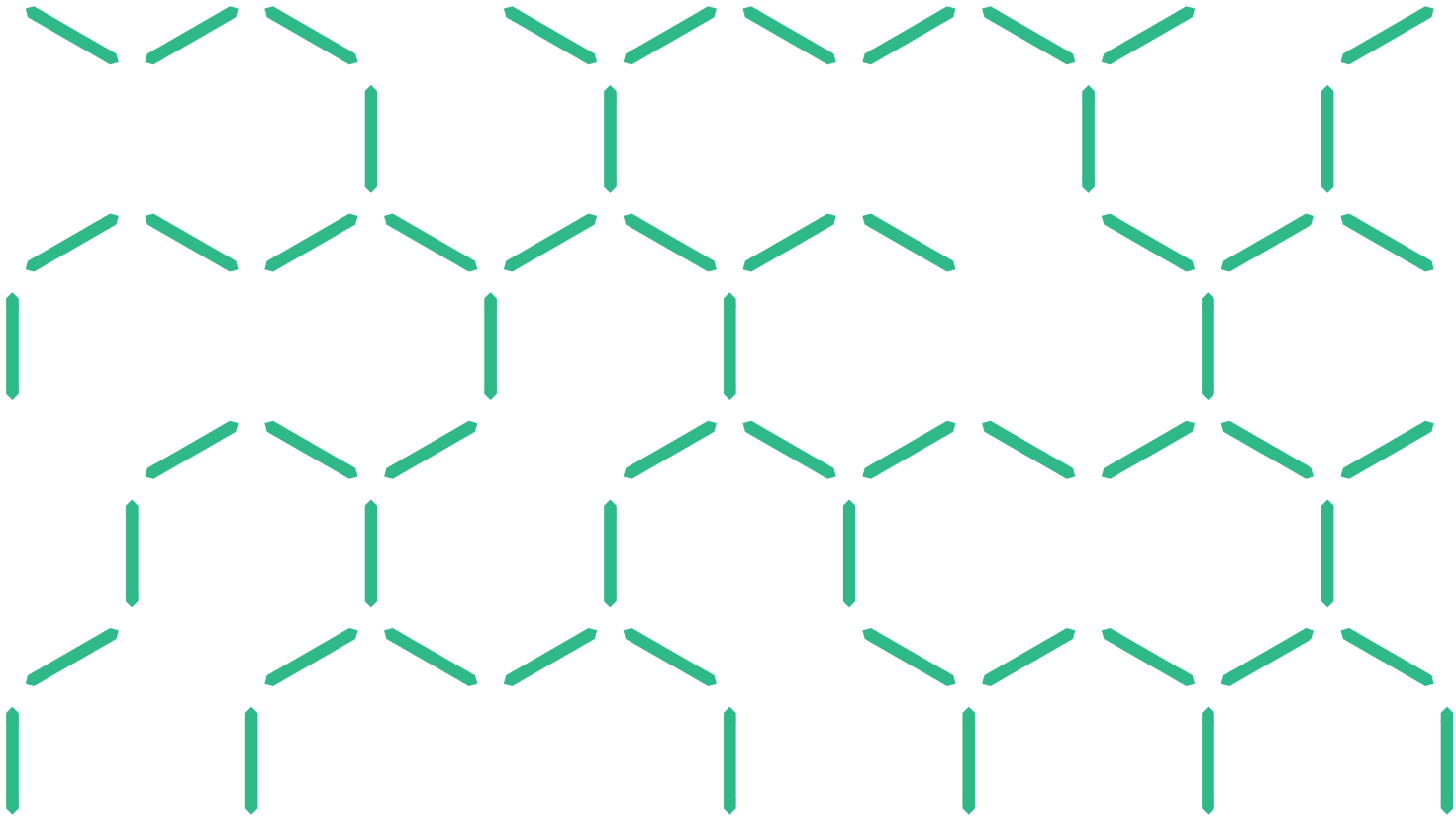

SPINdle

User Guide

Version 2.2.4

Ho-Pun Lam

2015/12/21

SPINdle - User Guide

Version 2.2.4

Prepared by:
Ho-Pun Lam
Email: brian.lam@nicta.com.au

Software System Research Group, Data61

Last update: 2015/12/21

For the latest version of this book and supplementary materials, please visit:
http://spin.nicta.org.au/spindle

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

License

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 [2, 19] 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. Version 1.x.x of the reasoner for basic defeasible logic is based on the algorithms proposed by Maher [18] while the reasoner for modal defeasible logic implements the algorithms of [9].

Version 2.x.x of the reasoner is based on the new algorithm that we devised [16], of which no removal of superiority relations is needed. Besides, our new algorithm also facilitates the computation of ambiguity propagation variants and well-founded semantics of defeasible reasoning [14].

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.

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.

Source code

The source code of SPINdle and the examples presented in this user guide can be obtained from the SPINdle web site:

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

or the SPINdle Sourceforge page:

`http://sourceforge.net/projects/spindlereasoner`

Questions

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

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-2.2.4

- Fixed minor bugs on XML Defeasible theory parsing and saving.

SPINdle-2.2.2

- Rules inference statuses logging and reasoning with negated modality are supported.
- The XML schema for defeasible theories and conclusions (and the associated theory parser and outputter) is updated.
- Minor bugs fixed.

SPINdle-2.2.0

- Rules inference statuses logging and reasoning with negated modality are supported.
- Minor bugs fixed.

SPINdle-2.1.1

- Minor bug fixed in DFL theory parser.

SPINdle-2.1.0

- A few system variables (for date and value computation) are introduced.
- Literal contents in a form of boolean expressions are supported.
- The exception classes are re-arranged and new exception classes are introduced.
- The DFL theory parser is updated with a new enhanced version.

SPINdle-2.0.5

- Minor bugs fixed in Well-Founded Semantics.

SPINdle-2.0.4

- Minor bugs fixed.

SPINdle-2.0.3

- Minor bugs fixed.

SPINdle-2.0.2

- Minor bugs fixed.

SPINdle-2.0.1

- Minor bugs fixed for Well-founded semantics.

SPINdle-2.0.0

- New reasoning algorithm without removing superiority relations is implemented.
- Ambiguity propagation variant with superiority relations is supported.
- A console application for theory testing is added to the package.

SPINdle-1.0.6

- Minor bugs fixed for MDL theory transformation.

SPINdle-1.0.5

- Bugs fixed in Well-Founded Semantics with conflicting literal(s) in loop(s).

SPINdle-1.0.4

- Reasoning with ambiguity propagation (*with no superiority relations in theory*) and well-founded semantics 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.
- 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 superiorities, 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

List of Figures

List of Tables

Chapter 1

Introduction

Defeasible Logic (DL) is a non-monotonic formalism originally proposed by Nute [19]. It is a simple rule-based reasoning approach that can reason with incomplete and contradictory information while preserving low computational complexity [18]. Over the years, the logic has been developed notably by [2, 3, 5]. Its use has been advocated in various application domains, such as business rules and regulations [1], agent modeling and agent negotiations [10], applications to the Semantic Web [4] and business process compliance [8]. It is suitable to model situations where conflicting rules may appear simultaneously.

SPINdle [15] is a defeasible reasoner for computing the consequences of defeasible theories. Its is capable to perform efficient and scalable reasoning over large volumes of defeasible rules, as required by many real world applications. The implementation covers both the standard defeasible logics and its modal extension. In addition, as of version 1.0.4, it also supports defeasible reasoning with other variants, 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 users, or agents, to issue queries on a given knowledge base, or a theory generated on the fly by an applications, and computes the consequences automatically.

The following are the features of SPINdle:

- It supports all rule types of DL, namely: *fact*, *strict rules*, *defeasible rules*, *defeaters* and *superiority relations*.
- It supports negation and conflicting (mutually exclusive) literals.
- It supports *conclusion reparation* with preference operators over *defeasible rules* [6, 7].
- It provides a flexible mechanisms that allows users to capture different intuitions of a defeasible theory. To be precise, SPINdle supports defeasible reasoning with different variants, which includes ambiguity blocking, ambiguity propagation, well-founded semantics, and their combinations.
- For each literals, two kinds of conclusions, definite provability and defeasible provability, are inferred and can be retrieved from the conclusions sets in accordance with the strength of proof that users interested in.
- It supports Modal Defeasible Logic (MDL) with modal operator conversions and conflict resolutions (cf. Section 1.1.2 and [9]).
- It is feasible for implementation in embedded platforms.
- It is extensible through additional elements to take advantages of the underlying reasoning facilities.

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

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

This paper is organized as follows. Section 1.1 describes the background information of DL and its modal extension. Readers with knowledge about these may skip this section.

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 prioritized. 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:

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

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 q is definitely rejected in D.

$+\partial q$**:** meaning that q is defeasibly provable in D.

$-\partial q$**:** meaning that q is defeasibly rejected in D.

There are different and sometimes incompatible intuitions behind what counts as a non-monotonic derivation. For a sequence of tagged literals to be a proof, it must satisfy certain conditions[1]:

$+\Delta$) If $P(n+1) = +\Delta q$ then either
(1) $q \in F$ or
(2) $\exists r \in R_s[q] \forall a \in A(r) : +\Delta a \in P(1..n)$.

That means, to prove $+\Delta q$ we need to establish a proof of q using only facts and strict rules. This is a deduction in the classical sense - no proofs for the negation of q need to be considered (in contrast to defeasible provability below where opposing chain s of reasoning must be taken into account). Thus it is a *monotonic proof.*

$-\Delta$) If $P(n+1) = -\Delta q$ then
(1) $q \notin F$ and
(2) $\forall r \in R_s[q]\ \exists a \in A(r) : -\Delta a \in P(1..n)$.

To prove $-\Delta q$, i.e., that q is not definitely provable, q must not be a fact. In addition, we need to establish that every strict rule with head q is *known to be* inapplicable. Thus in such rules there must at least one literal l in the antecedent for which we have established that l is not definitely provable ($-\Delta l$).

It is worth noticing that this definition of nonprovability does not involve loop detection. Thus if D consists of the single rule $p \rightarrow p$, we can see that p cannot be proven, but DL is unable to prove $-\Delta p$.

Defeasible provability requires considering the chain of reasoning for the complementary literal and possible resolution using superiority relation. Thus the inference rules for defeasible provability are bit more complicated than that of definite provability.

$+\partial$) If $P(n+1) = +\partial q$ then either
(1) $+\Delta p \in P(1..n)$; or
(2) (2.1) $\exists r \in R_{sd}[q]\ \forall a \in A(r), +\partial a \in P(1..n)$, and
(2.2) $-\Delta{\sim}q \in P(1..n)$, and
(2.3) $\forall s \in R[{\sim}q]$ either
(2.3.1) $\exists a \in A(s), -\partial a \in P(1..n)$; or
(2.3.2) $\exists t \in R_{sd}[q]$ such that
(2.3.2.1) $\forall a \in A(t), +\partial a \in P(1..n)$, and
(2.3.2.2) $t > s$.

[1] $P(1..n)$ denotes the initial part of the sequence of length n.

To show that q is defeasibly provable we have two choices: (1) we show that q is already definitely provable; or (2) we need to argue using the defeasible part of D as well. In particular, there must be a strict or defeasible rule with head q which can be applied (2.1). But now we need to consider possible "attack", i.e., reasoning chains in support of $\sim q$. To be more specific: to prove q defeasibly provable we must show that $\sim q$ is not definitely provable (2.2). Also, (2.3) we must consider the set of all rules which are not known to be inapplicable and which have head $\sim q$. Essentially, each such rule s attacks the conclusion q. For q to be provable, each such rule s must be counterattacked by a rule t with head q with the following properties: (i) t must be applicable at this point, and (ii) t must be stronger than s. Thus each attack on the conclusion q must be counterattacked by a stronger rule.

$-\partial$) If $P(n+1) = -\partial q$ then
(1) $-\Delta q \in P(1..n)$, and
(2) (2.1) $\forall r \in R_{sd}[q]\ \exists a \in A(r), -\partial a \in P(1..n)$; or
(2.2) $+\Delta \sim q \in P(1..n)$; or
(2.3) $\exists s \in R[\sim q]$ such that
(2.3.1) $\forall a \in A(s), +\partial a \in P(1..n)$, and
(2.3.2) $\forall t \in R_{sd}[q]$ either
(2.3.2.1) $\exists a \in A(t), -\partial a \in P(1..n)$; or
(2.3.2.2) not$(t > s)$.

Lastly, to prove that q is not defeasibly provable, we must first establish that it is not definitely provable. Then we must establish that it cannot be proven using defeasible part of the theory. There are three possibilities to achieve this: either (2.1) we have established that none of the (strict and defeasible) rules with head q can be applied; or (2.2) $\sim q$ is definitely provable; or there must be an applicable rule s with head $\sim q$ such that no applicable rule t with head q is superior to s.

A full definition of the proof theory can be found in [2]. 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. 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.,

$$\frac{\vdash \phi}{\vdash \Box\phi}\ \textbf{Necessitation} \qquad \frac{\vdash \phi \supset \psi}{\vdash \Box\phi \supset \Box\psi}\ \textbf{RM}$$

Both dictate conditions to introduce modalities purely based on the derivability and structure of the antecedent. These rules are related to the problem of logical omniscience [12, 17]: if $\Box$ corresponds to either INT (Intention), BEL (Believe), OBL (Obligation), they put unrealistic assumptions on the capability of an agent. However, if we take a constructive interpretation, i.e., if an agent can build a derivation of ψ then she can build a derivation of $\Box\psi$.

In the light of this, [9] has proposed an extension of DL to capture combinations of mental attributes and deontic concept, as well as multiple (defeasible) consequence relations. In the paper, the authors have proposed a framework that replace the derivability in classical logic with a practical and feasible notion like derivability in DL. Thus, the intuition behind this work is that we are allowed to derive $\Box_i p$ if we can prove p with the mode $\Box_i$ in DL.

The details of the framework are as follow. To extend DL with modal operators we have two options: (1) to use the same inferential mechanism as basic DL and to represent explicitly the modal operators in the conclusion of rules [20]; (2) introduce new types of rules for the modal operators to differentiate between modal and factual rules.

For example, the "deontic" statement "The Purchaser shall follow the Supplier price lists" can be represented as

$$AdvertisedPrice(X) \Rightarrow O_{purchaser} Pay(X)$$

if we follow the first option and

$$AdvertisedPrice(X) \Rightarrow_{O_{purchaser}} Pay(X)$$

according to the second option, where $\Rightarrow_{O_{purchaser}}$ denotes a new type of defeasible rule relative to the modal operator $O_{purchaser}$. Here, $O_{purchaser}$ is the deontic "obligation" operator parametrized to an actor/role/agent, in this case the purchaser.

As stated in [9], the differences between the two approaches, besides the fact that in the first approach there is only one type of rules while the second accounts for factual and modal rules, are two: (i) the first approach has to introduce the definition of p-incompatible literals (i.e., a set of literals that cannot be hold when p holds.) for every literal p. For example, we can have a modal logic where $\Box p$ and $\neg p$ cannot both be provable at the same time; (ii) the first approach is less flexible than the second: in particular in some situations it must account for rules to derive $\Diamond p$ from $\Box p$; similarly *conversions*, which permit to use a rule for a certain modality as it were for another modality, require additional operational rules in a theory. Hence, the second approach seems to offer a more conceptual tool than the first one.

It seems that the second approach can use different proof conditions based on the modal rules to offer a more fine grained control over the modal operators, which allows us to modal interaction over operators.

The language of Modal Defeasible Logic (MDL) consists of a finite set of modal operators $Mod = \{\Box_1, \ldots, \Box_n\}$ and a (numerable) set of atomic propositions $Prop = \{p, q, \ldots\}$[2]. Besides, the definition of literal (an atomic proposition or the negation of it) is supplemented with the following clause:

- If l is a literal then $\Box_i l$, and $\neg\Box_i l$, are literals if l is different from $\Box_i m$, and $\neg\Box_i m$, for some literal m.

which prevents literals from having sequences of modalities where successive occurrences of one and the same modality; however, iterations like $\Box_i\Box_j$ and $\Box_i\Box_j\Box_i$ are legal in the language.

Accordingly, a Modal Defeasible Theory is defined as follow.

Definition 1.1. *A Modal Defeasible Theory D is a structure $(F, R, \succ)$ where*

- *F is a set of facts (literals or modal literals),*
- *$R = R^B \cup \bigcup_{1 \leq i \leq n} R^{\Box_i}$, where R^B is the set of base (un-modalised) rules, and each $R^{\Box_i}$ is the set of rules for $\Box_i$ and*

[2]The language can be extended to deal with other notions. For example to model agents, we have to include a (finite) set of agents, and then the modal operators can be parameterised with the agents. For a logic of action or planning, it might be appropriate to add a set of atomic actions/plans, and so on depending on the intended applications.

- $\succ \subseteq R \times R$ *is the superiority relation.*

A rule $r \in R$ is an expression $A(r) \hookrightarrow_X C(r)$ such that ($\hookrightarrow \in \{\rightarrow, \Rightarrow, \rightsquigarrow\}$, X is B, for a base rule, and a modal operator otherwise), $A(r)$ the antecedent or body of r is a (possible empty) set of literals and modal literals, and $C(r)$, the consequent or head of r is a literal if r is a base rule and either a literal or a modal literal Yl where Y is a modal operator different from X. As in the sections before, given a set of rules R, we use R_s, R_d, R_{sd} to denote the set of strict rules, defeasible rules, and strict and defeasible rules respectively; and $R[q]$ to deonte the set of rules in R whose head is q.

The derivation tags are now indexed with modal operators. Let X range over Mod. A conclusion can now have the following forms:

$+\Delta_X q$**:** q is definitely provable with mode X in D (i.e., using only facts and strict rules of mode X).

$-\Delta_X q$**:** q is definitely rejected with mode X in D.

$+\partial_X q$**:** q is defeasibly provable with mode X in D.

$-\partial_X q$**:** q is defeasibly rejected with mode X in D.

That is, if we can prove $+\partial_{\Box_i} q$, then we can assert $\Box_i q$.

Similarly to standard DL, the provability of MDL is based on the concept of *derivation* in D, which corresponds to a finite sequence $P = (P(1), \ldots, P(n))$ of tagged (modal) literals satisfying the proof conditions. $P(1..n)$ denotes the initial part of the sequence P of length n.

Before introducing the proof conditions for the proof tags for modal X, we start with some auxiliary notions relevant to this thesis.

Definition 1.2. *[21] Let # be either Δ or ∂. Given a proof $P = (P(1), \ldots, P(n))$ in D, a (modal) literal q is Δ-provable in P, or simply Δ-provable, if there is a line $P(m)$ of the derivation such that either:*

1. *if $q = l$ then*
   - *$P(m) = +\#l$ or*
   - *$\Box_i l$ is #-provable in $P(1..m-1)$ and $\Box_i$ is reflexive*[3]
2. *if $q = \Box_i l$ then*
   - *$P(m) = +\#_i l$ or*
   - *$\Box_j \Box_i l$ is #-provable in $P(1..m-1)$, for some $j \neq i$ such that $\Box_j$ is reflexive.*
3. *if $q = \neg\Box_i l$ then*
   - *$P(m) = -\#_i l$ or*
   - *$\Box_j \neg\Box_i l$ is #-provable in $P(1..m-1)$, for some $j \neq i$ such that $\Box_j$ is reflexive.*

In a similar way, a literal defines to be Δ- and ∂-rejected by taking, respectively, the definition of Δ-provable and ∂-provable and changing all positive proof tags into negative proof tags, adding a negation in front of the literal when the literal is prefixed by a modal operator $\Box_j$, and replacing all the *or*s by *and*s. Thus, for example, a literal $\Box_i l$ is ∂-rejected if, in a derivation, a line $-\partial_i l$, and the literal $\neg\Box_i \neg l$ is ∂-rejected if we have $+\partial_i \neg l$ and so on.

Definition 1.3. *Let X be a modal operator and # is either Δ or ∂.*

[3] A modal operator $\Box_i$ is reflexive iff the truth of $\Box_i \phi$ implies the truth of ϕ. In other words $\Box_i$ is reflexive when we have the modal axiom $\Box_i \phi \rightarrow \phi$.

- *A literal l is $\#_X$-*provable *if the modal literal Xl is $\#$-provable;*

- *A literal l is $\#_X$-rejected if the literal Xl is $\#$-rejected.*

Based on the above definition of provable and rejected literals the authors give the conditions to determine whether a rule is applicable or the rule cannot be used to derive a conclusion (i.e., the rule is discarded).

The proof conditions for $+\Delta$ correspond to monotonic forward chaining of derivations. For a literal q to be definitely provable with the mode $\Box$, a strict rule for $\Box$ with head q whose antecedents have all been definitely provable previously. And to establish that q cannot be definitely proven we must establish that for every strict rule for $\Box$ with head q there is at least one antecedent which has been shown to be non-provable. Besides, a rule for Y can be used as a rule for a different modal operator X in case all literals in the body of the rule are modalised with the modal operator we want to prove. For example, given the rule:

$$p, q \rightarrow_{\text{BEL}} s$$

we can derive $+\Delta_{\text{INT}} s$ if we have $+\Delta_{\text{INT}} p, +\Delta_{\text{INT}} q$ and the conversion $Convert(\text{BEL}, \text{INT})$ holds in the theory.

Conditions for $\partial_\Box$ are more complicated. A rule for a belief is applicable if all the literals in the antecedent of the rule are provable with the appropriate modalities; while the rule is discarded if at least one of the literals in the antecedent is not provable. As before, for other types of rules we have to take conversions into account. We have thus to dertermine conditions under which a rule for Y can be used to directly derive a literal q modalised by X. Roughly, the condition is that all the antecedents a of the rule are such that $+\partial_\Box a$.

Definition 1.4. *Let X be a modal operator or B. A rule $r \in R$ is ∂_X-*applicable *iff*

1. *$r \in R^X$ and $\forall a_k \in A(r)$, a_k is ∂-provable; or*

2. *if $X \neq B$ and $r \in R^B$, i.e., r is a base rule, then $\forall a_k$, a_k is ∂_X-provable.*

Definition 1.5. *Let X be a modal operator or B. A rule $r \in R$ is ∂_X-*discarded *iff*

1. *$r \in R^X$ and $\exists a_k \in A(r)$, a_k is ∂-rejected; or*

2. *if $X \neq B$ and $r \in R^B$, i.e., r is a base rule, then $\exists a_k$, a_k is ∂_X-rejected.*

As a corollary of the above definitions, the following definition regarding the support of a literal is established.

Definition 1.6. *Given a theory D, a literal q is supported in D iff there exists a rule $r \in R^X[q]$ s.t. r is applicable, otherwise q is not supported. For X belongs to the set of modal operator in D, we use $+\Sigma_X q$ and $-\Sigma_X q$ to indicate that q is supported by rules for X.*

Defeasible derivations have an argumentation like structure which is divided into three phases. In the first phase, we have to prove that the conclusion is supported. Then in the second phase, we have to consider all possible (actual and not) reasons against the desired conclusion. Finally, in the last phase, which can be done in two ways: (i) by showing that some of the premises of a counterargument do not hold, or (ii) by showing that the argument is weaker than an an argument in favour of the conclusion. This is formalised by the following (constructive) proof conditions.

$+\partial_X$: If $P(n+1) = +\partial_X q$ then
(1) $+\Delta_X q \in P(1..n)$; or
(2) (2.1) $-\Delta_X \sim q \in P(1..n)$, and
(2.2) $\exists r \in R_{sd}[q]$: r is ∂_X-applicable, and
(2.3) $\forall s \in R[\sim q]$ either s is ∂_X-discarded; or
(2.3.1) $\exists w \in R[q]$: w is ∂_X-applicable, and
(2.3.2) $w > s$

$-\partial_X$: If $P(n+1) = -\partial_X q$ then
(1) $-\Delta_X q \in P(1..n)$ and either
(2) (2.1) $+\Delta_X \sim q \in P(1..n)$; or
(2.2) $\forall r \in R_{sd}[q]$, either r is discarded; or
(2.3) $\exists s \in R[\sim q] : s$ is applicable, and
(2.3.1) $\forall w \in R[q]$ either w is discarded; or
(2.3.2) not$(w > s)$; or
$w \in R^Z$, $s \in R^{Z'}$, $Z \neq Z'$

The above condition is, essentially, the usual condition for defeasible derivations in DL, we refer the reader to [9] for more through treatments. The only point we want to highlight here is that base rules can play the role of modal rules when all the literals in the body are $\partial_{\Box_i}$-derivable. Thus, from a base rule $a, b \Rightarrow_B c$ we can derive $+\partial_{\Box_i} c$ if both $+\partial_{\Box_i} a$ and $+\partial_{\Box_i} b$ are derivable while this is not possible using the rule $a, \Box_i b \Rightarrow_B c$.

On the top of these, SPINdle also supports modal operators interactions and conclusions reparation with preference operator over defeasible rules. Readers interested in this topic please refer to [9] and [6, 7] for details, respectively.

1.1.3 Ambiguity blocking and Ambiguity propagation

The treatment of ambiguity is one of the core of non-monotonic reasoning where intuitions can clash [13, 22]. Intuitively, 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.

In an ambiguity blocking setting, given the sceptical nature of the reasoning, the two conclusions are considered both as not provable, and we ignore the reasons why they were when we use them as premises of further arguments; whereas in an ambiguity propagation setting, since we were not able to solve the conflict we want to propagate the uncertainty to conclusions depending on the ambiguous literals.

Let us illustrate the distinction between ambiguity blocking and ambiguity propagation with the help of the following example.

Example 1.1 (Presumption of Innocence)**.** The presumption of innocence is the principle that one is considered innocent until proven guilty. Suppose that a piece of evidence A suggests that the defendant in a legal case is not responsible while a second piece of evidence B indicates that he/she is responsible; and the sources are equally reliable. According to the underlying legal system a defendant is presumed innocent (i.e., not guilty) unless responsibility has been proved (without any reasonable doubt).

The above scenario is encoded in the following defeasible theory:

$$
\begin{array}{lrl}
r_1: & evidence_A & \Rightarrow -responsible \\
r_2: & evidence_B & \Rightarrow responsible \\
r_3: & responsible & \Rightarrow guilty \\
r_4: & & \Rightarrow -guilty \\
r_5: & -guilty & \Rightarrow compensation
\end{array}
$$

and two facts:

$$evidence_A \qquad evidence_B$$

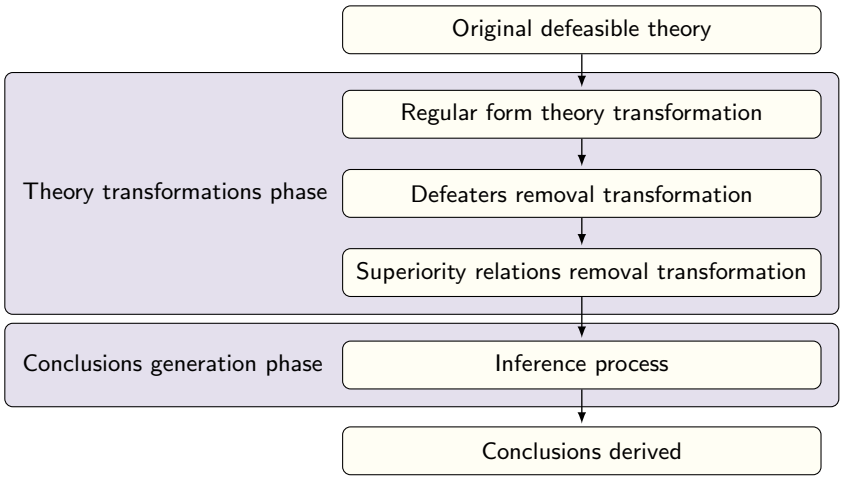

Figure 1.1: Defeasible Theory Inference process

Given both evidences A and B, ambiguity exists as it is unclear to us on whether the defendant should hold any responsibility to the case or not since there are two applicable rules (r_1 and r_2) with the same strength, each supporting the negation of the other. As a consequence, r_3 is not applicable, and so there is no applicable rule supporting the *guilty* verdict. Thus, under this situation, with the underlying legal system, we should conclude that the defendant is not guilty.

However, if we allow propagation of the ambiguity of responsibility to the next level of the reasoning chain, then the defendant's guiltiness becomes ambiguous; hence an undisputed conclusion cannot be drawn.

A preference for ambiguity blocking or ambiguity propagating behaviour is one of the properties of non-monotonic inheritance nets over which intuitions can clash [23]. Stein [22] argues that the ambiguity blocking results in unnatural pattern of conclusions in extensions of the above example. Ambiguity propagation results in fewer conclusions being drawn, which might make it preferable when the cost of an incorrect conclusion is high.

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

1.2 Inference Process

The inference process of a defeasible theory is divided into two phases, namely: (i) *theory transformations phase*, (ii) *conclusions generations phase* , as shown in Figure 1.1. The theory transformations phase is a *pre-processing phase* which transform a defeasible theory to an equivalent theory without superiority relations (version 1.X.X only) and defeaters using the techniques described in [2]. Then, the *conclusions generation* phase will: (1) assert whether a literal is provable (and the strength of its derivation); and (2) progressively reduce and simplify the theory.

In the light of this, SPINdle possess two groups of reasoning components which provides the theory transformations and reasoning capabilities as required. The *theory normalizers* implements various transformation algorithms that transform a defeasible theory to a form that can be processed by the associated inference engines; while the *inference engines* implement the defeasible reasoning mechanisms based on different reasoning approaches.

After a defeasible theory has been loaded into system, SPINdle will first determine the types of theory normalizer and inference engines to be used based on the elements that appear in the theory; or an exception will be thrown if no suitable theory normalizer or inference engine is available.

Next, SPINdle apply different types of transformations as of required by the associated inference approaches, or as are directed by the users.

Then, SPINdle will identify the constraints in the theory that cannot be honored by the inferencing engines by checking the elements in the theory (e.g., whether the theory contains any facts, defeaters, etc.) and send the candidate constraints to the user, if any.

Finally, SPINdle will inference on the theory, compute the conclusions and send it back to the users. In general, the inference engine 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.

However, 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.

Note that version 1.X.X of SPINdle implements the inference algorithm proposed in [18] while version 2.X.X implements the inference algorithm proposed by us [16]. The major different of the two approaches is that: in the former approach, the *support* of a literal due to superiority relation will not be removed even when the superior rule is applicable, which subsequently cause problems under ambiguity propagation variant of DL. However, in the latter approach, as superiority relations are still in place during the inference process, all representational properties of DL are preserved which thus resolved the problem that arise due to superiority relations.

1.3 Different SPINdle Front-Ends

Currently, there is a host of different front-ends to SPINdle.

1. `spindle.Reasoner` is the main reasoner class of SPINdle. It provides different methods for transforming defeasible theories and various reasoning engines to cater the needs of different reasoning scenarios.
2. `spindle.sys.Conf` provides configuration methods to set up the reasoning context.
3. `spindle.core.dom.Theory` provides the data structure for storing and/or manipulating defeasible theory in memory.
4. `spindle.io.TheoryParser` and `spindle.io.TheoryOutputter` is the API used for theory (and conclusions) parsing and theory (and conclusions) output. New parser and outputter can be registered by modifying the content of `spindle.resources.io_conf.xml` file, or can be searched by the SPINdle I/O manager automatically by setting the option `reasoner.io.searchClasses` to `true`.

5. `spindle.io.IOManager` is the I/O manager of SPINdle for selecting appropriate theory parsers and outputters for theories and conclusions written in different formalisms.

6. SPINdle theory editor is a Java based application for editing defeasible theory, and is available at:

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

Chapter 2

Installation and Invoking SPINdle

To get up and running with the binary edition of SPINdle, follow these steps:

- Make sure you have a Java environment installed. For the current version of SPINdle, the use of JRE runtime version 1.6 or later of is strongly recommended. However, if you want to compile the source code, a JDK version 1.6 is required.
- Download SPINdle. The latest stable version of SPINdle is available from the following URL:

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

  SPINdle is distributed as a `zip` compressed file. Depending on the preferences, user can choose to download the source, binary, or both source and binary edition of SPINdle.
- Uncompress the downloaded file into a directory.

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

`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.

  The SPINdle package comes with an Apache Ant's build file that you can use for building and running, or you can copy the `src` files into your preferred IDE.

2.1 Build SPINdle from source

Table 2.1 and table 2.2 show the tasks available in the build file and the libraries required for building SPINdle from the source, respectively. For example, to compile the source, type:

```
ant compile
```

2.2 Invoking SPINdle

The common line synopsis of SPINdle is as follows:

Task name	Description
`compile`	Compile the source code in `src` directory.
`javadoc`	Generate the JavaDoc API document.
`dist`	Compile the source code and generate the Java archive (`.jar`) file.
`run`	run SPINdle.
`clean`	Clean current folder.

Table 2.1: SPINdle build file tasks

Library name	Needed for	Available at
JDOM	Parsing defeasible theories represented using XML. (***Deprecated:*** *As of version 1.0.4, DOM and StAX are used in preference to JDOM in handling all XML documents operations. Since version 2.2.4, JAXB is used instead.*)	`http://java.jdom.org`

Table 2.2: SPINdle library dependencies

Usage: `java -jar spindle_<version>.jar`
`[--version]`
`[--license]`
`[--console]`
`[--log.level` *level*`]`
`[--app.showProgress = true | false]`
`[--app.showResult = true | false]`
`[--app.showStatistics = true | false]`
`[--app.progress.timeInterval = number]`
`[--app.io.searchClasses = true | false]`
`[--reasoner.multiThreadMode = true | false]`
`[--app.saveResult = true | false]`
`[--app.result.folder = string]`
`[--reasoner.version = number]`
`[--reasoner.logInference= true | false]`
`[--reasoner.ambiguityPropagation = true | false]`
`[--reasoner.wellFoundedSemantics = true | false]`
$[file_1 | dir_1 | URL_1]\ [file_2 | dir_2 | URL_2] \ldots$

The meaning of the options will be detailed in Section 2.2.1. To run and test the SPINdle distribution using build file task, type:

```
ant run [file1|dir1|URL1] [file2|dir2|URL2] ...
```

or after creating the Java archive file using the `dist` task, in the command prompt, type:

```
java -jar spindle_<version>.jar [--options] [file1|dir1|URL1]
                                              [file2|dir2|URL2] ...
```

Note that, in the distribution, some sample defeasible theories are available in the `samples` folder that can be used for testing.

By default, DFL format will be used as output format. However, it is also possible to set the output using the XML language supported by SPINdle[1].

2.2.1 Common-line options

By default, SPINdle shows all progress information and results computed to the standard output. However, it is possible to change its behaviors by passing different options to SPINdle. The following are the set of options available.

--version

Show the current SPINdle release version.

--license

Show the SPINdle license.

--console

Start the console application interface.

--log.level

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

--app.progress.timeInterval

Set the time interval for showing progress information.

--app.showProgress

Set whether to show progress information to the standard output (Default: `false`).

--app.showResult

Set whether to show the computed results to the standard output (Default: `true`).

--app.showStatistics

Set whether to show the computational statistics to the standard output (Default: `true`).

--app.io.searchClasses

Search for (new) I/O classes available on the classpath (Default: `false`).

--reasoner.multiThreadMode

Set to true if multiple instances of SPINdle reasoner are running on the same JVM at the same time (Default: `false`).

--app.saveResult

Save computed results to a file (Default: `true`).

--app.result.folder

Set the folder for storing the computed results.

--reasoner.version

Select the reasoner version to be used (1 or 2) (Default: 2) (cf. Section 1.2).

--reasoner.logInference

Set to `true` for logging rule inference status (*applicable*, *discarded* and *defeated*) while reasoning (Default: `false`).

--reasoner.ambiguityPropagation

Set to `true` when reasoning with ambiguity propagation (Default: `false`).

--reasoner.wellFoundedSemantics

Set to `true` when reasoning with well-founded semantics (Default: `false`).

[1]The XML schema supported by SPINdle can be found in Appendix A.

2.2.2 Console application interface

As of version 2.0.0, a console application was implemented to support efficient defeasible theory test. The command line synopsis of the console application is as follows.

```
java -jar spindle_<version_no>.jar --console [file]
```

and the following are the list of commands available in the console application:

Command	Description
`add`	add a new fact, rule, superiority relation and mode rule to the theory.
`clear`	clear the theory and the conclusions generated.
`conclusions`	generate the conclusions from theory.
`load`	load a theory.
`remove`	remove a fact, rule, superiority relation and mode rule from the theory.
`save`	save the theory or conclusions derived.
`set`	set console application environment property.
`show`	display the theory or conclusions derived.
`transform`	transform the theory to regular form, or to an equivalent theory without superiority relations or defeaters.
`history`	show the last 50 commands executed.
`help`	show the command menu.
`quit`	quit the console application.

Table 2.3: Console application commands.

Note that unlike invoking SPINdle as a Java application, which will read and compute all files if the input is a directory, in the console application, only the first theory file in the directory will be loaded.

2.3 Theory file extensions

SPINdle accepts theories to be described using different formalisms. The I/O manger (`spindle.io.IOManager`, as described before, will associate different file types with their respective theory parsers using their file name extensions. The current supported formalism includes: XML and DFL, and will be discussed later in Chapter 3. However, its definition should be self contained and easy to understand. Readers having difficulties 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 a 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).

2.4 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 perfor-

mance 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.

2.5 System limits

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.

SPINdle accepts defeasible theories written in any format provided that an associated theory parser is available in the SPINdle classpath. This chapter describes the SPINdle `DFL` theory language for theory modeling. The XML formalism using in SPINdle share the same set of components but represented in a different way. Users interested in this please refer to the XML schema in Appendix A. The same XML schema is also available in `<SPINdle_HOME>/src/spindleDefeasibleTheory2.xsd`.

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

** Note however that the current SPINdle implementation does not include the grounding process. All terms included in the parenthesis are treated as an instance value and will not be grounded.

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.: $-a$.

Literal variable	Predicate(s)	Description
@DATE	YYYY,MM,DD or YYYY,MM,DD,HH,MM,SS	A specific date.
@DURATION	aYbMcDdHeMfS[1]	A specific duration of time.
@NOW	-	Current date time.
@TODAY	-	Date of today.
@VAL	*number*	A numerical value.

Table 3.1: System literal variables

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

3.2.4 Literal Variable and Literal Boolean Function

As of version 2.1.0, SPINdle support the use of user defined variables (called *literal variable*) and boolean expression (called *literal boolean function*) in all rules.

A literal variable is a literal with a name started using a "@" sign. It can be used as a regular literal (basic literal or modalised literal) to represent knowledge in the body literals of all types of rules. Table 3.1 shows the list of system supported literal variables.

In a defeasible theory, a literal variable can be set using the following syntax:

$$\texttt{set } variableName \texttt{ = } variableValue$$

Example 3.1. The following syntax shows the syntax of setting a literal variable named @createDate to the date of today:

```
set @createDate = @TODAY
```

Literal boolean function, on the other hand, is a boolean expression that is started and ended with a "$" sign. It can be used as a value of a literal variable and the SPINdle reasoning engine will evaluate its truth value before performing any transformations or the inference process.

Example 3.1 (continuing from p. 20)**.** The following example shows how literal boolean function can be used to set the value of a literal variable.

```
set @isExpired = $@TODAY - @createDate >= @DURATION(3Y)$
```

The above example says that the literal variable @isExpired is *true* if it is 3 years after the create date; and *false* otherwise.

** Note that literal variables and literal boolean function should not appear in the head of a rule, or an exception will throw.

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

Table 3.2: Rule Types-Symbols association

3.3 Facts, Rules and Defeaters

A basic rule is of the form:

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

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 3.2).

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.

Besides, it is also important to note that both facts, strict rules and defeaters can have only **ONE** head literal and defeasible rule is the only rule type that can have multiple literals in its head. 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 Mode Conversions and Mode Conflicts

Mode conversion and mode conflict can be achieved using the symbols depicted in Table 3.3. The implementation is based on the extension proposed in [9], which has been discussed in section 1.1.2.

Type	Symbol
Mode Conversion	==
Mode Conflict	!=

Table 3.3: Mode Conversion and Mode Conflicts symbols

3.6 Conclusions Set

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

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

-D q This means that we have proved that q is reject definitely in D.

+d q This means that q is defeasibly provable in D.

-d q This means that we have proved that q is rejected defeasibly in D.

3.7 A practical example

The following is an example of a defeasible theory represented using the DFL language.

```
db20: =>-copyright(database)
db21: contentType(database),$(@dateToVerify-@creationDate) < @DURATION(15y)$ => copyright(database)
db22: contentType(database),$@dateToVerify-@lastModifiedDate < @DURATION(15y)$ => copyright(database)
db21>db20
db22>db20

gr01: => publicDomain
 r91: copyright(database)=>-publicDomain
r91>gr01
```

The theory states that a database is in public domain if it is 15 years after its creation date, or 15 years after its last modified date.

Then, if the following facts are available:

```
set @creationDate=@date(2011,12,1)
set @dateToVerify=@today
>> contentType(database)
```

we have the following conclusions (assuming that today is 11 May 2012):

```
+D contentType(database)      +d contentType(database)
-D copyright(database)        +d copyright(database)
-D -copyright(database)       -d -copyright(database)
-D publicDomain               -d publicDomain
-D -publicDomain              +d -publicDomain
```

This is due to the fact that the literal boolean function in rule `db21` is *true*. The conclusion of `db21` override the conclusion of `db20`, and subsequently proven $+\partial copyright$. By the same token, since $copyright(database)$ is provable, the conclusion of `r91` override the conclusion of `gr01`, and subsequently proven $+\partial \neg publicDomain$.

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. In the first part of this chapter, we will look at the way to embed SPINdle into a Java application. It is a work flow compliance checker which determines whether ths current task is compliance with the its constraints, and report to the user/application if there is any problem. The material in this chapter is easier to understand if you are familiar with SPINdle Java API.

At the end of this chapter, we will talk about some general considerations that are useful when planning an application that embeds the SPINdle defeasible logic engine.

4.1 Motivation

Imagine that you are implementing a business compliance checker for a business process in your company. The engine is supposed to look at the flow of tasks, together with the constraints that each tasks imposed, and then selects the appropriate task as the next task to work on, or reports the error to users. Imagine further that you have coded this up in a tradition Java class.

After a few weeks/months of work, some of the government regulations are changed. It is the responsibilities of your company to be compliance with the new government regulations which may involve certain modifications to the company's work flow. If you have been using the traditional programming techniques, your code is a gnarled mess as it has to do all sorts of constraints queries testing for each tasks separately.

However, if your had written this using a rule engine, like SPINdle, you have a nice clean code as the government regulations can be encoded using rules. If later on a new constraint is added to a task, it would be easier to find and add/modify it.

These kinds of systems have to take into account heterogeneous contexts in order to behave properly and effectively, but also dynamically reconfigurable whenever it is necessary. As a consequence, these systems do not only implement complex functionalities, but also have to embed a reasoning engine that can monitor the context in which system runs, take decision and report errors.

4.2 Doing it with SPINdle

To embed SPINdle into a Java application we have two components: a Java application component and a SPINdle language component. The Java code needs to instantiate an instance of the SPINdle defeasible reasoner, load in the set of rules (as the knowledge base of the application), then load in the contextual information and inference on the combined theory. This one instance of SPINdle reasoning engine can be reused to process each task. The `ProcessComplianceChecker` class package all this up.

Listing 4.1: Code snippet for instantiating the SPINdle reasoner and loading a defeasible theory from file

```
1  import java.io.File;
2
3  import spindle.Reasoner;
4  import spindle.core.dom.Theory;
5  import spindle.io.IOManager;
6  import spindle.sys.AppLogger;
7
8  public class ProcessComplianceChecker {
9      // SPINdle defeasible reasoner
10     private Reasoner reasoner = null;
11
12     // application knowledge base
13     private Theory knowledgeBase = null;
14
15     public ProcessComplianceChecker(File knowledgeBaseFile,
16             AppLogger logger) throws ParserException, IOException {
17         // create a SPINdle defeasible reasoner
18         reasoner = new Reasoner();
19
20         // load the application knowledge base (a defeasible theory)
21         // from a file using the I/O Manager
22         knowledgeBase = IOManager.getTheory(knowledgeBaseFile, logger);
23     }
24 }
```

As described before in Section 1.3, the `spindle.core.dom.Theory` class provides the data structure for storing and/or manipulating defeasible theory in memory. The call to SPINdle I/O Manager (`spindle.io.IOManager.getTheory`) will find the knowledge base not only in the current directory but even if it is located somewhere that can be located through an URL, including the case when it is packaged in a `jar` file. The second argument of I/O Manager is an application logger (`spindle.sys.AppLogger`) which can be used to log the information when loading a theory.

Listing 4.1 shows the code snippet for instantiating the SPINdle reasoner and loading a defeasible theory into the application using the I/O Manager. However, in some situations, we may decide to load the defeasible theory to the reasoner directly, which can be done using the following code:

```
reasoner.loadTheory(theoryFile);
```

and we can retrieval the theory currently saved in the reasoner using the following code:

```
knowledgeBase = reasoner.getTheory();
```

Then whenever we want to perform a compliance checking with the available information from the current/user specified task, the application need to do four things: (1) duplicate the set of knowledge base from the application, (2) add the context information to the duplicated theory, (3) execute the combined defeasible theory, and (4) compute the conclusions and return the result back to the application. Listing 4.2 shows how this can be achieved.

Listing 4.2: Code snippet for reasoning with theory duplication

```
public Map<Literal, Map<ConclusionType, Conclusion>> checkCompliance(
        Theory contextualInfo) throws TheoryException, ReasonerException {
    // duplicate a copy of the knowledge base
    Theory workingTheory = knowledgeBase.clone();

    // combine the contextual information with the knowledge base
    workingTheory.add("contextInfo", contextualInfo);

    // load theory to the reasoner
    reasoner.loadTheory(workingTheory);

    // generate the conclusions with theory transformations
    return reasoner.generateConclusionsWithTransformations();
}
```

Note that in the snippet above `spindle.Reasoner.generateConclusionsWithTransformations` method is a wrapper method of performing the whole defeasible inference process (as shown in Figure 1.1). However, depending on the applications, users/applications may choose to perform the transformations manually. Belows are the methods supported by the `spindle.Reasoner` class.

```
// transform theory to regular form
spindle.Reasoner.transformTheoryToRegularForm();

// transform theory to an equivalent theory without defeater
spindle.Reasoner.removeDefeater();

// transform theory to an equivalent theory without superiority relation
spindle.Reasoner.removeSuperiority();
```

However, it is important to note that the last method of eliminating superiority relations from a defeasible theory (`spindle.Reasoner.removeSuperiority()`) may cause problems when reasoning with ambiguity propagation using Maher's approach (reasoning engines version 1.X.X) as some of the representational properties of DL are removed [16].

4.2.1 Creating a Literal

Creating a literal is the most frequent job when parsing a theory. It can also be used to retrieve the result of a particular literal from the returned conclusions set. Instead of creating a literal using the constructors in `spindle.core.dom.literal` class according to different cases, users may consider calling the static methods in the `spindle.core.dom.DomUtilities` class, which can help in simplifying the task. Listing 4.3 shows the code of creating a literal programmatically using the second method and use it to retrieve the required result to the application.

Listing 4.3: Code snippet for creating a literal programmatically

```
public Map<ConclusionType, Conclusion> getLiteralConclusion(
        String literalName, boolean isNegation,
        Map<Literal, Map<ConclusionType, Conclusion>> conclusions) {
    // generate the literal on the fly
    Literal literal = DomUtilities.getLiteral(literalName, isNegation);

    // retrieve the result from the conclusions set
    return conclusions.get(literal);
}
```

Note that there are also other methods available in the `spindle.core.dom.DomUtilities` class that can be used to create a literal. Listing 4.4 shows some of them.

Listing 4.4: Methods provided in `spindle.core.dom.DomUtilities` class for creating a literal

```
1  public static Literal getLiteral(final Literal literal);
2
3  public static Literal getLiteral(final String name,
4      final boolean isNegation);
5
6  public static Literal getLiteral(final String name, final boolean isNegation,
7      final Mode mode,
8      final String[] predicates);
9
10 public static Literal getLiteral(final String name, final boolean isNegation,
11     final String modeName, final boolean isModeNegation,
12     final String[] predicates);
```

Here, `name` is the *name* of the literal, `isNegation` is the sign of the literal (`true` if the literal is negated; `false` otherwise). `mode` is the modal operator of the literal and can be instantiated using `spindle.core.dom.Mode` class, or created by providing the `modeName` and `isModeNegation`. `predicates` is the predicates that appear in the literal. Note however that, currently, we are still working on the theory grounding process of SPINdle and *no* variables substitution will be performed before or after the inference process. The inference process will be carried out as if every literals is grounded.

4.2.2 Creating Defeasible Theory from String

All that's left is how can we generate the contextual theory from the set of contextual information. The simplest way is to formalize the set of contextual information using one of the language supported by SPINdle, and then generate the defeasible theory by parsing the string.

So, assuming that the set of contextual information is represented using the `DFL` (or `XML`, respectively) language supported by SPINdle and is stored in an array of String, as show below:

```
String[] contextualInfoStr = new String[] { ">> constraint1",
                                            ">> constraint2" };
```

We can use the static methods provided by `spindle.io.parser.DflTheoryParser2` (or `spindle.io.parser.XmlTheoryParser`, respectively) class to complete the task. Listing 4.5 shows the code required.

Listing 4.5: Code snippet for generating a defeasible theory from string

```
1 protected static Theory generateTheory(String[] contextualInfoStr,
2             AppLogger logger) throws ParserException {
3     StringBuilder sb = new StringBuilder();
4     for (String rule : contextualInfoStr) {
5         sb.append(rule).append("\n");
6     }
7     return DflTheoryParser2.getTheory(sb.toString(), logger);
8 }
```

It is assumed that each rules in the string are separated by a new line character. The `spindle.io.parser.DflTheoryParser2.getTheory` method will then read in the string and generate the defeasible theory on the fly. This method is the actual underlying method that the SPINdle I/O Manager using. The different is that the I/O Manager will decide which theory parser should be used based on the input file extensions. However, in this case, since we are assuming that the contextual information is to be written using the SPINdle DFL language, we can then use this method directly. Besides, there is also no way for the I/O Manager to decide the represent format of the input string. So, in this situation, we have to select the theory parser manually.

4.2.3 Creating a Rule

Besides, the `spindle.io.parser.DflTheoryParser2` also provides method to generate a rule on the fly, as shown in the code below.

```
Rule rule = DflTheoryParser2.extractRuleStr(String ruleString);
```

The same also apply to the `XmlTheoryParser`.

In addition, it is also possible to create a rule using only the Java API without writing either a DFL language or XML version of the rule. This is not recommended as the rule class used for Maher's approach (version 1.X.X) and Lam and Governatori's approach (version 2.X.X) is different.

If it is really necessary for the application to create a rule using Java API, then the application can create the rule by executing the `spindle.core.dom.DomUtilities.getRule(String id, RuleType ruleType)` method with the required arguments: (1) a unique rule identifier, and (2) the rule type, and insert the body (if any) and head literals to the rule programmatically.

However, be aware that these API may change without notice.

4.2.4 Extracting the set of rules for literals

Sometime we may want to limit the inference process to the set of rules that are related to some literals only. We can use the following functions in `spindle.core.dom.Theory` class to achieve this:

```
Theory theory = origTheory.getRulesToDerive(Literal literal);
Theory theory = origTheory.getRulesToDerive(Set<Literal> literals);
```

Here, `origTheory` is the knowledge base of the application, `literal` or `literals` are literals that we interest in, and `theory` is the defeasible theory that contains only the rules that are required for inferencing the literals.

This methods are particularly useful when the knowledge base contains large number of rules which may not be useful all the time. It helps to remove the set of unnecessary rules from the knowledge base which subsequently improving the performance of the inference process.

However, for knowledge base with limited number of rules, the benefit of this operation may not be that significant and may incur addition overhead for creating a new theory in some cases. We advise users to test the performance of the application before and after theory extraction so as to have a clear picture of whether this operation is needed in the applications.

4.3 Configuring the reasoning environment

Users, or applications, can configure the reasoning environment through command-line option as described in Section 2.2, or can be done programatically using the `spindle.sys.Conf` class. The `spindle.sys.Conf` class is the main configuration class that provide all the options that SPINdle currently support. For example, the following code shows how to configure SPINdle to log the inference status of each rules (*applicable*, *discarded* and *defeated*):

```
spindle.sys.Conf.setLogInferenceProcess(true);
```

and retrieve the rules inference statuses using the following method:

```
spindle.tools.explanation.InferenceLogger =
                                spindle.Reasoner.getInferenceLogger();
```

The following methods show how to configure SPINdle to reason with ambiguity propagation and well-founded semantics, respectively:

```
spindle.sys.Conf.setReasoningWithAmbiguityPropagation(true);
spindle.sys.Conf.setReasoningWithWellFoundedSemantics(true);
```

Other features such as displaying the reasoning statistics, showing progress information, displaying the results of the inference process, etc., can also be set using the `spindle.sys.Conf` class.

4.4 Multiple Rule Engines

Each `spindle.Reasoner` class represents an independent defeasible reasoning engine. A single program can includes any number of defeasible reasoning engines. Each instance of the individual `spindle.Reasoner` class can have their own working memory, reasoning stages, rule base, etc, and can all function in separate threads. Applications can use multiple identical engines in a pool and can interact with the Java application in some way.

However, the same does not apply to the `spindle.sys.Conf` class which provides the system-wise configuration support to SPINdle, and should be patient with the system properties values when reasoning with different DL variants in the same JVM.

Running multiple `spindle.Reasoner` class in the same JVM requires additional command line option during the start-up process. Users, or applications, are required to set the option `--reasoner.multiThreadMode` to `true` when starting the SPINdle reasoner. Fail to do this may lead to the throw of *concurrent modification exception* during the inference process since the same reasoning engine may be used by multiple instances of the SPINdle reasoner. This option can also be set programatically using the following command:

```
Conf.setMultiThreadMode(true);
```

4.5 Error Reporting and Debugging

An exception will throw whenever an error appear. The exception, in most cases, will contain an explanation of the problem. User can print a stack trace of the exception whenever it is necessary to identify the problem.

In SPINdle, `spindle.lang.SpindleException` is the base class of all exception classes. Table 4.1 shows the commonly seen SPINdle exception classes and their description.

Exception name	Description
spindle.core.dom.TheoryException	Exception throw when problems appear during theory manipulation.
spindle.engine.ReasoningEngineException	Exception throw during the inference process.
spindle.engine.ReasoningEngineFactoryException	Exception throw when no inferencing engine of a particular theory type is found.
spindle.engine.TheoryNormalizerException	Exception throw when no theory normalizer of a particular theory type is found.
spindle.io.OutputterException	Exception throw when problems appear during the output process.
spindle.io.ParserException	Exception throw the inputted theory string contains error.
spindle.lang.ConfigurationException	Exception throw when problem occur with the configuration.

Table 4.1: Common SPINdle exception classes

Chapter 5

Extending SPINdle

Building and deploying monolithic applications is a thing in the past [24]. As the complexity of applications increases, decomposing a large system into several smaller and well-defined collaborating pieces, and hiding design and implementation details behind a stable application programming interface (API) is a much better way to go. It helps reducing the complexity of application development and improves the testability and maintainability of the whole application through enforcing the logical module boundaries.

SPINdle is an expandable system designed using plug-in architecture such that modules (or *components*) are loaded dynamically into the application when they are necessary, which subsequently helps reducing the footprint and memory consumption of the whole system; and can be extended easily according to the future needs. Figure 5.1 depicts the overall system architecture of SPINdle.

This chapter focuses on the implementation aspect of the latest version of SPINdle. It should be noted that SPINdle is under active development and the source code may differs in minor ways from the description here.

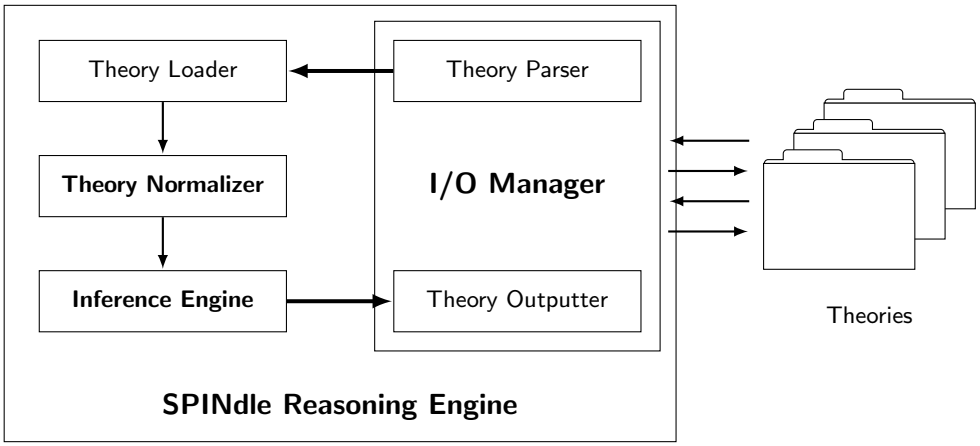

Figure 5.1: SPINdle Architecture

5.1 Theory modeling

The key to an efficient implementation of the algorithms proposed in [16, 18] is the data structure used to model a defeasible theory, especially the data structure used to represent rules such that rules in association with a particular literal (and its negation) and/or superiority relations can be retrieved from the theory with a limited amount of time. Here we have extended the data structure proposed by Maher [18] to handle modal literals and inferiorly defeated rules, and is exemplified (albeit incompletely) in Figure 5.2 for the theory below:

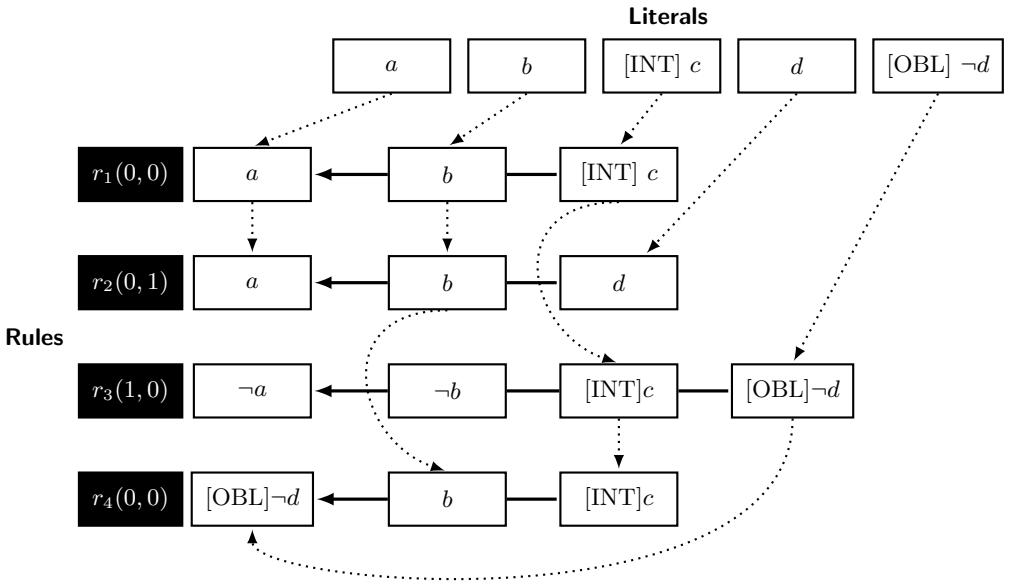

Figure 5.2: Data structure for literals-rules association

$$
\begin{array}{lrcl}
r_1 & : & b, [\text{INT}]c & \Rightarrow & a \\
r_2 & : & b, d & \Rightarrow & a \\
r_3 & : & \neg b, [\text{INT}]c, [\text{OBL}]\neg d & \Rightarrow & \neg a \\
r_4 & : & b, [\text{INT}]c & \Rightarrow & [\text{OBL}]\neg d \\
r_2 > r_3 & & & &
\end{array}
$$

Each rule body is represented as a set in the memory. For each literal p there are linked lists (the solid line) of the occurrences of p in the rules can be created during the theory parsing phase. Each literal occurrence has a link to the rules it occurs in (the dashed arrow). Using this structure, whenever a literal update occurs, the list of rules relevant to that literal can be retrieved easily. Thus instead of scanning through all the rules for empty head, only rules relevant to that particular literal will be scanned. Besides, this structure also help in retrieving information of the conflicting literal(s) that may appear in the theory, which thus further helps in speeding up the inference process.

Moreover, this structure also allows the deletion of literals and rules in time proportional to the number of literals deleted as it facilitates the process by detecting in constant time on whether a literal deleted was the only literal in that rule.

The two number sitting next to the rule label is the number of superior and inferior rules associated with a particular rule, which is particular important for implementing the approach proposed in [16]. Readers interested please refer to the paper for details.

Besides, the `Theory` class also provides useful methods that allows users to modify/ manipulate the theory according to their application needs, which helps in simplifying their development process. For details, please refer to the Java API.

5.2 I/O Manager

The I/O Manager manages the communications between SPINdle and users and/or applications. It provides users an interface to interact with SPINdle in loading and/or storing defeasible theories and conclusions after computation. It is composed mainly by two groups of functional elements: *theory parsers* and *theory outputters*, that carry out the required theories import and export processes, and can work independently from the SPINdle reasoning engine.

To be able to accept defeasible theories that are stored in local file system or anywhere else that can be accessed through the Internet, SPINdle uses URI to represent the location of the defeasible theories. The biggest advantage of this method is that URI

are independent of the system locale, meaning that no matter which locale SPINdle are running, the same defeasible theory files can be retrieved using the same URIs.

After receiving an URI, the I/O Manager then fetches the document from the source location and performs an initial analysis on the theory to decide which *theory parsers* should be employed to parse the document.

There are two ways for us to extend SPINdle in supporting different representation formalisms, either (1) create a theory theory parser/outputter-class by extending existing parser/outputter-classes; or (2) create an entirely new theory parser/outputter-class written in Java language.

5.2.1 Creating custom Theory Parser

There are numerous third party libraries available that can integrate into SPINdle as if necessary[1]. 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. Listing 5.1 shows the default content of the configuration file.

Listing 5.1: SPINdle I/O configuration file

```
1 <spindle>
2     <io classname="spindle.io.parser.XmlTheoryParser" />
3     <io classname="spindle.io.parser.DflTheoryParser" />
4     <io classname="spindle.io.outputter.XmlTheoryOutputter" />
5     <io classname="spindle.io.outputter.DflTheoryOutputter" />
6 </spindle>
```

Or, at users' preference, the I/O Manager can detect all the parser classes available on the classpath automatically (Section 2.2.1). However, it is not recommanded for big applications as considerable time is required.

5.2.2 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` class, which again, already

[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.

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.

5.3 Extending the Reasoning Components

5.3.1 Creating custom Theory Normalizer

The `spindle.engine.TheoryNormalizer` class is the base class of all theory normalizers in SPINdle. It provides most of the methods and items that are used in transforming a defeasible theory. To create a custom theory normalizer, users are required to extend this class through implementing the following method.

```
protected abstract void transformTheoryToRegularFormImpl()
              throws TheoryNormalizerException;
```

It is to be noted that due to the methods of eliminating superiority relations and defeaters are common among different rule type, it only requires user to provide the implementation on transforming defeasible theory into regular form. However, users are still free to override methods that appear in the class as necessary.

5.3.2 Creating custom Inference Engine

Researchers or application developers can extend the reasoning capabilities of SPINdle through creating their own inference engine. In order to be used by SPINdle, the newly implemented class should implement the `spindle.engine.ReasoningEngine` interface which provide method on computing the conclusions of a defeasible theory, and other auxiliary methods such as adding and removing the reasoning engine listener. Below are the methods required for implementing an inference engine.

```
public Map<Literal, Map<ConclusionType, Conclusion>> getConclusions
              (final Theory theory) throws ReasonerException;

public ProcessStatus clear();

public void addReasoningEngineListener(ReasoningEngineListener
              listener);

public void removeReasoningEngineListener(ReasoningEngineListener
              listener);
```

However, implementing an inference engine from scratch is a time consuming and tedious job. It is recommended that users can implement their own inference engine by extending some of the inference engines available in the SPINdle package. Table 5.1 shows some of them.

Theory type[a]	Variant[b]	Class name
SDL	AB	`spindle.engine.sdl.SdlReasoningEngine2`
SDL	AP	`spindle.engine.sdl.SdlReasoningEngineAP2`
MDL	AB	`spindle.engine.sdl.MdlReasoningEngine2`
MDL	AP	`spindle.engine.sdl.MdlReasoningEngineAP2`

[a]SDL - Standard Defeasible Logic, MDL - Modal Defeasible Logic
[b]AB - Ambiguity blocking (with optional well-founded semantics), AP - Ambiguity propagation (with optional well-founded semantics)

Table 5.1: Inference engines available in the SPINdle package

5.3.3 Configure the extended reasoning components

Configuring the extended reasoning components is different from configuring the theory parser or theory outputter classes. In order to make sure the correct theory normalizer and reasoning engine are retrieved with a particular theory type, it is necessary for us to modify the content of some of the SPINdle classes using the following steps.

1. If a new type of defeasible theory is introduced, you are then required to modify the content of the enumeration class `spindle.core.dom.TheoryType`.
2. Then, you have to modify the content of the `spindle.engine.ReasoningEngineFactory` according to the type of theory and the intuition that want to capture. Listing 5.2 below shows the content in the factory class that is used to retrieve a theory normalizer according to the theory type. The `theoryNormalizers` entity here is a hash map to store the objects instantiated so that that no object of the same theory type will instantiate twice.

Listing 5.2: Code snippet for returning theory normalizer from `ReasoningEngineFactory`

```
1  public static final TheoryNormalizer getTheoryNormalizer(
2          TheoryType theoryType) throws ReasoningEngineFactoryException {
3      if (theoryType == null) return null;
4      TheoryNormalizer theoryNormalizer = theoryNormalizers.get(theoryType);
5      if (null == theoryNormalizer) {
6          switch (theoryType) {
7          case SDL:
8              theoryNormalizer = new spindle.engine.sdl.SdlTheoryNormalizer();
9              break;
10         case MDL:
11             theoryNormalizer = new spindle.engine.mdl.MdlTheoryNormalizer();
12             break;
13         default:
14             throw new ReasoningEngineFactoryException
15                 (ErrorMessage.THEORY_UNRECOGNIZED_THEORY_TYPE);
16         }
17         theoryNormalizers.put(theoryType, theoryNormalizer);
18     }
19     return theoryNormalizer;
20 }
```

Chapter 6

Future development

Currently, we have a couple of ideas that will probably be included in SPINdle some day. They include, but not limited to:

- Extended support for data types. To support the use of set containment, union and join operations, and the use of literals as predicate of other literals, etc.
- Extended reasoning capabilities to support reasoning with Temporal Defeasible Logic [11].
- Implemented the theory grounding mechanism.
- Some sort of unified methods/frameworks that can be linked up with an DBMS such that the conclusions derived can be used to query a table/field in the database.
- Extended I/O support to defeasible theories represented using different formalisms, such as OWL, RDF, RuleML, etc.

If you have some suggestions or if you found a bug in SPINdle, please send an email to the author (oleklam@gmail.com).

Appendix A

SPINdle Defeasible Theory XML Schema

Listing A.1: SPINdle XML Schema (version 2.2.0)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    targetNamespace="http://spin.nicta.org.au/spindle/spindleDefeasibleTheory2.xsd"
    attributeFormDefault="unqualified" version="2.0.0"
    xmlns:spindle="http://spin.nicta.org.au/spindle/spindleDefeasibleTheory2.xsd">
    <xs:simpleType name="stAtom">
        <xs:restriction base="xs:string">
            <xs:pattern value="[a-zA-Z][a-zA-Z0-9_]*"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="stMode">
        <xs:restriction base="xs:string">
            <xs:pattern value="[a-zA-Z][a-zA-Z0-9_]*"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="stTime">
        <xs:restriction base="xs:long"/>
    </xs:simpleType>
    <xs:simpleType name="stRuleLabel">
        <xs:restriction base="xs:string">
            <xs:pattern value="[a-zA-Z][a-zA-Z0-9_]*"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="stPredicate">
        <xs:restriction base="xs:string">
            <xs:pattern value="[a-zA-Z0-9]+"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="stLiteralVariableAtom">
        <xs:restriction base="xs:string">
            <xs:pattern value="@.*"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="stLiteralBooleanFunction">
        <xs:restriction base="xs:string"> </xs:restriction>
    </xs:simpleType>
    <!--
    <xs:simpleType name="stLiteralBooleanFunction">
        <xs:restriction base="xs:string">
            <xs:pattern value="$.*$"/>
        </xs:restriction>
    </xs:simpleType>
    -->
    <xs:complexType name="ctInterval">
        <xs:sequence minOccurs="1">
            <xs:element name="start" type="spindle:stTime" minOccurs="0"/>
            <xs:element name="end" type="spindle:stTime" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ctPredicates">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="predicate" type="spindle:stPredicate"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ctLiteral">
```

```
        <xs:sequence>
            <xs:element name="atom" type="spindle:stAtom" minOccurs="0"/>
            <xs:element name="mode" type="spindle:stMode" minOccurs="0"/>
            <xs:element maxOccurs="2" minOccurs="0" name="not">
                <xs:complexType>
                    <xs:choice>
                        <xs:element name="atom" type="spindle:stAtom"/>
                        <xs:element name="mode" type="spindle:stMode"/>
                    </xs:choice>
                </xs:complexType>
            </xs:element>
            <xs:element name="interval" type="spindle:ctInterval" minOccurs="0" maxOccurs="1"/>
            <xs:element name="predicates" minOccurs="0" maxOccurs="1" type="spindle:
                ctPredicates">
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ctRuleHeadLiterals">
        <xs:sequence>
            <xs:element name="literal" type="spindle:ctLiteral" maxOccurs="unbounded" minOccurs
                ="1"
            />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ctRuleBodyLiterals">
        <xs:choice>
            <xs:element name="literal" type="spindle:ctLiteral"/>
            <xs:element name="and">
                <xs:complexType>
                    <xs:sequence maxOccurs="unbounded">
                        <xs:element name="literal" type="spindle:ctLiteral"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:choice>
    </xs:complexType>
    <xs:complexType name="ctRule">
        <xs:sequence>
            <xs:element maxOccurs="1" minOccurs="0" name="mode" type="spindle:stMode"/>
            <xs:element maxOccurs="1" minOccurs="0" name="interval" type="spindle:ctInterval"/>
            <xs:element name="head" maxOccurs="1" minOccurs="1" type="spindle:
                ctRuleHeadLiterals"/>
            <xs:element name="body" minOccurs="0" type="spindle:ctRuleBodyLiterals" maxOccurs="
                1"/>
        </xs:sequence>
        <xs:attribute name="label" type="spindle:stRuleLabel" use="optional"/>
        <xs:attribute name="ruletype" use="required">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="STRICT"/>
                    <xs:enumeration value="DEFEASIBLE"/>
                    <xs:enumeration value="DEFEATER"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
    <xs:complexType name="ctSuperiority">
        <xs:attribute name="superior" type="spindle:stRuleLabel" use="required"/>
        <xs:attribute name="inferior" type="spindle:stRuleLabel" use="required"/>
    </xs:complexType>
    <xs:complexType name="ctModeConversionRule">
        <xs:sequence>
            <xs:element maxOccurs="1" minOccurs="1" name="from" type="spindle:stMode"/>
            <xs:element maxOccurs="unbounded" name="to" type="spindle:stMode"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ctModeConflictRule">
        <xs:sequence>
            <xs:element maxOccurs="1" minOccurs="1" name="mode" type="spindle:stMode"/>
            <xs:element maxOccurs="unbounded" minOccurs="1" name="conflictWIth"
                type="spindle:stMode"/>
        </xs:sequence>
    </xs:complexType>
```

```
    <xs:complexType name="ctModeExclusionRule">
        <xs:sequence>
            <xs:element name="mode" type="spindle:stMode"/>
            <xs:element maxOccurs="unbounded" name="excludeWith" type="spindle:stMode"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ctLiteralVariable">
        <xs:sequence>
            <xs:element name="atom" type="spindle:stLiteralVariableAtom"/>
            <xs:element minOccurs="0" name="predicates" type="spindle:ctPredicates"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ctLiteralVariablePair">
        <xs:sequence>
            <xs:element name="name" type="spindle:ctLiteralVariable"/>
            <xs:element name="value" type="spindle:ctLiteralVariable"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ctLiteralBooleanFunctionPair">
        <xs:sequence>
            <xs:element name="name" type="spindle:ctLiteralVariable"/>
            <xs:element name="value" type="spindle:stLiteralBooleanFunction"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="theory">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="1" minOccurs="0" name="description" type="xs:string"/>
                <xs:element maxOccurs="unbounded" minOccurs="0" name="convert"
                    type="spindle:ctModeConversionRule"/>
                <xs:element maxOccurs="unbounded" minOccurs="0" name="conflict"
                    type="spindle:ctModeConflictRule"/>
                <xs:element maxOccurs="unbounded" minOccurs="0" name="exclude"
                    type="spindle:ctModeExclusionRule"/>
                <xs:element maxOccurs="unbounded" minOccurs="0" name="literalVariable"
                    type="spindle:ctLiteralVariablePair"> </xs:element>
                <xs:element name="literalBooleanFunction" maxOccurs="unbounded" minOccurs="0"
                    type="spindle:ctLiteralBooleanFunctionPair"> </xs:element>
                <xs:choice maxOccurs="unbounded">
                    <xs:element name="fact" type="spindle:ctLiteral"/>
                    <xs:element name="rule" type="spindle:ctRule"/>
                </xs:choice>
                <xs:element maxOccurs="unbounded" minOccurs="0" name="superiority"
                    type="spindle:ctSuperiority"> </xs:element>
            </xs:sequence>
        </xs:complexType>
        <!--
        <xs:key name="ruleLabelKey">
            <xs:selector xpath="spindle:rule"/>
            <xs:field xpath="@label"/>
        </xs:key>
        <xs:keyref name="superiorRuleRef" refer="spindle:ruleLabelKey">
            <xs:selector xpath="spindle:superiority"/>
            <xs:field xpath="@superior"/>
        </xs:keyref>
        <xs:keyref name="inferiorRuleRef" refer="spindle:ruleLabelKey">
            <xs:selector xpath="spindle:superiority"/>
            <xs:field xpath="@inferior"/>
        </xs:keyref>
        -->
    </xs:element>
    <xs:complexType name="ctConclusion">
        <xs:sequence>
            <xs:element name="tag">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="DEFINITE_PROVABLE"/>
                        <xs:enumeration value="NOT_DEFINITE_PROVABLE"/>
                        <xs:enumeration value="DEFEASIBLE_PROVABLE"/>
                        <xs:enumeration value="NOT_DEFEASIBLE_PROVABLE"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
```

```
            <xs:element name="literal" type="spindle:ctLiteral"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="conclusions">
      <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="1" minOccurs="0"
                name="description" type="xs:string" />
              <xs:sequence maxOccurs="unbounded" minOccurs="0">
                <xs:element name="conclusion" type="spindle:ctConclusion"></xs:element>
              </xs:sequence>
            </xs:sequence>
      </xs:complexType>
    </xs:element>
    <!-- For the use of parsing a single rule -->
    <xs:element name="rules">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="rule" type="spindle:ctRule"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="ctConclusions">
      <xs:sequence>
        <xs:element name="description" type="xs:string"></xs:element>
        <xs:sequence>
          <xs:element name="conclusion" type="spindle:ctConclusion"></xs:element>
        </xs:sequence>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

References

[1] G. Antoniou, D. Billington, G. Governatori, and M. J. Maher, "On the Modeling and Analysis of Regulations," in *Proceedings of the Australian Conference Information Systems*, 1999, pp. 20–29.

[2] G. Antoniou, D. Billington, G. Governatori, and M. J. Maher, "Representation Results for Defeasible Logic," *ACM Transactions on Computational Logic*, vol. 2, no. 2, pp. 255–286, 2001.

[3] G. Antoniou and M. J. Maher, "Embedding Defeasible Logic into Logic Programs," *Theory and Practice of Logic Programming (TPLP)*, vol. 6, no. 6, pp. 703–735, 2002.

[4] N. Bassiliades, G. Antoniou, and I. Vlahavas, "A Defeasible Logic Reasoner for the Semantic Web," *International Journal of Semantic Web and Information Systems (IJSWIS)*, vol. 2, no. 1, pp. 1–41, 2006.

[5] D. Billington, "Defeasible Logic is Stable," *J Logic Computation*, vol. 3, no. 4, pp. 379–400, 1993.

[6] M. Dastani, M. Birna Riemsdijk, and J.-J. Meyer, "Programming Multi-Agent Systems in 3APL," in *Multi-Agent Programming*, ser. Multiagent Systems, Artificial Societies, And Simulated Organizations, G. Weiss, R. Bordini, M. Dastani, J. Dix, and A. Fallah Seghrouchni, Eds., vol. 15, Springer US, 2005, pp. 39–67.

[7] M. Dastani, G. Governatori, A. Rotolo, and L. van der Torre, "Preferences of Agents in Defeasible Logic," in *AI 2005: Advances in Artificial Intelligence, 18th Australian Joint Conference on Artificial Intelligence*, rules with multiple conclusions, Springer, 2005, pp. 695–704.

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

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

[10] G. Governatori and A. Rotolo, "Deontic logic in computer science," in, A. Lomuscio and D. Nute, Eds., ser. LNAI 3065. Springer-Verlag, 2004, ch. Defeasible Logic: Agency, Intention and Obligation, pp. 114–128.

[11] G. Governatori and P. Terenziani, "Temporal Extension of Defeasible Logic," in *Proceedings of the IJCAI'07 Workshop on Spatial And Temporal Reasoning*, 2007.

[12] J. Hintikka, *Knowledge and Belief: An Introduction to the Logic of the Two Notions*. Cornell University Press, 1962.

[13] J. F. Horty, "Skepticism and floating conclusions," *Artificial Intelligence*, vol. 135, no. 1-2, pp. 55 –72, 2002.

[14] H.-P. Lam and G. Governatori, "On the Problem of Computing Ambiguity Propagation and Well-Founded Semantics in Defeasible Logic," in *Proceedings of the 4th International Web Rule Symposium: Research Based and Industry Focued (RuleML-2010)*, A. Rotolo, J. Hall, M. Dean, and S. Tabet, Eds., RuleML, Washington, DC, USA: Springer, 2010.

[15] H.-P. Lam and G. Governatori, "The Making of SPINdle," in *Proceedings of the 2009 International Symposium on Rule Interchange and Applications (RuleML 2009)*, A. Paschke, G. Governatori, and J. Hall, Eds., RuleML, Las Vegas, Nevada: Springer-Verlag, 2009, pp. 315–322.

[16] H.-P. Lam and G. Governatori, "What are the Necessity Rules in Defeasible Reasoning?" In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-11)*, J. Delgrande and W. Faber, Eds., Vancouver, BC, Canada: Springer Berlin / Heidelberg, 2011, pp. 187–192.

[17] B. van Linder, "Modal Logic for Rational Agents," Ph.D. Thesis, Department of Computer Science, Utrecht University, Gemert, 1996.

[18] M. J. Maher, "Propositional Defeasible Logic has Linear Complexity," *Theory and Practice of Logic Programming*, vol. 1, no. 6, pp. 691–711, 2001.

[19] D. Nute, "Defeasible Logic," in *Handbook of Logic for Artificial Intelligence and Logic Programming*, D. Gabbay and C. Hogger, Eds., vol. III, Oxford University Press, 1994, pp. 353–395.

[20] D. Nute, "Norms, Priorities, and Defeasibility," in *Norms, Logics and Information Systems. New Studies in Deontic Logic and Computer*, P. McNamara and H. Prakken, Eds., IOS Press, Amsterdam, 1998, pp. 201–218.

[21] D. H. Pham, G. Governatori, S. Raboczi, A. Newman, and S. Thakur, "On Extending RuleML for Modal Defeasible Logic," in *RuleML 2008: The International RuleML Symposium on Rule Interchange and Applications*, N. Bassiliades, G. Governatori, and A. Paschke, Eds., 2008.

[22] L. A. Stein, "Resolving Ambiguity in Nonmonotonic Inheritance Hierarchies," *Artificial Intelligence*, vol. 55, no. 2-3, pp. 259–310, 1992.

[23] D. S. Touretzky, J. F. Horty, and R. H. Thomason, "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*, Milan, Italy: Morgan Kaufmann Publishers Inc., 1987, pp. 476–482.

[24] C. Walls, *Modular Java - Creating Flexible Applications with OSGi and Spring*. The Pragmatic Bookshelf, 2009.