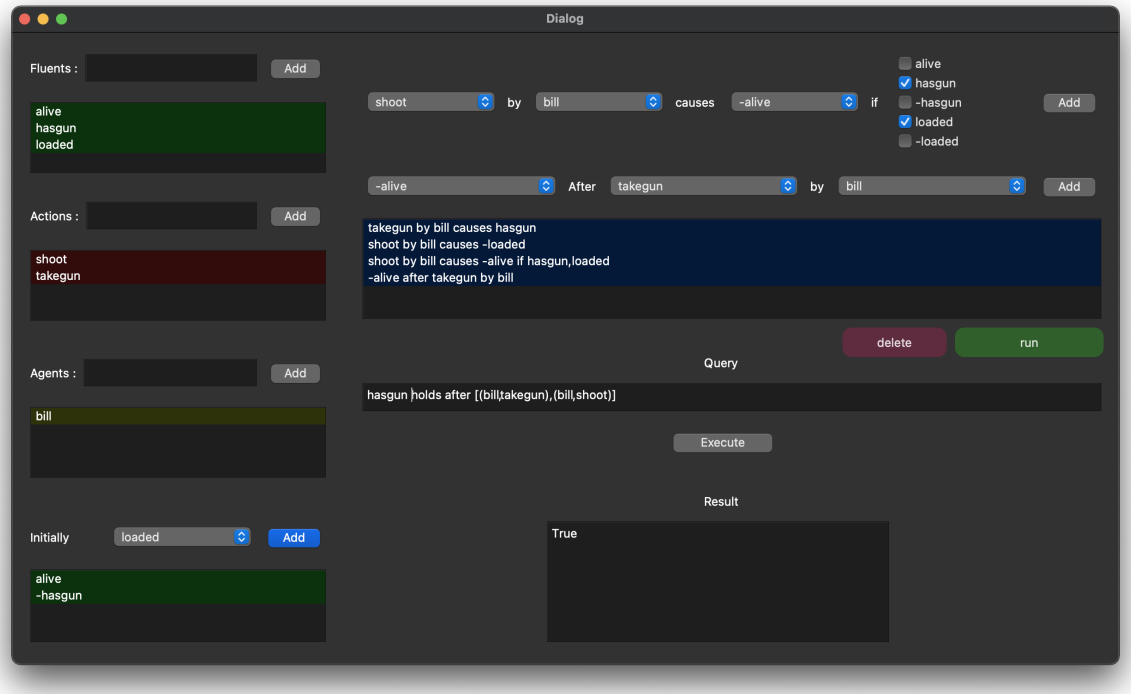
1. Query: loaded holds after (TAKEGUN, Bill). Answer: TRUE.



2. Query: -loaded holds after ((TAKEGUN, Bill), (SHOOT, Bill)). Answer: TRUE.



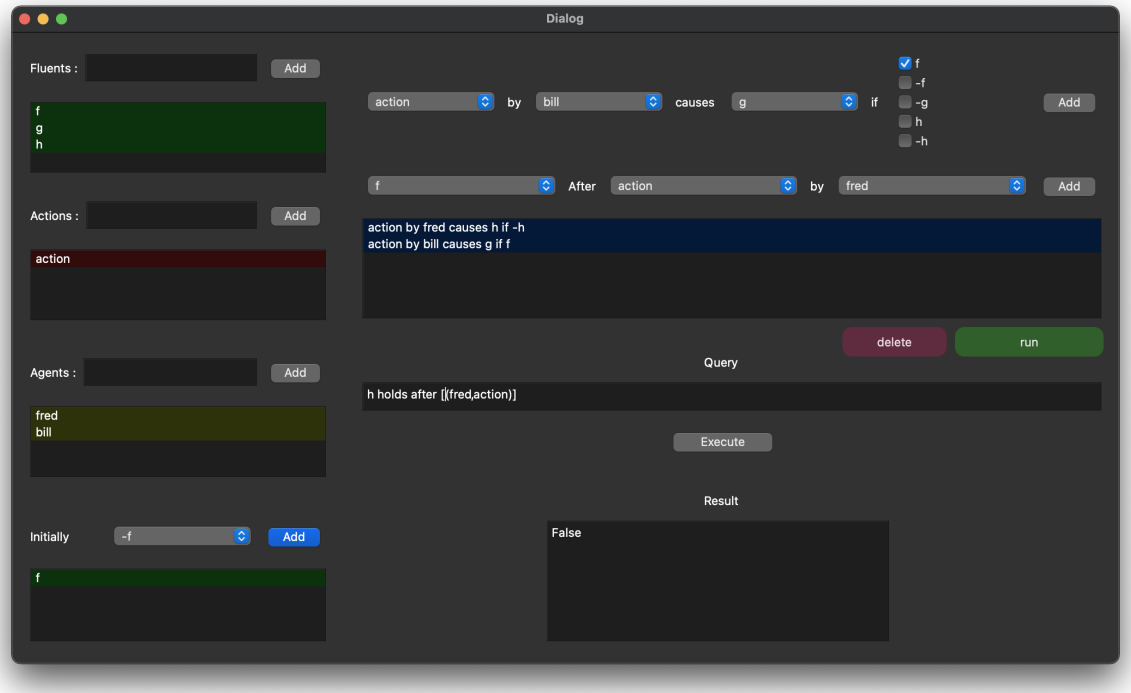
3. Query: hasGun holds after ((TAKEGUN, Bill), (SHOOT, Bill)). Answer: TRUE.



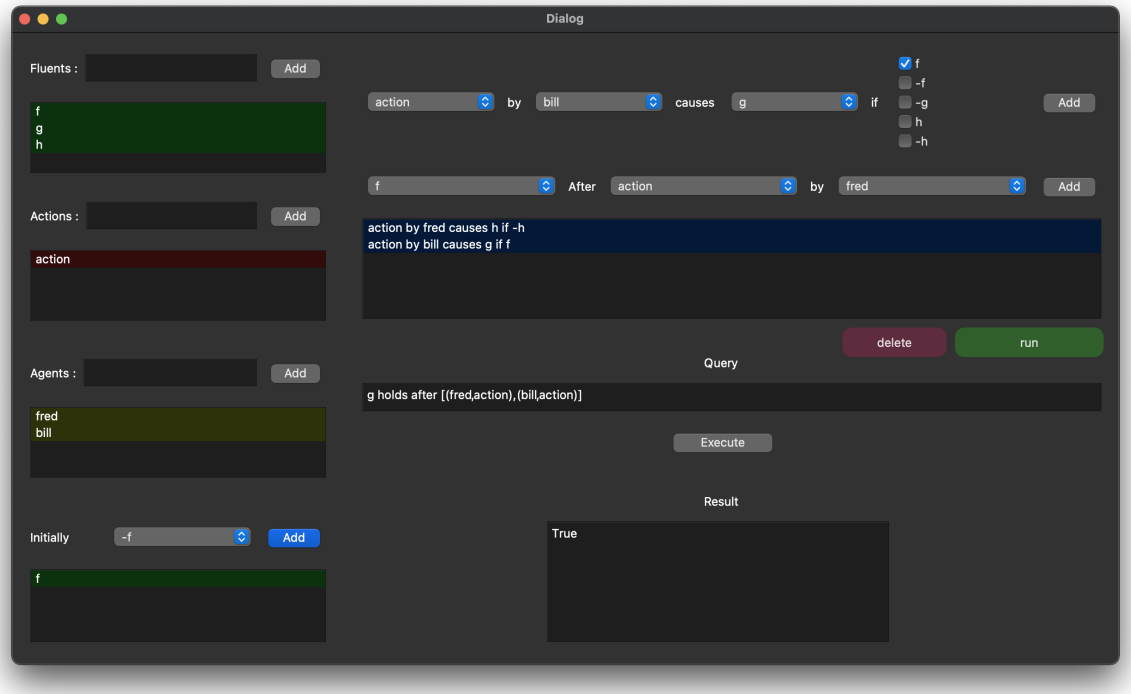
TEST CASE 6
Initially f;
ACTION by Bill causes g if f;
ACTION by Fred causes h if -h;

The above domain is inconsistent. There is no transition function defined for ACTION, Jim, and the initial state is satisfying f and g.

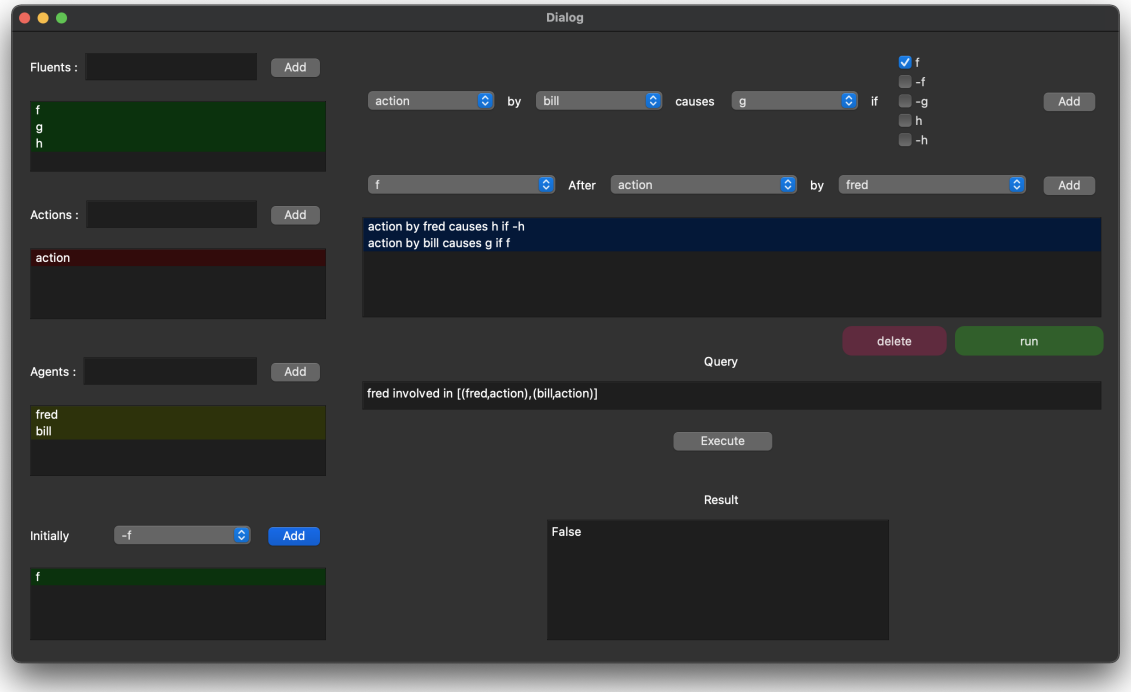
1. Query: h holds after (ACTION, Fred). Answer: FALSE.



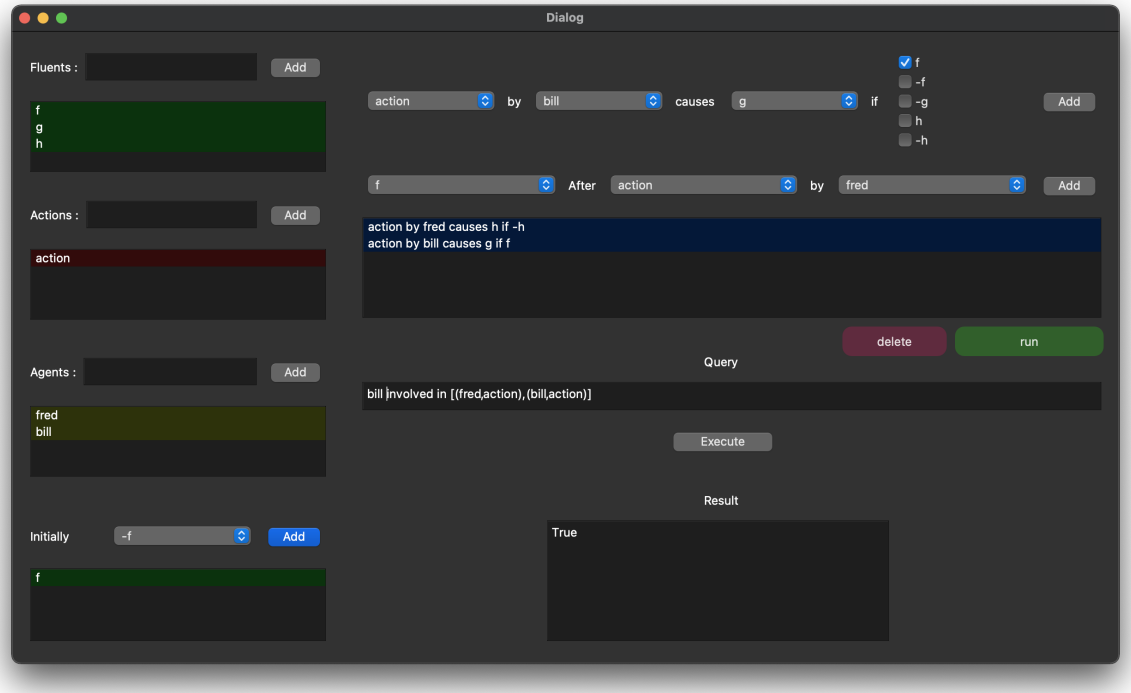
2. Query: g holds after ((ACTION, Fred), (ACTION, Bill)). Answer: TRUE.



3. Query: Fred involved in ((ACTION, Fred), (ACTION, Bill)). Answer: FALSE.



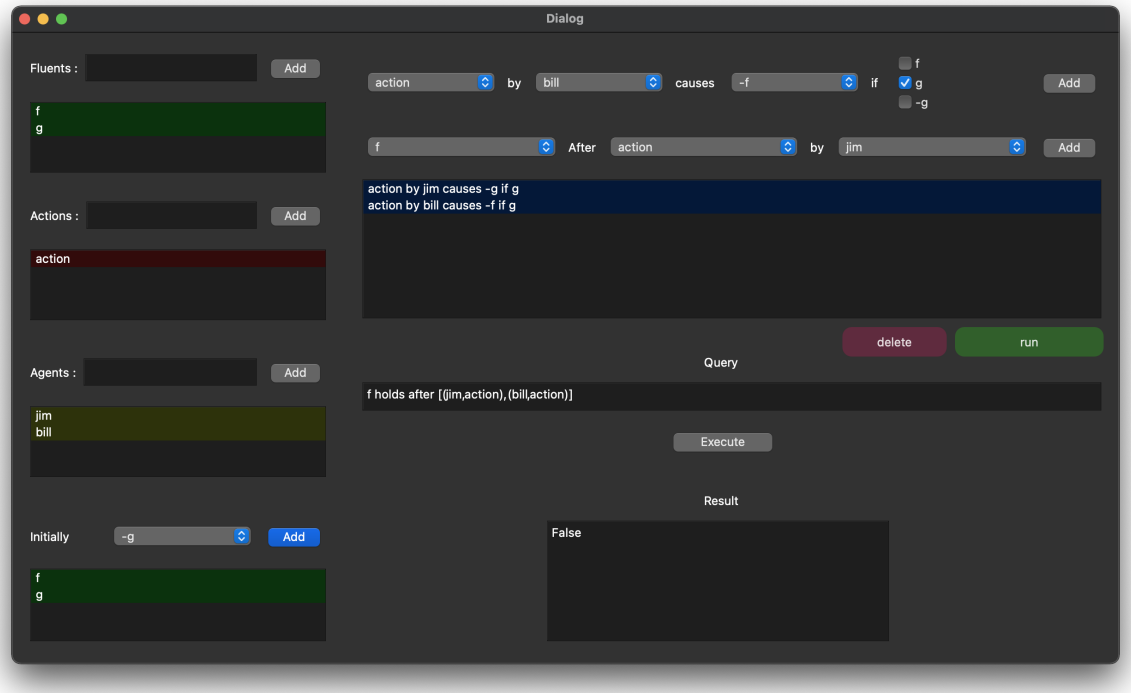
4. Query: Bill involved in ((ACTION, Fred), (ACTION, Bill)). Answer: TRUE.



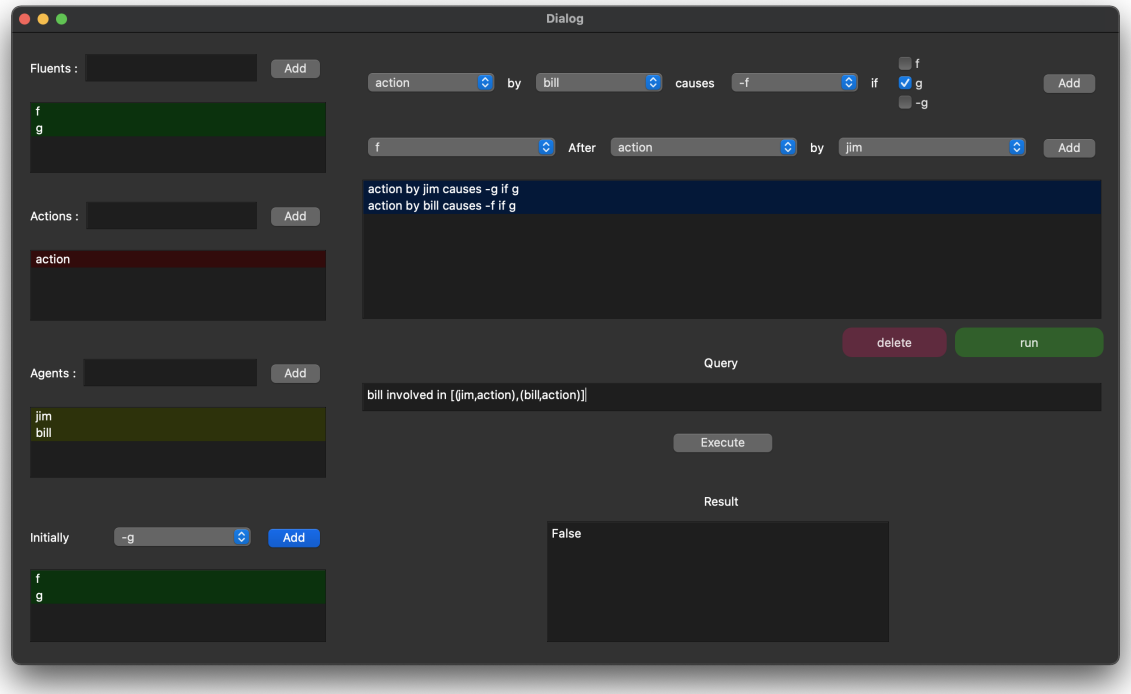
TEST CASE 7
Initially f;
Initially g;

ACTION by Jim causes -g if g
ACTION by Bill causes -f if g

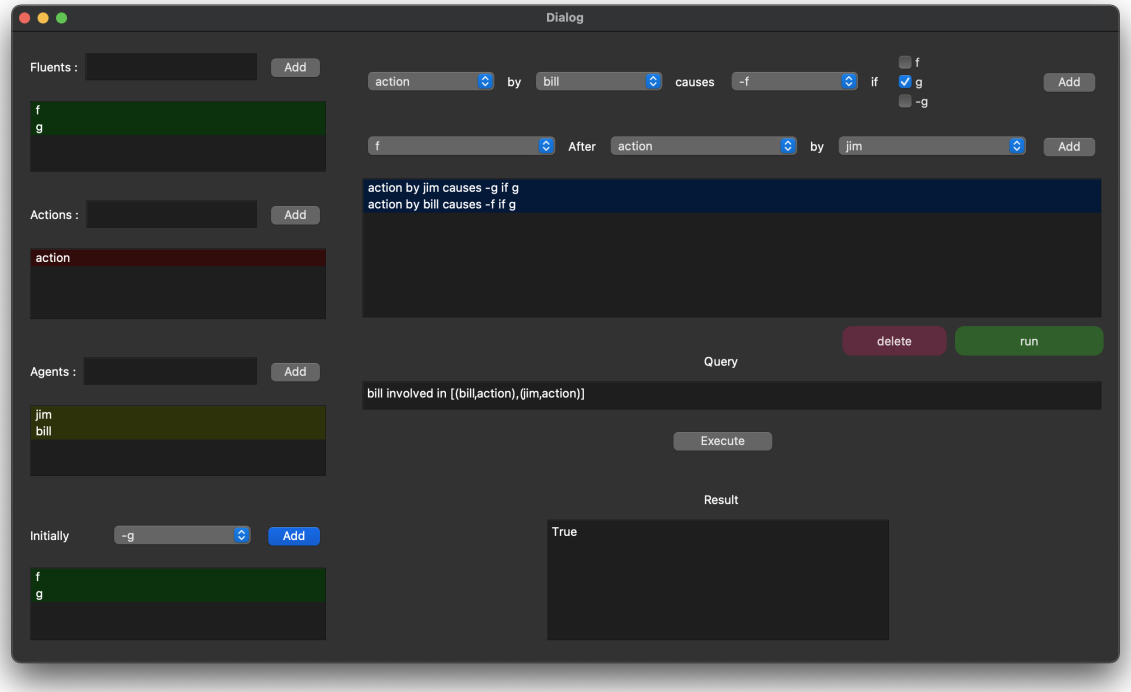
1. Query: f holds after ((ACTION, Jim), (ACTION, Bill)). Answer: FALSE.



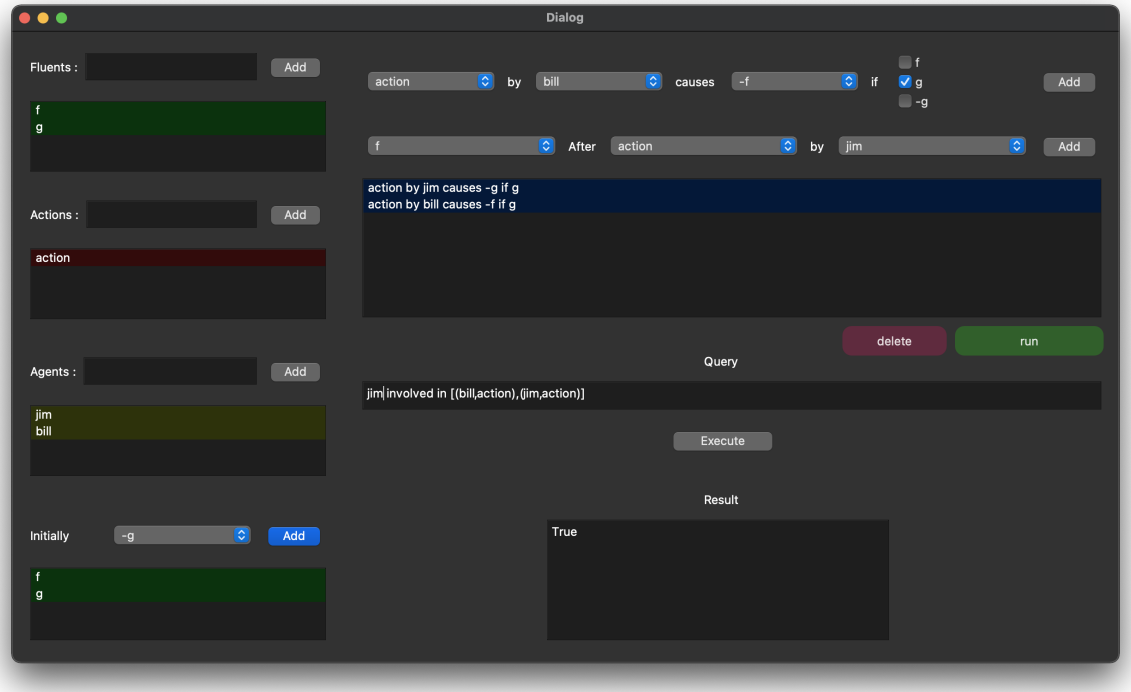
2. Query: Bill is involved in ((ACTION, Jim), (ACTION, Bill)). Answer: FALSE.



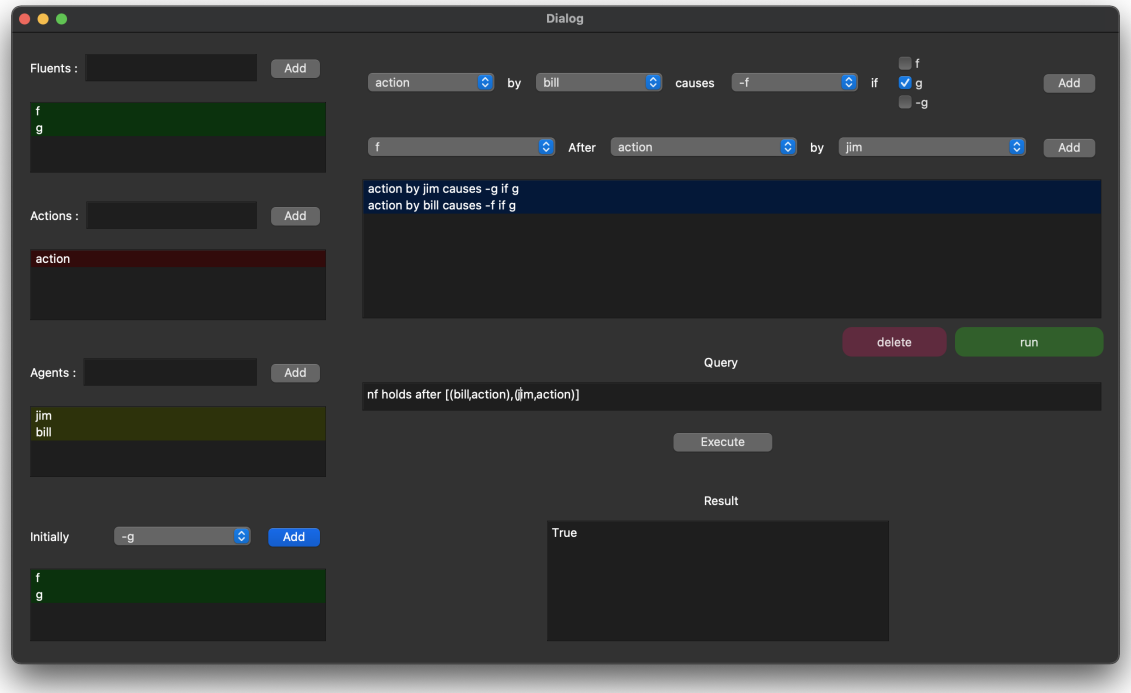
3. Query: Bill is involved in ((ACTION, Bill), (ACTION, Jim)). Answer: TRUE.



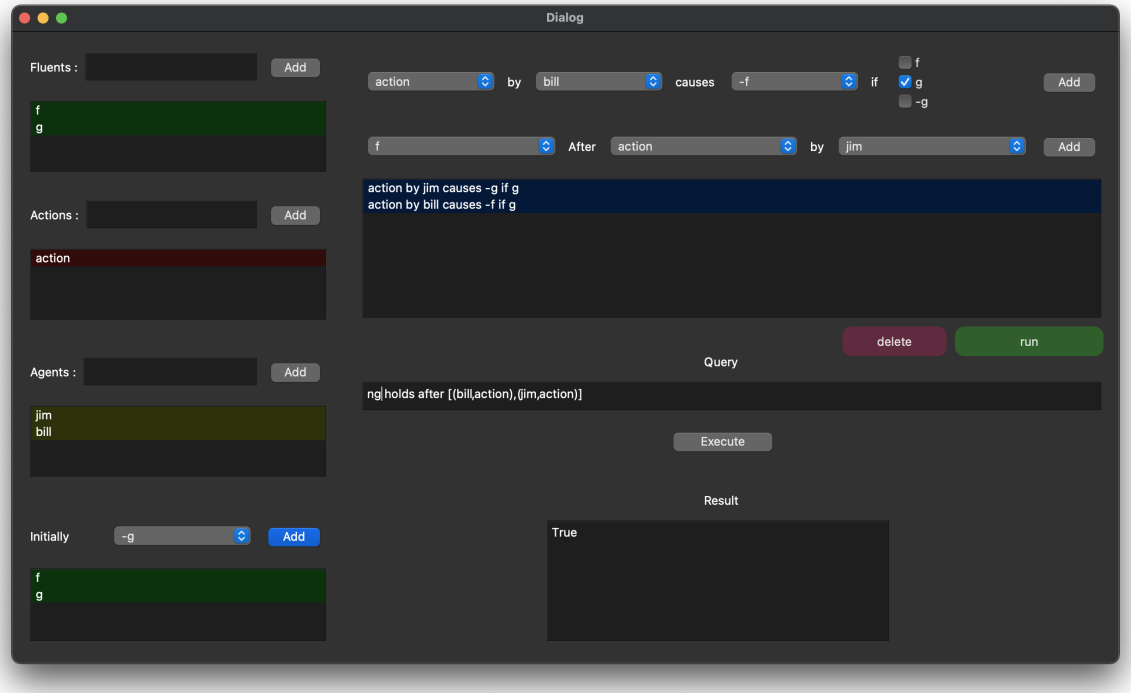
4. Query: Jim is involved in ((ACTION, Bill), (ACTION, Jim)). Answer: TRUE.



5. Query: -f holds after ((ACTION, Bill), (ACTION, Jim)). Answer: TRUE.



6. Query: -g holds after ((ACTION, Bill), (ACTION, Jim)). Answer: TRUE.



Chapter 5

CONTRIBUTIONS

Name	Contribution in Percentage
Adeyemi Adedayo Tolulope	

5.1 THEORETICAL PART

Ajewole Adedamola Jude	Hilal Saim	Adeyemi Adedayo Tolulope
Language Syntax	Query Language (Syntax and Semantics)	2nd Example
Language Model	Part of Language Semantics	Part of Language Semantics
Query Language (Syntax and Semantics)		Part of Language Model
1st Example		

5.2 IMPLEMENTATION

Adeyemi Adedayo TolulopeImplementation System analysis

5.3 TESTING

Adeyemi Adedayo Tolulope

- Provided test report

5.4 TECHNICAL DOCUMENTATION

Adeyemi Adedayo Tolulope

- Data Structure
- Project Structure/ Class Description.

- User Guide.
- Process Flow.
- Architecture.
- Software Development Life Cycle.