Final Report

KOJAK Group Finder: Scalable Group Detection via Integrated Knowledge-Based and Statistical Reasoning

Contract Number: N00173-05-1-G006
Contract Face Value: $326,744
Period of Performance: Feb-01-05 – Apr-30-06

By
University of Southern California
Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292

ISI Technical Contact: Dr. Hans Chalupsky
Tel: (310) 448-8745; Fax: (310) 823-6714
Email: **hans@isi.edu**

ISI Administrative Contact: Rita N. Egri
Post Award Contracts Manager
Tel: (310) 448-9175; Fax: (310) 822-1869
E-mail: **regri@isi.edu**

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
14-09-2006	Final Performance/Technical Report	Feb 2005 - Apr 2006

4. TITLE AND SUBTITLE
The KOJAK Group Finder:
Group Detection via Integrated Logic-based and Statistical Reasoning

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
N00173-05-1-G006

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)
Chalupsky, Hans

5d. PROJECT NUMBER
55-7035-05

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
N/A

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina Del Rey, California 90292-6601

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Naval Research Laboratory
4555 Overlook Avenue SW
Washington, DC 20375

10. SPONSOR/MONITOR'S ACRONYM(S)
NRL

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the KOJAK Group Finder which is a highly scalable, hybrid logic-based/statistical link discovery system designed to solve group detection problems. The system takes primary and secondary evidence as input and produces group hypotheses with ranked lists of group members as output. Under funding of this project, the Group Finder was developed from a research prototype into a mature software component that can be transitioned into operational environments. Among the many improvements, scalability and processing speed was improved by over a factor of 10, the software is now available in C++ and Java versions for Linux, as well as Windows platforms and now also supports Oracle databases in addition to MySQL. A Java-based GUI was added to facilitate usage by non-experts.

15. SUBJECT TERMS
KOJAK, link discovery, group detection

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Hans Chalupsky hans@isi.edu
U	U	U	UU	127	19b. TELEPHONE NUMBER *(Include area code)* (310) 448-8745

TABLE OF CONTENTS

A. KOJAK GROUP FINDER
1. INTRODUCTION....A-1
2. THE GROUP DETECTION PROBLEM....A-1
3. KOJAK GROUP FINDER SYNOPSIS....A-2
4. OBJECTIVES AND ACCOMPLISHMENTS....A-3
5. SCALABILITY....A-4
6. REFERENCES....A-6

B. SYSTEM DESIGN....B-1
C. INTERFACE REQUIREMENTS....C-1
D. HARDWARE/SOFTWARE REQUIREMENTS....D-1
D. KOJAK GROUP FINDER MANUAL....E-1

1. INTRODUCTION

Technology that could aid intelligence organizations in their efforts to detect and prevent threats to national security has become an important topic for research and development. One of the central steps in supporting such analysis is link discovery (LD) which is the process of identifying complex, multi-relational patterns that indicate potentially illegal or threat activities in large amounts of data [Senator, 2002]. Link discovery presents a variety of difficult challenges such as highly varying types of structured and unstructured data sources, that are high-volume, complex, multi-relational, noisy, incomplete, corrupted, full of unaligned aliases and contain mostly irrelevant connections.

To address these challenges we are developing a hybrid link discovery system called KOJAK that combines state-of-the-art knowledge representation and reasoning (KR&R) technology with statistical techniques from the area of data mining. Using KR&R technology allows us to represent extracted evidence at very high fidelity, build and utilize high quality and reusable ontologies and domain theories, have a natural means to represent abstraction and meta-knowledge such as the interestingness of certain relations, and leverage sophisticated reasoning algorithms to uncover implicit semantic connections. Using data or knowledge mining technology allows us to uncover hidden relationships not explicitly represented in the data or findable by logical inference, for example, entities that seem to be strongly related based on statistical properties of their communication patterns. The KOJAK Group Finder [Adibi et al. 2004, Adibi & Chalupsky 2005] is the first fully completed module of the KOJAK system which is capable of finding hidden groups and group members in large evidence databases. The KOJAK Group Finder has been successfully tested and evaluated on a variety of synthetic datasets within DoD programs and Project Insight that supported this research.

2. THE GROUP DETECTION PROBLEM

A central LD problem is the discovery of hidden organizational structure such as threat groups and their members. It can be further broken down into (1) discovering hidden members of known groups (or group extension) and (2) identifying completely unknown groups. A known group (e.g., a terrorist group such as Al Quaeda) is identified by a name and a set of known and suspected members. The problem is to discover potential additional members given evidence of communication events, business transactions, familial relationships, etc. For unknown groups neither name nor known members are available. All we know are certain suspicious individuals and their connection to certain events of interest. The main task here is to identify additional suspicious individuals and cluster them appropriately to hypothesize new real-world groups, e.g., a new money laundering ring or terrorist group.

One important problem characteristic that influenced our approach concerns the data. Evidence available to intelligence organizations can be classified into primary and secondary sources. Primary evidence is lower volume, high reliability, usually "owned" by the organization and can be searched and processed in arbitrary ways. Secondary evidence is usually not owned by the organization (e.g., might come from news articles or the Web), is higher volume, might

only be searchable for a cost in restricted ways. Our group detection approach takes these different characteristics into account by accessing secondary data sources in a very focused manner in order to minimize access cost.

3. KOJAK GROUP FINDER SYNOPSIS

The KOJAK Group Finder is a highly scalable, hybrid logic-based/statistical link discovery system designed to solve group detection problems. The system takes primary and secondary evidence as input and produces group hypotheses with ranked lists of group members as output. Primary evidence is usually lower volume, high reliability and owned by the intelligence organization, while secondary evidence (e.g., news articles on the Web, etc.) is not owned, can be very large scale, and is usually subject to querying restrictions. These scale and access restrictions heavily influenced the architecture of the Group Finder which uses a highly focused search from seed individuals to avoid having to scan all the data.

KOJAK works in four phases. First, a logic-based group seed generator analyzes the primary evidence and outputs a set of seed groups using deductive and abductive reasoning over a set of domain patterns and constraints. Second, an information-theoretic mutual information model finds likely new members for each group, producing an extended group. This is done by looking for people that are strongly connected with one or more of the seed members. To find two strongly connected entities, the Group Finder needs to aggregate known links between them and statistically contrast those with connections to potential other entities or the general population. This is done by a mutual information model which exploits data or evidence such as individuals sharing the same property (e.g., having the same address) or being involved in the same activity (e.g., sending email) to compute connection strength. Third, the mutual information model is used to rank these likely members by how strongly connected they are to the seed members. Fourth, the ranked extended group is pruned using a threshold to produce the final output.

The Group Finder has been successfully evaluated multiple times in formal DoD evaluation and has been applied to datasets with 1,000,000 entities and over 100,000,000 binary links. Table 1 summarizes performance results from evaluations in the past two years conducted on synthetic data.

	Y2	Y2.5	Y3
Entities	10,000	10,000	100,000
Links	100,000	100,000	1,000,000
F-value	0.505	0.680	0.498
Performance	1st Place	1st Place	1st Place

Table 1: Group Finder Performance

KOJAK is built on top of PowerLoom™, a knowledge representation system developed by our group over the last ten years that provides a language and environment for constructing intelligent applications [PowerLoom 2006]. PowerLoom uses a fully expressive, logic-based representation language, and a natural-deduction-style backward and forward chainer as its inference engine. PowerLoom is used to model domain-specific knowledge and rules, to find and extend seed groups based on logical reasoning, and to allow flexible mapping of KOJAK to arbitrary datasets and database schemata.

A key characteristic of KOJAK is its seeded approach where groups are built by extending seed groups or individuals provided by analysts or found by looking for certain patterns in primary evidence. Using this seeded approach has two advantages: (1) it allows a highly focused search around those seed groups instead of having to scan all of the data as in some clustering approaches, (2) it only finds groups of interest, i.e., "threat groups not country clubs", since seed groups will usually be groups of suspects an analyst is interested in. The excellent scalability of this approach was demonstrated recently by applying KOJAK to datasets with 1,000,000 entities and over 100,000,000 binary links.

4. OBJECTIVES AND ACCOMPLISHMENTS

The primary objective of this project was to mature the initial KOJAK Group Finder prototype developed during DARPA and other DoD programs into a robust, flexible and scalable application that could be transitioned into operational environments to support the mission of DHS agencies such as Customs and Border Protection. In particular, we focused on the following goals:

- Development of a robust and flexible Group Finder application
- Improved scalability
- Highly flexible data input and mapping
- Graphical user interface
- Improved hypothesis management

Major Accomplishments

Highly Scalable Group Detection: The Group Finder can detect groups of strongly connected individuals in very large datasets. It has now been successfully tested on synthetic datasets with over to 100,000,000 binary links which is about 20 times larger than the largest datasets analyzed before. Such a dataset containing about 1,600 seed groups can now be analyzed by KOJAK on a single standard desktop PC in about 4.5 hours with about 9.5 seconds/group processing time. The whole suite of IET's Y3 evaluation datasets can now be analyzed in about 1 hour which is more than a **factor of 10 speedup** compared to the version used in the evaluation in December 2004 prior to this project. The main reasons for this speedup are improved mutual information code, smarter database querying and representation, optimized analysis settings, C++ implementation and general code cleanup and refactoring.

Graphical User Interface: A **new Java-based GUI** allows selection of input data sources, configuration of analysis parameters, running the Group Finder, viewing and saving log output, viewing the resulting groups and editing and saving configuration files.

Flexible Input Formats and Mapping: the Group Finder supports a variety of different textual input formats as well as analyzing data directly on a relational database without first loading it

into KOJAK. A highly flexible data mapping interface can be used to adapt the Group Finder to different databases and schemata. Currently, the Group Finder ships with four different configurations for four different evidence database schemata. An **improved and simplified mapping language** allows adaptation to new datasets in much shorter time than before.

MySQL and **Oracle support**: The Group Finder supports MySQL as well as Oracle database servers for easy integration in production environments. It can now also run standalone without a backend database server. The MySQL support was generalized and robustified and the Oracle support is completely new.

Windows and Linux support: The Group Finder can be run on Linux and Windows (2000 and XP) platforms for easy installation in different environments. **Windows support** is new (currently only the Java version of KOJAK is supported under Windows).

C++ and Java versions: The Group Finder now comes in **C++ and Java versions** for easy integration with other link discovery software. Prior to this project, only a Lisp version was available.

5. SCALABILITY

Processing speed is fairly independent of dataset size (i.e., the number of nodes and links in a dataset) and is primarily determined by

1. the number of group hypotheses investigated in a particular dataset
2. average graph fanout, i.e., the number of links per node

KOJAK scales fairly linearly with the number of group hypotheses investigated in a particular dataset, i.e., if it takes x seconds to process one group, it takes $k \bullet x$ seconds to process k groups of similar size in the same dataset. Since the number of groups investigated depends on the number of seed groups provided by the user, processing time is really independent of dataset size. However, one would generally expect to have more seed groups of interest in larger datasets than in small ones, which is certainly the case for the synthetic IET datasets.

For a given seed group with m seed members, processing time is proportional to $m \bullet f^d$ where f is the average fanout (i.e., links per node) and d is the depth to which the seed graph is expanded (most commonly 1). The mutual information computation also adds $O(n^3)$ complexity for the number of link types n, however, since this number is usually small (< 10) we generally don't consider it.

Figure 1 shows KOJAK scalability as tested on IET's synthetic Y3 evaluation datasets. This is a suite of 25 datasets of varying size ranging from 1000 to 100,000 nodes and 250,000 to 7,500,000 binary links. The average runtime on such a dataset shown in column Avg(4018-

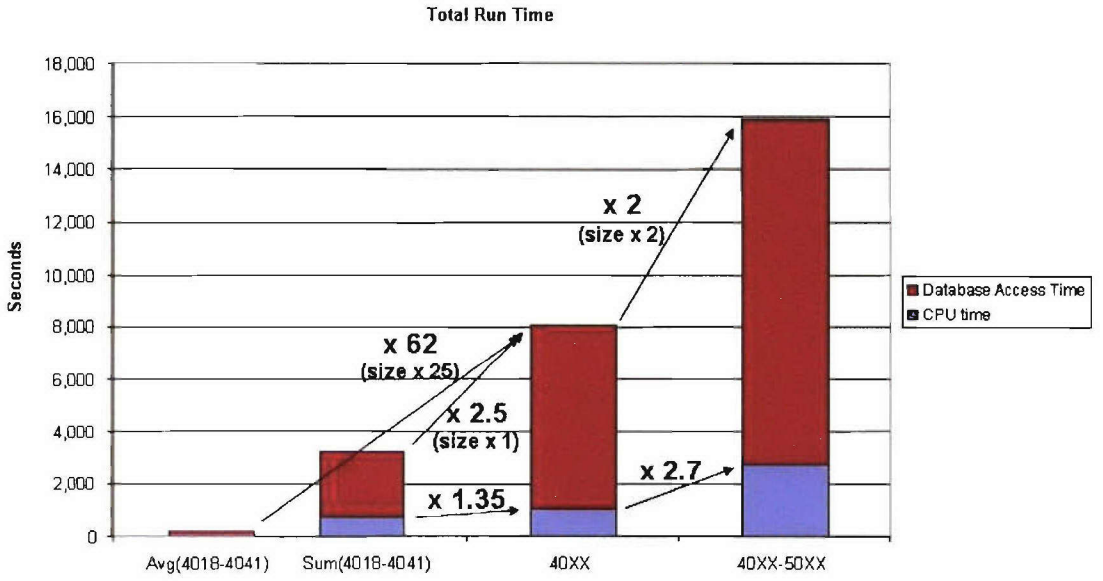

Figure 1: KOJAK Group Finder scalability

4041) is about 129 seconds. The second column Sum(4018-4041) simply sums the individual run times for easier comparison to the following columns. 40XX is a single dataset constructed by combining all 25 individual datasets into a single large database with about 500,000 entities and 75,000,000 binary links. This dataset is 25 times as big as the single average dataset and takes about 62 times as long to process, i.e., we have a slow-down of about 2.5 compared to truly linear scaling. 40XX-50XX combines the 50 wetrun and dryrun datasets into a single large database with about 1,000,000 entities and 150,000,000 links. Going from 40XX to 40XX-50XX with about twice the size we get truly linear scaling. Note that the increase in runtime with larger and larger datasets is primarily due to the increased number of groups investigated in each of them. The Avg(4018-4041) dataset has 34 seed groups while 40XX-50XX has 1679. Therefore, average processing time per group goes from 3.79 seconds/group to 9.46 seconds/group which is a slowdown of 2.5 with a size increase of about 50. These times have been measured on a standard Pentium IV desktop PC with 2GB of RAM.

KOJAK avoids ingestion by processing and aggregating directly on the database where the data resides instead of having to load it into KOJAK. If data is not available in a database, it can be loaded from a comma-separated value (CSV) file. Ingesting a CSV file into the KOJAK-internal database proceeds at a speed of about 250,000 links/minute when using a MySQL database server.

Processing speed was already discussed above. Processing time for the largest dataset tested so far with 1,000,000 entities, 150,000,000 binary links and 1679 seed groups was about 15,885 seconds (4:25 hours) on a standard Pentium IV desktop PC with 2GB of RAM. This amounts to an average processing time per seed group of 9.46 seconds.

6. REFERENCES

- J. Adibi, H. Chalupsky, E. Melz & A. Valente. The KOJAK Group Finder: Connecting the Dots via Integrated Knowledge-Based and Statistical Reasoning. IAAI 2004.
- J. Adibi, H. Chalupsky. Scalable Group Detection via a Mutual Information Model, IA 2005.
- PowerLoom. The PowerLoom knowledge representation & reasoning system. http://www.isi.edu/isd/LOOM/PowerLoom, 2006.
- T. Senator. Evidence extraction and link discovery. *DARPA Tech*, 2002.

KOJAK Group Finder
System Design

This manual describes the system design for
KOJAK Group Finder 2.3 or later.
Document 2.0

14 September 2006

The KOJAK development team
Hans Chalupsky
Jafar Adibi
Thomas A. Russ
{hans,adibi,tar}@isi.edu

Table of Contents

List of Figures **B-1**

1 Introduction **B-2**

1.1 The Group Detection Problem B-2

2 System Architecture **B-3**

2.1 System Components B-4

2.1.1 Group Seed Generation B-4

2.1.2 Mutual Information Group Expansion B-6

2.1.3 Report Generation B-6

2.2 Data Flow B-7

3 Design for Flexibility **B-8**

3.1 Mapping Databases onto the Group Finder Ontology B-8

3.1.1 Supporting Multiple Database Platforms B-9

3.1.2 Supporting Arbitrary Domains B-9

4 Graphical User Interface **B-14**

List of Figures

Figure 2.1: Group Finder Architecture 4
Figure 2.2: Group Expansion Algorithm 6
Figure 2.3: Group Finder Data Flow 7
Figure 3.1: Example Ontology Mapping 11
Figure 4.1: **Screenshot of the Analysis Tab of the KOJAK GUI** 15

1 Introduction

This document describes the KOJAK Group Finder which is a scalable, hybrid logic-based/statistical group detection system that can detect and extend groups in large evidence databases. For additional information on the KOJAK Group Finder see our IAAI-2004 and IA-2005 papers included in the 'doc' directory of the distribution or available on the web at http://www.isi.edu/~hans/publications.html. For additional documentation on the PowerLoom knowledge representation & reasoning system underlying the logic-based portion of the Group Finder please refer to http://www.isi.edu/isd/LOOM/PowerLoom/.

The Group Finder is part of a larger hybrid link discovery system called KOJAK that we are currently developing. It combines state-of-the-art knowledge representation and reasoning (KR&R) technology with statistical clustering and analysis techniques from the area of data mining. Using KR&R technology allows us to represent extracted evidence at very high fidelity, build and utilize high quality and reusable ontologies and domain theories, have a natural means to represent abstraction and meta-knowledge such as the interestingness of certain relations, and leverage sophisticated reasoning algorithms to uncover implicit semantic connections. Using data or knowledge mining technology allows us to uncover hidden relationships not explicitly represented in the data or findable by logical inference, use clustering techniques to find sets of entities that are temporally or organizationally related, or use clustering to improve efficiency by carving up a very large data space into smaller more manageable pieces.

1.1 The Group Detection Problem

A major problem in the area of link discovery is the discovery of hidden organizational structure such as groups and their members. There are of course many organizations and groups visible and detectable in real world data, but we are usually only interested in detecting certain types of groups such as organized crime rings, terrorist groups, etc. Group detection can be further broken down into (1) discovering hidden members of known groups (or group extension) and (2) identifying completely unknown groups.

A known group (e.g., a terrorist group such as the RAF) is identified by a given name and a set of known members. The problem then is to discover potential additional hidden members of such a group given evidence of communication events, business transactions, familial relationships, etc. For unknown groups neither name nor known members are available. All we know are certain suspicious individuals ("bad guys") in the database and their connection to certain events of interest. The main task here is to identify additional suspicious individuals and cluster them appropriately to hypothesize new real-world groups, e.g., a new money laundering ring. While our techniques address both questions, we believe group extension to be the more common and important problem.

The Group Finder is capable of finding hidden groups and group members in large evidence databases. Our group finding approach addresses a variety of important LD challenges, such as being able to exploit heterogeneous and structurally rich evidence, handling the connectivity curse, noise and corruption as well as the capability to scale up to very large, realistic data sets. The first version of the KOJAK Group Finder has been successfully tested and evaluated on a variety of synthetic datasets.

2 System Architecture

The KOJAK Group Finder application uses several processing steps to expand groups and rank hypotheses about group membership. The components are a combination of commercial off-the-shelf (COTS) products and custom code produced by USC/ISI.

Since the custom code is written using USC/ISI's STELLA programming language (http://www.isi.edu/isd/LOOM/Stella/), the KOJAK Group Finder is available for delivery in C++ and Java versions.

2.1 System Components

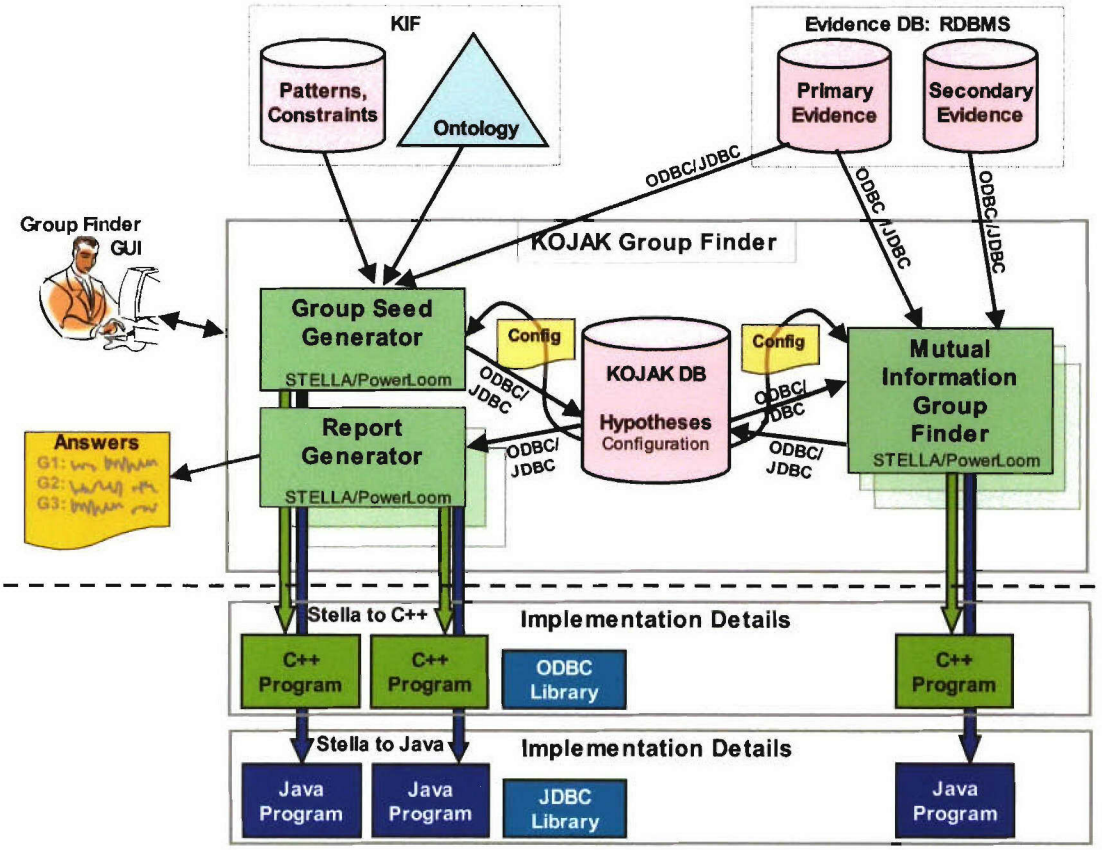

Figure 2.1: **Group Finder Architecture.** The KOJAK Group Finder has several modules coded using STELLA and PowerLoom. These modules communicate with a knowledge base using KIF and standard relational database systems using either standard ODBC or JDBC drivers. The actual delivered system is translated (via STELLA) into either C++ or Java source code for portable delivery. A graphical user interface allows users to setup configuration files and initiate processing of data.

Figure 2.1 shows the major components of the KOJAK group finder. The database systems are standard commerical database systems and are accessed using either ODBC or JDBC drivers. The systems is translated using the STELLA translators to produce either C++ or Java sources, which can then be compiled using standard tools.

The custom code sections are used for group seed generation, mutual-information based group expansion and report generation.

2.1.1 Group Seed Generation

Group seed generation begins with a set of asserted group seeds. These are then expanded using patterns and rules by KOJAK's PowerLoom KR&R engine. These seeds form the basis for subsequent group expansion using mutual information. By beginning with a set of known group members, the KOJAK Group Finder is able to concentrate its search in the data to likely relevant leads rather than needing to process the entire data set. This approach gives the following benefits:

- Focus on relevant groups. The Group Finder identifies threat groups or other groups of interest instead of finding country clubs.
- Reduction in search space. The reduced amount of data that needs to be processed results in faster answers, since only potentially relevant link information is examined.
- Clutter resistance. The focus on relevance means that the algorithm will scale well with an increase in irrelevant links, since they never get examined.
- Reduced costs. In a situation where secondary data comes into play and has a higher "cost", the focus on only relevant link information minimizes the need to consult the secondary data sources, thus reducing costs.

2.1.2 Mutual Information Group Expansion

The process of group expansion is shown in Figure 2.2.

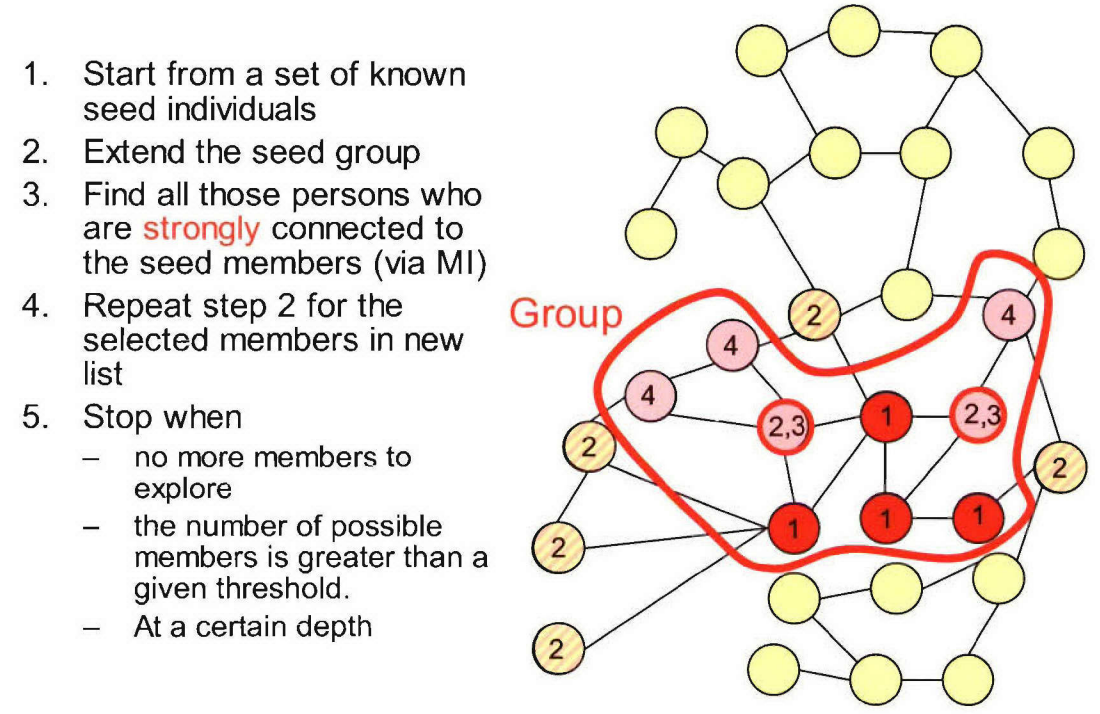

Figure 2.2: **Group Expansion Algorithm**

The group expansion component does an expansion of group membership by starting from a set of known members and expanding the membership based on link information. Individuals or entities linked to the group are then evaluated using a mutual information model to exclude links that are incidental contact from those that are more tightly connected to the group. This process enables us to concentrate on connections of group leaders to their lieutenants while disregarding phone calls to the pizza delivery man.

2.1.3 Report Generation

The report generator takes the hypotheses stored in the KOJAK hypothesis store and produces reports. The reports list group membership relations developed by the software. The relations can optionally include a score indicating the strength of the connection to the group.

Several report formats are supported, including comma-separated value files, a special IET reporting format and database tables.

2.2 Data Flow

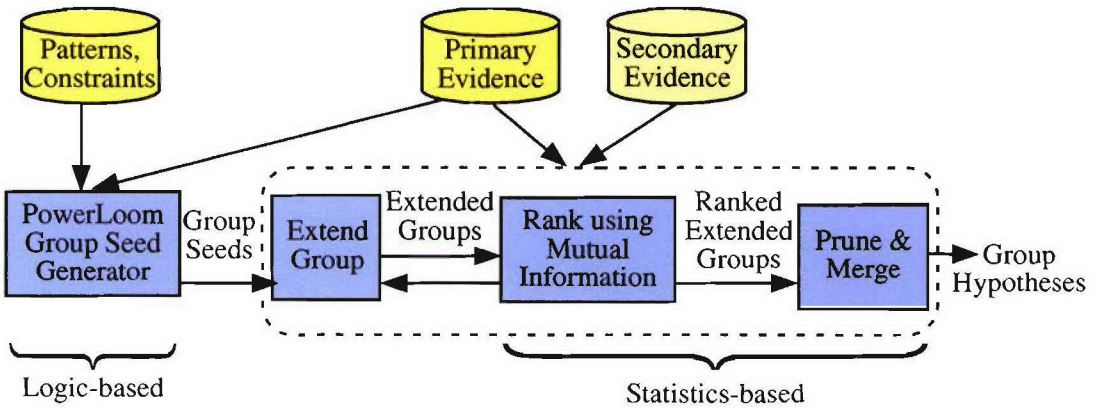

Figure 2.3: **Group Finder Data Flow.** The processing begins with the generation of group seeds using seed data and logic-based rules. The seeds are then expanded using mutual information to rank member candidates and control the expansion process. Finally the resulting groups are merged and pruned to eliminate redundant group identifications.

Figure 2.3 shows the basic processing steps and flow of data in the system:

1. Logic-based rules and patterns are used to generate and expand group seeds. This uses general rules about group membership and behavior. For example, taking part in a threat action that is linked to a particular group indicates that one belongs to that group.
2. The seed group is extended by following links from known seed members.
3. The new members are evaluated using a mutual information model and likely candidates are added to the pool of seeds. This process feeds back into step 2 and repeats until the group membership stablizes. The stability criteria can be adjusted by the user.
4. Finally, the group hypotheses are pruned and merged to eliminate redundant groups (i.e., where a single group emerged from two different starting seeds).

3 Design for Flexibility

Developing a data analysis tool such as the Group Finder that can be easily and successfully applied to arbitrary real-world datasets is a very challenging task, given that data can come in so many different formats and with so many statistical properties. To address this challenge, the Group Finder needs a very flexible and highly configurable interface.

The required flexibility is achieved via a variety of different mechanisms:

1. Input Data Formats: the Group Finder is designed to read data in a variety of different formats such as comma-separated value (CSV) files, IET evidence files, PowerLoom files or directly from a relational database such as MySQL or Oracle via ODBC or JDBC. Details about these various input formats can be found in the Interface Requirements manual.
2. Data Mapping: real data comes in many different formats and conceptualizations (or ontologies). To allow analysis of arbitrary data by the Group Finder it either has to be translated prior to analysis or appropriately mapped. Translation prior to analysis is only cost-effective for fairly small and static datasets. For larger datasets that change over time it is advantagous to access data directly from an existing database and translate it on-the-fly via mapping specifications. This mechanism is described in more detail below.
3. Configuration: a single analysis run depends on many different pieces of information from the location of input data files, their formats, a variety of analysis parameters to the location and formats of outputs. All this information can be captured in a configuration file and further modified via command-line flags or specific command parameters (useful for multiple runs with minor variations in parameters). Details of these mechanisms can be found in the Interface Requirements manual.
4. Scripting: even a relatively simple process such as group detection has many different stages, dependencies, variations depending on a particular dataset or application, etc. Capturing all these options in a control API is difficult and would make the necessary configuration very complex. Instead, the Group Finder supports flexible scripting of a large set of built-in commands for loading and importing data, loading of ontology, schema and mapping information, as well as for carrying out various stages of group detection and outputting of results. The syntax of scripts and the available commands can be found in the Interface Requirements manual.
5. Integration with 3rd-party Components: group detection is only one part of the intelligence information processing chain, and connectivity to components developed by other contractors or vendors is important. Such connection allows others to use group detection results as their input or can provide the Group Finder with relevant inputs (e.g., seed groups). A primary mechanism for this connectivity is the use of a relational database as a kind of blackboard where inputs can be set up and results can be posted. Being able to deliver the Group Finder in multiple languages such as C++ and Java also facilitates direct integration with other applications at a code level.

3.1 Mapping Databases onto the Group Finder Ontology

The first version of the KOJAK Group Finder was strongly oriented towards batch-processing of specific simulated datasets generated by Information Extraction and

Transport, Inc. (IET). In particular, it made strong assumptions about the domain, the types of entities and links expected in the data, their representation in the ontology and the evidence database and the one-shot processing and reporting style required by government evaluations. For flexible application to real-world datasets in arbitrary domains, however, all these restrictions need to be eliminated.

One relatively simple aspect of this is flexibility with input data formats, and the Group Finder can now accept a variety of data formats such as CSV files, various evidence formats or data directly coming from relational databases (details can be found in the Interface Requirements manual).

3.1.1 Supporting Multiple Database Platforms

Another conceptually simple but technically somewhat involved issue is how to support multiple database platforms such as Oracle, Sybase, SQL Server, MySQL, etc. The current version of the Group Finder supports MySQL and Oracle and uses SQL over ODBC and JDBC to access and query the database. While SQL and ODBC are (industry) standards, it is still very difficult to build database-neutral applications that will work across a variety of different database implementations. In particular, schema definition commands vary widely between different databases, and there are many other underlying assumptions that make it very easy to write non-portable code. For example, in Oracle identifiers such as table names need to be at most 30 characters long, while MySQL supports much longer names. Oracle doesn't support multi-row inserts in a single command, while MySQL does, etc.

Our approach to solving this problem has two parts:

1. A code abstraction layer called SDBC (for STELLA Database Connectivity) is used by the Group Finder to send queries, insert values, handle prepared statements, iterate over result sets, etc. This layer abstracts from the actual underlying connectivity protocol used, i.e., application code does not have to know whether it is communicating via ODBC, JDBC or a special API such as the MySQL C API. Since STELLA code can be translated into different native languages such as Java, C++ or Lisp, this level also abstracts from the differences of the native mechanisms used by those languages to call ODBC or some other protocol.
2. Instead of constructing SQL commands directly in code which would make it very difficult to adapt the Group Finder to a different database system, all Group Finder code issuing SQL commands instantiates named command templates. These templates can be read in from external files (or bootstrapped from the database) and allow adaptation to different database platforms without having to change any code.

3.1.2 Supporting Arbitrary Domains

The most difficult problem is how to map an arbitrary domain on the world of groups, entities, memberships and links understood by the Group Finder. The simplest way to do this is to generate an appropriate CSV input file from the source data that the Group Finder understands. How this can be done is described in the Interface Requirements and User Manuals. However, as mentioned above, this approach is only feasible for relatively small and static datasets. For large and dynamically changing datasets sitting on some corporate database, it is much better to access data directly from the database and to translate it on the fly by use of some mapping specifications. Similar to supporting multiple different

database systems, it must be possible to support multiple different mappings without having to modify any code.

To facilitate mapping to arbitrary domains, the Group Finder uses a very simple core representation of the group detection domain. Each domain is represented by a graph where nodes describe entities such as people, groups or events and links describe relationships between those entities. There are a very small number of predefined node and link types with specific meaning that are defined in a very small group detection ontology. The core concepts and relationships in this ontology are depicted by the triangle in the center of the mapping example in Figure 3.1. The `Group` class on the upper right is further specialized into `KnownGroup` and `UnknownGroup`. The logic-based component uses information about groups, known members and participation in events, etc. to infer additional seed members. Besides these predefined relationship types, there can be an arbitrary number of other link types between entities which are used by the mutual information module to infer connection strength. For example, such links might represent transactions such as phone calls, emails, telecons, goods shipped, etc.

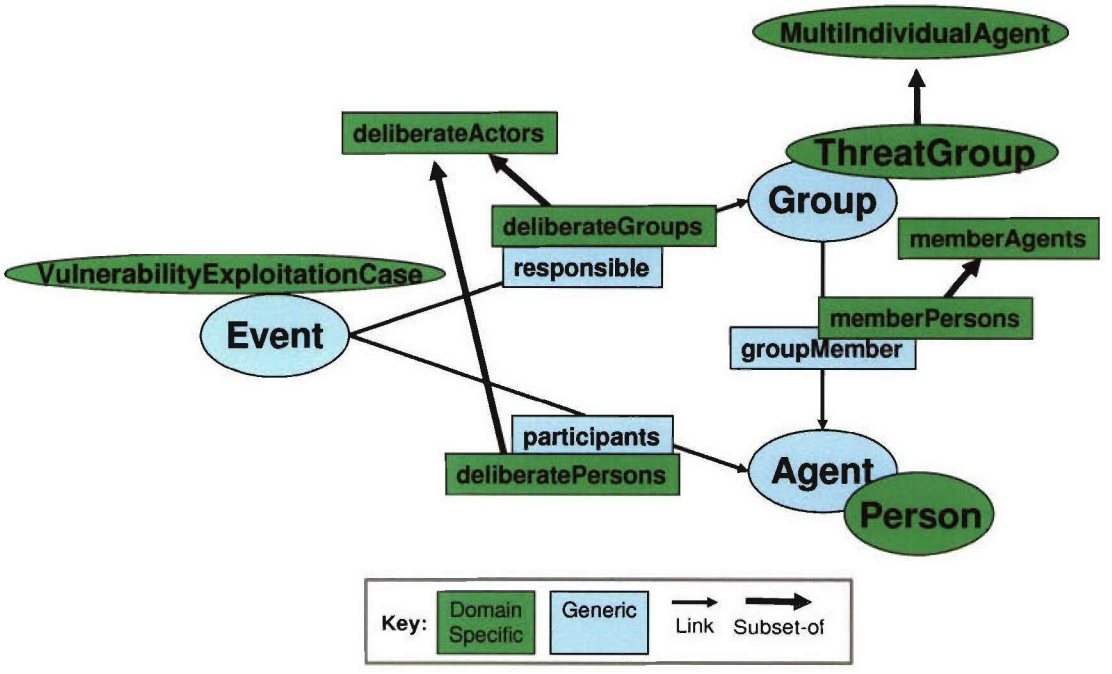

Figure 3.1: **Example Ontology Mapping**. Example mapping for the logic-based seed generator that maps the simple and generic Group Finder ontology onto the more complex ontology used by the IET simulator. While the mapping in this example is fairly simple, more complex mappings involving logical inference are also possible.

To map a new domain or database onto this internal representation three things need to be done:

1. The domain has to be modeled as a graph (at least conceptually). In many cases it is clear what to model as a node (e.g., a person) and what as a link (e.g., a family relationship or a financial transaction). Sometimes it can be difficult and requires some analysis, for example, should the shipment of a particular type of good become a specific link type? This is a manual and mainly conceptual step that influences how the next two mapping steps are carried out.
2. Certain link and node types of the graph representation have to be mapped onto the built-in types and relations understood by the Group Finder. For example, senders and receivers in a container data set might become `Agents` in the group detection ontology. What this part of a mapping might look like for a specific domain is shown in Figure 3.1.

In the system, this part of the mapping is expressed via a set of PowerLoom definitions and assertions.

3. The set of transaction link types that should be considered by the mutual information component need to be specified together with how to query for them for a specific pair or set of individuals. In the system this part of the mapping is expressed as a set of PowerLoom definitions together with query templates that map the particular PowerLoom relations onto database queries on the underlying evidence database(s).

As already indicated above, the PowerLoom knowledge representation and reasoning system underlying the KOJAK Group Finder plays a central part in these mapping specifications. Using PowerLoom allows us to define arbitrarily complex mappings that might involve various forms of logical inference. For example, here is how part of the mapping depicted in Figure 3.1 can be defined in PowerLoom. Note that the definition of the `participant` relation involves a simple rule-based inference that constrains the type of event the person participated in.

```
(ASSERT (SYNONYM Person GROUPS/Agent))
(ASSERT (SYNONYM ThreatGroup GROUPS/Group))
(ASSERT (SYNONYM VulnerabilityExploitationCase GROUPS/Event))
(ASSERT (SYNONYM memberAgents GROUPS/groupMember))

(DEFRELATION participatingAgent ((?c VulnerabilityExploitationCase)
                                 (?p Person))
  :<= (AND (VulnerabilityExploitationCase ?c)
           (deliberateActors ?c ?p)))

(ASSERT (SYNONYM participatingAgent GROUPS/participant))
```

PowerLoom also has a sophisticated database interface that makes it possible to map logical relations onto database tables, projections, joins or arbitrary queries. This mechanism allows for an efficient division of labor between the more expressive but also expensive logic-based inference system and the database system that can efficiently query very large numbers of tuples. For example, the definitions below define a few projections from a `TwoWayCommunicationEvent` table to some specialized PowerLoom relations as well as a relation that is defined via an SQL query template.

```
(DEFTABLE EDB-TwoWayCommunicationEvent
          edb "TwoWayCommunicationEvent"
          ((UID STRING) (context INTEGER)))

(DEFTABLE EDB-TwoWayCommunicationEvent_iteInitiator
          edb "TwoWayCommunicationEvent"
          ((UID STRING) (iteInitiator STRING) (context INTEGER)))

(DEFTABLE EDB-TwoWayCommunicationEvent_iteRespondent
          edb "TwoWayCommunicationEvent"
          ((UID STRING) (iteRespondent STRING) (context INTEGER)))

(DEFQUERY EDB-TwoWayCommunicationEvent_extendedGraph
    ((?seeds STRING) (?event STRING) (?caller STRING)
```

```
    (?callee STRING) (?context INTEGER))
  :query-pattern
  (RDBMS/SQL-QUERY edb
    "SELECT DISTINCT '', UID, iteInitiator, iteRespondent, context
FROM TwoWayCommunicationEvent
WHERE #${UID='?event'}
AND ((iteInitiator IN (#${'?seeds[0,,]'}#${,'?seeds[1,:,]'}))
     OR (iteRespondent IN (#${'?seeds[0,,]'}#${,'?seeds[1,:,]'})))
AND #${iteInitiator='?caller'}
AND #${iteRespondent='?callee'}
AND #${context='?context'}"))
```

Query templates such as the above are extremely powerful, since they allow smooth and efficient interaction between the PowerLoom inference engine and the underlying database. However, depending on the schema of the backend database the definition of appropriate query templates can be somewhat complex and daunting. Nevertheless, by providing some generic examples we can handle some common cases as well as provide good starting points for specific adaptations.

4 Graphical User Interface

To allow evaluation and experimentation with the KOJAK Group Finder by users other than ourselves (e.g. government SMEs or intelligence analysts) KOJAK comes with a simple graphical user interface that allows an evaluator to easily control the execution of the Group Finder and visualize results.

There are a couple of different choices possible with respect to the underlying GUI technology. For our PowerLoom KR&R system we have developed two different types of interfaces in the past. One is the Ontosaurus browser which is a dynamic HTML-based browser interface that allows a user to conveniently browse a PowerLoom knowledge base from an arbitrary Web browser. Another is our PowerLoom GUI which is a Swing-based Java application that can be run standalone but also from a browser via Java Webstart. For the KOJAK GUI we decided to use Java, primarily, because it will eventually make it easier to integrate 3rd-party visualization software such as Prefuse http://prefuse.sourceforge.net/. The GUI is Swing-based and was written using the Java NetBeans IDE. The connection between the GUI and the KOJAK backend code is via direct Java function calls (the two software components are simply separate Jar files). The GUI is currently only available for the Java version of KOJAK.

The primary function of the GUI is to display/generate/edit an appropriate configuration file to run KOJAK on some dataset. The following core operations are supported which allow a user to run the Group Finder end-to-end:

- selection of input data sets and sources
- specification of seed members for one or more groups,
- setting of algorithm parameters
- setup of simple run-scripts
- executing a run
- displaying results
- exporting results in different formats

A screenshot of the analysis tab of the GUI is shown in Figure 4.1.

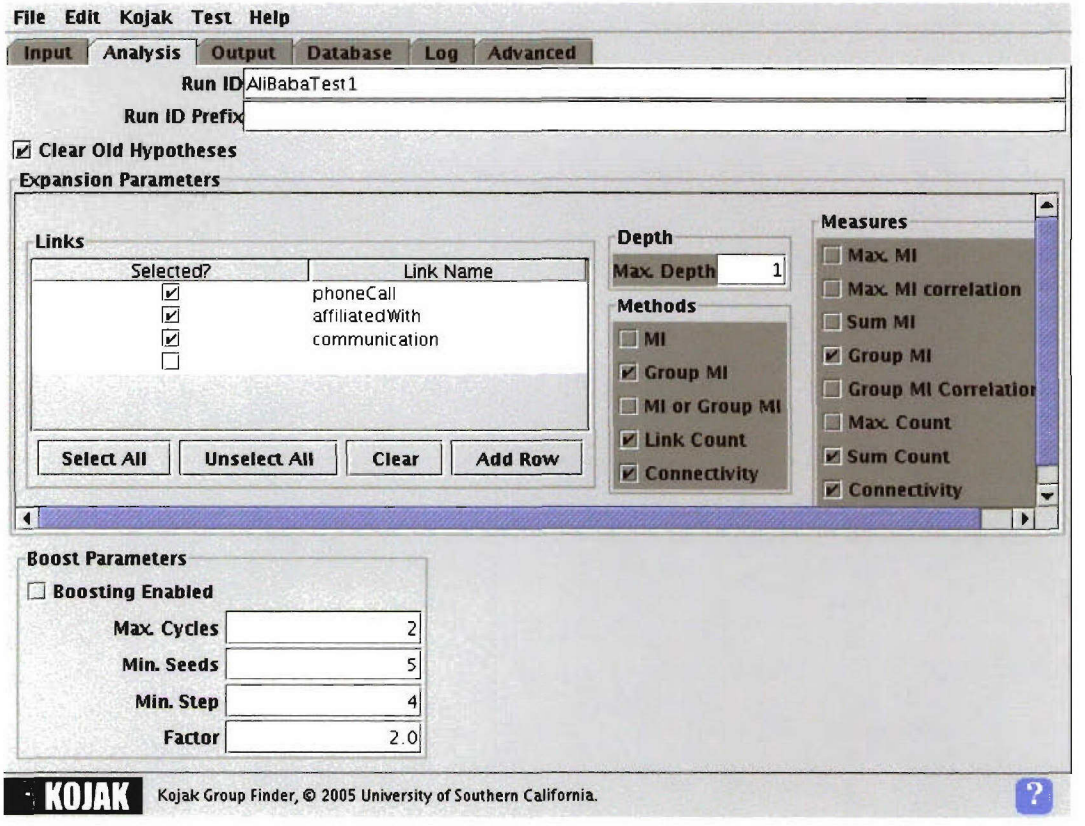

Figure 4.1: **Screenshot of the Analysis Tab of the KOJAK GUI**

Interface development is a very open-ended endevor and needs to be properly circumscribed. Our interface provides all the necessary functionality to access the relevant Group Finder features and allow a user to run it, but it does not come with all the "bells and whistles" one might expect from a commerical-grade GUI. Such GUI development can be performed further down the road once the basic functionality has been developed and evaluated. At such a point, integration with standard interfaces used by analysts such as, for example, I2's Analyst's Notebook will also be possible.

The following operations are also desirable and might be supported in some future version of KOJAK:

- exclusion of irrelevant entities
- merging of entities into super-entities
- specification of known alias information
- selection of temporal regions
- simple visualization of results

KOJAK Group Finder
Interface Requirements

This manual describes the interface requirements for
KOJAK Group Finder 2.3 or later.
Document 2.0

14 September 2006

The KOJAK development team
Hans Chalupsky
Jafar Adibi
Thomas A. Russ
{hans,adibi,tar}@isi.edu

Table of Contents

1 Input Data C-1
1.1 Types of Data C-1
1.1.1 Group Seeds C-1
1.1.2 Link Data C-2
1.2 Data Access Classes C-3
1.2.1 Primary Data C-3
1.2.2 Secondary Data C-3
1.2.3 Undifferentiated Data C-3
1.3 Data Formats C-3
1.3.1 Comma-Separated Value Format C-4
1.3.2 IET Format C-4
1.3.3 PowerLoom Format C-4
1.3.4 Relational Databases C-5

2 Configuration C-6
2.1 Configuration File C-6
2.1.1 General Options C-7
2.1.2 Input Specifications C-8
2.1.3 Script Specifications C-9
2.1.4 Analysis Specifications C-10
2.1.5 Output Specifications C-12
2.2 Command-Line Options C-14

1 Input Data

The Group Finder needs two separate types of input data, *group seeds* to identify the starting points for group expansion and *link data* which is analyzed to expand the groups.

The data can also be subdivided into different *access classes*, which is based on ease of availability or cost of access. These classes are called *primary data* for data that can be easily and cheaply accessed or *secondary data* for data that is harder to access or more expensive.

The data can also presented in one of several formats.

These options are described in the following sections.

1.1 Types of Data

Note that there is a fundamental assumption underlying all of the data, namely that **different entities have unique names**. One consquence of this is that if data is being transferred from a system that uses catch-all categories such as *unknown* or *other* that refer in general to several unnamed entities, the data must be prepared by generating unique ids for all such unamed entities. This can be accomplished by adding a numeric suffix to any such categories so as to create unique individual names.

1.1.1 Group Seeds

Group seed data describes one or more known or unknown groups with one or more known or hypothesized members for each seed. As mentioned previously, the Group Finder performs seeded group extension, therefore, it must have at least one seed group to start with. The rationale behind this seeded approach is (1) that it allows for a focused search through a potentially huge data space, and (2) that the seeds - if chosen properly - will automatically lead to the detection of groups of interest (e.g., threat groups). This is very important, since in general, every large dataset will contain large numbers of groups that are benign and uninteresting. Since KOJAK's extended groups are formed around seeds, they have a much higher likelihood of being of the same kind as the seed group. With respect to scale, this approach has also already paid off and allowed us to successfully analyze datasets with close to 10,000,000 links. Having said that, there are situations where seed information is not available and we are currently developing methods to generate such seeds automatically if necessary.

Group seed data is generally fairly small. In the simplest case it involves the specification of one or more groups with one or more known members for each of them. For example, the following specifies a seed group in comma-separated-value (CSV) syntax (which is described in more detail below):

```
KnownGroup, SeedGroup1
groupMember, SeedGroup1, MemberA
groupMember, SeedGroup1, MemberB
groupMember, SeedGroup1, MemberC
groupMember, SeedGroup1, MemberD
```

The first line specifies `SeedGroup1` as a group of type `KnownGroup`, and the following lines specify 4 known (or suspected) members of the group. The order of these lines is insignificant. The terms `KnownGroup` and `groupMember` are reserved words interpreted by

the Group Finder. It is possible to use different terms (i.e., a different "ontology"), but then appropriate mapping specifications are needed.

The Group Finder distinguishes between `KnownGroup`'s and `UnknownGroup`'s. The difference is that known groups are assumed to be groups with known identity (e.g., the RAF), and therefore such known groups will never be merged (even though their members might overlap). Unknown groups are groups whose identity we do not know. Unknown groups might turn out to be (part of) a known group or another unknown group. For unknown groups, the Group Finder will attempt to merge them with each other or some known group in case they are similar enough. Note that knowing a group's identity does not mean that we know its members. So, extension of known groups is still a very useful thing to do.

Depending on the domain, it is possible to model some aspects of it and infer additional seed groups or members from other data or link information. This is one of the functions of the logic module which uses a domain ontology and rules to do that. These rules, however, are highly dependent on a particular application domain (such as IET's simulated data generated within the EAGLE program), and will need to be handcrafted specially for each such domain. For example, the file '`kbs/iet-y3-seed-constraints.plm`' encodes such a set of rules and constraints for IET's simulated data. The released version of the Group Finder is generic and domain independent, however, and there is no set of generic, domain-independent seed generation rules. For a typical generic application, it is assumed that seed information will be given explicitly as described above. If a domain is more complex and has additional data that can be exploited, this domain information could be modeled similar to '`kbs/iet-y3-seed-constraints.plm`' to allow the automatic derivation of additional seed information.

1.1.2 Link Data

Link data describes known connections of different types between the entities (individuals) in a dataset. These entities should overlap with the ones provided in the seed groups, but, in general, there will be lots and lots of other irrelevant entities in this data. The Group Finder performs its main work on this link data to find the few (hopefully relevant) individuals that are strongly connected to the seeds. It does so by statistically analyzing and comparing the links from seed individuals to others in the data. In order for this analysis to be significant, there needs to be a sufficient amount of link data - the more, the better. A good example is communication data such as emails or phone calls, where each email or phone call can be viewed as a separate link between two entities. Therefore, there might be hundreds or thousands of links between a pair of entities for each link type. Financial transactions or other transactional data are good candidates also. Here is an example fragment of a link database in CSV syntax:

```
...
phoneCall,In9999,In15132
phoneCall,In9999,In24521
phoneCall,In9999,In27306
telecon,In10005,In11120,In15968,In19067,In2284,In32055,In6166
telecon,In10009,In18419,In28735,In33962,In35311,In36690,In4777,In5866
telecon,In10013,In1325,In24781,In33373,In38164,In38830,In4194,In8312
telecon,In10013,In1325,In24781,In33373,In38164,In38830,In4194,In8312
...
```

The first element in a row specifies the type of link such as `phoneCall` or `telecon`. These link types are not predefined and can be chosen arbitrarily. However, they should represent distinctions that are meaningful. Depending on how the Group Finder was configured, they might have to correspond to relations defined in the underlying ontology.

The remaining arguments are the set of individuals that participated in the particular link, transaction or event. For example, all participants of a telecon can be listed and are then assumed to be pairwise connected by a link of this type. Rows might be duplicated (such as the last one) indicating multiple events or links with the same participants. Again, the order of these rows is insignificant.

The Group Finder can function with data that contains just a single link type, but the mutual information model takes different link types into account, and the more such link-type-differentiated information is available, the better.

Example datasets derived from one of IET's simulated datasets are shipped in the '`data/example*`' subdirectories. They are provided in a comma-separated value (CSV) format that might be the easiest to imitate when applying the Group Finder to new domains. The data files have some more comments describing their structure. They contain about 10,000 entities and 250,000 links connecting them.

The Group Finder can read input data in a variety of formats described in the following sections.

1.2 Data Access Classes

Data can be differentiated based on a general notion of cost of access. If this is not a concern for a particular application environment, undifferentiated data may be used.

1.2.1 Primary Data

Primary Data are easy and cheap to access. They can be used freely for analysis and permit extremely general queries. Link data can be queried without a need to specify any of the participants.

1.2.2 Secondary Data

Seoncdary Data are more difficult or expensive to access. This is taken into consideration by the Group Finder code, which will try to minimize the amount of secondary data used in the analysis. It is also assumed that querying of such data is more limited, in that fully open-ended queries are not permitted. For example, rather than retrieving all link information, the queries might require that at least one of the participants in the link be specified in the query.

1.2.3 Undifferentiated Data

Undifferentiated data describes data which do not have differential access costs. The entire dataset is available for all analysis and no special processing is required.

1.3 Data Formats

The Group Finder supports several data formats, including comma-separated value files, a special IET data format, PowerLoom format and MySQL databases. Of these, the most easily used and adapted is the Comman-separated value format.

1.3.1 Comma-Separated Value Format

The comma-separated value (or CSV) format is probably the easiest to use for applying the Group Finder to a new domain. Each CSV file should have a '`.csv`' extension (or file type) so the Group Finder will use the appropriate data input method. CSV files have the following syntax:

1. Empty lines and lines starting with a # character will be ignored
2. All other content lines should have the following syntax:

   `<relation>, <arg1> {, <arg>}*`

   Whitespace following a comma up to the next non-whitespace character will be ignored. Relations and arguments can contain any arbitrary charactes. ,'s need to be escaped with a \ character.

A content line with one argument will be interpreted as a type specification, for example:

```
KnownGroup, Group1
```

The relations `groupMember` and `memberAgents` will be interpreted as a binary membership assertion between a group and a group member. For example:

```
groupMember, Group1, Member1
```

All other relations will be interpreted as event-style links. For example,

```
telecon, pA, pB, pC
```

will create a new link (or event) object of type `telecon` and attach `pA`, `pB` and `pC` as the link arguments (or event participants). The exact internal representation will differ dependent on whether this link is loaded into PowerLoom or into the KOJAK EDB. However, the treatment and interpretation will be the same.

1.3.2 IET Format

The IET format was developed by Information Extraction and Transport, Inc. to represent synthetic data. It is a Lisp-style format that uses CycL as the representation language. For example:

```
(isa Gr-16511 ThreatGroup)
(exploitsVulnerabilities Gr-16511 Mo-18843)
(memberAgents Gr-16511 In-17687)
(memberAgents Gr-16511 In-11314)
```

This format is not intended to be a primary input format and only supported to allow the Group Finder to be applied to synthetic data generated by the IET simulators (there are a two different variants for Y2 and Y3 datasets, both are supported). IET format files need to use the file extension `.iet` in order to be properly recognized and handled by the Group Finder.

1.3.3 PowerLoom Format

PowerLoom is the representation and reasoning engine underlying the KOJAK Group Finder. Consequently, all legal PowerLoom input can appear in PowerLoom data files. PowerLoom uses KIF as its input language which is a Lisp-style syntax for predicate logic. For example:

```
(ASSERT (KnownGroup UID-Group-34988))
(ASSERT (groupMember UID-Group-34988 UID-Indvdl-10641))
(ASSERT (groupMember UID-Group-34988 UID-Indvdl-14179))
(ASSERT (groupMember UID-Group-34988 UID-Indvdl-20265))
(ASSERT (groupMember UID-Group-34988 UID-Indvdl-2206))
```

PowerLoom files need to use the file extension `.plm` in order to be properly recognized and handled by the Group Finder. More information about the PowerLoom system and its input syntax can be found at http://www.isi.edu/isd/LOOM/PowerLoom/.

1.3.4 Relational Databases

Finally, data can come directly from a relational database such as MySQL or Oracle. No specific schema is assumed, however, for each schema an appropriate mapping specification/schema/ontology needs to exist that maps the database schema onto the Group Finder ontology. The Group Finder ships with various predefined mappings that support IET Y2, IET Y3 as well as KOJAK evidence databases (EDBs). CSV files can be imported into the KOJAK EDB which is useful for very large link databases or to support multiple analysis runs. Example mapping specifications can be found in the 'kbs' directory, for example, 'kbs/ali-baba-edb-schema.plm', 'kbs/iet-y2-edb-schema.plm', 'kbs/iet-y3-edb-schema.plm' and kbs/kojak-edb-schema.plm. Writing such mapping specifications is somewhat complex and requires some knowledge of the underlying PowerLoom system. It is described in more detail in ⟨undefined⟩ [Advanced Configuration], page ⟨undefined⟩.

2 Configuration

Developing a data analysis tool such as the Group Finder that can be easily and successfully applied to arbitrary real-world datasets is a very challenging task, given that data can come in so many different formats and with so many statistical properties. To address this challenge (at least partially), the Group Finder has a very flexible and highly configurable interface. One might almost say that the Group Finder is configurable to a fault, since there are so many different ways of mapping data, configuring it and running it. However, the generic and example configurations shipping with the distribution should cover a wide variety of situations and make easy departure points when applying the Group Finder to some new dataset.

The Group Finder uses the following configuration mechanisms:

1. A configuration file allows the specification of a large number of parameter such as datasets, connection information, analysis parameters, etc. Many of the available KOJAK commands take these parameters as inputs or as defaults.
2. A set of command line flags that can be used to specify and override some of the configuration parameters specified in the configuration file. This is useful for multiple runs over the same data with slightly different parameter settings, etc.
3. A run script that can execute an arbitrary sequence of KOJAK commands for loading/importing data, ontology and schema information, generating group seeds, extending them, thresholding, reporting, etc.
4. A variety of load scripts that control how and what data is loaded, which ontology and schema information is loaded, etc.
5. Ontology and knowledge base files that can leverage the full power of the PowerLoom knowledge representation and reasoning system to represent specifics of a particular domain, map to a particular external database schema, etc.

2.1 Configuration File

By default, the Group Finder looks for the file '`configuration.dat`' in the KOJAK software's installation directory to load its configuration information. Alternatively, the `-c` command-line option can be used to specify a different configuration file. It is an error if no configuration file can be found and none was specified on the command line.

There are a set of configuration files available in the '`config`' directory. By default the Group Finder uses the '`config/example1.dat`' configuration which runs the Group Finder on one of the example datasets that ship with the release. This file and its associated load and run scripts should be a good departure point for generating a configuration for some new dataset.

Configuration information is supplied in a Java-style property file with lines of the following form:

```
<parameterName>  =<parameterValue>
<parameterName> +=<parameterValue>
```

Empty lines or lines starting with a # character will be ignored. Whitespace leading up to the = character will be ignored, whitespace following it will become part of the parameter value. The += syntax allows the specification of list-valued parameters, for example, to

assign a set of input files to the `Data` parameter. Both parameter names and values are treated case-sensitively.

The configuration file can contain arbitrary user-defined parameters which can be accessed in run and load scripts via the functions `get-parameter`, `set-parameter`, and `add-parameter-value`. There are a multitude of built-in parameters that control the behavior of the Group Finder which are explained further below.

Various built-in parameters take filenames as arguments. These filenames can be absolute or relative physical pathnames (using syntax appropriate for the underlying operating system), or, they can be *logical pathnames* that are platform neutral and that can be translated automatically by KOJAK into the appopriate physical form. Logical pathname syntax is derived from Common-Lisp. Each logical pathname starts with a logical host (terminated by a colon) followed by a set of directory components (separated by ;) and ending in a filename with an optional extension. KOJAK logical pathnames need to be of the following form:

```
KJ:{<dirname>;}*[<basename>][.<extension>]
```

For example,

```
KJ:scripts;ali-baba-load-ontology.plm
```

which would translate into

```
C:\Program Files\kojak-2.2.0\scripts\ali-baba-load-ontology.plm
```

if the value of `KojakRootDirectory` (see below) is defined as '`C:\Program Files\kojak-2.2.0\`'. This means the logical host `KJ` is simply replaced by the value of `KojakRootDirectory` and the logical pathname separator ; is substituted with the pathname separator appropriate for the underlying OS.

2.1.1 General Options

`KojakRootDirectory` [Parameter]

This should be an absolute physical path pointing to the KOJAK directory (including a final directory delimiter). A relative path such as '`./`' will work also but requires that the Group Finder is run from the KOJAK directory (since the `run-kojak` scripts always change to the KOJAK directory relative pathnames are fine unless the KOJAK executables are invoked manually somehow). Logical pathnames starting with the `KJ` logical host will use this directory as their root.

`KojakDB` [Parameter]

The internal database used by KOJAK to store hypotheses, intermediate results, data and configuration information. Must be either a simple ODBC DSN or an ODBC or JDBC connection string (similar to `Data` specifications which see). If it is a DSN all necessary connection information must be specified in '`.odbc.ini`' (see ⟨undefined⟩ [.odbc.ini File], page ⟨undefined⟩). For example:

```
KojakDB =KOJAK
```

If it is an ODBC connection string, everything necessary can be specified, but partial connection strings are also possible in which case missing information is filled in from '`.odbc.ini`'. For example:

```
KojakDB =DSN=KOJAK;DB=kojak;SERVER=blackcat;UID=scott;
KojakDB =DSN=KOJAK;DB=kojak;
KojakDB =DSN=KOJAK;USER=KOJAK;DB=blackcat.isi.edu/oracledb;
```

If it is a JDBC connection string, all connection information must be specified. JDBC connection string also varies for different database systems and JDBC drivers. Here are two examples for the Oracle and MySQL drivers currently supported by KOJAK:

```
KojakDB =jdbc:mysql://blackcat:3306/kojak?user=scott&password=secret
KojakDB =jdbc:oracle:thin:KOJAK/secret@blackcat.isi.edu:1521:oracledb
```

Note that for Oracle databases the KOJAK schema name is the USER name while for MySQL it is the name of the database (DB). `KojakDB` can be set or overridden the command line options `-k` and `--kojak-db`.

`LogLevel` [Parameter]

Controls the amount of log output generated during an analysis run. Needs to be one of `none/low/medium/high`. This can also be specified via the command line options `-l` and `--log-level`.

`DBUser` [Parameter]

Default user name to use for connection information if a database is simply supplied as a DSN but not a connection string. If this is not provided, user information will need to be specified for the particular DSN in `.odbc.ini`, it will not be inherited from the [Default] data source.

`DBPassword` [Parameter]

Default password to use for connection information if a database is simply supplied as a DSN but not a connection string. If this is not provided, password information will need to be specified for the particular DSN in `.odbc.ini`, it will not be inherited from the [Default] data source.

2.1.2 Input Specifications

`Data` [Parameter]

`Data` can be used to specify data sources to be analyzed. Sources can either be files in various formats or evidence databases accessible via ODBC or JDBC. Currently supported file formats are comma-separated value files, IET report-format files or PowerLoom files (see Chapter 1 [Input Data], page 1). Appropriate file extensions must be used to indicate the type of file, for example, `data.csv`, `data.iet` or `data.plm`. Databases can be specified as a DSN or ODBC or JDBC connection string (see `KojakDB` for more details). This parameter can also be specified by pointing to another parameter such as `KojakDB` whose value will then be used. `Data` can have multiple values specified via the `+=` syntax. For example:

```
Data +=KJ:data;example1;ds1-group-seeds-all.csv
Data +=KojakDB
```

Data sources can also be specified via the -d or –data command line options which can be given multiple times.

The value(s) of the `Data` parameter will be used by default by the `load-data` command to load data (unless overridden by actual parameters). If a database source is

specified, the script specified by `LoadDataScript` will be used to load data from that database.

`DisabledData` [Parameter]
`DisabledData` is used by the Kojak interface code to record data specifications which are not currently used in the analysis, but which are made available for selection in the interface. It uses the same format as `Data`.

`PrimaryData` [Parameter]
Similar to `Data` but uses the value of `LoadPrimaryDataScript` to load database data sources. These data sources are not loaded by default by `load-data` but can be specified as follows:

```
(load-data :data-source "PrimaryData")
```

Primary data can also be specified via the `-pd` or `--primary-data` options.

`SecondaryData` [Parameter]
Similar to `Data` but uses the value of `LoadSecondaryDataScript` to load database data sources. These data sources are not loaded by default by `load-data` but can be specified as follows:

```
(load-data :data-source "SecondaryData")
```

Secondary data can also be specified via the `-sd` or `--secondary-data` options.

`DatasetName` [Parameter]
Serves as a single logical name for the supplied input data and all results generated from it. It can also be specified via the `-n` or `--dataset-name` command-line options. Dataset names are used as metadata for results deposited in the hypothesis tables of the KOJAK DB.

`DatasetType` [Parameter]
Indicates what kind of data we are looking at; mainly used to flag simulated data generated by IET, since some special assumptions apply there; if this is not given, it is guessed from some of the Data specifications.

```
DatasetType =IET_2004
```

It can also be specified via the `-t` or `--dataset-type` options.

2.1.3 Script Specifications

The Group Finder uses a variety of scripts to load data, ontologies, schema information, execute commands, etc. The following parameters allow customization of these scripts.

`LoadOntologyScript` [Parameter]
The default script used by `initialize-kojak` and `load-kojak-ontology` to load ontology information into PowerLoom.

`LoadEDBSchemaScript` [Parameter]
The default script used by `initialize-kojak` and `load-edb-schema` to load evidence database schema information into PowerLoom.

`LoadDataScript` [Parameter]
The default script used by `load-data` to load RDBMS data sources specified in the `Data` configuration parameter.

The following scripts are primarily relevant to IET data sources and somewhat obsolete. They are the default scripts used when an RDBMS data source is loaded from a `PrimaryData`, `SecondaryData`, `PrimaryLDData` or `PrimaryPLData` specification.

`LoadPrimaryDataScript` [Parameter]

`LoadSecondaryDataScript` [Parameter]

`LoadPrimaryLDDataScript` [Parameter]

`LoadPrimaryPLDataScript` [Parameter]

`RunKojakScript` [Parameter]
The top-level KOJAK run script. Commands in this script are executed when the `main` function of the KOJAK Group Finder is run in non-interactive mode, or when the `run-kojak` command is executed. This can also be specified via the `-s` or `--run-script` options.

2.1.4 Analysis Specifications

`RunID` [Parameter]
Marks a particular configuration used to analyze `DatasetName` when the configuration gets saved to the KOJAK database. If none is supplied a new one is generated automatically. This is useful to track different configuration settings across multiple analysis runs. It can also be specified via the -r or –run-id options.

`RunPrefix` [Parameter]
Used as a prefix when Kojak generates an automatic `RunID`. It can also be specified via the -rp or –run-prefix options.

`ClearOldHypotheses` [Parameter]
If true, hypotheses generated in previous runs will be deleted from the KOJAK database before any new ones are generated.

`ExpansionDepth` [Parameter]
Depth of graph expansion from seed groups to get extended groups. **IMPORTANT**: due to graph fanout this value has a big impact on run time. Use values larger than 1 only for cases with very small seed groups (e.g., 1-5 members) and increase it only one step at a time if needed. It can also be specified via the `--depth` option.

`ExpansionLinkTypes` [Parameter]
The default link types that should be considered by the `expand-groups` command when expanding seed groups and computing mutual information between entities. These should correspond to the relevant link types loaded and accessed from the evidence database (for example, `telecon`, `phoneCall`, etc.). If no link types are specified the default is `anyLink`. This is a multi-valued parameter.

`DisabledExpansionLinkTypes` [Parameter]

`DisabledExpansionLinkTypes` is used by the Kojak interface code to record link types which are not currently used in the analysis, but which are made available for selection in the interface. It uses the same format as `ExpansionLinkTypes`.

`ExpansionMethods` [Parameter]

The default strength computation methods considered by `expand-groups`. These are used to compute connection strength measures between entities and seed groups. This is a multi-valued parameter whose values are treated case-insensitively. Legal values are the following:

- `MI`: computes the mutual information value between an entity and a seed group member based on the complete set of links between two entities.
- `Group-MI`: views the whole seed group as a single individual and computes the MI between this "group individual" and an entity based on all the links between the entity and the "group individual".
- `MI-or-Group-MI`: computes both `MI` and `Group-MI` values.
- `Link-Count`: Computes link counts between an entity and seed group members.
- `Connectivity`: Computes the number of seed group members connected to an entity.

The set `Connectivity`, `Link-Count` and `Group-MI` form the default.

NOTE: this parameter will eventually go away, since which expansion methods are required can be inferred from the `ExpansionMeasures` described below.

`ExpansionMeasures` [Parameter]

The default strength computation measures that should be used by `expand-groups`. Each strength computation method generates a variety of results such as maximums, averages, etc. The set of expansion measures selects a subset of those result measures and averages them to generate a single result measure (this averaging is called "bagging"). This is a multi-valued parameter that can take values from the following set (values are treated case-insensitively):

- `max-MI`: computes the maximum MI between an entity and any of the seed group members as the connection strength to the seed group.
- `max-MI-correlation`: computes `max-MI` and normalizes it as a correlation value.
- `sum-MI`: sums up the MIs between an entity and each seed group member.
- `group-MI`: computes the MI between an entity and the set of seed group members when viewed as a single super individual (combining all their links).
- `group-MI-correlation`: computes `group-MI` and normalizes it as a correlation value.
- `MI-or-group-MI`: computes the maximum of the `max-MI` and `group-MI` measures.
- `MI-or-group-MI-correlation`: computes `MI-or-group-MI` and normalizes it as a correlation value.
- `max-count`: computes the maximum link count between an entity and any of the seed group members as the connection strength to the seed group.

- `sum-count`: sums up the link counts between an entity and each seed group member.
- `connectivity`: computes the number of seed members an entity is connected to.

Each of the above measures will be normalized to a value between 0 and 1 to allow proper averaging. The best performing measures in experiments performed so far are `sum-count`, `group-MI` and `connectivity` which form the default set if no measures are specified.

Boosting can be useful in cases where we have very few seed members for a group. Instead of running the group expansion only once, we take a small number of the top new members, add them to the seed group (i.e., "boost" the number of seeds) and run again. This is useful if the top members have very high likelihood of being correct. The parameters below control how and how often boosting is done. All of these values can be overridden when calling `expand-groups`.

`BoostEnabled` [Parameter]
A boolean parameter that determines whether boosting is performed or not. If this is not explicitly specified, then (for backward compatibility) boosting will be done if either `BoostMaxCycles` or `BoostMinSeeds` has a value.

`BoostMaxCycles` [Parameter]
Perform at most this many boosting cycles even if the targets have not been reached (leave this undefined if boosting is controlled by `BoostMinSeeds`):

`BoostMinSeeds` [Parameter]
Boost until seed groups have at least that many members; if both this and `BoostMaxCycles` are undefined boosting is disabled.

`BoostFactor` [Parameter]
In each boost iteration, increase the number of seed members by this factor.

`BoostMinStep` [Parameter]
In each boost iteration, increase the number of seed members at least by that many members, even if `BoostFactor` tells us otherwise.

2.1.5 Output Specifications

`ReportFile` [Parameter]
File to which Group Finder results should be reported. If no `ReportFormat` (which see) is specified, the file extension determines which format to use. The report file can also be specified via the `-o` or `--report-file` options. Note that results can also be output to the KOJAK database by using the `db-save-groups` command.

`ReportDirectory` [Parameter]
Directory path to prepend to `ReportFiles` which is useful for multiple runs via scripts.

`ReportFormat` [Parameter]
Format to use for report files generated by `report-groups`. Supported formats are `IET-Report` (equivalent to `IET-Y3-Report`), `IET-Y2-Report` and `CSV-Table`. If not

specified, the format will be guessed from the extension of `ReportFile`. A `.csv` extension means use the `CSV-Table` format, everything else is interpreted as `IET-Report`.

`ReportMemberWeights` [Parameter]
If true, membership weights will be reported for each group member.

`WeightDecimals` [Parameter]
The number of decimals behind the decimal point that should be used to print a weight in fixed-point format to the output file (defaults to 6).

`LastRunCutoff` [Parameter]
The computed number of to-be-reported members will usually wind up somewhere within a run of members with equal membership weights. Cutting right there would arbitrarily include and exclude members that are really indistinguishable by their membership weight. `LastRunCutoff` determines how this last run should be cut. Legal values are `Inclusive` which keeps the whole run, `Exclusive` which cuts at the end of the previous run, `Hard` which cuts right at the computed threshold point or `Closest` which cuts at the closest endpoint of the run. The default is `Inclusive`. If the last run is also the first run, it will always be reported in full unless the value is `Hard`; if we fall on the last member of a run the run will be reported in full even if the cutoff value is `Exclusive`.

`ReportNames` [Parameter]
If true, group and member print names will be looked up via the `GROUPS/nameString` relation and printed instead of the IDs or names identifying them in evidence. This is useful if groups and members are identified via unique but unreadable IDs in evidence and more readable reports need to be generated. This will only work if appropriate `GROUPS/nameString` assertions are available in the evidence, or if there are mapping rules that can retrieve them from an EDB.

The following parameters provide some basic control for thresholding of extended groups. Reported groups are sorted in descending order of membership weight and thresholding controls where this list is cut. Automatic thresholding is still not very well supported by the Group Finder, since there are no clear transitions indicating where true members stop and non-members start. Besides the very basic controls provided below, specialized functions can be written that provide thresholding appropriate for a particular domain (see the `threshold-groups` command).

`ReportGroupFraction` [Parameter]
Fraction of individuals in the extended graph that should be reported in the result. Includes seed members; 1.0 reports the whole group. This can also be specified via the `-gf` or `--report-group-fraction` options.

`ReportMinMembers` [Parameter]
Minimum number of group members to report (including seeds).

`ReportMaxMembers` [Parameter]
Maximum number of group members to report (including seeds).

2.2 Command-Line Options

What data is analyzed by KOJAK and how it analyzes it is controlled by a large number of parameters given in the 'configuration.dat' configuration file in the KOJAK directory. See the previous section for documentation of the various available configuration parameters. Many of these parameters can also be specified and overridden from the Unix command line. Here is the full list of available command-line options with a brief description of their functionality:

```
run-kojak [{-c|--configuration-file} FILE] [-i|--interactive]
          [{-l|--log-level"} LEVEL] [{-d|--data} DATASPEC]*
          [{-pd|--primary-data} DATASPEC]*
          [{-sd|--secondary-data} DATASPEC]*
          [{-k|--kojak-db} DBSPEC] [{-n|--dataset-name} NAME]
          [{-t|--dataset-type} TYPE] [{-r|--run-id} ID]
          [{-r|--run-prefix} PREFIX]
          [{-s|--run-script} FILE] [--depth N] [{-o|--report-file} FILE]
          [{-gf|--report-group-fraction} FRACTION]
```

'`--configuration-file` *FILE*'
'`-c` *FILE*' Specifies *FILE* to be used as the configuration file. By default, 'configuration.dat' in the KOJAK directory is used. If there is no valid configuration file specified or available, the Group Finder terminates immediately.

'`--interactive`'
'`-i`' Run the Group Finder interactively by bringing up the PowerLoom command loop. At that point any PowerLoom or KOJAK command can be executed. Calling the `run-kojak` command will run the Group Finder end-to-end.

'`--log-level` *LEVEL*'
'`-l` *LEVEL*'
Sets the `LogLevel` parameter to *LEVEL*.

'`--data` *DATASPEC*'
'`-d` *DATASPEC*'
Sets the `Data` parameter to *DATASPEC*. This option can be supplied multiple times.

'`--primary-data` *DATASPEC*'
'`-pd` *DATASPEC*'
Sets the `PrimaryData` parameter to *DATASPEC*. This option can be supplied multiple times.

'`--secondary-data` *DATASPEC*'
'`-sd` *DATASPEC*'
Sets the `SecondaryData` parameter to *DATASPEC*. This option can be supplied multiple times.

'`--kojak-db` *DBSPEC*'
'`-k` *DBSPEC*'
Sets the `KojakDB` parameter to *DBSPEC*.

'`--dataset-name` *NAME*'
'`-n` *NAME*' Sets the `DatasetName` parameter to *NAME*.

'`--dataset-type` *TYPE*'
'`-t` *TYPE*' Sets the `DatasetType` parameter to *TYPE*.

'`--run-id` *ID*'
'`-r` *ID*' Sets the `RunID` parameter to *ID*.

'`--run-prefix` *PREFIX*'
'`-rp` *PREFIX*'
Sets the `RunPrefix` parameter to *PREFIX*.

'`--run-script` *FILE*'
'`-s` *FILE*' Sets the `RunKojakScript` parameter to *FILE*.

'`--depth` *N*'
Sets the `ExpansionDepth` parameter to *N*.

'`--report-file` *FILE*'
'`-o` *FILE*' Sets the `ReportFile` parameter to *FILE*.

'`--report-group-fraction` *FRACTION*'
'`-gf` *FRACTION*'
Sets the `ReportGroupFraction` parameter to *FRACTION*.

'`--define` *PAR*`=`*VAL*`, --define` *PAR*`+=`*VAL*'
'`-D` *PAR*`=`*VAL*`, -D` *PAR*`+=`*VAL*'
Can be used to set an arbitrary configuration parameter *PAR* to value *VAL*. Both = and += syntax are supported.

KOJAK Group Finder
Hardware and Software Requirements

This manual describes the hardware and
software requirements for
KOJAK Group Finder 2.3 or later.
Document 2.0

14 September 2006

The KOJAK development team
Hans Chalupsky
Jafar Adibi
Thomas A. Russ
{hans,adibi,tar}@isi.edu

Table of Contents

1 Hardware Requirements D-1
1.1 KOJAK Group Finder D-1
1.2 Database Server D-1

2 Software Requirements D-2
2.1 Operating System D-2
2.2 Runtime Environment D-2
2.2.1 Operational Environment D-2
2.2.2 Build Environment D-2
2.3 Databases D-2
2.3.1 Database Systems D-2
2.3.2 Database Drivers D-3

1 Hardware Requirements

There are two parts to the hardware requirements. The machine that is used to run the KOJAK Group Finder software and the machine(s) that run the database systems.

The KOJAK Group Finder can be run on the same computer as the database server or it can be run on a separate computer. This decision is left up to the end user.

1.1 KOJAK Group Finder

The KOJAK Group Finder software runs on convential hardware. The suggested specifications are:

- Pentium-IV class processor
- Clock speed: 1GHz minimum, 3GHz recommended.
- RAM: 512MB minimum, 2GB recommended
- Disk Space.
  - 50MB the Group Finder software.
  - 100MB minimum for the Group Finder's internal database. More may be needed, depending on size and characteristics of the datasets to be processed.
  - 10 GB additional to hold data if used as database server as well.

Note that we have also run the Group Finder software on PowerPC G4 architecture processors as well.

1.2 Database Server.

Graph queries are hard on relational database management systems (RDBMS), so depending on the particular datasets involved, a powerful database server might needed. This would be any hardware that could run one of the supported database systems.

We do note that we have not needed specialized hardware for this purpose so far, and we have analyzed datasets with up to 100,000,000 binary links.

2 Software Requirements

The KOJAK Group Finder is distributed in both C++ and Java and supports Linux (C++ and Java) as well as Windows (Java only).

2.1 Operating System

The following operating systems are currently supported and have been tested for operation.

- Linux, Redhat 9.0 or Suse 9.2
- Mac OS X, 10.3.x
- Windows 2000 and XP

2.2 Runtime Environment

2.2.1 Operational Environment

This section details the requirements for running the pre-compiled versions of the Group Finder software. In addition to these requirements, the appropriate database support software will also be required.

- **C++ implementation** The C++ implementation requires a compiled and linked application for the hardware and operating system in question. It also requires ODBC drivers and a driver manager, [databases], page 2.
- **Java implementation** The Java implementation of KOJAK requires JRE version 1.4.2 or 1.5; J2EE version 1.3 or 1.4; and JDBC drivers for the database backend. [databases], page 2.

2.2.2 Build Environment

To build or compile from sources, the following resources are needed.

- **C++ implementation** C++ requires the g++ 3.3 compiler for Linux or Mac OS X. The C++ version is not (yet) available for Windows.
- **Java implementation** Java requires JDK version 1.4.2 or 1.5 and J2EE version 1.3 or 1.4.

2.3 Databases

KOJAK uses a database internally to store evidence and intermediate hypotheses. KOJAK can also optionally use a database as the source of the evidence used in finding groups.

2.3.1 Database Systems

The following database systems are supported by the KOJAK Group Finder.

- **MySQL** MySQL database, version 4.0 or later
- **Oracle** An Oracle Database 10g system.

2.3.2 Database Drivers

- **C++ implementation** C++ ODBC for database connectivity and an ODBC driver manager.

  Current versions are the MyODBC 3.51 driver for MySQL and either the iODBC or UnixODBC driver managers. We recommend using iODBC, since that is the standard configuration distributed with the KOJAK Group Finder.

  If you are using Oracle you can use a free open-source ODBC driver for Oracle under Linux available from `http://fndapl.fnal.gov/~dbox/oracle/odbc/`. There is also a somewhat expensive commercial driver available from Easysoft `http://www.easysoft.com/` which is a better alternative; however, due to its cost we have not yet experimented with this driver.

- **Java implementation** Java requires a JDBC driver for the database in question. For MySQL, the mysqlconnector version 3.0 is recommended (appropriate jar files ship with the distribution).

  For Oracle, '`ojdbc14.jar`' which is the Java 1.4 driver from the Oracle 10g distribution is recommended.

Please refer to the KOJAK Group Finder manual for more information on these requirements.

KOJAK Group Finder Manual

Scalable Group Detection via Integrated
Knowledge-Based and Statistical Reasoning

This manual describes
KOJAK Group Finder 2.3 or later.
Document $Revision: 1.6 $

14 September 2006

The KOJAK development team
Hans Chalupsky
Jafar Adibi
Thomas A. Russ
Andre Valente
Eric Melz
{hans,adibi,tar,valente}@isi.edu

Table of Contents

1 Introduction E-1

1.1 The Group Detection Problem E-1

2 Installation E-2

2.1 System Requirements E-2
2.2 Linux Installation E-2
2.2.1 Recompiling the C++ Sources E-3
2.2.2 Java Configuration for Linux E-3
2.3 Windows Installation E-4
2.3.1 Java Configuration for Windows E-4
2.4 Database Configuration E-5
2.4.1 KOJAK DB and EDB Schema E-5
2.4.1.1 MySQL Setup E-5
2.4.1.2 Oracle Setup E-6
2.4.2 ODBC and Driver Managers E-6
2.4.2.1 .odbc.ini File E-6
2.4.3 JDBC Drivers E-7

3 General Operation E-8

4 Input Data E-9

4.1 Types of Data E-9
4.1.1 Group Seeds E-9
4.1.2 Link Data E-10
4.2 Data Access Classes E-11
4.2.1 Primary Data E-11
4.2.2 Secondary Data E-11
4.2.3 Undifferentiated Data E-11
4.3 Data Formats E-11
4.3.1 Comma-Separated Value Format E-12
4.3.2 IET Format E-12
4.3.3 PowerLoom Format E-12
4.3.4 Relational Databases E-13

5 Configuration E-14

5.1 Configuration File E-14
5.1.1 General Options E-15
5.1.2 Input Specifications E-16
5.1.3 Script Specifications E-17
5.1.4 Analysis Specifications E-18
5.1.5 Output Specifications E-20
5.2 Command-Line Options E-22

6 Running the KOJAK Group Finder E-24

6.1 Example Run 1 ..... E-24
6.2 Other Example Runs E-32

7 Advanced Configuration E-33

7.1 PowerLoom E-34
7.2 File and Module Structure E-35
7.2.1 Module Structure E-36
7.3 The Ali Baba Configuration E-37
7.3.1 'ali-baba.dat' E-37
7.3.2 'ali-baba-ontology.plm' E-38
7.3.3 'ali-baba-seed-constraints.plm' E-42
7.3.4 'ali-baba-edb-schema.plm' E-46
7.3.4.1 Database Instances E-46
7.3.4.2 Table Mappings E-47
7.3.4.3 Materializing Type IDs E-50
7.3.4.4 Defining Link Count Relations E-52
7.3.5 'ali-baba-load-ontology.plm' E-58
7.3.6 'ali-baba-load-edb-schema.plm' E-58
7.3.7 'ali-baba-load-data.plm' E-59
7.3.8 'ali-baba-run-kojak.plm' E-61

8 KOJAK Commands E-62

8.1 Important PowerLoom Commands E-68
8.2 Important RDBMS Commands E-73
8.3 XTIE-Specific Commands for IET Datasets E-74

9 Group Finder Ontology E-76

10 Questions and Comments E-78

1 Introduction

This document describes the KOJAK Group Finder which is a scalable, hybrid logic-based/statistical group detection system that can detect and extend groups in large evidence databases. For additional information on the KOJAK Group Finder see our IAAI-2004 and IA-2005 papers included in the 'doc' directory of the distribution or available on the web at http://www.isi.edu/~hans/publications.html. For additional documentation on the PowerLoom knowledge representation & reasoning system underlying the logic-based portion of the Group Finder please refer to http://www.isi.edu/isd/LOOM/PowerLoom/.

The Group Finder is part of a larger hybrid link discovery system called KOJAK that we are currently developing. It combines state-of-the-art knowledge representation and reasoning (KR&R) technology with statistical clustering and analysis techniques from the area of data mining. Using KR&R technology allows us to represent extracted evidence at very high fidelity, build and utilize high quality and reusable ontologies and domain theories, have a natural means to represent abstraction and meta-knowledge such as the interestingness of certain relations, and leverage sophisticated reasoning algorithms to uncover implicit semantic connections. Using data or knowledge mining technology allows us to uncover hidden relationships not explicitly represented in the data or findable by logical inference, use clustering techniques to find sets of entities that are temporally or organizationally related, or use clustering to improve efficiency by carving up a very large data space into smaller more manageable pieces.

1.1 The Group Detection Problem

A major problem in the area of link discovery is the discovery of hidden organizational structure such as groups and their members. There are of course many organizations and groups visible and detectable in real world data, but we are usually only interested in detecting certain types of groups such as organized crime rings, terrorist groups, etc. Group detection can be further broken down into (1) discovering hidden members of known groups (or group extension) and (2) identifying completely unknown groups.

A known group (e.g., a terrorist group such as the RAF) is identified by a given name and a set of known members. The problem then is to discover potential additional hidden members of such a group given evidence of communication events, business transactions, familial relationships, etc. For unknown groups neither name nor known members are available. All we know are certain suspicious individuals ("bad guys") in the database and their connection to certain events of interest. The main task here is to identify additional suspicious individuals and cluster them appropriately to hypothesize new real-world groups, e.g., a new money laundering ring. While our techniques address both questions, we believe group extension to be the more common and important problem.

The Group Finder is capable of finding hidden groups and group members in large evidence databases. Our group finding approach addresses a variety of important LD challenges, such as being able to exploit heterogeneous and structurally rich evidence, handling the connectivity curse, noise and corruption as well as the capability to scale up to very large, realistic data sets. The first version of the KOJAK Group Finder has been successfully tested and evaluated on a variety of synthetic datasets.

2 Installation

2.1 System Requirements

To install and use KOJAK you'll approximately need the following amounts of disk space:

- 17-20 MB for the tar-red or zip-ped archive file
- 55 MB for the untarred C++ and Java sources, compiled KOJAK libraries, data and documentation

This version of KOJAK comes in C++ and Java versions and supports Linux (C++ and Java) as well as Windows (Java only). Linux distributions such as RedHat or Suse as well as Windows 2000 and XP should support it out of the box. It also requires access to a database system such as MySQL (4.0 or later) and/or Oracle with appropriate ODBC and/or JDBC drivers and driver managers. KOJAK was primarily developed and tested under Linux RedHat 8.0 and 9.0 as well as Suse 9.2 with MySQL 4.1.0-alpha and MyODBC 3.51.11. We successfully tested it with both the iODBC 3.0.6 and UnixODBC 2.2.9 driver managers. We also successfully ran it with Oracle-10g using the Oracle JDBC Thin Client as well as an experimental open-source Oracle ODBC driver for Linux from `http://fndapl.fnal.gov/~dbox/oracle/odbc/`. The closer your environment is to some cross-section of that the higher the chances that things will work right out of the box. This release is new and fairly complex, so there is a definite chance for problems when installing it in a different environment.

The C++ version of KOJAK can be about 1.5 times faster than the Java version; however, actual speedup depends significantly on how much time is spent for database access which will be the same for C++ and Java. If most time is spent waiting for the database server, elapsed run times of the C++ and Java versions will be very similar. The faster speed of the C++ version might have to be paid for with a potentially more difficult installation process of the necessary ODBC driver support.

2.2 Linux Installation

Both the C++ and Java versions of KOJAK can be run on Linux platforms such as Suse Linux 9.2 or RedHat 9.0. To install KOJAK under Linux choose an installation location and then uncompress and untar the file '`kojak-X.Y.Z.tar.gz`' (or unzip the file '`kojak-X.Y.Z.zip`') in the parent directory of that location. '`X.Y.Z`' are place holders for the actual version numbers. For example:

```
% cd install-dir
% tar xzf kojak-2.2.0.tar.gz
```

This will create the KOJAK tree in the directory '`kojak-X.Y.Z/`' ('`install-dir/kojak-2.2.0/`' in the example). All pathnames mentioned below will be relative to that directory which we will usually refer to as the "KOJAK directory".

Both the C++ and Java versions are already precompiled and should be ready to use. The C++ version is compiled with the iODBC driver manager library iODBC 3.0.6 and the Java version was compiled with Java J2SDK 1.4.2. If that matches your local setup you are done and can go on to Section 2.4 [Database Configuration], page 5. If you want to use a different ODBC driver manager such as UnixODBC go on to Section 2.2.1 [Recompiling

the C++ Sources], page 3. If you want to use a different version of Java or have it installed in a non-standard location go on to Section 2.2.2 [Java Configuration for Linux], page 3.

To run KOJAK use the `run-kojak` script and supply `c++` or `java` as the first argument to select which version should be run (the C++ version is run by default if neither `c++` or `java` is supplied). For example:

```
% cd <KOJAK directory>
% run-kojak c++ -c config/example3-no-db.dat
```

There are actually two script files: `run-kojak` for Linux and `run-kojak.bat` for Windows. When you execute the `run-kojak` command the OS will automatically select the appropriate script.

Note that for most of the configuration files shipping with KOJAK you need to have a database server and appropriate drivers set up first before you can run them (see Section 2.4 [Database Configuration], page 5).

A Java-based GUI is currently in a prototype stage and will ship with one of the next versions of KOJAK. This GUI will make it easier to edit configuration files and run KOJAK.

2.2.1 Recompiling the C++ Sources

If you want to use a different ODBC driver manager or want to use a different compiler or compiler settings you can recompile KOJAK with help of the top-level '`Makefile`'. Edit any of the variables in the '`Makefile`' if necessary and then call `make` from the KOJAK directory to recompile the system. Note that all the provided source code was generated automatically from STELLA sources (see http://www.isi.edu/isd/LOOM/Stella/index.html for information on the STELLA programming language).

If you want to use KOJAK with the UnixODBC driver manager instead of the iODBC manager it ships with, you need to recompile it by doing the following:

1. Edit the top-level '`Makefile`' in the KOJAK directory so that its `CFLAGS` and `LDFLAGS` will point to the appropriate locations for your UnixODBC installation (the include files provided in the top-level '`include`' directory come from iODBC but should work for UnixODBC as well). Also edit the value of the `ODBC-LIB` variable to use `-lodbc` as its value.
2. Run `make` in the KOJAK directory which will recompile it with the UnixODBC libraries on your machine. This will require GNU make as well as a recent version of g++ such as g++ 3.3.4.
3. If your UnixODBC library is installed in a non-standard place, you will need to update `LD_LIBRARY_PATH` in the `run-kojak` script accordingly.

2.2.2 Java Configuration for Linux

The Java version of KOJAK is already pre-compiled for Java J2SDK 1.4.2 and archived in the '`kojak.jar`' archive which can be found in '`native/java/lib/`'. Using Java 1.4 should run KOJAK without any problems. Using newer (or slightly older) versions of Java (e.g., the new version 1.5) should work as well, but we have not done any testing to that extent.

The default Java configuration in the '`run-kojak`' script looks for a `java` executable in the current command path. If you want to use a different version of Java or if it is no installed in the standard path please edit the `JAVA` variable in the '`run-kojak`' script accordingly.

2.3 Windows Installation

The Java version of KOJAK now supports Windows operating systems such as Windows 2000 and Windows XP (the C++ version is not yet available for Windows). To install KOJAK under Windows choose an installation location (e.g., 'C:\Program Files\') and then unzip the file 'kojak-X.Y.Z.zip' in that location using a utility such as WinZip (if you use a different utility make sure it does appropriate translation of line endings in text files from the Unix to the Windows convention). 'X.Y.Z' are place holders for the actual version numbers. This will create the KOJAK tree in the folder 'kojak-X.Y.Z\' (e.g., 'C:\Program Files\kojak-2.2.0\'). All pathnames mentioned below will be relative to that directory which we will usually refer to as the "KOJAK directory".

For most of this manual we will use Unix syntax for pathnames. To translate those into appropriate Windows pathnames simply substitute \ for the Unix / pathname separator. IMPORTANT: if you supply a physical Windows pathname somewhere in a KOJAK command that takes a quoted string as an argument, you will need to double the \ character, since it is also the escape character for strings (but you don't have to do that for pathnames in configuration files). For example:

```
(load-kojak-configuration :config-file "C:\\kojak\\myconfig.dat")
```

The Java version is already precompiled with Java J2SDK 1.4.2 and should now be ready to use. If that matches your local setup you are done and can go on to Section 2.4 [Database Configuration], page 5. If you want to use a different version of Java or have it installed in a non-standard location go on to Section 2.3.1 [Java Configuration for Windows], page 4.

To run KOJAK you have to launch a Command Prompt window and then run the run-kojak.bat script supplying java as the first argument (in fact, the Java version is run by default under Windows unless the first argument is c++). For example:

```
C:\> cd <KOJAK directory>
C:\...> run-kojak java -c config\example3-no-db.dat
```

There are actually two script files: run-kojak for Linux and run-kojak.bat for Windows. When you execute the run-kojak command the OS will automatically select the appropriate script.

Note that for most of the configuration files shipping with KOJAK you need to have a database server and appropriate drivers set up first before you can run them (see Section 2.4 [Database Configuration], page 5).

A Java-based GUI is currently in a prototype stage and will ship with one of the next versions of KOJAK. This GUI will make it easier to edit configuration files and run KOJAK without using a Command Prompt window.

2.3.1 Java Configuration for Windows

The Java version of KOJAK is already pre-compiled for Java J2SDK 1.4.2 and archived in the 'kojak.jar' archive which can be found in 'native\java\lib/'. Using Java 1.4 should run KOJAK without any problems. Using newer (or slightly older) versions of Java (e.g., the new version 1.5) should work as well, but we have not done any testing to that extent.

The default Java configuration in the 'run-kojak.bat' script looks for a java executable in the current command path. If you want to use a different version of Java or if it is no installed in the standard path please edit the %JAVA% variable in the 'run-kojak.bat' script accordingly.

2.4 Database Configuration

The Group Finder uses a relational database server such as MySQL and/or Oracle to store group and membership hypotheses, configuration information and analysis metadata, as well as to import evidence data if necessary. Additionally, the Group Finder can work directly with information stored in existing Oracle or MySQL databases (given an appropriate mapping specification) without having to translate the whole database or importing/loading it into memory.

While it is possible to run KOJAK without access to a database server (see for example the configuration file '`config/example3-no-db.dat`') this is not recommended for analyzing large datasets. If KOJAK analyzes a large dataset without a database server, it needs to load all data into main memory which might be prohibitive depending on dataset size. If it can use a database server, it will be able to load data very selectively and off-load a lot of aggregation and processing to the database which greatly improves scalability and reduces the memory footprint.

If you want to use MySQL as the database server, make sure you have a recent version of MySQL 4.0 or later installed or available on a server (see http://www.mysql.com/). Alternatively, you can use an Oracle database server such as Oracle 10g (see http://www.oracle.com/). Earlier versions of Oracle are likely to work as well, but we have only tested KOJAK with version 10g so far. KOJAK can also work with MySQL and Oracle simultaneously, for example, to store the internal KOJAK database under MySQL and access evidence from an Oracle server.

2.4.1 KOJAK DB and EDB Schema

The Group Finder relies on a set of database tables we call the "KOJAK database" to store group and membership hypotheses, various configuration information or to import evidence data from flat files. Before you can run KOJAK with a database you have to create these KOJAK database tables in your database server.

The KOJAK database tables should best reside in a separate schema to avoid conflict or loss of existing information. Under MySQL we create a new KOJAK database for this purpose, with Oracle we create a new KOJAK user to hold this schema. It is, however, possible to add these tables directly to an existing schema if so desired. In this case it is very important to ensure that there are no preexisting tables with the same name in the schema where they are added.

2.4.1.1 MySQL Setup

To create the KOJAK database under MySQL use the following steps in Linux (you will need the appropriate privileges to create these schema objects). For Windows users the steps should be similar, just use the appropriate Windows MySQL client:

```
% cd <KOJAK directory>
% mysql -u <dbuser> -p
mysql> source kbs/kojak-db-schema-mysql.sql;
mysql> source kbs/kojak-edb-schema-mysql.sql;
```

Make sure you load the two files exactly in that order. The first file will create the new KOJAK database and add the various KOJAK hypothesis and configuration tables. If you don't want to create a new KOJAK database but instead want to add the tables

to an existing database, you need to edit the database creation and use commands in 'kbs/kojak-db-schema-mysql.sql' accordingly before you load it.

2.4.1.2 Oracle Setup

To create the KOJAK database under Oracle use the following steps (you will need appropriate SYSDBA privileges to create these schema objects): First edit the file 'kbs/kojak-db-schema-oracle.sql' to use the appropriate password information for user KOJAK. Alternatively, you can have a DBA create the KOJAK user for you and simply comment or delete the user creation statements in the script. Under Linux you can use the SQL*Plus tool to load these scripts, for example:

```
% cd <KOJAK directory>
% sqlplus /nolog
SQL> connect sys as sysdba
SQL> start kbs/kojak-db-schema-oracle.sql
SQL> start kbs/kojak-edb-schema-oracle.sql
```

Alternatively (or for Windows users), you can paste the content of these files into the iSQL*Plus client of the Oracle Enterprise Manager. If you don't want to create a new KOJAK user but instead want to add the tables to an existing schema, you need to edit the user creation and connect commands in 'kbs/kojak-db-schema-oracle.sql' accordingly before you load it.

2.4.2 ODBC and Driver Managers

If you want to use the C++ version of KOJAK (currently only supported for Linux) you need to have an appropriate ODBC driver (e.g., MyODBC 3.51 if you are using MySQL) and ODBC driver managers (such as iODBC 3.0.6 or UnixODBC 2.2.9) installed on the machine where you run KOJAK.

Visit http://www.openlinksw.com/iodbc/ (or http://www.unixodbc.org/) for information on ODBC drivers and driver managers for Linux. The C++ version of KOJAK is precompiled with the iODBC driver manager library 'libiodbc.so.2.1.6'. It might also be necessary to uninstall pre-installed conflicting versions such as MyODBC-2.50 and UnixODBC which come pre-installed with some flavors of Linux. If you want to use KOJAK with the UnixODBC driver manager instead, you need to recompile it with the different libraries (see Section 2.2.1 [Recompiling the C++ Sources], page 3).

If you are using MySQL you will need to install the MyODBC 3.51 driver (see http://dev.mysql.com/downloads/connector/odbc/3.51.html for more information). If you are using Oracle you can use a free open-source ODBC driver for Oracle under Linux available from http://fndapl.fnal.gov/~dbox/oracle/odbc/. This is an unsupported alpha release (version 0.5.5) that takes some work to install, but we have used this version successfully. There is also a somewhat expensive commercial driver available from Easysoft http://www.easysoft.com/ which is a better alternative; however, due to its cost we have not yet experimented with this driver.

2.4.2.1 .odbc.ini File

After you have ensured the installation of proper ODBC drivers and driver managers for your database server, copy the provided '.odbc.ini' file to your home directory (editing the local copy in the KOJAK directory will not have any effect!) and adapt the [kojak]

entry with the appropriate access information (user, password, driver, server, etc.). Data source specifications given to KOJAK will use this information as defaults for fields that are not provided in ODBC connection strings specified in the configuration file.

2.4.3 JDBC Drivers

If you are running the Java version of KOJAK it will need to use JDBC to communicate with database servers. KOJAK currently supports the following two JDBC driver class implementations:

1. `com.mysql.jdbc.Driver` for use with MySQL
2. `oracle.jdbc.driver.OracleDriver` for use with Oracle (Thin Client)

KOJAK currently ships with various driver implementations of these classes (jar files located in directory '`native/java/lib/`'. For MySQL we use '`mysql-connector-java-3.0.17-ga-bin.jar`' and for Oracle we use '`ojdbc14.jar`' which is the Java 1.4 driver from the Oracle 10g distribution. These driver files are communicated to KOJAK via the environment variables in the '`run-kojak`' and '`run-kojak.bat`' scripts. If you want/need to use a different JDBC driver (for one or both of the driver classes shown above), you can do so by editing the appropriate variables in the '`run-kojak`' and/or '`run-kojak.bat`' scripts. You only need to supply a driver for the database system you are using, i.e., if you are only using MySQL you don't need to supply an Oracle driver and vice versa.

For MySQL a set of newer, alternative drivers are also shipped in the '`native/java/lib/`' directory, since different versions of MySQL seem to need different JDBC drivers. These drivers are '`mysql-connector-java-3.1.10-bin.jar`' and '`mysql-connector-java-3.2.0-alpha-bin.jar`'. If you experience problems with the default JDBC driver for MySQL try to substitute one of these alternatives and see whether it fixes the problem.

3 General Operation

The KOJAK Group Finder is really a "group expander', i.e., given one or more seed groups of interest, it finds entities strongly connected to known members of those seed groups by analyzing available link information such as, for example, communication logs (phone, email, etc.), financial transactions, etc. As a result it returns those additional entities that are most strongly connected to seed groups ranked by strength of their connection. For example, if in a terrorist domain we have a seed group of known terrorists, the Group Finder will return people that are very strongly connected to those terrorists based on the available link information. The assumption is that such strongly connected people might be additional (as yet unknown) terrorists or "bad guys" themselves. The result of the analysis is reported as a set of ranked lists (one per group) that is thresholded according to a number of user-supplied configuration parameters.

Larger seed groups (10-100 entities) usually work better than smaller seed groups (1-10 entities), since they provide more connections and a better overall signature. However, small seed groups (single persons in the extreme) can also be analyzed by following their connections to larger depth.

The Group Finder finder uses a logic-based model to combine fragmented evidence to generate the largest possible seed groups from the available information. Those seed hypotheses are then fed into a statistical mutual information model that grows a link graph around the seeds and then ranks the connection strength of those new entities found in the extended graph. Besides the mutual information model, simpler statistical methods such as link counting and connectivity are available as well. Once the mutual information model is done with its analysis, the extended group hypotheses can be deposited back into the KOJAK database and/or output to a file using a variety of report formats.

4 Input Data

The Group Finder needs two separate types of input data, *group seeds* to identify the starting points for group expansion and *link data* which is analyzed to expand the groups.

The data can also be subdivided into different *access classes*, which is based on ease of availability or cost of access. These classes are called *primary data* for data that can be easily and cheaply accessed or *secondary data* for data that is harder to access or more expensive.

The data can also presented in one of several formats.

These options are described in the following sections.

4.1 Types of Data

Note that there is a fundamental assumption underlying all of the data, namely that **different entities have unique names**. One consquence of this is that if data is being transferred from a system that uses catch-all categories such as *unknown* or *other* that refer in general to several unnamed entities, the data must be prepared by generating unique ids for all such unamed entities. This can be accomplished by adding a numeric suffix to any such categories so as to create unique individual names.

4.1.1 Group Seeds

Group seed data describes one or more known or unknown groups with one or more known or hypothesized members for each seed. As mentioned previously, the Group Finder performs seeded group extension, therefore, it must have at least one seed group to start with. The rationale behind this seeded approach is (1) that it allows for a focused search through a potentially huge data space, and (2) that the seeds - if chosen properly - will automatically lead to the detection of groups of interest (e.g., threat groups). This is very important, since in general, every large dataset will contain large numbers of groups that are benign and uninteresting. Since KOJAK's extended groups are formed around seeds, they have a much higher likelihood of being of the same kind as the seed group. With respect to scale, this approach has also already paid off and allowed us to successfully analyze datasets with close to 10,000,000 links. Having said that, there are situations where seed information is not available and we are currently developing methods to generate such seeds automatically if necessary.

Group seed data is generally fairly small. In the simplest case it involves the specification of one or more groups with one or more known members for each of them. For example, the following specifies a seed group in comma-separated-value (CSV) syntax (which is described in more detail below):

```
KnownGroup, SeedGroup1
groupMember, SeedGroup1, MemberA
groupMember, SeedGroup1, MemberB
groupMember, SeedGroup1, MemberC
groupMember, SeedGroup1, MemberD
```

The first line specifies `SeedGroup1` as a group of type `KnownGroup`, and the following lines specify 4 known (or suspected) members of the group. The order of these lines is insignificant. The terms `KnownGroup` and `groupMember` are reserved words interpreted by

the Group Finder. It is possible to use different terms (i.e., a different "ontology"), but then appropriate mapping specifications are needed.

The Group Finder distinguishes between `KnownGroup`'s and `UnknownGroup`'s. The difference is that known groups are assumed to be groups with known identity (e.g., the RAF), and therefore such known groups will never be merged (even though their members might overlap). Unknown groups are groups whose identity we do not know. Unknown groups might turn out to be (part of) a known group or another unknown group. For unknown groups, the Group Finder will attempt to merge them with each other or some known group in case they are similar enough. Note that knowing a group's identity does not mean that we know its members. So, extension of known groups is still a very useful thing to do.

Depending on the domain, it is possible to model some aspects of it and infer additional seed groups or members from other data or link information. This is one of the functions of the logic module which uses a domain ontology and rules to do that. These rules, however, are highly dependent on a particular application domain (such as IET's simulated data generated within the EAGLE program), and will need to be handcrafted specially for each such domain. For example, the file '`kbs/iet-y3-seed-constraints.plm`' encodes such a set of rules and constraints for IET's simulated data. The released version of the Group Finder is generic and domain independent, however, and there is no set of generic, domain-independent seed generation rules. For a typical generic application, it is assumed that seed information will be given explicitly as described above. If a domain is more complex and has additional data that can be exploited, this domain information could be modeled similar to '`kbs/iet-y3-seed-constraints.plm`' to allow the automatic derivation of additional seed information.

4.1.2 Link Data

Link data describes known connections of different types between the entities (individuals) in a dataset. These entities should overlap with the ones provided in the seed groups, but, in general, there will be lots and lots of other irrelevant entities in this data. The Group Finder performs its main work on this link data to find the few (hopefully relevant) individuals that are strongly connected to the seeds. It does so by statistically analyzing and comparing the links from seed individuals to others in the data. In order for this analysis to be significant, there needs to be a sufficient amount of link data - the more, the better. A good example is communication data such as emails or phone calls, where each email or phone call can be viewed as a separate link between two entities. Therefore, there might be hundreds or thousands of links between a pair of entities for each link type. Financial transactions or other transactional data are good candidates also. Here is an example fragment of a link database in CSV syntax:

```
...
phoneCall,In9999,In15132
phoneCall,In9999,In24521
phoneCall,In9999,In27306
telecon,In10005,In11120,In15968,In19067,In2284,In32055,In6166
telecon,In10009,In18419,In28735,In33962,In35311,In36690,In4777,In5866
telecon,In10013,In1325,In24781,In33373,In38164,In38830,In4194,In8312
telecon,In10013,In1325,In24781,In33373,In38164,In38830,In4194,In8312
...
```

The first element in a row specifies the type of link such as `phoneCall` or `telecon`. These link types are not predefined and can be chosen arbitrarily. However, they should represent distinctions that are meaningful. Depending on how the Group Finder was configured, they might have to correspond to relations defined in the underlying ontology.

The remaining arguments are the set of individuals that participated in the particular link, transaction or event. For example, all participants of a telecon can be listed and are then assumed to be pairwise connected by a link of this type. Rows might be duplicated (such as the last one) indicating multiple events or links with the same participants. Again, the order of these rows is insignificant.

The Group Finder can function with data that contains just a single link type, but the mutual information model takes different link types into account, and the more such link-type-differentiated information is available, the better.

Example datasets derived from one of IET's simulated datasets are shipped in the '`data/example*`' subdirectories. They are provided in a comma-separated value (CSV) format that might be the easiest to imitate when applying the Group Finder to new domains. The data files have some more comments describing their structure. They contain about 10,000 entities and 250,000 links connecting them.

The Group Finder can read input data in a variety of formats described in the following sections.

4.2 Data Access Classes

Data can be differentiated based on a general notion of cost of access. If this is not a concern for a particular application environment, undifferentiated data may be used.

4.2.1 Primary Data

Primary Data are easy and cheap to access. They can be used freely for analysis and permit extremely general queries. Link data can be queried without a need to specify any of the participants.

4.2.2 Secondary Data

Seoncdary Data are more difficult or expensive to access. This is taken into consideration by the Group Finder code, which will try to minimize the amount of secondary data used in the analysis. It is also assumed that querying of such data is more limited, in that fully open-ended queries are not permitted. For example, rather than retrieving all link information, the queries might require that at least one of the participants in the link be specified in the query.

4.2.3 Undifferentiated Data

Undifferentiated data describes data which do not have differential access costs. The entire dataset is available for all analysis and no special processing is required.

4.3 Data Formats

The Group Finder supports several data formats, including comma-separated value files, a special IET data format, PowerLoom format and MySQL databases. Of these, the most easily used and adapted is the Comman-separated value format.

4.3.1 Comma-Separated Value Format

The comma-separated value (or CSV) format is probably the easiest to use for applying the Group Finder to a new domain. Each CSV file should have a '`.csv`' extension (or file type) so the Group Finder will use the appropriate data input method. CSV files have the following syntax:

1. Empty lines and lines starting with a `#` character will be ignored
2. All other content lines should have the following syntax:

   `<relation>, <arg1> {, <arg>}*`

   Whitespace following a comma up to the next non-whitespace character will be ignored. Relations and arguments can contain any arbitrary charactes. `,`'s need to be escaped with a `\` character.

A content line with one argument will be interpreted as a type specification, for example:

```
KnownGroup, Group1
```

The relations `groupMember` and `memberAgents` will be interpreted as a binary membership assertion between a group and a group member. For example:

```
groupMember, Group1, Member1
```

All other relations will be interpreted as event-style links. For example,

```
telecon, pA, pB, pC
```

will create a new link (or event) object of type `telecon` and attach `pA`, `pB` and `pC` as the link arguments (or event participants). The exact internal representation will differ dependent on whether this link is loaded into PowerLoom or into the KOJAK EDB. However, the treatment and interpretation will be the same.

4.3.2 IET Format

The IET format was developed by Information Extraction and Transport, Inc. to represent synthetic data. It is a Lisp-style format that uses CycL as the representation language. For example:

```
(isa Gr-16511 ThreatGroup)
(exploitsVulnerabilities Gr-16511 Mo-18843)
(memberAgents Gr-16511 In-17687)
(memberAgents Gr-16511 In-11314)
```

This format is not intended to be a primary input format and only supported to allow the Group Finder to be applied to synthetic data generated by the IET simulators (there are a two different variants for Y2 and Y3 datasets, both are supported). IET format files need to use the file extension `.iet` in order to be properly recognized and handled by the Group Finder.

4.3.3 PowerLoom Format

PowerLoom is the representation and reasoning engine underlying the KOJAK Group Finder. Consequently, all legal PowerLoom input can appear in PowerLoom data files. PowerLoom uses KIF as its input language which is a Lisp-style syntax for predicate logic. For example:

```
(ASSERT (KnownGroup UID-Group-34988))
(ASSERT (groupMember UID-Group-34988 UID-Indvdl-10641))
(ASSERT (groupMember UID-Group-34988 UID-Indvdl-14179))
(ASSERT (groupMember UID-Group-34988 UID-Indvdl-20265))
(ASSERT (groupMember UID-Group-34988 UID-Indvdl-2206))
```

PowerLoom files need to use the file extension `.plm` in order to be properly recognized and handled by the Group Finder. More information about the PowerLoom system and its input syntax can be found at http://www.isi.edu/isd/LOOM/PowerLoom/.

4.3.4 Relational Databases

Finally, data can come directly from a relational database such as MySQL or Oracle. No specific schema is assumed, however, for each schema an appropriate mapping specification/schema/ontology needs to exist that maps the database schema onto the Group Finder ontology. The Group Finder ships with various predefined mappings that support IET Y2, IET Y3 as well as KOJAK evidence databases (EDBs). CSV files can be imported into the KOJAK EDB which is useful for very large link databases or to support multiple analysis runs. Example mapping specifications can be found in the 'kbs' directory, for example, 'kbs/ali-baba-edb-schema.plm', 'kbs/iet-y2-edb-schema.plm', 'kbs/iet-y3-edb-schema.plm' and kbs/kojak-edb-schema.plm. Writing such mapping specifications is somewhat complex and requires some knowledge of the underlying PowerLoom system. It is described in more detail in Chapter 7 [Advanced Configuration], page 33.

5 Configuration

Developing a data analysis tool such as the Group Finder that can be easily and successfully applied to arbitrary real-world datasets is a very challenging task, given that data can come in so many different formats and with so many statistical properties. To address this challenge (at least partially), the Group Finder has a very flexible and highly configurable interface. One might almost say that the Group Finder is configurable to a fault, since there are so many different ways of mapping data, configuring it and running it. However, the generic and example configurations shipping with the distribution should cover a wide variety of situations and make easy departure points when applying the Group Finder to some new dataset.

The Group Finder uses the following configuration mechanisms:

1. A configuration file allows the specification of a large number of parameter such as datasets, connection information, analysis parameters, etc. Many of the available KOJAK commands take these parameters as inputs or as defaults.
2. A set of command line flags that can be used to specify and override some of the configuration parameters specified in the configuration file. This is useful for multiple runs over the same data with slightly different parameter settings, etc.
3. A run script that can execute an arbitrary sequence of KOJAK commands for loading/importing data, ontology and schema information, generating group seeds, extending them, thresholding, reporting, etc.
4. A variety of load scripts that control how and what data is loaded, which ontology and schema information is loaded, etc.
5. Ontology and knowledge base files that can leverage the full power of the PowerLoom knowledge representation and reasoning system to represent specifics of a particular domain, map to a particular external database schema, etc.

5.1 Configuration File

By default, the Group Finder looks for the file '`configuration.dat`' in the KOJAK software's installation directory to load its configuration information. Alternatively, the `-c` command-line option can be used to specify a different configuration file. It is an error if no configuration file can be found and none was specified on the command line.

There are a set of configuration files available in the '`config`' directory. By default the Group Finder uses the '`config/example1.dat`' configuration which runs the Group Finder on one of the example datasets that ship with the release. This file and its associated load and run scripts should be a good departure point for generating a configuration for some new dataset.

Configuration information is supplied in a Java-style property file with lines of the following form:

```
<parameterName>  =<parameterValue>
<parameterName> +=<parameterValue>
```

Empty lines or lines starting with a # character will be ignored. Whitespace leading up to the = character will be ignored, whitespace following it will become part of the parameter value. The += syntax allows the specification of list-valued parameters, for example, to

assign a set of input files to the `Data` parameter. Both parameter names and values are treated case-sensitively.

The configuration file can contain arbitrary user-defined parameters which can be accessed in run and load scripts via the functions `get-parameter`, `set-parameter`, and `add-parameter-value`. There are a multitude of built-in parameters that control the behavior of the Group Finder which are explained further below.

Various built-in parameters take filenames as arguments. These filenames can be absolute or relative physical pathnames (using syntax appropriate for the underlying operating system), or, they can be *logical pathnames* that are platform neutral and that can be translated automatically by KOJAK into the appopriate physical form. Logical pathname syntax is derived from Common-Lisp. Each logical pathname starts with a logical host (terminated by a colon) followed by a set of directory components (separated by ;) and ending in a filename with an optional extension. KOJAK logical pathnames need to be of the following form:

```
KJ:{<dirname>;}*[<basename>][.<extension>]
```

For example,

```
KJ:scripts;ali-baba-load-ontology.plm
```

which would translate into

```
C:\Program Files\kojak-2.2.0\scripts\ali-baba-load-ontology.plm
```

if the value of `KojakRootDirectory` (see below) is defined as '`C:\Program Files\kojak-2.2.0\`'. This means the logical host `KJ` is simply replaced by the value of `KojakRootDirectory` and the logical pathname separator ; is substituted with the pathname separator appropriate for the underlying OS.

5.1.1 General Options

`KojakRootDirectory` [Parameter]

This should be an absolute physical path pointing to the KOJAK directory (including a final directory delimiter). A relative path such as '`./`' will work also but requires that the Group Finder is run from the KOJAK directory (since the `run-kojak` scripts always change to the KOJAK directory relative pathnames are fine unless the KOJAK executables are invoked manually somehow). Logical pathnames starting with the `KJ` logical host will use this directory as their root.

`KojakDB` [Parameter]

The internal database used by KOJAK to store hypotheses, intermediate results, data and configuration information. Must be either a simple ODBC DSN or an ODBC or JDBC connection string (similar to `Data` specifications which see). If it is a DSN all necessary connection information must be specified in '`.odbc.ini`' (see Section 2.4.2.1 [.odbc.ini File], page 6). For example:

```
KojakDB =KOJAK
```

If it is an ODBC connection string, everything necessary can be specified, but partial connection strings are also possible in which case missing information is filled in from '`.odbc.ini`'. For example:

```
KojakDB =DSN=KOJAK;DB=kojak;SERVER=blackcat;UID=scott;
KojakDB =DSN=KOJAK;DB=kojak;
KojakDB =DSN=KOJAK;USER=KOJAK;DB=blackcat.isi.edu/oracledb;
```

If it is a JDBC connection string, all connection information must be specified. JDBC connection string also varies for different database systems and JDBC drivers. Here are two examples for the Oracle and MySQL drivers currently supported by KOJAK:

```
KojakDB =jdbc:mysql://blackcat:3306/kojak?user=scott&password=secret
KojakDB =jdbc:oracle:thin:KOJAK/secret@blackcat.isi.edu:1521:oracledb
```

Note that for Oracle databases the KOJAK schema name is the USER name while for MySQL it is the name of the database (DB). `KojakDB` can be set or overridden the command line options `-k` and `--kojak-db`.

`LogLevel` [Parameter]

Controls the amount of log output generated during an analysis run. Needs to be one of `none/low/medium/high`. This can also be specified via the command line options `-l` and `--log-level`.

`DBUser` [Parameter]

Default user name to use for connection information if a database is simply supplied as a DSN but not a connection string. If this is not provided, user information will need to be specified for the particular DSN in `.odbc.ini`, it will not be inherited from the [Default] data source.

`DBPassword` [Parameter]

Default password to use for connection information if a database is simply supplied as a DSN but not a connection string. If this is not provided, password information will need to be specified for the particular DSN in `.odbc.ini`, it will not be inherited from the [Default] data source.

5.1.2 Input Specifications

`Data` [Parameter]

`Data` can be used to specify data sources to be analyzed. Sources can either be files in various formats or evidence databases accessible via ODBC or JDBC. Currently supported file formats are comma-separated value files, IET report-format files or PowerLoom files (see Chapter 4 [Input Data], page 9). Appropriate file extensions must be used to indicate the type of file, for example, `data.csv`, `data.iet` or `data.plm`. Databases can be specified as a DSN or ODBC or JDBC connection string (see `KojakDB` for more details). This parameter can also be specified by pointing to another parameter such as `KojakDB` whose value will then be used. `Data` can have multiple values specified via the `+=` syntax. For example:

```
Data +=KJ:data;example1;ds1-group-seeds-all.csv
Data +=KojakDB
```

Data sources can also be specified via the -d or –data command line options which can be given multiple times.

The value(s) of the `Data` parameter will be used by default by the `load-data` command to load data (unless overridden by actual parameters). If a database source is

specified, the script specified by `LoadDataScript` will be used to load data from that database.

`DisabledData` [Parameter]
`DisabledData` is used by the Kojak interface code to record data specifications which are not currently used in the analysis, but which are made available for selection in the interface. It uses the same format as `Data`.

`PrimaryData` [Parameter]
Similar to `Data` but uses the value of `LoadPrimaryDataScript` to load database data sources. These data sources are not loaded by default by `load-data` but can be specified as follows:

```
(load-data :data-source "PrimaryData")
```

Primary data can also be specified via the `-pd` or `--primary-data` options.

`SecondaryData` [Parameter]
Similar to `Data` but uses the value of `LoadSecondaryDataScript` to load database data sources. These data sources are not loaded by default by `load-data` but can be specified as follows:

```
(load-data :data-source "SecondaryData")
```

Secondary data can also be specified via the `-sd` or `--secondary-data` options.

`DatasetName` [Parameter]
Serves as a single logical name for the supplied input data and all results generated from it. It can also be specified via the `-n` or `--dataset-name` command-line options. Dataset names are used as metadata for results deposited in the hypothesis tables of the KOJAK DB.

`DatasetType` [Parameter]
Indicates what kind of data we are looking at; mainly used to flag simulated data generated by IET, since some special assumptions apply there; if this is not given, it is guessed from some of the Data specifications.

```
DatasetType =IET_2004
```

It can also be specified via the `-t` or `--dataset-type` options.

5.1.3 Script Specifications

The Group Finder uses a variety of scripts to load data, ontologies, schema information, execute commands, etc. The following parameters allow customization of these scripts.

`LoadOntologyScript` [Parameter]
The default script used by `initialize-kojak` and `load-kojak-ontology` to load ontology information into PowerLoom.

`LoadEDBSchemaScript` [Parameter]
The default script used by `initialize-kojak` and `load-edb-schema` to load evidence database schema information into PowerLoom.

`LoadDataScript` [Parameter]
The default script used by `load-data` to load RDBMS data sources specified in the `Data` configuration parameter.

The following scripts are primarily relevant to IET data sources and somewhat obsolete. They are the default scripts used when an RDBMS data source is loaded from a `PrimaryData`, `SecondaryData`, `PrimaryLDData` or `PrimaryPLData` specification.

`LoadPrimaryDataScript` [Parameter]

`LoadSecondaryDataScript` [Parameter]

`LoadPrimaryLDDataScript` [Parameter]

`LoadPrimaryPLDataScript` [Parameter]

`RunKojakScript` [Parameter]
The top-level KOJAK run script. Commands in this script are executed when the `main` function of the KOJAK Group Finder is run in non-interactive mode, or when the `run-kojak` command is executed. This can also be specified via the `-s` or `--run-script` options.

5.1.4 Analysis Specifications

`RunID` [Parameter]
Marks a particular configuration used to analyze `DatasetName` when the configuration gets saved to the KOJAK database. If none is supplied a new one is generated automatically. This is useful to track different configuration settings across multiple analysis runs. It can also be specified via the -r or –run-id options.

`RunPrefix` [Parameter]
Used as a prefix when Kojak generates an automatic `RunID`. It can also be specified via the -rp or –run-prefix options.

`ClearOldHypotheses` [Parameter]
If true, hypotheses generated in previous runs will be deleted from the KOJAK database before any new ones are generated.

`ExpansionDepth` [Parameter]
Depth of graph expansion from seed groups to get extended groups. **IMPORTANT**: due to graph fanout this value has a big impact on run time. Use values larger than 1 only for cases with very small seed groups (e.g., 1-5 members) and increase it only one step at a time if needed. It can also be specified via the `--depth` option.

`ExpansionLinkTypes` [Parameter]
The default link types that should be considered by the `expand-groups` command when expanding seed groups and computing mutual information between entities. These should correspond to the relevant link types loaded and accessed from the evidence database (for example, `telecon`, `phoneCall`, etc.). If no link types are specified the default is `anyLink`. This is a multi-valued parameter.

`DisabledExpansionLinkTypes` [Parameter]

`DisabledExpansionLinkTypes` is used by the Kojak interface code to record link types which are not currently used in the analysis, but which are made available for selection in the interface. It uses the same format as `ExpansionLinkTypes`.

`ExpansionMethods` [Parameter]

The default strength computation methods considered by `expand-groups`. These are used to compute connection strength measures between entities and seed groups. This is a multi-valued parameter whose values are treated case-insensitively. Legal values are the following:

- `MI`: computes the mutual information value between an entity and a seed group member based on the complete set of links between two entities.
- `Group-MI`: views the whole seed group as a single individual and computes the MI between this "group individual" and an entity based on all the links between the entity and the "group individual".
- `MI-or-Group-MI`: computes both `MI` and `Group-MI` values.
- `Link-Count`: Computes link counts between an entity and seed group members.
- `Connectivity`: Computes the number of seed group members connected to an entity.

The set `Connectivity`, `Link-Count` and `Group-MI` form the default.

NOTE: this parameter will eventually go away, since which expansion methods are required can be inferred from the `ExpansionMeasures` described below.

`ExpansionMeasures` [Parameter]

The default strength computation measures that should be used by `expand-groups`. Each strength computation method generates a variety of results such as maximums, averages, etc. The set of expansion measures selects a subset of those result measures and averages them to generate a single result measure (this averaging is called "bagging"). This is a multi-valued parameter that can take values from the following set (values are treated case-insensitively):

- `max-MI`: computes the maximum MI between an entity and any of the seed group members as the connection strength to the seed group.
- `max-MI-correlation`: computes `max-MI` and normalizes it as a correlation value.
- `sum-MI`: sums up the MIs between an entity and each seed group member.
- `group-MI`: computes the MI between an entity and the set of seed group members when viewed as a single super individual (combining all their links).
- `group-MI-correlation`: computes `group-MI` and normalizes it as a correlation value.
- `MI-or-group-MI`: computes the maximum of the `max-MI` and `group-MI` measures.
- `MI-or-group-MI-correlation`: computes `MI-or-group-MI` and normalizes it as a correlation value.
- `max-count`: computes the maximum link count between an entity and any of the seed group members as the connection strength to the seed group.

- `sum-count`: sums up the link counts between an entity and each seed group member.
- `connectivity`: computes the number of seed members an entity is connected to.

Each of the above measures will be normalized to a value between 0 and 1 to allow proper averaging. The best performing measures in experiments performed so far are `sum-count`, `group-MI` and `connectivity` which form the default set if no measures are specified.

Boosting can be useful in cases where we have very few seed members for a group. Instead of running the group expansion only once, we take a small number of the top new members, add them to the seed group (i.e., "boost" the number of seeds) and run again. This is useful if the top members have very high likelihood of being correct. The parameters below control how and how often boosting is done. All of these values can be overridden when calling `expand-groups`.

`BoostEnabled` [Parameter]
A boolean parameter that determines whether boosting is performed or not. If this is not explicitly specified, then (for backward compatibility) boosting will be done if either `BoostMaxCycles` or `BoostMinSeeds` has a value.

`BoostMaxCycles` [Parameter]
Perform at most this many boosting cycles even if the targets have not been reached (leave this undefined if boosting is controlled by `BoostMinSeeds`):

`BoostMinSeeds` [Parameter]
Boost until seed groups have at least that many members; if both this and `BoostMaxCycles` are undefined boosting is disabled.

`BoostFactor` [Parameter]
In each boost iteration, increase the number of seed members by this factor.

`BoostMinStep` [Parameter]
In each boost iteration, increase the number of seed members at least by that many members, even if `BoostFactor` tells us otherwise.

5.1.5 Output Specifications

`ReportFile` [Parameter]
File to which Group Finder results should be reported. If no `ReportFormat` (which see) is specified, the file extension determines which format to use. The report file can also be specified via the `-o` or `--report-file` options. Note that results can also be output to the KOJAK database by using the `db-save-groups` command.

`ReportDirectory` [Parameter]
Directory path to prepend to `ReportFiles` which is useful for multiple runs via scripts.

`ReportFormat` [Parameter]
Format to use for report files generated by `report-groups`. Supported formats are `IET-Report` (equivalent to `IET-Y3-Report`), `IET-Y2-Report` and `CSV-Table`. If not

specified, the format will be guessed from the extension of `ReportFile`. A `.csv` extension means use the `CSV-Table` format, everything else is interpreted as `IET-Report`.

`ReportMemberWeights` [Parameter]
If true, membership weights will be reported for each group member.

`WeightDecimals` [Parameter]
The number of decimals behind the decimal point that should be used to print a weight in fixed-point format to the output file (defaults to 6).

`LastRunCutoff` [Parameter]
The computed number of to-be-reported members will usually wind up somewhere within a run of members with equal membership weights. Cutting right there would arbitrarily include and exclude members that are really indistinguishable by their membership weight. `LastRunCutoff` determines how this last run should be cut. Legal values are `Inclusive` which keeps the whole run, `Exclusive` which cuts at the end of the previous run, `Hard` which cuts right at the computed threshold point or `Closest` which cuts at the closest endpoint of the run. The default is `Inclusive`. If the last run is also the first run, it will always be reported in full unless the value is `Hard`; if we fall on the last member of a run the run will be reported in full even if the cutoff value is `Exclusive`.

`ReportNames` [Parameter]
If true, group and member print names will be looked up via the `GROUPS/nameString` relation and printed instead of the IDs or names identifying them in evidence. This is useful if groups and members are identified via unique but unreadable IDs in evidence and more readable reports need to be generated. This will only work if appropriate `GROUPS/nameString` assertions are available in the evidence, or if there are mapping rules that can retrieve them from an EDB.

The following parameters provide some basic control for thresholding of extended groups. Reported groups are sorted in descending order of membership weight and thresholding controls where this list is cut. Automatic thresholding is still not very well supported by the Group Finder, since there are no clear transitions indicating where true members stop and non-members start. Besides the very basic controls provided below, specialized functions can be written that provide thresholding appropriate for a particular domain (see the `threshold-groups` command).

`ReportGroupFraction` [Parameter]
Fraction of individuals in the extended graph that should be reported in the result. Includes seed members; 1.0 reports the whole group. This can also be specified via the `-gf` or `--report-group-fraction` options.

`ReportMinMembers` [Parameter]
Minimum number of group members to report (including seeds).

`ReportMaxMembers` [Parameter]
Maximum number of group members to report (including seeds).

5.2 Command-Line Options

What data is analyzed by KOJAK and how it analyzes it is controlled by a large number of parameters given in the 'configuration.dat' configuration file in the KOJAK directory. See the previous section for documentation of the various available configuration parameters. Many of these parameters can also be specified and overridden from the Unix command line. Here is the full list of available command-line options with a brief description of their functionality:

```
run-kojak [{-c|--configuration-file} FILE] [-i|--interactive]
          [{-l|--log-level"} LEVEL] [{-d|--data} DATASPEC]*
          [{-pd|--primary-data} DATASPEC]*
          [{-sd|--secondary-data} DATASPEC]*
          [{-k|--kojak-db} DBSPEC] [{-n|--dataset-name} NAME]
          [{-t|--dataset-type} TYPE] [{-r|--run-id} ID]
          [{-r|--run-prefix} PREFIX]
          [{-s|--run-script} FILE] [--depth N] [{-o|--report-file} FILE]
          [{-gf|--report-group-fraction} FRACTION]
```

'`--configuration-file` *FILE*'
'`-c` *FILE*' Specifies *FILE* to be used as the configuration file. By default, 'configuration.dat' in the KOJAK directory is used. If there is no valid configuration file specified or available, the Group Finder terminates immediately.

'`--interactive`'
'`-i`' Run the Group Finder interactively by bringing up the PowerLoom command loop. At that point any PowerLoom or KOJAK command can be executed. Calling the `run-kojak` command will run the Group Finder end-to-end.

'`--log-level` *LEVEL*'
'`-l` *LEVEL*'
Sets the `LogLevel` parameter to *LEVEL*.

'`--data` *DATASPEC*'
'`-d` *DATASPEC*'
Sets the `Data` parameter to *DATASPEC*. This option can be supplied multiple times.

'`--primary-data` *DATASPEC*'
'`-pd` *DATASPEC*'
Sets the `PrimaryData` parameter to *DATASPEC*. This option can be supplied multiple times.

'`--secondary-data` *DATASPEC*'
'`-sd` *DATASPEC*'
Sets the `SecondaryData` parameter to *DATASPEC*. This option can be supplied multiple times.

'`--kojak-db` *DBSPEC*'
'`-k` *DBSPEC*'
Sets the `KojakDB` parameter to *DBSPEC*.

'`--dataset-name` *NAME*'
'`-n` *NAME*' Sets the `DatasetName` parameter to *NAME*.

'`--dataset-type` *TYPE*'
'`-t` *TYPE*' Sets the `DatasetType` parameter to *TYPE*.

'`--run-id` *ID*'
'`-r` *ID*' Sets the `RunID` parameter to *ID*.

'`--run-prefix` *PREFIX*'
'`-rp` *PREFIX*'
Sets the `RunPrefix` parameter to *PREFIX*.

'`--run-script` *FILE*'
'`-s` *FILE*' Sets the `RunKojakScript` parameter to *FILE*.

'`--depth` *N*'
Sets the `ExpansionDepth` parameter to *N*.

'`--report-file` *FILE*'
'`-o` *FILE*' Sets the `ReportFile` parameter to *FILE*.

'`--report-group-fraction` *FRACTION*'
'`-gf` *FRACTION*'
Sets the `ReportGroupFraction` parameter to *FRACTION*.

'`--define` *PAR*`=`*VAL*`, --define` *PAR*`+=`*VAL*'
'`-D` *PAR*`=`*VAL*`, -D` *PAR*`+=`*VAL*'
Can be used to set an arbitrary configuration parameter *PAR* to value *VAL*. Both = and += syntax are supported.

6 Running the KOJAK Group Finder

After it has been appropriately configured for a dataset you can run KOJAK simply by calling the following script in the KOJAK directory:

```
% ./run-kojak
```

This will read data from specified data files and/or databases, extend the given seed groups and write results to output files and/or the KOJAK database.

While the number of configuration options, scripts and possibilities might seem daunting, the setup provided with the Example 1 dataset should work more or less out of the box for many common cases. All that needs to be done is to generate link data files that have the appropriate link types for the new domain and adapting the call to `extend-groups` in the run script (or the setting of `ExpansionLinkTypes`) to use the appropriate link types for this new domain. In the next section we provide a detailed run-through for this example.

6.1 Example Run 1

Below we provide annotated output from running the Group Finder end-to-end on the Example 1 dataset that can be found in the '`data/example1/`' directory. The default configuration file '`configuration.dat`' in the KOJAK directory is setup for this example dataset. Therefore, simply running

```
% ./run-kojak
```

from the command line will run this example. In the description below we repeat the commands from the corresponding run script '`scripts/example1-run-kojak.plm`' and format the output to make it read better in the manual. Commands were typed interactively at the PowerLoom `|=` prompt. The sequence of commands, their parameter settings and their generated output should otherwise be identical. The result is written to '`example1-report.csv`' in the KOJAK directory. This run should take less than a minute on a state-of-the-art PC desktop. Running on previously imported data without calling `import-data-into-edb` again should only take a couple of seconds.

This dataset was derived from one of IET's Y2.5 evaluation datasets. It is represented in comma-separated value (CSV) format and very simple and generic. It should be a good starting point to use for preparing a new dataset and applying the KOJAK Group Finder to it.

All commands used in this script are documented in more detail in Section Chapter 8 [KOJAK Commands], page 62. The script uses a Lisp-ish input syntax which is the standard input syntax used by the underlying PowerLoom system. Knowledge of PowerLoom should not be required, but see `http://www.isi.edu/isd/LOOM/PowerLoom/index.html` for more information if necessary. Commands take zero or more required arguments and a set of options specified as keyword/value pairs. Default values for unspecified options are often taken from the configuration file.

The first two commands are a PowerLoom preamble and set up package and module information. All run scripts should start with these two commands.

```
|= (IN-PACKAGE "STELLA")

|= (IN-MODULE "KOJAK")
```

We start by initializing the KOJAK system. This will load ontology and EDB schema information using the scripts specified in the configuration. In this case we avoid loading the EDB schema information until later by setting `:load-edb-schema?` to `FALSE` (the reason for this is explained below). We also force clearing of old group hypotheses in the KOJAK DB regardless of what was specified in the configuration:

```
|= (initialize-kojak :load-edb-schema? FALSE :clear-old-hypotheses? TRUE)
[2005-APR-09 16:01:38.000 KOJAK] Loading KOJAK Ontology...
[2005-APR-09 16:01:38.000 KOJAK] Clearing out old hypotheses and configurations...
[2005-APR-09 16:01:38.000 SDBC] Connect via ODBC: DSN=KOJAK;DB=kojak_test;UID=root;
```

Print all current configuration settings to standard output which is useful for logging and archiving settings for a run:

```
|= (print-parameters)
------------------------------------------------------------
# Loaded KOJAK Configuration
ChunkSize = 0.1
ClearOldHypotheses = TRUE
Data = ("KJ:data;example1;ds1-group-seeds-all.csv" "KojakDB")
DatasetName = Example1
DatasetType = IET_2004
ExpansionDepth = 1
ExpansionLinkTypes = ("phoneCall" "telecon")
ExpansionMeasures = ("Sum-Count" "Group-MI" "Connectivity")
ExpansionMethods = ("MI" "Link-Count")
KojakDB = DSN=KOJAK;DB=kojak_test;UID=root;
KojakOutputDirectory = /tmp/
KojakRootDirectory = /home/hans/Projects/powerloom/sources/kojak2/group-finder/
LoadDataScript = KJ:scripts;load-data.plm
LoadEDBSchemaScript = KJ:scripts;load-edb-schema.plm
LoadOntologyScript = KJ:scripts;load-ontology.plm
LoadPrimaryDataScript = KJ:scripts;load-primary-data.plm
LoadPrimaryLDDataScript = KJ:scripts;load-primary-ld-data.plm
LoadPrimaryPLDataScript = KJ:scripts;load-primary-pl-data.plm
LoadSecondaryDataScript = KJ:scripts;load-secondary-data.plm
LogLevel = medium
MIWeakLinksIterations = 0
MIWeakLinksSaveRate = 0.5
ReportDirectory =
ReportFile = ./example1-report.csv
ReportFormat = CSV-Table
ReportGroupFraction = 1.0
ReportMaxMembers = 100
ReportMemberWeights = TRUE
ReportMinMembers = 0
RunID = Run_2005-04-09 16:01:10.000
RunKojakScript = KJ:scripts;example1-run-kojak.plm
Threshold = 0.1
```

```
boostFactor = 2.0
boostMinStep = 4
------------------------------------------------------------
```

kojak-log can be used to print various logging information to standard output depending on the current logging level. The first argument specifies the minimum log level at which the information should be printed:

```
|= (kojak-log :low "Executing KOJAK over dataset " (get-parameter "DatasetName") "..."
[2005-APR-09 16:01:42.000 KOJAK] Executing KOJAK over dataset Example1...

|= (kojak-log :low "Results will be reported to file " (get-parameter "ReportFile" "re
[2005-APR-09 16:01:43.000 KOJAK] Results will be reported to file ./example1-report.cs
```

Before we start loading any data, we import link data into the KOJAK EDB. Processing the data directly from the EDB is the most efficient way of dealing with large datasets. After the data has been imported, it can be analyzed multiple times with different parameters, group seeds, etc. without having to re-import it. To use it in this way, comment the import-data-into-edb command in this script after its first use. Otherwise, data will be cleared and re-imported every time this script is run. The :data-source option specifies from what file to import the data. If not specified, all CSV files specified in the Data parameter will be loaded. We also clear any pre-existing EDB content by setting :clear-EDB? to TRUE. Without that, data would be added incrementally which allows to import data spread over multiple files.

```
|= (import-data-into-edb :data-source "KJ:data;example1;ds1-links.csv" :clear-EDB? tru
[2005-APR-09 16:01:45.000 KOJAK] Importing data into KOJAK edb...
[2005-APR-09 16:01:45.000 KOJAK] Loading CSV evidence file KJ:data;example1;ds1-links.
[2005-APR-09 16:01:45.000 KOJAK]   Storing 10002 tuples into EDB...
[2005-APR-09 16:01:46.000 KOJAK]   Storing tuples into EDB done.
[2005-APR-09 16:01:46.000 KOJAK]   Storing 10000 tuples into EDB...
[2005-APR-09 16:01:48.000 KOJAK]   Storing tuples into EDB done.
    ..............
[2005-APR-09 16:02:45.000 KOJAK]   Storing tuples into EDB done.
[2005-APR-09 16:02:45.000 KOJAK]   Storing 10009 tuples into EDB...
[2005-APR-09 16:02:48.000 KOJAK]   Storing tuples into EDB done.
[2005-APR-09 16:02:48.000 KOJAK]   Storing 4307 tuples into EDB...
[2005-APR-09 16:02:49.000 KOJAK]   Storing tuples into EDB done.
```

Now we are ready to load EDB schema information which loads link types and associated information from data currently loaded in the EDB. Had we done that before data was imported, we might get incorrect information left over from previously imported data. Schema information is by default loaded with the script specified in the configuration file.

```
|= (load-edb-schema)
[2005-APR-09 16:02:52.000 KOJAK] Loading EDB schema information from DSN=KOJAK;DB=koja
[2005-APR-09 16:02:53.000 PL] Redefining the logic object named edb
[2005-APR-09 16:02:53.000 PL] Processing check-types agenda...
[2005-APR-09 16:02:53.000 SDBC] Connect via ODBC: DSN=KOJAK;DB=kojak_test;UID=root;
[2005-APR-09 16:02:53.000 PL] Processing check-types agenda...
```

Now we load all datasets specified with the `Data` parameter in the configuration file. In this example, we load group seed information directly into PowerLoom and link data from the `KojakDB` (which we just filled in the import step above). EDB data is loaded by pointing the PowerLoom `edb` object to the KOJAK DB and then loading the `LoadData` script. This load script in fact loads very little data (only any seed group information in case that was loaded into the EDB - not the case in our example), plus some mapping rules that will allow us to access link data on demand in the group expansion step.

```
|= (load-data)
[2005-APR-09 16:02:56.000 KOJAK] Loading data...
[2005-APR-09 16:02:56.000 KOJAK] Loading CSV evidence file KJ:data;example1;ds1-group-
[2005-APR-09 16:02:56.000 KOJAK]   Finished loading; generated 69 assertions.

[2005-APR-09 16:02:56.000 PL] Processing check-types agenda...
[2005-APR-09 16:02:56.000 SDBC] Disconnecting ODBC connection: DSN=KOJAK;DB=kojak_test
```

Now we are ready to execute phase 1 which will generate seed groups. In this example this phase is very simple, since all seed groups are explicitly specified. IET datasets allow some more sophisticated processing and inference of seed groups and additional seed members given other information.

```
|= (kojak-log :low "Running KOJAK phase 1: group seed generation...")
[2005-APR-09 16:02:58.000 KOJAK] Running KOJAK phase 1: group seed generation...
```

`retrieve-groups` finds all groups that were asserted or are inferable and generates special group objects for them containing all member information, etc. These group objects can be accessed with commands such as `get-all-groups` and friends and are required as inputs to various other commands.

```
|= (retrieve-groups)
[2005-APR-09 16:03:00.000 KOJAK] Found 3 known groups...
[2005-APR-09 16:03:00.000 KOJAK]   Looking for group memberships...
[2005-APR-09 16:03:00.000 KOJAK]   Done retrieving groups and members.
(|GROUP|DATASET/UID-Group-6704 |GROUP|DATASET/UID-Group-8866
 |GROUP|DATASET/UID-Group-34988)
```

`retrieve-events` finds all events that were asserted and for which we have some participant information (none available in this example).

```
|= (retrieve-events)
[2005-APR-09 16:03:03.000 KOJAK] Found 0 events...
[2005-APR-09 16:03:03.000 KOJAK]   Looking for event participants...
[2005-APR-09 16:03:03.000 KOJAK]   Done retrieving events and participants.
()
```

Next we hypothesize unknown groups based on known events where some partial participant information is known. Again, this is not relevant for this example and only exercised in IET datasets so far.

```
|= (hypothesize-unknown-groups :mode :conservative)
[2005-APR-09 16:03:04.000 KOJAK] Hypothesizing unknown group seeds...
[2005-APR-09 16:03:04.000 KOJAK]   Hypothesized 0 unknown group seeds.
()
```

Print out various information we found for logging purposes:

```
|= (kojak-log-objects :medium "Found " (get-nof-objects :AGENT) " seed members:" 'eol
                    (get-all-objects :agent))
[2005-APR-09 16:03:06.000 KOJAK] Found 63 seed members:

|AGENT|DATASET/UID-Indvdl-26714
|AGENT|DATASET/UID-Indvdl-5458
  ..............
|AGENT|DATASET/UID-Indvdl-7146
|AGENT|DATASET/UID-Indvdl-39046

|= (kojak-log-objects :medium "Found the following known threat groups:" 'eol
                    (get-all-known-groups))
[2005-APR-09 16:03:08.000 KOJAK] Found the following known threat groups:

[GROUP DATASET/UID-Group-34988
  known members: (|AGENT|DATASET/UID-Indvdl-27264 |AGENT|DATASET/UID-Indvdl-13433
                  |AGENT|DATASET/UID-Indvdl-15781 |AGENT|DATASET/UID-Indvdl-31658
                  ..............
                  |AGENT|DATASET/UID-Indvdl-27990 |AGENT|DATASET/UID-Indvdl-16643
                  |AGENT|DATASET/UID-Indvdl-26958 |AGENT|DATASET/UID-Indvdl-14179)
  inferred members: ()]
[GROUP DATASET/UID-Group-8866
  known members: (|AGENT|DATASET/UID-Indvdl-4816 |AGENT|DATASET/UID-Indvdl-11759
                  |AGENT|DATASET/UID-Indvdl-7961 |AGENT|DATASET/UID-Indvdl-18750
                  ..............
                  |AGENT|DATASET/UID-Indvdl-26958 |AGENT|DATASET/UID-Indvdl-26888
                  |AGENT|DATASET/UID-Indvdl-30764 |AGENT|DATASET/UID-Indvdl-1774)
  inferred members: ()]
[GROUP DATASET/UID-Group-6704
  known members: (|AGENT|DATASET/UID-Indvdl-715 |AGENT|DATASET/UID-Indvdl-16831
                  |AGENT|DATASET/UID-Indvdl-11247 |AGENT|DATASET/UID-Indvdl-30806
                  ..............
                  |AGENT|DATASET/UID-Indvdl-30510 |AGENT|DATASET/UID-Indvdl-24042
                  |AGENT|DATASET/UID-Indvdl-21092 |AGENT|DATASET/UID-Indvdl-23282)
  inferred members: ()]

|= (kojak-log-objects :medium "Hypothesized the following unknown group seeds:" 'eol
                    (get-all-unknown-groups))
[2005-APR-09 16:03:09.000 KOJAK] Hypothesized the following unknown group seeds:

|= (kojak-log-objects :medium "Found the following events:" 'eol
                    (get-all-objects :event))
[2005-APR-09 16:03:11.000 KOJAK] Found the following events:
```

The KOJAK database is not only used to store evidence data but also to store configuration information as well as resulting groups or intermediate hypotheses. Storing this information in the database is optional and only required if it is needed by some other com-

ponent or to preserve it for some later run or analysis. For example, we can save seed groups here and reload them later for some different analysis run. Originally, this mechanism was used to communicate seed information to a mutual information component written in Matlab. However, it turns out that using the database as a kind of blackboard for storing and communicating results is useful in many situations.

```
|= (kojak-log :low "Storing group seeds and configuration in KOJAK DB...")
[2005-APR-09 16:03:14.000 KOJAK] Storing group seeds and configuration in KOJAK DB...
```

Stores all current parameter settings in the `configuration` table of the KOJAK DB under the current 'RunID'. This configuration could be loaded later with `db-load-configuration` (NOT YET IMPLEMENTED).

```
|= (db-save-configuration)
```

Save known group seeds in the `hypothesis` table with `:source` and `:category` meta-information. The values used in these fields can be arbitrary and the ones used here only have historic significance. `:source` specifies the source module and `:category` the kind of hypothesis we are dealing with. These can be used later to select only specific hypotheses to load. The `hypothesis` table schema developed "organically" and has some deficiencies that will be eliminated in future versions. For example, if one wants to store hypotheses from the same dataset but multiple analysis runs, one would have encode the `RunID` manually in one of the `:dataset`, `:source` or `:category` fields.

```
|= (db-save-groups (get-all-known-groups)
                   :dataset (get-parameter "DatasetName")
                   :source "AV"
                   :category "P1")

|= (db-save-groups (get-all-unknown-groups)
                   :dataset (get-parameter "DatasetName")
                   :source "AV"
                   :category "T1")
```

Now we are ready to run phase 2 which forms the core of the KOJAK group detection process. In this step we extend seed groups using all available link data. We first grow a graph from the group seeds to all individuals reachable via the available link data. This graph extension can be done to a certain depth such as 2 or 3. The default depth is 1, since depending on fan-out growing to deeper depth might bring in very large amounts of data. Then we take the individuals found in the graph extension step and compute connection strength between them and the seed group given a variety of methods. A core method used is mutual information (MI), but we also have other methods available such as link counting or connectivity to group seeds. We can use multiple methods and measures and combine them using "bagging" which usually yields better performance than just using a single method or measure. Given these connection strength measures we can then compute a ranked list of extended members where those most strongly connected to the seed group are listed at the top.

```
|= (kojak-log :low "Running KOJAK phase 2: group extension...")
[2005-APR-09 16:03:23.000 KOJAK] Running KOJAK phase 2: group extension...
```

`extend-groups` takes a set of seed groups and a large number of optional parameters controlling the group expansion process. Most of these parameters take defaults from the configuration file, but we explicitly specify them here for documentation purposes.

`:mode` specifies whether groups will be expanded one at a time (`:individual`) or collectively (`:collective`). Collective expansion brings in more data which can improve quality but also increase run-time, since much more data might be loaded in at once. Boosting is not yet supported in `:collective` mode.

`:link-types` specifies the link types that should be considered to expand the graph and compute connection strength. These link types must match the ones in the link data. Using different link types particularly benefits the MI computation, since it can differentiate links of different types. A special `anyLink` type is also supported for EDB data, which treats all links as the same.

`:methods` specifies which strength computation methods should be used. Currently supported are link counting, MI and connectivity (which is computed implicitly as part of link counting). See the documentation of the `ExpansionMethods` parameter for more details.

Each strength computation method produces a number of associated measures such as maximum and average values, etc. See the documentation of `ExpansionMeasures` for a full list. Below we specify which measures we want to use for ranking members. The specified measures need to correspond to the `:methods` specified above. If more than one measure is specified, they will be averaged ("bagging").

Only keep the :top-N extended members in the resulting expanded groups (which is a simple form of preliminary thresholding). There are also various parameters for boosting control (boosting is disabled in this invocation). See Section 5.1 [Configuration File], page 14 for more information on boosting and associated parameters.

```
|= (extend-groups (get-all-groups)
                  :mode :individual
                  :link-types '("phoneCall" "telecon")
                  :methods '(:connectivity :link-count :group-mi)
                  :result-measures '(:sum-count :group-mi :connectivity)
                  ;:top-N 500
                  ;:boost-min-seeds 20 ;; no boosting by default
                  :boost-factor 1.75
                 :boost-min-step 3)
[2005-APR-09 16:03:36.000 KOJAK] Extending groups with options: (:MODE :INDIVIDUAL :LI
[2005-APR-09 16:03:36.000 KOJAK] Extending group UID-Group-34988 with 22 seeds...
[2005-APR-09 16:03:36.000 SDBC] Connect via ODBC: DSN=KOJAK;DB=kojak_test;UID=root;
[2005-APR-09 16:03:36.000 KOJAK]   Retrieved a total of 317 phoneCall links from 22 se
[2005-APR-09 16:03:38.000 KOJAK]   Retrieved a total of 2660 telecon links from 22 see
[2005-APR-09 16:03:40.000 KOJAK] Extending group UID-Group-8866 with 24 seeds...
[2005-APR-09 16:03:40.000 KOJAK]   Retrieved a total of 304 phoneCall links from 24 se
[2005-APR-09 16:03:42.000 KOJAK]   Retrieved a total of 2165 telecon links from 24 see
[2005-APR-09 16:03:43.000 KOJAK] Extending group UID-Group-6704 with 20 seeds...
[2005-APR-09 16:03:44.000 KOJAK]   Retrieved a total of 297 phoneCall links from 20 se
[2005-APR-09 16:03:45.000 KOJAK]   Retrieved a total of 2059 telecon links from 20 see
```

Next we merge expanded unknown group hypotheses with each other or known groups based on similarity. If we extend seeds for unknown groups we might wind up with a group that is already known or multiple unknown groups might really be the same group. Merging takes care of this duplicate elimination. In this example we do not have any unknown groups which makes this a no-op.

```
|= (kojak-log :low "Merging unknown groups...")
[2005-APR-09 16:03:50.000 KOJAK] Merging unknown groups...
```

If we merge groups thresholding them first is useful to get more accurate similarity measures. Thresholding is still a somewhat ad-hoc process, since there are no general indicators where the best place is for cutting the list. Specialized thresholding functions can be written that support thresholding for a particular domain (done for the IET datasets). There are also a few configuration parameters that allow some basic thresholding control (see the manual).

```
|= (threshold-groups (get-all-groups))

|= (merge-groups :similarity-epsilon 0.03
                 :merge-by :weight
                 :remove-merge-origins? TRUE)
[2005-APR-09 16:03:57.000 KOJAK]   0 unknown groups were merged with other groups.
()
```

Finally, we can output the resulting groups to the database and/or a file in multiple formats.

```
|= (kojak-log :low "Running KOJAK phase 3: storing and reporting results...")
[2005-APR-09 16:03:59.000 KOJAK] Running KOJAK phase 3: storing and reporting results.
```

As before, saving to the KOJAK DB is optional and could be commented out:

```
|= (db-save-groups (get-all-groups)
                   :dataset (get-parameter "DatasetName")
                   :source "AV"
                   ;; for final:
                   :category "F")
[2005-APR-09 16:04:02.000 SDBC] Renewing ODBC connection: DSN=KOJAK;DB=kojak_test;UID=
[2005-APR-09 16:04:02.000 SDBC] Connect via ODBC: DSN=KOJAK;DB=kojak_test;UID=root;
[2005-APR-09 16:04:02.000 SDBC] Disconnecting ODBC connection: DSN=KOJAK;DB=kojak_test
```

`report-groups` will do thresholding again, however, since we already did that above and no groups were merged, thresholding will be a non-op here. The output format can be specified via `ReportFormat` in the configuration, otherwise it will be inferred from the file extension. If `ReportMemberWeights` in the configuration is set to `true` (or specified here via the `:report-member-weights?` option) members will be output with member weight annotations. For this example, group reports will be written to '`example1-report.csv`' in the KOJAK directory.

```
|= (report-groups (get-all-groups))
[2005-APR-09 16:04:07.000 KOJAK] Group UID-Group-34988 has 100 potential members; repo
[2005-APR-09 16:04:07.000 KOJAK] Group UID-Group-8866 has 100 potential members; repor
[2005-APR-09 16:04:07.000 KOJAK] Group UID-Group-6704 has 100 potential members; repor
```

6.2 Other Example Runs

There are two more preconfigured examples that are variations of Example 1. The only difference between Example 1 and Example 2 is that Example 2 uses some slightly different ontology for groups and members (defined in '`kbs/example2-ontology.plm`') which is mapped onto the Group Finder's internal ontology via the synonym specifications provided in '`kbs/example2-seed-constraints.plm`'. This provides a simple example how a different data ontology can be mapped so that it can be properly understood by the KOJAK Group Finder. Example 2 can be run like this:

```
% ./run-kojak -c config/example2.dat
```

Results will be written to '`example2-report.csv`' in the KOJAK directory.

The difference between Example 1 and Example 3 is that here we load all link data into PowerLoom instead of querying it from the database. This is only recommended for small datasets. Example 3 can be run like this:

```
% ./run-kojak -c config/example3.dat
```

Results will be written to '`example3-report.csv`' in the KOJAK directory.

7 Advanced Configuration

Data describes or represents some aspects of the world. For each such aspect there are infinitely many ways in which it could be represented or conceptualized as well as how such a representation might be physically realized on a storage medium such as a computer file or a database. This openness presents a major challenge for applying and deploying a generic link discovery system such as the KOJAK Group Finder.

To do its work KOJAK conceptualizes the world as a graph where nodes represent entities such as groups (e.g., human organizations) and their members (e.g., people), and where links represent different kinds of relationships between them, e.g., that an entity is member of a certain group or that entity A paid money to entity B, etc. When KOJAK is applied to a new type of dataset, the data needs to be appropriately mapped onto this internal conceptualization to generate meaningful results.

KOJAK uses the classes (or concepts) and relations defined in its generic groups ontology (see file '`kbs/generic-groups-ontology.plm`' and also Chapter 9 [Group Finder Ontology], page 76) as its internal standard vocabulary to represent different kinds of groups, group members, membership relations, events, etc. This ontology also defines a set of abstract interface relations such as, for example, `GROUPS/linkCount` that allow the mutual information component to access link statistics for different types of links. Another example is the `GROUPS/nameString` relation which is used by the report generator to substitute name strings for entity IDs if such names are available.

In the simplest KOJAK configuration where data comes from CSV files seed groups and members are specified using this standard vocabulary, and links such as `phoneCall` or `telecon` use a standard representation format and are defined and mapped automatically by the KOJAK CSV file importer so that link statistics can be accessed via KOJAK's `GROUPS/linkCount` interface relation. In this case no special mapping is required, since the mapping is done in effect by the person who maps their data onto KOJAK's CSV input syntax.

There are basically three cases where this simple, automatic mapping is not sufficient:

1. If somebody wants to use a different vocabulary for things like groups and memberships. For example, if one wants to use the term `ThreatGroup` instead of the canonical `Group` or `KnownGroup`. This very simple case is illustrated by the Example 2 configuration and run script (see Section 6.2 [Other Example Runs], page 32).
2. If one wants to exploit additional domain knowledge or constraints to logically infer additional group members (besides those explicitly given). For example, one might want to add a rule that says that if two people participated in a certain type of link, say `robbedBankTogether`, then they must be in the same group. Therefore, if group membership of one person is known, group membership of the other person could be inferred from it. Rules of this type are used in the mapping for the IET synthetic datasets, for example, see the file '`kbs/iet-y3-seed-constraints.plm`' and the rules inferring threat groups and membership by participation in certain types of events.
3. If data comes from some existing relational database (with some arbitrary schema) it could potentially be very large and it might be cost prohibitive to export and translate all relevant data into KOJAK's CSV file format first. Instead, data should be accessed and aggregated on the database directly and the results mapped onto KOJAK relations

to achieve maximal scalability. This has also the advantage of always running against the most current version of the database instead of a potentially outdated translation.

In practice, any combination of the above cases might arise. In the following we describe a set of mechanisms that can be used to configure KOJAK for such situations and to appropriately map data onto the internal representation that KOJAK needs. This will be done more or less in a tutorial style that uses the mapping for RDEC's Ali Baba dataset as an example.

7.1 PowerLoom

The core mechanisms used to define mappings such as the one for the Ali Baba dataset are based on the PowerLoom knowledge representation and reasoning system. KOJAK is built on top of PowerLoom, so all of PowerLoom's functionality is available at any time. We will try to keep the description below self-contained, but a basic knowledge of PowerLoom and its capabilities will greatly facilitate understanding (see http://www.isi.edu/isd/LOOM/PowerLoom/ for more information about PowerLoom). For quick reference, an important subset of PowerLoom commands is documented in Section 8.1 [Important PowerLoom Commands], page 68.

PowerLoom is a logic-based knowledge representation and reasoning (KR&R) system that provides a language to define classes (called concepts), relations, instances, logic rules and mappings between PowerLoom relations and external database tables. PowerLoom also has a logical inference engine and query language that allows a user (or KOJAK) to query relationships that are not explicitly represented but logically follow based on some of the known rules.

PowerLoom uses the Knowledge Interchange Format (or KIF) as its basic representation language. KIF is a version of predicate logic that uses a Lisp-based uniform expression syntax. For example, the rule that all men are mortal could be represented in KIF like this:

```
(forall (?x) (=> (man ?x) (mortal ?x)))
```

KIF variables are symbols that start with a question mark. Note the Lisp-style prefix syntax where each expression is enclosed in parentheses and starts with the operator followed by a list of arguments. Besides the parentheses, no punctuation is required. For example, compare the following two expressions:

```
man(?x) => mortal(?x)       ;; infix notation
(=> (man ?x) (mortal ?x))   ;; KIF
```

Similar to the representation language, all PowerLoom commands need to be written in a Lisp-ish input syntax as well. Commands take zero or more required arguments and sometimes a set of options specified as keyword/value pairs. For example, we could use PowerLoom's `assert` command to tell it about the rule shown above:

```
(assert (forall (?x) (=> (man ?x) (mortal ?x))))
```

If we then tell it that Socrates is a man via

```
(assert (man (socrates)))
```

we can then ask whether Socrates is mortal like this

```
(ask (mortal (socrates)))
```

to which PowerLoom would return `TRUE` as the result.

Most of KOJAK's configuration files are PowerLoom files which contain a set of PowerLoom definitions and related commands. Such files can be loaded and interpreted at runtime to dynamically change the internal knowledge base and KOJAK's behavior. If KOJAK is run in interactive mode, such commands can also be typed in interactively - more examples on that are shown in some of the configuration sections below.

7.2 File and Module Structure

KOJAK divides its advanced configuration information into a set of PowerLoom knowledge base (KB) files and load scripts. The reasons for this separation are (1) to separate functionally different aspects of the configuration into separate files, and (2) that this information is organized into separate PowerLoom modules and currently each KB file can only be loaded into one single module. Each configuration file (e.g., '`config/ali-baba.dat`') points at its own set of KB files which are loaded during startup and initialization of KOJAK. When they are loaded the necessary definitions and mappings are established which can be completely different for two different configurations. There are the following types of KB files:

Ontology files define the vocabulary of classes and relations used by the Group Finder. Ontology files are somewhat analogous to a database schema definition. For example, '`kbs/generic-groups-ontology.plm`' is the central ontology file used to define the groups vocabulary of KOJAK. This file always needs to be loaded for proper operation. The file '`kbs/ali-baba-ontology.plm`' defines a set of additional classes and relations specific to the Ali Baba dataset. For example, it defines the relation `participatedInTerrorism` to represent and reason with a particular kind of information available in this dataset.

Seed constraint files define mappings between a dataset ontology and the generic groups ontology as well as rules that allow KOJAK to infer additional threat groups or seed members. For example, '`kbs/ali-baba-seed-constraints.plm`' defines the mappings between the Ali Baba ontology and the generic groups ontology as well as a rule that infers additional seed members based on joint participation in a terrorism event.

EDB schema files are a special kind of ontology file specifically dealing with the mapping of a relational evidence database (EDB) to the PowerLoom relations used by KOJAK. For example, '`kbs/kojak-edb-schema.plm`' is the file defining the mapping for KOJAK's own internal evidence database (the one into which data from CSV files gets imported) and '`kbs/ali-baba-edb-schema.plm`' defines the mapping for the Ali Baba EDB.

Data files define or load actual data instances such as certain groups, members, etc. We also put the actual mapping rules between KOJAK's abstract interface relations such as `GROUPS/linkCount` and particular EDB relations such as, for example, `EDB-link_count`, since they are in effect data import rules (conceivably, these could also go into the associated EDB schema file). For example, '`scripts/load-data.plm`' is the generic data file to load data from the internal KOJAK EDB, '`scripts/ali-baba-load-data.plm`' is the data file for the Ali Baba EDB.

Load files provide an extra level of indirection, e.g., to load multiple ontology files when the Group Finder ontology is loaded. For example, '`scripts/ali-baba-load-ontology.plm`' loads three different files. A load file could also contain actual definitions instead of just loading some other file as is the case for '`scripts/ali-baba-load-data.plm`'. The load file mechanism is not absolutely necessary and might be replaced in the future by simply specifying multiple EDB or ontology files

instead of specifying single load scripts as currently done via configuration parameters such as `LoadOntologyScript`, etc. Load files can use physical pathnames to load other files, but all files shipping with KOJAK use logical pathname syntax to load other files to make them platform and programming language independent. For more information on logical pathname syntax see Section 5.1 [Configuration File], page 14.

These various files need to be loaded in the correct order. For example, a relation such as `participatedInTerrorism` needs to be defined before it can be used in an assertion, rule or query. For this reason KOJAK first loads ontology files via the value of `LoadOntologyScript`, then EDB schema files via the value of `LoadEDBSchemaScript` and, finally, data via the value of `LoadDataScript`.

7.2.1 Module Structure

As mentioned above, PowerLoom KB files are loaded into modules. Modules are separate name and assertion spaces that can inherit from each other. Each PowerLoom KB file starts with an `IN-MODULE` declaration which specifies into which module the information should be loaded. The PowerLoom module system is very flexible and powerful but can also be confusing. **When generating a configuration for a new dataset it is therefore best to use an existing configuration and simply mirror its `IN-MODULE` declarations.**

Using appropriate module structure allows us to avoid name clashes (e.g., between user terms and the built-in groups ontology) or to separate non-volatile information such as schema definitions from more volatile information such as data assertions. For example, we could rerun certain commands on a different data set by clearing out the data module but leaving all ontology modules intact. The main modules used by KOJAK are described below:

KOJAK: This is a namespace that contains all KOJAK commands such as, for example, `extend-groups`. Run scripts such as '`scripts/ali-baba-run-kojak.plm`' use this as their module so commands can be written without the `KOJAK/` module prefix. Also, if KOJAK is started up in interactive mode it will initially be in the `KOJAK` module.

LD-ONTOLOGY: This is the top-level ontology module of KOJAK. All ontology files should be loaded into this module. It inherits the PowerLoom kernel module to make all built-in PowerLoom definitions and commands available as well as the `KOJAK` module which contains all KOJAK commands.

LD: This is the module used to contain mappings and seed constraints. It inherits `LD-ONTOLOGY`. Seed constraint files such as '`kbs/ali-baba-seed-constraints.plm`' should use this module.

GROUPS: This is a name space for the objects defined in KOJAK's generic groups ontology. The objects themselves are defined in `LD-ONTOLOGY` but their names come from the `GROUPS` name space to prevent clashes with user-defined concepts and relations. This is the reason why these terms always need to be prefixed as with, for example, `GROUPS/linkCount`.

EDB: This module is used to define database objects and schema mappings. It inherits `LD-ONTOLOGY` as well as commands from the PowerLoom RDBMS module. EDB schema files such as '`kbs/ali-baba-edb-schema.plm`' should use this module.

DATASET: This is the lowest level module intended to host data objects and assertions. It inherits all of the above. Data files such as '`scripts/ali-baba-load-data.plm`' should go into this module.

7.3 The Ali Baba Configuration

In the following we describe each and every aspect of the configuration for the Ali Baba dataset developed by SAIC for RDEC. The Ali Baba configuration ships with this KOJAK release and consists of the following files:

```
config/ali-baba.dat                      ...top-level config file
   kbs/ali-baba-ontology.plm             ...ontology file
   kbs/ali-baba-seed-constraints.plm     ...seed constraint file
   kbs/ali-baba-edb-schema.plm           ...EDB schema file
scripts/ali-baba-load-ontology.plm       ...ontology load script
scripts/ali-baba-load-edb-schema.plm     ...EDB schema load script
scripts/ali-baba-load-data.plm           ...data file
scripts/ali-baba-run-kojak.plm           ...run script
```

To create a new configuration it is best to first create appropriately named copies for each of these eight files (e.g., use a different prefix such as `my-edb-` instead of '`ali-baba-`') and then modify each file to fit the actual database and dataset that you are trying to map to. By starting with copies of an existing configuration each KB file will already have an appropriate `IN-MODULE` declaration.

Below we describe each and every file in this configuration in detail. It will be helpful to have printouts or electronic copies of these files handy when working through this section.

7.3.1 'ali-baba.dat'

This is the top-level configuration file to run KOJAK on the Ali Baba dataset. This file is what you will pass to KOJAK via the `-c` command line argument to configure it for this dataset, for example:

```
run-kojak -c config/ali-baba.dat
```

The set of available KOJAK configuration parameters is described in Chapter 5 [Configuration], page 14. The set of parameters that most definitely will need to be adapted for your dataset are the following:

```
Data                +=jdbc:mysql://blackcat:3306/ali_baba_v41?user=me
DatasetName          =AliBaba
LoadOntologyScript   =KJ:scripts;ali-baba-load-ontology.plm
LoadEDBSchemaScript =KJ:scripts;ali-baba-load-edb-schema.plm
LoadDataScript       =KJ:scripts;ali-baba-load-data.plm
RunKojakScript       =KJ:scripts;ali-baba-run-kojak.plm
ExpansionLinkTypes  +=phoneCall
ExpansionLinkTypes  +=affiliatedWith
ExpansionLinkTypes  +=communication
ReportFile           =./ali-baba-report.csv
```

You might want to add an additional `Data` entry to load seed information for your dataset, for example, from a CSV file. In the Ali Baba configuration seed information is loaded directly from the database in the '`scripts/ali-baba-load-data.plm`' script.

The various script variables need to point to the load and run scripts described below. `ExpansionLinkTypes` need to correspond to the link types defined and mapped in the ontology and EDB schema files. Alternatively, you can list those directly as an argument to `expand-groups` in the KOJAK run script.

You might also want to adapt various analysis parameters for your dataset, e.g., to control boosting and/or thresholding. Usually that doesn't need to be done right away but will be done over the course of multiple analysis runs.

7.3.2 'ali-baba-ontology.plm'

This file defines the Ali Baba ontology. An ontology defines a set of classes or types (in PowerLoom called "concepts") and a set of functions and relations. Under this view an ontology is roughly analogous to a database schema (however, ontologies usually also define hierarchical relationships, logic rules and constraints that further define or constrain the meaning of the defined terms).

The ontology file should define all terminology needed in addition to what is already available in the generic groups ontology. Additionally, it might define name variants or synonyms to some of the generic terms (which is what is done here).

The first part of the file defines a small person and group hierarchy that mostly mirrors the structure of 'kbs/generic-groups-ontology.plm' but uses the terms ThreatGroup and memberAgents, since those are the names used in the Ali Baba EDB. Using these definitions is not strictly necessary, since we could assert imported data directly in terms of the generic groups ontology. Classes of entities such as ThreatGroup can be defined with the PowerLoom DEFCONCEPT command (note that since the LD-ONTOLOGY module is case sensitive, PowerLoom commands have to be written in all upper-case):

```
(DEFCONCEPT Person)
(DEFCONCEPT Group)
(DEFCONCEPT ThreatGroup (Group))
(DEFCONCEPT NonThreatGroup (Group))
(DEFCONCEPT Event)

(DEFRELATION memberAgents (?group (?agent Person))
  :documentation "Asserts that ?agent is a 'Person' and a member of
?group.  Note that the argument order of 'memberAgents' links in the
Ali Baba EDB is actually reversed from the one used in the EAGLE Y2
and Y3 EDB schema!")
```

The classes ThreatGroup and NonThreatGroup are defined as subclasses of Group which is supplied as the superconcept in the DEFCONCEPT command.

memberAgents is a binary relation between a group and a Person which mirrors KOJAK's GROUPS/groupMember relation. The second argument of the DEFRELATION command is a list of variables defining the arguments and arity of the relation. If a variable is grouped with a concept name as done for ?agent the concept is interpreted to be the argument type for that position (somewhat similar to a column type in a database table).

Once these classes (or concepts) and relations have been defined, they can be used to introduce instances and assert relationships between them. Let us play with these definitions a little bit to get a better idea how they are used by KOJAK. To do so we'll start KOJAK in interactive mode via the -i command-line argument which instead of running an analysis end-to-end will bring up a command loop where we can execute KOJAK and PowerLoom commands:

```
% run-kojak -i -c config/ali-baba.dat
```

```
Running C++ version of KOJAK Group Finder...
Initializing STELLA...
Initializing PowerLoom...
Initializing KOJAK...
[2005-OCT-06 17:53:19.000 PL] Processing check-types agenda...
+------------------------------------------------------------+
|          Welcome to the KOJAK Group Finder v2.2.0          |
| Copyright (C) USC Information Sciences Institute, 1996-2005 |
| This software comes with ABSOLUTELY NO WARRANTY and is     |
| licensed for NON-COMMERCIAL EVALUATION PURPOSES ONLY!      |
|                  All rights reserved.                      |
+------------------------------------------------------------+

|=
```

Once we see the |= command prompt we can type commands and see their results. Let us start by examining and changing the current module with PowerLoom's `cc` command. Without an argument, the command will simply return the current module we are in which initially is the `KOJAK` module. Given a module name as an argument, it will switch the current module to that module:

```
|= (cc)

|MDL|/KOJAK

|= (cc LD-ONTOLOGY)

|MDL|/PL-KERNEL-KB/PL-USER/LD-ONTOLOGY
```

Now we are in the `LD-ONTOLOGY` module and ready to experiment with some ontology definitions. Note that this module is case sensitive (as opposed to the case insensitive `KOJAK` module) which means all KOJAK and PowerLoom commands have to be spelled in all upper-case. Let us start by running a PowerLoom `retrieve` query to see whether KOJAK knows about any `ThreatGroup`'s at this point. `retrieve` takes a logic sentence as an argument where at least one argument is a variable (a question mark symbol) and tries to find bindings for the variable(s) based on what is currently asserted in the knowledge base. The optional second argument specifies how many answers we are looking for. By default only one answer is retrieved, the `ALL` argument indicates that we are looking for all answers:

```
|= (RETRIEVE ALL (ThreatGroup ?g))

ERROR: Undeclared predicate or function reference: 'ThreatGroup'.
   Error occurred while parsing the proposition:
   (KAPPA (?g) (ThreatGroup ?g))
```

We get an error message, since at this point the `ThreatGroup` concept isn't yet defined. To define it we manually load the ontology file using the `load` command (in normal operation that's done automatically when `initialize-kojak` is called) and then run the query again.

```
|= (LOAD "KJ:kbs;ali-baba-ontology.plm")
```

```
|= (RETRIEVE ALL (ThreatGroup ?g))

No solutions.
```

This time we didn't get an error but also not any solutions, since the KB is still empty. Let's add some content with PowerLoom's `assert` command. We first switch to the `DATASET` module, since that is where we usually want to store any data:

```
|= (CC DATASET)

|MDL|/PL-KERNEL-KB/PL-USER/LD-ONTOLOGY/EVIDENCE-DATABASE-SCHEMA/EDB-
PRIMARY-DATA-PL/EDB-PRIMARY-DATA/DATASET

|= (ASSERT (ThreatGroup RAF))

|P|(ThreatGroup RAF)

|= (ASSERT (memberAgents RAF Ulrike_Meinhof))

|P|(memberAgents RAF Ulrike_Meinhof)

|= (ASSERT (memberAgents RAF Andreas_Baader))

|P|(memberAgents RAF Andreas_Baader)
```

Now we can ask some questions again:

```
|= (RETRIEVE ALL (ThreatGroup ?g))

[2005-OCT-06 18:00:34.000 PL] Processing check-types agenda...
There is 1 solution:
  #1: ?g=RAF

|= (RETRIEVE ALL (memberAgents RAF ?m))

There are 2 solutions:
  #1: ?m=Ulrike_Meinhof
  #2: ?m=Andreas_Baader

|= (RETRIEVE ALL (memberAgents ?g ?m))

There are 2 solutions:
  #1: ?g=RAF, ?m=Andreas_Baader
  #2: ?g=RAF, ?m=Ulrike_Meinhof
```

We can also illustrate some very simple logical inference based on these definitions. For example, we can retrieve all `Group`'s, since `ThreatGroup` is a subconcept of `Group`:

```
|= (RETRIEVE ALL (Group ?g))
```

```
There is 1 solution:
  #1: ?g=RAF
```

The following query retrieves all people. Note that we never explicitly asserted anybody to be a `Person`, but the system inferred `Person`-hood for the second argument of `memberAgents` assertions due to the type constraint given in the definition of the relation. This is different from the usual programming language semantics where a type constraint has to be satisfied before - say a function - can be applied to a particular argument. In PowerLoom we infer that the argument must be of that type if we are given the assertion of the relation. This is similar to, say, somebody tells you that X is Y's brother and you infer (given X is a person) that Y must be a person even if nobody has ever told you that explicitly.

```
|= (RETRIEVE ALL (Person ?x))

There are 2 solutions:
  #1: ?x=Andreas_Baader
  #2: ?x=Ulrike_Meinhof
```

This assertion of groups and their members is the basic mechanism via which KOJAK represents seed information. We haven't established the linkage yet between the terminology we used here and the generic groups ontology used by KOJAK, but once that is done (see next section) it can use queries just as the ones we used above to access this information.

Following these concept definitions is the definition of a `nameString` relation which can be used to assert a name with a particular entity. We do use this relation in '`scripts/ali-baba-load-data.plm`' to assert names for threat group entities, however, these name string are not yet used for anything. Conceivably, we could change the definition of the `GROUPS/nameString` rule at the end of the file to use those name strings. Just for illustration, here is how we can assert and retrieve such name strings:

```
|= (ASSERT (nameString RAF "Red Army Fraction"))

|P|(nameString RAF "Red Army Fraction")

|= (ASSERT (nameString RAF "Rote Armee Fraktion"))

|P|(nameString RAF "Rote Armee Fraktion")

|= (RETRIEVE ALL (nameString RAF ?name))

There are 2 solutions:
  #1: ?name="Red Army Fraction"
  #2: ?name="Rote Armee Fraktion"

|= QUIT

Really exit? (yes or no) yes
```

Next, the file defines `participatedInTerrorism` which we will use to materialize `terrorism` links from the Ali Baba EDB. Our interpretation of such links is that they

encode that two people committed an act of terrorism together (this interpretation might be wrong - unfortunately, the documentation of the Ali Baba dataset is somewhat lacking so we don't know for sure). Assuming our interpretation is correct, we will use such links to infer additional seed members for groups which will be illustrated in the next section.

The next three relations `affiliatedWith`, `communication` and `phoneCall` correspond to the three expansion link types that will be used by the mutual information component to extend groups. These relations will actually never be materialized but simply serve as names to link up the mutual information component's `GROUPS/linkCount` function to three different EDB relations. We could have done without actually defining these relations, but they allow us to document here what types of links we are using for group extension. The use and semantics of these relations will be described in more detail later.

Finally, what kind of information is exploitable to infer additional seed members or what link types are usable for group expansion via the mutual information component will of course be different for every dataset. What we used here only makes sense for the Ali Baba data and can only serve as an illustration. For each new dataset some manual analysis has to be done to determine what information is available and what portion of it might be usable by the KOJAK Group Finder (also see Section 4.1.2 [Link Data], page 10 for some more information on what types of data the Group Finder can exploit).

7.3.3 'ali-baba-seed-constraints.plm'

The seed constraints file defines two things: (1) the mapping between terms in the Ali Baba group ontology and the generic group ontology used by KOJAK, and (2) constraint rules (really just one in this case) for inferring additional seed members. There is no specific requirement that the term mapping be defined in this file - we could have just as well added it to the ontology file. However, the mapping might be more complex and involve rules which would interact with the seed constraint rules. For this reason, we usually put these two pieces together in this file so it is easier to see their connection and interaction. For example, look at '`kbs/iet-y3-seed-constraints.plm`' to a see such a more complex mapping.

Let us experiment again with KOJAK in interactive mode to see how the seed constraints file is used. We start by loading the two ontology files but not yet the constraints file:

```
% run-kojak -i -c config/ali-baba.dat
Running C++ version of KOJAK Group Finder...
Initializing STELLA...
Initializing PowerLoom...
Initializing KOJAK...
[2005-OCT-07 13:58:47.000 PL] Processing check-types agenda...
+------------------------------------------------------------+
|          Welcome to the KOJAK Group Finder v2.2.0          |
| Copyright (C) USC Information Sciences Institute, 1996-2005 |
| This software comes with ABSOLUTELY NO WARRANTY and is     |
| licensed for NON-COMMERCIAL EVALUATION PURPOSES ONLY!      |
|                  All rights reserved.                      |
+------------------------------------------------------------+

|= (LOAD "KJ:kbs;generic-groups-ontology.plm")
```

```
|= (LOAD "KJ:kbs;ali-baba-ontology.plm")
```

Next, we assert a ThreatGroup instance again and see whether we can retrieve the instance using KOJAK's generic group syntax:

```
|= (CC DATASET)

|MDL|/PL-KERNEL-KB/PL-USER/LD-ONTOLOGY/EVIDENCE-DATABASE-SCHEMA/EDB-
PRIMARY-DATA-PL/EDB-PRIMARY-DATA/DATASET

|= (ASSERT (ThreatGroup RAF))

|P|(ThreatGroup RAF)

|= (RETRIEVE ALL (GROUPS/KnownGroup ?g))

No solutions.
```

There were no solutions. The reason is that the linkage between the Ali Baba term ThreatGroup and KOJAK's GROUPS/KnownGroup hasn't been established yet. The first section of 'kbs/ali-baba-seed-constraints.plm' establishes this connection by declaring synonym relationships between KOJAK terms and Ali Baba terms. This is the simplest form of mapping where we simply declare equivalence between terms. More complicated mappings can be established by using rules. For example, suppose we have a memberAgents relation that uses arguments in the reverse order of GROUPS/groupMember. In that case we could use the following rule to map between them:

```
(ASSERT (<=> (memberAgents ?a ?g)
             (GROUPS/groupMember ?g ?a)))
```

We could have used rules like that instead of all the synonym mappings, however, the synonym mechanism is more efficient, since it doesn't require an extra inference step to carry out the mapping.

Now let us load the mapping and see how it changes things. We have to clear the data module first and reassert the information, since the synonym relation will not affect assertions that have already been made. After that the query that failed above does return the result we want:

```
|= (LOAD "KJ:kbs;ali-baba-seed-constraints.plm")

|= (CLEAR-MODULE DATASET)

|= (ASSERT (ThreatGroup RAF))

|P|(ThreatGroup RAF)

|= (RETRIEVE ALL (GROUPS/KnownGroup ?g))

There is 1 solution:
  #1: ?g=RAF
```

The second part of the constraints file contains the following rule:

```
(ASSERT
  (=> (AND (ThreatGroup ?g)
           (memberAgents ?g ?p1)
           (EXISTS (?e)
             (participatedInTerrorism ?e ?p1 ?p2)))
      (memberAgents ?g ?p2)))
```

What it models is the following: If there is a threat group *?g* with a member *?p1* and there is some terrorism event *?e* where some other person *?p2* participated with *?p1*, then that other person must also be a member of the threat group *?g*. Let us see this rule in action. We start by asserting some initial seed members again and see that we can only retrieve what we asserted:

```
|= (ASSERT (AND (memberAgents RAF Andreas_Baader)
                (memberAgents RAF Ulrike_Meinhof)))

(|P|(groupMember RAF Andreas_Baader) |P|(groupMember RAF Ulrike_Meinhof))

|= (RETRIEVE ALL (GROUPS/groupMember ?g ?m))

[2005-OCT-07 14:01:27.000 PL] Processing check-types agenda...
There are 2 solutions:
  #1: ?g=RAF, ?m=Ulrike_Meinhof
  #2: ?g=RAF, ?m=Andreas_Baader
```

Now we assert that some new person performed a terrorist act together with a known member of the RAF. When we ask for membership again KOJAK now finds the new member through logical inference, even though membership was not explicitly asserted:

```
|= (ASSERT (participatedInTerrorism US_Army_Corp_Bombing
                                    Andreas_Baader
                                    Gudrun_Ensslin))

|P|(participatedInTerrorism US_Army_Corp_Bombing
                             Andreas_Baader
                             Gudrun_Ensslin)

|= (RETRIEVE ALL (GROUPS/groupMember ?g ?m))

[2005-OCT-07 14:02:17.000 PL] Processing check-types agenda...
There are 3 solutions:
  #1: ?g=RAF, ?m=Ulrike_Meinhof
  #2: ?g=RAF, ?m=Andreas_Baader
  #3: ?g=RAF, ?m=Gudrun_Ensslin
```

We can use PowerLoom's explanation mechanism to illustrate how the rule was actually used here:

```
|= (SET-FEATURE JUSTIFICATIONS)
```

```
|1|(:JUSTIFICATIONS :EMIT-THINKING-DOTS :JUST-IN-TIME-INFERENCE)

|= (ASK (GROUPS/groupMember RAF Gudrun_Ensslin))

TRUE

|= (WHY)

1 (groupMember RAF Gudrun_Ensslin)
    follows by Modus Ponens
    with substitution {?g/RAF,
                       ?p2/Gudrun_Ensslin,
                       ?e/US_Army_Corp_Bombing,
                       ?p1/Andreas_Baader}
    since 1.1 ! (FORALL (?g ?p2)
                   (<= (groupMember ?g ?p2)
                       (EXISTS (?e ?p1)
                          (AND (ThreatGroup ?g)
                               (groupMember ?g ?p1)
                               (participatedInTerrorism ?e ?p1 ?p2)))))
    and   1.2 ! (ThreatGroup RAF)
    and   1.3 ! (participatedInTerrorism US_Army_Corp_Bombing
                                         Andreas_Baader
                                         Gudrun_Ensslin)
    and   1.4 ! (groupMember RAF Andreas_Baader)
```

Here we only have one constraint rule but there is no limit to the complexity of such a model. Any of the generic group terms can have rules associated with it to allow them to be inferred from other available information. For example, we might have a rule that states that an organization that received financial support from some known threat group is itself a threat group. How complex this needs to be will again strongly depend on the particular dataset and the available information. One important caveat is that such rules should be fairly reliable (but not necessarily 100%), since KOJAK currently treats such logically inferred information just as if it had been asserted explicitly (except that inferred seed members are annotated as such in the hypothesis tables of the KOJAK database).

KOJAK uses queries like the ones shown above to retrieve seed groups and their members. In fact, that's exactly what KOJAK's `retrieve-groups` command does, for example:

```
|= (RETRIEVE-GROUPS)

[2005-OCT-07 14:56:46.000 KOJAK] Found 1 known groups...
[2005-OCT-07 14:56:46.000 KOJAK]   Looking for group memberships...
[2005-OCT-07 14:56:47.000 KOJAK]   Done retrieving groups and members.
(|GROUP|RAF)

|= (KOJAK-LOG-OBJECTS :LOW "Found these groups:" (GET-ALL-GROUPS))

[2005-OCT-07 14:59:11.000 KOJAK] Found these groups:
```

```
[GROUP RAF
  known members: (|AGENT|Ulrike_Meinhof |AGENT|Andreas_Baader)
  inferred members: (|AGENT|Gudrun_Ensslin)]
```

Given appropriate mappings, it can retrieve this seed information even if it was asserted using a different vocabulary or classes and relations. In general, it is a good idea to use the built-in generic vocabulary wherever possible to simplify things and avoid errors, however, as the above shows it is possible to use custom terms if so desired.

7.3.4 'ali-baba-edb-schema.plm'

The EDB schema file defines the interface between KOJAK's internal logic-based representation (based on PowerLoom) and an external relational evidence database. Once the EDB schema file is in place the following operations are possible:

- Query external database tables fully transparently via PowerLoom's `retrieve` command just like any other PowerLoom relation
- Selectively import and assert information from the external database in KOJAK's internal KB
- Compute link statistics directly on the database and use them to perform group extension via KOJAK's mutual information component

The EDB schema file is usually the most complex part of every configuration. Creating it will require intimate knowledge of the content as well as the schema of the external EDB. Fortunately, the schema mapping only needs to be done for relevant tables of the database but not necessarily all of them.

PowerLoom commands to define database objects as well as the necessary supporting concepts and relations are all defined in the `RDBMS` namespace. This namespace is inherited in the `EDB` module and therefore these commands usually don't have to be explicitly qualified. Below, however, we do always describe them with their namespace prefix to be clear where they are coming from.

7.3.4.1 Database Instances

In KOJAK an external database is represented via a database instance which is a logic instance of type `RDBMS/SQL-Database`. All physical connection information is associated with this database instance and can be changed to link the instance to a different physical database. In this sense a database instance is very similar to an ODBC or JDBC data source. Mapping definitions such as table projections or queries are always associated with the logical database instance only to hide the physical connection details. These details would be different, for example, between the C++ and Java versions of KOJAK even if the same external database is used.

Database instances can be defined via the `RDBMS/defdb` command. The first command in '`ali-baba-edb-schema.plm`' defines the `edb` database instance which is the only one used (so far) by KOJAK. We don't assign any connection information, since that will be done programatically by KOJAK using the information from the configuration file. Otherwise, it could be explicitly asserted in a KOJAK run script via KOJAK's `db-assert-connection-info` command.

7.3.4.2 Table Mappings

The next section in the schema file defines a fairly extensive mapping between tables in the Ali Baba evidence database and corresponding PowerLoom relations. Mapping between a relational database table and a PowerLoom relation is fairly straightforward, since a database table is simply a set of tuples. Each such table could therefore be represented in PowerLoom with a corresponding relation that has as many arguments as the database table has columns. Usually, however, we will map a database table onto one or more PowerLoom relations that each correspond to a particular *projection* of the database table. The reasons for this are the following:

1. Only some columns of a database table might be needed in KOJAK; by restricting the mapping to the essential columns we avoid communication of unnecessary information which improves speed and scalability.
2. Relational database use a different modeling style where tables might often be very wide with many columns, while PowerLoom models are usually "skinny" where relations have rarely more than 3 or 4 arguments. Since columns are referred to explicitly in SQL queries, wide tables do not cause a problem there, however, using PowerLoom relations with many arguments is tedious, since all argument variables always need to be listed. Since a column in a database table often corresponds to a single "semantic aspect" of some object (usually represented by a key), the better mapping is to represent each such key/column combination as a separate PowerLoom relation.
3. PowerLoom uses predicate logic as the underlying representation mechanism which does not have a natural way of supporting NULL values. Rows containing a NULL value can currently not be returned by a PowerLoom query, since the NULL value is equivalent to an unbound variable (some of PowerLoom's partial inference mechanisms can be used to work around this problem, but those are beyond the scope of this document). Therefore, mapping a database table onto a PowerLoom relation where for some row one or more columns could be NULL would prevent one from being able to retrieve the non-NULL columns of such a row. By mapping the table onto multiple projections this problem can be avoided.

The main command to define the mapping between a database table and a PowerLoom relation is `RDBMS/deftable`. It is very similar to a standard `defrelation` command but takes two extra arguments: (1) a logical database instance on which this table resides (e.g., `edb`) and (2) the name of the database table to which this PowerLoom relation corresponds. The argument list then specifies which table columns map onto which relation arguments and what if any type coercions should be performed. For example, let us look at the following table definition from the Ali Baba schema file:

```
(DEFTABLE EDB-AliBabaName edb "AB_V41" (ID (NAME STRING)))
```

This defines a new binary PowerLoom relation `EDB-AliBabaName` (just as if we had used `defrelation`) and associates it with the table `"AB_V41"` on the database identified by `edb`. The table `"AB_V41"` is a wide table with 22 columns part of which are repeated below (the full table description is listed in the EDB schema file):

```
mysql> describe ab_v41;
+--------------+--------------+------+-----+---------+
| Field        | Type         | Null | Key | Default |
+--------------+--------------+------+-----+---------+
```

```
| ID           | int(11)      |      | PRI | 0       |
| ALIAS        | varchar(200) | YES  |     | NULL    |
| NAME         | varchar(200) | YES  |     | NULL    |
| POB          | varchar(200) | YES  |     | NULL    |
| DOB          | varchar(200) | YES  |     | NULL    |
  ..............................................
| ACTIVITY     | varchar(200) | YES  |     | NULL    |
+--------------+--------------+------+-----+---------+
22 rows in set (0.00 sec)
```

The purpose of `EDB-AliBabaName` is to map an entity ID onto its name string defined in this table. The specifics of this projection are defined by the argument list of `EDB-AliBabaName`. Each argument name has to correspond to the name of some table column (arguments can be specified in the standard question-mark syntax if needed, e.g., to reference them in axioms in the definition body). If the argument is specified without a type (as with the `ID` argument above), argument values brought in during a query will be coerced to standard logic objects. If the argument is followed by a type (as with the `NAME` argument above), column values will be coerced to this PowerLoom type. Logic objects have a lot of support machinery to handle name spaces, modules, indexing, equality, etc. which makes them somewhat more heavy weight. For cases where this machinery isn't required, it is better to coerce such columns into simpler types such as `STRING` or `INTEGER`, since they occupy significantly less storage.

Let us now show how all this works with an example. We start KOJAK in interactive mode again and define the database instance and table mapping described above. For simplicity, we add the connection information right into the database definition:

```
% run-kojak java -i -c config/ali-baba.dat
Running Java version of KOJAK Group Finder...
Initializing STELLA...
Initializing PowerLoom...
Initializing KOJAK...
[2005-OCT-10 15:31:59.000 PL] Processing check-types agenda...
+------------------------------------------------------------+
|          Welcome to the KOJAK Group Finder v2.2.0          |
| Copyright (C) USC Information Sciences Institute, 1996-2005 |
| This software comes with ABSOLUTELY NO WARRANTY and is     |
| licensed for NON-COMMERCIAL EVALUATION PURPOSES ONLY!      |
|                  All rights reserved.                      |
+------------------------------------------------------------+

|= (cc edb)

|MDL|/PL-KERNEL-KB/PL-USER/LD-ONTOLOGY/EVIDENCE-DATABASE-SCHEMA

|= (DEFDB edb
    :jdbc-connection-string
      "jdbc:mysql://blackcat.isi.edu:3306/ali_baba_v41?user=me&password=???"
    :SQL-database TRUE)
```

```
|i|edb

|= (DEFTABLE EDB-AliBabaName edb "AB_V41" (ID (NAME STRING)))

|r|EDB-AliBabaName
```

Now we are ready to ask some queries. Note that at this point we have nothing loaded into the KOJAK knowledge base, but these queries succeed, since they look up results directly in the external database. We start by asking for the name of the person with ID 404. Since in PowerLoom "404" would be interpreted as an integer, we enclose it in vertical bars to tell the system that this is a logic constant whose name is "404":

```
|= (RETRIEVE ALL (EDB-AliBabaName |404| ?name))

[2005-OCT-10 15:36:03.000 SDBC] Connect via JDBC:
  jdbc:mysql://blackcat.isi.edu:3306/ali_baba_v41?user=me&password=???
There is 1 solution:
  #1: ?name="Suraqah"
```

The log message shows that the connection to the database wasn't attempted until the first query was asked. The PowerLoom query above was translated into the following SQL query and its results where then fed back to the PowerLoom inference engine:

```
select NAME from AB_V41 where ID='404';
```

What this translation exactly looks like depends on the particular binding pattern. For example, the following asks the "reverse" query from the name to the corresponding ID. Since names are not unique, we get multiple solutions here. Also note that because the ID column gets coerced into PowerLoom logic objects, they are printed with vertical bars around them to distinguish them from regular integers:

```
|= (RETRIEVE ALL (EDB-AliBabaName ?id "Suraqah"))

There are 5 solutions:
  #1: ?id=|404|
  #2: ?id=|1616|
  #3: ?id=|1293|
  #4: ?id=|1865|
  #5: ?id=|4366|
```

Next, we leave both arguments unbound. We restrict the number of answers sought to 10, since otherwise this would bring in all 6000 or so rows of this table (in general, one should always be very careful when composing queries against a database, since result sets could be very large):

```
|= (RETRIEVE 10 (EDB-AliBabaName ?id ?name))

There are 10 solutions so far:
  #1: ?id=|402|, ?name="Abidin"
  #2: ?id=|403|, ?name="Sa'eed"
  #3: ?id=|404|, ?name="Suraqah"
  #4: ?id=|405|, ?name="Fikri"
```

```
#5: ?id=|406|, ?name="Mu'izz"
#6: ?id=|407|, ?name="Mulhim"
#7: ?id=|408|, ?name="Arfan"
#8: ?id=|409|, ?name="Usama"
#9: ?id=|410|, ?name="Ali"
#10: ?id=|411|, ?name="Mubarak"
```

We can also ask TRUE/FALSE questions:

```
|= (ASK (EDB-AliBabaName |402| "Abidin"))

TRUE
```

To illustrate the NULL-value problem outlined above, let us define a new table relation that maps an entity ID on its name and alias columns at the same time:

```
|= (DEFTABLE EDB-AliBabaNameAlias edb "AB_V41"
      (ID (NAME STRING) (ALIAS STRING)))

|r|EDB-AliBabaNameAlias
```

For entities that actually have both a name and an alias this does what is expected:

```
|= (RETRIEVE ALL (EDB-AliBabaNameAlias |2772| ?name ?alias))

There is 1 solution:
  #1: ?name="Gimmel Faruk", ?alias="Scar Face"
```

However, for entities that only have a name this now causes a problem, since the NULL value in the alias column prevents us from retrieving the row (`?alias` would remain unbound), so we never could retrieve the name of such an entity using `EDB-AliBabaNameAlias`:

```
|= (RETRIEVE (EDB-AliBabaNameAlias |404| ?name ?alias))

No solutions.
```

The next 16 `deftable` definitions in '`kbs/ali-baba-edb-schema.plm`' provide a fairly exhaustive mapping of the Ali Baba schema whose table structure follows the Year-2 EAGLE EDB schema. For the Ali Baba data only `EDB-LinkOfType`, `EDB-Organization` and `EDB-AliBabaName` are actually used to import data. Additionally, the various type ID tables are used to import type ID information. One important "wrinkle" of this mapping is that all type IDs (e.g., the first argument of `EDB-EntityType`) are explicitly coerced to integers to avoid conflicts with actual people entities whose ID might be the same.

Note that table relation such as `EDB-AliBabaName` can only be used to query a database. They can also be asserted via the PowerLoom `assert` command, however, such assertions do not update the database (future versions of PowerLoom might provide this feature). Also, database table queries do not materialize any assertions on the PowerLoom end (apart from caching). If that is desired PowerLoom's `assert-from-query` command can be used (more on that below).

7.3.4.3 Materializing Type IDs

What is described in this section is very specific to the "typeless" nature of the EAGLE EDB schema which is unlikely to occur often in real-life databases. The concepts and

mechanisms described here might still be useful, but if you are not dealing with such a type of schema you could safely skip this section.

The Year-2 EAGLE EDB schema is a very flexible and extensible schema where a single entity table holds entities of many different types, and, similarly, a single link table holds entity-entity links of many different types. This allows the introduction of new entity or link types without having to change the database schema. Instead of having separate tables for each entity and link type, each entry in the EDB `ENTITY` and `LINK` tables is associated with a type ID. Type IDs are mapped onto descriptive type names via tables such as `ENTITYTYPE` and `LINKTYPE`. Many of these descriptive names originate in an EDB ontology developed by Cycorp, but the Ali Baba database also uses many new and undocumented types.

Because of this "typeless" nature of the EAGLE EDB schema, each query will usually have to specify entity and link type IDs to restrict the types of entities and links that should be considered. This can convolute query expressions significantly, since it usually involves an extra constraint for the type ID plus an extra join if one doesn't want to hardcode integer type IDs but actually wants to key in on their descriptive name. For example, the following query would retrieve all organizations in the entity table:

```
select e.ENTITYID from ENTITY e, ENTITYTYPE et
       where e.ENTITYTYPEID=et.ENTITYTYPEID
       and et.CYCCOLLECTION='Organization';
```

Similarly, using the PowerLoom mapping we would have to phrase this query as follows:

```
(RETRIEVE ALL (AND (EDB-EntityType ?typeid Organization)
                   (EDB-EntityOfType ?id ?typeid)))
```

To avoid the extra join and database roundtrip as well as hard-coding integer type IDs in our queries, we define a set of PowerLoom functions that can map an entity type such as `Organization` onto its type ID by simply looking it up in a local table. This allows us to rephrase the query above like this:

```
(RETRIEVE ALL (EDB-EntityOfType ?id (EDB-EntityTypeID Organization)))
```

This looks up the type ID on the PowerLoom side and translates into the following much simpler SQL query:

```
select ENTITYID from ENTITY where ENTITYTYPEID=1071;
```

The type ID tables represented by the four PowerLoom functions `EDB-EntityTypeID`, `EDB-LinkTypeID`, `EDB-EntityAttributeTypeID` and `EDB-LinkAttributeTypeID` are populated using PowerLoom's `assert-from-query` command. For example:

```
(ASSERT-FROM-QUERY
 (RETRIEVE ALL (?pred ?id) (EDB-EntityType ?id ?pred))
 :relation EDB-EntityTypeID)
```

`assert-from-query` takes a retrieve command as its argument and then creates an assertion for each solution (or set of variable bindings) the query generates. There are different ways for creating these assertions (see the documentation of `assert-from-query` for more detail). One way used here is to provide a `:relation` argument which will assert the given relation for each set of bindings retrieved by the query. For example, one of the 122 assertions created by the above is the following:

```
(ASSERT (EDB-EntityTypeID Organization 1071))
```

Note that in the query above we supply the set of query variables (similar to output columns in a select statement) to get the appropriate argument order when bindings are supplied to `EDB-EntityTypeID`.

`assert-from-query` is very useful to selectively materialize portions of the external DB to avoid having to query for the same information over and over again. This mechanism will be used again when we discuss '`scripts/ali-baba-load-data.plm`'.

7.3.4.4 Defining Link Count Relations

Probably the most important but also challenging part of the Ali Baba EDB schema file is the section that defines the three link count relations `EDB-phoneCallCount`, `EDB-communicationCount` and `EDB-affiliatedWithCount`. These relations are used by KOJAK's mutual information (MI) component to perform group extension and compute connection strengths between individuals. The challenge here is not so much with the KOJAK or PowerLoom interface, but with the formulation of appropriate SQL queries which can become somewhat complex.

The MI component starts with a set of group seed entities and in its first step builds an extended graph around them. To do this it looks at each seed entity and finds all other entities connected to it by a link of one of the expansion link types listed in the configuration or directly supplied to the `expand-groups` command (which see). Additional to just looking for entities connected via such links, it also retrieves link counts, that is, how many links of a particular type lead to some new entity X. For example, it might retrieve that from seed entity 38 there are two `communication` links to entity 1155. These link counts are an important input to the MI computation.

The MI component uses `GROUPS/linkCount` as the central interface relation to access links and link counts from group seeds. This relation takes a set of seeds and an expansion link type as arguments and returns the retrieved entities and associated counts. For example, let us look at the following query:

```
|= (RETRIEVE ALL (GROUPS/linkCount communication (SETOF |342| |38|) ?p1 ?p2 ?c))
There are 9 solutions:
  #1: ?p1=|38|, ?p2=|2|, ?c=1
  #2: ?p1=|38|, ?p2=|22|, ?c=1
  #3: ?p1=|38|, ?p2=|264|, ?c=1
  #4: ?p1=|38|, ?p2=|639|, ?c=1
  #5: ?p1=|38|, ?p2=|1155|, ?c=2
  #6: ?p1=|342|, ?p2=|2|, ?c=1
  #7: ?p1=|342|, ?p2=|214|, ?c=1
  #8: ?p1=|342|, ?p2=|247|, ?c=1
  #9: ?p1=|342|, ?p2=|277|, ?c=1
```

In this query we supplied `communication` as the link type we are interested in and a set of two seed entities represented by the `setof` term. *?p1* will be bound to each seed member in the set and *?p2* to the various entities *?p1* is connected to via a `communication` link. The last argument reports how many such links are between the two entities.

For the Ali Baba dataset, the actual retrieval of these links and counts is done by the `EDB-communicationCount` relation which is defined in '`kbs/ali-baba-edb-schema.plm`'. For example, we could have used this relation directly to ask the query:

```
|= (RETRIEVE ALL (EDB-communicationCount (SETOF |342| |38|) ?p1 ?p2 ?c))

There are 9 solutions:
  #1: ?p1=|38|, ?p2=|2|, ?c=1
  #2: ?p1=|38|, ?p2=|22|, ?c=1
  #3: ?p1=|38|, ?p2=|264|, ?c=1
  #4: ?p1=|38|, ?p2=|639|, ?c=1
  #5: ?p1=|38|, ?p2=|1155|, ?c=2
  #6: ?p1=|342|, ?p2=|2|, ?c=1
  #7: ?p1=|342|, ?p2=|214|, ?c=1
  #8: ?p1=|342|, ?p2=|247|, ?c=1
  #9: ?p1=|342|, ?p2=|277|, ?c=1
```

GROUPS/linkCount and EDB-communicationCount are connected via the following rule which is defined in 'scripts/ali-baba-load-data.plm' (since it is a data import rule but it could have been defined in the EDB schema file as well):

```
(ASSERT
  (=> (AND (BOUND-VARIABLES ?seeds)
           (= ?ltype communication)
           (EDB-communicationCount ?seeds ?party1 ?party2 ?count))
      (GROUPS/linkCount ?ltype ?seeds ?party1 ?party2 ?count)))
```

So, whenever the MI components asks a GROUPS/linkCount query where *?ltype* is bound to communication, PowerLoom simply backchains into EDB-communicationCount which then does all the work. The bound-variables clause is there for safety and makes sure that the query is never asked with *?seeds* unbound (which could only happen if asked manually). The reason for this safeguard is that such a completely unconstrained query might bring in the whole database which could of course be very large and take forever.

The other two link types phoneCall and affiliatedWith are connected to corresponding EDB relations with similar rules. Now it should become clear why these relations defined in kbs/ali-baba-ontology.plm only serve as names but are never actually asserted or queried.

Having explained how GROUPS/linkCount is hooked up to EDB relations that can retrieve such links and associated counts, all that remains to be done is to show how an EDB relation such as EDB-communicationCount can be defined. Such link count relations are defined via parametric SQL queries specified via PowerLoom's RDBMS/defquery command (which see). For example, EDB-communicationCount is defined like this:

```
(DEFQUERY EDB-communicationCount (?seeds ?party1 ?party2 (?count INTEGER))
  :query-pattern
  (RDBMS/SQL-QUERY edb
    "SELECT ' ', P1, P2, COUNT(P2)
     FROM...<complicated SQL query>..."))
```

The RDBMS/defquery command defines a PowerLoom relation with the given name and arguments and links it to an SQL query whose arity (number of output columns) needs to match the arity of the relation. SQL output columns will be assigned by position to relation arguments, therefore, variable names such as *?seeds* do not have to match a column name

as with the RDBMS/deftable command. Similar to RDBMS/deftable, relation arguments can be associated with a type (e.g., as done for *?count*) to force type coercion.

The value of the :query-pattern argument has to be of the following form:

```
(RDBMS/SQL-QUERY <db-instance> "<sql query>")
```

The first element in this list is RDBMS/SQL-QUERY which is the name of a PowerLoom query specialist to handle arbitrary parametric SQL queries. RDBMS/defquery is very general and supports other queries such as joins or non-SQL queries as well, but for KOJAK SQL queries are all we need. <db-instance> needs to be a database instance which will usually be edb.

Writing Parametric SQL Queries

For the parametric SQL query there are two types of complexity that we need to handle: (1) how to pass in bound arguments that will result in appropriate constraints, and (2) writing an appropriate SQL query that can compute the desired link counts. Before we explain how count queries need to be constructed, let us explain the parametric query mechanism with a simple example. Below we define a relation myEntityOfType which has essentially the same functionality as EDB-EntityOfType which was defined via RDBMS/deftable earlier in the EDB schema file:

```
(DEFQUERY myEntityOfType (?id (?typeID INTEGER))
  :query-pattern
  (RDBMS/SQL-QUERY edb
    "SELECT ENTITYID, ENTITYTYPEID
     FROM ENTITY
     WHERE ENTITYID='?id'
     AND   ENTITYTYPEID IN '?typeID'"))
```

The SQL query has two output columns to match the relation variables *?id* and *?typeID*. The rest of the SQL query is fairly straightforward. What makes the query parametric is the mechanism by which relation variables are bound to column constraints. For example, the constraint

```
ENTITYID='?id'
```

would be translated into

```
ENTITYID='38'
```

if *?id* is bound to 38. The constraint

```
ENTITYTYPEID IN '?typeID'
```

would be translated into

```
ENTITYTYPEID IN ('1074')
```

if *?typeID* is bound to 1074. This allows us to answer a query such as this:

```
|= (ASK (myEntityOfType |38| 1074))

TRUE
```

If the relation variable in such a column constraint is unbound, the whole constraint is eliminated in the corresponding SQL translation (i.e., replaced by TRUE), which allows us to run queries like the following:

```
(RETRIEVE ALL (myEntityOfType |38| ?type))

There is 1 solution:
  #1: ?type=1074

|= (RETRIEVE 5 (myEntityOfType ?id 1071))

There are 5 solutions so far:
  #1: ?id=|1289|
  #2: ?id=|1290|
  #3: ?id=|1291|
  #4: ?id=|1292|
  #5: ?id=|1293|
```

Finally, binding a relation variable to a set of terms will translate into an appropriate `IN` constraint (even if the SQL query was specified with an = constraint for that column):

```
|= (RETRIEVE ALL (myEntityOfType (SETOF |38| |1296|) ?type))

  #1: ?type=1074
  #2: ?type=1071
```

This last PowerLoom query was translated into the following SQL query:

```
SELECT ENTITYID, ENTITYTYPEID
FROM ENTITY
WHERE ENTITYID IN ('38', '1296')
AND   TRUE
```

The parametric constraints shown above use a simplified, restricted syntax that has to follow one of the following patterns (`<ws>` means one or more whitespace characters):

```
<ws><column>=<?var><ws>
<ws><column>='<?var>'
<ws><column>="<?var>"
<ws><column>=‘<?var>‘
<ws><column> {in|IN} <?var><ws>
<ws><column> {in|IN} '<?var>'
<ws><column> {in|IN} "<?var>"
<ws><column> {in|IN} ‘<?var>‘
```

Note that the amount (or lack) of whitespace in each pattern is very specific and has to be strictly followed for the constraint to work. If quotes are used, the substituted value will be appropriately escaped for the chosen quote character. A more general parameter substitution syntax is also available to support the construction of more complex queries, but it is beyond the scope of this document.

Writing Link Count Queries

With this machinery in hand we can now go on to write link count queries for relations such as `EDB-communicationCount`. Let us again look at the overall structure of this relation:

```
(DEFQUERY EDB-communicationCount (?seeds ?party1 ?party2 (?count INTEGER))
  :query-pattern
  (RDBMS/SQL-QUERY edb
```

```
"SELECT ' ', P1, P2, COUNT(P2)
 FROM...<complicated SQL query>..."))
```

It needs to have four arguments as shown above (which could be given different names). The first is an "input" argument which will be bound to a set of seeds when the MI components asks the query, the remaining arguments are "output" arguments which will be generated by the embedded parametric SQL query.

The output columns of the SQL query should look as follows (again names could be chosen differently):

```
SELECT ' ', P1, P2, COUNT(P2) ...
```

The first value ' ' is only needed to match up with the *?seeds* variable position. Since that variable will be bound at query time we don't have to generate any actual output here. Make sure you use ' ' and not '', since the latter is interpreted as a NULL value on some database systems (e.g., Oracle) which would cause the query to fail in PowerLoom.

The next output column called `P1` above will be bound to each of the seed members supplied in *?seeds*. The third output column `P2` needs to generate all the entities (or nodes) that each seed member is connected to via the particular link type handled by this relation (`communication` in our example). Finally, the last column needs to generate the number of links of the particular type for each `P1,P2` node pair.

The query should generate exactly one row for each node pair which should contain the total number of links. Multiple entries as in

```
    ......
' ', 342, 214, 1
    ......
' ', 342, 214, 2
    ......
```

will not be aggregated by KOJAK (the second entry will simply be ignored). Duplicate entries that simply permute the arguments should be avoided but will be ignored if they occur, for example:

```
    ......
' ', 342, 214, 2
    ......
' ', 214, 342, 2
    ......
```

Such duplicates can not always be avoided by the query, since there might be links between two seed members.

Let us emphasize again that KOJAK performs best with high frequency transaction-type links such as, for example, communication events (see Section 4.1.2 [Link Data], page 10 for more discussion on that). If such data is available, it is important to compute accurate link counts for best performance. That is, it is much better for KOJAK to know that there were 23 phone calls and 17 financial transactions between two entities as opposed to just knowing that there were one or more but without knowing how many. The following example shows how such link counts can be computed for `'Communications'` links in the Ali Baba data. How this is done exactly will differ for each different evidence database schema and each link type. The parametric SQL query for `EDB-communicationCount` looks as follows:

```
SELECT ' ', P1, P2, COUNT(P2)
FROM (SELECT ENTITYID_ARG1 P1, ENTITYID_ARG2 P2, l.LINKID
      FROM LINK l, LINKTYPE lt, ENTITY e, ENTITYTYPE et
      WHERE l.LINKTYPEID=lt.LINKTYPEID
      AND lt.CYCCOLLECTION='Communications'
      AND l.ENTITYID_ARG2=e.ENTITYID
      AND e.ENTITYTYPEID=et.ENTITYTYPEID
      AND et.CYCCOLLECTION='Person'
      AND l.ENTITYID_ARG1 IN '?seeds'
      AND l.ENTITYID_ARG1='?party1'    -- optional
      AND l.ENTITYID_ARG2='?party2'    -- optional
     UNION
      SELECT l.ENTITYID_ARG2 P1, l.ENTITYID_ARG1 P2, l.LINKID
      FROM LINK l, LINKTYPE lt, ENTITY e, ENTITYTYPE et
      WHERE l.LINKTYPEID=lt.LINKTYPEID
      AND lt.CYCCOLLECTION='Communications'
      AND l.ENTITYID_ARG2=e.ENTITYID
      AND e.ENTITYTYPEID=et.ENTITYTYPEID
      AND et.CYCCOLLECTION='Person'
      AND ENTITYID_ARG2 IN '?seeds'
      AND l.ENTITYID_ARG1='?party2'    -- optional
      AND l.ENTITYID_ARG2='?party1')   -- optional
      SUBQUERY
GROUP BY P1, P2
```

The clauses marked as optional don't really need to be there, since the *?party1* and *?party2* variables will not be bound in queries generated by the MI component. They could be bound in manual queries, though, which is why they are added here. The query is structured as follows: the inner query named `SUBQUERY` retrieves all links in the `LINK` table whose link type is `Communications` and whose arguments are of type `Person`. Due to the fragmented nature of this schema this takes a somewhat complex join. The *?seeds* argument constrains the arguments of each link. Since a *?seeds* argument could be the first or second argument of such a link, we have to ask the query twice and union the results (note that an `OR` would not do the same thing here, since it would not guarantee to have only seed members for column `P1`). For each link we also output its `LINKID` to make sure that we get separate rows for each `Communications` link between two people. In the outer query we then group the result of the subquery first by seed members and then by second entity which allows us to compute accurate link counts for the final output column.

The other two link count relations `EDB-phoneCallCount` and `EDB-affiliatedWithCount` are defined similarly. `EDB-phoneCallCount` finds pairs of entities that are linked by having called the same phone number (which is more indirect than having called each other; however, direct phone call information doesn't seem to be available in this dataset). `EDB-affiliatedWithCount` uses the most complex SQL query, since it finds pairs of entities that are either directly `affiliatedWith` each other as well as those that are affiliated via an intermediary `Event` or `Organization`. Conceivably, these two different kinds of affiliations could be considered to be separate link types and handled by separate relations.

The way KOJAK computes and uses these link count statistics is one of the main reasons for its scalability. Instead of having to look at each individual link of a particular type between two entities, all of them can be abstracted into a single link count tuple. KOJAK also only computes these link counts in a very focused way starting from group seeds instead of having to do that for the whole database. Finally, by off-loading this data aggregation to the database, much less data has to be moved and the power of the relational database system can be exploited to perform these computations over large datasets. Nevertheless, depending on the nature and connectivity of the data, such queries can be time consuming and put significant stress on the database server.

7.3.5 'ali-baba-load-ontology.plm'

This file loads the Ali Baba ontology, the Ali Baba/groups ontology mapping and constraint rules. The value of `LoadOntologyScript` in '`config/ali-baba.dat`' should point to this file. It will be loaded when the KOJAK commands `initialize-kojak` or `load-kojak-ontology` (which see) are called in the run script. `load-kojak-ontology` allows the explicit specification of a script which would override what is specified in `LoadOntologyScript`.

PowerLoom KB files are loaded using PowerLoom's `load` command. The script is loaded into the `KOJAK` module which is case-insensitive. This means that commands can be spelled in upper or lower case. If the `load` command were to be used in a case-sensitive module such as `DATASET`, it would have to be spelled in all upper-case letters.

`load` takes the name of the file to be loaded as its argument. In this load script all files are specified via logical pathnames which will ensure their proper translation regarding of which version of KOJAK you are using and which OS it is run on (for more information on logical pathname syntax see Section 5.1 [Configuration File], page 14). You can also use physical pathnames appropriate for the OS KOJAK is run on. IMPORTANT: if you supply a physical Windows pathname you will need to double the \ character, since it is also the escape character for strings. For example:

```
(load "C:\\kojak\\my-db-ontology.plm")
```

The first file loaded is '`kbs/generic-groups-ontology.plm`' which defines KOJAK's generic groups ontology. This file always needs to be loaded as the first file in every configuration (future versions of KOJAK might do that automatically).

The other two files load (1) the Ali Baba ontology, and (2) the mappings between the Ali Baba ontology and KOJAK's generic groups ontology as well as a seed constraint rule. Note that the Ali Baba ontology file needs to be loaded first, since the objects defined in it are used in the seed constraints file.

7.3.6 'ali-baba-load-edb-schema.plm'

This file loads the Ali Baba EDB schema and mapping. The value of `LoadEDBSchemaScript` in '`config/ali-baba.dat`' should point to this file. It will be loaded when the KOJAK commands `initialize-kojak` or `load-EDB-schema` (which see) are called in the run script. `load-EDB-schema` allows the explicit specification of a script which would override what is specified in `LoadEDBSchemaScript`.

Since only one file '`kbs/ali-baba-edb-schema.plm`' is loaded by this script, `LoadEDBSchemaScript` could be pointed to that file directly as opposed to loading it via this script. However, it is conceivable that one might want to spread an EDB schema definition over multiple files in which case the load script needs to be used.

7.3.7 'ali-baba-load-data.plm'

This file loads any necessary data (e.g., seed information) and related information (e.g., names) into KOJAK. It also usually contains the data import rules that link up the GROUPS/linkCount relation used by KOJAK's MI component to individual parametric query relations defined in the EDB schema file. This file does not define any more mapping information but instead uses the various mappings defined in previous files.

The first command in 'scripts/ali-baba-load-data.plm' retrieves a set of seed group entities based on the names given in the setof term. The surrounding assert-from-query command then creates a ThreatGroup assertion for each of these entities and also assigns a nameString for them (the latter is not actually used). For example, running this command interactively generates the following assertions (see the EDB schema file section for more information on assert-from-query):

```
|= (ASSERT-FROM-QUERY
 (RETRIEVE all (?pred ?o ?name)
           (AND (= ?pred |ThreatGroup|)
                (MEMBER-OF ?name (SETOF "Al Qaeda" "Needabaath"
                                        "Pavdayeen" "Ali Baba"))
                (BOUND-VARIABLES ?name)
                (EDB-Organization ?o ?name)))
 :pattern (KAPPA (?pred ?o ?name)
            (AND (HOLDS ?pred ?o)
                 (nameString ?o ?name))))

(|P|(nameString 1290 "Al Qaeda") |P|(KnownGroup 1290)
 |P|(nameString 1294 "Ali Baba") |P|(KnownGroup 1294)
 |P|(nameString 1291 "Needabaath") |P|(KnownGroup 1291)
 |P|(nameString 1292 "Pavdayeen") |P|(KnownGroup 1292))
```

The EDB-Organization relation defined in the EDB schema file is used to import this information from the Ali Baba database. Note that due to the prior synonym assertion for ThreatGroup this actually results in GROUPS/KnownGroup assertions.

The next command asserts know members for the seed groups imported above. Note that links of type memberAgents in the Ali Baba LINK table actually use the reverse argument order of the memberAgents relation defined in the EAGLE ontology (which is version we are using). Therefore the group argument *?g* and member argument *?m* need to be flipped before we create the assertion. The EDB schema relation EDB-LinkOfType is used to access membership links directly on the Ali Baba database. This is what the command would produce when run interactively; again, due to the prior synonym assertions this actually generates GROUPS/groupMember links:

```
|= (ASSERT-FROM-QUERY
 (RETRIEVE ALL (?pred ?g ?m)
           (EXISTS ?id
             (AND (ThreatGroup ?g)
                  (BOUND-VARIABLES ?g)
                  (= ?pred memberAgents)
                  (EDB-LinkOfType ?id ?m ?g (EDB-LinkTypeID ?pred)))))
```

```
 :relation HOLDS)

(|P|(groupMember 1292 732) |P|(groupMember 1292 782)
 |P|(groupMember 1292 536) |P|(groupMember 1292 882)
    ......112 assertions suppressed......
 |P|(groupMember 1290 1229) |P|(groupMember 1290 1230)
 |P|(groupMember 1290 1231) |P|(groupMember 1290 1232))
```

This command could have been slightly simplified as follows:

```
(ASSERT-FROM-QUERY
 (RETRIEVE ALL (?g ?m)
           (EXISTS ?id
             (AND (ThreatGroup ?g)
                  (BOUND-VARIABLES ?g)
                  (EDB-LinkOfType ?id ?m ?g (EDB-LinkTypeID memberAgents)))))
 :relation memberAgents)
```

Note the use of the `bound-variables` clause to make sure the group argument *?g* is bound before `EDB-LinkOfType` is queried. PowerLoom's query optimizer might move clauses around which could result in the situation where `EDB-LinkOfType` is queried first which would logically be the same but potentially retrieve a lot of unnecessary intermediate information (all members of any groups not just the four threat groups we care about). Unfortunately, the current version of the query optimizer does not know about potential result set sizes on an external database which might result in incorrect clause order "optimizations".

Alternatively, group seed information could have been supplied in a CSV file in which case these import commands should be commented or deleted, since they would most probably conflict with that.

The next command imports links of type `Terrorism` and asserts them via the `participatedInTerrorism` relation defined in the ontology. Remember that '`kbs/ali-baba-seed-constraints.plm`' defines a rule that can exploit these links to infer additional group seed members.

The next section defines three data import rules that maps `GROUPS/linkCount` used by the MI component onto the appropriate EDB link count relation defined in the EDB schema file. The rules dispatch based on the link type argument which is one of the three expansion link type names `phoneCall`, `communication` and `affiliatedWith` (see the EDB schema section for more information on this).

The last rule does not have anything to do with loading data but simply supports more readable output generation. For lack of a better place it was added here. By default, entities such as groups and their members are reported via their IDs in KOJAK's report file. For the Ali Baba data this makes for somewhat unreadable output, since all IDs are simply numeric keys. To allow one to substitute names for IDs if desired, KOJAK uses the following mechanism: For each entity it is about to report, it tries to lookup a name via `GROUPS/nameString`. If no name could be found, it outputs the entity ID, otherwise, it outputs the name instead. Such names can be simply asserted or defined via a rule as done here. The rule first looks up the name of an organization or person with help of the EDB relations `EDB-Organization` and `EDB-AliBabaName`. If a name is found it concatenates

the entity ID to the end to ensure it is unique. Concatenation is done via PowerLoom's `string-concatenate` function.

7.3.8 'ali-baba-run-kojak.plm'

Finally, the run file determines the sequence of KOJAK commands executed to analyze a dataset such as Ali Baba. This file primarily controls what command parameters are taken from the configuration, whether (intermediary) results are deposited in the hypothesis tables of the KOJAK database, and where and how reports should be generated.

The Ali Baba run file is highly generic so we do not explain it in detail here. Refer to Chapter 6 [Running the KOJAK Group Finder], page 24 for more information on how to run KOJAK and how the run file controls this process.

8 KOJAK Commands

This section lists all KOJAK commands that can be called in KOJAK run scripts in alphabetical order. Additionally, all PowerLoom commands documented in the PowerLoom manual (see http://www.isi.edu/isd/LOOM/PowerLoom/) can also be called. KOJAK commands all reside in the KOJAK module and can usually be called without the `KOJAK/` module prefix as long as the KOJAK module is visible from the current module (set at the beginning of a script with `in-module`). The KOJAK module is case-insensitive, so commands can be written with any mixture of upper and lowercase characters.

All commands need to be written in a Lisp-ish input syntax which is the standard input syntax used by the underlying PowerLoom system. Knowledge of PowerLoom should not be required, but see http://www.isi.edu/isd/LOOM/PowerLoom/ for more information if necessary. Commands take zero or more required arguments and a set of options specified as keyword/value pairs. Default values for unspecified options are often taken from the configuration file. Some commands such as `get-parameter` return values that can be given as input to other commands. Here is an example command invocation:

```
(db-save-groups (get-all-groups)
                :dataset (get-parameter "DatasetName")
                :source "AV" :category "F")
```

In the descriptions below, parameter types are given in all-caps following a parameter name. If a function returns a value, the return type is given after the : following the parameter list. `options` is always a list of optional arguments that need to be supplied as keyword/value pairs. The documentation of a command describes which options are legal.

`add-parameter-value` ((*parameter* NAME) (*value* OBJECT)) : [Command]
Add *value* to the end of *parameters* (a string or symbol) value list in the configuration table. Coerces the current value to a list or initializes the list if it is as yet undefined. Allows incremental addition of values to list-valued parameters. Note that *parameter* is evaluated and will need to be quoted if supplied as a symbol. Symbols will also be upcased if this command is run in a non-case-sensitive module such as KOJAK.

`compute-avg-group-connectivity` [Command]
((*groups* (CONS OF GENERIC-GROUP))) : FLOAT
Compute the average number of groups in *groups* each of *groups* agents belongs to and return the result.

`db-assert-connection-info` ((*dbInstance* OBJECT) [Command]
&rest (*options* OBJECT)) :
Assert DB connectivity information about the DB instance *dbInstance* (for example, `edb`) according to *options*. Looks up *dbInstance* in the module specified by the :module option (defaults to current module) and retracts any preexisting connection information. New assertions are also made in that module. Understands the following options: :connection-string, :dsn, :db, :host, :port, :user, :password and :module. At least one of :connection-string and :dsn must be specified. If a connection string is given, all other connection options are ignored. The values of :user and :password default to the values of `DBUser` and `DBPassword` specified in the configuration file.

`db-clear-hypotheses` () : [Command]
Clear all hypothesis and associated tables in the KOJAK database.

`db-load-groups` (**&rest** *(options* OBJECT)) : (CONS OF GENERIC-GROUP) [Command]
Load groups from the KOJAK database according to *options* and return the result. Groups will also be interned into the global objects table. Understands the following metadata options to select groups from the database (all of which can be NULL): :dataset, :source and :category. Groups will be loaded into the module sepcified by the optional :module option (defaults to `DATASET`). If :MI-groups? is TRUE use `db-load-MI-groups` to load groups which understands some additional options and knows how to fixup results. This is a kludge to deal with preservation of some of the group annotations that Jafar's MI code doesn't carry over to extended groups.

`db-save-configuration` () : [Command]
Store all current KOJAK configuration and parameter information in the KOJAK database.

`db-save-groups` ((*groups* (CONS OF GENERIC-GROUP)) [Command]
&rest *(options* OBJECT)) :
Saves *groups* to the KOJAK database. If *groups* is NULL all groups currently loaded in the global objects table will be saved. Understands the following metadata *options* that will be stored with each group: :dataset, :source and :category which all have to be supplied as strings. :source and :category default to the the empty string. :dataset defaults to the `DatasetName` property given in the configuration file. Groups will be encoded relative to the module sepcified by the optional :module option (defaults to `DATASET`).

`extend-groups` ((*groups* (CONS OF GENERIC-GROUP)) [Command]
&rest *(options* OBJECT)) :
Extend *groups* based on relevant link types and connection strength computations specified in *options*. If *groups* is NULL all groups currently loaded into the objects table will be extended. Understands the following *options*:

:mode can be either :individual or :collective which controls whether the extension graph is grown for each group indiviually or fow all groups collectively. Collective extension requires more memory but will make MI computation somewhat more accurate in certain cases where individuals are members of multiple groups. Boosting is not yet supported in :collective mode.

:link-types specifies a single or list of link types that should be considered to expand the graph and compute connection strength. These link types must match the ones in the link data. Using different link types particularly benefits the MI computation, since it can differentiate links of different types. A special `anyLink` type is also supported for EDB data, which treats all links as the same. Link types can be supplied as symbols or strings. If none are given the values of `ExpansionLinkTypes` are used.

:methods is the list of connection strength computations that should be used. Currently supported are :mi, :group-mi, :mi-or-group-mi, :link-count and :connectivity

(which is computed implicitly as part of link counting). Defaults to the value of `ExpansionMethods` (or (:connectivity :link-count :group-mi) if not specified). :methods can be specified as symbols, keywords or strings.

:measures is a list of measures computed by the specified :methods that will be used to determine the final connection strength. If there are more than one measure, they will be averaged (bagging). Each strength computation method produces a number of associated measures such as maximum and average values, etc. See the documentation of `ExpansionMeasures` for a full list.

:boost?, :boost-min-seeds, :boost-factor, :boost-min-step and :boost-max-cycles control boosting. See the documentation of the corresponding configuration parameters for more information.

:depth specifies to which depth the graph around seed members should be expanded. Defaults to the value of `ExpansionDepth` (or 1 if not specified). IMPORTANT: due to graph fanout this value has a big impact on run time. Use values larger than 1 only for cases with very small seed groups (e.g., 1-5 members) and increase it only one step at a time if needed.

:top-N specifies how many of the members of an extended group should be kept after the strength computation is complete. If it is not supplied, all members will be kept.

:module can be specified to be the module in which group extension will be performed. It defaults to `DATASET`. All new extended members of each group will be interned into the global objects table.

`get-all-groups` () : (CONS OF GENERIC-GROUP) [Command]
Return all groups currently loaded into the objects table. The resulting list is a sorted list of known groups followed by a sorted list of unknown groups.

`get-all-known-groups` () : (CONS OF GENERIC-GROUP) [Command]
Return all known groups (i.e., groups with known name) currently loaded into the objects table. The resulting list will be sorted in ascending order by group name.

`get-all-objects` ((*kind* OBJECT-KIND)) : (CONS OF GENERIC-OBJECT) [Command]
Return a list of all currently defined objects of *kind* (a keyword).

`get-all-unknown-groups` () : (CONS OF GENERIC-GROUP) [Command]
Return all unknown groups (i.e., hypothesized groups whose name is unknown) currently loaded into the objects table. The resulting list will be sorted in ascending order by group name.

`get-nof-objects` ((*kind* OBJECT-KIND)) : INTEGER [Command]
Return the number of objects of *kind* (a keyword) stored in the object table.

`get-parameter` ((*parameter* NAME) **&rest** (*defaultValue* OBJECT)) : OBJECT [Command]
Lookup *parameter* (a string or symbol) in the configuration table and return its value. If it is undefined, return the optional *defaultValue*. Note that *parameter* is evaluated and will need to be quoted if supplied as a symbol. Symbols will also be upcased if this command is run in a non-case- sensitive module such as KOJAK.

`hypothesize-unknown-groups` (**&rest** *(options* OBJECT*)*) *:* (CONS OF GENERIC-GROUP) [Command]

Hypothesize seeds for unknown groups by collecting event participants into groups and return the list of hypothesized seed groups. Group and event information is taken from the groups and events currently loaded into the global objects table (i.e., this needs to run after `retrieve-groups` and `retrieve-events`). The set of events can be restricted by supplying them in the :events option. Runs in the the module specified by the :module option (defaults to `DATASET`). A :mode option can be supplied as :aggressive, :conservative (the default) or :very-conservative to control the manner in which unknown groups are hypothesized. :conservative will use events whose responsible group is not known but some or all of the event participants might have known group associations. :aggressive will collect event participants of all events into groups (even if the responsible group is known). :very-conservative is like :conservative but additionally requires that none of the participants have any known group associations.

`import-data-into-edb` (**&rest** *(options* OBJECT*)*) *:* [Command]

Import data from text files into the KOJAK EDB. This is useful if datasets are large but not available directly from an existing database. Also it allows for multiple runs without having to load the data every time. Currently only import from CSV files is supported, i.e., all source files must have a `.csv` extension. The :DATA-SOURCE option can be used to specify either directly the source file or a data key from the configuration file (the default is `Data`). If a data key is specified, all its values will be loaded. If :CLEAR-EDB? is supplied as TRUE, the current content of the EDB will be cleared before new content will be loaded. Without that, data will be added incrementally.

`initialize-kojak` (**&rest** *(options* OBJECT*)*) *:* [Command]

Initialize the KOJAK environment with configuration, ontology and EDB schema information. If :CONFIG-FILE is supplied re/load configuration information from the given file. The ontology will be loaded if :LOAD-ONTOLOGY? is TRUE (the default). Otherwise, the ontology can be loaded explicitly by calling `load-kojak-ontology` (which see). The EDB schema will be loaded if :LOAD-EDB-SCHEMA? is TRUE (the default). Otherwise, the schema can be loaded explicitly with `load-EDB-schema` (which see). If :CLEAR-OLD-HYPOTHESES? is TRUE (defaults to the value of the configuration parameter `ClearOldHypotheses`), old hypotheses will be cleared from the hypothesis tables of the KOJAK DB.

`kojak-log` (*(logLevel* KEYWORD) **&rest** *(message* OBJECT*)*) *:* [Command]

Log all elements of *message* to the current log stream if *logLevel* is lower or the same as the current KOJAK log level.

`kojak-log-objects` (*(logLevel* KEYWORD) **&rest** *(message* OBJECT*)*) *:* [Command]

Log all elements of *message* to the current log stream if *logLevel* is lower or the same as the current KOJAK log level. The last element of *message* is assumed to be a list of GENERIC-OBJECTs which will be printed with `print-object-list`.

`load-edb-schema` (**&rest** *(options* OBJECT*)*) *:* [Command]

Load relevant EDB schema & ontology information into PowerLoom to facilitate access to various EDB tables. Loading is done by loading the script pointed to

by the :script option which defaults to the value of the configuration parameter `LoadEDBSchemaScript` or `KJ:scripts;load-edb-schema.plm` if not specified in the configuration. Before the script is loaded, the `edb` instance will be pointed to the datbase identified by the :db option which should be a connection string. If :db is not specified, one of the RDBMS input data sources specified in the configuration file will be used (for example, one of the values of `PrimaryData`). The instance will be set in the module specified by the :module option which defaults to `EDB`. This module should match or be visible from the module used by the EDB schema script. If no RDBMS data source is specified, `edb` will remain unconnected which is ok as long as the EDB schema script does not make reference to it.

`load-iet-evidence-file` ((*file* FILE-NAME) **&rest** (*options* OBJECT)) : [Command]
Load the IET evidence or report file *file* containing synthetic evidence or reports generated by an LD component. Assertions are created in the module specified by the :module options (defaults to `DATASET`). If :assert-envelopes? is supplied as TRUE, assertion wrappers such as `containsInformation` or `reportContent` will be asserted also. Defines undefined concepts and relations on the fly to handle incompletely defined LD ontologies.

`load-csv-file` ((*file* FILE-NAME) **&rest** (*options* OBJECT)) : [Command]
Load evidence tuples from a comma separated values *file* into PowerLoom. Assertions are created in the module specified by the :module option (defaults to `DATASET`). Defines undefined concepts and relations on the fly to handle incompletely defined LD ontologies.

`load-data` (**&rest** (*options* OBJECT)) : [Command]
Load all primary data into the module specified by the :module option which defaults to `DATASET` (specific hard-coded PL and LD modules are used for certain IET EDBs). Loading data might mean actually loading the data into the PowerLoom KR&R system or only loading certain parts + mapping rules that connect PowerLoom to the evidence database. For large datasets, it is recommended that the data is stored in the database and only accessed selectively through PowerLoom's RDBMS interface. What data gets loaded and how can be controlled by a variety of configuration parameters and scripts (see `configuration.dat` for more info).

`load-kojak-configuration` (**&rest** (*options* OBJECT)) : [Command]
Load configuration information from :config-file (which defaults to the most recently loaded configuration file or `configuration.dat`). A list of command line arguments can be supplied via the :cmd-line-args which override any information with the same key provided in the configuration file. The syntax is a list of strings, for example, '("-c" "myconfig.dat").

`load-kojak-ontology` (**&rest** (*options* OBJECT)) : [Command]
Load the KOJAK ontology unless it is already loaded. Loads the ontology by loading the value of :SCRIPT which defaults to the value of the configuration parameter `LoadOntologyScript` (or `KJ:scripts;load-ontology.plm` if not specified in the configuration).

`merge-groups` (**&rest** (*options* OBJECT)) : (CONS OF GENERIC-GROUP) [Command]
Write Me

`print-parameters` () : [Command]
Print all current configuration and parameter information to standard output.

`print-similarity-matrix` ((*logLevel* KEYWORD) [Command]
(*baseGroups* (CONS OF GENERIC-GROUP))
(*otherGroups* (CONS OF GENERIC-GROUP)) **&rest** (*options* OBJECT)) :
If *logLevel* is <= the current log level print the similarity matrix matching *baseGroups* to *otherGroups* to the log output stream.

`report-groups` ((*groups* (CONS OF GENERIC-GROUP)) [Command]
&rest (*options* OBJECT)) :
Report each of *groups* to a report file specified by the :file option (defaults to the `ReportFile` parameter). If a :directory option is given (defaults to `ReportDirectory` or the empty string), it will be prepended to form the full report file name. :if-exists can be used to specify what to do if the report file already exists. Supported values are :append, :supersede (the default) or :error. :format (defaults to `ReportFormat`) controls the output format. Supported formats are `CSV-Table`, `IET-Report`, `IET-Y3-Report` and `IET-Y3-Report`. If no format is specified it will be inferred from the report file extension (`.csv` is interpreted as `CSV-Table`, everything else is interpreted as `IET-Report`). The :threshold-function option can be used to point to a specialized STELLA thresholding function. Without that, thresholding will be performed relative to the various thresholding parameters specified in the configuration file. :report-factor can be used to specify a fraction of group members to be reported, e.g., to force a certain percentage of over or underreporting. For example, 1.2 would specify 20% overreporting on top of what thresholding would normally produce (the default is 1.0). If :report-member-weights? is TRUE, hypothesized group members will be annotated with a weight indicating how strongly they are deemed to be a group member (defaults to the value of `ReportMemberWeights`). If weights are reported, the :weight-decimals option determines how many decimals behind the decimal point are used to print the weight in fixed-point format (defaults to the value of `WeightDecimals` or 6). The computed number of to-be-reported members will usually wind up somewhere within a run of members with equal membership weights. The value of :last-run-cutoff determines how this last run should be cut. The default is :inclusive which will keep all members of this last run (see the documentation of the `LastRunCutoff` parameter for more information). If :report-names? is TRUE groups and members will be reported with their namestrings if appropriately defined by `GROUPS/nameString` assertions or rules (defaults to the value of `ReportNames`). :module specifies the module relative to which output will be peformed (defaults to `DATASET`). Arbitrary other options can also be specified which are passed on to :threshold-function.

`reset-kojak` () : [Command]
Reset KOJAK to its initial state but preserve the currently loaded configuration and configuration file setting. `load-kojak-configuration` or `initialize-kojak` can be used subsequently to change the configuration if necessary. CAUTION: This will destroy all loaded data and ontologies and might break other loaded STELLA systems if they do reference KOJAK symbols in their code.

`retrieve-events` (**&rest** *(options* OBJECT)) : (CONS OF GENERIC-EVENT) [Command]

Retrieve all events and their participants from the PowerLoom KB in the module specified by the :module option (defaults to `DATASET`) and return them as a list of event objects. Also interns events in the global object table as a side effect.

`retrieve-groups` (**&rest** *(options* OBJECT)) : (CONS OF GENERIC-GROUP) [Command]

Retrieve all groups and their members from the PowerLoom KB in the module specified by the :module option (defaults to `DATASET`) and return them as a list of group objects. Also interns the retrieved groups in the global object table as a side effect.

`retrieve-modes` (**&rest** *(options* OBJECT)) : [Command]

For each group supplied to the :groups option (defaults to all known groups) retrieve all the vulnerability modes that group uses and assign the :MODES-USED attribute accordingly. Performs retrieval in :module which defaults to `DATASET`.

`run-kojak` () : [Command]

Top level invocation of the KOJAK Group Finder. All the actual work is done by the commands in the top-level run script specified by the `RunKojakScript` configuration property.

`set-parameter` ((*parameter* NAME) (*value* OBJECT)) : [Command]

Set *parameter* (a string or symbol) in the configuration table to *value*. Note that *parameter* is evaluated and will need to be quoted if supplied as a symbol. Symbols will also be upcased if this command is run in a non-case-sensitive module such as KOJAK.

`threshold-groups` ((*groups* (CONS OF GENERIC-GROUP)) **&rest** *(options* OBJECT)) : [Command]

For each group in *groups* compute the number of members to report or output and destructively remove all extraneous members. If *groups* is NULL all groups loaded into the current objects table will be thresholded. Uses the value of the :threshold-function option to do the actual computation (defaults to `compute-nof-members-to-report-from-configuration`). If :report-factor is given determine how many members should be reported and then multiply that by the given factor (default = 1.0) which allows over or underreporting. The computed number of to-be-reported members will usually wind up somewhere within a run of members with equal membership weights. The value of :last-run-cutoff determines how this last run should be cut. The default is :inclusive which will keep all members of this last run (see the documentation of the `LastRunCutoff` parameter for more information). Performs thresholding in the module specified by :module which defaults to `DATASET`. Any other options will be passed on to the threshold computation function.

8.1 Important PowerLoom Commands

The following commands are an important subset of PowerLoom commands that are used to define and load ontologies, make assertions, queries, etc. The full set of PowerLoom commands can be found in the PowerLoom manual (see http://www.isi.edu/isd/LOOM/PowerLoom/).

`ask` (**&rest** *(proposition***&options** PARSE-TREE*))* : TRUTH-VALUE [N-Command]
Perform inference to determine whether the proposition specified in *proposition&options* is true. Return the truth-value found. `ask` will spend most of its effort to determine whether the proposition is true and only a little effort via shallow inference strategies to determine whether it is false. To find out whether a proposition is false with full inference effort `ask` its negation.

KIF example: `(ask (happy Fred))` will return TRUE if Fred was indeed found to be happy. Note, that for this query to run, the logic constant `Fred` and the relation `happy` must already be defined (see `assert`). Use `(set/unset-feature goal-trace)` to en/disable goal tracing of the inference engine.

The `ask` command supports the following options: `:TIMEOUT` is an integer or floating point time limit, specified in seconds. For example, the command `(ask (nervous Fred) :timeout 2.0)` will cease inference after two seconds if a proof has not been found by then. If the `:DONT-OPTIMIZE?` is given as TRUE, it tells PowerLoom to not optimize the order of clauses in the query before evaluating it. This is useful for cases where a specific evaluation order of the clauses is required (or the optimizer doesn't do the right thing). If `:THREE-VALUED` is given as TRUE, PowerLoom will try to prove the negation of the query with full effort in case the given query returned UNKNOWN. By default, PowerLoom uses full effort to prove the query as stated and only a little opportunistic effort to see whether it is actually false.

`assert` ((*proposition* PARSE-TREE)) : OBJECT [N-Command]
Assert the truth of *proposition*. Return the asserted proposition object. KIF example: "(assert (happy Fred))" asserts that Fred is indeed happy. Note that for this assertion to succeed, the relation `happy` must already be defined. If the constant `Fred` has not yet been created, it is automatically created as a side-effect of calling `assert`.

`assert-from-query` ((*query* CONS) **&rest** *(options* OBJECT*))* : (CONS OF PROPOSITION) [N-Command]
Evaluate *query*, instantiate the query proposition for each generated solution and assert the resulting propositions. The accepted syntax is as follows:

```
(assert-from-query <query-command>
                   [:relation <relation-name>]
                   [:pattern <description-term>]
                   [:module <module-name>])
```

<query-command> has to be a strict or partial retrieval command. If a :relation option is supplied, <relation-name> is used as the relation of the resulting propositions. In this case the bindings of each solution will become arguments to the specified relation in the order of *query*s output variables (the arities have to match). The :pattern option is a generalization of this mechanism that specifies an arbitrary proposition pattern to be instantiated by the query's solution. In this case <description-term> has to be a SETOFALL or KAPPA expression whose IO-variables will be bound in sequence to the bindings of a query solution to generate the resulting proposition. Finally, if a :module option is specified, the assertions will be generated in that module. Note that for this to work the relations referenced in the query proposition or pattern have to be visible in the module. Also, instances will not be copied to the target

module, therefore, the resulting propositions might reference external out-of-module objects in case they are not visible there. Here are some examples:

```
(assert-from-query (retrieve all (foo ?x ?y)))
(assert-from-query (retrieve all (?y ?x)
                             (exists ?z
                               (and (foo ?x ?z)
                                    (foo ?z ?y))))
                   :relation bar :module other)
(assert-from-query
  (retrieve all (and (relation ?x) (symmetric ?x)))
  :pattern (kappa (?pred)
             (forall (?x ?y)
               (=> (holds ?pred ?x ?y)
                   (holds ?pred ?y ?x))))))
```

`cc` (**&rest** *(name* NAME*)*) : CONTEXT [N-Command]

Change the current context to the one named *name*. Return the value of the new current context. If no *name* is supplied, return the pre-existing value of the current context. `cc` is a no-op if the context reference cannot be successfully evaluated.

`clear-module` (**&rest** *(name* NAME*)*) : [N-Command]

Destroy all objects belonging to module *name* or any of its children. If no *name* is supplied, the current module will be cleared after confirming with the user. Important modules such as STELLA are protected against accidental clearing.

`defconcept` (**&rest** *(args* PARSE-TREE*)*) : NAMED-DESCRIPTION [N-Command]

Define (or redefine) a concept. The accepted syntax is:

```
(defconcept <conceptconst> [(<var> <parent>*)]
   [:documentation <string>]
   [:<= <sentence>] | [:=> <sentence>] |
   [:<<= <sentence>] | [:=>> <sentence>] |
   [:<=> <sentence>] | [:<=>> <sentence>] | [:<<=> <sentence>] |
   [:<<=>> <sentence>] |
   [:axioms {<sentence> | (<sentence>+)}] |
   <keyword-option>*)
```

Declaration of a concept variable `<var>` is optional, unless any implication (arrow) options are supplied that need to reference it. A possibly empty list of concept names following `<var>` is taken as the list of parents of `<conceptconst>`. Alternatively, parents can be specified via the `:=>` option. If no parents are specified, the parent of `<conceptconst>` is taken to be THING. `<keyword-option>` represents a keyword followed by a value that states an assertion about `<conceptconst>`. See `defrelation` for a description of `<keyword-option>`s.

`deffunction` (**&rest** *(args* PARSE-TREE*)*) : NAMED-DESCRIPTION [N-Command]

Define (or redefine) a logic function. The accepted syntax is:

```
(deffunction <funconst> (<vardecl>+) [:-> <vardecl>]
   [:documentation <string>]
```

```
[:<= <sentence>] | [:=> <sentence>] |
[:<<= <sentence>] | [:=>> <sentence>] |
[:<=> <sentence>] | [:<=>> <sentence>] |
[:<<=> <sentence>] | [:<<=>> <sentence>] |
[:axioms {<sentence> | (<sentence>+)}]
[<keyword-option>*])
```

Function parameters can be typed or untyped. If the `:->` option is supplied, it specifies the output variable of the function. Otherwise, the last variable in the parameter list is used as the output variable. See `defrelation` for a description of `<keyword-option>`s.

`defobject` (**&rest** *(args* PARSE-TREE*)*) : LOGIC-OBJECT [N-Command]

Define (or redefine) a logic instance. The accepted syntax is:

```
(defobject <constant>
   [:documentation <string>]
   [<keyword-option>*])
```

`<keyword-option>` represents a keyword followed by a value that states an assertion about `<constant>`. See `defrelation` for a description of `<keyword-option>`s.

`defobject` provides a sugar-coated way to assert a collection of facts about a logic constant, but otherwise adds nothing in terms of functionality.

`defrelation` (**&rest** *(args* PARSE-TREE*)*) : NAMED-DESCRIPTION [N-Command]

Define (or redefine) a logic relation. The accepted syntax is:

```
(defrelation <relconst> (<vardecl>+)
   [:documentation <string>]
   [:<= <sentence>] | [:=> <sentence>] |
   [:<<= <sentence>] | [:=>> <sentence>] |
   [:<=> <sentence>] | [:<=>> <sentence>] |
   [:<<=> <sentence>] | [:<<=>> <sentence>] |
   [:axioms {<sentence> | (<sentence>+)}]
   [<keyword-option>*])
```

Relation parameters can be typed or untyped. `<keyword-option>` represents a keyword followed by a value that states an assertion about `<relconst>`. For example, including the option `:foo bar` states that the proposition `(foo <relconst> bar)` is true. `:foo (bar fum)` states that both `(foo <relconst> bar)` and `(foo <relconst> fum)` are true. `:foo true` states that `(foo <relconst>)` is true, `:foo false` states that `(not (foo <relconst>))` is true.

`in-module` ((*name* NAME)) : MODULE [N-Command]

Change the current module to the module named *name*.

`load` ((*file* STRING) **&rest** *(options* OBJECT*)*) : [Command]

Read logic commands from *file* and evaluate them. By default, this will check for each asserted proposition whether an equivalent proposition already exists and, if so, not assert the duplicate. These duplicate checks are somewhat expensive though and can be skipped by setting the option :check-duplicates? to false. This can save time when loading large KBs where it is known that no duplicate assertions exist in a file.

`retrieve` (**&rest** *(query* PARSE-TREE*)*) : QUERY-ITERATOR [N-Command]

Retrieve elements of a relation (tuples) that satisfy a proposition. The accepted syntax is:

```
(retrieve [<integer> | all]
          [[{<vardecl> | (<vardecl>+)}]
          <proposition>])
```

The variables and proposition are similar to an `exists` sentence or **kappa** term without the explicit quantifier. If variables are declared, they must match the free variables referenced by <proposition>. Otherwise, the free variables referenced in <proposition> will be used as the query variables. If <proposition> is omitted, the most recently asked query will be continued.

A solution is a set of bindings for the listed variables for which <proposition> is true. The optional first argument controls how many solutions should be generated before control is returned. The keyword `all` indicates that all solutions should be generated. By default, `retrieve` returns after it has found one new solution or if it cannot find any more solutions.

`retrieve` returns an iterator which saves all the necessary state of a query and stores all generated solutions. When used interactively, the returned iterator will print out with the set of solutions collected so far. Calling `retrieve` without any arguments (or only with the first argument) will generate one (or more) solutions to the most recently asked query.

KIF examples:

```
(retrieve (happy ?x))
```

will try to find one happy entity and store it in the returned query iterator.

```
(retrieve 10 (happy ?x))
```

will try to find 10 happy entities.

```
(retrieve 10)
```

will try to find the next 10 happy entities..

```
(retrieve all (happy ?x))
```

will find all happy entities.

```
(retrieve all (?x Person) (happy ?x))
```

will to find all happy people. Here we used the optional retrieve variable syntax to restrict the acceptable solutions. The above is equivalent to the following query:

```
(retrieve all (and (Person ?x) (happy ?x)))
```

Similarly,

```
(retrieve all (?x Person))
(retrieve all (Person ?x))
(retrieve all ?x (Person ?x))
```

will find all people. Note that in the first case we only specify a query variable and its type but omit the logic sentence which defaults to TRUE. This somewhat impoverished looking query can be paraphrased as "retrieve all ?x of type Person such that TRUE."

```
(retrieve ?x (or (happy ?x) (parent-of Fred ?x)))
```

will try to find a person that is happy or has Fred as a parent.

```
(retrieve (?y ?x) (parent-of ?x ?y))
```

will try to find the one pair of parent/child and return it in the order of child/parent.

```
(retrieve all (?x Person)
          (exists (?y Person) (parent-of ?x ?y)))
```

will generate the set of all parents. Note, that for these queries to run, the class `Person`, the relations `happy` and `parent-of`, and the logic constant `Fred` must already be defined (see `assert`).

Use `(set/unset-feature trace-subgoals)` to en/disable goal tracing of the inference engine.

8.2 Important RDBMS Commands

The following commands are an important subset of PowerLoom commands that deal with its relational database interface. These commands are particularly important for advanced configuration with an external relational evidence database.

`defdb` (**&rest** *(args* PARSE-TREE*)*) : LOGIC-OBJECT [N-Command]

Define (or redefine) a database instance. The accepted syntax is:

```
(defdb <dbconst>
   [:protocol <protocol string>]
   [:server-type <server type string>]
   [:dsn <ODBC data-source name string>]
   [:odbc-connection-string <ODBC connection string>]
   [:jdbc-connection-string <JDBC connection string>]
   [:connection-string <connection string>] ;; deprecated
   [:host <server host string>]
   [:port <server host port>]
   [:user <user name string>]
   [:password <password string>]
   [:db-name <database name string>]
   [<keyword-option>*])
```

Connection information can be asserted explicitly later or be done programmatically, as long as it is done before the first time a connection attempt is made (e.g., during a query). This information is passed to `SDBC/connect` (which see for more documentation on ways to specify a connection). For example, for ODBC a :DSN, an :ODBC-CONNECTION-STRING or an appropriate combination of :HOST, :PORT, :USER, :PASSWORD AND :DB-NAME can be used. For JDBC a :JDBC-CONNECTION-STRING or combination of :HOST, :PORT, etc. can be used. See `defobject` for additional legal keyword options.

`defquery` (**&rest** *(args* PARSE-TREE*)*) : LOGIC-OBJECT [N-Command]

Define (or redefine) an (external) query and map it to a relation. The accepted syntax is:

```
(defquery <relconst> (<vardecl>+)
   {:query-pattern (<query-evaluator> <dbconst> <arguments-string>)}*
   [<keyword-option>*]).
```

<query-evaluator> has to be a `Computed-Procedure` that takes the following three arguments: the <relconst> query proposition, the database object <dbconst> and the arguments string. It needs to return an iterator that generates CONS tuples whose elements will be bound in sequence to <relconst>'s arguments.

See `defrelation` for other legal keyword options.

`deftable` (**&rest** *(args* PARSE-TREE*)*) *:* LOGIC-OBJECT [N-Command]

Define (or redefine) a database table and map it to a relation. The accepted syntax is:

```
(deftable <relconst> <dbconst> <tablename> (<columndecl>+)
   [<keyword-option>*]).
```

<columndecl>'s specify the names (and optionally types) of the columns of the database table. They have the same form as a PowerLoom <vardecl>, but column names can be specified with or without question marks. If there is a question mark it will be stripped to determine the column name, otherwise a question mark will be added to generate the relation variable. The optional type specifies the domain of that argument and controls any necessary coercion. The name of each column HAS TO MATCH one of the columns of <tablename>, however, the order is irrelevant and arbitrary projections can be defined by only specifying some subset of <tablename>'s columns.

See `defrelation` for legal keyword options.

8.3 XTIE-Specific Commands for IET Datasets

The following commands are specific to IET datasets and the XTIE collaboration during the Y2.5 and Y3 evaluations. They are not useful for general applications.

`retrieve-event-modes` (**&rest** *(options* OBJECT*)*) *:* [Command]

For each event supplied to the :events option (defaults to all known events) retrieve all the exploitation modes used in that event and assign the :MODES-USED attribute accordingly. Performs retrieval in :module which defaults to `DATASET`.

`xtie-assign-member-weights-from-annotations` [Command]
(**&rest** *(options* OBJECT*)*) *:*

KLUDGE: after loading CMU groups and objectifying them we look for `weight` assertion and use them to assign membership weights for each group object. This allows us to report members in reports in the correct order.

`xtie-load-case-teams` ((*file* STRING) **&rest** *(options* OBJECT*)*) *:* [Command]
(CONS OF GENERIC-GROUP)

Load case hypotheses from *file*, generate threat team hypotheses for them and return the resulting list of unknown groups/teams. Loads evidence into the module specified by the :module option (defaults to `DATASET`). Event teams won't be loaded into the global objects table to avoid interference with real groups.

`xtie-load-merge-and-report-case-teams` ((*sourceFile* STRING) **&rest** (*options* OBJECT)) : [Command]

Load case teams from *sourceFile*, merge them with currently loaded KOJAK groups and write the merged version of *sourceFile* to :report-file (defaults to <sourceFile>.merge).

`xtie-merge-case-teams-and-groups` ((*teams* (CONS OF GENERIC-GROUP)) (*groups* (CONS OF GENERIC-GROUP))) : [Command]

Foreach case team in *teams* find the best matching group in *groups* and make that the performer of the corresponding case event. Also migrate any modes used in the case to the responsible group.

`xtie-report-merged-case-teams` ((*sourceFile* STRING) **&rest** (*options* OBJECT)) : [Command]

Read expressions from *sourceFile* and copy them onto :report-file which defaults to <sourceFile>.merge. If we are copying a VulnerabilityExploitationCase and now its responsible group, add an appropriate `performedBy` assertion. The reason we do this almost literal copying is that the alert reports have various intricate structure and order that we don't want to mess up. For this to work it is important that `xtie-merge-case-teams-and-groups` was run first.

`xtie-report-secondary-data-accesses` ((*file* STRING) **&rest** (*options* OBJECT)) : [Command]

Write Me

`xtie-report-threat-individuals` ((*groups* (CONS OF GENERIC-GROUP)) (*file* STRING) **&rest** (*options* OBJECT)) : [Command]

Write Me

9 Group Finder Ontology

This section describes the generic concepts and relations defined in the KOJAK Group Finder ontology (see also '`kbs/generic-groups-ontology.plm`'. The names in this ontology are all case-sensitive.

`Agent` ((*?self* AGENT)) [Concept]
Agents can be members of groups and participate in events.

`Group` ((*?self* GROUP)) [Concept]
Groups are collections of agents that have some unifying goal or purpose.

`KnownGroup` ((*?self* GROUP)) [Concept]
Known groups are groups with a definite identity and/or name. Two distinct known groups will always be viewed as separate entities, even though some of their members might overlap.

`UnknownGroup` ((*?self* GROUP)) [Concept]
Unknown groups are groups without a definite identity or name. Unknown groups are usually hypothesized and two distinct unknown groups might be merged into a single group or with a known group if a certain similarity threshold is exceeded.

`groupMember` ((*?group* GROUP) (*?agent* AGENT)) [Relation]
?agent is a member of ?group.

`Event` ((*?self* EVENT)) [Concept]
Actions and events agents participate in and groups are responsible for.

`participant` ((*?event* EVENT) (*?agent* AGENT)) [Relation]
?agent participated in the event (or action) ?event.

`responsible` ((*?event* EVENT) (*?group* GROUP)) [Relation]
?group is responsible for the event (or action) ?event.

`nameString` ((*?object* THING)) :-> (*?name* THING) [Function]
?name is the (full) name of ?object. Used by the report generator to generate names instead of entity IDs. Only needed for cases where IDs and names are not conflated such as in the Ali Baba EDB. Requires the definition of mapping rules that retrieve names from appropriate EDB tables.

`linkCount` ((*?ltype* THING) (*?seeds* THING) (*?p1* THING) (*?p2* THING) (*?count* INTEGER)) [Relation]
Relation used by `extend-groups` to compute link count statistics for links of type ?ltype emanating from ?seeds. For any particular dataset schema, rules have to be connected to this relation to allow computation of statistics for each relevant link type. For each query ?seeds will be bound to a SETOF term containing all seeds and ?ltype will be bound to a particular relation of interest. The relation `GROUPS/anyLink` will be supplied if all link types should be considered.

`anyLink` ((*?event* THING) (*?p1* THING) (*?p2* THING)) [Relation]
Dummy link type used to indicate that any link should be considered.

`linkCountable` ((*?r* RELATION)) [Relation]

Marks a link relation as link countable and generates the appropriate rule for it that can generate these counts. A link relation ?r marked this way has to have the same signature as `GROUPS/anyLink`.

10 Questions and Comments

For questions or comments please contact Hans Chalupsky (hans@isi.edu), Jafar Adibi (adibi@isi.edu) or Tom Russ (tar@isi.edu).