TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TEEMU PÄÄKKÖNEN

HUMAN INTERACTION SIMULATION SOFTWARE FOR INFORMATION RETRIEVAL RESEARCH

Master of Science thesis

Examiner: Prof. Tommi Mikkonen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 13th January 2016

ABSTRACT

TEEMU PÄÄKKÖNEN: Human Interaction Simulation Software
for Information Retrieval Research
Tampere University of Technology
Master of Science thesis, 75 pages
April 2016
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Tommi Mikkonen
Keywords: Information Retrieval, simulator, stochastic, Monte Carlo

The concept of Information Retrieval deals with the obtaining of information from information sources, such as libraries, archives, databases, or the internet. It is a widely studied subject. Interactive Information Retrieval is the process where users interact with a search engine, using its interface to make queries in order to find documents relevant to their information needs.

Interactive Information Retrieval experimentation with real human users is costly, requiring time and money, and often runs into problems stemming from the use of human subjects. In lieu of human users, the experiments can be conducted via simulation. Simulations have been widely used in Information Retrieval research, but none of the previous simulators have been useful for generic research for various reasons.

In this thesis, due to the unavailability of generic purpose simulators, a project to create an Interactive Information Retrieval simulator software was carried out. The software was designed to support generic user modelling via a purpose-built language that describes the user model. The language was designed to support nearly any kind of user model that may be used in an Information Retrieval context.

The finished software was evaluated by running experiments with an established user model that has been previously used in Information Retrieval studies. The software was found to be able to replicate the results of the human users without significant statistical difference. However, the exact behaviour of the human users could not be replicated very accurately. Nonetheless, the software was found to serve its purpose well, being a useful tool for Information Retrieval research.

TIIVISTELMÄ

TEEMU PÄÄKKÖNEN: Käyttäjävuorovaikutuksen Simulointisovellus Tiedonhaun Tutkimukseen
Tampereen teknillinen yliopisto
Diplomityö, 75 sivua
Huhtikuu 2016
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: Prof. Tommi Mikkonen
Avainsanat: tiedonhaku, simulaattori, stokastinen, Monte Carlo

Tiedonhaku (engl. Information Retrieval) käsittelee tiedon hankkimista erilaisista tietolähteistä, kuten kirjastoista, arkistoista, tietokannoista tai internetistä. Tiedonhaku on laajasti tutkittu aihe. Vuorovaikutteinen tiedonhaku (engl. Interactive Information Retrieval) (IIR) on prosesssi, jossa käyttäjät ovat vuorovaikutuksessa hakukoneen kanssa, käyttäen sen käyttöliittymää kyselyden tekemiseen löytääkseen tiedon tarpeensa täyttäviä dokumentteja.

IIR-kokeiden järjestäminen vaatii normaalisti koehenkilöiden käyttöä, mikä on kallista ja vaatii aikaa sekä rahaa. Ihmisten sijaan, kokeet on kuitenkin mahdollista suorittaa simuloimalla. Simulaatiota on käytetty runsaasti tiedonhaun tutkimuksessa, mutta useista syistä johtuen simulaatiosovellukset eivät ole olleet käyttökelpoisia yleiseen tutkimukseen.

Tässä diplomityössä toteutettiin projekti IIR-simulaatiosovelluksen tuottamiseksi, koska käyttökelpoisia simulaattoreita ei ollut saatavilla. Sovellus suunniteltiin niin, että se tukee geneeristä käyttäjämallinnusta tarkoitukseen tehdyn käyttäjämallin kuvauskielen avulla. Kieli suunniteltiin niin, että se tukee lähes mitä tahansa tiedonhaun tutkimuksessa mahdollisesti käytettävää käyttäjämallia.

Valmiin sovelluksen käyttökelpoisuutta arvioitiin suorittamalla kokeita eräällä yleisellä käyttäjämallilla, joka on laajasti käytetty tiedonhaun tutkimuksessa. Sovelluksen todettiin pystyvän toistamaan todellisten käyttäjien tuottamat tulokset ilman merkittävää tilastollista poikkeavuutta. Ihmisten todellista vuorovaikutusta ei kuitenkaan pystytty tarkasti toistamaan. Siitäkin huolimatta sovelluksen todettiin soveltuvan hyvin käyttötarkoitukseensa: tiedonhaun tutkimukseen.

ALKUSANAT

> Evil begins when you begin to
> treat people as things
>
> *Tiffany Aching*
> *Terry Pratchett*

Tämä diplomityö on vuosikausia hauduteltu, pitkään ja hartaasti kypsytetty, iloa, raivoa ja turhautumista aiheuttanut iisakinkirkko. Kirjoitustyö ei olisi takuuvarmasti milloinkaan valmistunut ellei TTY olisi pakottanut vanhoista tutkinto-ohjelmista pois siirtymistä lukuvuoden 2015–2016 loppuun menessä. Näin ollen suurin kiitos kannustuksesta kuuluu TTY:n tutkintojen suunnittelijoille, jotka jaksavat uupumatta solmia opintosuunnitelmat uusiin umpisolmuihin, vuodesta toiseen.

Työ itse käsittelee käyttäjäinteraktion simulointia, tarkoituksenaan auttaa tutkijoita tuottamaan ihmisten käyttöön paremmin taipuvia järjestelmiä. Paradoksaalisesti, tätä siis yritetään saavuttaa korvaamalla ihmiset ohjelmalla. Ovathan oikeat ihmiset toki vain tutkimustyön esteenä. Pahuus alkaa koodista.

Haluaisin esittää pahoittelut läheisille, ystäville, työtovereille, sekä kaikille muille jotka ovat tarkoituksellisesti, tahattomasti tai tyhmyyttään joutuneet osaksi tämän diplomityön luontiprosessia. Olette kestäneet luomisen tuskaani paremmin kuin itse pystyisin ikinä. Toivottavasti mielen- ja ruumiinterveytenne on sen kestänyt. Omastani en enää mene vannomaan.

On esitetty väite että tiedon hakeminen epäonnistuu aina, paitsi vahingossa. Ehkä tekemästäni työstä on apua väitteen kumoamiseen. Softan tekeminen kuitenkin onnistuu joskus. Vahingossa.

Tampere, 17.3.2016

Teemu Pääkkönen

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

CG	Cumulated Gain
CLI	Command Line Interface
DCG	Discounted Cumulated Gain
EPIC	Executive Process-Interactive Control
GOMS	Goals, Operators, Methods and Selection rules
IDF	Inverse Document Frequency
IIR	Interactive Information Retrieval
IR	Information Retrieval
MAP	Mean Average Precision
nDCG	Normalised Discounted Cumulated Gain
NIST	National Institute of Standards and Technology
OS	Operating System
SCXML	State Chart XML
SERP	Search Engine Results Page
TF	Term Frequency
TREC	Text Retrieval Conference
W3C	World Wide Web Consortium
WWW	World Wide Web

Δ_X	transition set of IR simulator automaton
δ	automaton transition function
Σ	set of automaton input symbols
A	accumulator
C	set of conditions in IR simulator automaton
c	condition in IR simulator automaton
E	set of events in IR simulator automaton
F	set of final automaton states
f	function
P	set of transition targets in IR simulator automaton
Q	set of automaton states
q	automaton state
q_0	initial state of automaton
R	subset of U

r	result document in IR simulator automaton
T	sequence of transitions in IR simulator automaton
t	transition in IR simulator automaton
U	set of result document sets in IR simulator automaton
u	result document set in IR simulator automaton
V	probability function in IR simulator automaton
X	random variable, input of IR simulator automaton
x	random value for X

1. INTRODUCTION

Information Retrieval (IR) is a concept of information science that, according to Baeza-Yates and Ribeiro-Neto (2011), encompasses the "representation, storage, organisation of, and access to information items". With the notion of information items they refer to things such as documents, web pages, catalogues, records. Manning et al. (2009) add that IR often focuses on finding *unstructured documents* from within large collections. The concept revolves around the idea of obtaining information pertaining to an information need.

Baeza-Yates and Ribeiro-Neto (2011) note that IR is actually an area of computer science since it deals with computer systems. Information retrieval, however, has its roots in searching library card catalogues, and therefore does not concern itself on whether the document collections are digital or physical.

The notion of *Interactive Information Retrieval* (IIR) refers to the process of search engine usage where users interact with the system by actions such as issuing queries, scanning the result list, reading the result documents and assessing them. In addition to the search interface, the concept also deals with other surrounding aspects, such as the social context and the setting. For example, using a typical web search engine to find information can be considered an instance of an IIR process. (Ingwersen and Järvelin 2005)

IIR research can be executed through multiple types of experiments. Different types include the observation of human subjects performing real or simulated search tasks, and laboratory research where no human subjects are used. For observational experiments, the researcher needs to set up an environment where searchers can interact with a search engine in a controlled and supervised environment. They are given a task to complete, and their actions are recorded. Due to the need of human subjects, experiments can often be expensive and time consuming. The issues of learning effects, fatigue, and the scarcity of available subjects also present hurdles for IIR research. (Azzopardi et al. 2011)

In order to skirt the obstacles stemming from human aspects, a simulator software that models a human searcher's behaviour can be utilised (Azzopardi et al. 2011). While many experiments have used software-based simulators before, the software has either not been available for others to use, or has not been generic enough for general experimentation. Therefore, a project for creating a new generic IR simulator software had to be undertaken, with the aim of allowing researchers to carry out experiments that revolve around human interaction with search engines. It was postulated that a good enough simulation would produce results indistinguishable from actual human subjects, eliminating the need for actual humans entirely, and in the process also accelerating the speed of new research.

The objective of this thesis is to produce a simulator software for the research of Interactive Information Retrieval. The purpose of the software is to simulate the interaction between a user and an Information Retrieval system. The goal is to create a simulator software that can run simulations with arbitrary user models, producing results and behaviour that accurately replicate the actions of a human user.

The scope of this thesis is limited to building software that deals with simulating user interaction with interactive IR systems that find digital documents using keywords input by the user (a *query*) and return a ranked list of results. For example, simulating the interaction with web search engines, such as Google, falls within the scope, while simulating structured searching, such as making SQL queries, does not.

Chapter 2 outlines some basic background information on the subjects of IR, simulation and user modelling. Chapter 3 explains how the software was designed, while Chapter 4 documents the details of implementing it. Finally, the software and the process of creating it are evaluated in Chapter 5, and in Chapter 6 the thesis is wrapped up and conclusions made.

2. SIMULATING INFORMATION RETRIEVAL

Simulating human behaviour is a widely studied subject. In order to successfully simulate a user of a system, the behaviour under analysis has to be reduced into a *user model* that describes how the user interacts with the system. The user model must also describe, or include as a separate entity, a model of the system being used. This user model is then ingrained into the simulator software that then operates on it, producing data that can then be compared to real users. (Azzopardi et al. 2011)

From the very beginning, it was decided that the software should allow arbitrary user models, making it easy to experiment with any aspect of user behaviour. The main question was whether it would be possible to create such a software that, when given a good enough model, would predict real-life behaviour so accurately that it could be used for IIR research.

This chapter discusses how users of Information Retrieval systems can be simulated. Section 2.1 gives an overview of general IR terminology, Section 2.2 explains IR systems' inner workings from a theoretical viewpoint, Section 2.3 then shows how the performance of IR system users can be evaluated, and Section 2.4 discloses how user models can be created and how to simulate human behaviour.

2.1 Information retrieval terminology

Terminology related to IR is used throughout this thesis. Some of the IR terminology is explained here for later reference. Definitions are given in accordance with Manning et al. (2009), Croft et al. (2010) and Ingwersen and Järvelin (2005). Figure 2.1 also illustrates the relationships between the terms.

An *information need* signifies a perceived lack in the knowledge of the searcher. Information need is often the driving force behind search. In research context, the information need is often formalised into a *topic*.

An *information item* is a basic unit of information that may be retrieved by an IR system. An information item may, for example, be a document, a web page, a video, or a sound recording. IR systems manage the storage and retrieval of information items. In this thesis, information items are typically referred to as documents.

Relevance refers to the usefulness of an information item for a given information need. Relevance may be simply given in a binary fashion (relevant or non-relevant), or it can be multivalued and multidimensional, having different levels of relevance for different aspects of the search context. Croft et al. (2010) note that calculations that involve relevance actually deal with the *probability* of relevance, since the actual relevance of a document is a very subjective matter.

A *query* is a request for information that communicates the information need of the searcher. It is presented to an IR system in expectation of a reply consisting of information items relevant towards the information need. The IR system responds to the query in accordance to its indexing and retrieval algorithms (see Section 2.2).

Result documents are the information items that an IR system returns in response to a query. Depending on the type and complexity of the system, the results may be ranked, giving the result list an order, or they may be unranked, making no difference between different result documents. Results are often separated into pages. In the IIR context, these are often called *search engine result pages*, or SERPs. For any single result, a SERP often contains only a brief summary, a *snippet*, that in the web search engine context also contains a link to the actual result document.

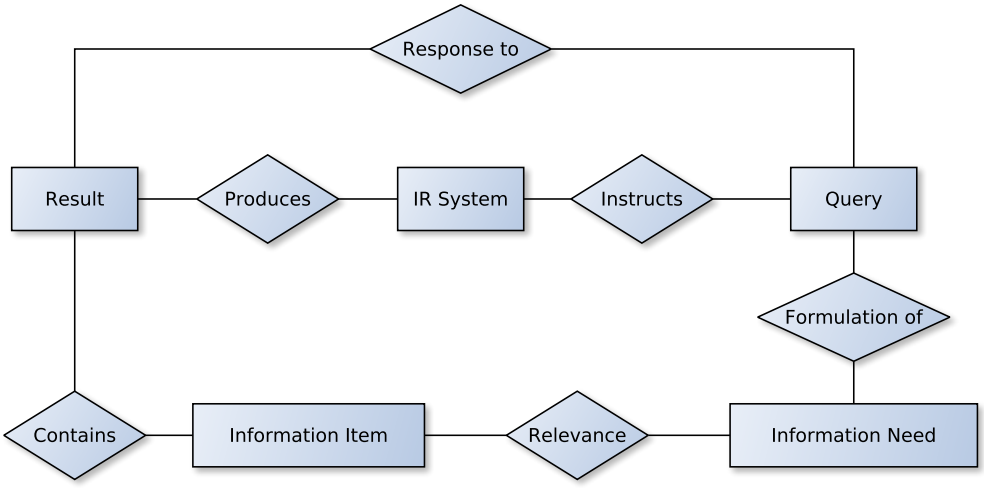

Figure 2.1 *An entity-relationship diagram of IR terminology*

2.2 How information retrieval systems work

While this thesis does not deal with details on how exactly IR systems find documents, just on how they respond to user input, a brief explanation on the internal workings of such systems helps understand the context. The algorithms and data structures of an IR system decide how different types of queries can bring forth better results than others.

2.2.1 Indexing

An IR system has a collection of documents that it allows users to search. Searching through the entire collection every time a user enters a query would be extremely inefficient. To avoid that, the document collection is *indexed*, so that when a query is made, instead of searching through the entire document collection, only the index, or indices when there are multiple, are scanned, speeding up searching enormously. (Manning et al. 2009)

A typical index is a word-oriented *inverted index* that contains a *vocabulary* of *terms* and the mapping of term *occurrences* in documents. This mapping is the reason why it is called an inverted index: the text can be reconstructed using the index. (Baeza-Yates and Ribeiro-Neto 2011)

To build a vocabulary of all the words that occur in a document, document contents are scanned, in order to turn the contents into a list of *tokens*. Loosely defined, a token is a piece of text, such as a word or a phrase. All occurrences of all encountered tokens are recorded while scanning the document. The end result is an index of locations of tokens within the document. (Manning et al. 2009)

A simple type of vocabulary contains a term-document matrix that simply lists how many times a term occurred in a document. This kind of index can be used in *TF-IDF* (term frequency–inverse document frequency) based ranking of documents. Document ranking and TF-IDF are given a more detailed take in Subsection 2.2.3. (Baeza-Yates and Ribeiro-Neto 2011)

The second part of the process of creating an inverted index involves building the map of occurrences, also known as the *postings list*. The list records which terms appeared in which documents. (Baeza-Yates and Ribeiro-Neto 2011)

In order to successfully index a document, some linguistic preprocessing is required. Since it is easier to compare equivalent strings when searching, each word is processed into some normalised form. The normalised form is then used for search term comparisons. This requires that the queries entered by the user are processed using the same method before actually beginning the search. (Kettunen 2007)

2.2.2 Retrieval models

There are plenty of ways to search for documents. Different kinds of methods offered by IR systems are called *retrieval models*. While this thesis concerns itself only with retrieval models used by modern IR systems, a brief description of their differences and evolution is offered here.

The *Boolean retrieval model* allows a query to use Boolean expressions for combining search terms with basic Boolean operators (`AND`, `OR` and `NOT`). For fast Boolean comparisons, the index may contain a binary *incidence matrix* where occurrences of words in documents are mapped. When a Boolean query is made, only the documents for which the expression is true are considered. (Manning et al. 2009)

The simple Boolean retrieval model is concise, but it lacks in user friendliness, and it is limited feature-wise. Extended Boolean retrieval models implement further operators, such as the *proximity operator* that allows specifying terms that must occur close to each other. Further extended features include for example allowing for spelling mistakes, and the use of synonyms and phrases in queries. (Manning et al. 2009)

The *vector space model* considers documents and queries a part of t-dimensional vector space, where t is the number of terms in the index. In the model, each document is represented by a vector $D_i = (d_{i1}, d_{i2}, ..., d_{it})$, where d_{ij} is the weight of the term t_j. In the simplest possible form, term weights simply represent the number of occurrences in the document. More advanced weights are discussed in Subsection 2.2.3. Having each document represented as a vector allows the document collection to be represented as a matrix of weights. Each query can also be represented as a vector of weights in a similar way, allowing cross-referencing a query with the document collection matrix, thus calculating which documents match the query well. (Croft et al. 2010)

The *probabilistic model* approaches information retrieval from a probabilistic standpoint. For each query there exists a set of documents that contains exactly the relevant documents. However, the properties of this set are unknown. The probabilistic model initially takes a guess at what these properties might be, and then starts improving the model by interacting with the user, gathering clues on what the user might consider relevant. Ultimately, the model involves calculating the probability of relevance, given a query, as well as the probability of non-relevance. Effectively, the model is a *Bayes classifier* that classifies documents as either relevant or non-relevant, based on those probabilities. (Baeza-Yates and Ribeiro-Neto 2011)

When document collections are large enough, the user cannot be expected to view all search results. Therefore, the documents must be ranked and ordered before presenting them. In *ranked retrieval*, documents are typically ordered by the likelihood of relevance to the user. This involves calculating a score for each document for a given query, as well as possibly some other aspects of the user, such as location and search history, and then sorting the results by that score. The score is supposed to indicate how well the document matches the query. (Manning et al. 2009)

2.2.3 Scoring

As mentioned previously, in order to rank documents, they must be given a score. Score calculations are extremely relevant in the context of this thesis, even though they are not directly studied. Queries give widely different results for each different scoring system. Therefore, the results of any IR simulation are only valid for the scoring system of the target IR system. That is why simulation can be a very efficient way to evaluate different IR systems: one can run the same simulation for multiple different result sets.

To give an explanation of how documents are ranked, one of the best known ranking algorithms, the *Okapi BM25* function (Robertson et al. 1994) serves as a good example. It uses a TF-IDF based approach, where a function based on the sum of an inverse document frequency and term frequency weighted query words is used as the score for ranking.

Term frequency and inverse document frequency are cornerstone weighing measures of scoring. Term frequency is the number of times a term occurs in a document.

Inverse document frequency is the inverse of the number of documents a term occurs in. The former is used for giving weight to how much a document uses a certain word, while the latter is needed for diminishing the weight of common words that may occur in a query. (Baeza-Yates and Ribeiro-Neto 2011)

With these definitions in place, the BM25 function takes the form $\mathrm{BM25}(D,Q) = \sum_{i=1}^{n} \mathrm{IDF}(q_i) \times \frac{\mathrm{TF}(q_i,D)\times(k_1+1)}{\mathrm{TF}(q_i,D)+k_1\times(1-b+b\times\frac{|D|}{|D_{avg}|})}$, where D denotes the document, Q the query, n the number of words in the query, q_i the words of the query (the query is considered to be a "bag of words"), $\mathrm{IDF}(q_i)$ the inverse document frequency for the query word q_i, $\mathrm{TF}(q_i, D)$ the term frequency of the query word q_i in document D, $|D|$ the document length and $|D_{avg}|$ the average document length in the entire collection. Parameters k_1 and b are free, and need to be optimised for the document collection. (Robertson and Zaragoza 2009, p. 360)

The BM25 algorithm is just one example of a ranking algorithm, and there are multiple different ones that perform differently in varying situations. Other algorithms include for example the web-oriented *PageRank*, used by Google, and the machine-learning based *RankNet*, used by Microsoft. (Richardson et al. 2006)

2.3 Evaluating the performance of IR systems and users

There are numerous different information retrieval systems in existence. To quantify their performance, multiple evaluation measures have been devised over time. The aim of an evaluation measure is to assess how well a system meets the information need of a user. Most measures are based on the relevance of the documents, given the topic that represents the information need. A typical measure is also based on rank, therefore being only suitable for assessing ranked retrieval.

The performance of a retrieval system also depends on the behaviour of the user. Different users make different queries, which results in a system giving better results for some users, and worse for some. Some of the performance measures can also be used to assess the effectiveness of user behaviour. Since the theme of this thesis revolves around simulation and user modelling, such performance measures are the ones given greater attention here.

2.3.1 Evaluation using relevance-based metrics

The notion of relevance pertains to the assessment of a document being relevant to a topic that represents the information need of a user. Relevance-based evaluation by laboratory testing has its roots in the *Cranfield experiments*, conducted in the 1960s at the Cranfield University. The aim of the research was to find out which indexing method produced the best results. In what has become the prototype of IR system effectiveness evaluation, the experiments used a test collection of documents accompanied by their topical relevance judgements, as assessed by graduate students, and verified by domain experts. The experiments were conducted using the same set of questions, each representing an information need, or a topic, for each indexing method. The paradigm of having test collections accompanied by relevance judgements made by experts eventually became a part of the de-facto standard process for evaluating IR systems. (Voorhees 2002; Harman 2011).

Since performance comparison requires a quantifiable metric that describes the user's experience, the experiments measured effectiveness using *recall* and *precision*, simple metrics that measure the effectiveness of search with regard to relevance. For an IR system that returns a set of result documents when a query is entered, precision is the fraction of returned results that is relevant to the information need, and recall is the fraction of relevant documents returned by the IR system in the entire document collection. Figure 2.2 illustrates the metrics in an example situation where precision is 3/6 and recall is 3/5. Since the Cranfield experiments, precision and recall have become the most widely used metrics in IR evaluation. (Baeza-Yates and Ribeiro-Neto 2011)

As established by the Cranfield experiments, in that style of IR systems evaluation, the relevance of a document is always first assessed by an expert. Then, the IR system is evaluated using measures based on relevance. For meaningful evaluation data, a test set-up requires a document collection that has been pre-assessed for relevance. To aid researchers in procuring such data, the participants of *TREC* (Text REtrieval Conference) have been producing high-quality and well-available document collections and relevance data since 1992. The data formats and evaluation metrics used by TREC have since become a de-facto standard for IR research data collections and software. (Harman 2011)

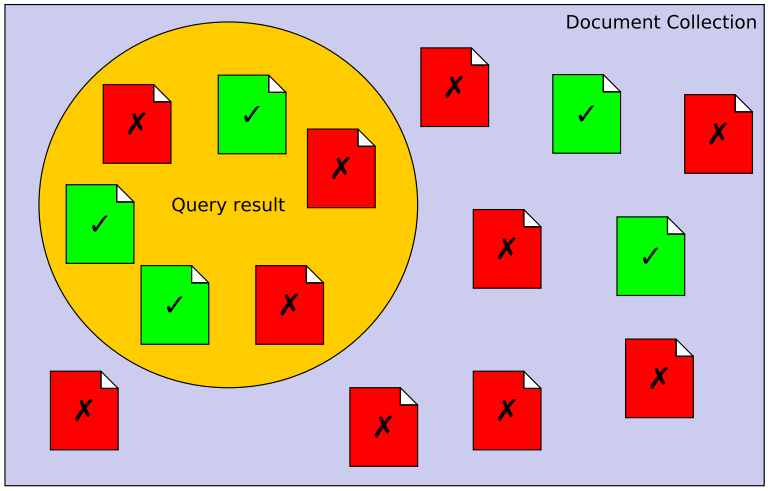

Figure 2.2 *Illustration of a query result within a document collection. The green documents marked with a check mark and the red documents marked with an X mark represent relevant and non-relevant documents, respectively. Here, precision is* 3/6*, and recall is* 3/5*.*

2.3.2 Assessing ranking quality

Relevance-based metrics can be used to assess the effectiveness of an IR system, whether it is a ranked retrieval system or uses some other retrieval model such as Boolean retrieval. However, modern IR systems, such as web search engines, index huge document collections. As such, they can never assume the user is interested in reading all the available documents. Therefore precision and recall, the traditional metrics, are rendered ineffective for meaningful evaluation of effectiveness.

Rank-based evaluation metrics take into account the fact that the documents that are ranked better also should have more weight when evaluating IR systems. Virtually all currently used evaluation metrics are therefore based on rank, in addition to relevance.

2.3.3 Precision-based metrics

The *precision at* n (P@n) metric gives the precision at any rank n, considering all the documents up to rank n as the set of retrieved documents. For example, if the first ten results contain seven relevant documents, P@10 gives a value of 7/10. A typical value for n is 10 (that is, P@10), corresponding to the precision value of the first page of a typical ten-results-per-page system. The greatest shortcoming of

the metric is that it does not take into account the positions of relevant documents (Croft et al. 2010). (Baeza-Yates and Ribeiro-Neto 2011)

To take the order of documents into consideration, one can use the *average precision* metric. In theory, this is the average value of precision $p(r)$ over the interval from recall $r = 0$ to $r = 1$, giving the equation $\int_0^1 p(r)dr$. However, since recall is typically unknown for large document collections, the integral is replaced with a sum over every rank: $\sum_{k=1}^{n} P(k)\Delta r(k)$, where k is the document rank, n is the number of documents to calculate the metric for ("number of retrieved documents"), $P(k)$ is the precision at rank k, and $\Delta r(k)$ is the change in recall from rank $k-1$ to k, that being $1/n$ when the document at rank k is relevant, or zero if not. (Baeza-Yates and Ribeiro-Neto 2011)

Since the average precision metric is only applicable to a single query, the *MAP* (mean average precision) metric is used for summarising the average precision across a collection of queries. The equation is simply given as $\frac{1}{|Q|}\sum_{i=1}^{|Q|} P_{avg}(q_i)$, where $P_{avg}(q_i)$ is the average precision for the query $q_i \in Q$, and $Q = \{q_1, q_2, ..., q_n\}$ is the set of queries. MAP is widely used as a metric for evaluating IR systems. (Baeza-Yates and Ribeiro-Neto 2011)

2.3.4 Cumulated Gain

The concept of *gain* is typically taken to mean improvement over random behaviour. With the *cumulated gain*, or *CG*, metric, however, gain is used to denote the usefulness of a document as a numeric value. All documents in a result set are given a gain value. Relevant documents are typically given a higher gain value than non-relevant documents. The metric can then be used to calculate a cumulated value for any rank p with the equation $\sum_{i=1}^{p} r(i)$, where $r(i)$ denotes the relevance-based gain value of the result at rank i. (Baeza-Yates and Ribeiro-Neto 2011)

Since CG does not take into account the order of results, a derived metric called *discounted cumulated gain*, or *DCG*, can be used to rectify the problem. With DCG, the gain value is reduced logarithmically proportional to the position of the result. The equation is typically given as $r(1) + \sum_{i=2}^{p} \frac{r(i)}{\log_2(i)}$, with the symbols having the same meanings as with the CG equation. A *DCG vector* is a vector of DCG values for a given list of results. (Croft et al. 2010)

A further derivative of the CG metric is the *normalized discounted cumulated gain*, or *nDCG*. The metric has been developed for the case where different kinds of IR systems need to be compared, but CG or DCG values are not directly comparable. To calculate the nDCG value, the *ideal DCG*, that is, the maximum possible DCG value for a result set, must first be calculated by sorting the documents in the collection by relevance and then calculating the DCG for that result set. The nDCG value is then calculated with the equation $\frac{DCG_p}{IDCG_p}$, where DCG_p denotes the DCG at rank p, and $IDCG_p$ denotes the ideal DCG at rank p. (Järvelin and Kekäläinen 2002; Baeza-Yates and Ribeiro-Neto 2011)

2.3.5 Session metrics

A *session* consists of multiple subsequent queries where the user attempts to seek information pertaining to a single topic. The queries are assumed to produce ranked result lists. Typical evaluation metrics, such as MAP, are intended for single-query contexts, and are often unsuitable for assessing the effectiveness of a multi-query session. CG-based metrics have been shown to fit the purpose of session evaluation. A session-based metric called *sDCG* has also been developed based on the DCG metric. The metric simply produces a vector of DCG vectors. (Järvelin et al. 2008)

2.4 User modelling and simulation

According to Johnson and Taatgen (2005), *user modelling* consists of gathering information about users' behaviour and making generalizations and predictions based on the gathered data. The information itself is called a *user model*. A user model can be used to "create an autonomous agent to fill a role within a system". In the case of simulation, the aim is to create exactly that – an autonomous agent that acts on its own. The user model is an integral part of the simulator software, providing the simulator with instructions on how to act.

A valid user model produces valid simulation results, performing in a similar fashion to actual users in any situation within the scope of the model (Law 2008). Therefore, a good simulator software must enable creating valid user models. In this chapter, means to achieve this are explored.

2.4.1 Criteria for a valid user model

A user model's validity cannot be directly assessed by just analysing the model. Acknowledging that, Johnson and Taatgen suggest using the following criteria as qualitative factors for determining the validity of a model:

- as few free parameters as possible,
- the ability to predict behaviour, instead of just describing it, and
- the ability to learn its own task-specific knowledge.

A user modelling simulator software should steer the creation of user models towards meeting these criteria in order to produce valid results. The criteria can therefore be used as guidelines for software requirements specification and software design.

The number of free parameters is well-controlled in a simulation environment, since the user model definition must be formalised. The ability to predict behaviour depends on how well the user model and the simulator represent real life user behaviour. Predictions can be made by simulating the stochastic nature of decision making. Learning task-specific knowledge implies that the model should only describe the task, and not the knowledge needed to complete it. This can be achieved by incorporating support for a learning user model into the software.

2.4.2 Techniques for creating a user model

The simulator software must facilitate creating accurate and various kinds of user models easily. Therefore, utilising widely-used and known techniques as the foundation for designing user model creation facilities is essential. Several techniques for creating a human-computer interaction user model have been developed.

A pioneer in the field is the *GOMS* (Goals, Operators, Methods and Selection rules) technique. As the name implies, GOMS divides interaction into four concepts. *Goals* are the desired end results. Operators are *actions* that are performed to reach the goal. Methods are sequences of operators. Selection *rules* decide which method is used to reach a goal when several are available. In GOMS, each operator can be

associated with a duration. The total duration of completing a task is given by summing the durations of constituent elementary actions. (John and Kieras 1996)

Another approach to cognitive modelling is *EPIC* (Executive Process-Interactive Control), aimed for human-computer interaction design purposes. EPIC is a full-fledged cognitive modelling architecture that takes into account perceptual, cognitive and motor activity (Kieras et al. 1995). The architecture can be used to simulate human behaviour down to the most basic level – for example moving the eye or registering a sound just heard.

EPIC's user model is defined by production rules. A rule contains a set of conditions that are tested against the current internal state (contrast with GOMS' selection rules), and a set of actions that occur when the conditions are satisfied (contrast with GOMS' operators). The system allows for actions to occur in parallel. For example, moving an eye to another element while evaluating the previously seen element is allowed. According to Johnson and Taatgen, parallelism is a very important aspect of user modelling, as it enables creating more accurate models. (Kieras et al. 1995; Kieras 2005)

These techniques can be reduced to the following elements: 1. goals, 2. action modules, 3. actions, 4. rules for determining subsequent actions, and 5. costs of actions (e.g. durations). Incorporating these elements into user model creation facilities should result in a system that allows creating accurate, valid and testable user models. In the case of IR simulation, besides the costs of actions, also *gains* must be considered.

2.4.3 Simulation as a state machine

Information retrieval, as performed by a single user, is a sequential process. Therefore, IR simulation can be described as a discrete-time stochastic process. The simulation forms a sequence of points in time. Each point represents the user completing some action, and each action affects the properties of the next point with some probability.

A Markov chain is a stochastic process that undergoes transitions from one state to another, in the same fashion as state machines. When the simulation considers each discrete point in time as a state, it can be described as a Markov chain. (Geyer 2011)

A state machine can also model parallel user actions by allowing several concurrent active states. This enables parallelism in user models, which can be useful when modelling user actions on motor or cognitive level, where the user may be performing two actions at the same time.

2.4.4 Computing results using Monte Carlo methods

When randomness is introduced into simulation, producing a large number of data samples becomes necessary in order to achieve statistically meaningful results. Thus, a method for computing results using a large set of random data is needed. Monte Carlo methods are intended for that very purpose (Ripley 1987).

In a Monte Carlo simulation, random numbers are generated and used as the input of the simulation. A great variety of Monte Carlo methods exist, many of them designed to work on Markov Chains. For example, the *Metropolis algorithm* works by proposing transitions, trying to move towards a value that best fits in a given distribution, and then either rejecting or accepting them based on the characteristics of the distribution. As the algorithm showcases, some knowledge about the distribution of variables is typically required for successful use of Monte Carlo methods. (Kalos and Whitlock 2008)

The combination of Markov Chains and Monte Carlo methods is often called simply *Markov Chain Monte Carlo*, or MCMC. According to Geyer (2011), most simulations can be considered MCMC if the entire state of the simulation is perceived as the state of the Markov Chain. To make an estimation of a value of a variable, a simple simulation can run multiple Monte Carlo iterations, sampling the variable, and then calculate the empiric mean of the samples.

3. SIMULATOR DESIGN

With the theoretical background established in Chapter 2, the simulator design can be bound to the IR domain. In this chapter, the theoretical framework is considered from software design viewpoint, and software requirements are established. Section 3.1 gives an overview of the context of the software system. Section 3.2 formally describes the simulator as an automaton, while Section 3.3 showcases how the automaton advances through states, and Section 3.4 gives an example of a simulation using the formal model. Section 3.5 then explains how the formal model is applied to software, and Section 3.6 establishes an object model for the concepts present in the model. Section 3.7 describes the input file formats and conventions, while Section 3.8 details what the software outputs. Finally, Section 3.9 defines how end users can configure the software.

3.1 System overview

The Cranfield model of using test collections and relevance judgements, as described in Subsection 2.3.1, was applied to the simulator software. Furthermore, the software was designed to accommodate the TREC data formats (see Section 3.7). Adopting these two cornerstones of IR research allows the software to be built to be useful in a great variety of research cases.

For measuring the effectiveness of simulations, Cumulated Gain was chosen as the main metric. However, the software was designed to be able to calculate any relevance-based measure for the results. Since the CG metric offers support for session-based evaluation, and also offers multiple derivative metrics, its use was deemed to be sufficient for most cases.

For the results calculated by the simulator software, it was decided that a simple approach to utilising Monte Carlo methods would suffice. Therefore, the simple accumulation of samples and the usage of their mean values was adopted as the

method of result approximation. More advanced methods were set aside, determined to be used later if needed for performance optimisation or precision improvement purposes.

Due to being designed for research purposes, the targeted users of the software are IR researchers. Figure 3.1 illustrates how a researcher would interact with the simulator software. First, a researcher needs to gather data from external sources in order to produce the user model and Cranfield-type input data for the software to use. The researcher extracts user model parameters from data gathered in user experiments, and uses the parameters while creating the user model for the software. The researcher also gathers Cranfield-type relevance data from external sources, typically test document collections, in order to feed the data, along with queries and their results, into the software. In response to the input, the software produces effectiveness metrics and other data, further explained in Section 3.8.

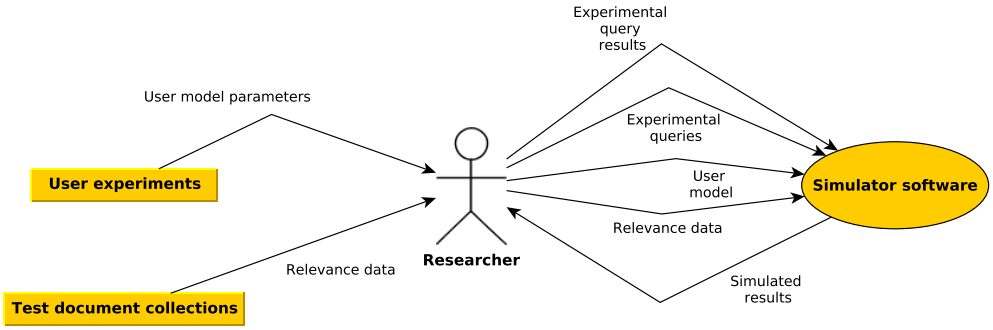

Figure 3.1 *Illustration of the simulator software context. Rectangles represent systems external to the simulator software.*

3.2 Formal definition of the IR automaton

Since the simulator can be described as a state machine, it can be formally defined as an extension of one. A formal definition helps the design and development of the software by establishing a robust framework for building the software foundations on.

For formal definition purposes, the simulator is considered a *finite automaton* with domain-specific modifications. A finite automaton is formally defined as a tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is a finite set of states, Σ a finite set of input symbols, δ a transition function $Q \times \Sigma \to Q$, $q_0 \in Q$ the initial state, and $F \subseteq Q$ a set of final states (Rabin and Scott 1959).

In order to utilise this definition in the context of information retrieval simulation, the definition requires an extension that incorporates a collection of documents and a user model into the automaton. In the extended *IR simulator automaton* (Pääkkönen et al. 2015), the finite automaton tuple is replaced by the tuple $\langle Q, X, \Delta_X, q_0, F, U \rangle$, where $X \in [0, 1]$ is a continuous random variable used as the input of the system, $U = \{u_1, u_2, \ldots, u_n\}$ a set of result document sets, and transition set Δ_X replaces the transition function δ.

When the simulation is run, it forms a temporal sequence of transitions $T = (t_1, t_2, \ldots, t_n)$. Each transition is a tuple $\langle u, r, q \rangle$, where $u = \{r_1, r_2, \ldots, r_n\}, u \in U$ is the result sequence of the last query made by the simulated user, $r \in u$ the result document at hand, and $q \in Q$ the source state.

The set of states Q contains *action* tuples $\langle f_{cost}, f_{gain}, E \rangle$, where $f_{cost} : R \to \mathbb{R}, R = \bigcup U$ is a cost determining partial function, $f_{gain} : R \to \mathbb{R}, R = \bigcup U$ a gain determining partial function, and E a set of events to be triggered. The functions determine how much gain and cost a state can accumulate, based on a document in any of the result sets. When running the simulation, to make changes to the current transition tuple, each set of events E can

- change the result document sequence $u \in U$, and/or
- change the current result document $r \in u$.

Transition set Δ_X contains tuples $\langle q_s, P \rangle$, where $q_s \in Q$ is the source state, and $P = \{\langle q_t, c, V \rangle\}$ a set of transition targets, where $q_t \in Q$ is the target state of the transition, $c \in C$ a condition that must hold in order for the transition to occur, and $V : C \to [0, 1]$ a probability function that determines the probability of transitioning from q_s to q_t, given a condition $c \in C$. Set $C = \{c_1, c_2, \ldots, c_n\}$ is a set of run-time conditions, such as "current document is highly relevant", or "current cumulated cost exceeds 1000", or any combination of such conditions.

3.3 Simulation cycle

The following simulation cycle definition is how Pääkkönen et al. (2015) define a single simulation step in their formal simulator model. The simulation cycle definition is used as the basis of the simulator software.

The simulation changes state in a similar way to a finite automaton, being an extension of one. Using the random variable X as input, the algorithm is defined as follows:

Let q_i be the current state, x_i a random value for X, and T_i the sequence of transitions at this point. Let A_{gain} and A_{cost} be accumulators for gain and cost, respectively.

1. *Trigger the set of events E associated with the current state q_i.*
2. *Let u_i be the current result set, and r_i the current result document, as set by events E.*
3. *Increment A_{gain} by $f_{gain}(r_i)$. Increment A_{cost} by $f_{cost}(r_i)$.*
4. *Stop if q_i is a final state ($q_i \in F$).*
5. *Establish accumulator $A_{prob} = x_i$.*
6. *Iterate over transition set Δ_X where $q_s = q_i$. For each transition, iterate over transition target set P.*
   (a) *Let $\langle q_p, c_p, V_p \rangle$ denote the current transition target $p \in P$.*
   (b) *If $V(c_p) + A_{prob} \geq 1$, choose q_p as target and end iteration.*
   (c) *Otherwise, increment A_{prob} by $V(c_p)$.*
7. *Insert transition element $\langle u_i, r_i, q_i \rangle$ into T.*
8. *Perform transition. Target state q_p becomes current state.*

Gains f_{gain} and costs f_{cost} can also be calculated after the simulation has finished by examining the sequence T and running each function as required.

3.4 An example IR simulation

Consider an IR simulator automaton $\langle Q, X, \Delta_X, q_0, F, U \rangle$, as illustrated in Figure 3.2, where

- $Q = \{q_{scan}, q_{read}, q_{skip}, q_{stop}\}$,

- $\Delta_X = \{\delta_{scan}, \delta_{skip}, \delta_{read}\}$
- $q_0 = q_{scan}$,
- $F = \{q_{stop}\}$, and
- $U = \{u_1\}$, $u_1 = (r_1, r_2)$.

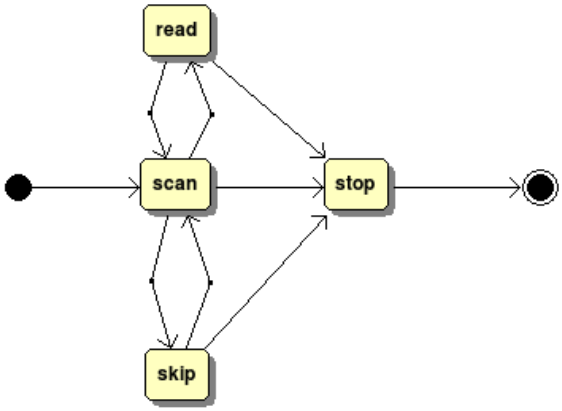

Figure 3.2 *The example IR simulator automaton*

Tuples $\langle f_{cost}, f_{gain}, E\rangle$ in Q are

- $q_{scan} = \langle f_{cost} = 1, f_{gain} = 0, \{\text{“Go to next result”}\}\rangle$,
- $q_{read} = \langle f'_{cost}(r) = \bigcup U \to [1, 10], f'_{gain}(r) = \bigcup U \to [0, 10], \emptyset\rangle$,
- $q_{skip} = \langle f_{cost} = 0, f_{gain} = 0, \emptyset\rangle$, and
- $q_{stop} = \langle f_{cost} = 0, f_{gain} = 0, \emptyset\rangle$.

Gain function f'_{gain} and cost function f'_{cost} are defined as

$$f'_{cost}(r) = \begin{cases} 1, & \text{when } r \text{ is short} \\ 5, & \text{when } r \text{ is long} \\ 10, & \text{when } r \text{ is very long} \end{cases} \tag{3.1}$$

$$f'_{gain}(r) = \begin{cases} 0, & \text{when } r \text{ is not relevant} \\ 5, & \text{when } r \text{ is moderately relevant} \\ 10, & \text{when } r \text{ is highly relevant} \end{cases} \tag{3.2}$$

Tuples $\langle q_s \in Q, P \rangle$ in transition set Δ_X are

- $\delta_{scan} = \langle q_{scan}, \{p_{scan \to read}, p_{scan \to skip}, p_{scan \to stop}\}\rangle$,
- $\delta_{skip} = \langle q_{skip}, \{p_{skip \to scan}, p_{skip \to stop}\}\rangle$, and
- $\delta_{read} = \langle q_{read}, \{p_{read \to scan}, p_{read \to stop}\}\rangle$,

and transition target tuples $\langle q_t \in Q, c \in C, V \rangle$ are

- $p_{scan \to read} = \langle q_{read}, c_{scan \to read}, V_{scan \to read}\rangle$,
- $p_{scan \to skip} = \langle q_{skip}, c_{scan \to skip}, V_{scan \to skip}\rangle$,
- $p_{scan \to stop} = \langle q_{stop}, c_{notavailable}, V_{onlyiftrue}\rangle$,
- $p_{skip \to scan} = \langle q_{scan}, c_{available}, V_{onlyiftrue}\rangle$,
- $p_{skip \to stop} = \langle q_{stop}, c_{notavailable}, V_{onlyiftrue}\rangle$,
- $p_{read \to scan} = \langle q_{scan}, c_{available}, V_{onlyiftrue}\rangle$, and
- $p_{read \to stop} = \langle q_{stop}, c_{notavailable}, V_{onlyiftrue}\rangle$

Conditions in C are

- $c_{available}$ = "Further results documents are available",
- $c_{notavailable} = \neg c_{available}$,
- $c_{scan \to read}$ = "Current result document is moderately or highly relevant", and
- $c_{scan \to skip}$ = "Current result document is not relevant"

Probability functions V are defined as

$$V_{onlyiftrue}(c) = \begin{cases} 1, & \text{when } c \text{ is true} \\ 0, & \text{when } c \text{ is false} \end{cases} \tag{3.3}$$

$$V_{scan \to read}(c) = \begin{cases} 0.85, & \text{when } c \text{ is true} \\ 0.25, & \text{when } c \text{ is false} \end{cases} \tag{3.4}$$

$$V_{scan \to skip}(c) = 1 - V_{scan \to read}(\neg c) \tag{3.5}$$

Assume that all documents are highly relevant and long. The simulation starts from q_{scan} according to the simulation cycle: *Let* $x_i = 0.1$.

1. Trigger the "Go to next result" event. At the start this is the first result r_1.
2. A_{cost} is incremented by 1.
3. This is not a final state $\rightarrow$ continue.
4. $A_{prob} = 0.1$.
5. Iteration of Δ_X finds δ_{scan}
   (a) Iteration of P finds $p_{scan \to read}$.
      i. $V_{scan \to read}(c_{scan \to read}) + A_{prob} = 0.85 + 0.1 = 0.95$. $0.95 < 1 \rightarrow$ continue iteration.
      ii. $A_{prob} = 0.1 + 0.85 = 0.95$
   (b) Iteration of P finds $p_{scan \to skip}$.
      i. $V_{scan \to skip}(c_{scan \to skip}) + A_{prob} = 0.15 + 0.95 = 1.10$. $1.10 \geq 1 \rightarrow$ stop iteration. Choose q_{skip} as target.

At this point, the run-time transition sequence T has one element, containing the tuple $\langle u_1, r_1, q_{scan} \rangle$ of the first transition. The current state is now q_{skip}. Advancing the simulation another cycle using $x_i = 0.4$ yields the following result:

1. Trigger nothing, since $E = \emptyset$.
2. Accumulators are unaffected.
3. This is not a final state $\rightarrow$ continue.
4. $A_{prob} = 0.4$.
5. Iteration of Δ_X finds δ_{skip}

(a) Iteration of P finds $p_{skip \to scan}$.

i. $V_{onlyiftrue}(c_{available}) + A_{prob} = 1 + 0.4 = 1.4$. $1.4 \geq 1 \rightarrow$ stop iteration. Choose q_{scan} as target.

Now $T = (\langle u_1, r_1, q_{scan} \rangle, \langle u_1, r_1, q_{skip} \rangle)$. The current state is again q_{scan}, from where the simulation can advance to q_{read} by "rolling" $x_i > 1 - 0.85$. Let us assume that happens. Now $T = (\langle u_1, r_1, q_{scan} \rangle, \langle u_1, r_1, q_{skip} \rangle, \langle u_1, r_2, q_{scan} \rangle)$ and $A_{cost} = 2$. The simulation has advanced to result document r_2, which is the final document, therefore $c_{notavailable}$ is now true. Now, advancing the simulation using $x_i = 0.05$ yields the following result:

1. Trigger nothing, since $E = \emptyset$.
2. A_{cost} is incremented by $f'_{cost}(r_2) = 5$. A_{gain} is incremented by $f'_{gain}(r_2) = 10$.
3. This is not a final state $\rightarrow$ continue.
4. $A_{prob} = 0.05$.
5. Iteration of Δ_X finds δ_{read}

(a) Iteration of P finds $p_{read \to scan}$.

i. $V_{onlyiftrue}(c_{available}) + A_{prob} = 0 + 0.05 = 0.05$. $0.05 < 1 \rightarrow$ continue iteration.

(b) Iteration of P finds $p_{read \to stop}$.

i. $V_{onlyiftrue}(c_{notavailable}) + A_{prob} = 1 + 0.05 = 1.05$. $1.05 \geq 1 \rightarrow$ stop iteration. Choose q_{stop} as target.

Now $T = (\langle u_1, r_1, q_{scan} \rangle, \langle u_1, r_1, q_{skip} \rangle, \langle u_1, r_2, q_{scan} \rangle, \langle u_1, r_2, q_{read} \rangle)$, $A_{cost} = 7$ and $A_{gain} = 10$. The simulation has now advanced to q_{stop} which is a final state. The last simulation cycle simply tells the simulation to stop since there are no events to trigger or costs or gains to accumulate. The simulation has now ended.

3.5 Applying the formal model to software

Since the formal model describes the simulation as an automaton, applying it to software is straightforward. However, the model only describes the mechanics of

the system in a somewhat abstract level. Therefore, some software design work was required in order to detail the specifics of how all the formally specified features should be implemented.

The formal model specifies a way to define probabilistic state machines that operate based on rules bound to the domain of information retrieval. The software needs to be able to run any such state machines. The first step towards that goal was to define how to run arbitrary probabilistic state machines. It was decided that an object model for such state machine would be defined first, the details of which are explained fully in Section 3.6. As explained in Subsection 2.4.4, in order to reduce ripple in a stochastic simulation, Monte Carlo methods need to be applied. Such a simulation needs to be run multiple times, in the process producing multiple data sets of the calculated metrics. The data sets then need to be averaged in order to produce the final data set of values. The object model was designed to accommodate this requirement.

As explained in Section 3.2, the formal model specifies that a simulation should contain a set of states (Q), a set of transitions (Δ_X), an initial state (q_0), a set of final states (F), and a set of result document sets (U). In other words, the simulator operates using a user model defined by Q, Δ_X, q_0 and F, on a collection of queries and their result documents defined by U. Since it was required that user models and result document sets should be independently changeable, it was decided that the user model would be defined in a *simulation description file* that describes how the simulator should advance, and that the queries and their results would be defined in a *configuration file* that describes what data the simulator should operate on. The language used for defining user models is visited in Section 4.2, and the configuration file format is explained in Section 3.9.

Applying the algorithm defined in Section 3.3 was deemed straightforward, due to it being naturally similar to a procedure or a function. Loosely, the algorithm was designed to be implemented by transforming each step into a function, and then calling them in the correct order. However, the run-time conditions in Δ_X being complex, and the probability functions V using them as their input set necessitated defining the conditions and probabilities in a more structured way to be useful in a software context. This structure was projected into the object model, as well as the simulation description language.

3.6 Object model

Designing a program that runs as a state machine is normally quite straightforward. However, in this case, the users can define their own states and transitions, and the software must be able to run any given arbitrary finite state automaton. Therefore, an abstract model describing an arbitrary state machine had to be devised. Furthermore, the transitions occur according to given probabilities, which means a random choice must be made at each state, instead of advancing deterministically. Complicating the design even more, each probability may be influenced by the current global state of the simulation. To tackle these obstacles, a suitable *object model* of the simulator was designed. Figure 3.3 illustrates the model graphically.

Modelling how a single Monte Carlo iteration holds the IR domain specific data, a base *Simulation Run* class encompasses a single iteration over the state machine from the initial state to one of the final states. A `Simulation Run` object contains *Simulation State* objects. Each `Simulation State` object records a discrete point within an iteration where a state transition has occurred.

A `Simulation Run` holds a reference to an ordered collection of *Query* objects. The `Query` objects contain the search queries that the simulated user is supposedly making. Each `Query` object contains a reference to an ordered collection of *Document* objects. This represents the result set received as a response to the `Query`. The `Simulation Run` also holds a reference to a single `Document` object. This `Document` represents the result document that the simulated user is currently handling. References to the current `Query` and `Document` are recorded to each `Simulation State` object.

Each `Query` object holds a reference to a *Topic* object that represents the information need of the simulated user making the `Query`. A `Topic` object holds a reference to a collection of *Relevance* objects. Each `Relevance` object contains a reference to a `Document-Topic` pair, and records the `Relevance` of the `Document` in the context of the `Topic`.

As illustrated in Figure 3.4, the meta model of the state machine part of the simulator consists of a *State* class and a *Transition* class. They represent the corresponding parts of the state machine. A `Transition` object links two `State` objects together, describing which `State` is the source and which `State` is the target.

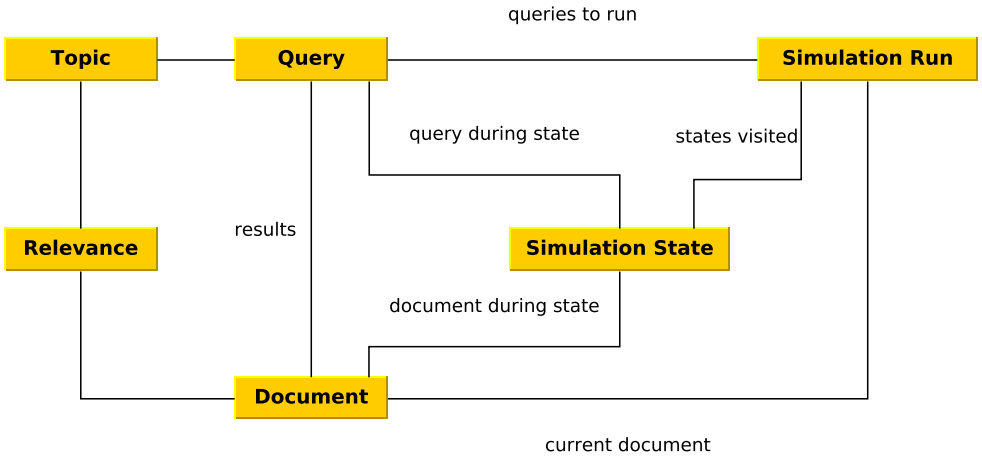

Figure 3.3 *The object model of IR data in a simulation iteration*

To allow for probabilistic transitions, each `Transition` object also refers to a *`Probability`* object. A `Probability` object contains the instructions and data on how to calculate the probability of the transition occurring. The simplest possible `Probability` object is of a *`Direct Probability`* type that only contains a direct probability value between 0 and 1. A *`Conditional Probability`* type references *`Condition`* objects that have the ability to perform checks on the current `State` and the sequence of `Transitions` that have occurred. The `Conditional Probability` will decide the final probability value based on the Boolean value of each referenced `Condition`.

Since `Conditions` need to assess the simulator's history and current status, they need a way to access the necessary object interfaces. This is achieved by defining a callback function in the public interface of the `Condition`. The callback function, analogically to the Visitor design pattern (Gamma et al. 1994, p. 331), takes the entire `Simulation Run` object as an argument, so it can access all the data it needs.

Each `State` object also references *`Gain`* and *`Cost`* objects that determine how much gain is added or subtracted at that `State`, and how much cost is incurred. Like `Conditions`, `Gains` and `Costs` have to define a callback function that has access to the `Simulation Run` object. That way they can calculate their values based on how the `Simulation Run` has advanced. Costs and gains are recorded in the `Simulation Run` object, as well as in the `Simulation State` objects, in order to record their history.

As with `Gains` and `Costs`, each `State` object can reference any number of *`Trigger`* objects. `Triggers` cause changes to the current `Simulation State` within the current `Simulation Run`. They also work by defining a callback that has access to the `Simulation Run` object. Changing the current `Document` and `Query` is done using `Triggers`.

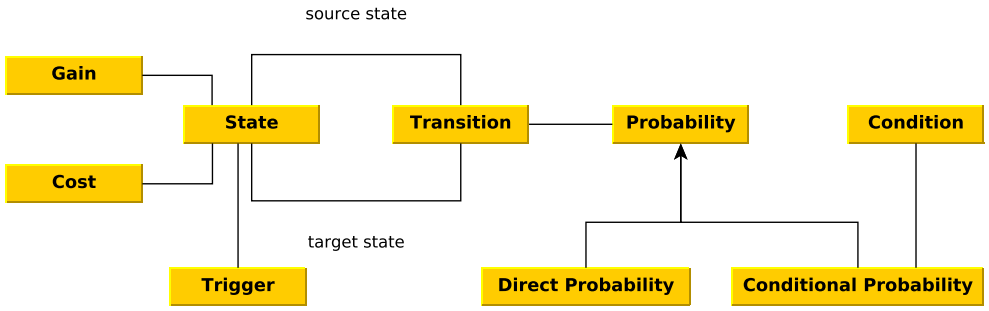

Figure 3.4 *The meta model of the IR state machine*

3.7 Input

As explained, the simulator, as per the formal model, was not designed to generate queries, or make any queries using a search engine on-the-fly. Instead, all the queries and their results are pre-determined, and the simulator concentrates on the analysis of how the user model behaves. Therefore, the software needs to read the queries and their results as its input.

As established in Subsection 2.3.1, file formats used in the TREC work groups have become a widely used de facto standard in IR research and evaluation. Since the software was built for research purposes, adopting the appropriate TREC file formats was deemed the best choice, the alternative being implementing new file formats just for the software. It was also decided that the file reading part of the software should be implemented in such a way that adding new file formats would be easy, in case new formats arise.

To provide all data the simulator needs to run, the input files must contain at least the following items: 1. information needs spelled out as topics, 2. queries to run against the topics, 3. result document lists for the queries, and 4. relevance assessments for each result document given the topics. Mostly, the software only needs to deal with the identifiers of each data item. However, all available data can

be used in the user model, and having more data may result in a more accurate simulation. The information on the topics is only required in order for the simulator to be able to look up the relevance of a result document. The relevance is used to calculate the main metric – cumulated gain – for the result.

3.7.1 TREC topic files

A TREC topic file contains a description of a single piece of information need in a format that resembles *SGML* (Standard Generalized Markup Language). The contents of the topic files have evolved year after year since the beginning of the conference, but the files still always contain at least a *topic number* that acts as a unique identifier for the topic, and a short description of the information need. As illustrated in Program Listing 3.1, further information may include a domain such as "economics", a title, a narrative description that communicates the information need and what is considered relevant more fully, and a list of concepts related to the topic. (Voorhees and Harman 2000)

Program Listing 3.1 *A truncated topic file from TREC-1 (Voorhees and Harman 2000)*

```
<num> Number: 051

<dom> Domain: International economics

<title> Topic: Airbus subsidies

<desc> Description:
Document will discuss government assistance to Airbus Industrie, or mention a
    trade dispute between Airbus and a U.S. aircraft producer over the issue of
    subsidies.

<narr> Narrative:
A relevant document will cite or discuss assistance to Airbus Industrie by the
    French, German, British or Spanish government(s)...

<con> Concept(s):
1. Airbus Industrie
2. European aircraft consortium, Messerschmitt-Boelkow-Blohm GmbH, British
    Aerospace PLC, Aerospatiale, Construcciones Aeronauticas S.A.
3. ...
```

With regard to topics, it was decided that the software should use topic numbers as unique identifiers, and that it should support reading TREC topic files when the user model requires information on the topic. However, such needs are very limited: the data is likely to be only required if the model generates its own queries using the topic description as a basis.

3.7.2 Indri query files

Since TREC does not directly deal with any particular search engine, there is no official TREC query file format. However, the *Indri* search engine, developed as a part of the Lemur project in the University of Massachusetts and first released in 2002, has been previously used in TREC evaluations and IR research in general (Strohman et al. 2005). Due to their association with IR research, the query file format defined by the Lemur project for Indri was deemed to be the best choice for the default format for the simulator software.

An Indri query file is an XML document that contains instructions for the search engine on how to perform a query. As illustrated in Program Listing 3.2, a query file contains one or more query definitions, each of which may contain a query number or identifier, the query text, and a query language type indicator. The default query language is "indri", the other choice being an XPath-based language called "nexi" that was not deemed to be relevant to support in the simulator software.

Program Listing 3.2 *An Indri query file*

```
<parameters>
 <query>
  <type>indri</type>
  <number>34</number>
  <text>
   #combine( metric system )
  </text>
 </query>
</parameters>
```

The Indri query language itself is somewhat complex, offering features for searching both structured and unstructured documents, using constraints that deal with for example proximity or syntax. Since, once again, the simulator software requires only the query identifier to be present, the query text itself was decided to be ignored, but still stored in case the user model needs to analyse it.

The drawback of choosing the Indri query file format is that there is no well-defined way to bind the queries to topics in any way. Often, researchers use a TREC topic identifier as the Indri query identifier, making the distinction between a topic and a query quite fuzzy. Due to this practice, it was decided that for use in the simulator software, the query identifier should always match a topic identifier, making it impossible for one query file to contain multiple queries for the same topic. However, that constraint made it possible to use file names as "real" identifiers for the

queries, which was, after some consideration, adopted as the standard practice with the simulator software.

3.7.3 TREC result files

The TREC result file format is designed to be used for evaluation against TREC relevance data, using the *TRECEVAL* software. As illustrated in Program Listing 3.3, the file consists of space-separated fields that make up a record, one per line, each of which represents a single result in the result list. Each record contains the following fields: query identifier, iteration identifier, document identifier, rank, similarity value, and run identifier. (Eckard and Chappelier 2007)

Program Listing 3.3 *Snippet from a TREC result file*

```
1  445 Q0 LA080390-0117 1 -7.04963 Exp
2  445 Q0 FT921-7364 2 -7.48736 Exp
3  445 Q0 LA080790-0085 3 -7.50261 Exp
4  445 Q0 FT924-8156 4 -7.5216 Exp
5  445 Q0 LA011589-0007 5 -7.54176 Exp
6  445 Q0 LA012490-0018 6 -7.54193 Exp
7  445 Q0 LA022589-0148 7 -7.57658 Exp
8  445 Q0 LA012290-0045 8 -7.65214 Exp
9  445 Q0 FT922-13513 9 -7.71308 Exp
10 445 Q0 LA052590-0060 10 -7.72208 Exp
```

The simulator software can safely ignore most of the fields in the result data: only the query identifier, document identifier and rank are needed for constructing result lists for the queries. Researchers often use the topic identifier in place of the query identifier, in a similar fashion as with the Indri query files. Therefore, it was decided that the practice would be mandated, and as with the query files, only the file name would be used as the query identifier. In theory, this meant that query files would not be needed since the additional information they contain was not required unless the user model needed it. However, since the user modelling part of the software was designed to include the concept of cost, storing the query was deemed necessary, due to the properties of the query having a direct impact on the cost of making the query.

3.7.4 TREC relevance files

As with the TREC result files, the TREC relevance files are used for evaluation using the TRECEVAL software. As illustrated in Program Listing 3.4, the file

also consists of one-line records with space-separated fields. Each record represents a "correct answer" to a given query, recording the relevance of a single document, and contains the following fields: query identifier, document identifier, and relevance degree number. The relevance degree number can be binary – the document is either relevant or not – or it can be graded, as in Program Listing 3.4, where numbers from 1 to 3 are used. Often, non-relevant documents are omitted entirely. In such cases, any result document not present in the relevance file must be considered as non-relevant. (Eckard and Chappelier 2007)

Program Listing 3.4 *Snippet from a TREC relevance file*

```
1  442 LA112290-0058 2
2  442 LA113089-0076 3
3  442 LA120990-0074 3
4  442 LA121089-0170 1
5  442 LA122890-0062 3
6  445 FT921-11838 1
7  445 FT921-11847 1
8  445 FT921-11857 1
9  445 FT921-4820 1
10 445 FT921-7364 2
```

As is the typical case with TREC files, the confusion with the separation of topics and queries continues in the relevance files. In a typical case, the query identifier actually denotes the topic, matching with its identifier. Since this is the typical practice, it was also mandated for the simulator software, in the process making it actually easier to match a document relevance with a topic.

3.7.5 Sessions

With the topic, query, result and relevance files having been assessed, the basis of the input data for the simulator software was in place. However, one further problem to solve was how to build sessions using those files. The TREC file formats are not designed for session-based evaluation, instead working entirely on a per-query basis. Therefore, a way to construct sessions from those files was needed.

Since a session consists of queries, the natural solution was to build the sessions by simply having multiple query files, and denoting the order of the queries in some way. Two approaches were considered: 1. have an explicitly defined ordered list of query and result files, or 2. use file names to denote the order. After some consideration, it was decided that since the researcher is likely to name the files so that they are

naturally ordered in any case, the second option would be the best choice, inducing the least amount of work for the researcher.

Having file names denote the order of queries posed two problems: 1. the result files must somehow be bound to the queries using file names, and 2. the directory structure must be such that the query and result files cannot be confused with any other files. The obvious solution to the first problem was to formulate a naming scheme for the files. Unfortunately, such approach is somewhat error-prone, and detecting any errors in file names can be very difficult unless the naming scheme is very strict. Nevertheless, this trade-off was considered to be a manageable one, and the following naming scheme was implemented: 1. the query files would have a name that follows the pattern ⟨`sessionID`⟩`_q`⟨`queryOrderNum`⟩, for example `sessionX_q1`, and 2. the result file corresponding to a query would have the same file name as the query, but with an `r` appended at the end, for example `sessionX_q1r`. The second problem with the directory structure was solved by simply mandating that the query and result files must reside in a directory with no other files present.

3.8 Output

Since the simulator was designed to be used for research, it was considered imperative that the software, when required, would output as much data as possible since every bit of it may be important, especially when making statistical analyses that are very important in high-quality research. Therefore, it was decided that the software would support both software and human-readable output formats, as well as configurable levels of output.

To select a good output format for data analysis, the methods used for making the analysis were first assessed. It was discovered that in the targeted user group, analysis is often done in either spreadsheet software or by using the *SciPy* software (Chauve et al. 2016) with the Python programming language. An output format usable for both cases was therefore desired.

For SciPy, the obvious choice would have been to simply output everything as Python objects. However, that was not a good option for spreadsheet software. After some consideration, it was decided that the output would be given in the *comma-separated values* (CSV) format, which is supported by both spreadsheet software and Python, and is even simple enough to be human-readable. The choice of CSV limits what

kind of data can be represented, but that was not deemed to be a problem since only simple spreadsheet-like data structures were perceived to be required to be present in the output.

To keep the software relatively simple, it was decided that the simulator would mostly output the raw numbers generated during the simulation runs. Further processing would then be carried out in other software. However, since it was also desirable to have results that would be quick to compare with existing data, it was specified that the simulator should calculate and output the averages over all Monte Carlo iterations. A truncated version of typical output is shown in Program Listing 3.5. Since cumulated gain was specified to be the main metric, it is used in all output.

Program Listing 3.5 *Example of program output for a run with 100 iterations*

```
1  cost,amt runs,avg gain,max gain,min gain
2  0,100,0.0,0.0,0.0
3  10,100,0.0,0.0,0.0
4  20,100,0.0,0.0,0.0
5  30,100,0.0,0.0,0.0
6  40,100,0.0,0.0,0.0
7  50,100,0.0,0.0,0.0
8  60,100,0.4,5.0,0.0
9  70,100,0.4,5.0,0.0
10 80,100,1.5,10.0,0.0
11 90,100,1.5,10.0,0.0
12 100,100,3.15,10.0,0.0
```

Traditionally, IR metrics are recorded per document rank. However, since for each rank there can be multiple simulated actions, it was deemed necessary to also record the metrics at finely-grained cost points. The notion of cost typically refers to time, but it is up to the researcher to define what the incurred cost actually means. For that reason, in the output, the granularity of cost points was decided to be user-definable in the configuration file (Section 3.9).

In order to make coarse comparisons of the results faster, it was decided that graphical plots of the data would also be produced. The plots were specified to show the average cumulated gain over cost and rank, along with the standard deviation and top and bottom percentiles for making observations that are not directly visible in the average plot. An example of such a graphical plot over cost is shown in Figure 3.5.

The full output of the simulator was specified to record each Simulation State, explained in Section 3.6, as they occur. A Simulation State contains information

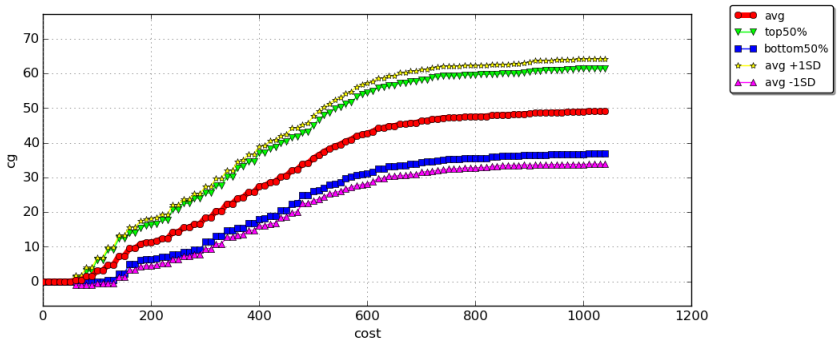

Figure 3.5 *Example of single-session output plot*

on what action was previously made, what query had last been made, the current document rank, current cumulated gain and cost, the last document seen, as well as information on the iteration number and so forth. An example of such data is shown in listing 3.6. This kind of data is typically not used in research, instead being more useful for user model debugging purposes in cases where errors are not readily detectable by software.

Program Listing 3.6 *Example of full program output*

```
1 sessId,iter,prevAction,queryIdx,totRank,currQueryRank,cumulGain,cumulCost,docId
2 1,0,,-1,0,0,0,0,
3 1,0,start,0,0,0,0,9.17,
4 1,0,issue_query,0,0,0,0,9.17,
5 1,0,scan_snippet,0,1,1,0,15.47,NYT19980914.0168
6 1,0,view_document,0,1,1,0,35.07,NYT19980914.0168
7 1,0,mark_as_relevant,0,1,1,0.0,35.07,NYT19980914.0168
8 1,0,scan_snippet,0,2,2,0.0,41.37,NYT20000718.0206
9 1,0,view_document,0,2,2,0.0,60.97,NYT20000718.0206
```

To ease the comparison between different user models, a further output mode was devised. In cross-session output mode, the software calculates mean cumulated gains over the average gains of multiple sessions. This way, the researcher can at a glance note the differences between multiple user models by comparing two graphical plots or raw data sets. The output format is nearly the same as in the normal case, as shown in Program Listing 3.5 and Figure 3.5, the difference being that the cross-session format outputs the average data sets for every session, along with the average-over-all-sessions data set, as illustrated in Figure 3.6.

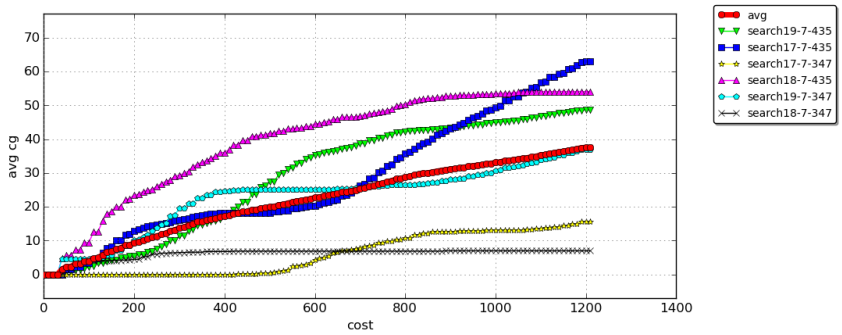

Figure 3.6 *Example of cross-session output plot*

3.9 Configuration

In order to make it possible to instruct the simulator what input it should operate on, a means had to be developed for declaring the locations of input files along with their formats, and also other parameters such as how much gain a certain relevance level should command and how to handle missing relevance assessments. Furthermore, as with input, a way to convey what kind of output is desired was also required.

An often used approach to configuring how a piece of software works is through the use of command line parameters. This approach was also considered for the simulator software. However, as the software was designed to be used for research purposes, it was deemed to be desirable to be able to store different configurations easily, and re-visit them if needed. Therefore, a command-line-only approach was scrapped in favour of using a configuration file to configure the software.

Since there were no plans to introduce a graphical user interface for creating configuration files, the file format had to be chosen so that it was both human-readable and machine-parsable. To satisfy those requirements, and for wide software support, as further discussed in Subsection 4.2.1, an XML-based file format was decided on.

The configuration file is divided into three sections: input, output, and options. As shown in Program Listing 3.7, the input section defines where the input files are located. There can be multiple relevance files, as well as multiple query and result files. Sessions are defined by indicating a directory where the query and result files can be found. The section also instructs the software to the location of the simulation description file, as well as the locations of optional callback files. Callbacks are further explained in Section 4.10.

Program Listing 3.7 *Snippet from the input section of a configuration file*

```
<files>
  <input-directory>iiix-sim2-gains-0-5-10</input-directory>
  <relevance-file format="trec">search10-6-347_qrels</relevance-file>
  <relevance-file format="trec">search10-6-435_qrels</relevance-file>
  <simulation>simulation-condition6-group4.xml</simulation>
  <condition-callbacks>customConditionCallbacks.py</condition-callbacks>
  <cost-callbacks>customCostCallbacks.py</cost-callbacks>
</files>

<sessions>
  <sessions-directory>sessions-condition6-group4</sessions-directory>
</sessions>
```

The output section defines the output format, a choice between CSV and Python. It also configures where to place the output files. Furthermore, as shown in Program Listing 3.8, it is also possible to define further metrics derived from gain to be calculated in the output. If a derived gain is defined, a Python function for making the calculation must be present, and the input section must contain a *derived-gains-callbacks* element that defines the file where to find the referenced calculation function.

Program Listing 3.8 *Derived gain definition in a configuration file*

```
<gain-types>
  <type id="dcg" function="calc_dcg">
    <argument name="base" value="2" />
  </type>
</gain-types>
```

The options section is used for defining further configuration. For example, the random seed used for initialising the random number generator can be defined here, so that the simulation can be repeated with the exact same results every time. As shown in Program Listing 3.9, the options section also contains instructions on how each relevance level affects the cumulated gain.

Program Listing 3.9 *Gains with their respective relevance levels in a configuration file*

```
<gains>
  <gain relevance-level="0" gain="0" />
  <gain relevance-level="1" gain="5" />
  <gain relevance-level="2" gain="10" />
</gains>
```

Since it was deemed desirable to be able to run the same simulation using multiple configurations, it was made a requirement that the software should take the configuration file name as a parameter. This made it possible to vary for example the gain levels or the sessions by producing multiple different configuration files and running the simulation once with each file.

4. SIMULATOR IMPLEMENTATION

Since the simulator is formally defined as an automaton, the main question was how to implement a system that allows the user to create and run arbitrary state machines. Furthermore, the software was required to support state transitions based on probabilities, in the fashion of Markov Chains.

Another implementation problem was how to bind the state machine to the IR domain. Arising from the formal definition, a requirement was that the simulator should be aware of search queries, result lists, result documents and their relevances, as well as multiple other aspects of information retrieval – for instance: how to calculate the gains and costs incurred by the user.

Since the software was written for research purposes, and mainly for academic professionals, some requirements were imposed on the run environment and maintainability of the software. This had an effect on the choice of implementation technologies.

This chapter discusses the specifics of implementing the software. First, Section 4.1 explicates the considered and selected technology choices. In Section 4.2 the development of a language for writing IR user models is explained in detail. Sections 4.3–4.5 detail general software development related aspects of the selected technologies. Section 4.6 explains the architecture of the software in general terms. Sections 4.7–4.9 describes how the input files are parsed, while Section 4.10 details the callback plug-in architecture. Sections 4.11–4.12 analyse how the simulator proceeds to run the actual simulations, and Sections 4.13–4.14 explain how the output is generated. Finally, Section 4.15 specifies how the software is distributed.

4.1 Technology considerations

While the implementation was free of the burden of pre-existing software needing support, the choice of technology was not unrestricted. The software was required

to be suitable for use in academic settings, by academic professionals. This boiled down to multiple requirements.

The software had to be implemented in such a way that it could be run on any computer available to researchers, be it their personal computer or a computing cluster where only a head-less terminal connection was available. This also meant that implementation technologies had to be chosen so that operating system support was wide enough: at least Windows XP and later, OS X 10.7 and later, and mainstream Linux distributions from 2012 onwards had to be supported.

The programming language for the software also had other requirements stemming from the academic environment and the project scope. It was acknowledged that support for the software would not continue indefinitely by the original author. Therefore, the researchers have to be able to make future changes and fixes to the software themselves. This requirement called for a language that was easy to approach and widely used in the academic setting.

A further point for consideration was the performance and scalability of the chosen programming language. The simulator was determined to need to run hundreds or thousands of iterations for each simulation for stable results. Therefore, memory consumption could be a very real issue, and a very large run could take hours or even days of time to finish if the performance of the software was bad enough. In the end, performance was considered to be more an issue of the software architecture, and not the chosen technology, per se. Therefore, performance was not taken into consideration in the selection process.

These requirements in mind, the following languages were considered: Python, R, Java and C++. The requirement of easy approachability led to C++ being rejected outright due to the number of quirks and pitfalls inherent in the language being perceived larger than the alternatives. Selecting C++ would also have meant a more difficult approach to cross-OS support. Java, on the other hand, while more approachable, was not as well known in the targeted research group, and was therefore rejected as well. R, a relatively unknown language outside of research environments, was also considered because of its merits in statistical computing. It was eventually rejected because it was deemed somewhat more obscure than Python, the eventual choice for the programming language.

Python, the final choice as the programming language, is a *dynamically-typed*, *interpreted* language. The language is designed to be very readable, and it is therefore well-suited to projects where the maintainer may change multiple times in the course of the software's lifespan, and is not necessarily a professional software developer. It is officially supported on multiple major operating systems, and a wide variety of third party libraries and frameworks is available for it. Further helping the case of easy maintainability, Python was ranked as the fourth most used programming language in the RedMonk Programming Language Rankings of June 2015 (O'Grady 2015). (Python Software Foundation 2016b)

4.2 Simulation description language

To allow the users of the simulator to write user models, an XML-based language was devised. Basically, the language allows users to configure the simulator automaton by defining the states and transitions. The language also offers ways to bind the automaton to the IR domain by defining gains, costs and triggers that change the query or the current document.

The language applies the theoretical guidelines laid down in Section 2.4 to the IR simulator software. The criteria for a valid user model control what features are present in the language, and the GOMS and EPIC techniques work as guidelines for how to structure those features.

4.2.1 Technology selection

The target audience of the software is academic professionals, who possess advanced skills in computer use, but not necessarily any programming experience. Therefore, the simulator software package required a simple-to-grasp but powerful mechanism for defining and fine-tuning the behaviour of the simulated user. It was concluded that the mechanism should be based on a user-written file that was easy to read, and also easily parsable by software. Due to the complexity of the software, and the possibly lesser skills of users, the file was also required to be validatable for syntax and semantic errors. For easier maintainability, a pre-existing language was decided to be the best option.

Since the choice of programming language was already made, options were reduced to languages that were parsable by the Python Standard Library, or an external

Python library. The standard library contains a "configuration file parser" that parses files that are formatted in the fashion of Windows INI files (Beazley and Jones 2013, p. 552). While the INI format is quite expressive, it is also schema-less, which means the configuration files would be hard to validate. Another file format with direct support, the *Javascript Object Notation* format (Beazley and Jones 2013, p. 179), while having a more strict formal definition than the INI format, suffers from the same validatability problem. Such approaches were abandoned.

The standard library also supports the XML (*Extensible Markup Language*) format (Beazley and Jones 2013, p. 183). XML is a structured language for arbitrary data. It offers facilities for defining the structure and contents of documents using a *schema language*, such as the *XML Schema*, or *RELAX NG*. An XML document can be validated against a schema definition. However, the Python Standard Library does not support reading such schemas or validating documents using them. Fortunately, an external Python library called *PyXB* (Python XML Schema Bindings) (Bigot 2014) offers this functionality. After some consideration, the XML format and the PyXB library were chosen for the project.

4.2.2 Development

The language was developed by writing an XML Schema for the format. The base aim was to produce a format that matches the object model of the simulator. This was approached by creating mappings between object classes and XML element definitions, such as the mapping from `State` class to *`Action`* element schema presented in Program Listing 4.1.

Program Listing 4.1 defines an `Action` element that corresponds to a `State` object. It may contain *`Trigger`* elements that correspond to `Trigger` object references. The `Trigger` elements may contain additional arguments given to the `Trigger` object's callback function. The `Action` element may also contain references to *`Gain`* and *`Cost`* elements that correspond to their namesake object references.

Referencing other elements, such as the `Gain` and `Cost` elements in Program Listing 4.1, is done by defining a key for the element being referenced, and then defining a key reference that tells XML parsers that an XML element is a reference to a defined key. Program Listing 4.2 presents a definition that defines the `id` attribute of `Gain` elements as a key. Program Listing 4.3 presents a key reference definition

Program Listing 4.1 *XML Schema for Action element*

```
<element name="action" minOccurs="1" maxOccurs="unbounded" form="qualified" >
  <complexType>
   <sequence>
    <element name="trigger" minOccurs="0" maxOccurs="unbounded" form="qualified"
    >
     <complexType>
      <sequence>
       <element name="argument" minOccurs="0" maxOccurs="unbounded" form="
    qualified">
        <complexType>
         <attribute name="name" type="string" use="required" />
         <attribute name="value" type="string" use="required" />
        </complexType>
       </element>
      </sequence>
      <attribute name="type" type="string" use="required" />
     </complexType>
    </element>
   </sequence>
   <attribute name="id" type="string" use="required" />
   <attribute name="cost" type="string" use="optional" />
   <attribute name="gain" type="string" use="optional" />
   <attribute name="final" type="boolean" use="optional" default="false" />
  </complexType>
 </element>
```

that defines the `gain` attribute of `Action` elements as being a reference to the keys defined in Program Listing 4.2. Defining the references this way allows XML parsers to validate them.

Program Listing 4.2 *XML Schema for Gain element keys*

```
<key name="gain-id">
 <selector xpath="qsdl:gains/qsdl:gain" />
 <field xpath="@id" />
</key>
```

Program Listing 4.3 *XML Schema for Gain element references*

```
<keyref name="gain-reference" refer="qsdl:gain-id">
 <selector xpath="qsdl:actions/qsdl:action" />
 <field xpath="@gain" />
</keyref>
```

4.2.3 Features

The simulation description language was designed such that a single description file contains a single user model. The model can be parametrized so that the parameter values are bound at run time using the configuration file, thus allowing the same user model to be used with multiple parameter sets, enabling easy experimentation.

A simulator description defines the finite state machine that makes up the user model, as explained in Section 3.2. The state machine part itself consists of states and transitions. In the simulator, `States` correspond to `Actions` that can incur gains and costs, and trigger changes in the global simulation state. An example of a set of `Action` definitions is found in Program Listing 4.4. A definition consists of an `Action` element whose optional attributes define the `Costs` and `Gains`, and optional child elements that define what global triggers to fire.

Program Listing 4.4 *A partial set of action definitions*

```
<actions initial="start">
  <action id="start" cost="formulate_query">
    <trigger type="jumpToQuery">
     <argument name="qidx" value="0" />
    </trigger>
  </action>
  <action id="view_document" cost="view_document">
    <trigger type="flagAsSeen" />
  </action>
  <action id="mark_as_relevant" gain="mark_as_relevant" />
  <action id="stop_session" final="true" />
</actions>
```

The `Transition` definitions describe what transitions from state to state are possible and when. Program Listing 4.5 shows a such a set of `Transition` definitions. A `Transition` must always contain a `Probability` reference, since all transitions in the simulator are probabilistic. In the program listing, some of the `Probabilities` are marked with *`always`* and *`remaining`*, the former of which is a built-in probability reference with a value of one, and the latter a reference with a value that is calculated as the "remaining" probability after the other transition targets' probabilities have been summed up.

`Probability` definitions step outside from the world of state machines into the domain of stochastic simulation. Each `Probability` definition contains either a direct probability value between zero and one, or a set of `Conditions` and their corresponding probability values. A partial set of `Probability` definitions is showcased in Program Listing 4.6. A probability value can also be marked with an asterisk, which means that the probability is calculated the same way as the transition probabilities marked as "remaining".

Program Listing 4.5 *A partial set of transition definitions*

```
<transitions>
  <from source="start">
    <to target="scan_snippet" probability="always" />
  </from>
  <from source="scan_snippet">
   <to target="view_document" probability="view_document" />
   <to target="stop_session" probability="stop_session" />
   <to target="scan_snippet" probability="keep_scanning" />
  </from>
  <from source="view_document">
    <to target="mark_as_relevant" probability="mark_as_relevant" />
    <to target="scan_snippet" probability="remaining" />
  </from>
  <from source="mark_as_relevant">
    <to target="stop_session" probability="stop_session" />
    <to target="scan_snippet" probability="remaining" />
  </from>
</transitions>
```

Program Listing 4.6 *A partial set of probability definitions*

```
<probabilities>
  <probability id="keep_scanning">
    <if condition="cost_exceeded" value="0" />
    <else-if condition="no_more_results_for_query" value="0" />
    <else value="*" />
  </probability>
  <probability id="stop_session">
    <if condition="cost_exceeded" value="1" />
    <else-if condition="should_change_query_but_none_available" value="1" />
    <else value="0" />
  </probability>
  <probability id="view_document">
    <if condition="cost_exceeded" value="0" />
    <else-if condition="document_is_not_relevant" value="0.284" />
    <else-if condition="document_relevance_equal_to_1" value="0.491363" />
    <else-if condition="document_relevance_equal_to_2" value="0.527680" />
    <else value="*" />
  </probability>
  <probability id="mark_as_relevant">
    <if condition="document_is_not_relevant" value="0.528443" />
    <else-if condition="document_relevance_equal_to_1" value="0.628906" />
    <else-if condition="document_relevance_equal_to_2" value="0.792411" />
    <else value="*" />
  </probability>
</probabilities>
```

`Condition` definitions describe what callback functions to call to resolve a conditional probability. The callback system is described more fully in Section 4.10. Program Listing 4.7 contains an example of a partial set of `Condition` definitions. Each `Condition` definition refers to a callback function name and may contain arguments for the function.

Program Listing 4.7 *A partial set of condition definitions*

```
<probability-conditions>
  <probability-condition id="document_is_not_relevant">
   <callback name="current_document_relevance_between">
    <argument name="min_inclusive" value="0" />
    <argument name="max_exclusive" value="1" />
   </callback>
  </probability-condition>
  <probability-condition id="cost_exceeded">
    <callback name="default_cost_exceeded">
      <argument name="cost_limit" value="1200" />
    </callback>
  </probability-condition>
  <probability-condition id="no_more_results_for_query">
    <callback name="default_current_document_is_ranked_last" />
  </probability-condition>
</probability-conditions>
```

The final part of a simulation description are the definitions of `Costs` and `Gains`. Program Listing 4.8 contains an example of how to define the `Gains` and `Costs` referenced in `Action` definitions. As with `Condition` definitions, `Gains` and `Costs` can be calculated at run time using callbacks, or they can be given constant values directly.

Program Listing 4.8 *A partial set of cost and gain definitions*

```
<costs>
  <cost id="formulate_query">
   <value>7.97</value>
  </cost>
  <cost id="view_document">
   <value>17.71</value>
  </cost>
</costs>

<gains>
  <gain id="mark_as_relevant">
   <callback name="get_current_document_gain" />
  </gain>
</gains>
```

4.3 Using third-party libraries in Python

As established earlier, Python is a very widely used programming language, and there is a large selection of software libraries and frameworks that can aid in developing software. Furthermore, the *Python Software Foundation* maintains a third-party software repository called *PyPI* (Python Package Index) that contains many of the major software packages available for Python. The repository hosts downloads for all the software packages, and also records documentation, categorisation

and further meta-data for each one of them. This approach helps developers to more easily find the third-party software they need, and to keep it up-to-date when new versions are released. (Python Software Foundation 2016a)

Since installing and updating software packages manually can be tedious, often consisting of steps such as downloading the packages, extracting them, compiling the sources and handling dependencies, the *Python Packaging Authority* maintains a software package installation tool called *pip*. With pip, developers can, with simple CLI commands, install packages from PyPI and other sources, list outdated packages, upgrade them and remove them, among other things. It is also possible to create requirement and constraint files that list all the software packages a project requires. All this makes it far easier to transfer a software project from one system to another, since the target system only needs to have pip installed, and all requirements can be installed with a simple command. (Danjou 2014)

4.4 Writing programming interfaces in Python

In Python there are no strictly defined object interfaces. Instead, the language uses *duck typing*, where any object that defines certain methods with certain names and parameters that define some behaviour, is considered to implement the interface for that behaviour without explicitly declaring so. The name duck typing stems from the *duck test*, a form of reasoning that states: "If it looks like a duck and quacks like a duck, it is a duck."

With duck typing, it generally makes little sense to try to define strict interfaces, since the language is not built to support them. Instead, it makes more sense to define behaviours with constructs such as *mix-ins*, *concerns* and *composition*. The subject is visited more thoroughly in Section 4.5.

The simulator software contains some parts where similarly behaving components can be observed. To give an example, one instance of similarly behaving components are the *result readers* that provide support for different kinds of result files. Result files contain ordered lists of documents that have been produced by a search engine as a response to a query or multiple queries. The result file may or may not contain the query text itself. The task of a result reader is to scan the result file and produce ordered containers of Document objects, that represent the result lists.

Since multiple formats had to be supported from the start, and since it is unknown how many different formats need to be supported in the future, a generic interface for the result readers was designed. On the file reading side of things, each reader class must implement a `can_parse` method that takes a file name as argument and returns a boolean value that tells if the file is parsable by the class. Each reader class must also implement a constructor that takes a file name as argument. The rest of the methods are for the simulation to use at run-time: `get_results_by_id` that returns a list of Documents for a given Query ID, `get_document_id` that returns a Document ID given a Query ID a rank number, and `get_results_length` that returns the number of documents in the result list for a given Query ID.

Since Python does not enforce that a class implements an interface, one needs to be careful when creating new classes that should implement an interface. Typically, a separate interface definition would, in addition to enforcing the implementation of methods, serve as documentation for future implementers of the interface. To achieve the documentation aspect, generic interfaces were defined as a separate class that contains a class variable that holds a list object with the names of the methods that should be implemented. The constructor of the class defines all the listed methods at run-time, and has them simply raise an error. This provides both security and documentation for the interface. An example of such a construct for the result reader interface is given in Program Listing 4.9.

Program Listing 4.9 *Result reader interface class*

```
1  class Interface:
2    @classmethod
3    def _add_error_raising_method(cls, method_name):
4      def r(*args):
5        raise MethodNotImplementedError( method_name )
6      r.__name__ = method_name
7      if not hasattr( cls, method_name ):
8        setattr( cls, method_name, r )
9
10   def __init__(self):
11     for method_name in self.__class__.public_methods:
12       self.__class__._add_error_raising_method( method_name )
13
14 class ResultReader(Interface):
15   public_methods = ['get_results_by_id', 'get_document_id', 'get_results_length'
         , 'can_parse']
```

Similar approach to interfaces was taken with query readers and relevance readers, the former being used for reading information on queries from different kinds of query files, and the latter being used for reading topic-based relevance assessments

from different kinds of relevance files. The approach was also applied to callback plug-ins, which are discussed in Section 4.10.

4.5 Re-usable code

Typical means of code re-use in object-oriented languages include inheritance, composition and the use of design patterns such as mix-ins (Gamma et al. 1994, p. 2). Code re-use can also be achieved by adhering to design principles such as *separation of concerns* and *don't repeat yourself* (Martin et al. 2009).

For a new piece of code to be able to use some existing code, there must exist an interface that allows the pieces to communicate. In Python, there are no separately defined interfaces. The bit of code needing to use a re-usable piece of code must trust that the interface is implemented. When required, the interface using code may, instead of checking for the existence of the interface, catch any errors that stem from the interface not being implemented. This is called the *EAFP* principle, or: "It is Easier to Ask for Forgiveness than Permission." This principle was followed throughout the process of writing the software, where applicable. (Beazley and Jones 2013; Vaingast 2014)

As mentioned in Section 4.4, a form of interface definition was used in this project, but mainly for documentation purposes. The interface-documenting classes are, in a way, a type of a mix-in class. Mix-in classes are a way to define and implement a behaviour separately from other classes, and to introduce that behaviour to any class where it is needed (Gamma et al. 1994, p. 16). Python does not support mix-ins per se, but it does support multiple inheritance, which can be used for *including* ("mixing in") mix-in classes. As with typical multiple inheritance, care must be taken not to introduce ambiguity problems caused for example by class members having the same names, or the *diamond problem* by having multiple mix-ins inherit a common ancestor, all of them overriding the same method, and then having a class include more than one such mix-in.

Mix-ins were used in this project for re-using some common behaviour in file reader classes, callback plug-ins and output classes. To give an example, the file reader classes use a mix-in called `OperatesOnFile` that contains helpers for the general operations needed when reading IR related files. As showcased in Program Listing 4.10, the mix-in class contains two methods, `get_bare_file_name` and `each_record`, the

former of which returns the file name of the file being operated on without any preceding path, and the latter of which acts as a *generator* method that produces records from the file. The each_record method expects including classes to implement a `get_record_iterator` method, as is evident from the use of the Interface class in the program listing. The method is expected to return an iterator that produces singular records from a file, given a file handle. In a typical case, the iterator simply produces all the lines of the file.

Program Listing 4.10 *OperatesOnFile mix-in class*

```
1  class OperatesOnFile(Interface):
2
3    public_methods = [ '_get_record_iterator' ]
4
5    def __get_bare_file_name(self):
6      head, tail = ntpath.split( self.file_name )
7      return tail or ntpath.basename( head )
8
9    def __each_record(self):
10     with open( self.file_name ) as file_handle:
11       record_iterator = self._get_record_iterator( file_handle )
12       for record in record_iterator:
13         yield record
```

Further code re-use in the project comprises the use of design patterns (Gamma et al. 1994), and inheritance for some of the classes.

4.6 Overall architecture

The overall architecture of the software is best described through the three *stages* of the program flow. First, there is the input parsing stage, where the simulation description, configuration, and IR data files are read and turned into object representations. Second, the simulation stage executes several iterations of the simulation, and gathers data on each one. In the third and final stage, the gathered data is aggregated into statistics and graphical representations, and output to files.

The architecture of the input stage is largely explained in Section 4.7, with further insight in Section 4.8 and Section 4.9. The second stage is visited in Section 4.11, and the output stage is analysed in Sections 4.13 and 4.14. Figure 4.1 illustrates how the different components work together.

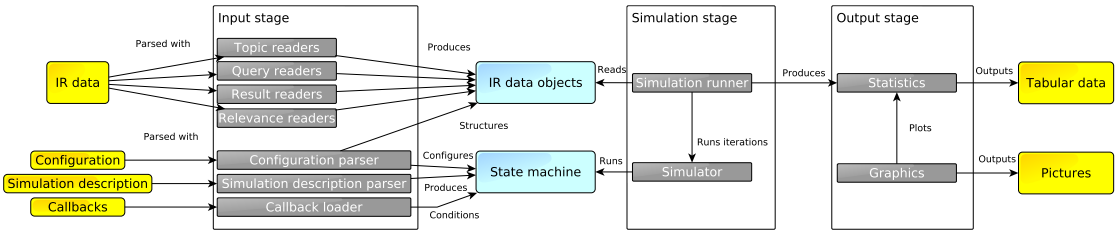

Figure 4.1 *Components of the simulator software. Yellow items represent input or output files, blue items represent internal data structures, and grey items represent software components.*

4.7 Parsing input files

As explained in Chapter 3, most of the input files do not follow any established file format, instead using their own proprietary, albeit simple, formats. As established in Section 4.5, file readers use the OperatesOnFile mix-in, which expects readers to implement a method that produces records from a file. Depending on the input file, the records can either be Topics, Queries, Documents or Relevances, as per the object model. Such a method had to be implemented for all the input file types.

The TREC result and relevance files are simple in their format: each line contains one record, and each value in a record is separated by white space. Reading such files with Python is quite straightforward. The built-in *open* function returns a file handle that can be iterated in a loop, producing one line from the file for each iteration. Using a *regular expression* to denote the structure of a line, creating a group for each value, and then matching the expression with the line produces a tuple of the values. The tuple can then be used to build the expected record type, either a Document or a Relevance. This pattern was used to implement the iterator methods that the OperatesOnFile mix-in expects.

The Indri query files are the exception among the input files in that they follow a file format that is well-defined. The file is in XML format, and is therefore easy to parse with Python, using the built-in *xml* module. Using the *ElementTree* class, an XML document can be parsed into a tree-like Python object. The object supports multiple ways to search for elements within the XML tree, and they were used to parse the query identifier and query text from the file. Due to the restriction that there should be only one query per query file, the file reader was made to throw an error if encountering multiple queries. For the same reason, the iterator method can only produce one Query record.

Parsing of TREC topic files was also considered, but due to their contents being largely irrelevant for the purposes of interaction simulation, it was decided that they would be ignored. Had they been found more useful, parsing would have been implemented using an SGML parser, for instance the built-in *sgmllib* that is only available in Python 2, or the external *Beautiful Soup* library that is compatible with Python 3.

4.8 Parsing the configuration file

As the configuration file was implemented in XML format, it was simple to read into objects using Python. The naïve approach would have been to use the built-in XML module. However, since an XML Schema was developed for the file format, it was deemed necessary to use PyXB to read the configuration file in order to validate it before using it. For parsing the file, a wrapper class was implemented, acting as a protective layer against changes in the configuration file format causing changes elsewhere in the code. Instead, future file format changes will only cause changes in the wrapper class. The PyXB API works almost identically to the built-in ElementTree class, making its use straightforward if already familiar with the native XML module.

4.9 Parsing the simulation description file

As with the configuration file, parsing of the simulation description file was done using the PyXB library. A wrapper class was also implemented, for the same purposes as explained for the configuration file. Due to the complexities present in defining a user model, the XML Schema based validation only works for limited purposes. Therefore, some of the validations had to be implemented in the wrapper class. These validations are limited to checking whether the callbacks used actually exist, and whether the transitions attempted by the simulator are actually valid.

4.10 Callback plug-ins

The architecture of the software allows end users to write user models by using a purpose-built language, as described in Section 4.2. While designing the language, it became apparent that users required a means to define how gains, costs and boolean

condition values were calculated. Furthermore, it was required that one could write new triggers that alter the simulation state, to be fired when entering a state.

At first, an attempt was made to have the user model language support such constructs. However, it soon became apparent that any such approach would only result in defining yet another programming language, which was both unnecessary and infeasible. Therefore, it was decided that the users would be allowed to write their calculation methods as callback functions in Python, and their code would be plugged-in to the software at run-time.

Python supports loading code dynamically using the standard library `imp` module that gives access to the mechanisms that are used to implement the built-in `import` statement. First, one must call the `imp.find_module` function with the file name to search for and a list of directories to search in. The function returns a tuple that can then be passed to the `imp.load_module` function that loads the file as a module, and returns a module object that can then be used for calling any code in the loaded file.

For the simulator software, a simple callback loader was implemented, as showcased in Program Listing 4.11. The loader implements a `get_callback_module` function that takes a file path as an argument. The function separates directory names from the path and passes the bare file name and the separated directory name, along with the current directory to the `imp.find_module` function. It then attempts to load the module using `imp.load_module`. The function returns the module if it was successfully loaded – otherwise it returns a `None` object, since it was designed not to crash even if the module is not found or cannot be loaded.

The user defines where to load the callbacks from in the configuration file, as defined in Section 3.9. The user is then free to use the callback functions in the user model. If the user attempts to use a callback function that could not be loaded, the simulator will stop executing and issue an error message.

Each callback file can contain multiple callback functions. It was decided that the file format would be such that the users can freely write the code however they wish. However, an entry point into the callbacks was required so that the software would know what to run when encountering such a file. Therefore, it was mandated that a callback file should implement a method called `get_callback_map` that returns a

Program Listing 4.11 *Dynamic code loader*

```
import imp
import os
import ntpath

def path_leaf(path):
    head, tail = ntpath.split(path)
    return tail or ntpath.basename(head)

def get_callback_module( name ):
    script_dir = os.path.dirname(os.path.realpath(__file__))
    callback_module_dir = script_dir + '/' + ntpath.dirname( name )
    callback_module_name = path_leaf( name )

    fp = pathname = description = None
    try:
        fp, pathname, description = imp.find_module(callback_module_name, [
            callback_module_dir, os.getcwdu(), script_dir])
        return imp.load_module(name, fp, pathname, description)
    except:
        return None
    finally:
        if fp:
            fp.close()
```

dictionary of callback functions mapped by their names. An example of the structure is presented in Program Listing 4.12.

Program Listing 4.12 *A callback file*

```
def contiguous_non_relevant_snippets_seen_reached( simulation, negation, amount
    ):
  seen_docs = simulation.get_current_query_seen_documents()
  relevance_levels = [simulation.get_document_relevance_level( docid ) for docid
       in seen_docs]
  count = 0
  while len( relevance_levels ) > 0:
    if str(relevance_levels.pop()) == '0':
      count += 1
    else:
      break
  return int(count) >= int(amount)

def get_callback_map():
    return { 'contiguous_non_relevant_snippets_seen_reached':
        contiguous_non_relevant_snippets_seen_reached }
```

4.11 Running a simulation

Stochastic simulation requires that the simulation is iterated multiple times in order to produce stable results. The process of running a single iteration is described by the algorithm in Section 3.3. This algorithm was implemented within the `Simulation`

class since objects of the class contain the high level knowledge of the entire simulation. The implementation follows the defined algorithm exactly.

Multiple iterations were implemented by creating `Simulation` objects in a loop that runs as many times as the user has specified in the configuration file. The Simulation objects are then let run the simulation cycle until they reach a final state, stopping the cycle. Each `Simulation` object is then stored in a list for further analysis that includes calculating statistics and plotting graphics for them.

4.12 Recording the simulation runs

In order for the simulator to be prepared for all research cases, all user actions and decisions leading to them are recorded. Such recording can be very widely spread across the program code. Utilisation of the *Observer pattern* (Gamma et al. 1994) allows the decision-making points to notify external recorders about an event that has occurred without taking part in the recording themselves.

The observer pattern is a design pattern that enables observable *subjects* to notify their *observers* about events occurring in the subject's context. An interface, implemented by the observers, defines the method that the observed subjects use to notify them of events. Another interface, implemented by the subjects, defines the methods with which the observer can register or unregister themselves for receiving notifications. The added layer of indirection serves to discouple observers from the implementation of the observable subjects, allowing the observers to observe different kinds of events and the subjects to be observed by any number of various kinds of observers.

In order to enable the observers receive more specific information about the events, an *event object* may also be sent to them along with the notification. Event objects implement an interface that defines methods for getting the *event identifier* and any properties associated with the event. The observable subject interface may also allow observers to register for only certain events by providing event identifiers along with the registration request.

In the simulator software's case, there are different kinds of observable events. First, the transitions, as they occur, must be recorded along with the full simulation state at that point: the transitions available, the transition chosen, the random probability

value used, the simulator events triggered and the current result document. This serves to record the sequence of transitions taken. Second, each iteration over the transition set Δ_X (see Section 3.2) must be recorded in order to record the factors that led to choosing each transition: the actions considered, and the probabilities of considered actions. The observable subject is the simulation object.

The elementary events that need to be observed are *concrete transitions* (as opposed to declared transitions) and *action considerations*. The observers form an hierarchy where an *abstract recorder* handles basic tasks of recording information into a log object, and two different types of *concrete recorders* handle the different event types. The log object in this case is the list that contains the history of simulation states, with all the observed events recorded with their respective states.

4.13 Calculating statistics

After all the simulation iterations have finished, there exists a list of Simulation objects, one per iteration. For each of these Simulation objects, a set of statistics over cost or rank can be calculated. When these sets are averaged, the end result is a stable set of averages. This method is the cornerstone of how Monte Carlo simulations are done.

To calculate these statistics, a separate module was devised. The module was decided to be kept separate from the simulation due to performance concerns and for the separation of concerns. If needed, the module could be moved into a separate application instead of being part of the simulator software itself, which makes it more complex. However, for the time being, the statistics module was left in.

Calculating the statistics themselves is fairly straightforward, due to them being mostly just a matter of calculating averages and standard deviations. The case of calculating the average of multiple different-length data sets, as is possible when simulations can end arbitrarily, was solved by considering all the data sets as being as long as the longest one of them, and then extending the shorter data sets by appending their last value to the end of the set as many times as needed, until the expected length was reached.

4.14 Drawing figures

In order to make the calculated statistics easier to give a quick glance or make a rough comparison, a graphical representation was required. This requirement was decided to be satisfied by plotting the data sets into image files that can then be looked at side by side using any available software that can display pictures.

As with the statistics module, the plotting module was implemented as an isolated module in order to make it easier to separate it from the main software later, if need be. For producing the image files, the `pyplot` interface of the *matplotlib* library (Hunter et al. 2016) was utilised. Plotting a figure with the interface is straightforward: a `Figure` object acts as a container for `Plot` objects that hold the actual data that should be drawn, and a figure object can then be saved into a file in many standard picture formats.

For producing the plots, the plotting module works in conjunction with the statistics module. Since most of the statistics have in usual cases already been calculated for textual output, the statistics module employs memoization in order to avoid making the expensive calculations again. The memoization process allows the statistics module to store the calculated statistics within the module, simply returning the calculated values when the calculation methods are called again.

4.15 Distribution

As established earlier, the intended user group consists of academic professionals with a moderate-to-high level of proficiency using different kinds of computer systems and software. Therefore, distributing the software to them in source format, without packaged-in dependencies does not pose a problem, as long as the dependencies are documented or a tool like pip is used to record what dependencies are required. However, distributing the software as an easy-to-install ready-made package saves a lot of time, especially when required software packages are added or changed.

The Python standard library offers a distribution tools package called *Distutils* (Beazley and Jones 2013, p. 435). With Distutils, developers can create distribution packages easily by defining what files should be included in the distribution. However, defining and packaging dependencies with it is cumbersome, and often leads

to cross-platform incompatibilities since a Python library may require different versions to be installed on different platforms. As long as a source-only distribution package without packaged dependencies is usable for the target users, Distutils does its job nicely.

Multiple enhanced distribution utilities also exist, such as *Setuptools* (Beazley and Jones 2013, p. 435) by the Python Packaging Authority and the *wheel* building utility of pip (Danjou 2014, p. 68). Wheel is a package format for distributing Python software. It aims to supersede the older *egg* packaging format, and offers a number of enhancements. A wheel package is easy to install with pip on any platform where dependencies can be met, therefore making it a far better alternative to just using Distutils.

In this project, the use of setuptools, eggs and wheels was considered. In the end, however, it was determined that the user base is comfortable with installing dependencies manually, or by using pip. Therefore, there was no perceived need to use any more advanced packaging than the source distribution packaging style offered by Distutils. The future of the software may still see the introduction of a more advanced packaging tool, where the most likely tool for introduction is wheel, using the pip wheel utility.

As established, the target user group was considered to be comfortable with installing dependencies manually or with the pip package installation tool. Therefore, the Distutils *sdist* source distribution package format was chosen for distributing the software to end users.

To build source distribution packages with Distutils, one must first write a `setup.py` file that describes which files to include in the distribution. The typical way to define what source files are included is by defining the *modules* that belong in the distribution. A module is basically a block of Python code contained in a single `.py` file.

With a bigger software package, defining all files one by one is impractical. Therefore, it is also possible to define the included files in terms of *packages*. A package is a module that contains other modules – that is, single files or other packages. Packages are contained in directories in the file system, and to mark a directory as a package, an `__init__.py` file must be present. The file may be left empty, or it may contain code for initialising the package.

Typically, the `setup.py` file also contains meta-information on the package as well. It may include items such as the name of the package, version number, a textual description, information on the author and the URL of the software's website.

Since the simulator software consists of a moderately big number of modules, they were originally divided into multiple sub-directories. Those directories turned very naturally into packages just by including an `__init__.py` file in each one of them. The packages were then included in the distribution simply by passing the *packages* argument when calling `setup` in the `setup.py` file, as shown in Program Listing 4.13 that showcases the actual `setup.py` file used. Furthermore, example configuration files and standalone Python modules were included in the distribution, as well as a helper script for running the software from the command line.

Program Listing 4.13 *Distutils setup.py file for distributing the software*

```
1  from distutils.core import setup
2  setup(name='irsim',
3        version='0.1.23',
4        py_modules=['irsim', 'callbackLoader', 'stats', 'figures'],
5        packages=['qsdl', 'qsdl.parser', 'qsdl.simulator', 'qsdl.simulator.errors'
             , 'example-config'],
6        package_data={ 'example-config': [ '*.xml', '*.xsd' ] },
7        scripts=['irsim'],
8        url='http://www.github.com/fire-uta/ir-simulation/',
9        author='Teemu Paakkonen, University of Tampere',
10       author_email='teemu.paakkonen@uta.fi'
11       )
```

After defining the distribution using the `setup.py` file, the package can be built by invoking the file from the command line with the `sdist` parameter. This builds a single package file that contains the specified source files, along with a `PKG-INFO` file that contains meta-data on the package. This package file was successfully used to distribute the simulator software to end users, along with short instructions on how to install dependencies. The approach proved to be sufficient.

5. EVALUATION

This chapter discusses the impact of design choices on the workings of the finished software, how the software fits its intended purpose, and also the impact of the software on research.

Section 5.1 evaluates the software through a case study that compares simulated behaviour to that of actual humans. Section 5.2 assesses the software architecture. In Section 5.3 the research work done using the software is visited. Section 5.4 discusses how the developed user modelling language fulfilled its purpose. Finally, Section 5.5 gives insight into the possible future of the software.

5.1 Testing the ability to predict behaviour

Since the main research question was whether simulating user behaviour accurately is possible, evaluation of simulation results was needed. Such evaluation can be looked into from two different perspectives. When absolute replication of behaviour is desirable, the behaviour of the user model needs to be evaluated against that of the real users. On the other hand, when merely similar results are required, for instance if the simulation is used to evaluate search engines, mostly the final results achieved by the simulation are important. Both perspectives were deemed to be worthy of examination. In order to undertake such evaluation, a case study was conducted.

5.1.1 User model evaluation

The simulator software requires a valid user model in order to produce valid results. In order to make sure a certain user model produces valid results, it must be tested against real world data, using actual data generated by human actions as a reference point (Johnson and Taatgen 2005).

The aim of evaluation is to confirm that the results produced are consistent with the observations they are based on – that is – the simulated user behaves like the actual users observed. This requires that the simulation records all simulated user actions, in order to facilitate reducing them into statistics, that can in turn be compared against the statistics gathered from user observation.

As mentioned, user model evaluation requires proper support from the software. As outlined in Section 4.12, the Observer pattern is utilised throughout the software in order to store accurate data that can later be used for analysis. This design choice proved to be useful for user model evaluation purposes as well.

5.1.2 Case study

In order to test user modelling, a case study was conducted. For the case study, a model was created using a previous laboratory study (Maxwell and Azzopardi 2014) as a reference point. Behavioural probabilities established by analysing the data gathered in the study were used for decision making in the model. The goal of the case study was to create a model that mimics previously observed real user behaviour. The results produced by the model were expected to then mirror the empirical research results, thus validating the model.

In their research, Maxwell and Azzopardi (2014) studied how network and data processing related delays affect user interactions with a search system. They found that users will spend more time reading documents and SERPs per query when different kinds of delays are present. They also found that users will make fewer queries when the relative cost of querying is higher, and that users will make more queries when the relative cost of reading documents is higher.

While the original research results aren't that interesting from the point of view of testing a user model, the data gathered from the user study is highly suitable for creating a testable user model. The user study set-up included a logging facility for capturing the following data:

- queries issued by the user;
- interactions with SERPs;
- documents viewed, and their relevance assessments.

Maxwell and Azzopardi point out that the gathered log data can be used to calculate times spent performing any activity. The same data, while not used for such purpose by Maxwell and Azzopardi, allows one to assess the probabilities of performing the actions based on such factors as document relevance, time spent, or cumulated gain.

Using the original log data, costs and probabilities were calculated for each document relevance level, in order to base a simple user model on them. Relevance levels were given using three-point assessment, where level 0 means the document is irrelevant, and levels 1 and 2 mean the document was relevant to a low or high degree, respectively. The original data contained four different types of query sessions: 1. ones with no artificial delays, denoted with BL, 2. ones with artificially introduced querying delays, denoted with QD, 3. ones with artificial document loading delays, denoted with DD, and 4. ones with both querying and document delays, denoted with QDD.

Since the separate delay types allowed for testing of multiple different types of user models, a separate evaluation was performed for each type, using four-fold cross-validation. Three different types of strategies for moving on to the next query (*stopping strategies*) were employed for each delay type, after Maxwell et al. (2015): a fixed-type strategy where the query is changed after n SERP snippets have been seen, denoted with SS-fix, a strategy where the query is changed after n non-relevant SERP snippets have been seen during a single query, denoted with SS-tot, and a strategy where the query is changed after n consecutive non-relevant SERP snippets have been seen during a single query, denoted with SS-seq. Table 5.1 illustrates the ranges of average times calculated using the log data, and Table 5.2 shows the average probabilities calculated using the data. The data also included practice sessions, which were expunged before any calculations were made.

Table 5.1 *Costs used for the user model (seconds)*

Property	**BL**	**QD**	**DD**	**QDD**
Document reading time	15.5–21.3	14.9–19.6	13.7–19.8	25.2–29.1
Query formulation time	8.0–8.6	7.8–8.4	8.2–9.9	9.5–10.8
Snippet scan time	4.9–6.3	5.7–6.6	6.2–7.2	8.6–9.4

In order to run simulations, the cost and probability data gathered by analysing the logs were transformed into a simulation description file. To keep things simple, the simulation was modelled after the very simple example automaton presented in Figure 3.2. Each cost and probability presented in Tables 5.1 and 5.2 was turned into

Table 5.2 *Probabilities used for the user model*

Property	Relevance	BL	QD	DD	QDD
Read probability	2	0.57	0.52	0.54	0.54
	1	0.55	0.45	0.53	0.53
	0	0.32	0.32	0.24	0.30
Consider-as-relevant probability	2	0.79	0.75	0.88	0.78
	1	0.61	0.64	0.76	0.59
	0	0.45	0.54	0.69	0.55

a cost or probability element, respectively, following the definitions in Section 4.2. The stopping strategy parameter values were also calculated by analysing the logs, and incorporated into the user model.

The automaton was set up to stop after 1200 seconds because that was the original time limit set by Maxwell and Azzopardi in the user study. The test subjects were, however, given the probability of stopping before reaching the time limit if they felt they had found enough relevant documents, or became "fed up". In other words, perceived success was the deciding factor behind stopping early. This was not modelled in the case study, making the reaching of the 1200 second time limit the only way to stop, in addition to running out of queries or results.

The document collection, search topics and search engine result lists used in the simulation also had to match the ones used in the empirical study. The documents and topics were acquired from the original source, the NIST TREC data collection, and the results were re-generated using the same search engine as the one used in the user study.

Four-fold cross-validation was performed by splitting each user group into four equal-sized sub-groups. Three of the sub-groups were used for calculating the cost and probability values, forming the *training data* for the user model, and the one remaining sub-group was used for building the queries and result lists. The simulation results were then compared to the sub-group whose queries were used in the simulation. The four sub-groups were then rotated and the experiment repeated using data re-calculated from that configuration. Four rotations were made in order to use all possible combinations.

The four-fold cross-validation was repeated for all user groups, as separated by their delay types, forming a total of 16 training data sets. Each data set was simulated

with all of the stopping strategies, making the total number of simulation runs 48. The results were calculated as the difference between average cumulated gains over each group of four training data sets that belong to the same delay type, and their corresponding groups of real user query sessions.

The results, showcased in Table 5.3, show that the error between the simulations and real users decreased with the use of the more complex stopping strategies SS-tot and SS-seq. In best cases the error remained below 10% and even in the worst cases, below 25%. Furthermore, statistical analysis showed that there was no statistically significant difference between the simulated users and the real ones, making the simulated users just as good for evaluation purposes as real users.

Table 5.3 *Average error percentages between real and simulated users' cumulated gains over entire sessions, for different delay types and stopping strategies.*

Delay type	**SS-fix**	**SS-tot**	**SS-seq**
BL	16.2 %	10.9 %	12.0 %
QD	23.7 %	18.4 %	9.5 %
DD	13.3 %	10.2 %	16.7 %
QDD	24.6 %	18.5 %	16.1 %
All	14.4 %	9.3 %	8.4 %

The results were also compared by recording the number of documents interacted with by the simulation, and comparing those to the numbers of documents interacted with by the real users. Three different interactions were recognised: scanning a SERP snippet, viewing a document, and marking the document as relevant. For real users, the marking interaction was explicit since the original experiment required it, but for simulated users there was no actual marking interaction. Instead, there was a state where the simulation would *consider* the document as relevant, and that was considered to correlate with the marking interaction.

After analysing the document interactions, it was found that on average, the simulations interact with far fewer documents than the real users. The error between real and simulated users was 18.5 – 41.9 % for the number of scanned snippets, 8.4 – 28.6 % for the number of viewed documents, and 6.3 – 23.6 % for the number of documents marked as relevant. The QD condition displayed the least amount of error for all measured values. The otherwise well-performing SS-seq stopping strategy incurred more error than the other tested strategies, showing that having low error

when comparing session effectiveness does not necessarily correlate with low error in the similarity of actions. The results are shown in Table 5.4.

Table 5.4 *Average error percentages between real and simulated users' document interactions: seen snippets, clicked documents and marked-relevant documents. Given per different delay types and stopping strategies.*

Delay type	Stopping strategy	Seen	Clicked	Marked
BL	SS-fix	33.2 %	21.7 %	18.9 %
	SS-tot	31.4 %	18.8 %	14.7 %
	SS-seq	41.9 %	28.6 %	22.1 %
QD	SS-fix	18.9 %	9.4 %	7.9 %
	SS-tot	18.5 %	8.4 %	6.3 %
	SS-seq	30.2 %	19.4 %	16.3 %
DD	SS-fix	25.8 %	12.6 %	11.7 %
	SS-tot	24.0 %	9.8 %	8.5 %
	SS-seq	34.8 %	21.8 %	20.2 %
QDD	SS-fix	27.4 %	16.1 %	15.4 %
	SS-tot	28.5 %	16.1 %	15.4 %
	SS-seq	38.8 %	25.9 %	23.6 %
All	SS-fix	26.2 %	14.9 %	13.4 %
	SS-tot	25.5 %	13.5 %	11.7 %
	SS-seq	35.4 %	22.5 %	19.6 %

The comparison between the final results showed that the simulated users are as good as real users when comparing session effectiveness. Therefore, a simulated user can be used for evaluating the impact of aspects such as the differences between search engine algorithms, differences in user interfaces, and so on. However, since there were big differences between the simulated users and real ones when considering the documents interacted with, simulated users don't appear to be good for evaluating the effects of differing user behaviours. The differences can likely be lessened by improving the user model, but without further testing it cannot be confidently asserted that the simulator would be useful for evaluating the effects of changes in searcher behaviour.

5.2 On the software architecture

Since the simulator software simply runs state machines, the architecture is quite simple. In theory, the architecture should be easy for others to understand and work on, so that anyone can continue developing and maintaining the software. In

practice, since, at the time of writing, there have been no other developers working on the software, the assertion remains to be tested.

One of the biggest problems with the architecture is that the program code is written to be compatible with Python 2 only. Migrating to Python 3 would require not only changes to the core software itself, but also the libraries used. For example, the PyXB library is incompatible with Python 3, and would have to be replaced. Fortunately, the technologies used in the software have all reached such a state of maturity that upgrading the libraries has become unnecessary, making staying with Python 2 a lesser issue.

The software was been designed in such a way that almost any kind of experiment should be possible to conduct without making changes to the software itself. This has proven to be true. Multiple experiments have been conducted with the software, and while some modifications to the software have been made in order to make it easier to use, the experimental set-ups themselves have only required making changes to the user model and simulator configuration.

The worst issue with the current state of the software is that with multiple Monte Carlo iterations run times can be very long. Long runs can also use a lot of memory. One way to improve performance would be to use another Python interpreter, such as PyPy that features a just-in-time compiler that should theoretically speed up just about any python program. Since Python programs use only a single processor core by default, most of the processing capacity of a typical modern multi-core computer is left unused. Making the software run multiple Monte Carlo iterations concurrently in separate threads could, in the best case, speed up running times by the number of physical processor cores present in the system.

5.3 Research done using the software

At the time of writing, there has been one published research paper where the simulator software has been used. The software was used to conduct an experiment on what separates an expert searcher from a novice. Multiple user models were used to test which behavioural factors contribute most to the effectiveness of searching. (Pääkkönen et al. 2015)

Another unpublished research paper is under peer review at the time of writing. There, the software was used to test how well a widely-used searcher user model

replicates the behaviour of real users. The findings were encouraging since the user model produced results that were statistically indistinguishable from real users. Some of the results were reported in Subsection 5.1.2.

Since the previous studies proved the usefulness of the software, further research work is also planned. Previous research uncovered the need for improving the commonly used searcher user model, and the software should prove useful for attempting to improve it.

5.4 On the simulation description language

As mentioned in Section 4.2, in the beginning of the project, a requirement for a language for defining user models for the simulator was specified. The resulting XML-based language is both complex and difficult to approach for newcomers. The schema also allows for "impossible" simulations where the problems can only be detected run-time, and in some cases, cannot be detected programmatically at all, requiring strict and often laborious manual scrutiny of the user model.

The main caveat of XML itself is its verboseness. Writing long XML files can be tedious when done manually, due to the syntax requiring all elements and attributes to have names. There is no array construct in XML, which makes simple lists of values tremendously more verbose than necessary. An actual simulation description can therefore be much longer than when written in some less verbose language. To battle the verboseness, a wizard-like graphical tool was considered. However, it was considered too much work for the time being, and was postponed for future development of the software.

While not a caveat of XML per se, the choice of XML also introduces the shortcomings of XML Schema. XML Schema suffers from a verbosity problem, since it is an XML-based language. Furthermore, it is designed to account for all kinds of complex document types, which makes writing schemas somewhat tedious. Even simple schemas can have very deeply nested element trees, which also hurts the document's human-readability.

Nevertheless, the language is versatile enough that it can be used to specify almost any kind of user model, as long as it can be reduced to a non-concurrent state machine. The real-world research cases highlighted in Section 5.3 stand as proof for usefulness despite the problems.

In the end, however, a ready-made and thoroughly thought out solution by a reliable third party seems like a more robust and workable approach. For example, W3C's *SCXML* (State Chart XML) (Hosn et al. 2015) offers many of the features of our in-house language. In addition to allowing the definition of states and transitions, it allows defining executable content to run when entering or leaving a state (analogous to triggers), storing data alongside the state machine in the same fashion how documents, queries, topics and relevances are stored now in the software, and conditional transitions (analogous to conditions). Furthermore, SCXML also features the possibility for multiple simultaneously active states, which would allow for EPIC-style user model parallelism.

What SCXML does lack, however, is direct support for probabilistic transitions. It is of course possible to write such conditional transitions that compare a probability value to a random variable whose value is set in an executable content section, effecting a probabilistic transition. In our case, an even better approach would be to write an extension of the schema, adding support for the simpler way of defining probabilities present in the simulation description language. Adopting SCXML in either manner would almost certainly lead to having to rewrite the entire software.

Since SCXML is a W3C recommendation, the specification also contains guidelines on how to implement software that can handle SCXML documents. Using such guidelines as a basis for automaton-based simulator software should in theory result in a better implementation, even if the simulator wasn't built to support SCXML in its entirety, but a subset thereof.

However, the software, as it is, would not benefit from the guidelines without the adoption of SCXML as well, since the simulation description language differs markedly enough from it. Therefore, all approaches to adopting SCXML, or parts of it, or even just the guidelines, would likely result in a complete rewrite of the software. There being no compelling basis for such revamp, the idea must be put aside for future projects.

5.5 Future work

Despite the fact that the software architecture has proved to be sufficient enough, so that there have been no major changes to the software since it was finished, there

is still much room for improvement. Maintaining the software in its current state is hard, and writing user models has also emerged as being difficult for end users.

As mentioned earlier, one of the problems with the software architecture is that the code base only works with Python 2. Migrating to Python 3 should make it easier to maintain the software in the future since most Python libraries currently work with Python 3, and may stop supporting Python 2 in the future. In the simulator software, the PyXB library used for XML handling only supports Python 2. Therefore, a replacement that supports Python 3 would be needed. Fortunately, there is an XML library called *LXML* that has mostly all the same features as PyXB, and supports Python 3. Therefore, in order to migrate to Python 3, the architecture would have to be changed to use LXML for XML handling. In addition, the code base would require some changes due to the changes mandated by Python 3 specifications.

Another problem with the current implementation is that the XML-based user modelling files and configuration files are somewhat complex to write by hand. Currently, end users are often having trouble with the files, and need constant support when making changes to them. The remedy for this would be to create a graphical tool that guides the user through creating the files. That way, the end user doesn't have to bother with understanding the internal data structure, and they can just concentrate on the user model itself. A likely platform for such a tool would be something web based, in order to cater for most operating systems and devices.

While the software architecture works well for Cranfield-type experiments, there is one aspect that is not supported. Since typical users always learn or forget things during a search session, the parameters of the user model are constantly changing. This can of course be modelled by using custom callbacks to calculate probabilities, costs and gains. However, the users also learn new information from the topic, and apply their newly gained knowledge by making better queries. Since the software can only use a fixed set for queries for each session, this aspect cannot be taken into account. The simulator software used by Maxwell et al. (2015) is able to formulate new queries based on topical information, while being otherwise quite different. Incorporating such features into this software would allow researchers to venture into new kinds of research areas, where query formulation is an integral part of the user model.

As discussed in Section 5.2, the worst issue with the software architecture is the performance with complex models and large Monte Carlo runs. The obvious way to improve performance would be to make the Monte Carlo iterations run in parallel threads or processes. While Python 2 already supports threading and launching processes, Python 3.2 has an improved feature for managing concurrency called *concurrent.futures*. It offers a way to create *futures* that act as proxy objects for values that are resolved later. This way, execution can continue without the value being immediately present, and will only stop to wait for the value when it is needed. With the use of futures, it would be quite easy to implement a solution that never waits for a single iteration to finish. The caveat of this approach would be that it requires Python 3, and it is currently not supported by the software. Therefore, in order to implement concurrent processing, there are two options: either make do with the old threads support of Python 2, or migrate to Python 3 for a better solution to concurrency.

6. CONCLUSIONS

In order to reduce the costs of experimentation in information retrieval research, a software that simulates user interaction with a search interface was created. The software was designed to emulate the Cranfield model of IR experiments, using Indri query files and TRECEVAL result files as input. An XML-based language for creating user models for the software was devised, and the software was designed to be able to utilise any user model described with the language. The language was designed to be as flexible as possible, to allow almost any kind of experimental set-up to be built. The software measures session effectiveness using cumulated gain as the main metric. Due to the stochastic nature of the user models, the simulator utilises Monte Carlo methods in order to produce meaningful results.

The software proved to be useful for research purposes, having been used for one published and one as-of-yet unpublished research paper, at the time of writing. The software architecture proved to be robust, since very little program code changes had to be made for the experimental set-ups. The user modelling language was found to be so flexible that multiple different kinds of experiments could be made with it. While comparing the simulated users behaviour to that of real users, it was found that there was no significant statistical difference when using a commonly-utilised IR user model, thus validating the simulator's use for many kinds of experimentation.

At the time of writing, the simulator software has been used for one published scientific paper (Pääkkönen et al. 2015). Another unpublished paper is pending peer review, and is likely to be published in 2016. More research is also planned, and further publications are forthcoming.

Shortcomings of the software include the dependency on Python 2, version 3 not being supported. The user modelling language is also too complex for end users to effectively write, as there is no assisting graphical tool to help. Furthermore, the software is unoptimised, performing poorly when the number of Monte Carlo iterations is high.

In the future, the simulator will be used for more research work, likely focusing on creating more advanced user models. Work on the software itself will also continue, the main focus being on performance optimisation and moving towards Python 3 support. A look into query formulation within the software will also be taken.

In conclusion, the software was successfully implemented, and it serves the purpose it was designed for. Work on the software will continue to make it even better.

BIBLIOGRAPHY

Azzopardi, L., K. Järvelin, J. Kamps, and M. D. Smucker (2011). "Report on the SIGIR 2010 Workshop on the Simulation of Interaction." In: *SIGIR Forum* 44.2, pp. 35–47.

Baeza-Yates, R. and B. Ribeiro-Neto (2011). *Modern Information Retrieval: the concepts and technology behind search, Second edition.* Reading, Massachusetts, USA: Addison-Wesley. 913 pp.

Beazley, D. and B. K. Jones (2013). *Python Cookbook, 3rd Edition.* Sebastopol, California, USA: O'Reilly Media. 706 pp.

Bigot, P. A. (2014). *PyXB: Python XML Schema Bindings.* URL: `http://pyxb.sourceforge.net/` (visited on 15/03/2016).

Chauve, A., A. Espaze, E. Gouillart, G. Varoquaux, and R. Gommers (2016). *Scipy : high-level scientific computing.* URL: `http://www.scipy-lectures.org/intro/scipy.html` (visited on 15/03/2016).

Croft, W. B., D. Metzler, and T. Strohman (2010). *Search Engines: Information Retrieval in Practice.* Reading, Massachusetts, USA: Addison-Wesley. 520 pp.

Danjou, J. (2014). *The Hacker's Guide to Python.* Lulu.com. 290 pp.

Eckard, E. and J.-C. Chappelier (2007). *Free Software for research in Information Retrieval and Textual Clustering.* Tech. rep. Lausanne, Switzerland: École polytechnique fédérale de Lausanne.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Massachusetts, USA: Addison-Wesley. 416 pp.

Geyer, C. (2011). "Introduction to Markov Chain Monte Carlo." In: *Handbook of Markov Chain Monte Carlo.* Ed. by S. Brooks, A. Gelman, G. L. Jones, and X.-L. Meng. Chapman and Hall/CRC. Chap. 1, pp. 3–48.

Harman, D. (2011). *Information Retrieval Evaluation.* San Rafael, California, USA: Morgan & Claypool. 107 pp.

Hosn, R., J. Carter, D. Burnett, T. Lager, J. Barnett, T. Raman, S. McGlashan, R. Auburn, J. Roxendal, K. Reifenrath, R. Akolkar, N. Rosenthal, M. Bodell, and M. Helbing (2015). *State Chart XML (SCXML): State Machine Notation for Control Abstraction.* Recommendation. W3C. URL: `http://www.w3.org/TR/2015/REC-scxml-20150901/`.

Hunter, J., D. Dale, E. Firing, and M. Droettboom (2016). *Matplotlib, Release 1.5.1.* URL: `http://matplotlib.org/contents.html` (visited on 15/03/2016).

Ingwersen, P. and K. Järvelin (2005). *The Turn: Integration of Information Seeking and Retrieval in Context.* Secaucus, NJ, USA: Springer-Verlag New York, Inc.

John, B. E. and D. E. Kieras (1996). "The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast." In: *ACM Transactions on Computer-Human Interaction* 3.4, pp. 320–351.

Johnson, A. and N. Taatgen (2005). "User Modeling." In: *The Handbook of Human Factors in Web Design.* Ed. by R. Proctor. Mahwah, NJ, USA: Lawrence Erlbaum Associates, pp. 424–439.

Järvelin, K. and J. Kekäläinen (2002). "Cumulated Gain-based Evaluation of IR Techniques." In: *ACM Transactions on Information Systems* 20.4, pp. 422–446.

Järvelin, K., S. Price, L. Delcambre, and M. L. Nielsen (2008). "Discounted Cumulated Gain Based Evaluation of Multiple-query IR Sessions." In: *Proceedings of the IR Research, 30th European Conference on Advances in Information Retrieval.* ECIR'08. Glasgow, UK: Springer-Verlag, pp. 4–15.

Kalos, M. H. and P. A. Whitlock (2008). *Monte Carlo Methods: Second Revised and Enlarged Edition.* Wiley-VCH Verlag.

Kettunen, K. (2007). *Reductive and generative approaches to management of morphological variation of keywords in monolingual information retrieval.* Tampere, Finland: Tampere University Press.

Kieras, D. E. (2005). "Fidelity issues in cognitive architectures for HCI modelling: Be careful what you wish for." In: *Proceedings of the 11th International Conference on Human Computer Interaction (HCII 2005).* Mahwah, NJ, USA: Lawrence Erlbaum Associates, pp. 22–27.

Kieras, D. E., S. D. Wood, and D. E. Meyer (1995). "Predictive Engineering Models Using the EPIC Architecture for a High-performance Task." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* CHI '95. Denver, Colorado, USA: ACM Press/Addison-Wesley Publishing Co., pp. 11–18.

Law, A. M. (2008). "How to Build Valid and Credible Simulation Models." In: *Proceedings of the 40th Conference on Winter Simulation.* WSC '08. Miami, Florida: Winter Simulation Conference, pp. 39–47.

Manning, C. D., P. Raghavan, and H. Schütze (2009). *An introduction to information retrieval.* Cambridge University Press.

Martin, R. C., M. C. Feathers, T. R. Ottinger, J. J. Langr, B. L. Schuchert, J. W. Grenning, and K. D. Wampler (2009). *Clean Code: A Handbook of Agile Software Craftsmanship*. Boston, Massachusetts, USA: Pearson Education, Inc. 431 pp.

Maxwell, D. and L. Azzopardi (2014). "Stuck in Traffic: How Temporal Delays Affect Search Behaviour." In: *Proceedings of the 5th Information Interaction in Context Symposium*. IIiX '14. Regensburg, Germany: ACM, pp. 155–164.

Maxwell, D., L. Azzopardi, K. Järvelin, and H. Keskustalo (2015). "An Initial Investigation into Fixed and Adaptive Stopping Strategies." In: *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '15. Santiago, Chile: ACM, pp. 903–906.

O'Grady, S. (2015). *The RedMonk Programming Language Rankings: June 2015*. URL: `https://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/` (visited on 11/01/2015).

Python Software Foundation (2016a). *The Python Package Index (PyPI) (Python 2.7.11 Documentation)*. URL: `https://docs.python.org/2/distutils/packageindex.html` (visited on 08/03/2016).

Python Software Foundation (2016b). *The Python Tutorial (Python 2.7.11 Documentation)*. URL: `https://docs.python.org/2/tutorial/` (visited on 08/03/2016).

Pääkkönen, T., K. Järvelin, J. Kekäläinen, H. Keskustalo, F. Baskaya, D. Maxwell, and L. Azzopardi (2015). "Exploring Behavioral Dimensions in Session Effectiveness." English. In: *Experimental IR Meets Multilinguality, Multimodality, and Interaction*. Vol. 9283. Lecture Notes in Computer Science. Springer International Publishing, pp. 178–189.

Rabin, M. O. and D. Scott (1959). "Finite Automata and Their Decision Problems." In: *IBM Journal of Research and Development* 3.2, pp. 114–125.

Richardson, M., A. Prakash, and E. Brill (2006). "Beyond PageRank: Machine Learning for Static Ranking." In: *Proceedings of the 15th International Conference on World Wide Web*. WWW '06. Edinburgh, Scotland: ACM, pp. 707–715.

Ripley, B. (1987). *Stochastic simulation*. Wiley Series in Probability and Statistics. J. Wiley.

Robertson, S. E., S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford (1994). "Okapi at TREC-3." In: *Proceedings of the 3rd Text Retrieval Conference*. ACM, pp. 109–126.

Robertson, S. and H. Zaragoza (2009). "The Probabilistic Relevance Framework: BM25 and Beyond." In: *Foundations and Trends in Information Retrieval* 3.4, pp. 333–389.

Strohman, T., D. Metzler, H. Turtle, and W. B. Croft (2005). "Indri: A language model-based search engine for complex queries." In: *Proceedings of the International Conference on Intelligent Analysis*. Vol. 2. 6. University of Massachusetts, pp. 2–6.

Vaingast, S. (2014). "Python for Programmers." In: *Beginning Python Visualization*. Springer, pp. 55–108.

Voorhees, E. (2002). "The Philosophy of Information Retrieval Evaluation." English. In: *Evaluation of Cross-Language Information Retrieval Systems*. Ed. by C. Peters, M. Braschler, J. Gonzalo, and M. Kluck. Vol. 2406. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 355–370.

Voorhees, E. M. and D. Harman (2000). "Overview of the Sixth Text REtrieval Conference (TREC-6)." In: *Information Processing & Management* 36.1, pp. 3–35.