The University of Queensland

School of Information Technology and Electrical Engineering

An Implementation of Temporalised Defeasible Logic

by

Andrew Newman

The School of Information Technology and Electrical Engineering
The University of Queensland

Submitted for the degree of Bachelor of Information Technology (Honours)
8th November 2007
Abstract
Defeasible logic is a nonmonotonic language that allows new inferences to be made as new information is discovered. It was developed in order to be efficiently and easily implemented. However, in order to be more widely applicable the addition of expressing time has been highlighted. This allows a defeasible theorem to reason over statements that may have been true in the past but may not be true in present or in the future.

In order to add this extra dimension, a defeasible logic system has been developed and the additional functionality added. An architecture has been developed that allows the expression of temporal rules and literals and for the system to be able to reason over these rules.

The efficiency of the implemented defeasible logic system and its temporal counterpart is investigated. This includes the effect of the developed architecture on the time and resources required to reason over these temporal theorems.

To help implement this system and to allow end users to create theorems a mini-language, or domain specific language, has also been developed. This provides a restricted but powerful set of commands to be used in order to create defeasible and temporal defeasible logic theorems.
Acknowledgments
I would like to thank the following people for making this thesis possible:

- My family, especially Donna and Woz, and
- My supervisor Guido Governatori.
1. Introduction
Defeasible logic is a logic that has the notion of rules that can be defeated. It is part of a more general area of research, defeasible reasoning [14] [17] [36]. Defeasible reasoning in contrast with traditional deductive logic, allows the addition of further propositions to make an existing belief false, making it nonmonotonic [29]. It is also notable for its low computational complexity (it has linear complexity [30]).

The interest in defeasible logic is mainly in the areas of control systems [16], in modelling regulations, business rules and contracts [9] [3] [18] [22], legal reasoning [20], agents [39] [26] [24] [11] [25] [14], and the Semantic Web [6] [21] [19] [5] [11].

In early work performed by Pollock [37] and more abstract issues such as the Yale shooting problem [13][34], it has been shown that there is a requirement for the addition of a temporal aspect to be added to defeasible reasoning. More recently, temporal extensions to defeasible logic have been discussed for agents and legal reasoning domains [27] [24] [28].

We discuss the previous work in this area including Nute’s and Billington’s defeasible logic and previously discussed temporal extensions and the motivation for efficiency. The design and implementation of the project will be discussed including general architecture and performance. Furthermore, we will discuss further improvements that can be made to the system.

2. Related Work
Defeasible reasoning has a number of approaches. Some are focused on it as an area of logic and others treat it as epistemology (the study of knowledge) [29]. While they are related, the former is the main area of study with the later used for context. Many of the applications using a logical approach are applicable in areas such as argumentation [14].

2.1. Argumentation
An early formalisation of nonmonotonic reasoning that used defeasible reasoning was argumentation. Argumentation was originally, formally defined in the legal domain in the 1950s by Hart’s “Ascription of Responsibilities and Rights” before it was introduced to computing systems [15]. It was based on the older, Aristotelian tradition and was seen by Chesñevar et al [15] as:

“Modelling arguments appears at the foundation of AI’s understanding of rule-base systems (where rules can come into conflict)”.

Many different types of defeasible logic were derived from this approach including: Pearl’s System Z, Geffner’s conditional entailment, and Simari’s and Loui’s argument-based system [17]. This lead to the implementation of systems including: Nathan and EVID [14] [15]. Argumentation also formed the basis for applications in natural language, hypertext and decision support [15].
2.2. Basic Defeasible Logic

Nute’s original approach took defeasible reasoning and produced a basic defeasible logic [35]. It drew on previous work by Reiter’s default logic from the late 1970s [34] and Pollock’s defeasible reasoning from the 1960s [29]. Antonious et al [8] characterised Nute’s logic as:

“...designed to be easily implementable right from the beginning, unlike most other approaches. In fact it has an implementation as a straightforward extension of Prolog”

In the original definition of his defeasible logic, it was described as an intuitive approach to defeasible reasoning that included: strict rules, defeasible rules, defeaters, precedence relations and a directly skeptical approach to resolving conflicting rules. It included an implementation of this logic as an extension to Prolog, called d-Prolog. Importantly, Nute’s basic defeasible language was designed to be straightforward to implement and low in computational complexity [32] [9]. This approach contrasts with previous approaches in that the proof theory came before the semantics [35]:

“In the case of defeasible logic, the proof theory came first and the semantics came much later.”

In describing the properties of defeasible logic, Nute describes defeasible rules based on Pollock’s work [36]. As such, there are two ways in which a rule can be defeated by another:

- Rebutted – Where another rule attacks the conclusion of the rule.
- Undercut – Where the link between the premise and conclusion does not apply.

From default logic there is a concept of a default rule that consists of three components: prerequisite, justification and consequent [29]. Defeasible rules and normal default rules have the same justification and consequent.

In the basic defeasible logic, a skeptical strategy is employed. That is, if two rules conflict there is no result, they are said to defeat one another [17] [8]. This differs with default logic where conflict resolution first takes a credulous approach (that finds as many conclusions as possible) and then considers the intersection of extensions to the theory (known as indirectly skeptical) [17]. An extension to a theory is where each rule is considered as either being: inapplicable, applied or defeated [17].

2.3. General Defeasible Logic

Billington’s general defeasible logic is a modified, formal definition of Nute’s work [13]. Nute’s work was seen as attractive as it had several implementations and solved “commonly troublesome problems” [13].

The key differences of general defeasible logic to previous logics were:

- Superiority relationships were made explicit [13].
- It was more maintainable through a lack of probabilistic rules [13].
A normal form was developed where literals occur only in rules, strict rules cannot be overruled and there are no facts [8] [32].

It was rule based without disjunctions [12].

The development of a normal form of general defeasible logic was developed [8]. This entailed removal of facts and superiority relationships without the loss of expressivity. It also suggested that strict rules could almost be eliminated where they only have interactions with defeasible rules. The minimal set of expressiveness for a defeasible theory was reduced to: strict rules, defeasible rules and defeaters. Furthermore, it was shown that while defeaters do not add expressiveness in terms of conclusions [32], they were still required, and could not be replicated using just strict and defeasible rules.

Two implementations based on using normal forms of general defeasible logic were developed: Deimos and Delores [32]. These two systems consisted of a backward and forward chaining theorem prover respectively. The implementation of the forward chaining prover has been shown to be linear in complexity [30].

Further investigations lead to the conclusion that the transformation of priority relationships and defeaters were similar [10]. The development of this normal form created new literals as intermediate forms between rule heads and bodies. For each original rule involved in a superiority relationship two new rules are generated and three new rules are created for each defeater rule [10]. This contrasts with the previous result and was based on a different variation of defeasible logic where defeaters could not be used to support positive conclusions through counter attacks [10].

The semantics of defeasible logic has been described in terms of logic programs [7], model theory [31] and argumentation [23].

2.4. Adding Defeasible Logic to the Semantic Web

Defeasible logic has been applied to the Semantic Web as a way to handle conflicts in rule systems [12] [6] [22] [21] [18]. Defeasible reasoning as applied to the Semantic Web overcomes problems associated with inferencing by allowing:

- Reasoning over incomplete information.
- Conflicting rules or rules with exceptions to be resolved.
- Ontologies to express default inheritance.
- Resolving differences when merging ontologies.

This was achieved through extending RuleML and combining description logics and defeasible reasoning [4]. An alternative to extending description logic with defeasible logic used defeasible rules and defeasible logic proof theory has also been presented [19].

2.5. Temporal Defeasible Logic

In Pollock’s investigation of agents navigating a changing world, he highlighted some of the difficulties of temporal reasoning [37]:

“An agent that did all of its temporal reasoning...would be led into crippling computational complexities. Every time the agent wanted to reuse a belief about its
surroundings, it would have to reinfer it for the present time...Human beings achieve the same result in a more efficient way by employing the temporal indexical “now”. Rather than repeatedly reinferring a property as time advances, they infer it once as holding now, and that single belief is retained until it becomes defeated. The mental representation (i.e., formula) believed remains unchanged as time advances, but the content of the belief changes continuously in the sense that at each instant it is a belief about that instant.”

He also introduced the idea of a persistent rule [37]:

"A persistent rule is a rule whose conclusion holds at all instants of time after the conclusion has been derived, unless interrupting events occur; transient rules, on the other hand, establish the conclusion only for a specific instant of time."

Similarly, in a legal framework temporal awareness is required for supporting normative conditions and positions [27] [24] [28].

These clearly expressed the importance of introducing a temporal dimension to rules and literals.

In temporal defeasible logic, there are two types of temporal literals. The first, an expiring temporal literal, consists of a pair \( l : t \) that denotes a literal, \( l \), that is valid for \( t \) time instances. The second type of temporal literal, a persistent temporal literal, consists of a pair \( l \@ t \) and denotes a literal, \( l \), that is active after \( t \) time instances have passed and is valid thereafter.

A temporal rule takes the form:

\[
\begin{align*}
a_1 & : d_1 \ldots a_n : d_n \Rightarrow^d b : d_b
\end{align*}
\]

This is where pairs, \( e : d \), represent an event identifier, \( e \), and a duration \( d \). Instantaneous events have duration of 0, while persistent events have an infinite duration. The \(" \Rightarrow^d \)" indicates the delay between the cause (\( a_1 : d_1 \ldots a_n : d_n \)) and the effect (\( b : d_b \)). There are many interpretations to the exact semantics of this delay [28]. The delay maybe the delay between the first cause or the last cause and the effect; the delay may be interpreted instead to begin at the start or the end of the given durations.

3. The Design of a Temporal Logic System

The temporal defeasible logic system developed is composed of four main components:

- A defeasible logic system.
- Temporal additions.
- A language to express temporal defeasible rules.
- A user Interface.

The defeasible logic system implements a forward chaining inference engine that computes all conclusions. It supports strict rules, defeasible rules and priority assertions.

The temporal additions support both types of temporal literals with some limitations (persistent and expiring) and temporal rules.
A user interface must allow users to enter new rules into the system, to modify the time and to display the results.

### 3.1. Defeasible Logic System

The defeasible logic system produced follows the basic architecture and data structures developed in the Delores forward chaining inference defeasible logic system [33]. It also creates priority assertions using the method described in [8], facts are converted into strict rules without bodies and all strict rules are added as defeasible rules.

The algorithm, shown below, describes a system with only defeasible rules.

```
initialize S
    K = ∅
    while (S ≠ ∅)
        choose s ∈ S
        add s to K
        case s of
            + Δp:
                delete all occurrences of p in rule bodies
                whenever a body with head h becomes empty
                    record + σ h
                    CheckInference(+ σ h, S)
            - Δp:
                delete all rules where p occurs in the body
                whenever there are no more rules for a literal
                    record - σ h
                    CheckInference(- σ h, S)
        end case
    end while
```

**Figure 1 - Defeasible Logic Algorithm**

Where “p” is all literals, “s” is all conclusions, “K” is the set of all proven conclusions and “S” is the set of conclusion that have yet to be use to prove more conclusions.

The set, “S”, is initialised with all facts (rules with heads but no bodies). To prove the positive conclusion of a rule the head of the rule is retrieved. This is then used to retrieve any possible rules with the negative version of either the head or in the body. If there are no negative rules, negative literals in the head of the rules, then the conclusion to the rule is added to “S”. Negative literals in the head of rules cause a search for rules with the positive version of the literal in the head or body. Again, if there are no positive rules then it is added to “S”. All collected conclusions in “S” are used to remove literals from rules. If the literal has been proven it is removed from all the bodies of the rules.

A given example of the algorithm can be demonstrated from [33] with the following rules:

(r1) b, c, d ⇒ a
One of the unique aspects of the method described in [33] is the datastructure used to perform the searching and removal of literals. The can be shown in the following diagram representing the three rules using the datastructures described in [33] (which is missing ¬a and ¬b along the top):

Across the top of the diagram it shows maps of literals to the rules that they appear in. Dashed lines represent the heads of rules. This structure allows literals to be found in their respective rules by linearly searching through the list.

To implement priority assertions, a rule that is used in a priority assertion is transformed into more rules. For example, the rule:

\[(r1) \ a \Rightarrow \ b\]

is transformed into,

\[a \Rightarrow \neg\text{inf}(r1)\]
\[\neg\text{inf}(r1) \Rightarrow b\]

If rule r1 is inferior to another rule, r2, then another rule is also generated:
3.2. Adding Time

To add time to a defeasible engine modifications were made to literals, rules and the selection of defeasible rules for a specific time. Literals that expire have two additional properties added: an offset and duration, while persistent literals have just an offset. The offset is the time, \( t \), from which the literal becomes valid. Temporal literals are given the current time and return whether they are valid for the given period - based on whether it is after the offset and if the duration is within that period in the case of expiring literals.

The key insight in the temporal defeasible engine is that while time adds extra properties to rules and literals, it acts to remove rules and literals to be processed by the underlying defeasible engine. In other words, a non-temporal defeasible engine deals with permanent literals with offsets of 0 and rules without delays between the cause and effect. The addition of time has the effect of removing which rules and literals are relevant.

The architecture of the system is shown in the following figure. Temporal rules are processed and the set of applicable literals, strict rules, defeasible rules and priority assertions are created and passed into the defeasible logic engine. The results are then passed back to which are presented as the result of interpreting the temporal defeasible theory.

![Figure 3 - Architecture of Temporal Defeasible Logic Engine](image)

Temporal rules contain temporal literals. To correctly model the delay between the body and the head, rules are created with the heads of the rules modified to accommodate the delay. For example:

\[(r1) \Rightarrow a@1\]
(r2) \[ a@1 \rightarrow^7 b:3 \]

This creates a literal, “a”, that becomes active at time offset 1, does not cause the head to be fired until time 8, and the result, “b”, only lasts until time 10, thereafter, only the fact “a” remains. So when “b” is created it is an expiring temporal term beginning at time 0. When the rule is created, “b” is modified to become active from 0 to 8 (the start time of the literal plus the delay).

### 3.3. A Domain Specific Language for Temporal Defeasible Logic

Domain specific languages (DSLs), or little languages [2], are defined as [1]:

“...a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.”

Many languages used today are domain specific language including: BNF, Excel and Word macro languages, HTML, LaTeX, Make and SQL [40]. There is also increasing support from commercial vendors to help develop DSLs such as JetBrains’ MPS¹ and Microsoft’s Domain Specific Language Tools².

The stages used in the development of a DSL is based on the stages described in [40], these are:

- **Decision** - Determining if a DSL is needed.
- **Analysis** - Gathering information about the problem domain and acquiring domain knowledge.
- **Design** - Whether to use an existing language (internal) or develop one separately (external) and whether to describe it formally or informally.
- **Implementation** - Whether to develop the solution by creating an interpreter, compiler, preprocessor, or whether to embed it into an existing language.

There are many reasons to develop a DSL [40]. For temporal defeasible logic the reasons to develop a DSL include:

- To more easily represent data structures in defeasible logic theorems such as rules and literals,
- To improve the ability to manipulate the complicated data structures such as: representing various temporal elements, adding literals to rules, to create different types rules, and to represent rules as part of priority relationships.
- To allow users to directly interact with the system by creating theorems using the DSL. This allows users to write theorems with scripting languages that can be executed without the need of compiling and provides rapid feedback.
- To closely model the formal definitions of temporal defeasible logic more closely than the development of a typical programming interface.

---

¹ [http://www.jetbrains.com/mps/](http://www.jetbrains.com/mps/)
Andrew Rock (with Deimos [38]) has previously completed the required analysis, through the formal definitions of the defeasible logic system, for a defeasible logic DSL. This has been used as the basis for developing a DSL for this system.

An internal DSL was chosen to improve the readability of exiting Java code and to provide a way to describe rules using a scripting language supported by the JVM, Javascript.

Shown below are the mappings from the standard representation of temporal logic theories to their representation in Javascript and Java. To try and improve readability, variable assignments and type definitions are not shown.

<table>
<thead>
<tr>
<th>Standard Syntax</th>
<th>Javascript DSL</th>
<th>Java DSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>term(&quot;a&quot;);</td>
<td>term(constant(&quot;a&quot;));</td>
</tr>
<tr>
<td>~a</td>
<td>termNot(&quot;a&quot;);</td>
<td>termNot(constant(&quot;a&quot;));</td>
</tr>
<tr>
<td>a@10</td>
<td>pterm(&quot;a&quot;, 10);</td>
<td>PersistentTemporalTerm.term(constant(&quot;a&quot;), 10);</td>
</tr>
<tr>
<td>a:10</td>
<td>term(&quot;a&quot;, 10);</td>
<td>ExpiringTemporalTerm.term(constant(&quot;a&quot;), 10);</td>
</tr>
<tr>
<td>r1: a -&gt; b</td>
<td>term(&quot;a&quot;).are(&quot;b&quot;);</td>
<td>a = term(constant(&quot;a&quot;)); b = term(constant(&quot;b&quot;)); aAreB = a.are(b);</td>
</tr>
<tr>
<td>r1 : a =&gt; b</td>
<td>term(&quot;a&quot;).usually(term(&quot;b&quot;));</td>
<td>a = term(constant(&quot;a&quot;)); b = term(constant(&quot;b&quot;)); aUsuallyB = a.usually(b);</td>
</tr>
<tr>
<td>a</td>
<td>r1 = term(&quot;a&quot;).usually(term(&quot;b&quot;)); r2 = term(&quot;c&quot;).usually(termNot(&quot;b&quot;)); assertion = r1. isMoreSpecificThan(r2); result = prove([term(&quot;a&quot;)], [], [r1, r2], [assertion]);</td>
<td>r1 = term(constant(&quot;a&quot;)).usually(term(constant(&quot;b&quot;))); r2 = term(constant(&quot;c&quot;)); usually(termNot(constant(&quot;b&quot;))); assertion = r1. isMoreSpecificThan(r2); result = constant(&quot;a&quot;).prove(EMPTY_SET, asSet(r1, r2), asSet(assertion));</td>
</tr>
</tbody>
</table>
While not nearly as succinct as the standard representation, it is much less verbose than standard object construction in Java. For example, to create the defeasible rule shown above, using standard Java idioms, requires the following code:

```java
Term aTerm = new TermImpl(new LiteralImpl(new ConstantImpl("a")));
Term bTerm = new TermImpl(new LiteralImpl(new ConstantImpl("b")));
DefeasibleRule r1 = new DefeasibleRuleImpl("r1", aTerm, bTerm);
```

3.4. User Interface

The user interface developed allows standard defeasible theories and temporal defeasible theories to be entered using the Javascript DSL. The display provides a slider and spinner to modify the current time. An example of the user interface is shown in the following figure.

![Temporal Defeasible Logic Engine's User Interface](image_url)

**Figure 4 - Temporal Defeasible Logic Engine's User Interface**

When the time changes or the “Prove” button is pressed the temporal defeasible logic system is invoked and the result displayed on the bottom half of the screen. The rules and literals that are currently applicable for the displayed time is passed to the defeasible logic engine. The current time can be moved forwards or backwards from the current time within the minimum and maximum times (0 to 100).
4. Implementation

The system was developed primarily using Java 1.5. This included many of the new features of the language including generics to reduce typing errors and static imports to reduce the verbosity to references of static methods and constants.

In total, 1508 NCSS lines of code were developed within 54 class, including tests and a performance harness based on the previously developed DTScale. Javascript was also used to allow the dynamic definition of theories. Swing was use to develop the user interface.

5. Methodology

The performance of the system uses a version of DTScale that was available as part of Deimos. DTScale automatically generates rules based on different algorithms. It was re-implemented in Java. Each of these algorithms generates a number of rules based on parameters given. This includes:

- nChain - This generates \( n \) rules with a body “\( ai \)’ and a head “\( ai + 1 \)’”. It also generates one rule with an empty body and a head “\( a0 \)’”.
- theoryWidth - This generates two sets of \( n \) rules. The first rules has a body “\( ai \)’ and a head “\( bi + 1 \)’’. The second with an empty body and a head “\( ai \)’”.
- nPerBody - This generates two sets of \( n \) times 4 rules. The first has a body “\( ai \)’”, “\( bi \)’”, “\( ci \)’”, “\( di \)’” and a head of “\( ei \)’”. The second rule generates 4 rules each with an empty body and a head of either “\( ai \)’”, “\( bi \)’”, “\( ci \)’”, or “\( di \)’”.
- evenOdd - This generates \( n \) times 3 or 4 rules. For each iteration 3 rules are generated. The first two rules have the same body, “\( ai \)’”, and one with a head “\( ai+1 \)’” and “\( ai+2 \)’”. The third rule has an empty body and a head “\( bi \)’”. Ever other iteration generates a fourth rule with a body “\( bi+1 \)’” and a head “\( ai*2 \)’” (i times 2).
- doubleChain - This generate two sets of \( n \) rules plus two other rules. For each iteration the two rules generated have the same body consisting of “\( ai \)’” and “\( bi \)’”. The heads of the rules consist of “\( ai+1 \)’” and “\( bi+1 \)’” respectively. The other two rules don’t have bodies and have heads “\( a0 \)’” and “\( b0 \)’” respectively.
- threeTree - This generates two sets of rules. The first set generates a set of rules from 0 to \( 3^{n-1} \) rules. The second set of rules is generated from \( 3^{n-1} \) to \( 3^n \). This leads to a total of \( 3^n \) rules. The first set of rules have a body with literals “\( ai*3+1 \)’”, “\( ai*3+2 \)’” and “\( ai*3+3 \)’” and a head “\( ai \)’”. The second set of rules have an empty body and a head “\( ai \)’”.

The testing was performed on both the defeasible logic system and the temporal defeasible logic system. By running the same tests on both systems we seek to determine the base performance and memory usage of the defeasible logic system and the impact of handling temporal theories using the new engine. The tests were performed by combining all the different types of generated rules into one (“All” test). Separate tests were also run with just one algorithm used with the same parameters. The parameters used are shown in the following table.
Table 1 - Rules Algorithms and Parameters Used for Results

<table>
<thead>
<tr>
<th>Rules Generation Algorithm</th>
<th>Parameters</th>
<th>Number of Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>nChain</td>
<td>(2,2), (30,30), (200, 200), (400,400), (700, 700)</td>
<td>6, 930, 40200, 16040, 49070</td>
</tr>
<tr>
<td>simpleChain</td>
<td>(10), (100), (200), (400), (4000)</td>
<td>11, 101, 201, 401, 4001</td>
</tr>
<tr>
<td>nPerBody</td>
<td>(5), (50), (500), (750), (7500)</td>
<td>25, 250, 2500, 3750, 37500</td>
</tr>
<tr>
<td>theoryWidth</td>
<td>(10), (100), (200), (400), (4000)</td>
<td>20, 200, 400, 800, 8000</td>
</tr>
<tr>
<td>evenOdd</td>
<td>(10), (100), (200), (400), (4000)</td>
<td>36, 351, 701, 1401, 14001</td>
</tr>
<tr>
<td>doubleChain</td>
<td>(10), (100), (200), (400), (4000)</td>
<td>22, 202, 402, 802, 8002</td>
</tr>
<tr>
<td>treeThree</td>
<td>(2), (3), (4), (5), (10), (12)*</td>
<td>9, 27, 81, 243, 59049, 531441*</td>
</tr>
</tbody>
</table>

* Not used in the “All” tests.

The tests were performed on a 2GHz Intel Core Duo with 2GB of main memory. The tests were run using IntelliJ 7 on OS X 10.5 with JVM parameters set to 1GB of available memory (-Xms1024m -Xmx1024m). Each test was performed 3 times, the values recorded from IntelliJ statistics panel, and the averages calculated.

6. Results

The following shows the results of testing the defeasible engine and the temporal defeasible engine. The defeasible engine is shown on the left with the temporal defeasible engine results on the right. It should be noted that the memory here is the peak memory usage and so is not an entirely accurate representation of the total memory usage, which is difficult to do, based on Java’s garbage collection behavior. Some of the temporal tests failed to complete including: the 5th All Rules, the 5th nChain and 5th treeThree for the temporal defeasible logic engine.
The difference in performance for the tests with the most number of rules is shown below.

### Table 2 - Comparison of Temporal Defeasible Engine and Defeasible Engine

<table>
<thead>
<tr>
<th>Test</th>
<th>Time Difference</th>
<th>Memory Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (4th Test)</td>
<td>98%</td>
<td>126%</td>
</tr>
<tr>
<td>nChain(400,400)</td>
<td>108%</td>
<td>119%</td>
</tr>
<tr>
<td>simpleChain(4000)</td>
<td>10605%</td>
<td>145%</td>
</tr>
<tr>
<td>nPerBody(7500)</td>
<td>164%</td>
<td>164%</td>
</tr>
<tr>
<td>theoryWidth(4000)</td>
<td>166%</td>
<td>206%</td>
</tr>
<tr>
<td>evenOdd(4000)</td>
<td>152%</td>
<td>177%</td>
</tr>
<tr>
<td>doubleChain(4000)</td>
<td>110%</td>
<td>127%</td>
</tr>
<tr>
<td>treeThree(10)</td>
<td>133%</td>
<td>102%</td>
</tr>
</tbody>
</table>

### 7. Discussion

It is obvious that the temporal logic engine, in general, requires greater memory and takes more time in order to process the same rules. The overhead is between $1 \frac{1}{2}$ times slower to being roughly the same. The peak memory required is a little over 2 times.

There are some obvious outliers, however. The “SimpleChain” test is 10,000 times slower than the standard engine. This is largely due to the small amount of processing required of the rules generated and most of the time is taken determining if the rules are applicable rather than processing them in the underlying engine.

The other discrepancy occurs for the “doubleChain”, “nChain” and “All” tests. These tests require 25% more memory than the defeasible engine but have roughly the same runtime. This is largely because of optimisations available by the Java runtime environment and that checking if a rule is applicable is quite small in comparison with the overall processing of large numbers of rules.

### 8. Future Work

#### 8.1. Persistent Rules and other Extensions

The current temporal engine fully supports persistent temporal terms but it only supports expiring terms on the right hand side of rules only. Modifications are required to support these rules on the left hand side. For an example of the problem consider the rule:
a:2 → 7 b:3. This indicates that if “a” is proven at times 0 to 2 then “b” is true for 3 time units at the time “a” is true plus 7 (from 7 to 10). As the body is active from 0 to 2 and the head from 7 to 10, there is no overlap and as the system is currently implemented it is never considered active.

The system also fails to represent expiring literals that begin with an offset. This is a combination of the two values, which can be represented as \( l @ t_1 : t_2 \). This is a literal that becomes active at time \( t_1 \) that is valid for \( t_2 \) time units. The following diagram represents an temporal defeasible theory with an example of a literal that is an expiring literal with an offset, literal “a3” which would be represented as “a3@4:3”. Expiring literals on the left hand side have an implicit 0 for their offset and as discussed previously, expiring literals on the left hand side inherit their offset based on delay between the two.

![Figure 5 - Example of Temporal Theorem with Expiring Literals](image)

Also, while the current rule engine does provide similar performance over large rule sets, it does have an increased runtime over small to medium rule sets and incurs up to 2 times the amount of memory required.

The best way to overcome both problems is to only have one set of rules and literals that are used. The easiest way to make this modification would be to combine the checking of available rules and literals into the standard defeasible engine. This would not incur any restrictions for standard theorems because, as discussed above, normal defeasible theorems are simply temporal theorems with persistent literals and rules without any delays between the firing of these rules.

**8.2. Creating a Ruby Internal Domain Specific Language**

The Javascript DSL has some advantages in its deployment within Web browsers and Java 6. However, the language itself does not support the best possible features in order to develop a
user friendly internal DSL. There are other languages that are probably more suitable for development of an internal DSL.

For example, Ruby has many language features that make it a better choice to develop an internal DSL:

- Operator overloading – Methods on objects can use symbols so methods can be called “=>”, “->” and “>” rather than “usually”, “are” and “isMoreSpecific” that is currently being used by the Java/Javascript DSL.
- Missing Methods – When passing parameters to methods parenthesis are optional.
- Matching Methods – Using regular expressions to match method names, ruby can implement features such as S-Expressions and pattern matching which can be used to create languages such as Logo within Ruby\(^3\).

Ruby can also be successfully deployed using .NET or Java using JRuby or IronRuby implementations – these can leverage existing APIs and deployment situations.

9. Conclusion

Adding a temporal dimension to defeasible logic offers a large increase in the expressiveness of a defeasible theory. The system developed successfully added key temporal features including: temporal literals (which expire after a given time period) and permanent literals (which become valid after a given time).

Two internal domain specific languages were implemented. These provide a simpler, more expressive means for developers and end-users to write rules. The integration of the Javascript DSL was used to provide a graphical user interface for end users to enter rules and for them to see the results of these rules over time.

The implementation developed shows that a simple approach of testing the entire set of rules at each time period can provide a high performing system. However, the current implementation requires a significant amount of extra memory and complexity to provide this extra functionality. The most promising solution to reduce the resources required is the integration of temporal rules into a single engine.

---

\(^3\) [http://www.artima.com/rubycs/articles/patterns_sexp_dslsP.html](http://www.artima.com/rubycs/articles/patterns_sexp_dslsP.html)
References

[40] T. Sloane, M. Mernik, and J. Heering, “When and how to develop domain-specific languages,” 2003,