UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

– Faculty of Computer Science –

Theorem prover with Among Us

Documentation for the Second Assignment of the AI Laboratory:

by

Braica Patricia

Course Examiner	:	Prof. Dr.-Ing. Groza Adrian
Laboratory Examiner	:	Prof. -Ing. Lecu Alexandru

CONTENTS

Part I

THESIS

1 PROJECT OVERVIEW

The **Among Us with Prover9** project integrates the deductive reasoning capabilities of the **Prover9 automated theorem prover** with a **customized version of the popular multiplayer game, Among Us**. This innovative project explores the application of formal logic and automated reasoning in the context of interactive, deduction-based gameplay.

Project Objective: The primary aim is to simulate the logical reasoning process of determining impostors in a murder-mystery game setting. By leveraging Prover9, players and AI agents analyze evidence, validate claims, and deduce the identity of impostors through systematic reasoning.

Key Features:

- Integration of Prover9 to generate and verify logical proofs from in-game events and accusations.
- A modified game loop that incorporates deductive reasoning as a core mechanic.
- Tools for players to interact with Prover9, inputting hypotheses and receiving logical conclusions in real time.
- Dynamic and logical game narratives that adapt based on player and Prover9 interactions.

Usage Instructions:

- **To start the game:**

```
python main.py
```

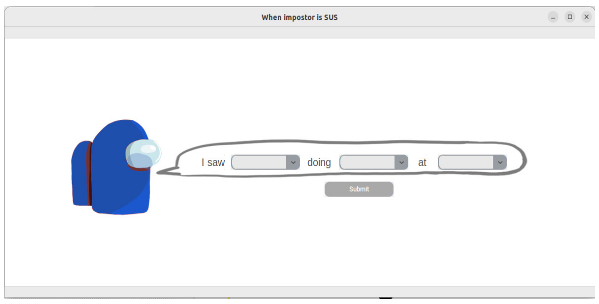


Figure 1.1: A gameplay scenario illustrating Prover9 deductive reasoning.

2 CONTENTS OF THE AMOGUS_DYNAMIC.IN FILE

The `amogus_dynamic.in` file is an input file for Prover9 and Mace4 that defines the logical rules and constraints for a dynamic version of the "Among Us" game scenario. It includes assumptions, tasks, crewmates, impostors, and the initial state of the game. The file is structured into multiple parts, each focusing on different aspects of the game world.

1. Prover9 and Mace4 Configuration:
The file starts with configuration options for Prover9 and Mace4:

- The maximum time allowed for the search is set to 100 seconds:

```
assign(max_seconds, 100).
```

- The assumption formulas are defined for Prover9:

```
formulas(assumptions).
```

2. Crewmates and Impostors:
The file defines the characters (crewmates and impostors) as follows:

- At least one crewmate:

```
crewmate(Blue) | crewmate(Yellow) | crewmate(Red) | crewmate(Green).
```

- At least one impostor:

```
impostor(Blue) | impostor(Yellow) | impostor(Green) | impostor(Red).
```

- Exactly 3 crewmates and 1 impostor:

```
all x (crewmate(x) <-> -impostor(x)).
all x (crewmate(x) | impostor(x)).
exists x (impostor(x) & all y (impostor(y) -> x = y)).
```

3. Tasks and Locations:
The tasks that crewmates can perform are defined along with their associated locations. The file specifies that each task must be assigned to a unique location:

- Defined places (locations):

```
place(Medbay).
place(Admin).
place(Cafeteria).
place(Reactor).
place(O2).
```

- Tasks must be performed at specific locations:

```
task_name(Scan).
task_name(SwipeCard).
task_name(InsertCode).
task_name(FixMeltdown).
task_name(FlushLeaves).
```

- Constraints on task uniqueness:

```
Scan!=SwipeCard.
Scan!=InsertCode.
Scan!=FixMeltdown.
Scan!=FlushLeaves.
SwipeCard!=InsertCode.
SwipeCard!=FixMeltdown.
SwipeCard!=FlushLeaves.
InsertCode!=FixMeltdown.
InsertCode!=FlushLeaves.
FixMeltdown!=FlushLeaves.
```

4. Task Validity and Interactions:
The file defines conditions that validate each task-location pair and prevents overlapping tasks:

- Task validity across locations:

```
all x all y (is_valid(task(x, y)) -> task_name(x) & place(y)).
```

- Specific task-location assignments are given:

```
task(Scan, Medbay) != task(SwipeCard, Admin).
task(Scan, Medbay) != task(InsertCode, Admin).
task(SwipeCard, Admin) != task(Scan, Medbay).
```

5. Dead and Kill Constraints:
The file enforces logical relationships between crewmates, victims, and impostors:

- Victims are predefined:

```
all x (victim(x)) -> x = Brown | x = Purple | x = Pink.
```

- Only one victim at a time:

```
all x all y (victim(x) & victim(y) -> x = y).
```

- A kill is only possible if the killer is an impostor:

```
all x all y (kill(x, y) -> impostor(x) & victim(y)).
```

6. Problem Situations:
The file defines certain in-game situations like a dead body, reactor activation, and oxygen depletion:

- Dead body found:

```
DeadBody ->
(exists x (dead(x))) & -doTask(x, task(y, Reactor)) & -doTask(x, task(y, O2)
```

- Reactor activated:

```
ReactorActivated ->
(all x (-victim(x))) & -doTask(x, task(y, O2)).
```

- Oxygen depletion:

```
OxygenDepletion ->
(all x (-victim(x))) & -doTask(x, task(y, Reactor)).
```

7. Lies and Truths:
The file models the behavior of impostors and crewmates in terms of what they see:

- Impostors and crewmates make contradictory statements about tasks and killings:

```
all z all x all y all u (impostor(z) & saw(z, doTaskObj(x, task(y,u)))
-> -doTask(x, task(y,u))).
all z all x all y all u (crewmate(z) & saw(z, doTaskObj(x, task(y,u)))
-> doTask(x, task(y,u))).
```

8. Initial Situation:
The initial state of the game is defined, with the situation starting without a dead body and with the reactor activated. The file also specifies the tasks witnessed by certain players:

- Initial conditions:

```
-DeadBody.
ReactorActivated.
-OxygenDepletion.
```

- Observations:

```
saw(Blue, doTaskObj(Blue, task(InsertCode, Medbay))).
saw(Green, doTaskObj(Green, task(FlushLeaves, Cafeteria))).
saw(Yellow, doTaskObj(Yellow, task(FixMeltdown, Reactor))).
saw(Red, doTaskObj(Red, task(FlushLeaves, Cafeteria))).
```

9. Goal:
Finally, the goals of the problem are stated. In this case, the goal is to identify the impostor:

- Goal:

```
formulas(goals).
impostor(Blue).
```

This defines the goal of the logical puzzle in the game: to prove that Blue is the impostor.(In a particular example)

3 APPLICATION FLOW

This chapter details the implementation of the application where users deduce the identity of an **Impostor** based on logical statements from four players: Blue, Green, Yellow, and Red. The program integrates a GUI designed with `CustomTkinter` and logical analysis to ensure consistency between the inputs and outputs.

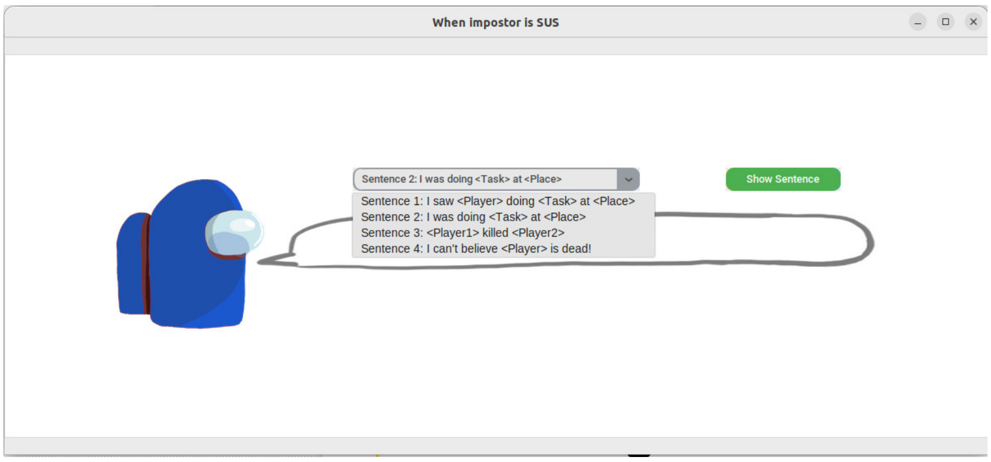
3.1 SENTENCE TYPES

The game begins by allowing users to select the type of sentence each player will provide. Four predefined sentence templates are available:

- **Sentence 1:** `I saw <Player> doing <Task> at <Place>`
- **Sentence 2:** `I was doing <Task> at <Place>`
- **Sentence 3:** `<Player1> killed <Player2>`
- **Sentence 4:** `I can't believe <Player> is dead!`

Below is the Python code used for this feature:

```python
sentence_options = [
    "Sentence 1: I saw <Player> doing <Task> at <Place>",
    "Sentence 2: I was doing <Task> at <Place>",
    "Sentence 3: <Player1> killed <Player2>",
    "Sentence 4: I can't believe <Player> is dead!"
]
sentence_selector = ctk.CTkComboBox(root, values=sentence_options, width=350, **
    dropdown_style)
```


When impostor is SUS

Sentence 2: I was doing <Task> at <Place>

Sentence 1: I saw <Player> doing <Task> at <Place>
Sentence 2: I was doing <Task> at <Place>
Sentence 3: <Player1> killed <Player2>
Sentence 4: I can't believe <Player> is dead!

Figure 3.1: Sentence options display.

3.2 PLAYER STATEMENTS

Once the sentence type is selected, users define its components using drop-down menus. For example, if the selected sentence is:

```
"I saw <Player> doing <Task> at <Place>",
```

the components `<Player>`, `<Task>`, and `<Place>` are chosen through dedicated combo boxes.

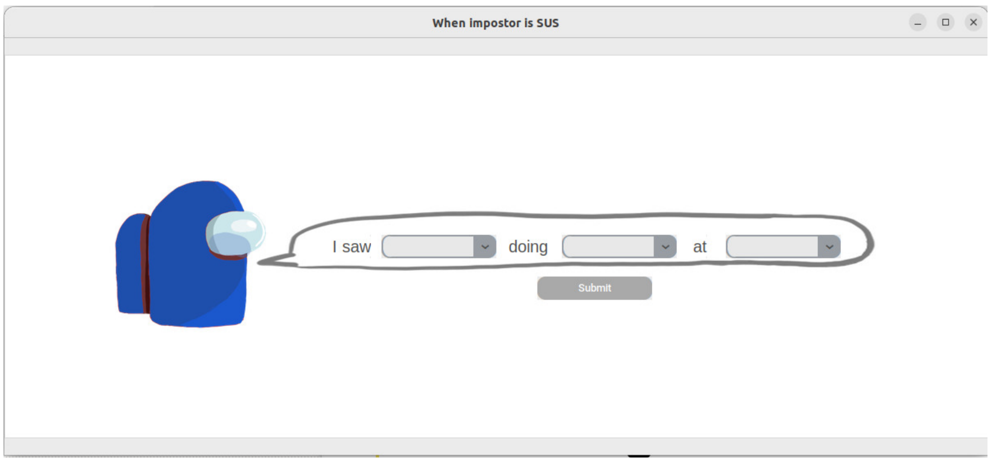


Figure 3.2: Sentence generated by user choice.

Example Code

```python
label1 = ctk.CTkLabel(root, text="I saw")
combo1 = ctk.CTkComboBox(root, values=["Blue", "Yellow", "Green", "Red"])
label2 = ctk.CTkLabel(root, text="doing")
combo2 = ctk.CTkComboBox(root, values=["Scan", "SwipeCard", "FlushLeaves", "FixMeltdown", "InsertCode"])
label3 = ctk.CTkLabel(root, text="at")
combo3 = ctk.CTkComboBox(root, values=["Medbay", "Admin", "Cafeteria", "Reactor", "O2"])
```

Each player's finalized sentence is saved in a dictionary as follows:

```python
player_sentences[current_player] = f"I saw {combo1.get()} doing {combo2.get()} at {combo3.get()}"
```

Listing 3.1: Saving Sentences

3.3 LOGIC EVALUATION

After all players have provided their statements, the program analyzes them using logical constraints:

- **Crewmates always tell the truth.**
- **The Impostor always lies.**

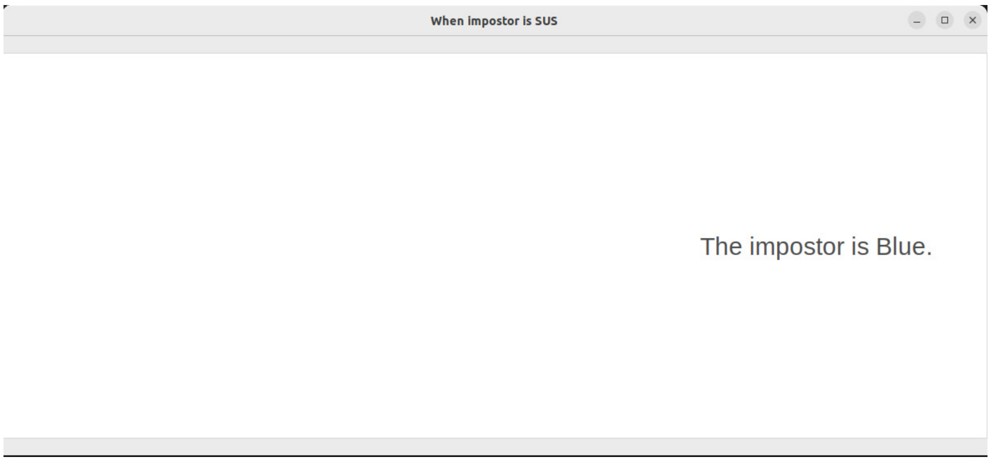
The `prover.py` module deduces the Impostor by checking consistency across the statements. Depending on the analysis:

- If the program cannot determine the Impostor, it outputs: `"I am not sure"`.
- If a single Impostor is identified, it outputs: `"Impostor is <Player>"`.

DISPLAYING RESULTS

```python
def show_impostor(text):
    impostor_label = ctk.CTkLabel(root, text=text, font=("Arial",30), text_color="#4D4D4D", fg_color="white")
    my_canvas.create_window(850, 230, window=impostor_label, anchor="w", tags="sentence_widgets")
```

Listing 3.2: Display Impostor


When Impostor is SUS

The impostor is Blue.

Figure 3.3: Sentence generated by user choice.

3.4 USER INTERACTION AND CONSTRAINTS

Interaction Guidelines

- Users must construct sentences carefully, considering the truthfulness of Crewmates and the lies of the Impostor.
- The game allows only one Impostor and three Crewmates.
- Consistent sentence logic is crucial for accurate deduction.

GUI Features

The GUI dynamically updates based on user selections. Player images are animated using the `PIL` library to enhance the user experience.

4 CONCLUSION

4.1 OVERVIEW OF THE PROJECT

This project set out to integrate game mechanics from the popular "Among Us" game with formal logical reasoning using the Prover9 theorem prover. The goal was to create a system where players could interact with each other, share observations, and use logic to identify the impostor(s) in the game. Through this approach, we aimed to combine game design and artificial intelligence to demonstrate how formal logic can be applied to real-time decision-making processes in games.

4.2 KEY ACHIEVEMENTS

The project achieved the following key milestones:

- Developed a dynamic graphical user interface (GUI) using Python's `CustomTkinter` library. The GUI allowed players to provide their observations and make logical deductions based on those observations.
- Integrated Prover9, a powerful theorem prover, to evaluate and validate logical hypotheses about the players' roles in the game. By translating player observations into logical formulas, we could automate the process of identifying the impostor.
- Designed dynamic input and output systems that adapted in real-time based on player interactions. This allowed for an interactive and engaging user experience.
- Implemented logical rules and constraints within the system that ensured the integrity of the game mechanics. For example, rules that guaranteed the correct number of crewmates and impostors were followed, ensuring that the system always produced valid results.
- Incorporated animations using sine wave transformations, which added an engaging visual element to the game, improving the user experience.

4.3 CHALLENGES FACED

While the project was successful in demonstrating the potential of using formal logic in game design, several challenges were encountered:

- Integrating Prover9 with a real-time graphical interface required careful handling of the interaction between the game's logic and the theorem prover. The system had to efficiently convert dynamic player inputs into logical statements and then process them through Prover9 without causing significant delays.
- The complexity of the logic involved in representing all possible game states and player actions made the system difficult to scale. As the number of players or the complexity of player interactions increased, the logical rules became more difficult to manage.
- Ensuring that the animation and interactive elements remained responsive while simultaneously processing the logical deductions posed a challenge, particularly in terms of performance optimization.

4.4 FUTURE DIRECTIONS

Despite these challenges, the project provides a solid foundation for future exploration and development in this area. Some potential directions for future work include:

- **Enhanced Game Mechanics:** Expanding the game mechanics to include more detailed interactions between crewmates and impostors, such as sabotage actions or more complex statements from players.
- **Real-Time Player Feedback:** Improving the responsiveness of the system to provide real-time feedback to players based on the evolving state of the game. This could include more sophisticated logic to deal with uncertain or contradictory information.
- **Scalability Improvements:** Optimizing the integration between the game's user interface and the Prover9 theorem prover to handle more players and more complex game scenarios. This could involve parallelizing the logical processing or employing more efficient algorithms.
- **Machine Learning Integration:** Exploring the possibility of integrating machine learning techniques to improve the system's ability to identify patterns in player behavior, perhaps predicting who the impostor is based on observed actions and interactions.
- **Enhanced Graphics and Animation:** Adding more complex animations and visual effects that could further immerse players in the game world, making the game more engaging and visually appealing.

4.5 CONCLUSION

In conclusion, this project successfully demonstrated how formal logic, specifically the use of Prover9, can be integrated into an interactive gaming environment to automate decision-making processes. The combination of Python's powerful libraries, interactive design, and logical reasoning offers a novel approach to game design that is both educational and entertaining. Despite the challenges faced during implementation, the project provides a strong foundation for future development and innovation in game AI and formal reasoning. As such, it opens up exciting possibilities for creating intelligent, dynamic games that rely on logical reasoning for problem-solving and game-play mechanics.

Part II

APPENDIX

A

MAIN.PY

```python
import prover as p
import interpret as i
import customtkinter as ctk
from PIL import Image, ImageTk
import math
from tkinter import font, CENTER

# Function headers only
def display_sentence(selection):
    """
    Dynamically display the selected sentence.
    :param selection: The sentence option selected from the dropdown.
    """

def animate_first_image(impasta_image, canvas_impasta_image, first_image_phase
    =0):
    """
    Function to animate the first image.
    """

def get_values_sentence1():
    """
    Retrieve and print selected values from the dropdowns for sentence 1.
    """

def get_values_sentence2():
    """
    Retrieve and print selected values from the dropdowns for sentence 2.
    """

def get_values_sentence3():
    """
    Retrieve and print selected values from the dropdowns for sentence 3.
    """

def get_values_sentence4():
    """
    Retrieve and print selected values from the dropdowns for sentence 4.
    """

def init_ctk():
    """
    Initialize the custom tkinter root and canvas.
    """

def load_and_display_image_impasta(impasta_image_path):
    """
    Load and display the impasta image.
    """
```

```
def load_and_display_sentence_outline():
    """
    Load and display the sentence outline image.
    """

def sentence_option_1():
    """
    Setup dropdowns for sentence option 1.
    """

def sentence_option_2():
    """
    Setup dropdowns for sentence option 2.
    """

def sentence_option_3():
    """
    Setup dropdowns for sentence option 3.
    """

def sentence_option_4():
    """
    Setup dropdowns for sentence option 4.
    """

def switch_player():
    """
    Switch to the next player and update the current state.
    """

def recurrent_design():
    """
    Redesign the interface for the current player.
    """

def start_animation(image, canvas_image):
    """
    Start animating the provided image on the canvas.
    """

def build_design():
    """
    Build and display the main application design.
    """

build_design()
```

B INTERPRET.PY

```python
import prover as p
from prover import player_sentences

def kill_sentence_decoder(rawString, sentence):
    """
    Decodes a sentence about a kill and generates a logical statement for the
     FOL.
    """

def is_dead_decoder(rawString, sentence):
    """
    Decodes a sentence about a dead player and generates a logical statement.
    """

def i_do_task_decoder(rawString, sentence):
    """
    Decodes a player’s own task statement.
    """

def he_does_task_decoder(rawString, sentence):
    """
    Decodes a statement about another player’s task.
    """

def recieve_sentences_as_strings(sentences):
    """
    Processes all sentences and routes them to the appropriate decoder function.
    """
```

C PROVER.PY

```python
import subprocess

base_content = """
assign(max_seconds, 100).
formulas(assumptions).
% Base assumptions
"""

# Generate the .in file with additional player statements
def generate_amogus_in(player_statements):
    """
    Generates the dynamic .in file with player statements added.
    """

# Run Prover9 with a specific hypothesis
def check_impostor(hypothesis, player_statements):
    """
    Checks if the given hypothesis (impostor(player)) is provable using Prover9.
    """

# Main logic for identifying the impostor
def identify_impostor():
    """
    Identifies the impostor by testing each player with Prover9.
    """

# Example usage with initial player statements
player_sentences = {
    "Blue": "doTaskObj(Red, task(FixMeltdown, Reactor))",
    "Green": "",
    "Yellow": "",
    "Red": ""
}
```

Part III

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Python Software Foundation. *Python Tkinter Documentation.* Retrieved from `https://docs.python.org/3/library/tkinter.html`.

[2] Real Python. *Introduction to Tkinter: Create GUIs with Python.* Retrieved from `https://realpython.com/python-gui-tkinter/`.

[3] William McCune and Contributors. *Prover9 and Mace4 Manual.* Retrieved from `https://www.cs.unm.edu/~mccune/prover9/manual/`.