有效率的知識擷取方法：基於模型驅動軟體發展的同步機制

An Efficient Algorithm of Knowledge Extraction for the Synchronization of Model-Driven Software Development

鍾峰宜
逢甲大學資訊工程學系
albertchung.tw@gmail.com

楊東麟
逢甲大學資訊工程學系
dlyang@fcu.edu.tw

摘要

在軟體發展的過程中，以模型為驅動的軟體發展架構(Model-Driven Architecture)是目前最受重視的發展方法，它在不同階段以不同的觀點去描述模型的規格，而當模型的演進到另一個觀點所需的規格時，除了目標模型(Target Model)所需的特定規格之外，也會因分析者更完善的觀察及改善(Refine)而使得分析出當初來源模型(Source Model)不足的物件，而這原屬於來源模型物件的部分，常因為進入到目標模型的分析而無法同步於來源模型規格，而且，目前並無一個有效的機制將這整個軟體發展過程中的領域知識(Domain Know-How)，轉換成結構化的知識表現方式，來做為模型發展提升品質的參考。基於此，本論文發展出一個方法，基於一個有效的模型同步機制及知識規則擷取演算法，並透過發展以規則為集合的規則知識庫(Repository)來分別儲存模型同步及知識規則資訊，藉由這個規則知識庫可以進一步的去維護模型規格及規則，以確保模型的可用性及可維護性。

關鍵詞：規則知識庫，RETE 演算法，模型驅動架構，可延伸標記語言轉化資料交換格式，模型轉換樣式

Abstract

In the software development process, the Model-Driven Architecture is currently the most attractive methodology. It describes model specifications with specific views in different stages. In addition to the specific specifications of a target model, after more complete observations and refinements from system analysts, the required objects can be thus identified when a model evolves to another specification under another view. The source model objects usually could not synchronize with source model specifications due to transforming to another target model. When developing software, there is no efficient mechanism to transform and present the structured domain knowledge and provide references for quality improvement. Therefore, in this research we develop a methodology to separately store the information of model synchronization and knowledge rules by using an efficient model synchronization mechanism and a knowledge extraction algorithm. With this rule knowledge repository, one can further maintain the model rules and specifications to ensure the model usability and maintainability.

Keywords：Rule-Set Repository，RETE，MDA，XMI，Model Transformation Pattern

一、簡介

以模型為驅動的軟體發展架構(Model-Driven Architecture)是由 OMG 組織所訂定的標準[1]，它的目的在於從分析三個不同的觀點來分別描述軟體規格，分別是運算獨立模型(Computing Independent Model, CIM)、平台獨立模型(Platform Independent Model, PIM)、及特定平台模型(Platform Specific Model, PSM)[2]，藉由 PSM 的規格讓程式發展人員得以遵從而進一步將軟體實作出來。而從某觀點所發展出來的模型，會因另一個觀點的規格需要而逐步演進成另一個模型的完整規格(例如：從 PIM 演進到 PSM)，而這個演進的過程稱之為模型的轉換(Transformation)過程。

在模型的轉換過程中，其轉換規格往往是依據目標模型(Target Model)的實際規格需要，以及為了達成這個實際規格所需的運算過程描述，而這樣的模型規格轉換描述除了明確的描述(Specify)出模型變遷(Transition)過程的轉換規格之外，更進一步的透過這個規格轉換的描述來得到軟體發展的可維護性目的。

然而，在一個正向工程的反覆式(Iterative)軟體發展程序中[3]，當模型演進到另一個觀點所需的規格時，除了目標模型(Target Model)所需的特定規格之外，也會因分析者在這個觀點更完善的觀察(Refine)，而分析出當初來源模型(Source Model)不足的物件，而這原屬於來源模型物件的部分，常因

為進入到目標模型的分析以及沒有一個良好的反饋(Feedback)機制，而無法同步來源模型規格，而且，軟體所發展的領域知識(Domain Know-How)通常是隱性的描述於模型的結構之中，而這些領域知識也缺乏一個有效的演算法來將這些寶貴的知識擷取出來。

在這個研究中，我們透過一個規範良好的模型同步機制，將模型在演進過程中，去同步二個異質觀點且具有因果關係的模型規格，並設計一個模型發展知識規則擷取架構來動態的擷取模型的規則知識。並透過這個架構，可以進一步的去維護模型規格及規則，以確保模型的可用性及可維護性。在這個研究中，我們可以達成以下目標：

(1) 解決異質觀點模型規格間，但具有因果關係物件不同步的問題。

(2) 提供一個有效的模型發展知識規則擷取(Rule Extraction)演算法。

(3) 發展一個可維護的規則知識庫。

在本論文的結構中，首先，在有關模型轉換策略及模型轉換描述的相關研究，我們在「背景和相關研究」中描述，接下來我們提出的應用一個模型發展知識規則擷取架構來處理不同模型卻具有因果關係的物件同步方法及模型規則知識擷取演算法，是在「模型發展知識規則擷取架構」這一節中討論，然後，我們依據所設計好的架構，實際針對模型的同步管理及規則知識擷取做實驗證明，這部分我們在「實驗與結果」這節作說明，最後，我們針對實驗的結果做結論，並對於以後可以改善的地方作討論，這部分是在「結論及未來研究方向」這個章節陳述。

二、背景和相關研究

目前軟體發展過程中，對於在不同觀點的模型規格同步應用中，大部分的應用還是著重在模型與程式碼同步及同質觀點的模型同步這個處理上[15]，事實上，不管在任何觀點所發展出來的模型規格，它都必須與上一個階段演進而來的模型做適當的同步處理，才能實際反應整個軟體發展過程中的一致性與連慣性，這樣的好處，當一個成功完整發展出來的軟體設計規格，它可以完整的呈現出整個軟體所代表的商業邏輯規則及每一個物件的演進過程，進一步的成為企業的重要資產。

以模型為驅動的軟體發展架構規格被嚴謹的定義在國際物件管理組織學會(OMG)，這個架構明確的定義出三個不同觀點來描繪出軟體從商業邏輯分析到實作的完整規格，並藉由這樣的模型演進過程描述，進一步的取得可維護性及可再用性(Reuse)，而不同觀點模型之間的變遷過程稱之為模型的轉換(Transformation)，而 MDA 也針對這樣的作法提出數種轉換策略來建議採用這種發展架構的開發者參考，讓軟體開發者順利從一個觀點的模型順利變遷成為另一個觀點的模型規格。

2.1 以轉換樣式(Transformation Pattern)作為模型轉換規格描述技術

不同觀點的模型規格變遷考量除了所涉及與相關技術的方法論外，往往還涉及開發者對於這個領域知識(Domain Know-How)的考量，讓模型的規格更完整，更符合軟體不管是邏輯上的或是實作上的需求。

不過，這部分專家的知識往往是屬於隱性的，不容易於建構模型需求規格或是其他設計規格文件被描述，所以會去發展一個模型轉換樣式(Transformation Pattern)來做為不同觀點之間的轉換描述，一方面作為軟體發展的標準發展樣版，另一方面將專家的知識描述在這個轉換樣式中，成為顯性知識，不但得以維護，更可以進一步成為企業的實體資產。

在軟體發展模型轉換樣版的應用中，大致區分為三大類[4]：1)直接透過來源模型內部的操作程序去轉換成一個目標模型，2)透過匯出的方式，就來源模型規格先匯出到中間層的規格描述，藉由這個中間層的規格描述提供給外部轉換工具做為模型轉換所需的來源規格，3) 經由支援轉換語言的描述與運算，能將來源模型規格精確的轉換成目標模型所需的規格。本研究中，則採用發展模型轉換語言去定義轉換樣式對於模型之間的轉換規格的描述。

將目標模型規格描述於模型轉換樣式之中，雖然可以正確的產生目標模型所需的規格並加以再使用(Reuse)，不過，當二個模型同步在異動時，反饋(Feedback)控制機制相對顯得複雜而難以維護，而使得整個軟體發展過程中，具有因果關係的模型規格可能會產生不一致的現象。

2.2 XMI 標準的限制

XMI(XML Metadata Interchange) [10]是一種結合 XML(Extensible Markup Language)與 UML(Unified Modeling Language)[11]的標準，相關規格目前在 OMG 這個組織有嚴謹的定義，它常被廣泛應用在以物件導向方法論的軟體發展中，做為一個開放的資訊交換模型，它可以完整描述以 UML 所發展的模型規格。

事實上，XMI 是描述 UML 模型中物件的相關規格，但它並無法有效的描述二個 UML 模型物件之間的演化(Evolution)及相依關係，而二個具有演化關係的 UML 模型，往往需要更多額外的註解來說明彼此之間的演化過程，而軟體分析師也無法經由直覺化的觀察來歸納彼此之間的規則關係。

2.3 RETE 演算法目前的應用

RETE 演算法最早是由 Forgy 在 1979 所提出

[5]，目前廣泛的被應用在人工智慧中有關邏輯推理過程的部分，因規則生成系統(Production System)或稱之為規則知識庫系統(Rule-Base System, RBS)，所採用的 Naïve 演算法因在推論過程所產生大量的規則組合的一種生成系統改良演算法。RETE 演算法是利用元素串列(List of Element)的方式來避免在工作記憶體(Work Memory)重複比對的問題，及利用樹狀結構排序網路(Tree-Structure Sorting Network)來避免規則記憶體(Rule Memory)在重複比對的問題，之後在相關研究提出改良平行處理或是以線性方式來降低時間複雜度的 RETE 演算法[6][14]。

近年來，RETE 演算法則被應用在軟體發展過程中是在程式碼與商業邏輯規則管理的議題之上，特別是以 Java 為軟體建置語言上的應用[7][12]，學者認為，透過商業邏輯的有效管理，則可以有效的管理相對應的程式碼，提升軟體的可維護性，及滿足千變萬化的商業邏輯需求。為了實現這部分理論，JCP(Java Community Process)則在 2004 年完成了 JSR-94 標準的制訂[8]，這個標準所呈現的也就是 Java Rule Engine API 來讓以 Java 語言所發展的軟體可以實現程式碼與商業邏輯規則同步的理論。

然而，在軟體發展分析的過程中，並非只有商業邏輯的蒐集、滿足、及建置，它是一個包含各個層次的分析建置成果，所以，這樣的應用大致滿足部分的軟體功能建置時的需求，及解釋並維護軟體的建置規格(程式碼部分)，但是並無法滿足在軟體發展過程中因不同觀點所發展出數個模型具有因果關係物件之間同步的問題。

三、模型發展知識規則擷取架構

基於軟體發展過程中模型的演進會因發展者的維護而造成來源模型與目標模型之間具有因果關係的規格不一致情況，本研究提出了一個以 XMI 格式為交換基礎的模型同步引擎來確保異質觀點下模型規格的同步，及設計一個有效的規則建立引擎，擷取出結構化的規則知識表現方式的處理架構，如圖 1 所示。

經由這個架構的處理機制來達成異質觀點模型規格同步及可維護知識的目的，並透過一個規則知識庫(Repository)的儲存機制，將模型同步及規則知識的資訊作有效率的存放，以達成整個架構的可用性及可維護性目的。我們將整個架構的處理方法區分為三個層次，分別是：模型轉換層、運算層、及儲存層。

模型轉換層，最主要是處理二個異質觀點的模型轉換處理，它經由一個模型轉換樣式從來源模型轉換到目標模型規格，並產生模型的資訊交換所需的 XMI 檔案；運算層，最主要是處理有關異質觀點模型之間規格的同步機制，以及模型規格的規則知識擷取處理；儲存層，最主要是處理模型轉換層及運算層處理的輸出資訊，藉由這個機制，可以進一步支援模型轉換層及運算層所需運算的相關資訊，以及提供操作及維護的相關服務。每個層次的架構及處理細節原理，我們分別討論及說明如下:

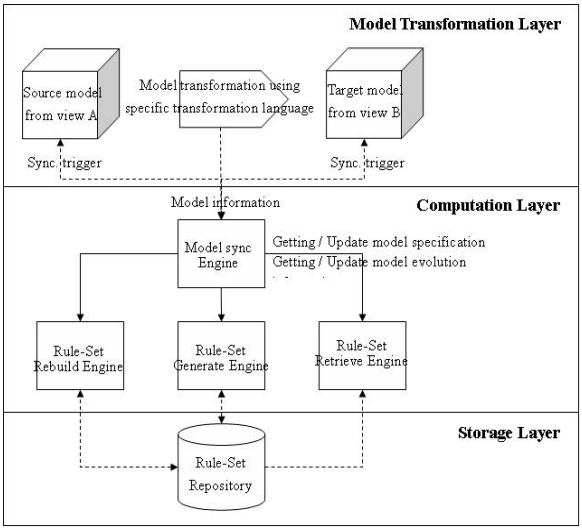

圖 1 模型發展知識規則擷取架構圖

3.1 模型轉換層設計

在過去的研究中[9]，我們透過特定的模型轉換語言(Transformation Language)設計模型轉換樣式(Transformation Pattern)順利的將來源模型轉換成目標模型所需的規格，現在我們延伸這個應用，將模型轉換的過程描述在轉換樣式中，而來源模型及目標模型規格的規格描述，本架構中則是透過 XMI 來做為資訊交換格式。

我們會藉由這個交換格式來判讀模型中的物件的特徵(Feature)、屬性(Attribute)、及關係(Relationship)，來當成規則運算層所需的輸入來源，並藉由這個模型資訊交換標準來做為模型彼此之間的溝通及運算基礎。

3.2 規則運算層設計

這個層級的設計最主要是發展一個有效的機制來確保不同觀點但具有因果關係的模型規格同步，因此，我們發展一個模型同步引擎機制來動態的確保模型規格的同步，並開發一個規則建立引擎來萃取出模型發展知識規則，以確保在模型設計過程中(Design-Time)的同步。

3.2.1 模型同步引擎(Model Sync Engine)設計

模型中物件的規格，首先經由一個模型同步引擎(Sync Engine)依據模型所要求的動作(模型規格的擷取或更新)，來做適當的運算，整個處理架構

如圖 2 所示。

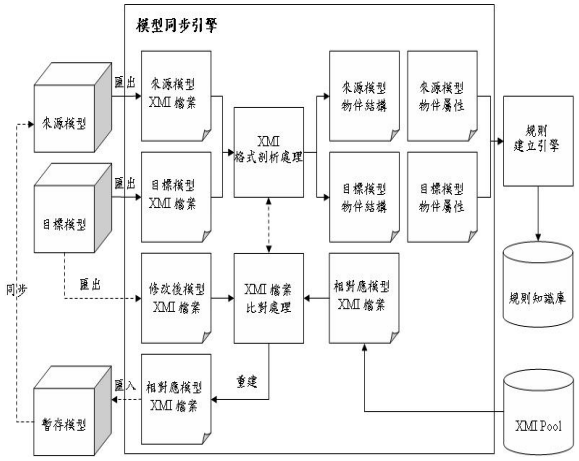

圖 2 模型同步引擎架構圖

在模型規格擷取的運算中，模型同步引擎是透過一個 XMI 剖析運算器來剖析 XMI 檔案，這個 XMI 剖析運算器除了剖析 XMI 檔案結構，最主要是將剖析後的結果區分為結構與屬性二個部分，結構是為了快速建立 RETE 網路的前置處理，而屬性是為了模型物件規格同步的比對前置處理，而這二個部分所擷取到的模型資訊如表 1 所示。

表 1 擷取 XMI 檔案資訊分類表

物件結構	物件類型(Type) 識別代碼(Code) 通用唯一識別碼(UUID) 多樣屬性(Multiplicity) 組合(Association)關係 特別化(Specialization)關係 一般化(Generalization)關係 具體化(Realization)關係
物件屬性	物件屬性值(Attribute) 操作(Operation) 標籤(Label) 識別代碼(Code) 模版(Stereotype)

在模型規格更新部分，修改過的模型規格經由 XMI 檔案匯出，透過一個 XMI 檔案比對處理機制，從一個 XMI Pool 中擷取該模型本身所涉及到的相關連的模型 XMI 檔案進行比對，XMI 檔案比對處理機制會依據該模型相對應的關連模型差異規格產生一份新的相對應模型差異規格 XMI 檔(觸發該動作的模型可能是某個模型的目標模型及另一個模型的來源模型角色)，藉由這一份相對應模型差異規格 XMI 檔案內容去重建出一個暫存模型，再進一步的同步到相關連模型規格中，達到模型設計同步的目的。

3.2.2 規則建立引擎(Rule-Set Generation Engine)設計

在規則建立引擎設計部分，最主要是將 XMI 剖析運算器所運算出有關結構屬性的部分，依據 RETE 演算法，將結構屬性運算出以 RETE 樹狀結構來表現的規則結構，並藉由這個規則結構來分別呈現二個具有因果關係的 UML 模型之間的演化規則。規則建立引擎的架構如圖 3 所示。

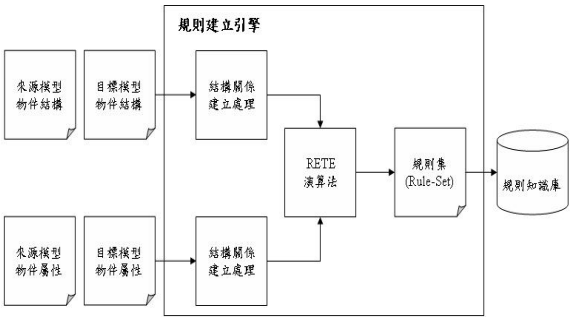

圖 3 規則建立引擎架構圖

以類別圖(Class Diagram)為例，其所描述的是 UML 模型結構的規格，藉由模型同步引擎所前置處理後的 XMI 的格式，來產生 UML 模型關連(Relationship)結構規格，再經由這個透過 RETE 演算法來建立 RETE 規則樹狀結構，基於 UML 模型本身結構規格的描述，我們依據模型中關連的多樣屬性(Multiplicity)區分為正相關、模糊相關、及循環相關三種關連結構，來分別修正 RETE 演算法，以符合模型知識中規則的擷取。

在這個方法中，為了資料庫可以正確且順利的做存取處理，我們則將規則運算以特定的符號來表示 (如表 2 所示)。

表 2 規則運算符號對照表

( )	具有運算優先權
^	AND
|	OR
→	THEN
↓	強度遞減(生成元素中包含 OR None)

3.2.2.1 正相關規則樹狀結構建立

二個具有正相關的類別，其組合(Association)關連中的多樣屬性(Multiplicity)是基於(1..1)或是(1..*)的關係，表示這二個類別物件具有絕對的相依(Dependency)關係，其規則樹狀結構建立如圖 4 所示，而規則的描述方式如下：

(A | B) ^ C → D

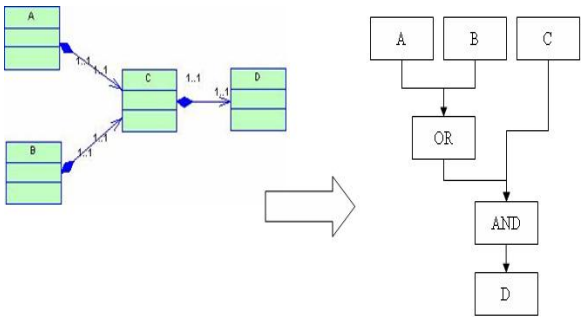

圖 4 具有正相關的規則樹狀結構

3.2.2.2 模糊相關規則樹狀結構建立

二個具有模糊相關的類別，其組合(Association)關連中的多樣屬性(Multiplicity)是基於(0..1)或是(0..*)的關係，表示這二個類別物件具有參考(Reference)關係，其規則樹狀結構建立如圖 5 所示，而規則的描述方式如下：

(B ^ C) | A → D

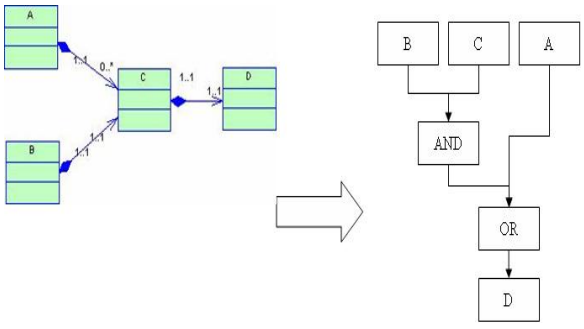

圖 5 具有模糊相關的規則樹狀結構

3.2.2.3 循環相關規則樹狀結構建立

在一組類別物件的結構中，可能存在著彼此之間的循環參考或是相依的關係。這種結構下，RETE 樹狀結構並不容易建立，因此，我們提供一個修剪 RETE 樹的方法：

步驟一:首先，尋找出具正相關且(AND)造成循環相關的物件組合，修改成為輻射結構關連關係(如圖 6 所示)。

步驟二:依據樹末端的物件開始尋找正相關的物件路徑。

步驟三:修剪具有循環相關中距離樹末端最遠的模糊相關關係(如圖 7 所示)。

步驟四:建立步驟二、步驟三所修剪後的 RETE 樹狀結構。

步驟五:建立被修剪關係物件到距離該物件最短路徑的正相關物件 RETE 樹狀結構。

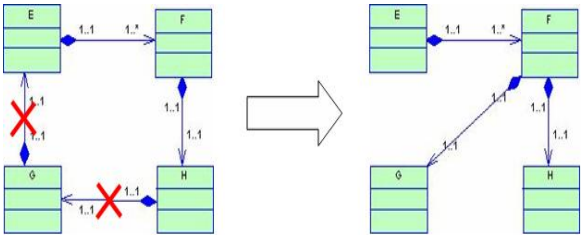

圖 6 修改具有正相關且(AND) 具有循環相關的規則樹狀結構

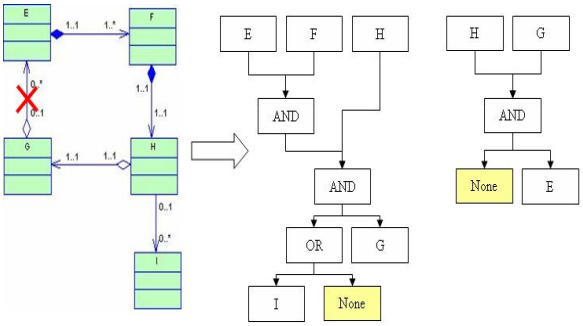

圖 7 修剪具有循環相關的規則樹狀結構

依據這些步驟，我們分別針對修剪後的 RETE 樹狀結構建立以下規則：

RULE 1：

(E ^ F) ^ H → G | I↓

RULE 2：

H ^ G → E↓

當模型發展過程中進一步的演進時，我們則可以透過模型同步引擎所提供的同步後 XMI 檔案，並經由這個引擎來重建 RETE 樹，而不同觀點的模型規格我們會分別建立二棵 RETE 規則樹狀結構，來分別建立不同觀點之下模型發展的知識規

則，如圖 8 所示。

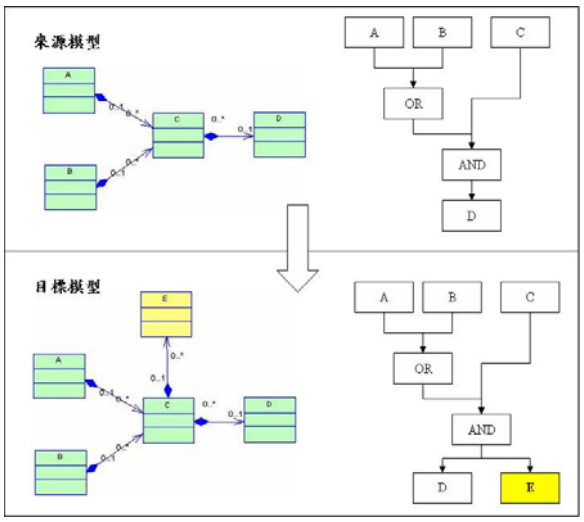

圖 8 不同觀點模型分別建立 RETE 規則樹

由圖 8 中我們可以看出，來源模型與目標模型的差異性在於目標模型多了一個 E 這個類別，這個 E 類別是來自於模型轉換樣式依據目標模型規格所規範的規格所產生的，它所代表的是設計這個目標模型所在的觀點知識規則，假設這個 E 類別是因為分析師在來源模型所忽略屬於來源模型觀點的規格，那麼我們則透過模型同步引擎去同步回來源模型，並重新匯出 XMI 檔案，再重建來源模型的 RERE 規則樹狀結構，以確保整個模型與 RETE 規則樹狀結構發展的同步。

3.3 儲存層設計

此層次最主要是設計一個規則儲存機制來記錄運算層所執行的結果，並提供維護時所需的相關資訊，就本架構中，我們所要處理的資訊大致分為三大類，如表 3 所示。

表 3 規則知識庫儲存資訊分類表

模型觀點資訊	CIM，PIM，PSM 觀點
模型規格資訊	XMI 檔案
RETE 規則資訊	規則集(Rule-Set)

透過這個儲存機制，一方面可以實現整個同步架構所需的運算相關資訊，另一方面則提供了資訊維護的一個可操作機制。

四、實驗與結果

基於這個架構，我們採用實驗的環境包括：電腦輔助軟體工程工具、XMI 匯出引擎、及一個關連式資料庫當成建立我們的規則知識庫。

4.1 模型轉換處理

我們先透過一個電腦輔助軟體工程工具的輔助，快速的發展出一個特定應用的 UML 模型規格，如圖 9 所示。

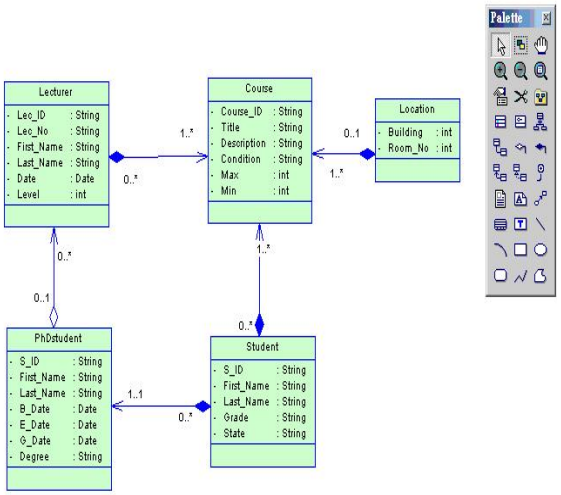

圖 9 來源模型類別圖規格

接下來，我們透過一個特定的模型轉換語言所設計的模型轉換樣式(Transformation Pattern)順利的將這個模型轉換成特定的目標模型，如圖 10 所示。

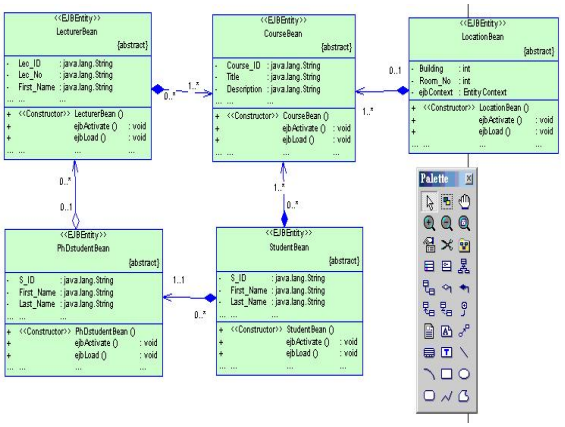

圖 10 目標模型類別圖規格

4.2 模型規格運算處理

轉換成目標模型規格之後，我們則將來源模型

與目標模型分別以 XMI 匯出引擎匯出成為 XMI 檔案。依據這一份 XMI 內容，XMI 剖析運算器會將來原模型的 XMI 檔案屬性值加寫『xmi：label = “source”』，在目標模型的 XMI 檔案屬性值加寫『xmi：label = “target”』，以方便異動後模型同步比對時的參考標籤。

然後將 XMI 文件中以 <UML：Class>與</UML：Class>範圍的本文部分(如圖 11 所示)，轉換成為描述物件的屬性，並將以 <UML：AssociationEnd> 與</UML：AssociationEnd>範圍的本文部分(如圖 12 所示)，來描述每一個類別的物件的結構。

```
<UML:Class name="Student" isLeaf="false" xmi.id="{6107DAC0-CED0-4CBC-970E-2B5A5572AE12}"
    <UML:Classifier.feature>
        <UML:Attribute name="S_ID" xmi.id="{7CE7A092-AF52-4EF3-8C48-376FB96B036B}" owner
            <UML:StructuralFeature.type>
                <UML:Classifier xmi.idref="DttpO"/>
            </UML:StructuralFeature.type>
        </UML:Attribute>
        <UML:Attribute name="First_Name" xmi.id="{FD57094E-D1B4-44DB-98F5-F94DFFAEB30A}"
            <UML:StructuralFeature.type>
                <UML:Classifier xmi.idref="DttpO"/>
            </UML:StructuralFeature.type>
        </UML:Attribute>
        <UML:Attribute name="Last_Name" xmi.id="{56A562FF-4D75-4A57-BD38-0A8109AD0F57}"
            <UML:StructuralFeature.type>
                <UML:Classifier xmi.idref="DttpO"/>
            </UML:StructuralFeature.type>
        </UML:Attribute>
        <UML:Attribute name="Grade" xmi.id="{551FEEC0-C87C-428A-91B3-A0194AD4BDE9}" owne
            <UML:StructuralFeature.type>
                <UML:Classifier xmi.idref="DttpO"/>
            </UML:StructuralFeature.type>
        </UML:Attribute>
        <UML:Attribute name="State" xmi.id="{7F8701F5-84DE-4E9D-97B2-B2F58D8C0B5C}" owne
            <UML:StructuralFeature.type>
                <UML:Classifier xmi.idref="DttpO"/>
            </UML:StructuralFeature.type>
        </UML:Attribute>
    </UML:Classifier.feature>
</UML:Class>
```

圖 11 描述物件屬性的 MXI 檔案內容

```
</UML:Association>
    <UML:Association.connection>
        <UML:AssociationEnd xmi.id="{0E7E5194-4CF3-4114-AEAD-E927A2F915B2}a" visibility="public">
            <UML:AssociationEnd.aggregation xmi.value="composite"/>
            <UML:AssociationEnd.changeability xmi.value="changeable"/>
            <UML:AssociationEnd.isNavigable xmi.value="false"/>
            <UML:AssociationEnd.ordering xmi.value="unordered"/>
            <UML:AssociationEnd.type>
                <UML:Classifier xmi.idref="{AC571A47-8D64-4EB8-8CFD-586BFBE13507}"/>
            </UML:AssociationEnd.type>
            <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                    <UML:Multiplicity.range>
                        <UML:MultiplicityRange>
                            <UML:MultiplicityRange.lower>0</UML:MultiplicityRange.lower>
                            <UML:MultiplicityRange.upper>1</UML:MultiplicityRange.upper>
                        </UML:MultiplicityRange>
                    </UML:Multiplicity.range>
                </UML:Multiplicity>
            </UML:AssociationEnd.multiplicity>
        </UML:AssociationEnd>
        <UML:AssociationEnd xmi.id="{0E7E5194-4CF3-4114-AEAD-E927A2F915B2}b" visibility="public">
            <UML:AssociationEnd.aggregation xmi.value="none"/>
            <UML:AssociationEnd.changeability xmi.value="changeable"/>
            <UML:AssociationEnd.isNavigable xmi.value="true"/>
            <UML:AssociationEnd.ordering xmi.value="unordered"/>
            <UML:AssociationEnd.type>
                <UML:Classifier xmi.idref="{99B66A16-200C-43B9-B9A6-31EZBE3D1706}"/>
            </UML:AssociationEnd.type>
            <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                    <UML:Multiplicity.range>
                        <UML:MultiplicityRange>
                            <UML:MultiplicityRange.lower>0</UML:MultiplicityRange.lower>
                            <UML:MultiplicityRange.upper>-1</UML:MultiplicityRange.upper>
                        </UML:MultiplicityRange>
                    </UML:Multiplicity.range>
                </UML:Multiplicity>
            </UML:AssociationEnd.multiplicity>
        </UML:AssociationEnd>
    </UML:Association.connection>
</UML:Association>
```

圖 12 描述物件結構的 MXI 檔案內容

4.3 規則建立處理及解釋

接下來，我們可以透過 RETE 演算法來產生這個模型的知識規則，首先，我們建立並修剪 RETE 規則樹狀結構，如圖 13 所示。

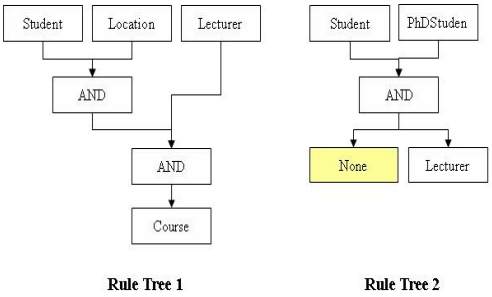

圖 13 修剪後 RETE 規則樹狀結構

接下來，我們將修剪好的 RETE 樹狀結構轉換成 XMI 檔案，並透過撰寫好擷取 RETE 規則產生器來產生規則，如圖 14 所示。

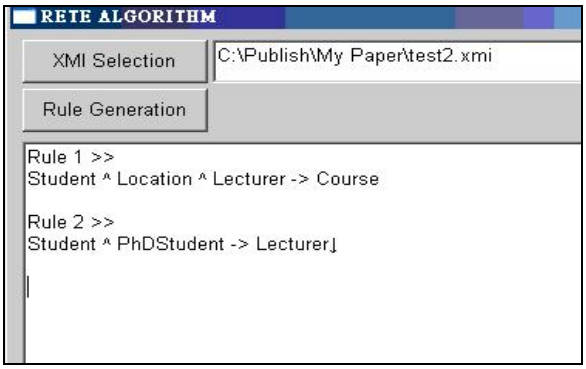

圖 14 規則產生器處理介面及輸出結果

以這個實驗所產生的知識規則中，我們可以針對規則 1 的結果解讀為：課程這個類別，它必須一定要有學生、講師、及上課地點所組成，而將規則 2 的結果解讀為：只要是博士班的學生，他有可能會具有講師身份，因此，若有關於課程的運作規則(Business Rule)，我們可以依據這個規則來驗證或是提出建議。

4.4 模型規格同步與規則重建處理

最後，我們將目標模型的規格作部分的異動，並將異動後的目標模型再次透過 XMI 匯出引擎匯出成為 XMI 檔案，並觸發 XMI 檔案比對處理，將 XMI 檔案中，規格屬於 xmi：label = “source”的規格擷取出來，並與資料庫中保留的來源檔案中的 XMI 檔案作差異性比對處理，針對差異的部分重建一個差異規格的 XMI 檔案。

我們針對這份比對後差異規格的 XMI 檔案透過電腦輔助軟體工程工具的匯入(Import)處理建立一個暫存的模型規格，並透過這個暫存模型規格與工具中已存在的來源模型規格作比對處理，如圖 15 所示。

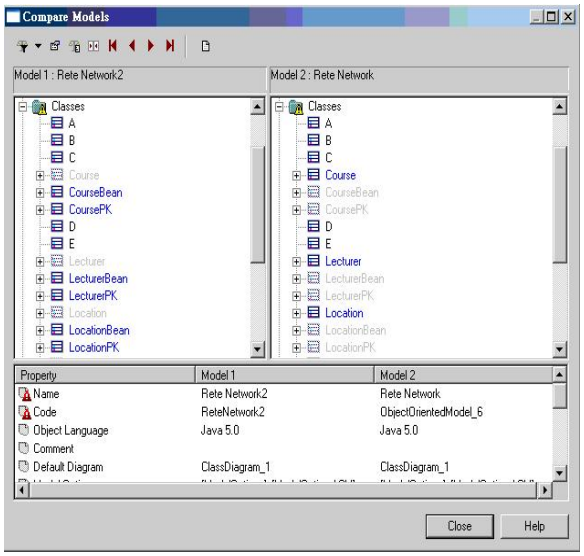

圖 15 模型規格比對管理介面

比對完成之後，依據比對報告作模型合併處理，將目標模型中，具有因果關係物件規格同步到來源模型，接下來則重複 4.2 節的處理，將資料庫中的 XMI 檔案覆蓋掉，並重建知識規則，以達到模型規格及知識規則的同步。

五、結論及未來研究方向

在過去，從不同觀點的模型發展演進過程中，具有因果關係的模型規格，以及在每一個觀點中發展出模型規格的知識，都是屬於隱性的或是非結構化的表現方式，例如：口頭說明講解、不完整或是過時的技術文件...等。本論文提供一個模型發展架構，透過這個架構所提供的機制從模型的轉換到去整合因不同設計觀點但是具因果關係的模型規格之同步，以及藉由一個有效的知識規則擷取(Rule Extraction)演算法來將模型發展的知識以顯性的、結構化的規則來表示，並透過設計良好的知識庫儲存機制來存取的一個完整處理的架構，這樣不但是提供了可操作的機制，更為未來的應用提供了一個知識發掘(Knowledge Discovery)的基本單元。

在這個架構所擷取到的模型發展知識規則，它是一個反應模型所要呈現滿足業務需求的軟體發展結構，也反映出專家對於要滿足這個業務需求所提供的解決架構，這些知識規則對於企業而言是非常有用的：

- 它將專家發展軟體的解決架構以結構化的方式呈現，進一步的成為企業的資產，不會因專家的離開而造成困難。
- 提供具有相似領域或是相似功能軟體發展時的一個結構化知識及驗證機制，來確保不會因分析人員的因素所造成的軟體品質差異性，而進一步的提升軟體發展品質。
- 當規則知識庫的內容以及所涉獵的業務領域越豐富時，企業則擁有專家的知識及決策能力，以進一步的提升企業的競爭力及能量。

針對這個研究的其他應用，我們提出了一些可以應用的方向及延伸本架構的看法：

5.1 完整支援 UML 模型規格

本研究在規則知識擷取的方法上，是以類別圖(Class Diagram)來做為討論的基礎，事實上，在 UML 模型中，除了描述模型結構的類別圖之外，還有在描述模型行為部分的互動圖(Interaction Diagram)，未來，在這部分我們將朝這方面研究，以提供完整的 UML 模型處理架構，並進一步的延伸到軟體重構(Refactoring)這個議題領域上的應用。

5.2 延伸至知識管理相關應用

未來可以參考更好的知識庫的架構設計及應用[16]來提升知識庫的機制，並實驗更多的知識規則擷取演算法，來提升本架構整體的可信度及可用度。我們將延伸這個規則知識庫的應用，來進一步做為軟體的發展行為的預測及分析，這部分，可以應用資料採礦(Data Mining)的技術來分析，例如：軟體發展階段的可信度模糊分類決策分析、軟體模型轉換關連分析…等。

5.3 支援 CMMI 相關議題

我們的架構提供了模型轉換、同步、與規則知識擷取的處理機制，因為有了這個過程的處理程序，我們可以輕易的掌握基於這個架構下的設計物件追蹤，並進一步的進行衝擊分析來確保整個軟體發展的一致性。未來，我們將基於既有的架構規格上，朝向如何去整合 CMMI 在 Level 2 的需求管理及 Level 3 的設計變更管理、組態管理、確認與驗證(Validation and Verification Model)等相關議題上，來符合軟體發展所需的品質管理政策。

六、誌謝

本研究部份經費由國科會計畫補助，計畫編號 NSC93-2213-E-035-032，特表誌謝。

七、參考文獻

[1] Object Management Group: Model Driven Architecture, http://www.omg.org/mda/.

[2] Jishnu Mukerji and Joaquin Miller, "MDA Guide Version 1.0.1," OMG June 2003, http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[3] Jacobson, I., Booch, and J. Rumbaugh, The Unified Software Development Process, Addison Wesley, 1999.

[4] Shane Sendall and Wojtek Kozaczynski, "Model Transformation - The Heart and Soul of Model-Driven Software Development," IEEE Software, pp. 42–45, Sep. / Oct. 2003.

[5] Forgy, C. L., "RETE: A fast algorithm for the many pattern/many object pattern match problem," Artificial Intelligence, 19(1), 1982, pp. 17-37.

[6] T. Ishida, "Parallel rule firing in production systems," IEEE Transactions on Knowledge and Data Engineering, vol. 3, no. 1, pp.11-17, 1991.

[7] Drools: Rules Engine Implementation Based on Charles Forgy's Rete Algorithm, http://drools.org/.

[8] The Java Business Rules Community, Java Rule Engine API (JSR 94), http://www.javarules.org/.

[9] Feng-I Chung and Zhen-Nan Chen, "A Robust Software Development Approach - Using a Heuristic Transformation Pattern Based on Case Practice," Proceeding of the First Taiwan Software Engineering Conference, June 2005.

[10] Object Management Group: XML Metamodel Interchange, http://www.omg.org/technology/documents/formal/xmi.htm.

[11] Object Management Group: Unified Modeling Language, http://www.uml.org/.

[12] Jess: the Rule Engine for the Java Platform, http://herzberg.ca.sandia.gov/jess/.

[13] Jeong A. Kang, Albert Mo Kim Cheng, "Shortening Matching Time in OPS5 Production Systems," IEEE Transactions on Software Engineering, vol. 30, no. 7, pp. 448-457, July 2004.

[14] X. Wu, "LFA: a linear forward chaining algorithm for AI production systems," Expert Systems, vol. 10, no. 4, pp.237-242, 1993

[15] Igor Ivkovic and Kostas Kontogiannis, "Tracing Evolution Changes of Software Artifacts through Model Synchronization," Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004

[16] Padmal Vitharana, Fatemeh "Mariam" Zahedi, and Hemant Jain, "Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis," IEEE Transactions on Software Engineering, Vol. 29, No. 7, July 2003