CHALMERS

Locating faulty data in an harvested database

Extending a Metadata language with support for semantic rules to find erroneous data in a vast and incomplete database

Master of Science Thesis TDATA

Per Gundberg
Joel Steen Timle

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, February 2012

Locating faulty data in an harvested database
Extending a Metadata language with support for semantic rules to find erroneous data in a vast and incomplete database

Per Gundberg
Joel Steen Timle

Examiner: David Sands

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden February 2012

Abstract

This thesis deals with the task of finding erroneous entries in a large database whose content have been automatically collected by scanning different sources on the world wide web. The information is divided into different events, organized in different event classes.

As part of the thesis work, a language to describe semantic and structural rules on the information has been designed as an extension to the already existing Metadata language of the database. A set of rules has been written in this language which describes the extended demands. A tool to test the information in the database against rules described in the extended language has also been implemented. The result of the evaluation not only reports if an entry does not fulfill a rule, but also what part of the entry breaks the rule. This information is stored in a database for further analysis and use. Subsets of the database have been checked and during these tests, about five percent of the events did not fulfil all of the rules defined for its event class.

Table of Contents

List of abbreviations

API	- Application Programming Interface
AST	- Abstract Syntax Tree
RF	- Recorded Future

1 Introduction

1.1 Background

Recorded Future [1] is a company that catalogs data with a focus on their temporal and relational context. The data is collected from different sources on the the Internet. Some of the sources provide well formed data, e.g. external databases. However, a portion of the data is collected from natural languages sources.

The plain text data is analysed with linguistic methods [2] to find what information they contain. The first step is to decide what entities are mentioned. This can be companies, organizations, towns etc. The next step is to describe what event or fact the text describes. This can be anything from the travel plans of a person, a legal issue between two competing companies, or the result of an election.

The data is then scanned for temporal information about when the event took place, or when it is going to take place if the text describes a future event. This can be both in absolute form, e.g. "in 1995" or "on the sixth of may", and relative to the date the text was written, e.g. "last week", "in the past month" or even "in the near future". The actual time that these temporal expressions describes are also added to the information about the event. The events and entities are finally linked together to create a structured unit from the text.

The data is also analysed further to extract more information such as momentum and sentiment. Such meta information is not only derived from the text itself, but also existing knowledge such as the source's trustworthiness and what is known about the entities involved.

This creates a large information platform that is both accessible through a graphical user interface and processed further with statistical analysis. This can not only give information about what has happened, but can also show sequences of events and even give hints about what is likely to happen.

Due to the large amount of data processed, and the nature of current natural language analysis, it is inevitable that some faulty data finds its way into the database. Even if the analysis works as intended, there is still the possibility that the source itself had errors in it.

To address this, Recorded Future developed a descriptive language called *Metadata* that describes the data-structures against which the data is then tested. This language models the structures in an object oriented manner with inheritance and attributes.

Besides describing the basic layout of the information, it also supports a few structural constraint expressions. This addressed strictly structural demands, but does little to detect non-structural faults.

1.2 Purpose

The purpose of this thesis is to increase the quality, correctness and consistency of the data stored in RF's databases. To achieve this, a language was designed in which rules can be formulated. The language needed to be expressive enough so that the rules written in it can be used to describe infractions corresponding to an adequate subset of the errors found in the database. A front end and back end was implemented to interpret the rules, verify them, and apply them to the database.

This tool will be called *Vulpecula* for the purpose of this report.

This thesis will describe the background work done and the resulting language and interpreter.

In chapter two, the ecosystem in which Vulpecula operates is described, including a more in-depth description of the preexisting tool Metadata, on which Vulpecula is based.

In chapter three, the underlying analysis is discussed, together with the pilot study. This is where the data stored in the database is analysed for the types of errors that should be handled in the end.

In chapter four, a literature study that will show some related work, and how it has inspired the design of Vulpecula.

In chapter five, the design is described in seven parts: the language design, the internal representation of the described rules, the front end, the application design, event evaluation, the layout of the log database, and finally a description of the rule-design ideas.

In chapter six, the result of this thesis work applied to Recorded Futures database is presented.

In chapter seven, the conclusions of the thesis work is discussed together with the authors' thoughts about the outcome.

Finally, in chapter eight, some possible improvements and extensions to this work are discussed. It also mentions what parts of the work that have yet not been fully evaluated.

1.3 Functional Requirements

1. A compilation of extended demands on consistency, correctness, and quality of the data stored in the RF's database.
2. Extensions to the preexisting formal language used by Metadata, allowing for further demands on the content to be described.
3. Implementation of a tool that given a description in the extended language will perform a verification of the content of the database, and mark content which does not meet the demands.

2 Background: the operational context

Int this chapter, the context in which Vulpecula operates is described. These are the tools that are available for interacting with the data stored in RF's databases, and the tools that will be supplemented. Here follows a short description of each and how they relate to this project.

2.1 API

The data that will be sought through for errors are stored in RF's databases. An API was made available to interface with one of the databases. The interface is net-based, and communication to and from the server-side service is done using JSON-encoded data. All communication with the database was done through this interface.

2.2 Metadata

The structure of the entries in the database are described using a descriptive language called the *Metadata language*. The design is that of an object oriented language, where each category of data present in the database is represented by a class. Each class can inherit attributes from another class, i.e. the language supported single inheritance. The attributes of the classes are all assigned a type and can also be assigned one or more constraints. There are also constraints that act on the classes themselves. All constraints are local to the instances of the classes, and can not say anything about the relation between two instances.

This language addresses some of the same problems this thesis work is concerned with. It also provides a model for the structure of the data. This made it a natural starting point.

2.2.1 Syntax notation

Throughout this document, a notation loosely based on EBNF [3] will be used to describe the syntax of the two languages relevant to this thesis. Terms can be separated by any number of white space characters. The meaning of the terms are described in Table 1.

Table 1: Grammar for the EBNF notation used in this document.

`a`	an identifier
`"a"`	a token
`a b`	Expression a followed by expression b, possibly separated by whitespaces
`a|b`	either expression a or expression b
`{a}`	zero or one occurrences of expression a
`[a]`	one or more occurrences of expression a
`()`	grouping

2.2.2 Syntax

A Metadata program is a set of classes (1). Each class has zero or more attributes, zero or more constraints, and can extend one other class (2). If a class extends another class, it inherits that class's attributes as well.

```
metadata ::= [class]                                        (1)
class ::= "class" class_name {"extends" class_name} "{"
   [(attribute|constraint) ";"]                             (2)
"}"
```

An attribute is a piece of information associated with the class, and they each have a type, a name and zero or more constraints (3).

```
attribute ::= type name [constraint]                        (3)
```

A type can be either a simple type (4), which is either a primitive type or a reference (5), or it can be an algebraic type (4). The primitive types are strings, long integers, doubles and dates (6). A reference points to an instance of some class. The algebraic types are union types and sets (7). Unions take a comma separated list of types, which must be simple types, and matches any of the listed types. A set takes one type, which must also be of simple type, and which describes the type of the elements in the set.

```
type ::= simple_type | algebraic                              (4)
simple_type ::= primitive | class_name                        (5)
primitive ::= string | long | double | date                   (6)
algebraic ::= "union" "<" simple_type ["," simple_type] ">" (7)
            | "set" "<" simple_type ">"
```

A constraint is a limitation to an attribute or to a class. Syntactically, it is an identifier preceded by an at-sign and it can have any number of arguments (8). If it has no arguments, then the parenthesis are not written out. The arguments can be either identifiers used to describe
attributes, strings, or integer literals.

```
constraint ::= "@" name {"(" constant ["," constant] ")"}   (8)
```

2.2.3 Constraints

There are seven constraints built into Metadata. They are included to allow for finer control of the attributes, and even allows for some constraints on the values, as opposed to the structure. All but two are attached to an attribute and only put constraints on that attribute, and all are special cases and built into the language.

`Scope`
This unary constraint lets the user declare what scope the associated attribute belongs to. The possible values are *id*, *instance*, and *core*. The meaning of this constraint is not relevant for this thesis, and was not carried over to Vulpecula.

`required`
Indicates that the associated attribute must be filled in.

`at_least_one`
A polyadic constraint that takes attribute-names from the same class as arguments. This constraint operates on the whole class rather then on one specific attribute, something it only shares with the next constraint. If it is declared, then at least one of the attributes must be set.

`exactly_one`
A polyadic constraint that act like the previous one, with the exception that one, and exactly one is the only number of attributes allowed.

`minsize`
A unary constraint applied to sets. It takes a integer literal as its only argument. If applied, the set must contain as many as, or more, elements than the specified integer.

`size`
A unary constraint that act like the previous one, with the exception that that it matches exactly as many elements as the specified integer.

`enum`
A polyadic constraint applied to strings, that takes a number of strings. These values are the only allowed values for that string.

3 Analysis

In this chapter, the analytical work, and what descriptions it lead to, is described. It describes the analysis of the data stored in the server, and which represents the state of the data at that time.

In order to give an understanding of what errors existed in the database, the first part of the project was dedicated to the analysis of the existing data in a quite manual fashion. Due to the large size of the database and the limitation of time, only a small subset of the existing information could be examined.

Since the information is catalogued in different classes, each representing a different kind of event with different attributes and relation between them, the subset to examine was chosen in such a manner that every class of event was represented, and with about as many instances each. They were also sampled from different time periods. This was due to the fact that if the current Metadata description had been updated since the data was harvested it is possible that it breaks the current, updated, scheme.
The emphasis was on a data set collected close to the start of the study, 2010-05-26 and backtracked day by day until it found a total of 100 events for each event class. This was done by a minimal *python script* that interacted with the database using the database API.

One possible disadvantage with this approach is that the event instances collected this way could be very closely connected due to the ordering of the event from the API. This sometimes lead to several events from the same article, and with very closely related interpretations, ending up in the data set. This was not considered severe enough to sample in a different way though.
Another problem is that the events are far from evenly distributed between the event classes. This lead to a higher percentage of event from the classes with fewer instances being collected. This was not considered a problem since the purpose of the pilot study was to find as many different types of problem as possible.

This selected set of events was collected to give the possibility to easily return to inspected instances when new insights and ideas were gained.

The database was accessed through the web service API accepting Queries in JSON format. The result is returned in JSON which could easily be interpreted and also dumped to a local file. The collecting of the events was automated by creating a small python script that iterated through the event types and looped until enough events were collected.

The purpose of the examination was to find both improbable values for specific data fields, such as field describing a year having the value "Monday", as well as to find where connected values creates doubtful information. An example of the latter is when an electronics company is connected to a event that should only accept pharmaceutical companies.

The manual part of the process was to step through event by event and using our common sense to identify errors in the data set. To simplify the examination of the data some small scripts in *awk* and python were created to automate the process. These helped with stepping through the events and also displayed the events in a more humanly readable fashion. A screen from the python script is shown in Figure 1.

```
{fragment: Orsu Announces Completion of Preliminary Assessment Study for
Karchiga Project, Base Case Study NPV of US$138M and IRR of 40.5% Over 10
Year Life of Mine.
time_fragment: 2010
stop: 2010-12-31T23:59:59.000Z
start: 2010-01-01T00:00:00.000Z
attributes: {status: NA
       product: {type: Product
              name:
              momentum: 0.0126290612419}
       product_type: drug
       positive: 0.2727273
       company: {industries: [{type: Industry
                     name: Copper Ores
                     momentum: 0.0
                     sic_code: 1021}
                     {type: Industry
                     name: Gold Ores
                     momentum: 0.0
                     sic_code: 1041}]
              redirects: http://d.opencalais.com/comphash-1/ab0f4b35-d66c-
3fc3-9ef2-23826d8c322d
              type: Company
              name: Orsu Metals Corporation
              momentum: 0.000386664522332}
       negative: 0.0
       inherited_locations: [{type: Country
              name: United Kingdom
              momentum: 0.235266010589}
              {redirects: http://d.opencalais.com/genericHasher-1/6fda72fd-
105c-39ba-bb79-da95785a249f
              type: City
              name: London
              momentum: 0.662902719033}
              {type: Country
              name: Kazakhstan
              momentum: 0.0128797296636}]}
document: {url: http://www.marketwire.com/mw/release.do?
id=1266319&sourceType=3
       downloaded: 2010-05-26T06:01:31.000Z
       published: 2010-05-26T06:00:00.000Z
       source: {media_type: News_agency
              name: marketwire
              description: MarketWire}
       title: Orsu Announces Completion of Preliminary Assessment Study for
Karchiga Project, Base Case Study NPV of US$138M and IRR of 40.5% Over 10
Year Life of Mine}
type: FDAPhase
id: 210125168
momentum: 0.0}
```

Figure 1: Interface for the pretty printer when manually checking the events. In this case a suspicious event is shown. A mining company is not likely to releases a new drug.

All new suspicious findings were recorded and a note was added, describing the problem, what attributes caused it, as well as possible ways of catching it.

A more deductive approach to find what possible problems there might be was also used. The existing Metadata description of the database was evaluated in order to find things that should be limited even if no existing errors were found in the inspected subset of the database.

As a third way to find out what possible problems could exists in the database, customer feedback from the Recorded Future service was used.

When summarised, the majority of problems were found during the examination of the data subset and a few was added during the deductive analysis. The customer feedback only confirmed that some of the problems that could be spotted with the added set of rules actually have been reported by the customers.

During the manual examination of the data set some entities was found that were highly over-represented in erroneous events in relation to the total number of events that referred to them. After a closer examination of these entities it was found that these entities had erroneous information about them. For example one company instance had the empty string as its name which probably made the semantic analyser to match it up in a lot of texts. This was a motion picture company and is therefore very unlikely to release a new drug. These instances were later on called “black holes”.

These “black holes” are worth some more attention since they point out an important problem with a too naive approach when the evaluating and logging the rules in the context of an event. If an entity is over-represented in rule infractions where its values are used as during the evaluation of the rule then perhaps the entity is a problematic one and need some extra attention or even blacklisting.

As a compliment to checking one event at a time, a comparison of the data values from all the attributes with the same name from different items was conducted. This was done by simply iterating through all the events and their associated items, that were fetched for the analysis and saving all values in a record indexed by attribute name. During this step it was found that the values of the same attributes could sometime have very diverse formats. E.g. the age attribute of a person could be “12”, ”30’s” or “young”. At this point, it was decided not to try to interpret the values in a more advanced way than the parse method of the corresponding class in Java, since that was what would be used.

At the end of the study the errors was examined and it was decided what expressiveness would be needed to detect them. For example, potential errors such as a meetings lasting for a years, or trips being planned as far as nine years in advance, both require operations on dates to be detected. This could have been achieved with a number of built-in commands, trying to anticipate all needs that would arise. This was rejected and instead a time period was introduced and arithmetic operations on it defined. This way, concepts such as “tomorrow” could easily be realised as “today” + “one day”. So, to detect these errors, first order periods and arithmetic was needed. But to be able to use it in a rule, you would also need to compare it to something. And for that, equality and inequality is needed.

The focus during this pilot study were to find out what expressive power the tool needed to be able to catch erroneous entries in the database. This was the reason to do a more quantitative in order to find as many different error types as possible. During later stages of this thesis work it was found that a more qualitative logging of, including references to faulty entries, would have been beneficial in order to find how many faulty entries that the application missed.

In order to be able to catch these errors the following extended expressiveness to the language was decided:

- Logical operators: not, and, or and implies
  These were added to allow for more complex expressions, and implies also takes on the role of a conditional statement.

- Simple arithmetic
  E.g. addition, subtraction, etc. Added to make it possible to tweak the data before using it in a comparison. An example would be to convert between units, such as converting knots into m/s. Another would be to indicate that an event must happen before today+7 days, or in plain text, within a week. From the last example it's clear that it must work on more than simple integers and floats.

- Equality and inequality
  E.g. equals, larger than, etc. Even these must work for more than simple primitive data-types such as integers.

- Basic set operations: quantifiers, and element
  E.g. the universal quantifier, the existential quantifier, and element-of operator. The existence of sets in the language necessitates the inclusion of operators operating on them. Since the sets are not true sets in the mathematical sense, an example use could be to make sure that all elements of a set are unique. This could be done by making sure that for all elements in the list, there exist exactly one copy in the list.

- Navigational expression: possibility to use attributes attribute
  The attributes that are instances of class can themselves have attributes. These should be accessible. A classic example would be a person having

a parent, and that parent having parents, so to find the grandparent, you have to access parent of parent of person.

4 Literature study

The primary problems when constructing the tool for this project was the combination of vastness, inaccuracy and incompleteness of the data. These are problems that other project have faced before when working against large databases, built up by automatic processes.
Since the this project focus on finding semantic errors and inconsistencies in the database, a search for existing work that strives towards the same goals have been conducted. None of the examined projects fits the demands of this project in such a way that it could be directly translated into a solution. The most common differences are that they either are too tightly bound to their intended target or are designed to be able to conduct further analysis. Often about what other rules could or should be enforced on the database information.

The works that have had some influence on this thesis or are similar in their problem are presented in this chapter.

4.1 CoLan

The CoLan constraint language [4] was designed to express constraints on object orientated data in a structured, declarative form. The important issue that CoLan addresses is to formalise semantic constraints on the information in the database.

CoLan introduces the constraints on the entity classes and are checked whenever any change to the database is performed. These constraint are expressed as Boolean valued expression constructed in a functional style, one of the main motivations for this is that the authors opinion that this results in a higher readability. It also supports quantifiers over sets, both the classic existence and universal but also specific numerical quantifiers such as 'at least', 'at most' and 'exactly'. Since the intended target is an object oriented database CoLan also allows what they call "navigational" constraints, this gives the ability to access another object that is referred in the current object.
CoLan also supports some basic arithmetic operations in order to express more complex constraints. It also allows constraints that spans over different entity classes, and thus limiting their combined information.
This was one of the most influential papers that was part of the literature study. The basic tool set was much inspired from CoLan. Similarities can be found in the existence quantifier and navigation of objects especially.

4.2 OCL

Much like CoLan, Object Constraint Language or OCL [5] is a declarative language designed to express constraints on object-oriented data-structures. Created by IBM, it has now been adopted by the Object Management Group (OMG) and is used to apply constraints on all their meta-models.

4.3 Semantic Web

Semantic web is a term coined by Tim Berners-Lee at World Wide Web Consortium (W3C) . The idea is that the information on the Internet should be organized in a way that machines could automatically understand. The term "Semantic Web" is used ambiguously, but is often used to describe the W3Cs models and technologies for formalising and interchanging data and information between different sources.

4.4 Weak negation

The concept of weak negation, as opposed to strong negation, is something that is widely used in databases. Weak negation, also known as negation-as-failure, is closely related to the concept of closed-world-assumption (CWA) in that they both treat unprovable predicates as if they were false. Strong negation, on the other hand, will only prove the negation of provable predicates. In “How to Design a General Rule Markup Language” [6] and “Web Rules Need Two Kinds of Negation” [7], the author deals with the subject of strong and weak negation, or more precisely the need for supporting both in rule-languages; In “A Database Need Two Kinds of Negation” [8], he covers the same subject, but in the context of databases. He postulate that the addition of a strong or strict negation is needed to counteract the limitations inherent in CWA.

Even though his implementations of coexisting strong and weak negation don't fit nicely into this thesis work, the need for handling unprovable data is obvious. Rules that involve undefined attributes, while unprovable, should not detract from the reliability of that instance; not unless the attribute is required. Ultimately, CWA is too restrictive for this thesis, and therefore an approach more in the line of the open-world-assumption is needed. Adding strong negation should allow the expression of rules that will not demote incomplete, but otherwise correct, data. Meanwhile weak negation will handle those few instances where a more strict rule is placed.

5 Design

In this chapter we discuss the design of Vulpecula. First the tools used in the development and deployment are described, and then a detailed description of

Vulpecula follows. The detailed description is divided into seven parts. In the first part we describe the syntax and feature set of the Vulpecula language. In the second part we describe the abstract syntax tree. In the third, we describe the front end, which in this case is the parser and type checker. In the fourth part, we application design, in the fifth we describe the rules, in the sixth the logging, and finally rules

5.1 Tools

These are the tools used in the production of Vulpecula.

5.1.1 Java

Java is a very popular object oriented language developed by Sun Microsystems, which is now owned by Oracle Corporation. Syntactically, it belongs to the same family of languages as C and C++. However, while being object oriented and therefor seemingly close to C++, it does have a more limited feature set and a simpler object model, such as only allowing single inheritance. Java is typically compiled to bytecode and then run on the Java Virtual Machine, giving it a good cross platform support.

The decision to use Java came naturally as it was the language Metadata was implemented in and there was a wish to be able to reuse some of the existing work. It might also be worth noting that Metadata, and therefore also Vulpecula, share similarities with Java.

5.1.2 JavaCC

JavaCC [9], or Java Compiler Compiler, is a parser generator for Java. A formal syntax in EBNF notation is used to generate a top-down parser. The Metadata language, which the Vulpecula language is based on, is parsed using JavaCC. To facilitate code reuse, a decision was made to use JavaCC for the Vulpecula language too.

5.1.3 Joda Time

The standard libraries in Java have basic representation for a point in time and a time duration in form of Date and Time. These are limited when it comes to expressing duration in units more often used in natural languages, e.g. years, months, due to the fact that their length depends on the context they are applied to. Joda Time [10] is a open source library that is intended as a replacement for Javas Date and Time classes.

The basic concepts in Joda Time are:

- Instant - a specific point in time
- Interval - a interval from one instant to another instant
- Duration - a duration of time measured in milliseconds
- Period - a duration of time measured in any mixture of years, months, weeks, days, hours, minutes and seconds

Joda Time was used since Period, when compared to Java's Time, more closely resembles the natural language descriptions often found when dealing with harvested data of the kind RF deals with. It was also already in use at RF which made the choice even more natural.

5.1.4 JSON

JSON [11], JavaScript Object Notation, is a data-as-text interchange format that is designed to be lightweight and independent of programming language. JSON is used for the communication with the API.

5.1.5 Stringtree JSON

Stringtree JSON [12] is a lightweight JSON implementation for Java. The Java representation of the JSON data structures are shown in Table 2.
There are several implementation of JSON for Java and the choice fell on Stringtree JSON since it was already used at RF.

Table 2: JSON type to Stringtree JSON type lookup

JSON	**Stringtree**
array	Collection (ArrayList)
object	Map (HashMap)
number	Number (Long, BigInteger, Double, BigDecimal)
string	String object
“true” and “false”	Boolean
null	null

5.1.6 SQLite

The goal of Vulpecula is to detect and record entries in the database that are most likely erroneous. Since this information should be easily read and summarized the choice was to use a database to store all rule infractions. It was decided to use a local database since there would be a lot of tuning going on until a good method of logging the infractions was found. The choice fell to SQLite since it is lightweight and modular, the database consist of a single file that could be moved and copied.

5.2 Detailed design

When designing the Vulpecula language, a decision was made to base it on the pre-existing Metadata language. Syntactic backwards-compatibility with Metadata was maintained, which facilitates writing rules based on the preexisting Metadata file developed by RF. Note that this does not mean that Vulpecula will be able to run all Metadata programs. It was also decided that the front end would be based on the equivalent code for Metadata. This reuse was meant to give an established code base to work from. The downside was that the choice of tools, such as implementation language, was limited, and that the structure might not have been ideal for the extension.

5.2.1 Language design

The Vulpecula language is based on the Metadata language described in 2.2.1. The added features are meant to extend the language enough so that rules can be written to catch problems with the data, as well as the structure. The following examples give a taste of the language. In Figure 2, three example-classes written in the Metadata language are shown. This code defines an Entity class that describes entities. In this example, its only attribute is its name. The Person class is a sub-class to the Entity class. As such it inherits the name attribute from the Entity class, while also defining an attribute describing the birthdate of the person. The ParentsOf class describes a relationship between a child and two parents, represented by a Person and a set of Persons respectively. The set must have two elements (ignoring cloning for now), as dictated by the use of the size-constraint.

```
Class Entity {
  string name;
}

class Person extends Entity {
  date birthdate;
}

class ParentsOf {
  set<Person> parents @size(2);
  Person child;
}
```

Figure 2: An example of Metadata describing a relation between a child and its parents, all being People.

There are some structural constraints in place here, but there is still room for some improvement. One could, for example, make sure that all parents are old enough to have this child. Figure 3 shows the Vulpecula code implementing such a rule in the ParentsOf class. For this example, it's assumed that the parents must both be at least eighteen years older than the child.

```
class ParentsOf {
  set<Person> parents @size(2);
  Person child;

  rule forall parent in parents: parent.birthdate >
                                 child.birthdate+period(18,0,0)
    as "parent must be old enough";
}
```

Figure 3: Vulpecula code realising the rule that a child's parents must be old enough to have it.

The size-constraint could also be implemented using the extensions introduced in Vulpecula. In Figure 4, this is demonstrated together with a demonstration of the macro facility.

```
macro size(s, n) exist [n] e in s: true;

...

class ParentsOf {
  set<Person> parents;
  Person child;

  rule size(parents, 2) as "a child must have two (biological) parents";

  rule forall parent in parents: parent.birthdate >
child.birthdate+period(18,0,0)
    as "parent must be old enough";
}
```

Figure 4: Replacing the constraint with a macro-call, using the additions introduced in Vulpecula.

This will be further explained in the following sections.

5.2.1.1 Macros

For long or reused expressions, a macro facility was implemented. Macros are declared alongside classes at the top level (9) using the `macro` keyword. Each macro takes zero or more comma-separated parameters (11). Originally it was implemented as syntactic sugar and any macro call would be replaced by the body of the macro in the parser. However, to increase traceability when an error is found, it was later promoted to be included in the AST produced by the parser. An example of macro usage can be seen in Figure 4.

5.2.1.2 Rules

The most important addition introduced with Vulpecula is the *rule*. A rule is applied to instances of the class it is declared in (10).

```
metadata ::= [macro|class]                                          (9)

class ::= "class" class_name {"extends" class_name} "{"
  [(attribute|constraint|rule) ";"]                                 (10)
"}"

macro ::= "macro" name "(" {name ["," name]} ")" expression  (11)
```

A rule is a Boolean expression preceded by the keyword `rule` (12). Each rule specifies certain limitations for the instance of the class it is attached to. An infraction is indicated by a negative return value. Each rule is also given a name in it is declaration in the form of a string, with which the rule is identified in case of an infraction. A concrete example can be seen in Figure 3.

```
rule ::= "rule" expression "as" string                        (12)
```

A new primitive type, period (13), was added to the language. It is equivalent to the Joda Time concept of the same name. Figure 3, contains a rule that uses the period macro, which has this type.

```
attribute ::= type name [constraint]

type ::= simple_type | algebraic
simple_type ::= primitive | class_name
primitive ::= string | long | double | date | period         (13)
```

Two built-in macros, period and date, were added to be used as constructors for periods as well as the pre-existing dates. Both macros take three integer arguments, year, month, and day, and constructs a value of respective type.

5.2.1.3 Expressions

The expressions cover basic arithmetic and logic operators. All operators are typed and have a precedence. Here follows a walk-through of the expressions in increasing order of precedence.

Both the universal and existential quantifier take an identifier, a set, and a Boolean expression (14) & (16). The existential quantifier also takes a range (15), allowing the programmer to specify how many elements must conform to the rule. A range can be either an absolute number, or it can be a range between two numbers. There are default values for the upper and lower bounds if either one of them is left out. The lower bound defaults to zero, while the upper defaults to accepting as many positive replies as possible. Forall and exists can be seen used in Figure 2 and Figure 4 respectively.

```
expression ::= exists | forall | expreseeion1

exists ::= "exists" {range} name "in" expression ":" expression   (14)
range ::= "[" (exactly|to|from|fromto) "]"                          (15)
exactly ::= integer
from ::= integer ":"
to ::= ":" integer
fromto ::= integer ":" integer

forall ::= "forall" name "in" expression ":" expression           (16)
```

The Boolean operators implication (17), disjunction (18), and conjunction (19) were included. For reasons that will be discussed in the section on logging, a decision was made that they would all be treated strictly. Equality and inequality (20) can compare any two values as long as they are of the same type.

```
expression1 ::= expression2 "->" expression1                (17)
expression2 ::= expression3 "||" expression2                (18)
expression3 ::= expression4 "&&" expression3                (19)
expression4 ::= expression5 ("=="|"!=") expression4         (20)
```

A element operator (21) was added. An expression is compared with a set containing elements of the same type as the expression being compared. If the set contains an element comparable with the expression, then true is returned.

```
expression5 ::= expression6 "in" expression6                (21)
```

The ordering operators (22) can be used to decide the order of two values, as long as they are of the same type and can be ordered. Types that can be ordered are: long, double and date.

```
expression6 ::= expression7 ("<"|"<="|"=>"|">") expression6   (22)
```

To allow for more advanced rules, a few basic arithmetic operators were added (23) & (24). All pairs of values that are of the same type and belong to the numeric types can be used with these operators. The numeric types are: long and double. Additionally, add and subtract (23) can be used to add and subtract two dates or a date and a period. In the case of an addition/subtraction of two dates, the result is a period, whereas a date and a period would result in a new date.

```
expression7 ::= expression8 ("+"|"-") expression7           (23)
expression8 ::= expression9 ("*"|"/") expression8           (24)
```

Three unary operators (25) which are all related to negation are implemented (25). The first two are familiar and work as expected. The logical negation takes a Boolean expression and negates it. The arithmetic negation takes a numerical expression and changes the sign. The third one, however, will require some explanation, and this is the negation-as-failure operator (~).

```
    expression9 ::= ("!"|"-"|"~") expression9 | expression10        (25)
```

All expressions in Vulpecula can depend on an attribute that is not implemented. This is the case since not all attributes are required. The default behaviour when encountering an expression that depends on a non-existing attribute is to fall back to the nearest Boolean expression that surrounds it, and that does not rely on this attribute for it is value. The choice was made to implement it like this since a missing attribute, if not required, will neither confirm nor deny the truth of a rule, so it should be ignored and not let to affect the result negatively. The negation-as-failure operator changes this behaviour. The basic idea is that a missing attribute should result in a false value if the expression is prefixed with a negation-as-failure operator. Two different strategies were attempted to implement this check.

In the first strategy, the fall back was intercepted by the operator and changed so that the behaviour further up the line at the Boolean expression was altered. It was actually toggled, which meant that an expression embedded in two negation-as-failures would act just as the default behaviour. In this model, the operator was a typeless, strange beast that had no peer in any of the other operators.

The second strategy saw the negation-as-failure turned into a more normal operator that took a Boolean expression as it's only parameter and returned a Boolean value. If the operator was passed an expression that relied on a undefined attribute, it would simply return false. Otherwise it would act as a identity function.

There are pros and cons with both strategies, the first not being confined to Boolean expressions but being more complex, or the second one being easier to understand but lacking the power to toggle. In the end the second strategy won out because of the cleaner semantics and because it was easier to reason about.

A form of type-assurance (26) and (27) was added so that values that are known to be of a certain type, but who do not have this type in Metadata, and thus not in Vulpecula, can be used appropriately.

```
     expression10 ::= {assurance} expression11                      (26)

     assurance ::= "(" type ")"                                     (27)
```

In some cases, values clearly given one type can be assured to be another type. In this case it is rather a cast than an assurance. For these casts to pass the type checker, the expression tested must either be of a union-type that includes the type being casted to, or be castable to it.

```
class Object {
  string weight;
}

class Package {
  long max_weight;
  Object content;

  rule max_weight <= (long)content.weight as "Must be light enough";
}
```

Figure 5: Vulpecula-code utilising type-assurance.

The rules for castability are as follows:

- strings are castable to longs and doubles
- longs and doubles are mutually castable, truncating where necessary.
- a set is castable to another set if the element-types are castable
- a union is castable to a type if any of its types are castable to that type

The type assurance/cast is therefore involved in the parsing of data from the database at a later stage of the execution. An example can be seen in Figure 5.

The macro calls (28) & (35) and attributes (29) can be used freely in any expression, as long as their type id correct. If the attribute (29) is a reference to another instance, then that instance's attributes are also available through the navigation operator (36), including any possible references and further attributes belonging to them, and so on.

The following literals are also available: Integers of the type long (30), floating point numbers of the type double (31), strings (32), Booleans (33), and sets (34) and (37). The set must be homogeneous with respect to the types of its elements.

```
expression11 ::= macro_call                                    (28)
               | variable                                      (29)
               | "(" expression ")"
               | integer                                       (30)
               | float                                         (31)
               | string                                        (32)
               | "true"  | "false"                             (33)
               | set                                           (34)

macro_call ::= name "(" {expression ["," expression]} ")"      (35)

variable ::= name ["." {assurance} name]                       (36)

set ::= "{" {expression ["," expression]} "}"                  (37)
```

Besides the two built in macros mentioned earlier, one additional macro is predefined. This is the isDefined macro, which simply returns false if it is given an attribute that is undefined as its parameter, and otherwise returns true. Unlike the other two, this could be implemented in the Vulpecula language (38).

```
macro isDefined(val) ~(val==val)                                    (38)
```

5.2.2 Abstract Syntax Tree

Acting as the interface between the front end and the back end, the Abstract Syntax tree serves a central role in the project. When designing the AST, one of the primary considerations was that it should be usable both by the front end, and by the back end. What was needed was a way to separate the structure (i.e. the AST) from the operations that would be applied on it (i.e. the type checker and the back end), so that the tree could be passed between them easily. A decision was made to use the Visitor pattern [13] that would give all those advantages.

The tree is first created by the parser and then passed to the type checker.

5.2.3 Front end

The front end is where the rules file is interpreted and verified and typed so that it can be passed as a complete annotated AST to the back end.

5.2.3.1 Parser

The parser reads a rules file and generates an AST from that file. This is done using javaCC. The parser is based on the parser developed for the Metadata language. The addition of our syntactical elements was mostly straightforward.

5.2.3.2 Type checker

After the parser has been applied and generated an AST, that AST is passed through the type checker.

First the macros are controlled. Missing macros, over lapping instances, macros that are cyclical and macros with the wrong number of arguments are all rejected. This is also where some constraints are translated into macros. The constraints in question are `exactly_one` and `at_least_one, required, enum, size` and `minsize`. The reason these are translated is that they can be expressed using the Vulpecula language. The remaining constraint, `scope`, is beyond the capabilities of Vulpecula and is discarded.

Secondly, all classes are analysed to verify that there are no two classes with the same name. Then each class is tested in turn for the following.

- The inheritance must not be cyclical.
- All parent classes must exist.
- All attributes must be unique not only in the class their defined, but also over all parent classes

The final step is to check the rules in each class in accordance with the rules described in the language design.

During the type checking, every expression is assigned a type.

5.2.3.3 The evaluation tree

The evaluation of the rules are performed by creating an evaluation tree from the type annotated tree. This is done iterating through the type annotated tree with a visitor that build a new evaluation tree. The reason to build a new tree instead of reusing the AST is that the evaluation tree can be simplified since it no longer needs to contain information used to describe where the user have entered erroneous vulpecula code in the source file. At each nod this tree will have the an evaluation function that given an variable assignment context returns the result of the evaluation.

5.2.4 Application design

The last part of the project was to test the existing information in the database against the rules written in the Vulpecula language. To manage this task, a standalone application was written that given an input of the extended Metadata language started traversing the database and logged any instances that did not fulfill them.
The basic program flow is shown in Figure 6.

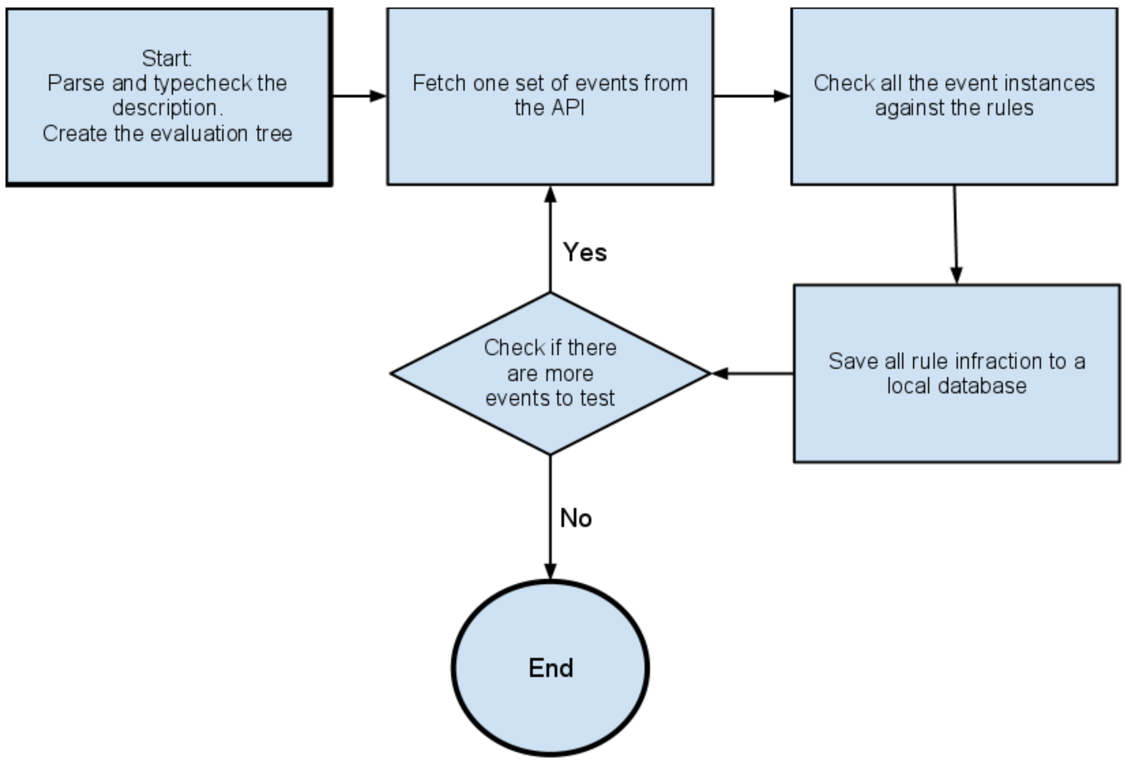
Start:
Parse and typecheck the description.
Create the evaluation tree
Fetch one set of events from the API
Check all the event instances against the rules
Save all rule infraction to a local database
Check if there are more events to test
Yes
No
End

Figure 6: The basic program flow of Vulpecula

5.2.4.1 Iterating through chunks

The program is supposed to iterate through the entire database and only test every event instance once. The simple and way to do this would be to iterate over a continuous and ordered identification numbers, ids, for all the events. At the time for this work it was not possible to request a range ids of from the API so another method of iteration had to be used. A simple partition process was decided as the most appropriate. The delimiters between the partition was decided to be the time the event was harvested and event type.

The event contains several different timestamps, such as time of harvest, time the document was published and the time of the event. The event time can even be multiple, e.g. start time and end time. The harvest time was decided to use as a delimiter since this is the only that can not be set to a previous date. If the the event time stamp would be used as a delimiter there are no guarantee that Recorded Future will not harvest an event in the future and tag it up with a date that the application would have considered as processed. This will not happen when using the time stamp at which the event was harvested.

Another natural delimiter method that was applied was by event type. This was first implemented to be able to test specific types as the different rules was implemented. This delimiter was left since it reduced the chunk size and would also allow special measures to be applied on different event types. One of those is limiting the referential depth of the response from the API.

5.2.4.2 Limit the referential depth

The response from a query does not only contain the local information about the event but also information about items referenced by the event. This inclusion of referenced events was done recursively and could therefore stack up a lot of information.

The API allowed for limitation of of the referential depth of the answer to the query. This is to prevent excessive amount of data to be fetch. The referential depth was given a default value of three by the API. Since the maximum referential depth of the rules easily could be added as a property during the type checking of the rules, and each API request only contains one type of event, the depth parameter could be set to prevent excessive data fetching. This was thought of as a way to speed up the API calls.

A small test run was made to find out if this could be used to improve the fetch times from the API but the results were indistinct, party due to large variation. These variations appeared to be random and even identical requests could vary form a couple of seconds up to a couple of minutes. In Figure 7 the response times for 20 queries with different referential depth is plotted. When first conducting these test requests it was noted that the first time a query was run it took considerably longer time than the following requests, independently of the depth of the first requests. This was thought to be due to caching effects on the database server and in order to eliminate these initial behaviors each set of queries had a dummy query with depth 0 as the first request.

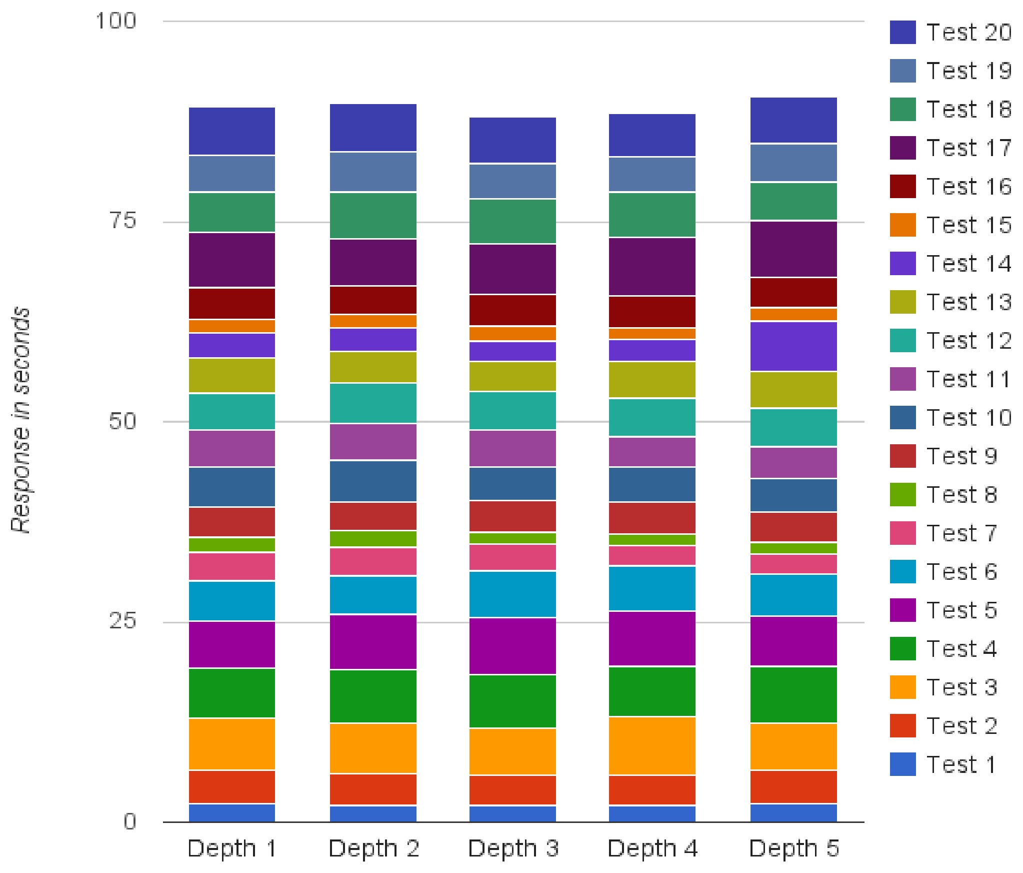

Figure 7: Response time depending on referential depth

Due to the indistinct test runs this approach of optimization was discussed with the experts at RF. It was then discarded since the effect would probably not be notable due to other factors of the response time of the API.

5.2.4.3 Information from the API

When the traversing order of the database was decided the queries used by the application to loop over the database could be designed. These queries are formed from a standard template where just the event type and the harvest date are filled in to create the complete query for the API. The response from the API is divided in two parts. One list of the events that matched the query, stored as an array, and a dictionary of all the referenced items by the events. This dictionary is a map from `long`, the items internal ID in the RF database, to an `json` object. Since the Vulpecula language does not allow arbitrary binding of variables, these dictionaries can be used to look up variable throughout the evaluation. The only thing that needed special binding are the bound variables in the quantifier expressions. To get the context of a single event the associated element from the list is singled out and the dictionary of referenced items is attached as the context for the referenced items.

The attribute look up is called with the entire chain of attribute names used to navigate from the current event, possibly navigating through referenced items, to the final attribute. Each step is annotated with its type. During the look up the type in the API response is tested against the assured or expected type in every step. This extra caution is implemented since the initial type check only applies to the events members.

The type verifications are performed in different ways depends on the expected type. When a primitive type is expected, the object from Stringtree JSON is checked against the corresponding types in Java, according to `Table 2: JSON type to Stringtree JSON type lookup on page 17`. This is done by testing the data with the Java keyword `instanceof`. If the data object is of a type that is convertible to the internally representation in Vulpecula it is converted and returned. If none of these match and the object is a string the expected type's parse method is used as a last resort to interpret the value. If all this fails, or the value can't be found in the first place the look up is considered a failure and an exception is thrown. If the expected type is declared in the metalanguage the item is looked up in the dictionary of referenced items, the entries "type" attribute is looked up and matched against the expected type or if it inherits from the expected type. If any of these steps fails the look up is considered as a failure and nothing is returned. The document and source objects are handled as special cases due to their representation in in the response from the API.

5.2.4.4 Type assurance

Since the type checks against the data from the API not only tests the type but also try to convert the same logic could be used for type assurance. This leads to that if the type check and conversion fails the data and an exception will be thrown. This is considered to be feasible implementation for assurances since it covers all the cases that are accepted by the type checker.

5.2.5 Evaluation of an event

The purpose of this application is to check if the events in the database fulfills the basic typing and structural requirements already defined in the Metadata language as well as the rules defined in Vulpecula. These two tasks are run consecutively. This is for to two reasons:

- The rules introduced with Vulpecula often relies on the structural requirements of the event in order to be evaluated. E.g. the rule to check that two companies are competitors by comparing their industries can not be evaluated if one of the companies is in fact a person. This could generate erroneous logs of the refereed instance. This is described more thoroughly in chapter 5.2.6.
- The difference in severeness between breaking one of the structural requirements and one of the rules in Vulpecula.

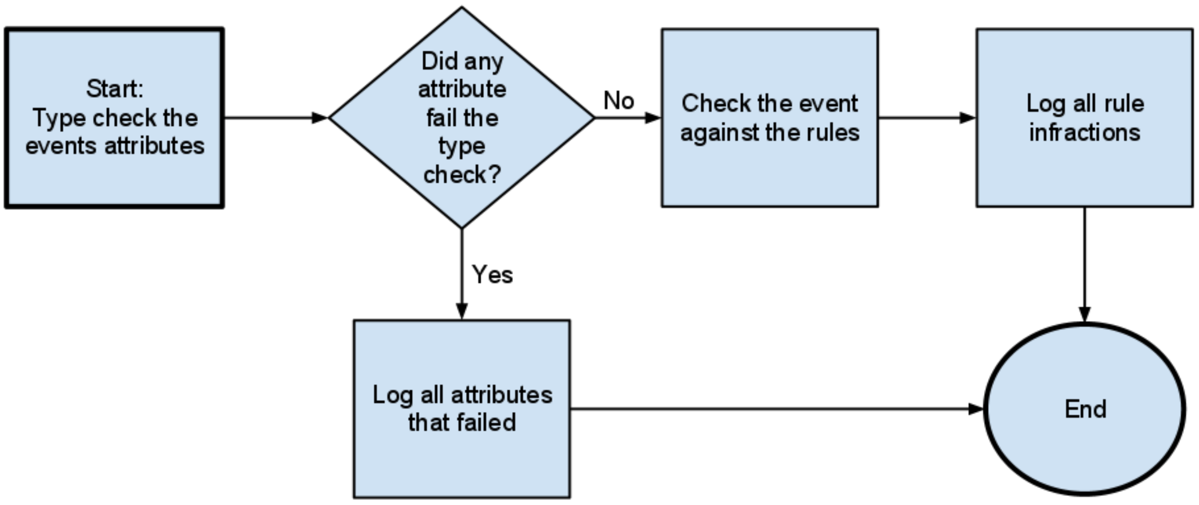
Start:
Type check the events attributes
Did any attribute fail the type check?
No
Check the event against the rules
Log all rule infractions
Yes
Log all attributes that failed
End

Figure 8: Flowchart of event evaluation

In Figure 8, the evaluation of an single event is shown. Note that the rules are not evaluated if any of the event's attributes fails the type check. The rules are evaluated by evaluating the tree corresponding to the rule with the current event as its context. The rule evaluation is simply a recursive call through the evaluation tree and the Boolean returned tells if the rule is fulfilled or not. If the evaluation is True, the rule is fulfilled and the evaluation moves on to the next rule. If the rule expression is evaluated to false the rule has not been fulfilled and an entry is added to the database containing which event, what rule and what attribute and items that contributes to the outcome of the rule evaluation.

Due to the incompleteness of the information the evaluation can try to access an attribute that is undefined in the context. This can either be because the incompleteness of the data or a chain reference that is longer than the referential depth of the API request. When this happens the current evaluation is halted and an exception is thrown. This exception contains information about which attribute could not be looked up. If the exception is caught at the top level of the rule evaluation then the standard procedure is to consider the rule as not breached. The exception can be caught by several operators in the evaluation tree. The most notably is the negation-as-failure operator, '~', that simply evaluates to false if any attribute look up in the inner expression fails. The exception is also caught by the the binary and n-ary boolean operators since it might still be possible to evaluate the complete tree.

If a type check fails this is also considered as an undefined attribute.

5.2.5.1 Rule evaluation and what to blame

Even if a rule is simply a function from the context of an event to a Boolean value, the return type of the rule evaluation is more complex in order to make it more useful; it is not only interesting to know if the event fulfills the rule or not, but also what events and entities lead to the result if it fails. This leads to a more complex evaluation model to be used. Beside the actual Boolean value the evaluation also returns references to all attributes that affects the value of the evaluation. In order to keep track of which attributes are responsible to each infraction every operator in the evaluation tree was given extra logic to keep track of what attributes lead to its return value.

The two base cases are a constant that simply contains no attribute references and a variable which only contains the reference to it self.

At the first approach the references passed by the unary operators and merged by binary and n-ary operators. After some tests it showed that this generated vast quantities of false positive references; there where many evaluation that returned references to attributes that was not responsible for the false value of the rule evaluation. E.g. if a for all quantifier failed at one of their elements, all of them would be logged. This lead to the design of a more complex logic for reference tracking.

Instead of just keeping track on which attributes that were used in the evaluation, two sets of attributes were returned: one which contains all the attributes which contributes for the evaluation to be false, and one containing all attributes that contributes for the evaluation to be true. The logic introduced for each operator determines how the sets from the child nodes should be combined to the create the sets for the current node.

In order to determine how dependent the value on an expression was of an attribute, a concept of "strong reference" was introduced. The reference between the attribute and the evaluated value was considered strong if the the attributes value must be changed to change the value of the expression. A weak reference is a variable that can be changed to change the value of the expression. Examples of strong reference are a single variable and and-expression. If any of these expressions evaluates to false all their child that evaluates to false must change for expression to evaluate to true. In the case of a binary arithmetic operators and or-expressions where both the left hand and right hand expressions contains variable references, both their sets are merged but the reference is considered as weak.

5.2.6 The format of the rule infraction log

The log of all rule infractions needs to be searchable and able to summarize not only on events but also on referenced entities as described in the pilot study. The following information is saved in the log:

- The event in which context the rule have been breached
- The source from where the event had been harvested
- What rule have been breached
- The attributes, complete with their navigational path, that made the rule evaluation false
- If the attribute reference is strong or weak infraction

In order to achieve this, a database scheme was developed to contain this information about the infractions. The event, the source and the items that where navigated through where initially only represented by its id in the RF database. But during the first test runs it was found to be very cumbersome to have to look up the information against the API in order to reason about the infraction. In order to ease the process there where some extra columns added to the scheme. These where:

- Source name
- The name of each item that had been navigated through

The final database scheme is divided in three tables, one with the topmost information about the rule infraction, one table with basic information about the attribute and one table containing the path from the event to the final value. The design of this database are shown in Table 3, Table 4 and Table 5 .

Table 3: Scheme of the Rule Infraction table

Column name	**Description**
RuleName	Textual representation of the rule
RuleInstanceID	The event the rule vas evaluated for
Sourceid	The sourceid of the event
SourceName	The sourcs name
RuleInfractionID	Primary key

Table 4: Scheme of the Attribute table

Column name	**Description**
RuleInfractionID	Foreign key
IsStrong	Bit, representing if the attribute is directly liable
AttributeID	Primary key

Table 5: Scheme of the Attribute Path table

Column name	**Description**
AttributeID	Foreign key
Instance	The id of the current instance
AttributeName	The name of the attribute in the current item
Value	The value of the local attribute
ValueName	If the value is a reference to an item, this contains the items name
AttributePath	Primary key

In the topmost table (RuleInfraction) one row will be created for each rule infraction. In the second table (Attribute) one row is created for each attribute that contribute to breaching the rule. It is also noted here whether the attribute is strongly associated to the breach. Since each infraction can depend on several attributes there is a one-to-many relation between the RuleInfraction and Attribute tables. The last table (AttributePath) contains information about what path was navigated to reach the attribute. There is a one to many relation between Attribute and AttributePath.

The AttributePath table is recursive in the meaning that one row value can point to another row instance. To retrieve the entire navigated path a recursivel look up is performed. The recursions initial state is the row which InstanceID equals the InstanceID from the RuleInfractionID and AttributeID from the Attribute. The recursive step is performed by selecting a new row with the same the same AttributeID and whose InstanceID equals the Value of the current row. This is repeated until all rows with the the current AttributeID have been visited. If the attribute names are noted during this recursion they can be concatenated to create the path to the attribute as it could have been written in the Vulpecula language.

In the first implementation of the SQLite log each row was written as soon as it was know. This became a quite time consuming and some tests showed that batching 1000 rows together before writing could speed up the database time significantly.

The most straightforward use of this log is to look at what events that have been logged in the RuleInfraction tabel. Another possibility is to look for frequent occurrences of the same InstanceID in the AttributePath table. This could be an indication of an problematic item due to the fact that it is involved in a lot of rule infractions. Note that this item does not have to be an entity such as a company or person. It could just as well be a document or a source. The only requirement is that the rule have navigated through the item when looking up the value. The SourceID from the RuleInfraction table could also be used directly to find sources that produces large quantities of erroneous events. If the source is selected in this way it does not have to be part of the navigational path since it is always included.

5.2.7 Rules

The rules are designed to minimize the false positive logging of events. This is due the fact that the information from the RF service will still pass through the human filter of the user.

```
macro maxPrediction(years, months, days)
   document.published + period(years, months, days) >= start

macro statusLimit()
   (status == "planned" -> stop >= document.published) &&
   (status == "future" -> stop >= document.published) &&
   (status == "past" -> start <= document.published)
```

Figure 9: Simple macros for enhanched readability

In order to simplify the rules and make them more readable some macros where introduced. Some of these where for a simplified writing of how long in the future, the events time stamp compared to the documents time-stamp, an event type could been predicted. Examples of these are shown in Figure 9.

```
macro matchProductTypeWithCompany(pt, c)
   (pt=="drug" ->
      gicsSicMatch(c, "Health Care", {2833, 2834, 2836}, {})) &&
   (pt=="car" ->
      gicsSicMatch(c, "Consumer Discretionary", {3711, 3714}, {})) &&
   (pt=="electronics" ->
     gicsSicMatch(c, "Information Technology", {3651},
                  {357, 366, 367, 481, 482, 489})) &&
   (pt == "weapon" ->
     gicsSicMatch(c, "Industrials", {3761, 3795}, {348, 372, 373})) &&
   (pt == "aircraft" →
     gicsSicMatch(c, "Industrials", {}, {372})) && (pt == "other" -> true)
```

Figure 10: Macro for looking up company classification from product type

A more elaborated macro is shown in Figure 10. This works as an look up from the product_type that where found in by the listing of all values by attribute name in the pilot study. These where then matched with their corresponding industries, Gics and Sic number corresponding to these industries [14],[15]. If any of these where found in the company's gics listing or the company's industries it was considered a match.

6 Results

The result of this master thesis work is the extended rules implemented in the Vulpecula language and an application that given a declaration in the Vulpecula language tests the information in the RF database.

The application does not, in its present form, check all the events in the database but rather a subset of them. This is due to the immense time it would take to run through the entire database. The large variation between different fetches makes is hard to predict how long it would take to check the entire database. When conducting some test it was shown that the fetch time for one event type and one day could vary between some tenth of a second up to over ten minutes. The Figure 11 shows how much the fetch time varies with different API-calls.

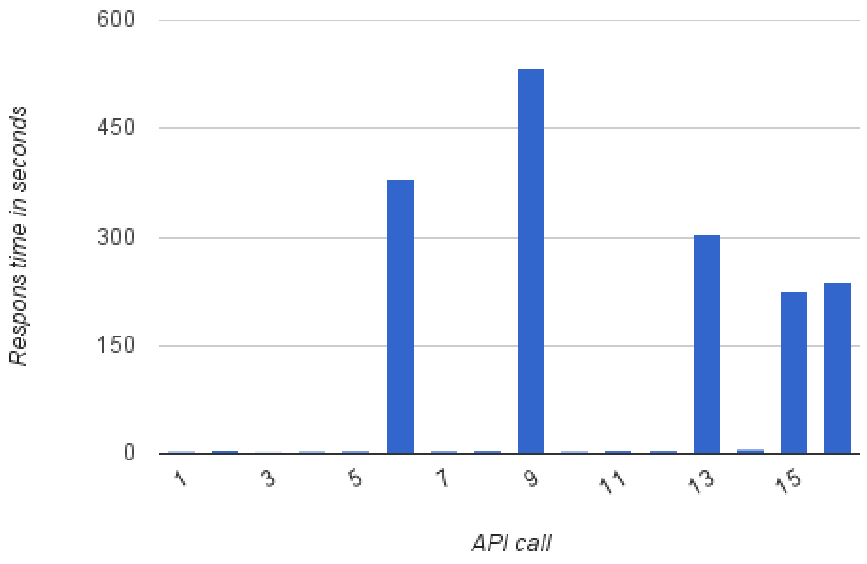

Figure 11: Difference in data fetching times

Due to the large variation in time its hard to say if this is a feasible way to check the entire database. If an estimate is done by taking the average response time per event and multiply it with the number of events in the database it would be in the order of years.

During one run with the rules written as part of this thesis one on a limited set of a maximum of twenty events from each event type each day and over the course of ten days a total of 680 infractions was caught, spread out over 468 different event instances. The vast majority of these infractions, 590 of them, are infractions of rules that either compares publish time of the associated document with the event time, sometimes limited by extra information in the event or knowledge about the event type that have been used at when designing the rules. Both the `statusLimit` and `maxPrediction` macro in Figure 9 exemplifies these kind of rules. During a closer examination was made of some of the events that failed these kind of rules it was found that one of the major causes was due to the linguistic interpretor which had created the event from an date notation in the document that was not the notation of the parsed events time. Most of the time there were multiple instances from the same document with different dates from the document as the event time. There where almost always one event that had been based on the correct date notation, and considered sane by the rules. Sometimes there can be be multiple events that was considered sane by the rules even though only one of them are correct.

Since there are 98 event types this means that about 2% of the tested events did not fulfill all their rules.

During this run the distribution between the different phases of the application was:

- Fetching: 1705 sec
- Evaluating: 0.189 sec
- Logging: 2.932 sec

7 Conclusion

This thesis report has investigated the possibility to apply semantic rules on an database built up by automatic harvesting of several different web sources, organized as different events. We have designed a language to describe rules about how the information of the different types are expected to be. The expressiveness of these rules have been limited due to the performance when accessing the information from the database. They can only operate on one event instance at a time. Even if there have not been any complete check against the entire database we have performed smaller tests which shows that the design and implementation works. The result would benefit greatly by having the rules revised by non-laymen. In order to do a complete check of the database the tool will need to use a more efficient way of accessing the information from the database since the current method is not feasible.

8 Future works

There are still parts of the work that could be extended upon or improved. Those that have been discussed during the terms of this work are presented here.

8.1 Evolving the rules

The rules where written by laymen without any good insight in the cooperate and economic world and could probably considerably improved. Not only in the can the existing improved but there are probably a lot of rules that are not even introduced during this work.

8.2 Using the log

The information from the log could be used in several different ways. It could be used as a log of events and items that need to be observed or removed from the database. Another use is to give events that does not fulfill the rules a lower likeliness to be show to the users. This is not limited to events but could also be applied to sources and other entities that are over referenced from the rule infractions.

The difference between strong and weak references for attributes were introduced late during the work and has not been properly evaluated yet. This concept is something that needs more consideration.

8.3 Global rules

It would be quite simple to extend the Vulpecula language to accept rules that are not only applied to a single event at the time. It could be both multiple events from the same event type of from different event types. This would give a new aspect to what kind of semantic rules that could be expressed. E.g. a company should not have any event about it that is timestamped earlier than a year before it was founded. In order to make them more efficient they they should probably be limited to express relations between events that share a common item. One idea that was discussed during the part of this project was to not actually compare every event but rather creating an allowed space for the attributes of the events describing the current entity.

8.4 Fuzzy logic

When the extended demands on the data were formalized it became clear that not all the rules where discrete, a lot of them describe demands that gradually go from feasible to unlikely. Some example of this the maximum prediction time of an event type and the age of a person. Both will start to continuously become more and more unlikely for higher values rather than fail at a specific value. In order to better represent these conditions some form of fuzzy logic could be used in stead of the two value logic of the current implementation. The comparison of attributes would be done against an interval instead of a fixed value. If this was to be introduced the logic of referenced values and strong/weak references would need to be rewritten or even scrapped.

9 References

[1] “Recorded Future” Internet: https://recordedfuture.com/ [November 15, 2011].

[2] “Recorded Future – A White Paper on Temporal Analytics.” Internet: http://blog.recordedfuture.com/2010/03/13/recorded-future-%E2%80%93-a-white-paper-on-temporal-analytics/, Mar. 13, 2010 [May 17, 2010].

[3] ISO/IEC 14977, Information technology - Syntactic metalanguage - Extended BNF.

[4] N. Bassiliades, P . M . D. Gray. “CoLan: A functional constraint language and its implementation”. *Data & Knowledge Engineering,* vol. 14, pp. 203-249, Feb. 1995.

[5] “Object Constraint Language Version 2.2.” Internet: http://www.omg.org/spec/OCL/2.2, [May 27, 2010].

[6] Gerd Wagner. “How to Design a General Rule Markup Language?” *In Invited talk at the Workshop XML Technologien für das Semantic Web*, 2002, pp. 19-37.

[7] Gerd Wagner. “Web Rules Need Two Kinds of Negation.” *Principles and Practice of Semantic Web Reasoning,* 2003, pp. 33-50.

[8] Gerd Wagner. “A Database Needs Two Kinds of Negation.” *Proceedings of the 3rd symposium on Mathematical fundamentals of database and knowledge base systems,* 1991, pp. 357 – 371.

[9] “Java Compiler Compiler.” Internet: http://javacc.java.net/, [Jun. 13, 2010].

[10] ”Joda Time - Java date and time API.” Internet: http://joda-time.sourceforge.net/, [2010].

[11] “Introducing JSON.” Internet: http://www.json.org/, [Sep. 1, 2011].

[12] Frank Carver. “Stringtree JSON.” Internet: http://www.stringtree.org/stringtree-json.html, [Aug. 6, 2010].

[13] “Visitor Pattern.” Internet: http://en.wikipedia.org/wiki/Visitor_pattern, [Sep. 1, 2011].

[14] “Standard Industrial Classification (SIC).” Internet: http://www.sec.gov/info/edgar/siccodes.htm, May 13, 2008 [Aug. 8, 2010].

[15] “GICS STRUCTURE & SUB-INDUSTRY DEFINITIONS (English) Effective as of Close of June 30, 2010.” Internet: http://www.mscibarra.com/resources/xls/GICS_map2010.xls, [Aug. 22, 2010].