Capstone Project Phase A

Asynchronous Hyperautomata Simulator

Nevo Gottlieb

Raz Kessel

Supervisor: Dr. Sarai Sheinvald

Git repository: https://github.com/RazKessel/Capstone-Project---Hyperautomata-Simulator.git

Table of contents

Abstract. Automata, a fundamental concept in computer science, are models used to study how computers process information. An automata can be viewed as a machine that reads a sequence of symbols and determines whether to accept or reject it based on predefined rules. This concept is crucial for understanding how computers recognize patterns and make decisions. Hyperautomata extend this concept by allowing the simultaneous processing of multiple sequences, making them valuable for analyzing more complex systems, particularly those involving parallel processes. Importantly, hyperautomata provide a basis for verification algorithms, as they can model different security properties.

While most existing research focuses on synchronous hyperautomata, where all sequences are processed in perfect alignment, real-world systems often operate asynchronously, with different components progressing at varying speeds. Our project explores asynchronous hyperautomata, which more accurately reflect the behavior of many real-world systems. We will develop a simulator to study these models, providing a tool for researchers to analyze their properties and behaviors. This work has potential applications in distributed computer networks, multi-robot systems, and biological processes, where components must collaborate without perfect synchronization.

By facilitating the study of asynchronous hyperautomata, this project aims to advance the design and understanding of complex, flexible systems capable of managing asynchronous, multi-part processes. The ability to model and verify security properties through hyperautomata further enhances the significance of this research, making it a valuable contribution to the field of computer science.

1 Introduction

The field of formal languages and automata theory has long been a cornerstone of computational research, providing fundamental insights into the design and analysis of computational systems. Traditional automata theory deals with languages defined over single sequences of words, but the advent of hyperproperties has expanded this paradigm to consider sets of sequences, thus enabling the expression of more complex system behaviors. Hyperproperties are particularly significant in domains such as security, where policies often need to be defined over multiple concurrent executions of a system.

Hyperautomata are computational models that recognize hyperlanguages, which are sets of sets of words. These models can be used to specify and verify properties that are inherently hyper in nature, such as non-interference in security systems or consistency in distributed databases. While synchronous hyperautomata, which process multiple words in a synchronized manner, have been extensively studied, asynchronous

hyperautomata, which allow for independent progression of words, remain relatively unexplored. This gap is critical, as many real-world systems operate asynchronously due to the independent nature of their components.

Our project aims to address this gap by developing a simulation tool for asynchronous hyperautomata and hyperlanguages. The primary objective of this tool is to provide researchers with a means to investigate and verify the properties of asynchronous hyperlanguages. This investigation is essential for understanding the dynamics of systems that operate concurrently without requiring synchronized progression, such as distributed systems, multi-threaded applications, and certain types of network protocols.

To illustrate the difference between regular automata and hyperautomata, consider a simple finite automata that reads a single word, letter by letter, and determines whether the word belongs to a specific language. Each transition in this automata depends on the current letter of the word. In contrast, a hyperautomata reads multiple words simultaneously. Each transition is based on reading letters from several words at once, allowing it to recognize more complex relationships between these words.

In synchronous hyperautomata, all the words are processed in parallel, meaning that at each transition, a new letter from each word is read simultaneously. For example, if there are three words being processed, the automata would move to a new state after reading the next letter from each of the three words at the same time. This synchronized progression ensures that all words advance together, making it easier to analyze relationships that depend on the alignment of different sequences.

Asynchronous hyperautomata, on the other hand, allow for more flexibility. In these models, not all words need to advance at the same rate. At a given transition, some words may move to their next letter while others remain at the same position. This reflects the behavior of systems where different components may operate at varying speeds, such as in distributed computing, where some processes may experience delays or progress faster than others.

Next, we will implement algorithms that simulate these models, ensuring they can handle various scenarios and configurations typical of asynchronous systems. The simulator will be designed to be user-friendly and flexible, allowing researchers to easily define and test different hyperlanguages and their properties. For instance, given a hyperlanguage $L \subseteq (\Sigma^*)^k$ (where k is the number of concurrent words), the simulator will facilitate the exploration of properties such as closure under union and intersection.

Validation of the simulator will be conducted through a series of test cases and real-world applications, demonstrating its effectiveness and reliability. By providing this tool, we aim to facilitate deeper insights into the nature of asynchronous hyperlanguages and support the development of more robust, secure, and efficient systems.

In conclusion, our project seeks to fill a significant research gap by focusing on asynchronous hyperautomata and hyperlanguages. The development of a dedicated simulation tool will not only advance theoretical understanding but also provide practical benefits for researchers and practitioners working with complex,

asynchronous systems. This work has the potential to significantly impact various fields, including distributed computing, security, and formal verification, by enabling more comprehensive analysis and verification of asynchronous behaviors.

2 From Automata to Hyperautomata

2.1 Understanding Automata

To grasp the concept of hyperautomata, we must first understand the basics of traditional automata.

2.1.1 What is an Automata?

An automata is a mathematical model of computation. It's like a simple machine that reads an input (typically a string of symbols) and decides whether to accept or reject that input based on a set of predefined rules.

2.1.2 Formal Definition of a Finite Automata

Formally, a deterministic finite automata (DFA) is a 5-tuple $(Q, \Sigma, \delta, q0, F)$, where:

- Q is a finite set of states
- Σ is a finite set of input symbols (the alphabet)
- $\delta: Q \times \Sigma \rightarrow Q$ is the transition function
- $q0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final (accepting) states

2.1.3 Simple Example: Even-Length Binary Strings

Let's consider a simple automata that accepts binary strings of even length.

States: $Q = \{q0, q1\}$, Alphabet: $\Sigma = \{0, 1\}$, Initial state: q0, Final states: $F = \{q0\}$

Transition function:

- $\delta(q0, 0) = q1$
- $\delta(q0, 1) = q1$
- $\delta(q1, 0) = q0$
- $\delta(q1, 1) = q0$

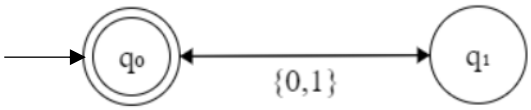

Fig 1: Simple Automata

This automata starts in q0, and alternates between q0 and q1 with each input symbol. It accepts if it ends in q0, which happens for strings of even length.

2.1.4 Running the Automata

Let's run this automata on the input "1010":

1. Start in q0
2. Read '1': Move to q1
3. Read '0': Move to q0
4. Read '1': Move to q1
5. Read '0': Move to q0

The automata ends in q0, an accepting state, so it accepts "1010" which indeed has an even length.

2.2 Introducing Hyperautomata

Now that we understand basic automata, let's explore hyperautomata.

2.2.1 What is a Hyperautomata?

A hyperautomata is an advanced type of automata designed to work with hyperlanguages, which are sets of sets of words. Unlike traditional automata that accept or reject individual words, hyperautomata accept or reject collections of words based on defined relationships between them. This connection allows hyperautomata to capture complex system properties that involve multiple execution traces, making them essential for analyzing behaviors that depend on the interaction of multiple inputs.

2.2.2 Formal Definition of a Hyperautomata

A hyperautomata is a 6-tuple (Q, Σ, k, δ, q0, F), where:

- Q is a finite set of states
- Σ is a finite set of input symbols (the alphabet)
- k is the number of input tapes (number of strings processed simultaneously)
- δ: $Q \times (\Sigma \cup \{\#\})^k \rightarrow 2^Q$ is the transition function
- q0 ∈ Q is the initial state
- F ⊆ Q is the set of final (accepting) states
- α is a quantification condition over tapes

Note that the transition function δ maps to a set of states (2^Q), not a single state. This is because hyperautomata are inherently nondeterministic in nature. Even if we process all k tapes synchronously, the automata may have multiple possible next states for a given input combination.

2.2.3 Simple Example: For each word there is a longer word

Let's consider a hyperautomata that accepts pairs words X, Y.

Quantifier: $\forall X \exists Y$

States: Q = {q0, q1}, Alphabet: Σ = {a}, Number of tapes: k = 2, Initial state: q0, Final states: F = {q1}

Transition function (simplified representation):

- δ(q0, (a,a)) = q0
- δ(q0, (#,a)) = q1
- δ(q1, (#,a)) = q1

$\mathcal{L} = \{L | L \subseteq \{a^n | n \geq 0\}, |L| = \infty\}$ X is the first tape and Y is the second tape, for each word in the language has a longer word in the language. which is equivalent to infinite language.

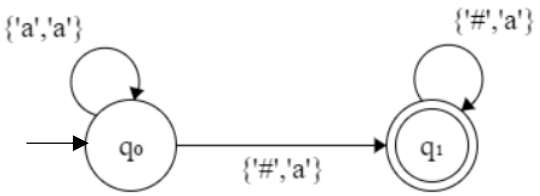

Fig 2:Simple Hyperautomata

2.2.4 Running the Hyperautomata

Let's run this hyperautomata on the input pair (X="aa", Y= "aaa"):

1. Start in q0
2. Read ('a', 'a'): Stay in q0
3. Read ('a', 'a'): Stay in q0
4. Read ('#', 'a'): Move to q1 (# represents the end of the first string)

The hyperautomata ends in q1, which is an accepting state, so it accepts this pair of strings (which indeed the word 'aa' has a longer word 'aaa' in the set).

2.3 From Synchronous to Asynchronous Hyperautomata

2.3.1 Synchronous Hyperautomata

The example we just saw is a synchronous hyperautomata. It processes one symbol from each input string in each step. This is like having multiple tapes that always move in sync.

2.3.2 Asynchronous Hyperautomata

An asynchronous hyperautomata, on the other hand, allows for independent progression on different input strings. In each step, it may process a symbol from any subset of the input strings.

2.3.3 Formal Definition of an Asynchronous Hyperautomata

An asynchronous hyperautomata can be defined as a 6-tuple (Q, Σ, k, δ, q0, F), where:

- Q, Σ, k, q0, and F are as in a synchronous hyperautomata
- δ: $Q \times (\Sigma \cup \{\#\})^k \rightarrow 2^Q$ is the transition function

Here, # represents "no move" on a particular tape.

2.3.4 Simple Example: Total Even Number of 1s

Let's consider an asynchronous hyperautomata that accepts pairs of binary strings X,Y with the same number of 1s.

Quantifier: $\forall X \forall Y$

States: Q = {q0, q1}, Alphabet: Σ = {0, 1}, Number of tapes: k = 2, Initial state: q0, Final states: F = {q0}

Transition function (partial representation):

- δ(q0, (1,#)) = δ(q0, (#,1)) = q1
- δ(q0, (0,#)) = δ(q0, (#,0)) = q0
- δ(q1, (1,#)) = δ(q1, (#,1)) = q0
- δ(q1, (#,0)) = δ(q1, (0, #)) = q1

$\mathcal{L} = \{L | L \subseteq \{0,1\}^* | \forall w \in L: \#_1(w) \equiv 0 mod 2 \vee \forall w \in L: \#_1(w) \equiv 1 mod 2\}$

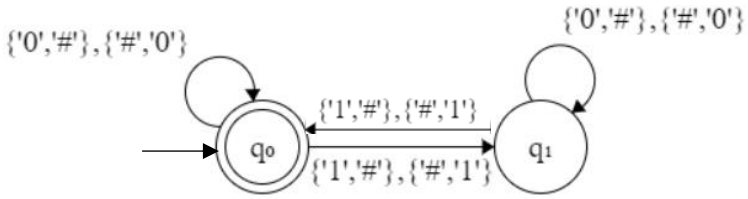

Fig 3: Asynchronous HyperAutomata

2.3.5 Running the Asynchronous Hyperautomata

Let's run this on the input pair ("1101", "0111"):

1. Start in q0
2. Read ('1',#): Move to q1
3. Read ('1',#): Move to q0
4. Read (#,'0'): Stay in q0
5. Read (#,'1'): Move to q1
6. Read ('0',#): Stay in q1

7. Read (#,'1'): Move to q0
8. Read ('1', #): Move to q1
9. Read (#, '1'): Move to q0

The hyperautomata ends in q0, an accepting state, correctly recognizing that the total number of 1s in both strings is even (6 1s in total).

2.4 Conclusion

Hyperautomata provides a powerful framework for analyzing systems with multiple concurrent processes. While synchronous hyperautomata offer a simpler model, asynchronous hyperautomata more closely reflect the behavior of many real-world systems where different components may progress at varying rates. Understanding these models opens up new possibilities for designing and verifying complex, distributed systems.

The progression from traditional automata to synchronous hyperautomata and finally to asynchronous hyperautomata reflects the increasing complexity of the systems we aim to model and analyze in computer science. As we move towards more sophisticated computational models, we gain the ability to represent and reason about more complex behaviors and interactions in distributed and concurrent systems.

3 Literature Review

Automata theory has been a cornerstone in the study of formal languages and computational systems, traditionally focusing on individual sequences of symbols (or traces) to define system behaviors. However, as computational systems have grown more complex, especially in the realm of distributed systems and multi-agent architectures, the need for a broader analytical framework emerged. This led to the development of hyperproperties, which extend traditional trace properties by examining sets of execution traces rather than individual ones.

Hyperautomata were introduced to handle this new level of complexity by operating on hyperlanguages, which are sets of sets of words. Unlike traditional automata, which evaluate single words or traces, hyperautomata allow for the verification of properties across multiple concurrent executions. They play a significant role in analyzing cyber-physical systems, which require the evaluation of system properties involving multiple parallel processes, such as sensitivity and consistency in distributed computing [1].

In [1], the authors introduce nondeterministic finite-word hyperautomata (NFH), which extend the traditional automata framework to operate over finite-word hyperlanguages. These languages are especially useful for systems that operate on finite traces, as seen in applications involving security policies or distributed computing. NFH is based on standard automata models but is designed to recognize sets of words, making it suitable for analyzing multi-execution systems [1].

Hyperlanguages, as studied in this framework, possess important properties, such as closure under union, intersection, and complementation. These properties ensure that hyperlanguages remain well-behaved under standard operations, facilitating their

analysis and verification. Additionally, the article explores the complexity of decision problems related to hyperautomata, such as nonemptiness, universality, and membership.[1].

A more generalized approach to automata for hyperlanguages was introduced by the authors [1], where nondeterministic finite-word hyperautomata (NFH) were proposed to handle sets of finite words, termed hyperwords. These automata are designed to accept regular hyperlanguages, which are closed under basic Boolean operations such as union, intersection, and complementation. The authors also demonstrate that while the nonemptiness problem for NFH is undecidable in general, it is decidable for certain fragments [1].

This new model plays a significant role in analyzing information-flow security policies and consistency conditions in distributed systems, as these hyperproperties often require reasoning about multiple execution traces. The study highlights how regular hyperlanguages can express these complex properties for systems operating on finite traces [1].

Further expanding on automata's role in security analysis, Khoumsi et al. (2014) present an automata-based approach specifically designed for the formalization and verification of security policies, such as those implemented in firewalls. Their framework synthesizes an automata to model filtering rules, enabling the detection of anomalies, verification of completeness, and identification of functional discrepancies between different policy implementations. This method leverages the rigorous theory of automata to enhance security policy design and management, offering a unified approach to analyze and optimize such systems [5].

While NFH and regular hyperlanguages provide a robust framework for reasoning about systems with synchronous multi-executions, recent research has shifted focus toward models that handle asynchronous behaviors. Asynchronous systems, where components may operate at different speeds or times, pose additional challenges, especially in security-critical contexts.

Further developments in automata models have introduced Alternating Asynchronous Parity Automata (AAPA) to address asynchronous hyperproperties. This model provides a more expressive framework for analyzing systems with asynchronous behaviors, especially for verifying security properties such as non-interference in systems where execution granularity varies. However, the complexity of decision problems, including model checking and satisfiability, remains high, with some problems being undecidable [3].

While AAPA model provides a powerful tool for handling asynchronous hyperproperties, there are other significant contributions in the realm of asynchronous automata. Finite asynchronous automata, introduced by Zielonka [7], focus on recognizing regular trace languages and executing independent actions simultaneously. These automata are crucial for studying concurrent systems, particularly those involving independent or partially commutative actions, and offer a formal framework for reasoning about asynchronous behaviors in multi-process systems. Their ability to

recognize subsets of free partially commutative monoids makes them a key tool for analyzing asynchronous systems [7].

More developments in the field of hyperproperties are discussed in a more recent study, where Hyper Temporal Stream Logic (HyperTSL) is introduced as a formalism for expressing hyperproperties in software verification. TSL extends Linear Temporal Logic (LTL) with memory cells, functions and predicates, making it a convenient and expressive logic to reason over software and other systems with infinite data domains [2]. HyperTSL further extends TSL to the specification of hyperproperties– properties that relate multiple system executions. HyperTSL extends linear temporal logic by incorporating quantifiers over multiple execution traces, allowing it to handle more complex hyperproperties, much like the goal of hyperautomata which is the main subject of our project. [2].

Additionally, the study outlines model checking algorithms designed to handle these hyperproperties, showing the practical applications of automata-based approaches in verifying software systems. These algorithms are particularly relevant when verifying properties such as non-interference in security contexts, where hyperautomata and hyperlanguages play a critical role in ensuring information flow policies are upheld across different system executions [2].

Another approach of understanding the dynamics of systems that operate concurrently without requiring synchronized progression, such as distributed systems is the concept of distributed automata which offers additional insight into modeling distributed systems, closely aligning with our project's focus on asynchronous hyperautomata. Distributed automata consist of multiple automata collaborating to accept a language, reflecting key aspects of parallelism, concurrency, and communication found in distributed computing environments [4]. In a manner similar to asynchronous hyperautomata, distributed automata allow components to progress independently, albeit by adhering to specific modes of interaction. This shared principle of decentralized computation underscores their applicability in modeling multi-agent systems and distributed networks, where synchronization cannot be guaranteed. Notably, the computational power of distributed Pushdown Automata (PDA) equates to Turing Machines, highlighting the advanced capabilities of distributed models [4]. In our project, the development of a simulator for asynchronous hyperautomata builds upon this foundational idea by enabling researchers to explore the behavior of systems with independently progressing components, further enhancing the analysis and verification of asynchronous distributed systems.

Having discussed the technical and theoretical foundations of automata, it's clear that hands-on tools are crucial for reinforcing these concepts and enhancing students or researchers in our case comprehension. To help students grasp these foundational concepts of automata and how they assist in analyzing computational processes, automata simulation tools have been developed over the decades. One such tool is the *Automata Simulator* presented in [6], a mobile app designed to teach the theory of computation by allowing users to model and simulate finite accepters, pushdown accepters, and Turing machines. This tool, when combined with the desktop-based *JFLAP*, improved students' understanding and engagement in automata-related courses,

making the learning process more interactive and contributing positively to overall comprehension [6]. This highlights the importance of simulation tools in simplifying complex theoretical models and preparing the ground for more advanced systems like hyperautomata.

In conclusion, the evolution of automata theory from single-sequence models to more advanced frameworks like hyperautomata and distributed automata highlights the increasing complexity of modern computational systems. These advancements, particularly in handling hyperproperties and asynchronous behaviors, provide powerful tools for analyzing multi-execution systems such as distributed networks, cyber-physical systems, and concurrent software architectures. The literature reveals the critical role of automata in formal verification, security analysis, and ensuring consistency in distributed processes. By building on these foundational concepts, our project seeks to contribute to this growing body of research through the development of a simulator for asynchronous hyperautomata, enabling deeper insights into the dynamics and verification of complex, asynchronous systems.

4 Project Implementation

4.1 Software Architecture and Functionality

The core of our project is the development of a simulator for asynchronous hyperautomata and hyperlanguages. This simulator will be designed as a modular, extensible software system comprising several key components:

1. Automata Definition Module: Allows users to define asynchronous hyperautomata using friendly UI .
2. Input Processing Module: Handles the input of multiple words to be processed concurrently by the automata.
3. Asynchronous Execution Engine: Simulates the asynchronous progression of the automata on input words.
4. State Visualization Component: Provides a graphical representation of the automata's state transitions during execution.
5. Analysis Module: Computes and reports various properties of the simulated hyperlanguage.
6. User Interface: Offers an intuitive interface for researchers to interact with the simulator.

The simulator will operate by allowing researchers to input an asynchronous hyper-automata definition and a set of input words. It will then simulate the automata's behavior, showing how different components progress asynchronously. The system will provide both step-by-step visualization and summary reports of the automata's behavior.

4.2 Algorithms and Research Methodology

The core algorithm of our simulator will be based on a modified breadth-first search approach, adapted to handle the asynchronous nature of the automata.

This algorithm will be optimized to handle the potentially exponential state space efficiently, employing techniques such as lazy evaluation and pruning of redundant states.

Our research methodology will involve iterative development and testing of the simulator.

4.3 Key Project Requirements

Functional Requirements:

- Define and input asynchronous hyperautomata
- Simulate automata execution on multiple input words
- Visualize automata states and transitions
- Export simulation results

Non-Functional Requirements:

- Performance: Handle complex automata with multiple input words efficiently
- Usability: Intuitive interface for researchers without extensive programming knowledge
- Reliability: Ensure consistent and accurate simulation results

4.4 Requirements Gathering Process

Our requirements were gathered through a combination of methods:

- Literature review: Analyzing existing research on hyperautomata and hyperlanguages
- Expert interviews: Consulting with researchers in formal language theory and distributed systems
- Use case analysis: Identifying potential applications and user stories for the simulator
- Competitive analysis: Examining existing tools for automata simulation and their limitations

4.5 Research and Development Process

Our development process will follow an agile methodology, with iterative cycles of implementation and testing. Key steps include:

1. Detailed design of the simulator architecture
2. Implementation of core modules (automata definition, execution engine)
3. Development of the user interface and visualization components
4. Integration of analysis tools and report generation features
5. Extensive testing and validation using known hyperlanguage examples
6. Refinement based on user feedback and performance analysis
7. Documentation and preparation of case studies

4.6 Anticipated Challenges

We anticipate several challenges in the development of this simulator:

- Efficient handling of the potentially infinite state space of asynchronous executions
- Designing an intuitive representation for complex hyperautomata
- Ensuring the correctness of the simulation for edge cases and complex automata
- Balancing between generality (to handle various types of hyperlanguages) and specificity (to provide detailed analysis for particular classes)

4.7 Development Tools

We will utilize the following tools and technologies:

- Programming Language: Python for its rich ecosystem in scientific computing and ease of prototyping
- GUI Framework: PyQt for creating a cross-platform user interface
- Visualization Libraries: Matplotlib and NetworkX for state and transition visualizations
- Version Control: Git for collaborative development and version management
- Testing Framework: Pytest for unit testing

4.8 Client Interaction During Development

We will maintain regular communication with our academic advisors and potential end-users throughout the development process. This will include:

- Bi-weekly progress meetings to discuss developments and challenges
- Monthly demonstrations of new features and capabilities

4.9 Testing Procedures

Our testing strategy will encompass:

- Unit Testing: Verifying the correctness of individual components
- Functional Testing: Validating that the simulator meets all specified requirements
- Performance Testing: Assessing the system's efficiency with large and complex automata
- User Acceptance Testing: Gathering feedback from potential end-users

We will develop a comprehensive test suite including known hyperlanguages with well-understood properties to validate the simulator's accuracy.

4.10 Success Metrics

The success of our project will be evaluated based on the following criteria:

- Correctness: Accurate simulation of asynchronous hyperautomata behavior
- Performance: Ability to handle complex automata within reasonable time and memory constraints
- Usability: Positive feedback from researchers on the tool's interface and functionality
- Extensibility: Successful implementation of additional features or analysis techniques post-initial development
- Research Impact: Use of the simulator in published research papers or academic projects
- Comparison with Theoretical Results: Alignment of simulator outputs with known theoretical properties of hyperlanguages

By meeting these criteria, we aim to provide a valuable tool for the research community, advancing the study of asynchronous hyperautomata and hyperlanguages.

5 Expected Achievements

The development and implementation of a simulator for asynchronous hyperautomata and hyperlanguages is anticipated to yield several significant achievements, both in theoretical computer science and practical applications.

Primarily, we expect this project to bridge a crucial gap in the study of formal languages and automata theory. By providing a robust tool for simulating and analyzing asynchronous hyperautomata, we aim to facilitate a deeper understanding of the behavioral properties of these complex computational models. This understanding is expected to contribute to the broader field of concurrent and distributed systems, where asynchronous operations are prevalent.

A key anticipated achievement is the empirical validation of theoretical properties of asynchronous hyperlanguages. The simulator will allow researchers to test hypotheses and explore the implications of various automata configurations, potentially leading to new insights or the discovery of previously unknown properties. This empirical approach complements theoretical work and may guide future research directions in the field.

The development of efficient algorithms for simulating asynchronous operations on multiple input words is another expected achievement. These algorithms may have broader applications beyond the scope of this project, potentially influencing approaches to parallel computation and distributed systems design.

Additionally, we anticipate that the user-friendly interface and visualization components of our simulator will make the study of asynchronous hyperautomata more accessible to a wider audience of researchers and students. This increased accessibility may foster greater interest and participation in this specialized field, potentially accelerating progress and innovation.

Lastly, we expect that the modular and extensible nature of our simulator will provide a foundation for future research. By allowing for the easy integration of new analysis

techniques or automata types, we aim to create a versatile tool that can evolve alongside theoretical advancements in the field.

In summary, the expected achievements of this project encompass both theoretical advancements and practical tools for the study of asynchronous hyperautomata and hyperlanguages. These achievements are poised to contribute significantly to our understanding of complex, concurrent systems and to provide valuable resources for ongoing research in this critical area of computer science.

6 Proposed Data Structure

The main data structure responsible for holding the automata is a Map of Lists. Each primary index in this list represents a state in the automata, and the list stored at each index contains Transition objects. Each Transition object consists of a SymbolVector (representing a vector of inputs, such as ('1', #)) and the state that results from applying this transition.

1. Tape Object
    - Purpose: Manages the input word and tracks the tape's current position.
    - Attributes:
        - symbols: A string representing the word on the tape.
        - current_position: An integer representing the current index in the word that the tape is processing.
        - Path_tracker: a PathTracker Object
    - Methods:
        - read(): Advances the tape's position when a symbol is read.
        - no_read(): Keeps the position unchanged when no symbol is read.
2. SymbolsVector Object
    - Purpose: Represents the inputs that trigger a transition.
    - Attributes:
        - size: An integer representing the size of the vector due to the amount of tapes.
        - vector: A list of symbols in size of size containing the input symbols (such as ['1', #]).
3. Transition Object
    - Purpose: Defines a transition based on a vector of inputs.
    - Attributes:

        - Symbols_vector: A SymbolsVector object that defines the input causing the transition.
        - state: An integer representing the state that results from applying this transition.

4. Simulation Object
    - Purpose: Manages the automata's current state, the tapes, and the history of transitions.
    - Attributes:
        - Id: an integer representing the id of the execution.
        - current_state: An integer representing the automata's current state.
        - tapes: A list of Tape objects, each representing an independent input word.
        - history: A stack (implemented as a list of lists) where each cell holds a list of the current state and the positions of each tape at every step of the simulation. For example, [[1, 2, 1], [2, 3, 1]] means that the last step transitioned to state 2, with the first tape at index 3 and the second tape at index 1.
    - Methods:
        - Step_forward(int state): This method navigates to the transition list for the given state in the main data structure and selects a Transition to execute, updating the tapes and state accordingly.
        - Step_back(): This method pops from the history stack and revert the simulation accordingly.
5. PathTracker Object
    - Purpose: Tracks the specific path of tape through the automata states. It records each transition step, ensuring the paths of all tapes are accurately represented for later analysis or backtracking.
    - Attributes:
        - path: A list that tracks the sequence of transitions (state and position) for a particular tape.
    - Methods:
        - record(): Records the current state and the position of the tape.
        - get_path(): Retrieves the path of a specific tape.
6. Manager Object

- Purpose: Coordinates the entire simulation, managing configurations, state updates, path tracking, and user commands.
- Attributes:
    - Automata: a Map of list of Transitions. Each key represents a state, and for each state there is a list of Transitions that can be executed from it.
    - simulations: List of Simulations, which keeps track of the state, tapes, and history.
    - Accepting_state: a list of all accepting states in the automata.
    - Tapes: List of Strings.
    - Visited_simulation: a set of visited simulation to manage visited simulations to ease the process.
- Methods:
    - run_step(Simulation s1): Executes a single step of simulation. It applies the transition associated with the current state and inputs.
    - undo_step(): Reverts the last step by updating the Configuration and PathTracker to the previous state.
    - add_word(String word): Adds a new tape to the configuration and updates the PathTracker to start tracking its transitions.
    - get_current_state(Simulation s1): Retrieves the automata's current state, reflecting the latest step in the simulation.

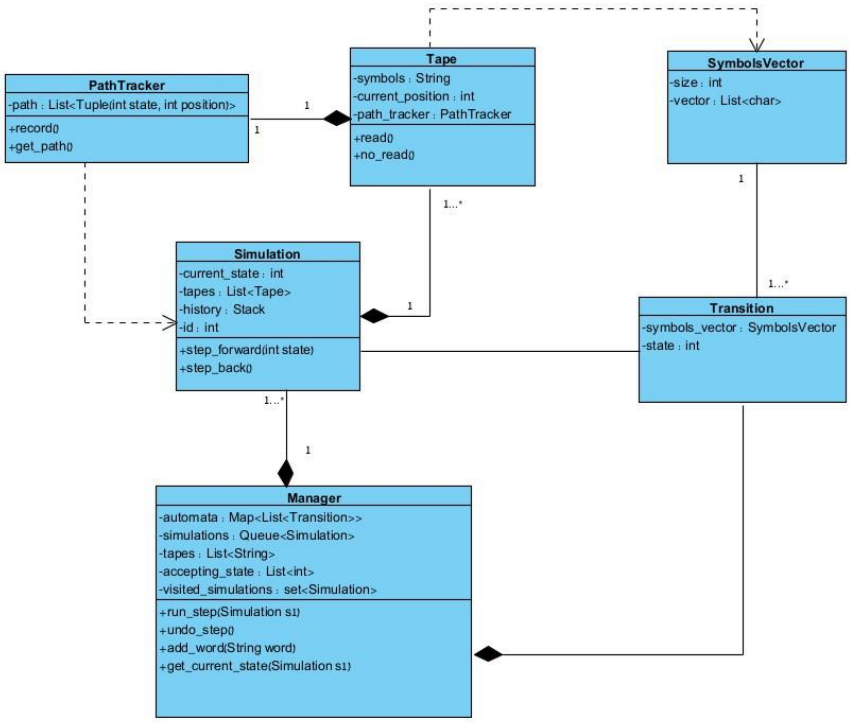

Fig 4: Class diagram

6.1 How This Data Structure Serves the Simulation

1. **Precise State and Position Tracking**
   Each Tape object independently tracks its own progression, while the Configuration object keeps the automata in sync, ensuring all states are accurately managed.

2. **Modular and Reversible Simulation**:
   The Manager and PathTracker objects enable a modular simulation where each step can be executed or reversed, supporting detailed analysis of each transition.

3. **Path Management**:
   PathTracker ensures that each tape's unique path is recorded, allowing the system to backtrack accurately, even when only some tapes advance.

4. **Coordinated Control**:
   The Manager object oversees the simulation flow, coordinating state updates, path tracking, and user interactions, ensuring the system behaves correctly under complex asynchronous conditions.

This design provides a robust, flexible, and clear framework for simulating asynchronous hyperautomata, making it easier to analyze and debug the unique behaviors of hyperlanguages.

7 The Algorithm

The core algorithm for simulating asynchronous hyperautomata is based on a modified breadth-first search approach.

Why BFS?

- BFS ensures that the algorithm explores all possible runs of the automata in a level-wise manner. This is important because asynchronous automata can have multiple possible transitions at each state, and we want to ensure that we find an accepting state if one exists without diving too deep into a single path before checking others.

7.1 Algorithm description:

1. Initialization:
    - Initialize the automata with the starting state (q0)
    - Load input words onto separate tapes
    - Initialize an empty queue for BFS.
    - Initialize an empty set for visited simulations
    - Create an initial Simulation with the starting state and the initial positions of all tapes.
    - Add this simulation to the visited set.
    - Add this initial Simulation to the BFS queue.
2. Main Loop:
    - While not all input words are consumed and no accepting state is reached AND queue is not empty
        a. dequeue the next Simulation.
        b. Check if current state is accepting state and all tapes are consumed:

            If yes, return simulation history.

        c. For each transition in current state that can progress:
            1. Create a new Simulation by:

                - Updating the state to the new state defined by the transition.
                - Advancing the positions of the tapes that have read symbols (based on the `SymbolsVector`).
                - Record the transition in the history.
                - Check if simulation exists in 'visited': if yes- continue, otherwise- add simulation to 'visited' set.
                - Add the new Simulation to the BFS queue

3. Return last simulation history

7.2 Explanation of the Algorithm

1. **Initialization**:

    - A Manager object is created to oversee the simulation process. Each input word is turned into a Tape object, and their starting positions are initialized to 0.
    - The BFS algorithm is initialized by creating a queue where each item in the queue represents a configuration of the automata at a given point in the simulation (i.e., a specific state and the positions of all tapes).
    - An initial configuration is created representing the automata's start state with all tapes at their initial positions. This configuration is added to the BFS queue to start the simulation. A set is also initialized to track visited configurations so that the algorithm does not revisit the same configuration multiple times.

2. **Main BFS Loop**:

    - The core of the algorithm uses Breadth-First Search (BFS) to explore all possible configurations the automata can move through. BFS ensures that the algorithm explores all transitions level-by-level, avoiding deeper paths until all transitions at the current level have been explored.
    - At each iteration, the algorithm dequeues the next configuration from the queue, which consists of the automata's current state and the positions of the tapes.

3. **Termination**:

    - The algorithm terminates when an accepting state is found and all tapes have been consumed (in which case, the accepted path is returned) or when the queue becomes empty (in which case, no valid path was found, and the simulation is considered failed, return the last simulation path).

4. **Output**:

    - After termination, the history of the simulation steps is output, showing the sequence of states and transitions taken during the run. This can be used for debugging, analysis, or visualization.

8 Testing Process

The testing process will be comprehensive, covering various aspects of the simulator:

1. Unit Testing:
    - Test individual components (e.g., automata definition parser, transition function, state management)
    - Use frameworks like Pytest to automate unit tests
    - Ensure each module behaves correctly in isolation
2. Functional Testing:
    - Create a suite of test cases covering various types of asynchronous hyperautomata
    - Include edge cases and complex automata configurations
    - Verify that the simulator produces correct results for known hyperlanguages
3. Performance Testing:
    - Benchmark the simulator with increasingly complex automata and longer input words
    - Measure execution time and memory usage
    - Identify and optimize performance bottlenecks
4. Usability Testing:
    - Conduct user sessions with potential end-users (researchers, students)
    - Gather feedback on the interface, visualization, and overall user experience
    - Iterate on the design based on user feedback
5. Validation Testing:
    - Compare simulator results with theoretical predictions for well-understood hyperlanguages
    - Collaborate with domain experts to verify the correctness of complex simulations
6. Acceptance Testing:
    - Demonstrate the simulator to project stakeholders (academic advisors, potential users)
    - Verify that the tool meets all specified requirements and project goals

Throughout the testing process, we'll maintain a test log, track bug reports, and continuously refine the simulator based on test results.

9 Product Diagrams and GUI

In this section, we describe our work process and the initial draft of the proposed GUI for the system. The front-end system is designed using Python to facilitate user interaction with the underlying hyperautomata simulation engine. Figure [1] illustrates a sequence diagram that details our work process within the proposed system architecture. The user begins by defining the automata within the UI. This triggers the creation of the automata object, which is passed on to the input processor. The user then inputs words, which the input processor handles to simulate the automata's behavior

based on the provided inputs. Finally, the execution engine simulates the automata, updating both the UI and visualizer components to reflect the simulation results.

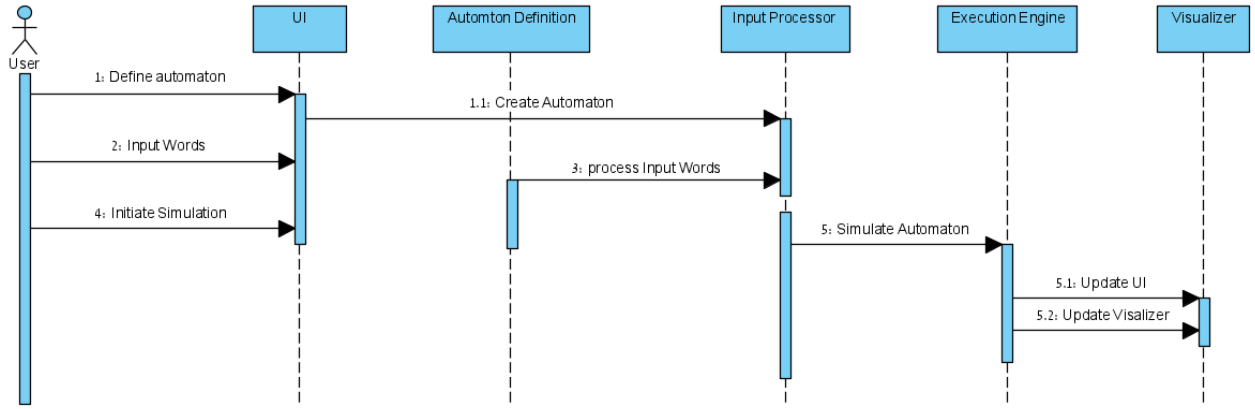

Fig 5: Sequence Diagram

The design of the user interface (UI) prototype, as shown in Figure [6,7,8,9], serves as the interaction point for users to initialize, define, manipulate, and simulate hyperautomata. The UI includes an automata view where users can visualize the current state of the automata, as well as input controls to manage the simulation process. Users can add new words, navigate through the states of the automata, and observe how the automata reacts to different inputs, providing immediate feedback on the state transitions. The combination of these tools supports the development and analysis of hyperautomata, enabling researchers and developers to explore complex systems and properties within this domain.

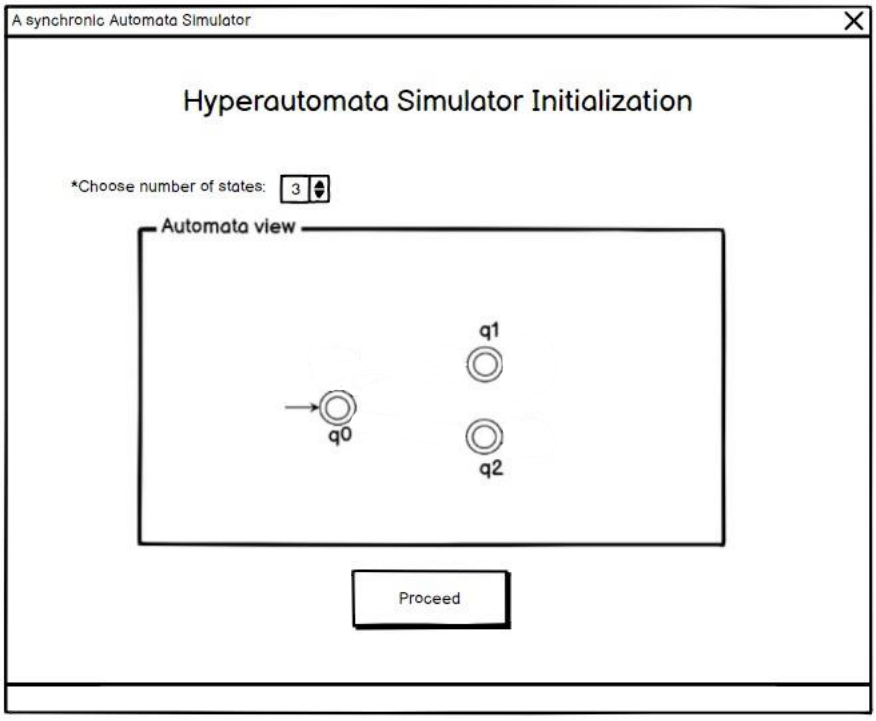

Fig 6: The screen allows the user to define numbers of state in the automata. Each change of the input causin new simulation of the automata according to the user choose

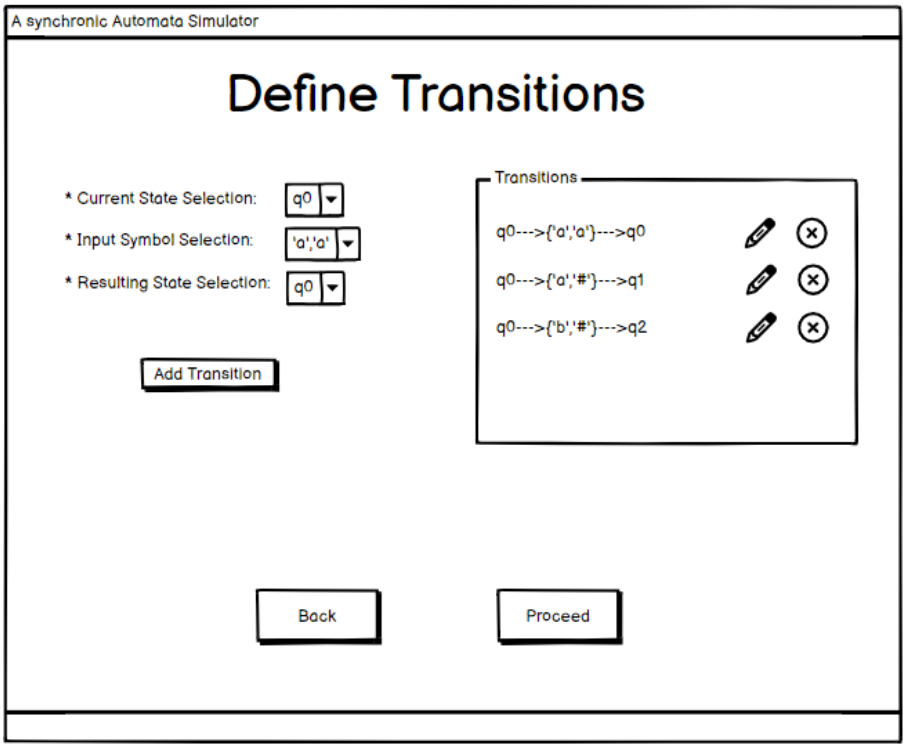

Fig 7: The screen allows the user to define transitions for the automata. It includes options to select the current state, input a symbol, and choose the resulting state. On the right side, a list of defined transitions is displayed.

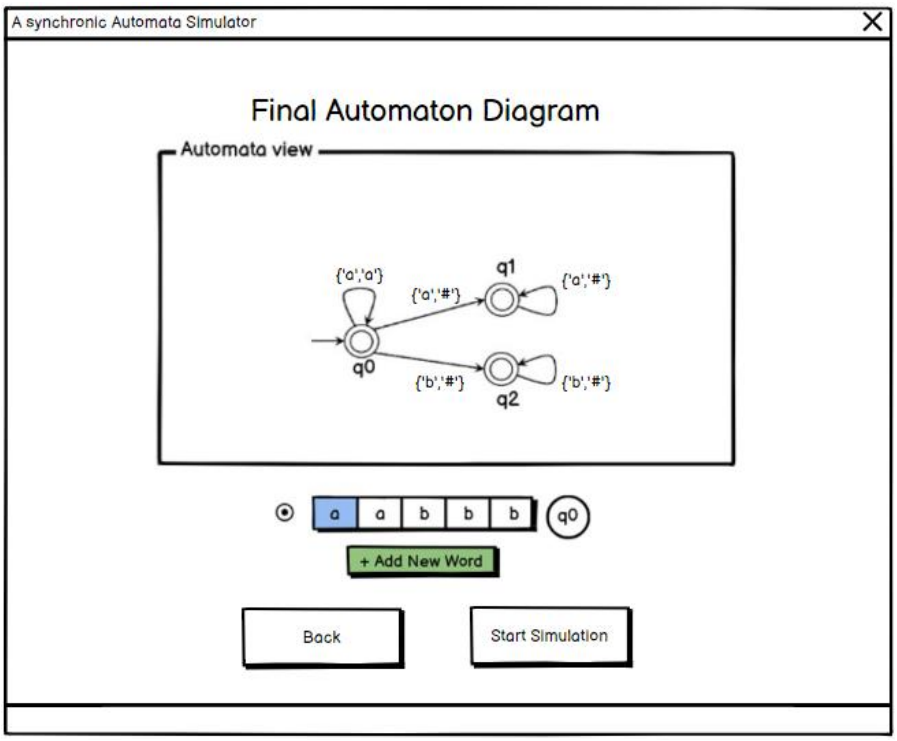

Fig 8: This screen displays the final automata defined by the user, showing all states and transitions visually. It includes an input field for entering words and a button to start the simulation.

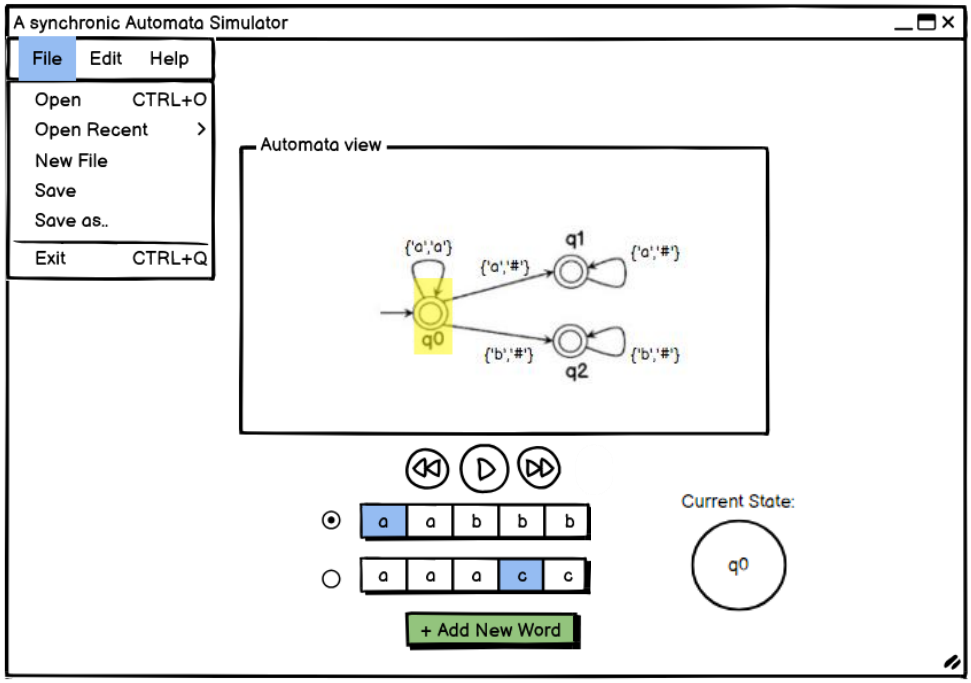

Fig 9: The final screen shows the simulation in action, with controls to stop, reverse, and forward the simulation. Users can add more words during the simulation, and for each word, the current position is highlighted.

Bibliography

[1] Bonakdarpour, B., & Sheinvald, S. (2023). Finite-word hyperlanguages. *Information and Computation, 295,* 104944.

[2] Finkbeiner, B., Frenkel, H., Hofmann, J., & Lohse, J. (2023, May). Automata-based software model checking of hyperproperties. In *NASA Formal Methods Symposium* (pp. 361-379). Cham: Springer Nature Switzerland.

[3] Gutsfeld, J. O., Müller-Olm, M., & Ohrem, C. (2021). Automata and fixpoints for asynchronous hyperproperties. *Proceedings of the ACM on Programming Languages, 5*(POPL), 1-29.

[4] Krithivasan, K., Balan, N. S., & Harsha, P. (1999). Distributed processing in automata. *International Journal of Foundations of Computer Science, 10*(04), 443-463.

[5] Krombi, W., Erradi, M., & Khoumsi, A. (2014, July). Automata-based approach to design and analyze security policies. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust* (pp. 306-313). IEEE.

[6] Singh, T., Afreen, S., Chakraborty, P., Raj, R., Yadav, S., & Jain, D. (2019). Automata simulator: A mobile app to teach theory of computation. *Computer Applications in Engineering Education, 27*(5), 1064-1072.

[7] Zielonka, W. (1987). Notes on finite asynchronous automata. *RAIRO-Theoretical Informatics and Applications, 21*(2), 99-135.