1 Introduction

Problem

Large and complex online systems no longer only involve servers performing deterministic operations; there are cases where human input is integral to completion of certain tasks or advancing through continuous processes. With the rise of the World Wide Web people have been prompted to not only interact with each other online, but with systems they can contribute to. Such systems can be classified as social machines, as the dependency of their states on the non-deterministic nature of human participants conveys the interweaving of computer logic with real-life knowledge. Social machines have received a fair amount of academic attention in the form of designs and frameworks to help form a more rigorous outline of their structure and behaviour. However, there are fewer attempts in developing related software for further experimental exploration and validation, and no known simulator of a formally defined social machine scheme exists at the time of writing this report.

Goals

This project aims at evaluating the feasibility of modelling social machine components via Turing machine simulations. There are three major stages the objective will branch into:

Stage 1 **Analysis and specification**
Based on the results of a literary review, formally define the different behaviours and components of the desired implementation, using a selection of models and classification frameworks. Compose an elicitation of functional requirements regarding those findings.

Stage 2 **Implementation**
In order to ensure that all effort is concentrated on implementing original features, rather than developing a full automata simulator from scratch, elect an appropriate open source simulation framework, and extend it. Derive the latter's implementation prerequisites from the already established specification, but allow additions if relevant, or compromises if necessary.

Stage 3 **Evaluation**
Assess the performance, accuracy, and scientific merit of the resulting application. Due to the lack of example data, a set of observed and theoretical scenarios will be used to assess the algorithms, and make further conclusions about the employed model.

Though principally sequential as each stage depends on the previous, it is anticipated that there may be some amount of overlap. Over the course of writing this report the many iterations over its content will most likely bring additional observations.

Scope

Clearly, this being an individual assignment bound by time strongly implies that motivations should be narrowed down to a feasibly-sized task. One of the precautions taken is the aforementioned choice of building an extension, instead of a new topic-focused program.

Another is adjusting to the insufficient amount of data by fabricating configuration cases that demonstrate the solution's ability to satisfy this project's goals specifically. Following review of literature it would seem that most previous endeavours into the domain have remained theoretical, which brings the need for custom data; populating such an expansive dataset for testing all possible configurations would be useful, but will have to be set aside as a prompt for future work. Finally, the lack of practical experiments to lean on prefigures the unlikelihood of a comparative evaluation, which will also have to be left for future analysis once alternative solutions are available.

Structure

To begin, chapter 2 is going to introduce the concepts leant on to formulate the basis of the problem being solved. Chapter 3 is going to focus on investigating the knowledge acquired from the literary review and lay out the specific requirements for completing the stated goals. Thereon, chapter 4 will document the process of fulfilling those requirements and any impediments encountered during its course. Subsequently, chapter 5 is going to report any experimentation results with various scenarios and determine how well the final product holds up to the expectations set. Lastly, chapter 6 will give the final run-down of the project and its outcomes, reflecting on the development process, and suggest any future work that could build on the conclusions provided.

Ethics

Even though the topic of social machines involves, and can be used to simulate, people and behaviour, this project does not include any outside participation, therefore safety, privacy and disclosure of medical, financial, any other sort of personal or statistical data are not issues of concern.

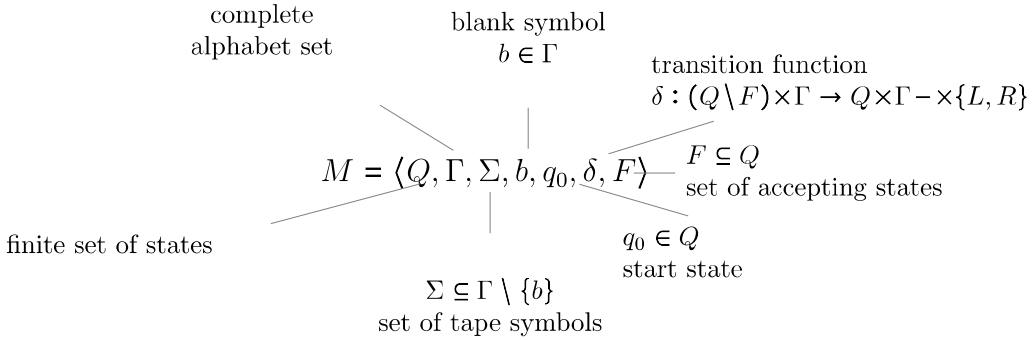
2 Literature Review

2.1 Turing Machines

The initial concept of the now famous Turing machine was created as a mechanism to prove that the *Entscheidungsproblem* is unsolvable [1] but turned out to have many further applications. While Alan Turing's take of machine learning [2] and human-like consciousness [3], although possible in theory to emulate with state machines, were way ahead of their time, his model of abstract automatic machines contributed greatly on the foundations of the computer science field.

Concept Essentially, a Turing machine is an abstract mechanism that can be in a number of pre-defined states. These states represent literal conceptual states of a system such as awaiting input or writing to a printer of which the machine can only be in one state at a time. To keep track of past inputs and actions the machine also has a tape (of hypothetically unbound length) which represents a string of symbols (characters) with a head pointing to a specific location within that string, acting as a form of memory. To navigate between these states and manipulate information stored on the tape, the machine also has a transition function which determines which states can be moved between based on what input from the tape and what to write to the tape. This allows formal definition of all 'Turing Computable' problems; i.e. all fully observable systems, like the modern computer (albeit at a very rapid rate) as defined below:

Formal definition [4]



There are variations of this definition with regards to accepting and rejecting (halting) states, but they are widely accepted as equivalent. This one follows the definition taken from 'Introduction to automata theory, languages, and computation' [4]. In section 3.2 single *accept* and *reject* states are used instead, following the 'Introducing the Ω-machine' paper [5].

2.2 O-machines

There exists a lesser known idea for an abstract machine by Turing, which was introduced in his PhD thesis - the *o-machine* [6]. Similar to automatic machines, his classic and most popular abstract computational model, o-machines can be described through function tables. The main difference between o-machines and other abstract machines, such as a-machines, is the black box element, called an 'oracle'. The latter's definition is not expanded upon in Turing's paper, beyond being depicted as *"some unspecified means of solving number theoretic problems"*. The only specification is that no details on the internal configuration of an instance of such a machine are available, as the oracle is supposed to be capable of solving non-Turing-machine-computable[1] problems, such as the Halting problem, that there is no known solution for. Essentially, imagine a machine that is able to provide the solution to a given problem, but there is no way of telling how it got that solution.

Figure 1: The computer that always has an answer? [7]

Oracle Definitions The typical definition of an oracle machine suggests that it is a unification of a Turing machine and an oracle, whose components are similar to those of a regular Turing machine, but also consist of unobservable internal configurations from which the function is derived.

According to van Melkebeek [8], an o-machine has the following components:

[1] In other words - theoretically 'unsolvable'

- Shared with traditional TMs:

  Work Alphabet. A set of symbols which can be written on the work tape

  Work Tape. An infinite sequence of cells that can either be empty or contain a symbol from the tape alphabet

  Read/Write Head. A component located on a single tape cell at a time that can move left or right on the work tape, as well as read and write[2] symbols on it

  Control Mechanism. Responsible for performing actions such as moving the head, manipulating data and switching states

- Additional components

  Oracle Alphabet. A set of symbols that may or may not be different from the work alphabet

  Oracle Tape. A semi-infinite tape of cells that can either be empty or contain a symbol from the oracle alphabet

  Oracle Head. Acts the same way as the R/W head, but on the oracle tape

  `ASK` and `RESPONSE` states. When the oracle enters `ASK` state, the current oracle tape contents are taken as a problem instance and given as an input to the oracle, which, upon finding the problem solution, replaces the work tape contents with its output; the head is moved to the beginning of the tape and the state is set to `RESPONSE`

Alternative specialist definitions of oracle machines also exist in order to abstract away the Turing Machine components. For example Adachi [9] suggests an oracle that takes a tape composed only of binary symbols that can respond to ASK states with YES or NO states in a single step and as such is specialised for decision problems (e.g. "Does the element x belong in set A?") . In order to keep future work as flexible as possible and enable application of the oracles onto functional problems (e.g. "What is the value of $f(x)$?") though, this paper will focus on the extensible Turing Machine based definitions for the remainder of this project.

2.3 Social Machines

The concept of a social machine is based on the idea of collaborative problem solving [10]. As defined by Tim Berners-Lee, they are *"processes in which the people do the creative work and the machine does the administration"* [11].

The combination of predefined machine transitions and probabilistic human input allows for introducing a plethora of outcomes for even a single system if it gets exposed to different selections of people. This is closer to the reality of modern computing, than the strict schemas of Turing machines. While non-deterministic counterparts of the latter exist, they are closer in potential to being abstract machines than actual representations of the ever-increasing and practically boundless total of branches each interaction presents in real

[2] or delete by placing a 'blank symbol'

time.

By reinforcing connection and collaboration between people, social machines indicate existing relationships among themselves. One type of association they exhibit is belonging to one another and thus having types that can be arranged in a hierarchical tree with the World Wide Web as an infrastructure being its root (though the scope of any given tree can easily be constrained such that any given service could be the infrastructure underpinning the associated social machine) (See Figure 2) [12].

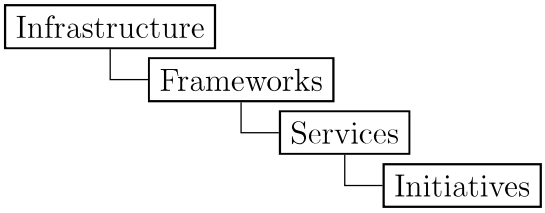


Figure 2: Granularity levels of social machines - *Initiatives* (a.k.a. *Projects*) using *Services* built upon *Frameworks* (communication protocols) built upon the Infrastructure

2.3.1 Social Machine and Computation Examples

Given a lack of evaluation data found during the literature review, a few real-world social machines have been identified with the possibility of simulation by reducing the domain and action set of the oracles and machine respectively to a few representative outcomes (both positive and negative):

Wikipedia
Serving as one of the most well-known cases of social machines, Wikipedia would not exist without the endless flow of human input. It confirms the capacity of social computation by being the largest encyclopaedia in existence, that is being updated at such a rate that physically publishing its content is absolutely redundant. Unfortunately, it is speculated that Wikipedia has recently been in decline [13]. A correct simulation of the lifetime of a similar (and preferably simplified) online collaborative knowledge project could predict tendencies in increases and declines of activity.

Games with a Purpose
Undoubtedly, gamification is a powerful motivation method - simply adding a score number next to a person's username sparks the desire to increase said number. As such, it can be used to encourage meaningful contributions to a real-world broad scope problems through an entertainment platform [14]. However, this genre of games is far from being as popular as other genres. The few notable examples that have been used to justify the importance and efficacy of GWAPs [15] are as of today defunct. Carrying out simulations of such games during their development may demonstrate potential issues, whose resolution may lead to the games' longer life and higher recognition.

The DARPA Network Challenge
The Defense Advanced Research Projects Agency organises annual prized challenges to stimulate problem solving through group effort in search of revolutionary research approaches.

In 2009 DARPA presented the efficiency of social computation and crowdsourcing with the following challenge: ten weather balloons were scattered across the United States and had to be located in the shortest time possible. The participating teams used various forms of social networking to achieve their goal, and the winning team managed to complete the task in just 9 hours by putting a referral system and cash incentives to use. Many of the strategies were essentially incorporating social computation to tackle the problem in a much faster way by engaging people, rather than by setting up a massive fully automated detection system [16]. As several of the winning strategies are publicly shared [17], they could potentially be simulated and compared given differing oracle behaviours as part of this project.

2.4 Tying It Together

This section has covered a number of novel automaton models and demonstrated how machines established solely on the basis of computable functions would fall flat in modelling present-day computing, considering their trademarks are *interaction* and *reactivity* [18]. Having in mind the notion that interactive systems are more expressive than algorithmic ones [19], it is possible to look into evolving the traditional models into something more applicable to the current technological circumstances.

In the case of modelling social machines with the help of Turing's brainchildren, exploration would involve tying together the explicit automatic machines with the enigmatic potential of oracles. The role of a computer system in that case could then be given to a continuously executing Turing machine, with work and output tapes[3], allowing for indefinitely long runs of information streams.

Following this, the black box characteristic of the oracles allows for assumptions about what o-machines could represent. One proposed interpretation is that a human brain can be simulated by an o-machine [20]. Within the context of modelling social machines, a set of oracles can be utilised as stand-ins for a group of human individuals. The capacity of answering questions that a normal computer can hardly give the answer to matches with the natural unpredictability of people.

Incorporating these ideas into a coherent formation has already been proposed in "Introducing the Ω-machine" [5], thus presenting a straightforward prompt for exploring the matter further in the next section.

[3]Possibly an input tape as well, if determined as needed.

3 Problem Analysis

In order to build any simulator, a model of the simulated environment needs to be established and followed. The first step is to break the concept in question down to its discernible parts. Then, the model chosen for this paper will be introduced, and its components will be linked to their corresponding role in the frame of the environment in question. This will help formulate the functional requirements for the practical part of the project.

3.1 Formalising the concept

So far, chapter 2 has covered the concept of social machines, oracles and automata that can be used to model existing computer systems, but as of yet no specification has been given for how these can be joined to represent a system. Here is defined an internet-based system representing a social machine (as laid out in figure 2) where the framework for communication is the passing of network data packets, the service is some modern computer system in and of itself, and the initiative is the high-level task laid out by the system's designers. This task, however, will incorporate not only the service, but a number of non-simulatable by TM actors providing some input which can easily be imagined as humans and represented by o-machines with their individual behaviours carried out by the oracles therein. If it can be theorised that the service is a Turing machine, the humans are o-machines and the data packets sent between are merely expanded symbols on a tape instead of packets on a wire then the system can be specified by an Omega Machine.

3.2 The Omega Machine

The proposed implementation of a social machine model is the Ω-machine, that serves as an extended model of the Turing machine, employing o-machines whose oracles act as human individuals [5].

Dependent oracles
The initial configuration based on the omega-machine idea is:

$$\Omega = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}, \Gamma_\omega, \delta_\omega \rangle$$

where Γ_ω is the oracle alphabet and δ_ω is the transformation function for the oracles.
Upon creation, each oracle belonging to a certain omega-machine instance only needs to have its `ASK` and `RESPONSE` states be specified. In this case:

$$O = \langle M, s_{ASK}, s_{RESPONSE} \rangle$$

where M is the automatic machine that the oracle belongs to and from which it *inherits* the oracle alphabet and the transformation function. With this setting all oracles have identical behaviour as they have the same transformation functions (unless it's probabilistic, but then the probabilities themselves will be the same). The first version of the social machine model will be built following this structure.

Divergent dependent oracles
If time allows, the the δ_ω component may be broadened into Δ_ω, which would support a different function for each oracle instance, looking like:

$$O = \langle M, \delta_\omega, s_{ASK}, s_{RESPONSE} \rangle$$

where $\delta_\omega \in \Delta_\omega$, meaning there is a variety of functions that could be assigned to the oracle from a predefined whitelist, and the a-machine would be modified into:

$$\Omega = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}, \Gamma_\omega, \Delta_\omega \rangle$$

In this way, the oracle cluster part of the model can come closer to resembling a crowd of diverse individuals, or imagined in a smaller scale - a group whose members make different choices, but are still constrained by the system and have the same 'language'.

Independent oracles
Ideally, the ultimate configuration goal would be to have o-machines be as independent from the core a-machine as possible:

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}, \rangle$$

$$O = \langle M, \Gamma_\omega, \delta_\omega, s_{ASK}, s_{RESPONSE} \rangle$$

where the o-machine has a reference to the a-machine, but the a-machine does not have a reference to any of the o-machines, mimicking the behaviour of a system, which is logically independent from the specific properties of its users. Likewise, the behaviour of users is defined separately from the system they are tied to Conclusively, a social machine, or an omega-machine, can then be defined as:

$$\Omega = \langle M, \mathbb{O} \rangle$$

The pairing of the two serves as a communication layer between M as technical platform, or the *machine* part, and $\mathbb{O}$ as the set of oracles, or the *social* part.

Summary
Since the definition of a Turing Machine is already globally accepted, the set of oracle components are the variant sub-components of an Omega machine that determine their implementation here. Each definition is an extension of the prior, and each trends towards oracles having more autonomy, independent of the Turing Machine core specification. As shown below, Omega Machines whose oracles are either dependent or divergent are essentially an extension of the classic Turing Machine, however in the case of Omega Machines with independent oracles the Turing machine is simply an element of the overarching system.

Dependent	Divergent	Independent
$M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject} \rangle$	$M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject} \rangle$	$M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}, \rangle$
$\Omega = M \mid\mid \Gamma_\omega \mid\mid \delta_\omega$	$\Omega = M \mid\mid \Gamma_\omega \mid\mid \Delta_\omega$	$\Omega = M \mid\mid \mathbb{O}$
$O = \langle M, s_{ASK}, s_{RESPONSE} \rangle$	$O = \langle M, \delta_\omega, s_{ASK}, s_{RESPONSE} \rangle$	$O = \langle M, \Gamma_\omega, \delta_\omega, s_{ASK}, s_{RESPONSE} \rangle$

3.3 Specification

3.3.1 Requirements Elicitation

Having established the formal mathematical definitions to use for modelling social machines within this project, it now falls to laying out a set of specifications that will guide the implementation of the framework. These goals are listed in order of importance rather than anticipated implementation order (red being an absolute must through to green being optional if time allows) as some pieces of functionality may be desirable and useful during implementation, but not critical to regard the software overall a success with relation to the goals defined in chapter 1:

Define Turing Machines	🔴	Preferably via GUI
Define Omega Machines	🔴	The model itself can be simple, as long as every oracle is connected to the main Turing Machine
Define the behaviour of an oracle	🔴	It should be possible to either pick from a whitelist, or ideally describe through user input
Simulate the execution of a Turing machine	🔴	Base it on an input state from each oracle; show the final output; show the output to each oracle
Continuously simulate the execution of an Omega machine	🔴	Demonstrate the input and output between each oracle including multiple executions; include communications between oracles
Save Turing Machines	🟡	Preferably via common transferable object serialisation format
Save Omega Machines	🟡	Full hierarchical specification including internal objects specifications and meta-data
Save oracle Machines and/or oracles	🟡	Including all components as behaviours
Enable connections between oracles	🟡	This requirement is tied to the rumour spreading scenario [5]. It is not crucial for the minimal viable product, but greatly desired.
Enable stepped execution of the Omega machine	🟢	Allow for debugging of individual oracle interactions, showing state at each step

3.3.2 Existing Software

As when considering the implementation of any piece of software, it never makes sense to reinvent the wheel unnecessarily. Given Turing Machines are also such a fundamental element of computer science, it makes sense that some form of design and simulation framework exist that can be taken advantage of here. As such, this section will evaluate existing Turing Machine design and simulation frameworks in order to get a head-start on the implementation goals laid out previously.

Criteria

Deciding on a framework to extend is a virtually irreversible given the tight timeline of the project (See Table 4 in Appendix A.1) and as such should be considered strongly before making a decision. The web contains a myriad of open source software options, most of which can be found on websites such as `sourceforge.net`. Notwithstanding, many, if not most, of those projects lack support, reviews, and proper documentation, which marks them as unsound options. To support the final choice, each option was evaluated against the following list of criteria (each with a tentative set of questions to consider):

Recognition An acclaimed, or at least favourably reviewed piece of software promises a good groundwork for the experiment and increases the likelihood of it being built upon in future work. If the application is not particularly popular, or no information is given about its feedback, existing affiliation with any institutions increases trust in its quality.
Questions to ask: *Where does the work come from? Is there any academic or industrial usage? Is the overall feedback, if any, thorough and positive?*

Language Implementation in a modern and widely utilised language ensures quick uptake.
Questions to ask: *What language is it written in? Is it an adequate choice for the task? Is it a familiar one?*

Maintenance Proper and frequent support promises fewer performance issues and bugs. Also, large involvement with the project indicates large amount of effort, and consequently - good quality of the product.
Questions to ask: *What are the activity levels on any associated forums or code repositories? How often have past updates been rolled out? When was the most recent update? Is the project still consistently maintained?*

Features A large code-base reduces the need to implement basic or common features.
Questions to ask: *What are the exclusive features, if any? How are machines stored? Is the program expressive enough to be extended efficiently? Are there components that can be reused in multiple ways? Is there any trivial or desirable for the project functionality missing?*

Documentation Well commented code will be easier to work with; any additional instructive materials, such as help pages, example tutorials, readme files, or full manual documents provide evidence of usability and accessibility; thorough change logs give more insight on development, and further inform about handled or potential issues.

Questions to ask: *What documentation is provided, if any? Is there a dedicated website, user manual, or a readme file? Are there any examples or tutorials? Is there a comprehensive change log for recent updates? Is the code well commented?*

Additional factors Any extra details that do not fall into the aforementioned categories are noted separately. They are not as deal-breaking and mostly reflect aspects that may not directly affect the goal, but are worth mentioning.
Questions to ask: *What is the user experience like? Are there any specific requirements to use the software? Are there any peculiarities?*

Options

On the next page is a selection of ten contenders that were considered.

Review and choice

The majority of open source projects available are implemented in Java and have user interfaces built with the Swing toolkit. After inspecting the source code contents and considering factors such as support, functionality, public feedback and quality of documentation, it was strongly decided in favour of JFLAP [21], as it excelled in all the aforementioned requirements in comparison to the rest of the projects. In fact, its diverse functionality and multitude of classes for any sub-entity that might be needed will enable easier and cleaner implementation of the social machine model.

Table 1:

Option	Recognition	Language	Maintenance	Features	Docs	Extra notes
JFLAP [21]	Educational software by the Duke University. The creator has been recognised as an ACM Distinguished Member Widely used - most popular candidate.	Java	Available since 1996 but still being updated; the latest release is from 2018, running on Java 8	Simulations for numerous other topics on automata and formal languages; XML files with a custom `.jff` extension	Dedicated website, books, papers, video tutorials	Comprehensive class hierarchy; UI in Swing.
Tursi [22]	Creator in AIS, University of Freiburg	Java	Written in 2013; v1.1 released in 2014	Custom .tm files; Exporting tables to state diagrams, history and break states, wildcards; Console mode option	Dedicated website with FAQ and a manual	UI in Swing
Owen's Turing Machine Simulator [23]	Used by Rensselaer RAIR Lab for testing and debugging while working on the 'Busy Beaver Superpaper' [24]	Java	Last updated in 2005 (according to file properties, no change log available)	Save the execution of a TM over time; Custom .tmo files.	Comments	It seems to be an internal project; UI in Swing
Tuatara [25]	The only project in the SourceForge selection with several recent downloads, and a review.	Java	Last modified in 2007	*Basic*	`Help` tab within app, very short readme file, comments	Screenshots of Windows XP; UI in Swing.
TMSimulator [26]	Made by students from the technical university of Vienna.	Java	Available since 2010; Last modified in 2011	*Basic*	A readme file	The only Java UI in JavaFX instead of Swing. Lots of typing and spelling errors
Alan [27]	A second term programming project at the University of Applied Sciences Rosenheim	Java	Last modified in 2005	*Basic*	SourceForge hosted website, function docs within the app, German documentation	The only project in the SourceForge selection to have a customised web page
TURING MACHINE [28]	A list of 29 institutions are listed are allegedly using the project, according to its website	JavaScript (jQuery)	Copyleft 2017	Runtime adjustable simulation speed	Tutorial PDFs, example machines	Web-based; Clean, modern interface
JSTuring [29]	Made by a mathematics tutor and lecturer at the University of Melbourne	JavaScript (jQuery)	Initially committed in 2014; Latest commit is from 2018.	Uses plain `.txt` files; saved machines are stored as Gists	Syntax explanations, some example machines	Web-based.
Tm [30]	Made by a staff member of Hobart and William Smith Colleges, Department of Mathematics and Computer Science	HTML5 (inline JS & CSS)	Original Java implementation dates back to 1996; current version is from 2017	Hard-coded sample machines	A help page	Web-based. All the code is in a single `.html` file
TuringSym [31]	No feedback or uses found	Delphi	Last modified in 2004	*Basic*	Not even comments	For Win32; Variable and function naming in Spanish

4 Method

With a clear set of specifications established based on the surrounding literature review, this chapter will cover the design and implementation of the extension of the JFLAP automaton design and simulation framework to include Omega machines. Design section 4.1 will cover a class-level overview of existing components and work to be done to achieve a functional software implementation while attempting to compensate for potential setbacks, while the Implementation section (4.2) will focus on the development environment and specific implementation details noted as part of the development process.

4.1 Design

Existing tools for Turing machine creation
The first step towards defining Omega machines is to allow definition of the core persistent Turing machine, re-using as many of the existing JFLAP components as possible. JFLAP uses a directed graph-style GUI, connecting labelled states (vertices) via transition functions (edges) that are each labelled with a table of symbols to read per tape (as the branch requirement necessary to be fulfilled in order to traverse the edge), write per tape and direction of head movement per tape. As described before, the GUI is built in Java Swing, meaning that each change to the machine is drawn (via the `gui.viewer.AutomatonDrawer` class) onto a `javax.swing.JPanel`, all of which is contained within a `gui.environment.AutomatonEnvironment` which manages the event driven interactions that trigger when the viewing Pane needs to be redrawn, in order to accurately reflect the current state of the machine. States and transitions are managed in the GUI by a set of tools listed under the `gui.editor` package. The `ArrowTool`, `DeleteTool`, `TransitionCreator` and `StateTool` are of primary interest. Most of these are fairly self-explanatory - creating or removing components within the environment. The `ArrowTool`, however, is the general attribute manipulation tool, which inspects the component that it is being used upon, in order to provide a context-aware menu for manipulating properties. This can be as benign as changing labels, or associated with more core functionality such as marking initial and final states, or modifying the transition function definitions directly.

Object-oriented Omega machines
As defined in section 3.2, the desired Omega machine model is primarily composed of a persistent Turing machine core and a number of oracle machines to interact with. However, to more discreetly define the relationships between oracles (for example when modelling Rumour Spreading [5]), it would also be useful to be able to select which oracles are neighbours (using an undirected graph-style UI similar to that of the Turing Machine). For encapsulation, any new components created as part of this project will be kept within the `social` package, and any non-trivial additions to other classes will be described as appropriate (i.e. changes to object factories and other such boiler-plate code will not be listed). As such, at a high level the `OmegaMachine` class can be defined as an object which will contain a single `automate.turing.TuringMachine` representing its core, with a set of labelled `OracleMachine`-s, and a set of bi-directional connections between them.

Recycling JFLAP components for Omega machine creation
Similar to the set of tools defined for manipulating Turing Machines, new tools for registering Omega Machine components will need to be added. Some of the existing tools will need to be extended to recognise these new components such as `DeleteTool` and `ArrowTool`, as they will need to be able to identify and display the new functionality based on the indicated component type and register subsequent changes. For creation, though, it will be simplest to implement dedicated tools, given these will represent entirely new components and containers. For example, while the PTM core of the Omega machine is really no different from a standard Turing Machine, the tool will need to be able to register either an existing Turing Machine definitions file, or create a new one, and then open a new environment for editing the Turing Machine directly (using the existing tools), while registering these changes with the parent OmegaMachine.

File structure

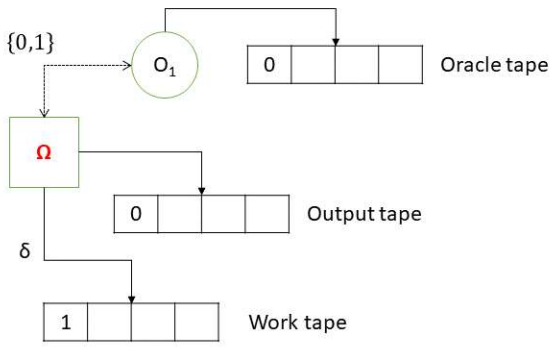


Figure 3: Lie-detector structure

Using these components, the simplest functional Omega Machine example that could be built is a lie detector, consisting of a single oracle which the core TM will send a symbol to, asking the question "What symbol did I just send you?". If the oracle responds with the correct symbol then the TM outputs `0`, indicating the oracle lied, or `1` otherwise. In this example, the Turing Machine will consist of 3 tapes (input, output, and communication with the oracle), 4 states (`init`, `transmit`, `false`, and `true`), and 3 transitions functions (writing a symbol to the oracle and reading either a matching or non-matching symbol from the oracle). Its structure mirrors figure 4.1 This is already representable by the XML[4] below:

[4]Even though JFLAP saves files with `.jff` extension, they are, regular XML files

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><!--Created with JFLAP 7.1.--><structure>&#13;
        <type>turing</type>&#13;
        <tapes>3</tapes>&#13;
        <automaton>&#13;
                <!--The list of states.-->&#13;
                <state id="0" name="q0">&#13;
                        <x>101.0</x>&#13;
                        <y>139.0</y>&#13;
                        <label>init</label>&#13;
                        <initial/>&#13;
                </state>&#13;
                <state id="1" name="q1">&#13;
                        <x>381.0</x>&#13;
                        <y>35.0</y>&#13;
                        <label>transmit</label>&#13;
                        <final/>&#13;
                </state>&#13;
                <state id="2" name="q2">&#13;
                        <x>380.0</x>&#13;
                        <y>141.0</y>&#13;
                        <label>false</label>&#13;
                        <final/>&#13;
                </state>&#13;
                <state id="3" name="q3">&#13;
                        <x>381.0</x>&#13;
                        <y>251.0</y>&#13;
                        <label>true</label>&#13;
                        <final/>&#13;
                </state>&#13;
                <!--The list of transitions.-->&#13;
                <transition>&#13;
                        <from>0</from>&#13;
                        <to>2</to>&#13;
                        <read tape="1"/>&#13;
                        <write tape="1"/>&#13;
                        <move tape="1">R</move>&#13;
                        <read tape="2"/>&#13;
                        <write tape="2">0</write>&#13;
                        <move tape="2">R</move>&#13;
                        <read tape="3">0</read>&#13;
                        <write tape="3"/>&#13;
                        <move tape="3">R</move>&#13;
                </transition>&#13;
                <transition>&#13;
                        <from>0</from>&#13;
                        <to>1</to>&#13;
                        <read tape="1"/>&#13;
                        <write tape="1"/>&#13;
                        <move tape="1">R</move>&#13;
                        <read tape="2"/>&#13;
                        <write tape="2"/>&#13;
                        <move tape="2">R</move>&#13;
                        <read tape="3"/>&#13;
                        <write tape="3">1</write>&#13;
                        <move tape="3">R</move>&#13;
                </transition>&#13;
                <transition>&#13;
                        <from>0</from>&#13;
                        <to>3</to>&#13;
                        <read tape="1"/>&#13;
                        <write tape="1"/>&#13;
                        <move tape="1">R</move>&#13;
                        <read tape="2"/>&#13;
                        <write tape="2">1</write>&#13;
                        <move tape="2">R</move>&#13;
                        <read tape="3">1</read>&#13;
                        <write tape="3"/>&#13;
                        <move tape="3">R</move>&#13;
                </transition>&#13;
        </automaton>&#13;
</structure>
```

In order to enable re-use of an encompassing Omega Machine in a similar manner, an `OmegaTransducer` will need to be implemented. It will parse and write XML to build to / save from an `OmegaEnvironment`.

As part of this transducer, the existing `file.xml.TMTransducer` can be used to save the XML from the Turing machine as the core, as demonstrated in the shown Omega machine XML file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><!--Created with JFLAP 7.1.--><structure>&#13;
	<type>omega</type>&#13;
	<machine>&#13;
		<ptmCore name="binary-TM">&#13;
			<!--The list of states.-->&#13;
			<state id="0" name="q0">&#13;
				<x>101.0</x>&#13;
				<y>139.0</y>&#13;
				<label>init</label>&#13;
				<initial/>&#13;
			</state>&#13;
			<state id="1" name="q1">&#13;
				<x>381.0</x>&#13;
				<y>35.0</y>&#13;
				<label>transmit</label>&#13;
				<final/>&#13;
			</state>&#13;
			<state id="2" name="q2">&#13;
				<x>380.0</x>&#13;
				<y>141.0</y>&#13;
				<label>false</label>&#13;
				<final/>&#13;
			</state>&#13;
			<state id="3" name="q3">&#13;
				<x>381.0</x>&#13;
				<y>251.0</y>&#13;
				<label>true</label>&#13;
				<final/>&#13;
			</state>&#13;
			<!--The list of transitions.-->&#13;
			<transition>&#13;
				<from>0</from>&#13;
				<to>3</to>&#13;
				<read tape="1"/>&#13;
				<write tape="1"/>&#13;
				<move tape="1">R</move>&#13;
				<read tape="2"/>&#13;
				<write tape="2">1</write>&#13;
				<move tape="2">R</move>&#13;
				<read tape="3">1</read>&#13;
				<write tape="3"/>&#13;
				<move tape="3">R</move>&#13;
			</transition>&#13;
			<transition>&#13;
				<from>0</from>&#13;
				<to>2</to>&#13;
				<read tape="1"/>&#13;
				<write tape="1"/>&#13;
				<move tape="1">R</move>&#13;
				<read tape="2"/>&#13;
				<write tape="2">0</write>&#13;
				<move tape="2">R</move>&#13;
				<read tape="3">0</read>&#13;
				<write tape="3"/>&#13;
				<move tape="3">R</move>&#13;
			</transition>&#13;
			<transition>&#13;
				<from>0</from>&#13;
				<to>1</to>&#13;
				<read tape="1"/>&#13;
				<write tape="1"/>&#13;
				<move tape="1">R</move>&#13;
				<read tape="2"/>&#13;
				<write tape="2"/>&#13;
				<move tape="2">R</move>&#13;
				<read tape="3"/>&#13;
				<write tape="3">1</write>&#13;
				<move tape="3">R</move>&#13;
			</transition>&#13;
			<!--Location parameters-->&#13;
			<x>168.0</x>&#13;
			<y>136.0</y>&#13;
		</ptmCore>&#13;
		<omSet>&#13;
			<oracleMachine id="0" name="Someone">&#13;
				<x>423.0</x>&#13;
				<y>136.0</y>&#13;
			</oracleMachine>&#13;
		</omSet>&#13;
		<cSet/>&#13;
	</machine>&#13;
	<tapes>3</tapes>&#13;
</structure>
```

Running simulations
Once everything is loaded, simulation of the automata can be initiated. In JFLAP, a Turing Machine under active simulation is represented by `automata.turing.TMConfiguration`. This configuration is generated at the start of simulation, based on the provided definition and inputs, containing the current "state" (i.e. current TM state and tape contents) and all information required to step through the machine (i.e. all other TM states and transition functions). For execution, though, the `automata.turing.TMSimulator` is responsible for indicating accept states and managing execution flow, at the most basic level checking branch conditions based on current overall state and applying transition steps to the configuration.

The execution loop
The Omega Machine simulations will function very similarly, though it will not be possible to directly extend the `TMSimulator` class as much of its functionality is implemented in-line, as opposed to being exposed via `protected` helper functions. To keep the project scope reasonable, existing code will not be refactored (especially given the pervasiveness of `MERLIN MERLIN MERLIN` comments in this class indicating unsavoury practices). However, extending the `TMConfiguration` class will be possible, as this is simply a container for TM-related member variables, which can be wrapped with the Omega machine context. As described above, this context will contain a number of oracle machines, each of which may have their own behaviour. To allow for this, and to take the unrestricted black-box implementation into account, the oracles should run arbitrary Java code when requested to execute by the `OmegaSimulator`. With this in mind, and in order to take into account the persistent nature of the core, the execution loop for the Omega machine will be as follows:

1. Simulate the Turing Machine in line with the existing JFLAP simulation loop, treating writes / reads to tapes $3..n$ as `ASK` / `RESPOND` calls to oracles $1..n-2$

2. When the simulation reaches a 'final' state within the TM, execution will be passed to each of the oracles in turn, along with the information left to them on their read / write tapes

3. After the oracles complete their execution, the TM will re-run from its 'initial' state but retaining all tape contents from prior executions, including modifications by oracles

4. These steps will loop until the TM reaches a 'final' state with no symbols on any tape except for the output tape, indicating there is no more work to be done and the machine has reached an 'accept' state.[5]

Gossiping neighbours
To include communication between oracles (such as in the rumour-spreading scenario described by Zhang et al. [5]), a `gossip` interface can be exposed on the oracles alongside `execute`, to be called as part of neighbouring oracles' execution cycles. This will require an oracle to be aware of its neighbours, but then the oracles will be free to communicate as

[5]Termination is not a compulsory step - the machine can run indefinitely long, similarly to a web application with an unknown lifespan.

many times as necessary during their execute cycle via a tape I/O interface, like that of the core Turing Machine.

Stepping through the execution of an Omega machine
The last sub-task, as laid out in section 3.3.1, is to enable debugging and tracing of Omega Machine executions. JFLAP again provides the base elements required via the components listed above for execution and state tracking, in tandem with the `gui.sim.SimulatorPane`, which contains a `gui.sim.ConfigurationController` for monitoring and stepping through states. Luckily, this `ConfigurationController` uses the more standardised `automata.Configuration` interface for manipulating state, and as such should require minimal modification to also work on `OmegaConfiguration` objects. Manipulating the `SimulatorPane`, however, may be more difficult as the `OmegaMachine` can be viewed from both inside the Turing machine and at the oracle interaction layer, potentially requiring greater environment manipulation or conceptualisation. As such, this will be left as the final implementation stage before completion, potentially being left for future work.

4.2 Implementation

Having explored the components available for building a Social Machine design and simulation framework, and identified the work required to achieve the goals set out in chapter 1, implementation can now begin. A functional product is one of the key deliverables of this project, in order to enable acceleration of future work concerning Social machines, as such this section will list key details discovered during implementation to assist with continuing to build upon the framework laid out here. As the majority of the class specifications were set out in section 4.1 though, it is not necessary to re-enumerate every class and function call. This section will, instead, focus on required tools and libraries, and significant changes / issues encountered on implementation compared to the design phase.

Setup This reference implementation of our Omega Machine design framework is made available on GitHub [32], based entirely on freely accessible tools and libraries. The project was built using Eclipse 2019-03 [33], using the Java SE-1.8 JDK [34] and a number of Apache SVG manipulation libraries that are required to compile and run JFLAP [35].[6]

Restrictions

- ✗ **XML structure:**
  Transducing machines was particularly difficult, not because of XML complexity, but the structure of the existing transducer logic. While XML DOM (Domain Object Model) manipulation libraries are in use within JFLAP, the existing code elects primarily to work from the root node of the object as opposed to relative nodes. This lead to some issues with the nested TM core, as data points, such as the number of tapes

[6] Required libraries are: `batik-dom-1.7.jar`, `batik-svg-dom-1.7.jar`, `org.apache.batik.ext.awt.1.6.0-20081006.jar`, `org.apache.batik.svggen-1.6.0-20081006.jar` and `org.apache.batik.swing-1.6.0-20070705.jar`. It should be noted that these versions are specifically required due to JFLAP incompatibilities with newer version APIs.

available, were expected in the root machine node rather than as part of the individual automaton, requiring specific workarounds as part of the `OmegaTransducer` to address. In future, it is considered that the `ISerializable` interface could be used along with the more modern XML manipulation APIs, to automatically generate and read from structured data based on class specifications, rather than needing to manipulate XML elements directly.

✗ **Tape count limitation:**
An unanticipated limitation of JFLAP is the hard-coded limit of 5 tapes on a Turing Machine. In chapter 2, it is suggested that each oracle is supposed to have 2 tapes (input and output) for communication with the core. However, if this is strictly followed, alongside an input and output tape for manipulating the core itself, the machine can only ever have a single oracle network, which obviously will not suffice. Upon lifting this limit though, it is immediately obvious why this restriction is in place, because having too many tapes per transition function (See fig.4) becomes unwieldy, potentially taking up the entire GUI with the tapes required for a single transition on larger networks, making it difficult for the user to identify which tapes map to which use-cases. Similarly, one could look in the opposite direction: as any multi-tape Turing machine

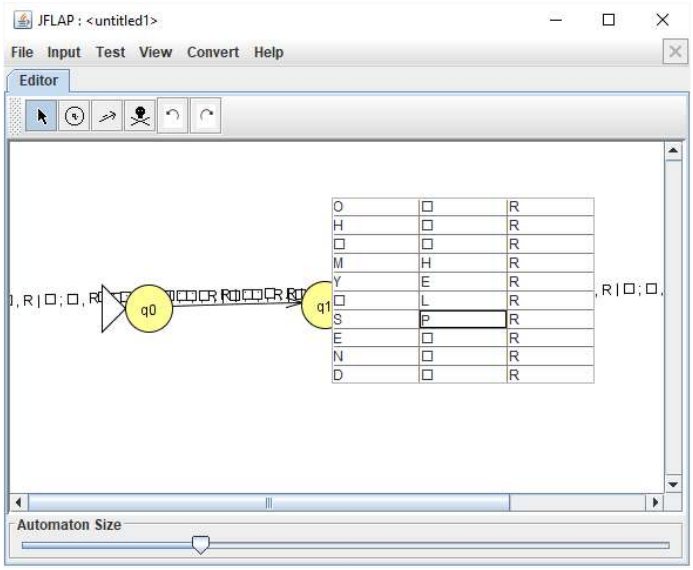


Figure 4: An example of a 10-tape T-machine, with only 3 states, the last of which is obscured by the transition table

can be represented by a single-tape one, as could the interactions with oracles be read from a single tape at pre-defined offsets. This again, however, would become infeasible with larger networks, both on the simulation side, navigating back and forth along the tape, and from the design perspective whereby the user would need to build this navigation logic into the Turing machine specifically tailored for the number of attached oracles. As a compromise, it was decided to retain a sensible limit on the number tapes available for this first foray into simulation, while also modifying the model to use only a single tape for I/O to each oracle, enabling functional small network design and simulation.

✗ **Code quality:**
On top of the issues listed so far, there are also inevitably some that arise just from working with legacy software. In this case, they spawn from it being initially created in an older version of Java[7] and expanded over the years with new features put on top of ageing code without too much refactoring. Modern Object-Oriented design principles, such as generic typing, are scarcely used. Despite the plentiful commenting that mostly provides the insight needed, there are still quite a few code blocks framed with "`MERLIN MERLIN MERLIN`" comments with no further explanation, leading to difficulty extending the software with new functionality without worrying about accidentally breaking underlying logic. Another contributing factor to the suboptimal nature of the framework, given that it entered development two decades ago, is the use of Java Swing, which is now an outdated graphic user interface library. The resulting GUI implementation is hundreds of Java code lines that would have been much easier to maintain and modify through, e.g. FXML files instead. It is a minor inconvenience compared to the functional elements of the code, though it may hinder development time given surrounding documentation may be out of date. Additionally, it is slightly unsightly.[8] Notwithstanding, since improving code quality is not the aim of this project, and time constraints are in place, focus has been driven towards implementing the required functionality as part of the project goals.

✗ **Oracle behaviours not definable at runtime:**
A desirable feature identified as part of the design process for the framework was the ability to have multiple oracle machine functions as part of a single Omega machine, enabling the simulation of multiple actors in a scenario with different behaviours. This is implementable by specifying a `social.oracle` package from which to select, but given JFLAP is implemented in Java, there is no way to dynamically load oracles at runtime from within the language. The ideal representation would include a text box per oracle that can either inherit pre-defined behaviour, or allow arbitrary user customisable code for run-time interpretation which it was theorised could be implemented using Java Lambda functions; this however is not the case. Potential solutions are to include the JDK into the framework for compilation at simulation time to execute separately within the JVM, or calling out to a functional language interpreter sub-process (such as Python or JavaScript), but as these require extensions to the code-base too great to consider here, users requiring custom oracle functionality will need to implement it within the project directly.

Despite the aforementioned issues, the framework was completed to a viable degree. As such, constructing social machines for simulation and evaluation can be looked into.

[7] Most likely 1.0 given the known timeline [36]

[8] Yet, most of the other simulators written in Java, all of which newer than JFLAP, use the same GUI libraries anyway, so the archaic experience is inescapable.

5 Evaluation

At this stage, JFLAP's Omega Machine definition and simulation framework has been built and now it falls to proving that the implementation is satisfactory based on the goals laid out in chapter 2, and providing metrics to improve upon in future work, given the restricted development window and compromises taken laid out in chapter 3. Given at the time of writing this appears to be the first piece of work towards *simulating* social machines, there are no alternatives for comparative evaluation against, nor a collection of metrics established to measure and report for empirical evaluation. This section will enumerate a set of simple example scenarios, reporting on qualitative metrics for future comparative analysis against this base implementation and relating back to the goals they satisfy. Once complete, the section will finish with some general comments on the software's overall fit for purpose and a critical reflection.

5.1 Lie Detector

The simplest possible Ω-machine that can be defined is one with a single actor alongside the core; a useful machine concerning itself with a single unknown actor could be that of a lie detector. In the simplest terms, the core machine will provide the actor with a symbol and 'ask' it what symbol it was passed. On receiving a response, the core will report whether this was the correct symbol (truth) or otherwise (a lie). To achieve this, on top of the base Turing Machine for manipulating interactions with the actor oracle, a `LyingResponderOracle` was implemented with a customisable parameter in the range `0.0-1.0` indicating the probability of 'lying', and placing a different symbol back on its communication tape. As shown in appendix A.4, the software is able to not only define and simulate such a machine, but also step through the states whereby interaction with the probabilistic actor takes place while producing the expected results, satisfying goals the majority of the specification set out in section 3.3.1. Unfortunately, this machine is so simple that there are no meaningful measurements to be derived from it, other than the responsiveness of the application (i.e. each configuration step, including those determined by the oracle, takes less than 0.1 seconds), but it still serves as a minimum requirement for any future simulator to be able to replicate this basic decision problem's definition and simulation.

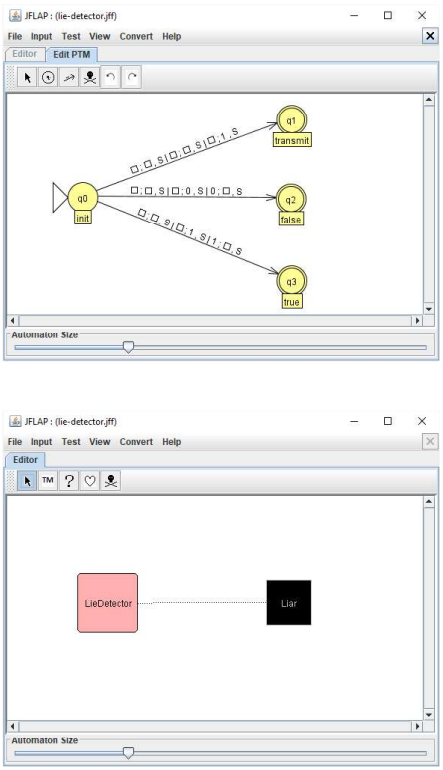


Figure 5: A simple lie-detecting Omega Machine

5.2 Collective Intelligence

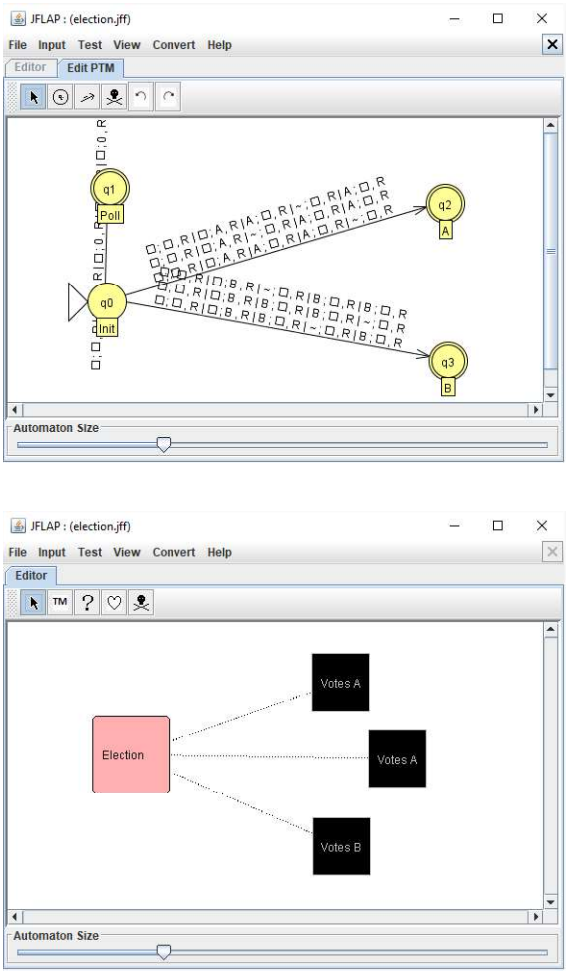


Figure 6: A simple collective intelligence gathering Omega Machine

In 'Introducing the Ω-machine' [5], the idea of gathering collective intelligence from a group of actors in Omega Machines is introduced, which given all of the examples depicted in section 2.3.1 are implementations of gathering information from unknown actors, is an important concept to cover here for comparison against future implementations. In this simplified example, corroboration of distributed information (as Wikipedia might be considered) could be considered a voting application whereby actors vote on the correct definition of information (in the same way individuals contribute and verify article edits). At a high level, the core 'polls' all actors for information (though Wikipedia doesn't actively request information, this could be considered analogous to notifying interested parties to an edit of a page) on a topic, as indicated by the symbol placed on each oracle's communication tape. In response, the oracle replaces that symbol with a new candidate symbol representing that information in a condensed manner to be communicated back to the service. On receipt of the information, the service simply chooses the response with the most 'votes' for publishing to the output tape. To replicate this system, an `InformationOracle` was implemented, which can be configured to hold a string of information on setup and retrieved on `ASK`ing with the corresponding symbol index to abstract away the deep knowledge acquisition process as part of researching a topic. On the service side, however, in order to count votes it must either be configured to track these on a work tape while iterating over oracles, or be pre-loaded with all possible combinations of votes as transition functions to the relevant output state. For simplicity in this example, the transition functions have been hard-coded to outputs to avoid the tape manipulation and comparison logic but neither of these implementations scale particularly well with increasing numbers of options to vote upon, nor voting actors. While this implementation is functional and relatively easy to understand, with n possible outcomes and m actors, in the worst case each poll requires evaluating n^m possible transition functions or maintaining a large amount of tape manipulation logic depending on implementation. As such, it would be worth investigating re-usable Turing Machine sub-components and abstractions for manipulating communications with large numbers of external sources.

5.3 Rumour Spreading

Another possibility of using Ω-machines is to simulate rumour spreading [5] among a network. This is reflected particularly well in the real world by marketing on social media platforms where each actor in a network of friends may or may not be shown an advertisement which may or may not be shared among their network and result in an observable purchase (through some tracking token). For this example, only a perfect product is considered[9] that is *always* shared with friends and *always* purchased when heard about (as implemented in the `RumourSpreadingOracle` class). With this information, the service can then determine connections in the network between known actors, by probing them individually as a form or graph exploration. To this end, the service iterates over each known actor sending them the advertisement, then after 'waiting' for some time window, checking to see which actors purchased the product by placing an order on the communication tape. Once the orders are received, the service can then output which actors are connected by identifying which actors placed an order that were not advertised to in that iteration (see appendix A.4 for an example trace). Similar to the Collective Intelligence example though, the design suffers with scale albeit more in machine complexity than time complexity. This derives from the fact that in the Collective Intelligence example, the machine is looking to extract the same information repeatedly from every oracle, but in this graph exploration each `ASK` and `RESPONSE` cycle needs to be context aware, meaning that for each oracle added to the network, a new set of branches for every existing execution branch on the Turing Machine needs to be added (see figure 7). As such, this demonstrates the need for a component to better enable conditional looping over the set of oracles from within the Turing Machine, in order to model larger networks.

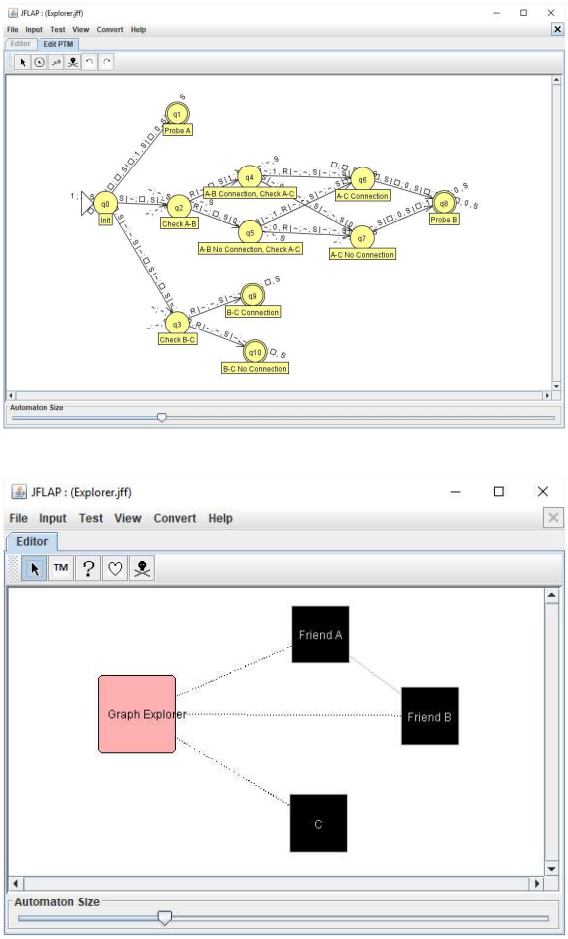


Figure 7: A simple rumour spreading Omega Machine

It is also worth noting that the initial requirement to enable debugging of communications between oracles was not able to be implemented here. Given the black-box style execution of the oracles represents a single step in configuration of the overall Ω-machine, it would require the implementer of the oracle to yield control back to the encompassing simulator and provide internal configuration information which is not the purpose of the o-machine as

[9] What that is however, is left as an exercise to the reader.

defined in [6]; meaning this is an acceptable implementation detail to be absent.

5.4 Software Suitability

Overall, according to the specifications laid out in section 3.3.1, this project is successful as all requirements (stopping just short of debugging individual oracle interactions as part of the Omega machine) have been fulfilled by the chosen / implemented software. However as part of this evaluation, a number of unanticipated deficiencies have been identified with the concept of designing Omega machines in the same way as traditional Turing machines, the most significant of which is scalability. In general, while social machines can be used to model interactions with a small number of unknown actors, when modelling public internet-based systems the number of actors can be anywhere from in the hundreds all the way through to millions of unique individuals. Each of these actors of require `almost` identical treatment, but using the current set of components available for building Turing machines, this would require a multitude of branches and large amounts of tape manipulation logic, making design at that scale infeasible. While one solution might be to abstract the system away again to a higher level, focussing instead on oracles representing large groups of actors and the overall result returned from a set, this is a limitation that should be acknowledged when considering the applicability of the software provided here.

5.5 Requirements completion

Define Turing Machines	✓	Satisfied by existing JFLAP implementation
Define Omega Machines	✓	Satisfied by implementation of new tools and classes on top of JFLAP definitions
Define the behaviour of an oracle	≈	Satisfiable by creating classes within JFLAP source, but not at runtime
Simulate the execution of a Turing machine	✓	Satisfied by existing JFLAP implementation
Continuously simulate the execution of an Omega machine	✓	Satisfied by implementation of new simulation components on top of JFLAP components
Save Turing Machines	✓	Satisfied by existing JFLAP implementation
Save Omega Machines	✓	Satisfied by extending JFLAP XML transducers to include new components
Save oracle Machines and/or oracles	≈	Class-level behaviours and parameters are saved to XML, but specific execution details are omitted
Enable connections between oracles	✓	Satisfied by oracle class definition and interface implementation
Enable stepped execution of the Omega machine	≈	TM and oracle interactions observable by building on top of existing JFLAP interface, but oracle interactions and state remain unobservable to outside processes

6 Conclusion

Reflection

In chapter 1, the interest surrounding social machines was identified, along with the utility in being able to formally specify and simulate them. In chapter 2, the building blocks underpinning automata definition were explored, leading to the choice of Omega machines as a mathematically formulated model around which to build social machines, by taking advantage of the existing software for designing and simulating similar Turing machines exhibited in chapter In chapter 4, the class structure of the chosen JFLAP design and simulation framework was investigated and the required work to satisfy the project requirements established.3. In chapter 5, predominant implementation discrepancies were identified to assist with future work upon the framework, and finally in chapter 6 the final software was put to the test by analysing the process surrounding design and implementation of a number of example social machines with potential real world application.

Following the evaluation of the completed software, a number of unforeseen factors were discovered, impacting the process of manually defining Ω-machines, primarily surrounding the scalability of hand-manipulating states and transitions. By all accounts, the project goals and software requirements have been satisfied, but it would be foolish to announce investigation into the possibilities of modelling Ω-machines complete, given the described difficulties in implementing the example scenarios. It would also be short-sighted to call this project a failure, given the collation of knowledge, crystallisation of conceptual knowledge into concrete class implementations, elicitation of previously unregistered issues surrounding working with Turing machines at scale, and suggestion of potential solutions to these issues through further work.

Improvements

Having now overseen the project to completion, if an attempt were made to re-tackle this same problem with the knowledge in-hand gained here, multiplexing features of the UI and framework itself would be the elements to focus on, as opposed to oracle simulations. With sufficient time, it would be a significant improvement upon JFLAP to recreate the same behaviour through an interpreted language, implementing the manipulation and simulation logic in order to promote more active development both by freeing the software from the shackles of legacy language features and libraries, and being 'maintained' and distributed via an antiquated version control system, alongside better reflecting the rapidly iterative design-based goals. It would be unreasonable to expect a single researcher to be able to recreate this framework and achieve the results expressed in this paper within the same time span, so instead sticking with JFLAP, but creating more custom specialised state / transition amalgamation components for use within the TM cores at an early development stage, is advised. The only other major improvement would be to contact the social machine community [37] to gather and establish quantitative metrics for evaluation of the final framework, which, as well as providing a benchmark against which to compare future work, may have promoted interest in expanding upon this work by advertising it to the established community at large.

Future work

As previously mentioned, two major improvements on this work would be to both rewrite the existing Java code using a more modern API and putting consideration into specialised multiplexing components for manipulating functions over large numbers of oracles / tapes within Turing machines. However, future comparative works might centre around utilising information cascades as posited by Luczak-Rosch [10] to represent the same examples provided here and a similar simulation framework for modelling the transfer of information through them. This might (i) help identify new social machine interactions which need to be considered in order to create a fully representative model, (ii) function as an entirely independent model that makes social machines easier to design and simulate than the framework provided, or (iii) be used as part of a network representing the interactions between social machines themselves.

Closing thoughts

All in all, this has been a successful dive into the world of system modelling frameworks and social machines, highlighting future benefits of continued research into this field, while also identifying a set of key weaknesses using the existing components that need to be rectified before the full potential of the work can be realised. The broad scope and potential for novel application design of social machines is an exciting opportunity to bring new knowledge to the fundamental understanding of machine modelling for computer scientists everywhere.