SPINdle

User Guide

Version 1.0.4

Document build
April 19, 2010

Author:

Ho-Pun Lam (Brian)
Email: `oleklam@gmail.com`

License

Copyright © 2009-2010 National ICT Australia (NICTA)

This program is a free software; you can redistribute it and/or modify it under the terms of the Lesser GNU General Public License (the "License") as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version. You may not want to use this software except in compliance with the License.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the Lesser GNU General Public License for more details;

You should have received a copy of the License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

** If you have noticed any mistakes in this document, please email the author at your convenience.

Recent Changes

This section give quick overview on recent changes that are made in the software. The more detailed version notes are contained in the RELEASE_NOTES in the distribution directory.

SPINdle-1.0.4

- Reasoning with ambiguity propagation and well-founded semamtics are supported.
- Theories and conclusions saved can be loaded using a URL.
- New theory parser and outputter classes can be configured using the configuration file stored in "<SPINdle_HOME>/src/spindle/resources/io_conf.xml"; or can be searched by the I/O Manager automatically (section 2.1.1).
- JDOM library will be deprecated; instead, DOM and StAX will be used to handle all XML related document processing.
- The reasoning time is now sub-divided into the time used to transform the theory (into regular form), the timed used to remove defeaters, and time used to remove superiorties, and the time used for reasoning (conclusion generation).

SPINdle-1.0.3

- SDL and MDL reasoning with multiple heads is supported.

SPINdle-1.0.2

- Fixed incorrect conclusions for MDL reasoning.
- Reasoning with ambiguity blocking is supported.

Contents

License i
Recent Changes ii
Contents iii
List of Figures v
Preface vi

1 Introduction **1**
1.1 Defeasible Logic: An informal introduction 2
1.1.1 Basics of Defeasible Logic 2
1.1.2 Modal Defeasible Logic 4
1.1.3 Ambiguity blocking and Ambiguity propagation 4
1.2 SPINdle Architecture 5
1.3 Fornt-Ends support 7

2 Running SPINdle **8**
2.1 Invoking SPINdle 8
2.1.1 Command-line Options 9
2.2 Theory file extensions 10
2.3 Theory directory 10
2.4 System limits 10
2.4.1 The JVM heap size 10

3 Language **11**
3.1 Comment 11
3.2 Atoms and Literals 11
3.2.1 Atom 11
3.2.2 Basic Literal 12
3.2.3 Modalised Literal 12
3.3 Facts, Rules and Defeaters 12
3.4 Superiority relations 12
3.5 Conclusions 13
3.6 An example 13

4 Embedding SPINdle in JAVA applications **15**
4.1 The reasoner 15
4.1.1 Loading a Theory 15
4.1.2 Getting the theory 16

4.2 The Reasoning process 17
4.3 Extending the I/O interfaces 17
4.3.1 Creating Custom Theory Parser 18
4.3.2 Creating Custom Theory Outputter 18
4.3.3 Some useful information 19

Appendix **I**

A XML schema for defeasible theory **II**

Bibliography **VI**

List of Figures

1.1 A defeasible theory with ambiguity 4
1.2 SPINdle architecture 5
1.3 Theory Normalizer process 6

3.1 A defeasible theory with ambiguity (in DFL) 13
3.2 Conclusions of the ambiguous theory (in DFL) 14

Preface

The main goal of this guide is to provide an introduction to the defeasible logics reasoner – SPINdle . It teaches researchers how to carry out research using SPINdle and application developers how to embed SPINdle into their applications. In addition, it also shows readers how to make proper use of available features. It does not attempt to cover the theoretical aspect of defeasible logic in detail. Readers interested in this subject can refer to (Nute 1994; Antoniou *et al.* 2001) for details.

Intended audience

SPINdle , written in Java, implements reasoner to compute the consequence of theories in defeasible logic. The implementation covers both basic defeasible logic and modal defeasible logic. The reasoner for basic defeasible logic is based on the algorithm proposed by Maher (Maher 2001) while the reasoner for modal defeasible logic implements the algorithms of (Governatori & Rotolo 2008) .

This guide is intended for researchers and/or software developers who already have some knowledge in defeasible logic and wish to carry out research or build applications using SPINdle .

SPINdle users who are not familiar with defeasible logic nor the Java programming language will be benefit from consulting research papers or books on those subjects.

Before you begin

Before you begin, we assume that you have done the following:

- Installed the Java JDK6.
  If you do not have a recent version of the Java JDK6 installed, download and install Sun Java Standard Edition JDK[1].

- Installed Eclipse or your facourite Java IDE.
  In this guide, we use Eclipse[2] as our IDE environment. However, the reasoner does not tie you to Eclipse. You can use IntelliJ, NetBeans or any Java IDE

[1] Java JDK: http://java.sun.com/javase/downloads/?intcm=1281#need
[2] Eclipse: http://www.eclipse.org/

you prefer. If you use Java IDE other than Eclipse, the screen shot and some of the specific instructions in this guide will be different.

If your Java IDE does not include Apache Ant[3] support, you can download and unzip Ant to easily compile and run the reasoner. A sample "build.xml" file is included in the package for convenience.

- Downloaded the JDOM[4] library.
  (**Deprecated**. *As of version 1.0.4, DOM and StAX are used in preference to JDOM in handling all XML documents operations.*)
  JDOM (Java Document Object Model) is an open source Java-based document object model for XML document. It integrates with Document Object Model (DOM) and Simple API for XML (SAX), supports XPath and XSLT. It provides a way to represent XML document for easy and efficient reading, manipulation, and writing.

You may also optionally do the following:

- Download and install Eclipse SWT[5] library.
  There is also a defeasible theory editor that comes with SPINdle and is available for download through our web site. Before you can compile and run this editor, you need to download and install the SWT user interface library. Details regarding how to set up the SWT library on your platform can be found on the SWT FAQ page.

Getting ready

Downloading the source code

Interested readers can download the source codes (and/or the binaries) from

`http://spin.nicta.org.au/spindle`

SPINdle is distributed as a single `.zip` file that can be used on all supported platform. However, the SPINdle defeasible theory editor is platform dependent. You should download the editor according to your platform if you want to use it.

When SPINdle is unpacked, you should have a directory named `SPINdle_<version_no>` and contains the following files and sub-directories.

[3]Apache Ant: http://ant.apache.org
[4]JDOM: http://www.jdom.org
[5]Eclipse SWT: http://www.eclipse.org/swt

`README`	A quick start guide.
`LICENSE`	SPINdle license agreement.
`RELEASE_NOTES`	Release notes about recent changes.
`build.xml`	A sample Apache Ant build file.
`src` (optional)	It contains the full source code for SPINdle .
`lib` (optional)	It contains all the libraries required for building SPINdle .
`samples`	It contains some defeasible logic theories written in plain text and XML format.
`dist` (optional)	If this directory is present, it contains a current version of SPINdle archived as an executable jar file.

Building SPINdle from source

You can compile and build Java archive (.jar file) using Apache Ant with the `build.xml` file included. To compile the source, type:

```
ant compile
```

To build Java archive, type:

```
ant dist
```

There are also other Ant tasks available in the `build.xml`. Please refer to the `README` file for details.

Testing the program

There are some sample defeasible logic theories (in the "`samples`" directory) that you can use to test that you have installed/compiled SPINdle properly. To run SPINdle as a standalone application, type:

```
ant run
```

or in the command prompt, type:

```
java -jar spindle_<version_no>.jar [file1|dir1|url1] [file2|dir2|url1]...
```

You can add the theory file name(s) as Java argument(s) in the `build.xml`. However, if no theory file are given, the following output will be shown on the screen:

```
c:\SPINdle_1.0.0>java -jar dist\spindle_1.0.4.jar
*****************************************************************************
* SPINdle (version 1.0.4)                                                   *
* Copyright (C) 2009 NICTA Ltd.                                             *
* This software and its documentation is distributed under the terms of the *
* FSF Lesser GNU Public License (LGPL).                                     *
*                                                                           *
* This program comes with ABSOLUTELY NO WARRANTY; This is a free software   *
* and you are welcome to redistribute it under certain conditions; for      *
* details type:                                                             *
*         java -jar spindle-<version>.jar --app.license                     *
*****************************************************************************
```

```
Usage: java -jar spindle.jar [--options] [file1|dir1|url1] [file2|dir2|url2]..

where options include:
    --app.version                       show software version
    --app.license                       show software license
    --app.credit                        show credit for library embedded

    --log.level                         log level (ALL,INFO,FINE,FINEST)
    --app.showProgress                  show progress time interval
    --app.showResult show result on screen
    --app.progress.timeInterval         show progress time interval
    --reasoner.io.searchClasses         search for I/O classes
    --app.saveResult                    is save conclusions
    --app.result.folder                 folder for storing conclusions

    --reasoner.ambiguityPropagation     ambiguit propagation support
    --reasoner.wellFoundedSemantics     well-founded semantics support

** No file to process!
```

If a theory file was given as an arguement (for example, `samples/sdlTestTheory.dfl`), then the conclusions and performance statistics will be shown instead:

```
C:\SPINdle_1.0.0>java -jar dist/spindle_bin_1.0.0-0.jar samples/sdlTestTheory.dfl
*****************************************************************************
* SPINdle (version 1.0.0)                                                   *
* Copyright (C) 2009 NICTA Ltd.                                             *
* This software and its documentation is distributed under the terms of the  *
* FSF Lesser GNU Public License (LGPL).                                     *
*                                                                           *
* This program comes with ABSOLUTELY NO WARRANTY; This is free software, and *
* you are welcome to redistribute it under certain conditions; for details   *
* type:                                                                     *
*         java -jar spindle-<version>.jar --app.license                     *
*****************************************************************************

=========================
== application start!! ==
=========================
initialize application context - start
    setting I18N info...start
    reading system message......completed

...

=== Listener message: System info: 10 literal(s) pending to process
All pending conclusion(s) are evaluated!
Conclusions:
    +D a(X)
    +D b(X)
    +D c(X)
    -D -c(X)
    +d a(X)
    +d b(X)
    +d c(X)
    -d -c(X)
Reasoning time ended: Sat Apr 10 23:01:03 EST 2010
============================================================
File name      : file:samples/sdlTestTheory.dfl
no. of rules   : 4
no. of literals: 4
Load theory start at: Sat Apr 10 23:01:03 EST 2010
Load theory end at  : Sat Apr 10 23:01:03 EST 2010
```

```
    time used for loading theory: 31 ms (31 ms)
Regular form transformation time used: 15 ms
Defeaters removal time used: 0 ms
Superiorities removal time used: 16 ms
Reasoning start at: Sat Apr 10 23:01:03 EST 2010
Reasoning end at  : Sat Apr 10 23:01:03 EST 2010
    time used for reasoning: 15 ms (15 ms)
Total time used: 77 ms (77 ms)
Max. memory used: 1.055450439453125 MB
==============================================================

====================================
== Performance statistics summary ==
====================================
== I/O classes configuration time used: 47 ms
== No. of record(s) found: 1
== --- start
+------------+------------+-----------------+-----------------+ ... +-----------------+ ... +-------------+-----
|   No. of   |   No. of   |   Time used on  |   Time used on  | ... |  Time used on   | ... | Max. Memory |
|    Rules   |  Literals  |  loading theory | transform theory| ... |    reasoning    | ... |     used    | filename
+------------+------------+-----------------+-----------------+ ... +-----------------+ ... +-------------+-----
|          4 |          4 |       0.031 sec |       0.015 sec | ... |       0.015 sec | ... |     1.06 MB | file:...
+------------+------------+-----------------+-----------------+ ... +-----------------+ ... +-------------+-----
== --- end

Calling the shutdown routine...
terminate application context - start
terminate application context - end
========================================
=== Application shutdown completed! ===
========================================
```

Programming with SPINdle

To use SPINdle as a library or embedded rule engine, the file `spindle_<version_no>.jar` must either be on your classpath, be installed as a standard extension, or being recognized by your development tools (such as Eclipse). You should refer to the documentations of the development tools that you are using for details.

Questions

You can contact us at `oleklam@gmail.com` if you are having a problem with some aspect(s) of this guide, and we will do out best to address it.

Chapter 1

Introduction

Defeasible logic (DL) is a non-monotonic formalism originally proposed by Nute (Nute 1994). It is a simple rule-based reasoning approach that can reason with incomplete and contradictory information while preserving low computational complexity (Maher 2001). Over the years, the logic has been developed notably by (Antoniou *et al.* 2001; Antoninou & Maher 2002; Billington 1993). Its use has been advocated in various application domains, such as business rules and regulations (Antoniou *et al.* 1999), agent modeling and agent negotiations (Governatori & Rotolo 2004), applications to the Semantic Web (Bassiliades, Antoniou, & Vlahavas 2006) and business process compliance (Governatori, Milosevic, & Sadiq 2006). It is suitable to model situations where conflicting rules may appear simultaneously.

SPINdle is a reasoner for computing the consequences of defeasible theories. Its capable to perform efficient and scalable reasoning defeasible theories (including theories with over 1 million rules). The implementation covers both the standard defeasible logics and its modal extension. In addition, the current version (version 1.0.4) also support other variants of defeasible reasoning, such as ambiguity propagation and well-founded semantics.

SPINdle can be used as a standalone theory prover and can be embedded into any applications as a defeasible logic rule engine. It allows user, or agents, to issue queries on a given knowledge base, or a theory generated on the fly by other applications and produces the conclusions of its consequences automatically.

The following are some features of SPINdle :

- It supports all rule types of defeasible logic, such as fact, strict rules, defeasible rules, defeaters, and superiority.
- It supports Modal Defeasible Logic (Governatori & Rotolo 2008) with modal operator conversions.
- It supports reasoning with ambiguity propagation and well-founded semantics.
- It supports negation and conflicting (mutually exclusive) literals.
- Defeasible theory can be represented using plain text (Chapter 3) and XML

format. Besides, users also allowed to define their own representation format by implementing the associated theory parser- and outputter-classes.

SPINdle has been developed in the National ICT Australia - Queensland Research Laboratory. The newest versions of SPINdle and this document are available at

`http://spin.nicta.org.au/spindle`

1.1 Defeasible Logic: An informal introduction

1.1.1 Basics of Defeasible Logic

A *defeasible theory* D is a triple $(F, R, >)$ where F and R are finite set of facts and rules respectively, and $>$ is a superiority relation on R. Here SPINdle only considers rules that are essentially propositional. Rules containing free variables are interpreted as the set of their ground instances.

Facts are indisputable statements, represented either in form of states of affairs (literal and modal literal) or actions that have been performed. Facts are represented by predicates. For example, "John is a human" is represented by $human(John)$.

A *rule*, on the other hand, describes the relations between a set of literals (premises) and a literal (conclusion). We can specify the strength of the rule relation using the three kinds of rules supported by DL, namely: *strict rules*, *defeasible rules* and *defeaters*; and can specify the mode the rules used to connect the antecedent and the conclusion. However, in such situations, the conclusions derived will be *modal literals*.

Strict rules are rules in the classical sense: whenever the premises are indisputable (e.g. facts) then so is the conclusion. An example of a strict rule is: "human are mammal", written formally:

$$human(X) \rightarrow mammal(X)$$

Defeasible rules are rules that can be defeated by contrary evidence. An example of such a rule is "mammal cannot flies"; written formally:

$$mammal(X) \Rightarrow \neg flies(X)$$

The idea is that if we know that X is a mammal then we may conclude that it cannot fly *unless there is other, not inferior, evidence suggesting that it may fly* (for example that mammal is a bat).

Defeaters are a special kind of rules that cannot be used to draw any conclusions. Their only use is to prevent some conclusions. That is, they are used to defeat some defeasible rules by producing evidence to the contrary. For example the rule

$$heavy(X) \rightsquigarrow \neg flies(X)$$

states that an animal is heavy is not sufficient enough to conclude that it does not fly. It is only evidence against the conclusion that a heavy animal flies. In other

words, we don't wish to conclude that $\neg$*flies* if *heavy*, we simply want to prevent a conclusion *flies*.

DL is a "skeptical" nonmonotonic logic, meaning that it does not support contradictory conclusions. Instead DL seeks to resolve conflicts. In cases where there is some support for concluding A but also support for concluding $\neg A$, DL does not conclude neither of them. However, if the support for A has priority over the support for $\neg A$ then A is concluded.

As we have alluded to above, no conclusion can be drawn from conflicting rules in DL unless these rules are prioritised. The *superiority relation* is used to define priorities among rules, that is, where one rule may override the conclusion of another rule. For example, given the following facts:

$$\rightarrow bird \qquad\qquad \rightarrow brokenWing$$

and the defeasible rules:

$$\begin{array}{rrcl} r: & bird & \Rightarrow & fly \\ r': & brokenWing & \Rightarrow & \neg fly \end{array}$$

which contradict one another, no conclusion can be made about whether a bird with a broken wing can fly. But if we introduce a superiority relation $>$ with $r' > r$, then we can indeed conclude that the bird cannot fly.

We now give a short informal presentation of how conclusions are drawn in DL. Let D be a theory in DL (as described above). A *conclusion* of D is a tagged literal and can have one of the following four forms:

$+\Delta q$ meaning that q is definitely provable in D (i.e. using only facts and strict rules).

$-\Delta q$ meaning that we have proved that q is not definitely provable in D.

$+\partial q$ meaning that q is defeasible provable in D.

$-\partial q$ meaning that we have proved that q is not defeasible provable in D.

Strict derivations are obtained by forward chaining of strict rules while a defeasible conclusion p can be derived if there is a rule whose conclusion is p, whose prerequisites (antecedent) have either already been proved or given in the case at hand (i.e. facts), and any stronger rule whose conclusion is $\neg p$ has prerequisites that fail to be derived. In other words, a conclusion p is (defeasibly) derivable when:

- p is a fact; or
- there is an applicable strict or defeasible rule for p, and either
  - all the rules for $\neg p$ are discarded (i.e. not applicable) or
  - every rule for $\neg p$ is weaker than an applicable rule for p.

A full definition of the proof theory can be found in (Antoniou *et al.* 2001). Roughly, the rules with conclusion p form a team that competes with the team consisting of the rules with conclusion $\neg p$. If the former team wins p is defeasibly provable, whereas if the opposing team wins, p is non-provable.

1.1.2 Modal Defeasible Logic

Modal logics have been put forward to capture many different notions somehow related to the intensional nature of agency as well as many other notions. Usually modal logics are extensions of classical propositional logic with some intensional operators. Thus any modal logic should account for two components: (1) the underlying logical structure of the propositional base and (2) the logical behavior of the modal operators. Alas, as is well-known, classical propositional logic is not well suited to deal with real life scenarios. The main reason is that the descriptions of real-life cases are, very often, partial and somewhat unreliable. In such circumstances, classical propositional logic is doomed to suffer from the same problems.

On the other hand, the logic should specify how modalities can be introduced and manipulated. Some common rules for modalities are, e.g., **Necessitation** (from $\vdash \phi$ infer $\vdash \Box\phi$) and **RM** (from $\vdash \phi \rightarrow \psi$ infer $\vdash \Box\phi \rightarrow \Box\psi$). Both dictate conditions to introduce modalities purely based on the derivability and structure of the antecedent. These rules are related to well-known problem of omniscience and put unrealistic assumptions on the capability of an agent. However, if we take a constructive interpretation, we have that if an agent can build a derivation of ψ then she can build a derivation of $\Box\psi$.

To this end, SPINdle follows the semantics proposed by (Governatori & Rotolo 2008) on reasoning with modal defeasible logic. Readers interested in understand the semantics, modal operator conversions, conflict detections, conflict resolutions, and algorithm implemented in SPINdle please refer to the paper for details.

1.1.3 Ambiguity blocking and Ambiguity propagation

A literal is *ambiguous* if there is a chain of reasoning that supports a conclusion that p is true, another that supports $\neg p$ is true, and the superiority relation does not resolve this conflict.

Example 1 The following is a classic example of non-monotonic inheritance:

$$
\begin{array}{lrcl}
r1: & & \rightarrow & \mathit{quaker} \\
r2: & & \rightarrow & \neg\mathit{republican} \\
r3: & \mathit{quaker} & \Rightarrow & \mathit{pacifist} \\
r4: & \mathit{republican} & \Rightarrow & \neg\mathit{pacifist} \\
r5: & \mathit{republican} & \Rightarrow & \mathit{footballfan} \\
r6: & \mathit{pacifist} & \Rightarrow & \mathit{antimilitary} \\
r7: & \mathit{footballfan} & \Rightarrow & \neg\mathit{antimilitary}
\end{array}
$$

```
The superiority relation is empty.
```

Figure 1.1: A defeasible theory with ambiguity

$pacifist$ is ambiguous since the combination of r_1 and r_3 support $pacifist$; while the combination of r_2 and r_4 support $\neg pacifist$. Similarly, $antimilitary$ is ambiguous.

In DL the ambiguity of $pacifist$ results in the conclusions $-\partial pacifist$ and $-\partial\neg pacifist$. Since r_6 is consequently not applicable, DL concludes $+\partial\neg antimilitary$. This behaviour is called *ambiguity blocking*, since the ambiguity of $antimilitary$ has been blocked by the conclusion $-\partial pacifist$ and an unambiguous conclusion about $antimilitary$ has been drawn.

A preference for ambiguity blocking or ambiguity propagating behaviour is one of the properties of non-monotonic inheritace nets over which intuitions can clash (Touretzky, Horty, & Thomason 1987). (Stein 1992) argues that the ambiguity blocking results in unnatural pattern of conclusions in extensions of the above example. Ambiguity propagation results in fewer concluions being drawn, which might make it preferable when the cost of an incorrect conclusion is high.

For this reason, ambiguity propagation version of DL is of interest and thus both are supported by SPINdle .

1.2 SPINdle Architecture

The SPINdle architecture consists of three major components, which exchange information with each other or with users, as shown in Figure 1.2.

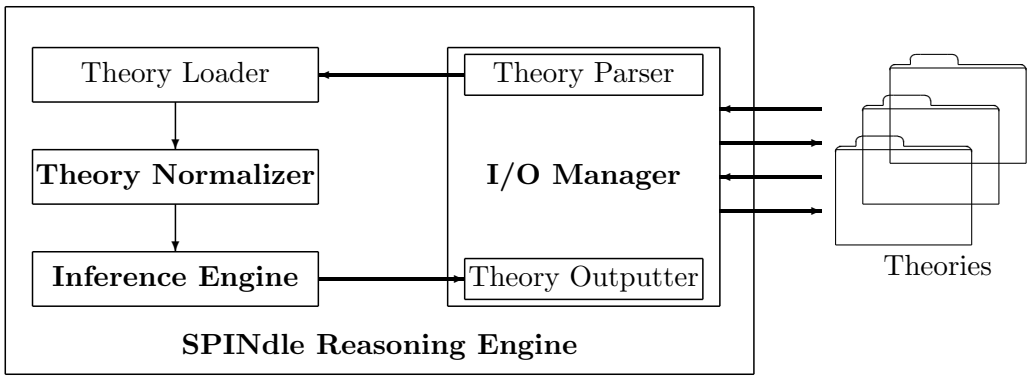

Figure 1.2: SPINdle architecture

I/O Manager The role of this module is to provide an interface to the users in loading the defeasible theories and storing the conclusion after computation. SPINdle accepts defeasible theories represented using different formats, and can be stored in the local file system or somewhere that can be accessed through the internet.

The *Theory Parser* is used to download theory files from a given URL (file location), parse the document, and translate the document into a data structure that can be processed in the next module. After a theory is loaded into SPINdle , users can modify/manipulate the theory according to their applications need.

The *Theory Outputter*, on the other hand, is used to store the modified theories or conclusions derived to a file, or export them as XML string for agent communications, which is a common scenario in the Semantic Web.

Theory Normalizer The whole inference process has two phases: A *pre-processing* phase where we transform the theory using the techniques described in (Antoniou *et al.* 2001) into an equivalent theory *without superiority relation* and *defeaters*, which later helps to simplify the reasoning process.

The *theory normalizer* module is responsible to this task by further breaking it down into three linear transformations: one to transform the theory to regular form, one to empty the superiority relation and one to empty the defeaters.

In addition to this, the theory normalizer also transform defeasible rules with *multiple heads* into an equivalent sets of rules with single head. It is expected that the transformed theory will produce the same sets of conclusions in the language of the theory they transform.

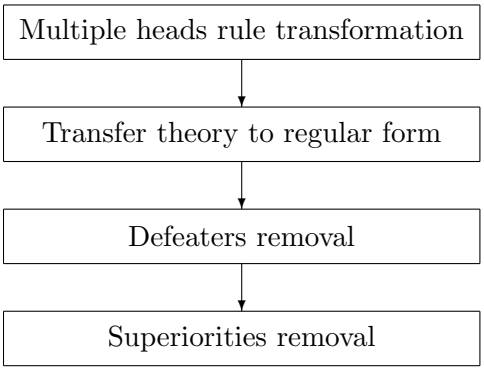

Figure 1.3: Theory Normalizer process

Inference Engine Following (Maher 2001) the *conclusions generations* phase (the *Inference Engine* module) is based on a series of (theory) transformations that allow us (1) to assert whether a literal is provable or not (and the strength of its derivation); (2) to progressively reduce and simplify a theory. The reasoner will, in turn:

- Assert each fact (as an atom) as a conclusion and remove the atom from the rules where the atom occurs positively in the body, and "deactivates" (remove) the rule(s) where the atom occurs negatively in the body. The atom is then removed from the list of atoms.
- Scan the list of rules for rules with empty head. It takes the head element as an atom and search for rule(s) with conflicting head. If there are no such rules then the atom is appended to the list of facts and the rule will be removed from the theory. Otherwise the atom will append to the pending conclusion list until all rule(s) with conflicting head can be proved negatively.

- Repeats the first step.
- The algorithm terminates when one of the two steps fails or when the list of facts is empty. On termination the reasoner output the set of conclusions.

It is important to note that the aforementioned algorithm is a generalized version that is common in inferencing both standard defeasible logic and modal defeasible logic. In the case of modal defeasible logic, due to the modal operator conversions, an additional process adding extra rules to the theory is needed. In addition, for a literal p with modal operator $\Box_i$, besides its complement (i.e., $\Box_i \neg p$), the same literal with modal operator(s) in conflict with $\Box_i$ should also be included in the conflict literal list (step 2) and only literal with strongest modality will be concluded.

1.3 Fornt-Ends support

There is a defeasible theory editor that can be used for editing standard defeasible theory and is available for download through the SPINdle web site.

Chapter 2

Running SPINdle

SPINdle comes with a `build.xml` that users can use to compile, build Java archive (.jar file) and run SPINdle using Apache Ant.

To compile the source, type

```
ant compile
```

To build Java archive, type

```
ant dist
```

The Java archive generated will be stored in `<SPINdle_HOME>/dist` directory with the name `spindle_<version_no>.jar`.

To build Java API documents (Javadoc), type

```
ant javadoc
```

To run SPINdle , type

```
ant run
```

And to clean up the work place, type

```
ant clean
```

Some test theories are available in `<SPINdle_HOME>/samples` directory. Users can make use of these theories to run and test the software accordingly.

2.1 Invoking SPINdle

The command line synopsis of SPINdle is as follows:

`java -jar spindle_<version_no>.jar [--options] [`$file_1$`|`dir_1`|`url_1`] [`$file_2$`|`dir_2`|`url_2`]...`

The meanings of the options will be detailed in section 2.1.1.

2.1.1 Command-line Options

By default, SPINdle shows all progress information and results derived onto the standard output. However, it is possible to change its behaviors by passing different options to SPINdle during start up.

The full set of command-line options is given below:

–app.license
Show the SPINdle license agreement.

–app.version
Show the current SPINdle release version.

–app.credit
Show the list of credit.

–log.level
Set the reasoner log level, values include: ALL,INFO,FINE,FINEST

–app.showProgress
Set whether to show the current progress or not.

–app.showProgress.timeInterval
Time interval for showing progress.

–app.showResult
Set whether to show the conclusions on screen or not (default true).

–app.saveResult
Set whether to save the conclusions or not

–app.result.folder
Folder location for storing the conclusions.

–reasoner.io.searchClasses
Instruct the I/O manager (`spindle.io.IOManager`) to search for (new) I/O classes available on the class path automatically.

–reasoner.ambiguityPropagation
Reasoning with ambiguity propagation.

–reasoner.wellFoundedSemantics
Reasoning with well-founded semantics.

2.2 Theory file extensions

SPINdle accepts theories to be described using different formalisms. The I/O manger (`spindle.io.IOManager`, as descibed before, will assoicate different file types with their respective theory parsers using their file name extensions. The current supported formalism includes: XML and DFL. The syntax of DFL theory will be discussed later in Chapter 3. However, its definition should be self contained and easy to understand. Readers having difficulities in understanding the syntax can send an email to the author for clarification. For the XML file, an XML schema of the defeasible theory can be found in Appendix A and the soft copy is available at

`<SPINdle_HOME>/src/spindleDefeasibleTheory.xsd`.

Saving conclusions/theories is just the same as loading a theory. Users are allowed to save the conclusions in different format. The current supported formalisms at this moment, again, are XML and DFL.

Users interested in developing their own theory parsers and outputter classes can refer to `spindle.io.parser` and `spindle.io.outputter` packages of the source code for details. More information about this will be discussed later (Chapter 4).

2.3 Theory directory

As shown above in the command line synopsis, users can put together different theories in the same directory and SPINdle will reason on them one-by-one and output the performance statistics at the end of the reasoning process. However, although theories are put together in the same place, they are still be reasoned separately and **NO** theories join operations will be performed.

2.4 System limits

2.4.1 The JVM heap size

With the heap, we refer to the memory area used by the JVM. The initial heap size of JVM is 64M bytes by default. However, for exceptionally large theories, you may need to increase the heap size (-Xms and -Xmx options) of the JVM or an exception will be thrown.

Chapter 3

Language

As described in section 1.1, a common structure for a defeasible theory includes:

- Facts, in form of states of affairs (literal or modal literal), describing the current domain, or some contextual information that is known to be truth;
- Rules describing the relations between a set of literals (premise) and a literal (conclusion);
- Defeaters that can be used to prevent some conclusions from occur; and
- Superiority relations that define priorities among rules.

This chapter describes the SPINdle DFL language which can be used to describe a defeasible theory. It is simple, self-contained and of easy to understand.

3.1 Comment

Comments are important features of all computer programs. In DFL, a comment is a sequence of characters beginning with a "#" character and is terminated by the next newline character. Any characters in-between will be ignored.

3.2 Atoms and Literals

3.2.1 Atom

An *atom* is a predicate symbol (the name of the atom) that is optionally followed by a parenthesized list of terms.

Sample atoms: $foo(X)$, a, abc, foo

> However, the current SPINdle implementation is still propositional based. All terms inside the parenthesis will be ignored at this moment.

3.2.2 Basic Literal

A *basic literal* is either an atom a or its negation $\neg a$.

The classical negation $\neg a$ of an atom a is denoted by a minus sign $(-)$ that is immediately before the atom name, i.e.: $-a$.

3.2.3 Modalised Literal

A *modalised literal* is of the form

$$[\Box]a$$

where $\Box$ is the modal operator representing mental states of the literal.

Sample literals: $[INT]foo(X)$, $[OBL]\neg smoke$, $[PER]leave(X)$

3.3 Facts, Rules and Defeaters

The following is the basic form of rule in DFL:

$$Rx[\Box] : a_1, a_2, \ldots, a_i, -b_1, -b_2, \ldots, -b_j \triangleright c_1, c_2, \ldots, c_k$$

where Rx is the rule label, $\Box$ is the modal operator of the rule, $a_1, a_2, \ldots, a_i, -b_1, -b_2, \ldots, -b_j$ and $c_1, c_2, \ldots, c_k$ are the body and head (modalised) literals of the rule respectively, and $\triangleright$ is the rule type symbol (Table 3.1).

A rule label will be generated by the theory parser if the rule label is missing in the original rule. While a rule can have no body literals, there should be at least one literal in the head, or an exception will be thrown while adding the rule to the theory.

Rule Type	Symbol
Fact	`>>`
Strict Rules	`->`
Defeasible Rules	`=>`
Defeaters	`~>`

Table 3.1: Rule Type-Symbol association list

Besides, it is also important to note that both facts, strict rules and defeaters can have only **ONE** head literals while defeasible rules is the only rule type that can have multiple head literals. In addition, note also that there should be **NO** body literals for facts. If a fact can only be derived under some constraints, it should therefore be represented using a strict rule instead.

3.4 Superiority relations

A superiority relations is represented by putting a "`>`" symbol between two rule label. For example: `r1 > r2`, where `r1` and `r2` are the rule labels of the two rules with conflicting (head) literal(s).

3.5 Conclusions

After running SPINdle , various conclusions of a defeasible theory T can be printed on the screen and/or stored in the local file system. A conclusion of T is a tagged literals and can have one of the following four forms:

$+Dq$ This means that q is definitely provable in T. That is, it is a fact, or can be proved using only strict rules.

$-Dq$ This means that we have proved that q is not definitely provable in T.

$+dq$ This means that q is defeasibly provable in T.

$-dq$ This means that we have proved that q is not defeasibly provable in T.

3.6 An example

The following example shows the same theory in Figure 1.1 but written in DFL:

$r1$:		-> *quaker*
$r2$:		-> -*republican*
$r3$:	*quaker*	=> *pacifist*
$r4$:	*republican*	=> -*pacifist*
$r5$:	*republican*	=> *footballfan*
$r6$:	*pacifist*	=> *antimilitary*
$r7$:	*footballfan*	=> -*antimilitary*

```
The superiority relation is empty.
```

Figure 3.1: A defeasible theory with ambiguity (in DFL)

and the followings are the conclusions derived with ambiguity blocking:

$$
\begin{array}{ll}
+D & quaker(X) \\
+D & republican(X) \\
-D & antimilitary(X) \\
-D & \text{-}antimilitary(X) \\
-D & footballfan(X) \\
-D & pacifist(X) \\
-D & \text{-}pacifist(X) \\
+d & \text{-}antimilitary(X) \\
+d & footballfan(X) \\
+d & quaker(X) \\
+d & republican(X) \\
-d & antimilitary(X) \\
-d & pacifist(X) \\
-d & \text{-}pacifist(X)
\end{array}
$$

Figure 3.2: Conclusions of the ambiguous theory (in DFL)

Chapter 4

Embedding SPINdle in JAVA applications

SPINdle can be used as a standalone theory prover and can be embedded into any Java applications as a defeasible logic rule engine. This chapter provides the information on how SPINdle can be used as a Java library, providing defeasible reasoning features to any Java applications. Users are also encouraged to refer to the API documentation for the complete story on these classes.

4.1 The reasoner

The `spindle.reasoner` class if the defeasible rule engine itself. Each reasoner has its own set of working memory, logger, rules, etc. To embed SPINdle in a Java application, users simply need to create one or more instances of the `spindle.reasoner` class in their applications and manipulate them appropriately. We will cover this in more details later. Here we will cover some general features of the `spindle.reasoner` class.

4.1.1 Loading a Theory

Defeasible theory can be store as a string, in a file, or somewhere on the internet. According to different scenarios, theories are required to be retrieved from different locations using different methods. Currently there are four methods to load a theory onto the reasoner:

```
loadTheory(Theory theory);
loadTheory(URL url);
loadTheory(File file);
loadTheory(String xmlString);
loadTheory(String[] dflString);
```

As shown above, theories can be loaded using a URL or a File location. For theories already available on the memory, the can be loaded onto the reasoner using the other two methods: `loadTheory(String)` and `loadTheory(String[])`.

The former is for theory represented using XML formalism while the latter is for theory represented using DFL.

The following code snippet shows how to load a defeasible theory from a URL:

```
public static void main(String[] args) {
    try {
        URL url = new URL("http://somewhere.net/defeasibleTheory.xml");
        Reasoner reasoner=new Reasoner();
        reasoner.loadTheory(url);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Note that if the URL does not provide any information about how the theory is formalized, then the "XML" theory parser will be used as XML is the commonly used formalism in internet communication.

In addition, the two theory parser classes included also provide some static methods to parse the stored information into their associated classes. For details, users can take a look at the Java documentation of the two parser class (`spindle.io.parser.DflTheoryParser` and `spindle.io.parser.XmlTheoryParser`):

```
Theory DflTheoryParser.getTheory(String theorySttring,
    AppLogger logger) throws ParserException
Map<Literal, Map<ConclusionType, Conclusion>> DflTheoryParser.getConclusions(
    String theorySttring, AppLogger logger) throws ParserException

Theory XmlTheoryParser.getTheory(String theorySttring,
    AppLogger logger) throws ParserException
Map<Literal, Map<ConclusionType, Conclusion>> XmlTheoryParser.getConclusions(
    String theorySttring, AppLogger logger) throws ParserException
```

The following code snippet shows how to generate a theory (written in DFL) from a string already stored in the memory:

```
public static void main(String[] args) {
    String theoryStr="a->b \n b=>c \n b >>a";
    try {
        Theory theory = DflTheoryParser.getTheory(theoryStr, null);
    } catch (ParserException e) {
        e.printStackTrace();
    }
}
```

4.1.2 Getting the theory

After loading the theory into the resoner, it is possible to retrieve it back from the reasoner to the applications. The `spindle.reasoner.getTheory()` method

will return an instance of the currently working theory to the users. However, it is important to note that depending on the methods that you may have called after loading, the theory returned may not necessary be the same as the theory originally loaded onto the reasoner.

4.2 The Reasoning process

As discussed before in section 1.2, the reasoning process is divided into two main steps: Theory transformation and Conclusions generation. And the theory transformations is further divided into three sub-process, namely: (1) Theory transform to regular form, (2) defeaters removal and (3) superiorties relation removal, as shown in Figure 1.3.

Inside the reasoner, the processes mentioned above can be done using the following methods:

```
ProcessStatus transformTheoryToRegularForm() throws ReasonerException

ProcessStatus removeDefeater() throws ReasonerException

ProcessStatus removeSuperiority() throws ReasonerException

Map<Literal, Map<ConclusionType, Conclusion>> getConclusions()
    throws ReasonerException
```

As the names suggested, the `transformTheoryToRegularForm()` method will transform rules with multiple heads into rules with single head and then transform the theory into normal form; the `removeDefeater()` and `removeSuperiority()` methods will remove all defeaters and superiority relations in the theory respectively; and the `getConclusions()` method will perform the real reasoning process and return the set of conclusions.

Note that the first three methods will return a `com.app.utils.Utils.ProcessStatus.SUCCESS` if the process is success; or an exception will throw otherwise.

4.3 Extending the I/O interfaces

As discussed earlier in this user guide, defeasible theory can be represented using different formalism and what we needed in SPINdle is to associated them with their respective theory parser- and theory outputter-classes. And SPINdle makes this task easy. There are mainly two strategies to follow:

- Create a theory theory parser/outputter-class by extending existing parser/outputter-classes; or
- Create an entirely new theory parser/outputter-class written in Java language.

There are numerous third party libraries available that can integrate into SPINdle as if necessary[1].

4.3.1 Creating Custom Theory Parser

The most effective way to create new theory parsers is to extend the `spindle.io.parser.TheoryParserBase` class, which already contains some useful constants and methods that are ready to use.

However, if you decided to create a new theory parser on your own, you should implement all the three methods as stated in the `spindle.io.TheoryParser` interface, as shown below:

```
String getParserType();

Theory getTheory(InputStream ins) throws ParserException;

Map<Literal, Map<ConclusionType, Conclusion>> getConclusions(InputStream ins)
    throws ParserException;
```

The `getParserType()` method simply return the parser type, or to be more concise, the file extensions that is going to associate with this theory parser and it is what the I/O Manager (`spindle.io.IOManager` used to retrieve the correct parser from its parsers set.

The second and third method, `getTheory()` and `getConclusions()`, accept an input stream as an input and return the theory parsed or the conclusions stored.

After completed implementing the theory parser, users can then modify the configuration file located at `spindle/resources/io_conf.xml`, by adding a line indicating the class name of the new parser. The following code shows the default content of the configuration file.

```
<spindle>
    <io classname="spindle.io.parser.XmlTheoryParser" />
    <io classname="spindle.io.parser.DflTheoryParser" />
    <io classname="spindle.io.outputter.XmlTheoryOutputter" />
    <io classname="spindle.io.outputter.DflTheoryOutputter" />
</spindle>
```

Or, at users' preference, the I/O Manager can detect all the parser classes available on the classpath automatically (section 2.1.1). However, it is not recommanded for big applications as considerable time is required.

4.3.2 Creating Custom Theory Outputter

Similiarly, creating custom theory outputter classes is as simple as creating theory parser and the most effective way is to extend the `spindle.io.outputter.TheoryOutputterBase`

[1]For example, users may need to consider using Jena (http://jena.sourceforge.net/) to parse the RDF file while creating a RDF theory parser.

class, which again, already contains some useful contants and methods for the theory outputters. However, users can implement their own theory outputter class from scratch by implementing all the methods stated in the `spindle.io.TheoryOutputter` interface, as shown below:

```
String getOutputterType();

void save(OutputStream os,
    Theory theory) throws OutputterException;

void save(OutputStream os,
    List<Conclusion> conclusionsAsList) throws OutputterException;
```

Again, the `getOutputter()` method will return the outputter type, i.e. the file extendsions that it is going to associate with. Both the two `save()` methods will take an output stream as an argument, followed by either a theory or the list of conclusions that is going to be saved.

After completed implementing the theory outputter, users can then modify the configuration file as shown above, or ask the I/O Manager to identify the theory outputter classes automatically during the start up.

4.3.3 Some useful information

Creating a Theory

Users can create a theory (`spindle.dom.Theory`) in memory simply by typing the following command:

```
spindle.dom.Theory theory = new spindle.dom.Theory();
```

Creating a Rule

A rule in SPINdle can be generate using the `spindle.dom.Rule` class. Users are requried to pass the *rule label* and *rule type* as an initial arguments. If the original rule statment does not contains information about the rule label, it is, thus, the responsiblities of the parer class to generate a dummy/temporary rule label to the rule. The theory class will **NOT** assign any rule label to a rule. It is also the responsibilities of the parser class to make sure that the rule label being generated are of unique, as no rules with the same rule label is allowed.

Creating a Literal

Generating a literal is the most freqent job when parsing a theory. Instead of creating a literal using the constructors in `spindle.dom.literal` class according to different cases, users may consider calling the static methods in the `spindle.dom.DomUtilities` class, which can help in simplifying the task.

```
public static Literal getLiteral(final String name, final boolean isNegation);
```

```
public static Literal getLiteral(final String name, final boolean isNegation,
    final String modeName, final boolean isModeNegation,
    final String[] predicates);

public static Literal getLiteral(final String name, final boolean isNegation,
    final String modeName, final boolean isModeNegation,
    final String[] predicates,
    final boolean isPlaceHolder);

public static Literal getLiteral(final String name, final boolean isNegation,
    final Mode mode,
    final String[] predicates,
    final boolean isPlaceHolder);
```

Appendix

Appendix A

XML schema for defeasible theory

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://spin.nicta.org.au/spindleOnline/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://spin.nicta.org.au/spindleOnline/">

<xsd:element name="theory" type="ctTheory"></xsd:element>

<xsd:complexType name="ctTheory">
<xsd:sequence>
<xsd:element name="fact" type="ctFact" maxOccurs="unbounded"
minOccurs="0">
</xsd:element>
<xsd:element name="rule" type="ctRule" maxOccurs="unbounded"
minOccurs="0">
</xsd:element>
<xsd:element name="superiority" type="ctSuperiority"
maxOccurs="unbounded" minOccurs="0"></xsd:element>
<xsd:element name="coversion" type="ctConversion"
maxOccurs="unbounded" minOccurs="0"></xsd:element>
<xsd:element name="conflict" type="ctConflict" maxOccurs="unbounded"
minOccurs="0"></xsd:element>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ctLiteral">
<xsd:sequence>
<xsd:choice maxOccurs="1" minOccurs="0">
<xsd:element name="mode" type="stMode"></xsd:element>
<xsd:element name="not" type="ctNegMode"></xsd:element>
</xsd:choice>
<xsd:choice maxOccurs="1" minOccurs="0">
<xsd:element name="atom" type="stAtom"></xsd:element>
<xsd:element name="not" type="ctAtom"></xsd:element>
</xsd:choice>
</xsd:sequence>
<xsd:attribute name="mode" type="xsd:string"></xsd:attribute>
</xsd:complexType>

<xsd:complexType name="ctLiteralList">
<xsd:choice>
<xsd:element name="literal" type="ctLiteral" minOccurs="1"
maxOccurs="1">
</xsd:element>
<xsd:element name="and" type="ctAndLiterals" maxOccurs="1" minOccurs="1"></xsd:element>
```

```
</xsd:choice>
</xsd:complexType>

<xsd:complexType name="ctRule">
<xsd:sequence>
<xsd:element name="head" type="ctLiteralList" maxOccurs="1"
minOccurs="1">
</xsd:element>
<xsd:element name="body" type="ctLiteralList"
maxOccurs="1" minOccurs="0">
</xsd:element>
<xsd:choice maxOccurs="1" minOccurs="0">
<xsd:element name="mode" type="stMode"></xsd:element>
<xsd:element name="not" type="ctNegMode"></xsd:element>
</xsd:choice>
</xsd:sequence>
<xsd:attribute name="label" type="xsd:string"></xsd:attribute>
<xsd:attribute name="strength" type="xsd:string"></xsd:attribute>
</xsd:complexType>

<xsd:complexType name="ctAndLiterals">
<xsd:sequence>
<xsd:element name="literal" type="ctLiteral" maxOccurs="unbounded"
minOccurs="1"></xsd:element>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ctConversion">
<xsd:sequence>
<xsd:element name="from" type="xsd:string" maxOccurs="1"
minOccurs="1"></xsd:element>
<xsd:element name="to" type="xsd:string" maxOccurs="unbounded"
minOccurs="1"></xsd:element>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ctConflict">
<xsd:sequence>
<xsd:element name="conflictMode" type="xsd:string" maxOccurs="1"
minOccurs="1"></xsd:element>
<xsd:element name="conflictWith" type="xsd:string"
maxOccurs="unbounded" minOccurs="1"></xsd:element>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ctSuperiority">
<xsd:attribute name="superior" type="xsd:string"></xsd:attribute>
<xsd:attribute name="inferior" type="xsd:string"></xsd:attribute>
</xsd:complexType>

<xsd:complexType name="ctFact">
<xsd:sequence>
<xsd:element name="literal" type="ctLiteral"></xsd:element>
</xsd:sequence>
<xsd:attribute name="label" type="xsd:string"></xsd:attribute>
<xsd:attribute name="mode" type="xsd:string"></xsd:attribute>
</xsd:complexType>

<xsd:simpleType name="stMode">
<xsd:restriction base="xsd:string"></xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="ctNegMode">
<xsd:sequence>
<xsd:element name="mode" type="stMode"></xsd:element>
```

```
</xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="stAtom">
<xsd:restriction base="xsd:string"></xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="ctAtom">
<xsd:sequence>
<xsd:element name="atom" type="stAtom" maxOccurs="1" minOccurs="1"></xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Index

I/O Manager, 5, 9, 10
    Theory Outputter, 6, 10
    Theory Parser, 5, 10
IDE
    Eclipse, v

Library
    JDOM, vi
    SWT, vi

System Architecture, 5
    I/O Manager, *see* I/O Manager
    Inference Engine, 6
    Theory Normalizer, 6

References

[Antoninou & Maher 2002] Antoninou, G., and Maher, M. 2002. Embedding defeasible logic into logic programs. *Theory and Practice of Logic Programming (TPLP)* 6(6):703–735.

[Antoniou *et al.* 1999] Antoniou, G.; Billington, D.; Governatori, G.; and Maher, M. J. 1999. On the modeling and analysis of regulations. In *Proceedings of the Australian Conference Information Systems*, 20–29.

[Antoniou *et al.* 2001] Antoniou, G.; Billington, D.; Governatori, G.; and Maher, M. J. 2001. Representation results for defeasible logic. *ACM Transactions on Computational Logic* 2(2):255–286.

[Bassiliades, Antoniou, & Vlahavas 2006] Bassiliades, N.; Antoniou, G.; and Vlahavas, I. 2006. A defeasible logic reasoner for the semantic web. *International Journal of Semantic Web and Information Systems (IJSWIS)* 2(1):1–41.

[Billington 1993] Billington, D. 1993. Defeasible logic is stable. *J Logic Computation* 3(4):379–400.

[Governatori & Rotolo 2004] Governatori, G., and Rotolo, A. 2004. *Deontic Logic in Computer Science*. Number 3065 in LNAI. Springer-Verlag. chapter Defeasible Logic: Agency, Intention and Obligation, 114–128.

[Governatori & Rotolo 2008] Governatori, G., and Rotolo, A. 2008. Bio logical agents: Norms, beliefs, intentions in defeasible logic. *Journal of Autonomous Agents and Multi Agent Systems* 36–69.

[Governatori, Milosevic, & Sadiq 2006] Governatori, G.; Milosevic, Z.; and Sadiq, S. 2006. Compliance checking between business processes and business contracts. In *10th International Enterprise Distributed Object Computing Conference (EDOC 2006)*, 221–232. IEEE Computing Society.

[Maher 2001] Maher, M. J. 2001. Propositional defeasible logic has linear complexity. *Theory and Practice of Logic Programming* 1(6):691–711.

[Nute 1994] Nute, D. 1994. Defeasible logic. In Gabbay, D., and Hogger, C., eds., *Handbook of Logic for Artificial Intelligence and Logic Programming*, volume III. Oxford University Press. 353–395.

[Stein 1992] Stein, L. A. 1992. Resolving ambiguity in nonmonotonic inheritance hierarchies. *Artif. Intell.* 55(2-3):259–310.

[Touretzky, Horty, & Thomason 1987] Touretzky, D. S.; Horty, J. F.; and Thomason, R. H. 1987. A clash of intuitions: the current state of nonmonotonic multiple inheritance systems. In *IJCAI'87: Proceedings of the 10th international joint conference on Artificial intelligence*, 476–482. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.