國立政治大學資訊科學系
Department of Computer Science
National Chengchi University

碩士論文

Master's Thesis

利用模型驅動技術快速產生領域專屬語言之執行與偵錯環境

Rapid Generation of Executing and Debugging Environments for Domain-Specific Languages based on Model-Driven Technology

研 究 生：趙仁鋒

指導教授：陳正佳

中華民國一〇一年一月

January 2012

利用模型驅動技術快速產生領域專屬語言之執行與偵錯環境

Rapid Generation of Executing and Debugging Environments for Domain-Specific Languages based on Model-Driven Technology

研 究 生 ： 趙仁鋒　　Student: Jen-Feng Chao

指 導 教 授 ： 陳正佳　　Advisor: Cheng-Chia Chen

國立政治大學

資訊科學系

碩士論文

A Thesis

submitted to Department of Computer Science

National Chengchi University

in partial fulfillment of the Requirements

for the degree of

Master

in

Computer Science

中華民國一〇一年一月

January 2012

利用模型驅動技術快速產生領域專屬語言之執行與偵錯環境

摘要

領域專屬語言的設計理念是希望能夠協助使用者在特定領域上解決特定問題。然而，大多數領域專屬語言的開發環境與工具均非常貧乏，這將會增加使用者在開發程式上的困難。

所以本研究利用模型導向技術來建構一套生成系統，使用者只要輸入領域專屬語言的語意以及偵錯定義，就能快速生成語言的執行與偵錯環境，並提供完善的操作介面，輔助使用者加快程式開發的速度。

Rapid Generation of Executing and Debugging Environments for Domain-Specific Languages based on Model-Driven Technology

Abstract

The purpose of creating domain-specific languages (DSLs) is to help user solve problems in a particular domain. However, most DSLs are lack of development environments and tools, which would be more difficult to develop programs.

This thesis is aimed to develop a generating system based on model-driven technology. Given the semantics and debugging definitions of a domain-specific language, the system would be able to generate an executing and debugging environment for the language with friendly user interface, thus improving efficiency and productivity of using the language.

誌　　謝

政大資科的研究生涯，讓我歷經了許多成長。學習期間幸蒙　陳正佳老師的循循善誘以及敦敦教誨，帶領我開啓學術研究的大門，一窺資訊科學的奧妙。而在研究方向的訂定、基本觀念的建立、研究目標的實現，以及論文的撰寫等各階段也都受益匪淺。

口試期間，亦承蒙淡江大學資訊與圖書館學系 歐陽崇榮老師以及世新大學資訊管理學系　鄭武堯老師的鼓勵與指教，指引學生由不同的角度探討本研究成果在未來的發展性，也指明學生論文上的疏漏之處，使得本論文可更臻完善。

在論文研究期間，非常感謝曾仲瑋學長一路的陪伴與扶持，也感謝曾在同一實驗室的學長們的協助，以及在此期間一起相互砥礪的學弟們。除此之外，也對不吝嗇給予鼓勵及幫助的老師、助教和同學們致上由衷的感謝。

最後特別要感謝我的家人，無條件給予我許多的支持與關懷，讓我能夠順利地完成學業，願以此成果與家人共享。

趙仁鋒 謹誌於

政治大學資訊科學研究所

中華民國 一〇一年 一月

目　　錄

表　目　錄

圖　目　錄

第 一 章

序論

1.1 問題動機

領域專屬語言 (domain-specific language, DSL) [1] 是開發人員爲了解決特定領域的問題所定義的專用語言。所以除了軟體工程師之外，許多毫無程式設計經驗的使用者也會利用 DSL 來解決問題。然而大多數 DSL 的開發環境與工具均非常貧乏，缺乏開發環境與工具的輔助將會降低使用者開發程式的效率。

鑑於上述情況，擁有良好的輔助工具環境例如直譯器以及偵錯器將會對使用者有莫大的幫助。但從頭開發直譯器與偵錯器需要耗費大量的時間、金錢與人力，爲了促進程式開發效率而花費太多時間去建置輔助工具也顯得本末倒置。所以，如何快速產生 DSL 的執行與偵錯環境，並提供良好的操作介面，協助使用者加快程式開發的效率，即爲本研究最主要的動機。

1.2 實現策略

爲了要達到快速並簡易地產生 DSL 執行與偵錯環境的目的，我們利用模型驅動技術

(MDA) [2] 來建構一套自動生成系統。開發人員只要定義出核心的骨幹，就能夠在不需要太多額外的前置作業或背景知識下快速打造出特定 DSL 的開發環境。

本研究首先定義一套實作正規語意並可加入偵錯定義的語言，讓開發人員可以利用此語言來描述 DSL。接著建立生成器來讀入這些描述，產生具備偵錯機制的執行元件。執行元件可被執行平台下的直譯器與偵錯器通用框架載入，以便執行與偵錯 DSL 程式。

除了執行與偵錯程式外，提供整合式開發環境也利於使用者撰寫程式。所以還需要建構一個擁有人性化介面的環境，才能有助於增進程式開發速度的目的。由於 Eclipse [3] 提供了完善的使用者介面，以及強大的偵錯平台 [4]，開發人員能夠輕易地將自製的偵錯器整合進 Eclipse 內，輔助使用者開發 DSL 程式。所以我們決定將系統整合至 Eclipse 平台，讓使用者可以在友善的整合式開發環境下執行與偵錯 DSL 程式。

1.3 特色貢獻

本研究建立一套自動生成執行與偵錯環境的系統，能將 DSL 的正規語意與偵錯定義轉換成具備偵錯機制的執行元件，此元件可由直譯器與偵錯器通用框架載入以便執行與偵錯 DSL 程式，並整合至 Eclipse 平台。本系統擁有下列特色與貢獻：

- 使用正規語意來描述語言：採用正規語意 [5] 來描述 DSL 的語意，而不是直接以程式語言實作。使開發者能在較抽象的層面上描述一個語言如何在機器上運作，而不用去顧慮其他實作的細節。除此之外，我們也更容易追蹤程式的執行流程以及驗證程式的正確性。
- 抽象化偵錯機制：開發者不用一一實作每個偵錯元件或是功能，只要在 DSL 的語意內加入偵錯定義，即可讓生成的執行元件擁有基本的偵錯機制，能夠偵錯 DSL 程式。

- 提供人性化開發環境：本系統建構在 Eclipse 平台之上，以便提供人性化的開發環境讓使用者撰寫程式。使用者偵錯程式時，也可利用 Eclipse 提供的偵錯介面來逐步追蹤程式的狀態。

1.4 章節架構

首先在第一章內，我們簡介本研究的動機、策略與貢獻。第二章則介紹相關的背景知識、工具以及相似系統，此章首先介紹正規語意，再探討編譯器與直譯器生成工具。接著會描述使用者偵錯介面，其中包括了 Eclipse 與其偵錯平台，並介紹語言開發框架 (framework)，最後則是幾個與本研究相近的系統分析與比較。

第三章分析本系統的架構與輸入模型，並對系統各部分詳加介紹。第四章闡述如何建構執行與偵錯環境生成系統。第五章用範例實證本研究的成果。第六章則是結論與未來展望。

第 二 章

相關研究探討

本研究所需的背景知識、相關工具以及相似系統，可分爲下列幾個部分詳細探討：正規語意、編譯器與直譯器生成工具、使用者偵錯介面、語言開發框架以及相似系統分析與比較。首先，2.1 節概述作爲本系統輸入模型的正規語意。2.2 節會介紹編譯器與直譯器的相關生成工具。接著在 2.3 節，針對 Eclipse 所提供的偵錯平台 (debug platform) [4] 作一個簡介。然後在 2.4 節中介紹語言開發框架。最後在 2.5 節中，介紹與本研究同樣都架構在 Eclipse 上的相似系統，並比較各系統的差異。

2.1 正規語意

在程式語言中，正規語意 (formal semantics) 是以嚴謹的數學方法來研究還有並定義程式的意義 [6]。使用正規語意不僅利於各種分析與驗證，也能精確地定義程式的結構。常見的正規語意有操作語意 (operational semantics)、指稱語意 (denotational semantics) 以及公理語意 (axiom semantics) 三種。其中的操作語意 [5] 藉由數學的推論系統來描述程式在機器上執行時的狀態，讓我們不僅獲得執行的結果，還能夠知道程式在執行的過程中，狀態如何遞變。而操作語意又包含結構操作語意 (structural operational semantics, SOS) [7] 與自然語意 (natural semantics, NS) [8] 兩種方法。結構

操作語意又稱 small-step semantics，因爲它關注於每個獨立計算步驟該如何執行；而自然語意又稱 big-step semantics，因爲它著重在如何獲得整體的計算結果。

因爲正規語意使用嚴謹的數學方法來定義程式的意義，所以採用正規語意描述程式比較能確保程式的正確性。再加上操作語意比較貼近語言的實作，且自然語意也容易定義程式的意義。所以本系統將採用自然語意來描述 DSL 的語意並作爲輸入模型。

2.2 編譯器與直譯器生成工具

編譯器 (Compiler) 是一個可以讀入原始語言 (source language)，並將它轉換成另一種標的語言 (target language) 的程式 [9]。然而轉換的過程需要經過許多繁複的階段與轉換，包括字彙分析、語法分析、語意分析，生成中間碼、程式碼最佳化、產生標的碼等等。直譯器雖不會產生標的碼，而是直接執行原始碼所描述的操作。但大部分的情況下，還是需要對原始碼進行字彙分析與語法分析，將它轉換成語法樹以方便直譯器解譯。

編譯器與直譯器的實作不但複雜且容易出錯，因此許多的生成工具因應而生。這些生成工具藉由讀入語言的規格描述，即可生成相對應的分析器。例如 Lex [10] 是以正規表達式當做輸入的字彙分析生成器，以及用文法描述當做輸入的 Yacc [10]語法分析生成器等等。而生成語意實作的工具也不少，如 Relational Meta-Language [11]、k-framework [12]、ASF+SDF [13]、Stratego/XT [14]、TXL [15] 等等。由於本研究希望以自然語意爲輸入模型，其中 RML 雖然爲實作自然語意的語言，但並不包含偵錯器的生成。所以本研究決定採用 RML 的文法形式並加以擴充、修改來當作輸入語法，最後再加入偵錯定義，以便能夠生成直譯器以及偵錯器。

2.2.1 Relational Meta-Language (RML) And Tools

RML 是一套將自然語意編譯成高效率的可執行碼的系統，藉此實作語言的語意。自然語意是以 Plotkin 的結構操作語意 [7] 爲基礎並由 INRIA 的 Kahn [8] 延續發展出來。它的規格包含了資料型態宣告以及規論規則 (inference rule) 的集合。推論規則是以類似 Gentzen 的 Sequent Calculus 格式描述物件之間的關聯 (relation) 來進行自然推論 (natural deduction) [16]。

RML 不但容易使用且提供了高度抽象能力來表達語言轉換與分析工作。雖然已有 TYPOL [17] 可生成自然語意的實作，但卻缺乏效率。而 RML 提供了一個表達自然語意的強型超語言生成工具— rml2c ，足以產生約略手寫般高效率的 C 語言實作 [18]。

2.3 使用者偵錯介面

本節將介紹 Eclipse 及其底下的偵錯平台。

2.3.1 Eclipse Debug Platform

Eclipse 是一套開放原始碼的程式整合式開發環境 (IDE)，其可擴充的框架，允許第三方撰寫名爲 plug-in 的擴充套件，將既有的或是新開發的工具程式整合至此平台。Eclipse 在 plug-in 機制中定義了一組稱作延伸元件 — 延伸點 (extension - extension point) 的概念，延伸元件代表延伸出來的功能，而延伸點則代表允許被延伸的功能 [19]。如果將延伸元件比擬成插頭，那麼延伸點就如同插座。圖 2.1 即爲延伸元件與延伸點的運作方式，一個延伸元件對應至一個延伸點，但延伸點可以對應至多個延伸元件。

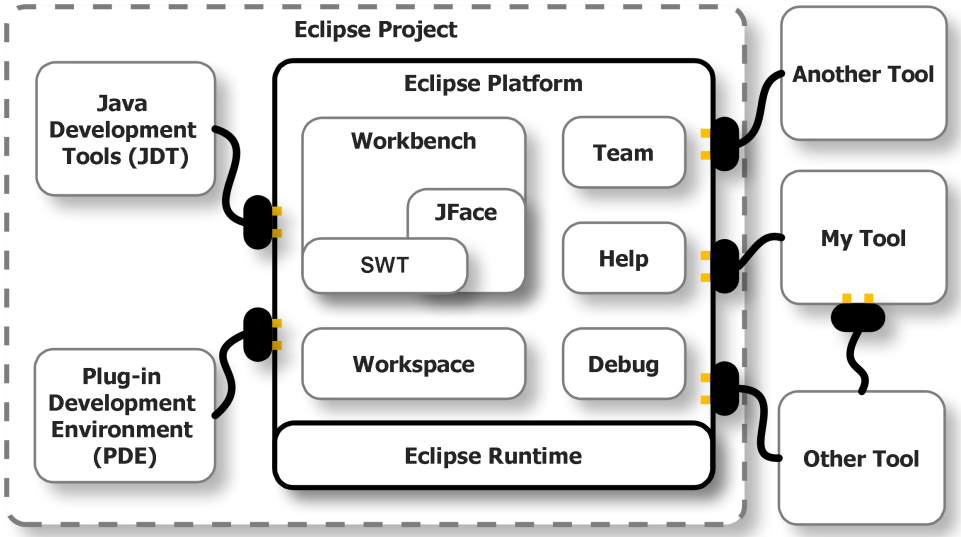

圖 2.1: Eclipse 套件架構

Eclipse 專案除了提供 Java 的開發工具 (JDT) [20] 外，也提供了擴充套件發展環境 (PDE) [21]，讓開發人員可以藉此開發擴充套件。Eclipse 並無任何偵錯實作，但提供了一個用於建構與整合偵錯器的偵錯框架，統稱爲偵錯平台 (debug platform) [4]。偵錯平台包含了偵錯模型 (debug model) 與偵錯視圖 (debug perspective)，偵錯模型爲一組供實作的通用偵錯介面，包括偵錯器常見的元件(例如執行緒、變數和中斷點) 與動作 (例如追蹤、暫停、回復與中止)。而偵錯視圖可操作偵錯模型，並提供相關元件的 views ，讓使用者能夠透過偵錯視圖逐步追蹤程式的執行流程與目前的狀態。換言之，開發者只要實作偵錯框架的核心介面而不用撰寫任何使用者介面的程式碼，便可以產生一個基本的偵錯器。

2.4 語言開發框架

本系統希望建構一個生成器，可以讀入 DSL 的自然語意並產生實作語意的執行元件。除此之外也需要建立 DSL 的編輯環境。所以我們利用語言開發框架— Xtext 來建構這些組件。

2.4.1 Xtext

Xtext [22] 爲 Eclipse 下的套件，它是奠基於 Eclipse、Eclipse Modeling Framework (EMF) [23] 以及 ANTLR 解析器生成器 [24] 的語言開發框架。使用者可以自訂一個 DSL 並描述它的文法，Xtext 即可在 Eclipse 下產生 DSL 的開發環境，包括了編輯器以及解析器，還有程式碼生成器等組件。

程式碼生成器可以解析 DSL 模型，並依照以 Xtend [25] 語言描述的樣板 (template) 來生成實作的程式碼。Xtend 爲一個靜態型別 (statically-typed) 樣板語言，它可以幫助開發與維護程式碼生成器以及其他極度依賴字串連接 (string concatenation) 的程式，也可直接被編譯成可閱讀的 Java 程式碼，文法也與 Java 相似但較爲精簡。

2.5 相似系統

本研究與 Wu 等人的 DSL Debugging Framework [26]、EProvide [27] 以及 Spoofax [28] 的偵錯器生成框架 [29] 均利用 Eclipse 平台建構偵錯器，故特此提出介紹並比較。

2.5.1 DSL Debugging Framework

DSL Debugging Framework (DDF) 的目標在於提供一個偵錯器的生成框架，協助領域專家與使用者在他們熟悉的抽象層級上偵錯 DSL [30] 程式，而不用仰賴通用語言 (GPL) 的知識。DDF 利用文法驅動 (Grammar-Driven) 技術，藉由 DSL 文法生成 DDF 所需要的連結資訊，讓它能與整合式開發環境交流、互動，協助使用者偵錯 DSL 程式。

DDF 的做法是由 DSL 的文法生成映對資訊，將 DSL 原始碼轉換成實作的 GPL

程式碼，然後利用已存在的 GPL 偵錯器來偵錯 GPL 程式碼，最後再將偵錯結果對映回 DSL 層級。首先，DSL 原始碼會經由轉換器產生 GPL 程式碼以及與 DSL 原始碼映對的資訊，包含以下三種:

- 原始碼映對 (source code mapping)：描述 DSL 原始碼中哪一行會對映至相關聯 GPL 程式碼的哪個區段。
- 偵錯方法映對 (debugging methods mapping)：對映 DSL 層級的偵錯命令至 GPL 層級。
- 偵錯結果映對 (debugging results mapping)：將偵錯 GPL 程式碼得到的結果對映回 DSL 層級。

接著 GPL 程式碼與前兩種映對資訊將會被重新解譯成偵錯命令並送至 GPL 偵錯伺服器。GPL 偵錯伺服器負責處理從重新解譯器送來的偵錯命令，並將產生的結果藉由偵錯結果映對轉換至 DSL 層級並回傳，讓使用者透過偵錯視圖來觀測 DSL 程式目前的執行狀態。

Wu 等人 [31] 也利用此框架生成命令式 (imperative)、宣告式 (declarative) 以及混合式 (hybrid) 的 DSL 偵錯器。證明只要利用 DFF 以及少量的額外實作，即可將現有的偵錯工具重複應用至不同種類的 DSLs 上。

2.5.2 EProvide

EProvide 爲一個 Eclipse 上的專案，它允許使用者定義 DSL 的操作語意以及視覺化的直譯器與偵錯器 [27]。

要利用 EProvide 開發 DSL 的執行與偵錯環境，首先需要定義 DSL 的超模型。除此之外，還需要在超模型內描述 DSL 可能存在的狀態 [32]。定義 DSL 操作語意的方式

即爲實作模型的初始化狀態以及狀態遞變的步驟。當 EProvide 執行 DSL 模型時，DSL 的執行狀態會被初始化，接著再逐步遞變，並將狀態保存在模型之內。

接著還需加入 DSL 的偵錯描述，EProvide 內的通用偵錯器才能在 Eclipse 上偵錯 DSL 程式 [33]。偵錯描述包含以下項目：

- 偵錯狀態：由於 DSL 模型內的執行狀態爲目前處於哪個節點，以及各自定義的變數等等。所以需要進行模型轉換，將 DSL 模型轉換成符合 Eclipse 偵錯模型的偵錯狀態，以便透過偵錯視圖呈現給使用者觀看。
- 程式位址：描述 DSL 的程式位址，以便高亮程式內的元素，顯示目前執行至何處。每個執行緒需要記錄目前執行緒執行至何處，而每個 stack frame 也需記錄在此 stack frame 下，程式執行至何處。
- 中斷點：允許特定的模型物件能夠新增或移除中斷點。
- Step 操作：描述各種 step 的操作，讓偵錯器得知程式進行各種 step 的操作時，需要執行至何處才完成。

2.5.3 Spoofax 偵錯器生成框架

Spoofax 是一個開發文字式 (textual) DSL 的工作平台，且擁有 Eclipse 編輯器功能。使用者可以使用 SDF [34] 來描述 DSL 的文法，Spoofax 則會自動生成編輯器並提供簡單的編輯輔助。除此之外，Spoofax 平台下也有一套偵錯器生成框架，讓使用者能以抽象化的方式定義 DSL 偵錯器。

偵錯器生成框架使用宣告式的 SEL 語言來定義 DSL 的通用偵錯模型，偵錯模型以偵錯事件爲基礎來擷取程式的執行狀態。`step` 事件表示 DSL 陳述式 (statement) 的順序；`enter` 與 `exit` 事件表示 `step` 之間的階層關係；`var` 事件則指定目前階層上，

需要宣告的變數。SEL 描述如何在 DSL 語法內產生偵錯事件，以及如何提取相關的偵錯資訊。當 DSL 程式碼被解析成抽象語法樹後，系統會依據 SEL 的描述，將偵錯事件加入至語法樹內。除此之外，系統還需要實作原生偵錯函式庫，以便將偵錯事件傳遞給偵錯器。偵錯器的執行時期的架構分爲四層，其中包含一層通用的基礎建設 (infrastructure)。藉由分層的架構，可以重複使用通用的組件，以便在新增 DSL 偵錯器時，減少需要實作的組件。

2.5.4 相似系統比較

Wu 等人 [26] 的 DDF 僅提供了一個基本的生成框架，語言的實作與偵錯器的生成仍需在通用語言的層級下描述。至於 EProvide 則將程式的執行抽象化爲 `reset` 與 `step` 兩個操作，實作此兩個操作可以讓 DSL 程式逐步執行。但若要生成 DSL 的偵錯器，則還得進行模型轉換並一一實作所需的偵錯介面。相較於上述兩者，本系統以自然語意以及偵錯定義來生成語言的實作及偵錯器，讓使用者能夠於更抽象化的層級上描述語言，不用涉及程式碼的實作以及額外的偵錯介面實作。

而 Spoofax 的偵錯器生成框架也提供了名爲 SEL 的語言來描述 DSL 的偵錯定義，與本系統描述偵錯機制的做法頗爲相似。然而它是以偵錯事件爲基礎並與 Wu 等人 [35] 同樣以 AOP (Aspect-oriented programming) [36] 的做法，將偵錯資訊 weave 至 DSL 的抽象語法樹。最後還需實作 DSL 的偵錯函式庫來傳遞偵錯事件。本系統則於描述程式語意的自然語意內加入偵錯定義，相較於從語法樹分析再加入的做法，在語意內描述偵錯定義會有比較大的彈性。除此之外，本系統也不需額外的實作來處理偵錯定義。

第 三 章

系統架構分析

本章先於 3.1 小節簡單介紹本系統的架構概要，再於 3.2 小節介紹系統輸入模型， 3.3 至 3.5 小節則分析系統各部架構。

3.1 架構概要

本論文主要任務是實作一套可用以快速產生 DSL 執行與偵錯環境的系統，我們將之命名爲 RageDen，它主要可分成三個部分：

- RGD2Code
- RageDen 執行平台
- RageDen 偵錯套件

除了產生執行與偵錯環境之外，仍需要 DSL 編輯環境和偵錯視圖以便與執行與偵錯環境互動。以下將分別敘述系統生成時期與執行時期的架構以及各部元件。

3.1.1 系統生成時期架構

要生成特定 DSL 的執行與偵錯環境，使用者需要定義 DSL 的輸入模型，包括文法規則、自然語意還有偵錯定義。圖 3.1 顯示 RageDen 的生成時期架構，其主要元件如下：

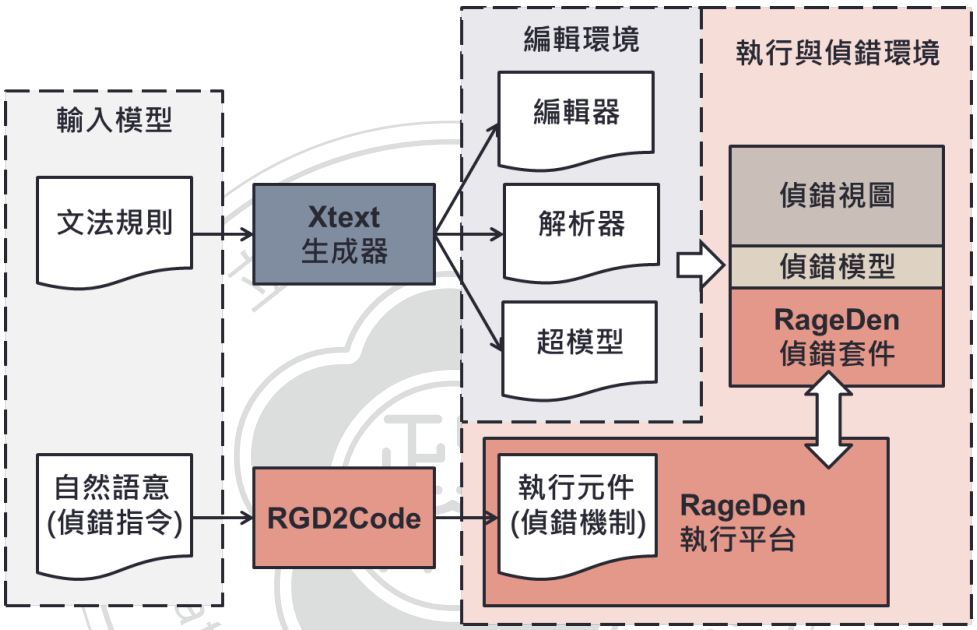

圖 3.1: 系統生成時期架構

- Xtext 生成器：Xtext 爲 Eclipse 上既有的工具，是一個語言開發框架。其中 Xtext 生成器可讀入 DSL 的文法規則，並生成編輯器、解析器與超模型等等組件，構成 DSL 的編輯環境。編輯器輔助使用者編輯程式碼，解析器則會將 DSL 程式碼轉換成以 EMF 實作的抽象語法樹，而超模型可以描述語法樹的結構。

- RGD2Code：語意定義描述 DSL 程式該如何執行，RGD2Code 能經由語意定義分析產生實作語意的執行元件。執行元件可以讀入 DSL 的抽象語法樹並解譯程式。若加入偵錯定義，則產生的執行元件會具備偵錯機制，能夠與偵錯器互動並偵錯程式。

- RageDen 執行平台：每個 DSL 均需直譯器與偵錯器來接收命令，控制程式流程並取得結果。而我們建構的執行平台內，包含了執行元件所需的函式庫以及直譯器與偵錯器通用框架。當使用者執行或偵錯一個特定 DSL 的程式時，會啓動執行平台下的直譯器或偵錯器通用框架，此框架會載入對應此 DSL 的執行元件來執行與偵錯程式。

- 偵錯模型：Eclipse 平台下的偵錯模型爲一組大部分偵錯器通用的元件與動作的抽象化介面，常見的元件例如 thread、stack frame、variable 等，常見的功能則有暫停、回復、中止等等。

- 偵錯視圖：Eclipse 平台提供了偵錯視圖，讓使用者能夠利用人性化的使用者介面來偵錯程式。偵錯視圖包含許多種類的 view，呈現代表被偵錯程式的偵錯模型資訊。每個 view 也提供許多偵錯命令供使用者下達，並經由偵錯視圖操作與觀測偵錯模型。

- RageDen 偵錯套件：偵錯套件可視爲偵錯視圖與偵錯器之間溝通的橋樑，它實作了偵錯模型介面以便與偵錯視圖互動。使用者經由偵錯視圖透過偵錯套件發送偵錯命令給執行平台，程式執行的狀態則由執行平台透過偵錯套件呈現在偵錯視圖上。

3.1.2 系統執行時期架構

圖 3.2 爲 RageDen 執行時期的架構。使用者可在編輯環境內編輯 DSL，偵錯或執行 DSL 程式時，RageDen 偵錯套件會從編輯器取得 DSL 的抽象語法樹，並啓動 RageDen 執行平台下直譯器或偵錯器的通用框架。直譯器或偵錯器的通用框架會載入相對應的 DSL 執行元件，再直接由執行元件解譯或偵錯抽象語法樹。偵錯模式下，使用者可以利用偵錯視圖操作偵錯模型並下達命令，實作偵錯模型的偵錯套件會將命令發送至偵錯器，偵錯器則依此命令控制執行元件的執行流程。當執行暫停時，使用者可經由偵

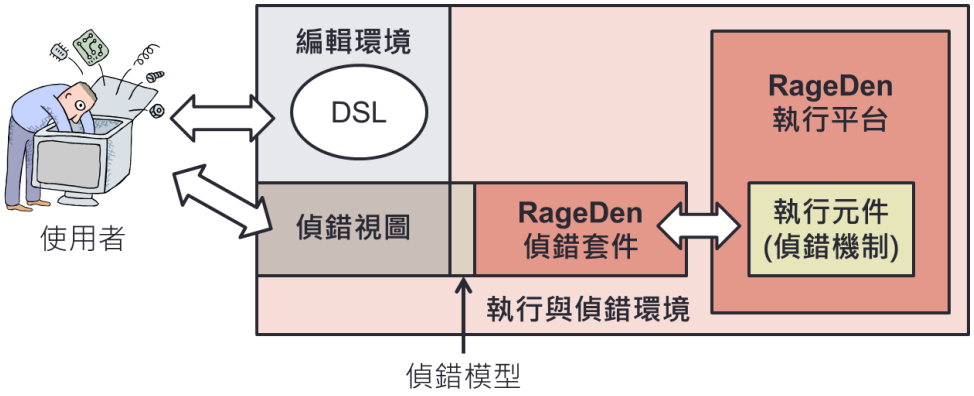

圖 3.2: 系統執行時期架構

錯視圖透過偵錯套件控制程式的執行流程以及觀測程式目前的執行狀態。

3.2 系統輸入模型

系統的輸入模型可分爲文法、自然語意與偵錯指令三種。文法描述語言的具體語法與抽象語法，自然語意描述語言的執行語意，偵錯指令則描述語言的偵錯定義。

3.2.1 文法

文法的目的在於描述 DSL 語言的語法結構，以及如何將程式碼對應成結構的規則。本系統使用 Xtext 定義的文法規則當作輸入模型，當給定一個名稱爲 L 的語言之後，即可利用 Xtext 在 L.xtext 檔案內編輯 L 的文法規則。Xtext 生成器會依照文法規則生成編輯器、解析器以及超模型等元件，將 L 程式碼轉換成抽象語法樹。

舉例來說，表 3.1 爲一個 Robot DSL 文法範例，它是一個 Wu 等人 [31] 定義的機器人語言，可藉由上下左右的命令來移動機器人的位置，也可執行函式呼叫，讓機器人依照函式內的命令移動，除此之外還可印出機器人所處的位置。要描述 Robot DSL 的文法規則，首先得於開頭處宣告文法的名稱— `Robot`，接著匯入 `org.eclipse.xtext.`

common.Terminals，以便使用 Terminals 預設的文法規則。由於 Xtext 依照文法推論並自動產生 Ecore 模型，所以在第四行得宣告此模型的 EPackage 名稱爲 robot，nsURI 則爲 "http://www.rageden.org/examples/robot/Robot"，Xtext 接著會根據不同的解析規則 (parser rule) 產生各種模型類別。

生成 Ecore 模型後，Xtext 會依照解析規則將 Robot 程式碼轉換成符合 Ecore 模型的抽象語法樹。首先 Robot 解析規則會讀入整個程式碼，此規則包含了一組函式宣告與主函式，CallFunc 解析規則會解析函式宣告並產生 IFunc 的物件回傳給 arg2，其中 arg2+=CallFunc* 表示 arg2 爲一個串列 (list)，會將任意個 CallFunc 回傳的物件均串接起來。MainFunc 則會解析主函式宣告並將回傳的物件指定給 arg1，最後構成一個名爲 IBOT 且包含 arg1、arg2 兩個特徵項目 (feature) 的模型類別，並視爲 IRobot 的子類別回傳。

3.2.2 自然語意

本研究採用 RGD 的語法規格來作爲 DSL 自然語意的輸入，它是以 Xtext 實作的一個 DSL，其格式則參考 RML 並加以修改而形成的。RGD 的架構主要是由模組 (module) 組成，每個模組定義了各自的結構，這些結構包含以下六種宣告：

- 匯入宣告 (Import declaration)：以 import <PackageName> 表示。匯入其他套件的模組或 DSL 抽象語法樹的模型類別。
- 型態宣告 (Type declaration)：以 type = <ID> 表示。給予型態另外一個別名。
- 變數宣告 (Variable declaration)：以 val = <Expr> 表式。宣告變數並指定初始值。
- 資料型態宣告 (Datatype declaration)：宣告結構化的資料型態以及它的數值，以便進行模式比對 (pattern match)。

表 3.1: Robot.xtext

```
grammar org.rageden.examples.robot.Robot
  with org.eclipse.xtext.common.Terminals

generate robot "http://www.rageden.org/examples/robot/Robot"

Robot returns IRobot:
  {IBOT} arg2+=CallFunc* arg1=MainFunc;

MainFunc returns IFunction:
  {IMAIN} 'robot' arg1+=Command+ 'end';

CallFunc returns IFunction:
  {IFUNC} 'function' arg1=ID ':' arg2+=Command* 'end';

Command returns ICommand:
  {IUP}    'up'
| {IDOWN}  'down'
| {ILEFT}  'left'
| {IRIGHT} 'right'
| {IWHERE} 'where'
| {ICALL}  '>' arg1=ID
| {ISET}   'set' '(' arg1=INT ',' arg2=INT ')';
```

- 提取器宣告 (Extractor declaration)：類似資料型態，宣告抽象語法樹的結構與型態，並利用模式比對來匹配語法樹。
- 規則集合宣告 (RuleSet declaration)：宣告一組規格集合，利用數學推論的方式來描述語意。

其中引入宣告、型態宣告、變數宣告均爲大部分程式語言常用的宣告，正式定義請參見附錄 A，於此不多介紹。我們在此僅詳加說明另外三個宣告：

資料型態宣告

在 RGD 內，資料型態就像 SML [37] 的 `datatype`，例如表 3.2 宣告一個名爲 `Value` 的資料型態，並宣告 `eval` 規則集合來解譯 `Value`。`Value` 的數值不是 `Empty`、`Real`，就是 `Max` 建構式。其中 `Empty` 建構式 (constructor) 沒有任何參數，而 `Real` 建構式包含了一個型態爲 `Float` 的參數，`Max` 建構式則包含了兩個 `Int` 型態的參數。

資料型態可當作規則集合的輸入參數並進行模式比對，以便判斷該用哪條規則來推論參數。表 3.2 的 `eval` 規則集合會讀入 `Value` 資料型態的數值當作參數，並進行模式比對。在相對應的規則內描述該如何計算數值，最後回傳 `Int` 型態的數值當作結果。例如 `eval(Empty())` 的輸入參數會符合 `Empty()` 結構式模式，則推論結果爲 `0` 並回傳。如果計算 `eval(Real(3.2))`，則輸入的參數會符合 `Real(fVal)`，`Real` 建構式內的 `3.2` 會指定給 `fVal` 變數，此規則會呼叫 `FloatToInt` 將 `fVal` 轉換成整數 `3`，並將結果指定給 `iVal`，最後將 `3` 作爲計算的結果回傳。若要計算 `eval(Max(5, 2))`，此參數會符合 `Max(x, y)`，則 `Max` 建構式內的參數 `5`、`2` 會分別指定給 `x` 與 `y`，在推論過程中，若 `x` 比 `y` 的數值大，則 `x > y => true` 成立，並推導出 `5` 做爲結果。然而，若 `x` 比 `y` 小，則 `x > y => true` 不成立，接著繼續比對下一條規則的模式。此輸入參數仍會符合 `Max(x, y)`，並同樣將 `Max` 內的參數分別指定給 `x` 與 `y`，推導出 `x > y => false` 成立，回傳 `y` 做爲計算的結果。

表 3.2: 資料型態宣告

```
datatype Value =
  Empty
| Real of Float
| Max of Int * Int

set eval(Value): Int = {

  axiom Empty() => 0

  rule FloatToInt(fVal) => iVal
       ------------------------
       Real(fVal) => iVal

  rule x > y = true
       ----------------
       Max(x, y) => x

  rule x > y = true
       ----------------
       Max(x, y) => y
}
```

提取器宣告

由 Xtext 生成的 DSL 編輯器、解析器與超模型，會將程式碼轉換成以 EMF 爲基礎的抽象語法樹。由於抽象語法樹爲模型物件，所以需要宣告提取器來分析與分解抽象語法樹，並以代數的方式來表示這些模型物件，以便進行模式比對。

假設要生成一個 Fruit DSL 的執行與偵錯環境，首先需要新增表 3.3 的文法規則至 Fruit.xtext 檔案並生成編輯環境。表 3.4 則宣告了相對應的提取器，這樣才能在 RGD 內使用代數建構式 `Apple`、`Orange` 來分析與分解抽象語法樹，並提取語法樹內的節點進行模式比對。

宣告提取器提取抽象語法樹的節點有以下兩個限制：

- 文法內所描述的模型類別名稱需與提取器的類別名稱相同，但多個前贅字 '`I`'。
- 文法的解析規則裡，模組類別的特徵名稱需以 `arg1`、`arg2`、`arg3` 等依序命名，而提取器內的參數型態則需符合類別內屬性的型態。例如 `Int` 對應至 EMF 的 `EInt`，`String` 對應至 `EString`，`List` 對應至 `EList`。

除此之外，提取器的功用在於提取抽象語法樹進行模式比對，它只能讀取語法樹但不能進行任何修改，不像資料型態一樣能夠任意新增或修改。

規則集合宣告

自然語意使用邏輯上的推論規則 (rule) 及公理 (axiom)，藉此描述程式的語意。規則通常表示成：

$$\frac{premises}{conclusion}$$

表 3.3: Fruit.xtext

```
Fruit returns IFruit:
  Apple | Orange;

Apple returns IApple:
  'apple' arg1=INT;

Orange returns IOrange:
  'orange' arg1=STRING '@' arg2=INT;
```

表 3.4: Fruit.rgd

```
extractor Fruit =
  Apple of List[Int]
| Orange of String * Int
```

表 3.5: 規則的形式

```
rule premise1  &
     premise2  &
     premise3  &
       ...
     premiseN
     -----------
     conclusion
}
```

一條規則會有數個前提 (premises) 與一個結論 (conclusion)，當全部的前提均成立的情況下，結論才會成立。公理則可視爲沒有前提的規則。

表 3.5 爲規則在 RGD 呈現的形式，一條規格以關鍵字 `rule` 開頭，後面接著數個由 `&` 區隔的前提，中間隔著一組破折號 `-` 構成的虛線，最底下則爲結論 [18]。結論在 RGD 下的形式爲 `<Pattern> => <Expression>`，左端的 `<Pattern>` 爲模式，右端的 `<Expression>` 爲表達式。當規則集合在進行推導時，首先會把輸入的參數與結論左端的模式進行比對，判斷此表達式是否符合。模式比對的結果若成功，則會利用匹配的變數依序推論前提，直到全部的前提均滿足時，代表結論成立，並以右端的表達式計算的結果做爲回傳。

規則 (rule) 描述一次執行中，其初始 (initial) 與最終 (final) 狀態之間的關聯 (relation) [5]。擁有相同輸入與輸出型態的規則可被組織成表 3.6 的規則集合，名稱爲 `name`，輸入的參數型態爲零或數個 `inputType1, ...`，回傳結果的型態爲 `resultType`。

模式的種類有以下幾種：

表 3.6: 規則集合的形式

```
set name(inputType1, ...): resultType = {
  axiom pattern1 => result1
  rule premise1  &
       premise2  &
       premise3  &
       ...
       premiseN
       -----------------
       pattern2 => result2
}
```

- 萬用字元模式 (Wildcard pattern)：萬用字元模式 (_) 會符合任何輸入，亦即任何輸入與 _ 進行模式比對均會符合。
- 常數模式 (Constant pattern)：常數模式包括了整數、浮點數、字串與布林值等等，其中空串列可用 `Nil` 表示。
- 變數模式 (Variable pattern)：變數模式就像萬用字元一樣，可以符合任何輸入，但它會綁定 (bind) 一個變數名稱給此輸入。另外，`head :: rest` 的形式會比對輸入的參數是否爲串列，若是的話，則取出串列內第一個元素，並綁定給 `head`，而剩下的串列則綁定給 `rest`。
- 建構式模式 (Constructor pattern)：建構式模式會比對資料型態或是提取器宣告的建構式是否與之符合。
- Tuple 模式 (Tuple pattern)：Tuple 的形式就像 `(a, b, c)`，它會對每個 tuple 內的元素也進行比對。

- 變數綁定 (Variable binding)：除了模式比對外，我們可以將任何一個符合模式比對的參數綁定至一個變數，在後續的推論中即可以使用此變數來代表此參數。它的格式爲 `variable as pattern`，亦即如果比對的參數符合 `pattern` 時，參數的數值會被指定給 `variable`。

當我們藉由模式比對，找出符合的規則並進行匹配之後，還要依序推導其前提是否全部成立，前提的種類有以下幾種：

- 相等前提 (Equal premise)：以 `<ID> = <Expr>` 的形式表達。如果名稱爲 `ID` 的變數與 `Expr` 表達式的運算結果符合的話，則前提才會成立。
- 否定前提 (Not premise)：以 `not <Premise>` 的形式表達。如果 `Premise` 成立的話，則此否定前提不成立，反之亦然。
- 假設前提 (let premise)：以 `let <Pat> = <Expr>` 的形式表達。此前提會假設 `Expr` 表達式會符合 `Pat` 模式。
- 呼叫前提 (Call premise)：以 `<ID>(<Expr>, <Expr>, ...)=> <Pat>` 的形式表達。呼叫名爲 `ID` 的規則集合，並以 `Expr` 當作參數，最後將回傳結果與 `Pat` 模式比對。
- 運算前提 (Operation premise)：以 `<Operation> => <Pat>` 的形式表達，`Operation` 定義了基本的二元運算與關聯運算，運算結果會與 `Pat` 模式比對。

當一條規則的前提均成立之後，我們需要回傳推論的結果。結果會以表達式 (expression) 的形式呈現，可分爲以下幾種：

- 常數表達式 (Const Expression)：常數表達式包括了整數、浮點數、字串與布林值等等。

表 3.7: main.rgd

```
1 module Main
2 set main() {
3   rule parseSource() => model &
4        Robot.evalRobot(model)
5        ---------------------------
6        _ => ()
7 }
```

- 變數表達式 (Variable Expression)：變數表達式以英文字母或是 _ 爲起頭，之後可以參雜數字來命名。
- 結構式表達式 (Constructor Expression)：建構式表達式爲資料型態或是提取器宣告的建構式。
- 串列表達式 (List Expression)：串列的表達方式爲 `[a, b, c]`，代表此爲包含 `a, b, c` 三個元素的串列。
- Tuple 表達式 (Tuple Expression)：Tuple 的表達方式爲 `(a, b, c)`，代表此爲包含 `a, b, c` 三個元素的 tuple。

最後我們需要定義一個名爲 `Main` 的模組，並宣告一個 `main` 的規則集合當作 RGD 程式的進入點（entry point)。例如表 3.7 爲 Robot 的程式進入點，`main` 沒有輸入參數以及回傳結果，所以我們以萬用字元模式與參數進行比對，而回傳值則以空括號 `()` 表示並無回傳值。模式比對符合之後，會呼叫 `parseSource` 來讀取抽象語法樹，並將取得的抽象語法樹綁定給 `model`。接著再呼叫 `Robot.evalRobot`，將 `model` 傳遞給 `Robot` 模組內的 `evalRobot` 規則集合進行解譯。

3.2.3 偵錯指令

實作語意的執行元件雖然能夠解譯 DSL 抽象語法樹，計算程式執行的結果，但無法進行偵錯。所以我們需在 RGD 內加入偵錯指令，讓產生的執行元件能夠具備偵錯機制，與偵錯器互動。我們實現以下兩種偵錯機制：

- 偵錯模式下，執行元件能夠在特定的位置暫停，並等待偵錯器的回應。
- 暫停時，能控制程式的執行流程並觀測程式目前的狀態。控制執行流程的命令包括了新增、移除中斷點與恢復執行的偵錯命令，以及各種 Step 的操作。程式的狀態則包括了函式 (也可能是子程序或副程式等等) 呼叫順序以及函式內的區域變數及其數值。

所以我們定義四種偵錯指令來描述這些機制，以下將一一解釋這些指令的用法與意義：

- 中斷檢查點：檢查執行元件執行至此處時是否需要暫停。
- 新增 stack frame：表示執行元件開始執行函式呼叫，並新增一個 stack frame 記錄函式的資訊。
- 移除 stack frame：表示執行元件已離開目前所在的函式，並移除記錄函式資訊的 stack frame。
- 更新變數：更新目前所在函式內的區域變數及其數值。

中斷檢查點

中斷檢查點的語法規則爲 `SuspendCheck <ID>`，其中 `ID` 爲綁定抽象語法樹節點的變數名稱。例如表 3.8 中的 `evalCommand` 會解譯表 3.1 中 Robot DSL 的 `ICommand` 類

表 3.8: 中斷檢查點設置

```
set addDebugInstruction(Command, Location): Location = {
  rule SuspendCheck cmd &
       evalCommand(cmd, c) => c2
       --------------------------
       (cmd, c) => c2
}

set evalCommand(Command, Location): Location =
{
  ...
}
```

別，其中 Command 爲 ICommand 的提取器，Location 爲記錄機器人座標的資料型態。若要以 ICommand 作爲執行一次步驟 (single-step) 的最小單位，則可額外宣告一個 addDebugInstruction 規則集合，在執行 evalCommand 解譯 ICommand 命令前，加入 SuspendCheck cmd。表示在偵錯模式下，若執行元件解譯至此指令時，會檢查目前是否需要暫停，若爲是則暫停並等待偵錯器的回應。cmd 則是 ICommand 在模式比對時綁定的變數，它代表著 ICommand 的模型物件。我們可利用此物件取得目前 ICommand 所在的程式碼行號，以便判斷目前執行元件執行至 DSL 程式碼的哪一行。

新增 stack frame

新增 stack frame 的語法規則爲 StackFrame <Expr> @ <Expr>。前者的 Expr 代表此函式的名稱，它必須爲字串，或是數值爲字串的變數。後者的 Expr 則代表此函式的抽象語

表 3.9: 新增與移除 stack frame 設置

```
set evalCall(Location, Function): Location = {
  rule StackFrame fid @ node &
       evalCommands(cmds, c) => c2 &
       endStackFrame
       ----------------------------------
       (c, node as FUNC(fid, cmds)) => c2
}
```

法樹節點。例如表 3.9 中的 evalCall 爲 Robot 解譯函式呼叫的規則集合，而其中的 evalCommands 會解譯函式內包含的命令，並回傳執行完函式呼叫後，機器人所處的座標。我們需要在執行 evalCommands 前加入 StackFrame id @ node，指示偵錯器新增一個 stack frame 來記錄函式內的資訊。

移除 stack frame

移除 stack frame 的語法規則爲 endStackFrame。執行元件在執行完函式呼叫後，需要加入 endStackFrame，指示偵錯器移除記錄此函式的 stack frame。

更新變數

更新變數的語法規則爲 Variable <Expr> : <Expr> = <Expr>。前者的 Expr 表示此變數的名稱，中間的 Expr 表示此變數的型態名稱，最後的 Expr 則表示此變數的數值。如表 3.10 所示，Variable "x" : "int" = x 以及 Variable "y" : "int" = y 會新增名爲 x 與 y，型態名稱均爲 int ，數值分別爲 x、y 的變數，並加入至目前所處的 stack

表 3.10: 更新變數設置

```
set updateCoord(Int, Int) {
  rule Variable "x" : "int" = x &
       Variable "y" : "int" = y
       ----------------------
       (x, y) => ()
}
```

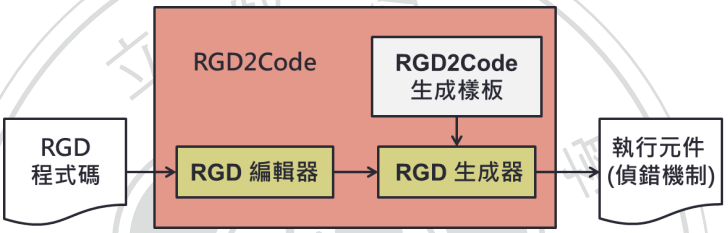

圖 3.3: RGD2code 架構

frame 內。若 stack frame 內已有名稱爲 x 或 y 的變數，則會更新它們的數值。

3.3 RGD2Code

RGD2Code 可以讀入描述 DSL 自然語意以及偵錯指令的 RGD 程式碼，轉換並生成具備偵錯機制的 DSL 執行元件。圖 3.3 爲執行元件的生成過程，使用者可以利用 RGD 編輯器編輯 RGD 程式碼，RGD 生成器則會從編輯器取得由 RGD 程式碼轉換成的抽象語法樹，並依照 RGD2Code 生成樣板的定義，解析抽象語法樹並生成實作語意的偵錯元件。

3.4 RageDen 執行平台

RageDen 執行平台是一個讓執行元件能夠運作的環境，它包含了執行元件所需的函式庫，以及直譯器與偵錯器通用框架。當使用者想執行或偵錯 DSL 程式時，只要啓動直譯器或偵錯器通用框架，載入特定 DSL 的執行元件，即可對特定的 DSL 程式碼進行執行與偵錯。由於 DSL 的執行元件本身即可直接解譯 DSL 程式碼，所以直譯器僅需載入並呼叫執行元件。至於偵錯器則需要控制執行元件，讓使用者跟隨程式的執行流程，在任何想要的地方暫停執行元件並觀測程式的狀態。

加入偵錯指令的 RGD 會生成具備偵錯機制的執行元件，其中暫停的機制由中斷檢查點來實現。另外還有新增與移除 stack frame，以及更新變數的偵錯指令，這些指令指示了 DSL 的函式呼叫流程。所以偵錯器需要一個控制堆疊 (control stack) 來紀錄 DSL 函式呼叫的順序，以及函式內的區域變數。圖 3.4 爲我們定義的控制堆疊超模型，`ControlStack` 爲一個堆疊 (Stack)，它包含了一組記錄函式資訊的 `ActivationRecord`。當偵錯器收到新增 stack frame 的指令時，代表此時執行元件已進入一個 DSL 函式，便會產生一個新的 `ActivationRecord`，然後呼叫 `push` 函式將它推入堆疊的最頂層。若在函式下又再度收到新增 stack frame 的指令，則會新增另外一個 `ActivationRecord`，同樣推入堆疊的最頂層。所以顯示堆疊最頂層的 `topMostFrame` 函式代表執行元件目前所處的 stack frame。直到接收到離開 stack frame 的指令時，表示此時已離開函式，才會呼叫 `pop` 函式將紀錄目前函式資訊的 `ActivationRecord` 從堆疊的最頂層取出並丟棄。`ActivationRecord` 中的 `name` 表示函式名稱，`line` 表示在此函式下，程式執行至哪一行，`source` 則爲對應此函式的 DSL 抽象語法樹節點，`callerFrame` 則指向呼叫此函式的 `ActivationRecord`。

偵錯器若收到更新變數的指令時，會將變數的資訊儲存在 `topMostFrame` 的 `variables` 內，每個 `Variable` 均有一個 `name` 儲存變數的名稱，且擁有一個 `Value`

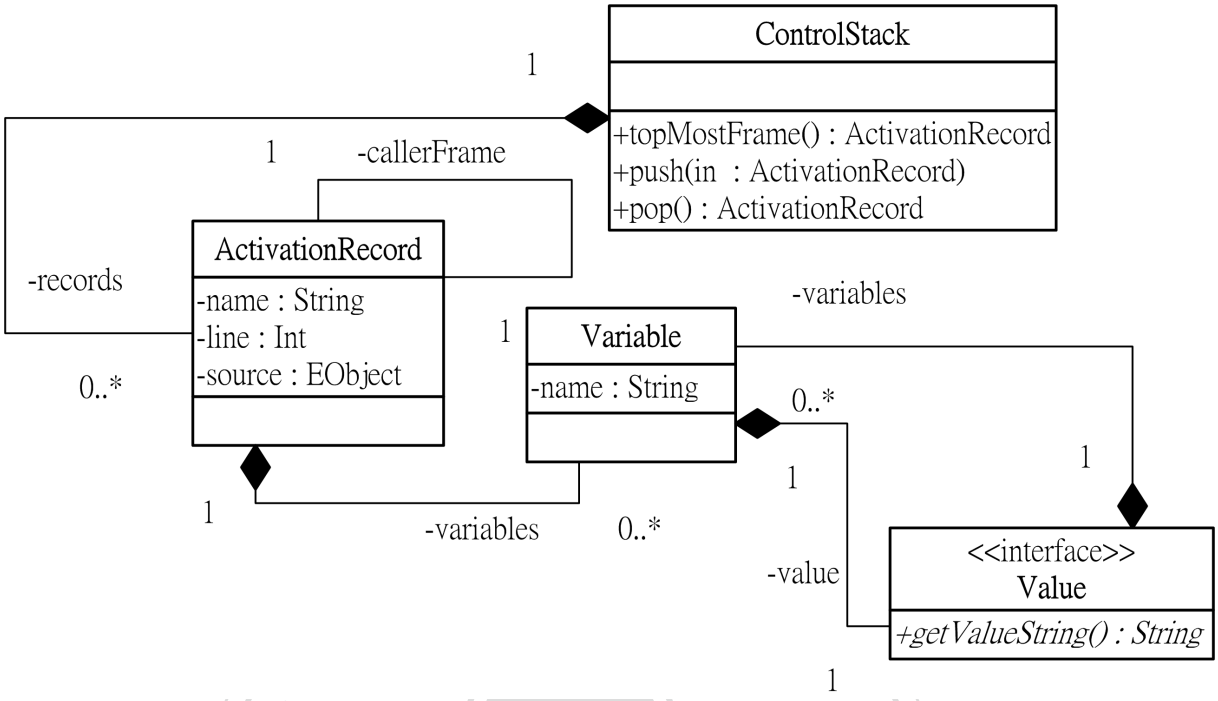

圖 3.4: 控制堆疊超模型

。`Value` 爲一個介面，它定義 `getValueString` 抽象方法，目的在於以字串的形式表達目前數值的內容。若 `Value` 包含一個以上的 `Variable`，則表示此 `Value` 擁有樹狀的結構。此控制堆疊可傳遞給偵錯套件，呈現目前 DSL 的函式呼叫順序。

3.5 RageDen 偵錯套件

能夠執行與偵錯 DSL 程式後，還需要提供使用者圖形介面，以方便使用者在偵錯模式下操作與觀測被偵錯的 DSL 程式。由於 RageDen 偵錯器沒有使用者圖形介面，所以我們利用 Eclipse 的偵錯視圖做爲使用者與偵錯器互動的介面。RageDen 偵錯套件會實作 Eclipse 偵錯平台提供的框架，並定義一套偵錯命令與事件與偵錯器交流，讓使用者透過偵錯視圖經由偵錯套件來與偵錯器互動。以下是偵錯套件需具備的元件與功能：

3.5.1 啓動器

整合式開發環境其中一個基本功能便是啓動（執行或偵錯）開發中的程式碼 [38]。然而 Eclipse 只是個工具平台而不是工具，它只提供了啓動框架而不具備功能。所以 RageDen 偵錯套件需要實作一個啓動器，讓 Eclipse 能藉由啓動器呼叫具備啓動功能的程序來啓動程式。本套件內的啓動器會因應啓動模式的不同而選擇呼叫直譯器或偵錯器來啓動程式碼。

3.5.2 偵錯模型

Eclipse 的偵錯模型 (debug model) 包含了實際執行環境下常見的偵錯介面 [39]。圖 3.5 即爲偵錯模型的 UML 示意圖，它定義了 processes 、 threads 、 stack frames 、 variables 等介面。偵錯目標 (debug target) 代表一個可被偵錯的執行內容，像是程序 (process) 或是虛擬機器 (virtual machine)。每個偵錯目標會包含一或多個執行緒，執行緒 (thread) 包含著 stack frames，stack frame 下則包含了變數，變數內擁有數值的資訊，上述元件均繼承自偵錯元素 (debug element)。當一個 DSL 程式碼在偵錯模式下被啓動時，它即爲一個偵錯目標，而這些抽象化的介面即爲此偵錯目標的組件。

3.5.3 中斷點機制

中斷點機制是現代偵錯器的主要元件之一，它能讓程式執行到特定位置時暫停以便觀察此時程式的執行狀態。本偵錯套件的中斷點機制包括了以下三項：

- 通用中斷點—我們實作了通用中斷點，可供任何 DSL 使用，它會記錄中斷點於 DSL 程式碼內的位置。

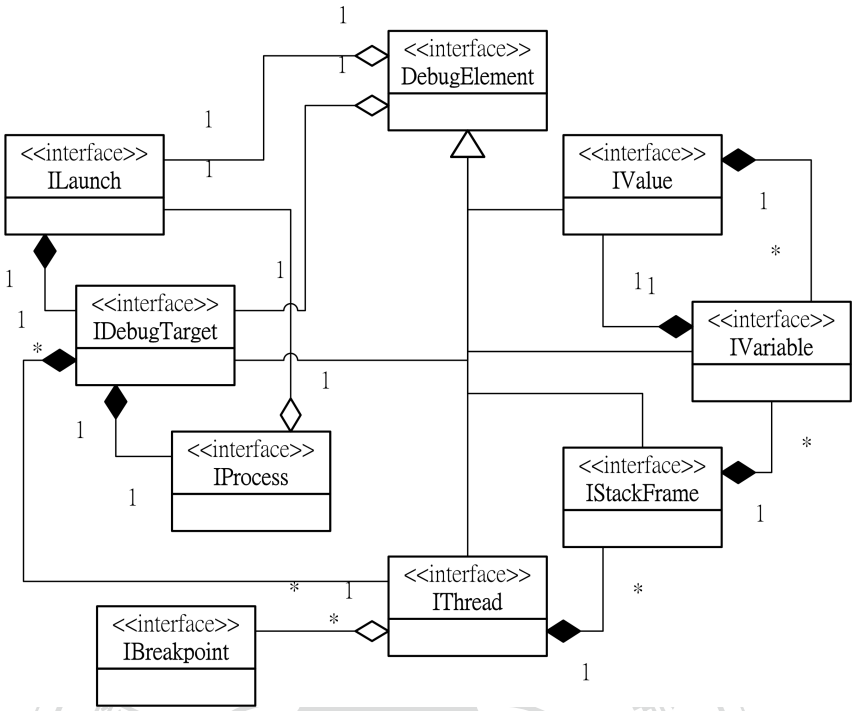

圖 3.5: Eclipse 內的偵錯模型 UML

- 中斷點管理—中斷點的新增與移除，還有判斷程式目前是否遭遇到中斷點之類的管理。
- 觸發中斷點—偵錯套件需要在 DSL 編輯器內加入觸發中斷點的動作，讓使用者能夠在編輯器內新增或是移除中斷點。

3.5.4 原始碼整合顯示

偵錯套件需要將程式的執行狀態與程式碼整合，讓我們能夠高亮某行程式碼，顯示目前選擇的 stack frame 下，程式執行至何處。原始碼查看是用來顯示 stack frames 對應至程式碼的偵錯框架，當一個 stack frame 被選擇時，會有一個原始碼查看器 (source locator) 去搜尋 stack frame 所對應的來源元素 (source element)。接著偵錯模型呈現 (debug model presentation) 會找出與來源元素相對應的 editor input 與 editor id，最後偵錯平台會依照 editor input 與 editor id 開啓相對應的編輯器，並移至此 stack frame

目前執行到的程式碼行，爲它加上一個指令點 (instruction point) 的標記。

第 四 章

系統實作

本章將闡述 RageDen 系統的實作方法。首先我們要建立 RGD2Code，以便讀入自然語意與偵錯指令，並生成具備偵錯機制的執行元件。接著建構 RageDen 執行平台，它可以啓動直譯器或偵錯器的通用框架，載入相對應的執行元件來解譯語法樹並接收偵錯命令以便進行偵錯。最後實作 RageDen 偵錯套件，將直譯器與偵錯器整合至 Eclipse 內。

4.1 建立 RGD2Code

RGD2Code 是以我們自定的 RGD 語言來作爲輸入模型，並產生執行元件的生成器。我們利用 Xtext 套件來建立 RGD2Code，圖 4.1 使用 Xtext 專案精靈 (wizard) 建立 `org.rageden.rgd2code` 的專案。完成專案精靈後，工作區會出現三個專案：

- `org.rageden.rgd2code`
- `org.rageden.rgd2code.tests`
- `org.rageden.rgd2code.ui`

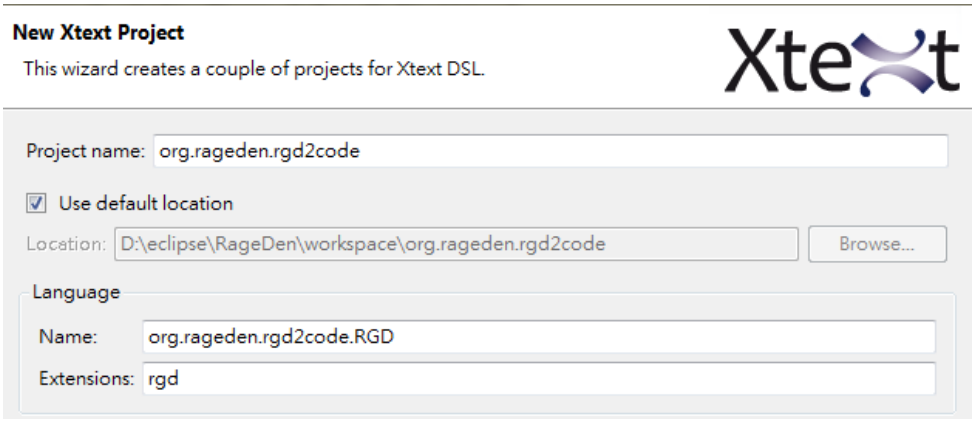

圖 4.1: 藉由 Xtext 專案精靈新增 RGD2Code 專案

`org.rageden.rgd2code` 包含了文法定義與 runtime 元件，`org.rageden.rgd2code.tests` 用來進行單元測試，`org.rageden2code.rgd.ui` 則擁有編輯器與其他圖型介面相關的功能。我們需要在 `org.rageden.rgd2code` 內的 RGD.xtext 描述 RGD 的文法，接著我們即可啓動 Xtext 的生成器，生成編輯器、解析器、以 EMF 爲基礎的 RGD 模型，以及其它額外的 Xtext 組件。

由 RGD.xtext 生成了 Xtext 組件之後，一個程式碼生成器的根存 (stub) 會放在 rgd2code 的 runtime 專案內，接下來我們要將程式碼生成器整合至 Eclipse。表 4.1 爲 `org.rageden.rgd2code.generator` 套件內的 `RGDGenerator.xtend` 檔案，其中的 `RGDGenerator` 爲用來在互動式 Eclipse 環境下生成 RGD 模型的實作程式碼的物件，物件內的 `doGenerator` 函式會執行 RGD 程式碼生成器。首先得匯入 `ResourceExtensions`，它提供 `allContentsIterable` 的功能，將 `Resource` 反覆取出並篩選出類別爲 `Module` 的物件。接著依照取出的 `module` 物件，由 `module.fileName` 函式取得檔案名稱，呼叫 `module.compile` 函式取得生成程式碼。再執行 `fsa.generateFile(module.fileName, module.compile)` 生成檔名爲 `module.fileName`，內容爲 `module.compile` 的實作程式碼。

除了依照 RGD 模組的結構一一生成相對應的程式碼外，我們還要實作 RGD 的程式進入點。當生成器找到模組名稱爲 `Main` 時，表示 RGD 已定義程式進入點。`fsa.`

generateFile(module.actorPath, module.actor) 生成 RGDExecutionArtifactScala.scala，此類別會新增一個執行緒，執行 Main 模組內名爲 main 的規則集合。fsa.generateFile(module.artifactPath, module.artifact) 生成 RGDExecutionArtifact.java，此類別繼承 ExecutionArtifact 介面當作 DSL 的執行元件，以供直譯器或偵錯器通用框架載入。

表 4.2 內的 module.fileName 會取得 module 生成的檔案名稱。當一個類別爲 Module 的物件呼叫 fileName 時，它會取得此物件的套件名稱與物件名稱，並將名稱中的句點 (.) 取代爲斜線 (/) 以便符合檔案系統名稱。最後面再加上 .scala 副檔名，因爲我們生成的語意實作程式碼爲 Scala [40] 程式。

4.2 建構 RageDen 執行平台

RageDen 執行平台包含執行元件需要的函式庫、直譯器與偵錯器通用框架。只要呼叫直譯器與偵錯器通用框架，並載入特定 DSL 的執行元件，便可對特定的 DSL 程式碼進行執行與偵錯。因爲 DSL 的執行元件本身即可直接解譯 DSL 程式碼，所以 RageDen 直譯器只需載入並啓動執行元件。至於 RageDen 偵錯器除了載入執行元件外，還需要控制執行元件，以便讓使用者跟隨程式的執行流程，在任何想要的地方暫停程式並觀測程式的狀態。以下將敘述 RageDen 執行平台各部分的建立。

4.2.1 載入執行元件

我們定義 ExecutionArtifact 介面與名爲 execute 的抽象方法來載入並啓動執行元件。接著新增一個名爲 org.rageden.runtime.execute 的延伸點，讓其他套件能夠藉由新增此延伸點所定義的延伸元件來實作 DSL 的執行元件。此延伸元件需要定義以下屬性：

表 4.1: RGDGenerator.xtend

```
1  package org.rageden.rgd2code.generator
2  import org.eclipse.emf.ecore.resource.Resource
3  import org.eclipse.xtext.generator.IGenerator
4  import org.eclipse.xtext.generator.IFileSystemAccess
5  import static extension org.eclipse.xtext.xtend2.
6            lib.ResourceExtensions.*
7  import org.rageden.rgd2code.rGD.*
8
9  class RGDGenerator implements IGenerator {
10   override void doGenerate(Resource resource,
11                            IFileSystemAccess fsa) {
12     for(module: resource.allContentsIterable.
13       filter(typeof(Module))) {
14       fsa.generateFile(module.fileName, module.compile)
15       if(module.moduleName == "Main"){
16         fsa.generateFile(module.actorPath, module.actor)
17         fsa.generateFile(module.artifactPath,
18                          module.artifact)
19       }
20     }
21   }
22 }
```

表 4.2: Module fileName 樣板碼

```
1 def String fileName(Module m) {
2   m.packageName.replaceAll("\\.","/") + "/" +
3     m.getName.replaceAll("\\.","/")+".scala"
```

- id：定義 DSL 的 id
- name：定義 DSL 的名稱
- extensions：定義 DSL 的副檔名
- executionArtifact：定義 DSL 執行元件，此類別需要繼承 `ExecutionArtifact`，實作 `execute` 函式，並在函式內呼叫執行元件來解譯程式。

4.2.2 實現偵錯機制

由於僅由語意生成的執行元件不具備偵錯功能，所以我們在 RGD 內加入偵錯指令以便實現偵錯機制，讓執行元件偵錯模式下，能夠在特定的位置暫停。在暫停時，能追蹤程式的執行流程並觀測程式目前的狀態。所以我們實作 `DebugActioner` 處理偵錯指令，並定義 `ControlStack` 記錄程式目前的執行狀態。

具備偵錯機制的執行元件會依據偵錯指令呼叫 `DebugActioner` 內相對應的函式來控制流程，並更改 `ControlStack` 的內容以更新程式的執行狀態。以下爲 RGD 偵錯指令的實現方式：

中斷檢查點

當執行元件執行至中斷檢查點的偵錯指令時，DebugActioner 會檢查執行元件是否需要暫停。使用者可以透過偵錯視圖經由偵錯套件下達偵錯命令控制執行流程，偵錯命令可分以下幾種：

- 直接改變執行流程：暫停、恢復、中止等偵錯命令
- Step 操作：Step Into 會執行至下一個中斷檢查點；Step Over會執行至同一層 stack frame 內的中斷檢查點；Step Return 會執行至上一層 stack frame 內的中斷檢查點。
- 新增、移除中斷點

而在執行 Step 的過程中暫停可能會發生衝突，所以需要額外的優先序判斷。本系統的做法爲若是由暫停命令中斷，則恢復執行後 Step 操作會被取消。若爲遭遇中斷點而暫停，則恢復執行後仍會完成 Step 操作。

新增 stack frame

當執行元件執行新增 stack frame 的偵錯指令時，DebugActioner 會呼叫表 4.3 的 methodCall 函式儲存執行完函式呼叫後，該回到的程式位址 (program location)。首先判斷目前執行元件是否位於某個 stack frame 內，若爲是的話則將 returnLocation 設爲 frame.line。接著 if(isStepOver && overFrame == None) 判斷目前執行元件是否正在執行 StepOver 操作，若爲是且 overFrame == None 表示目前的執行元件尚未執行函式呼叫。將執行完 StepOver 之後需回傳到的 stack frame，也就是目前控制堆疊內的 topMostFrame 儲存至 overFrame 內，以便判斷處於哪個 frame 時，StepOver 會完成操作。

表 4.3: methodCall 函式

```
1  def methodCall {
2    controlStack.topMostFrame match {
3    case Some(frame) =>
4      returnLocation = frame.line
5          case None =>
6      returnLocation = -1
7    }
8    if(isStepOver && overFrame == None)
9      overFrame = controlStack.topMostFrame
10 }
```

執行 methodCall 函式後，我們需呼叫表 4.4 的 enterActivationRecord 函式新增 ActivationRecord 來儲存目前函式的資訊。我們首先會新增名爲 newActivationRecord 的 ActivationRecord 類別，並從偵錯指令 StackFrame : labelText @ eobject 取得名稱標籤以及模型物件。其中 labelText 代表此 ActivationRecord 的名稱，而 eobject 表示此 stack frame 所對應抽象語法樹的模型物件。接著從模型物件內取得起始行號並存進 line 中代表目前的程式位址，並且將目前程式所在的 stack frame 指定給 callerFrame 與 newActivationRecord.callerFrame，最後再將 newActivationRecord 推入堆疊。

移除 stack frame

執行元件執行移除 stack frame 的偵錯指令時，DebugActioner 會呼叫表 4.5 的 leaveActivationRecord 函式處理。我們將目前所處的 ActivationRecord 由控制堆

表 4.4: enterActivationRecord 函式

```
def enterActivationRecord(name: String, eobject: EObject) {
  var newActivationRecord = new ActivationRecord
  newActivationRecord.name = name
  newActivationRecord.source = eobject
  var node: ICompositeNode = NodeModelUtils.getNode(eobject)
  if(node.isInstanceOf[AbstractNode])
    newActivationRecord.line =
      node.asInstanceOf[AbstractNode].getStartLine
  else
    newActivationRecord.line = -1
  newActivationRecord.callerFrame = controlStack.topMostFrame
  controlStack.push(newActivationRecord)
}
```

表 4.5: leaveActivationRecord 函式

```
1 def leaveActivationRecord {
2   controlStack.pop
3   methodReturn
4 }
```

疊中取出並丟棄。

最後呼叫表 4.6 的 methodReturn 函式來處理函式回傳後的 Step 操作。當目前正在執行 StepOver 操作，且目前執行元件所在的 topMostFrame 等於 overFrame 時，表示 StepOver 操作完成並清空 overFrame。若目前正在執行 StepReturn 操作，且執行元件所在的 topMostFrame 等於 returnFrame 時，表示 StepReturn 操作完成並將 returnFrame 指定爲 topMostFrame.callerFrame，然後發送暫停的命令。

更新變數

執行元件執行 Variable name : type = value 的偵錯指令時，DebugActioner 會呼叫表 4.7 的 updateVariable(name, type, value) 函式處理。其中 name 爲變數名稱，type 爲型態的名稱標籤，value 則爲此變數的數值。函式首先比對 value 的型態，再依照匹配的結果指定相對應其型態的類別給 newValue。例如匹配結果爲串列型態，則新增一個 ListValue(valueType, l)，否則即爲基本型態(primitive type)，新增 PrimValue(valueType, p)。

最後再搜尋目前 topMostFrame 是否已存在名稱標籤爲 name 的變數。若已存在，則我們將 newValue 指定給變數內的數值。若尚不存在，則我們新增一個 newVariable，將名稱標籤設爲 name，newValue 指定給此 newVariable 的數值，最後再加進 topMostFrame 下的變數串列。

表 4.6: methodReturn 函式

```
def methodReturn {
if(isStepOver && controlStack.topMostFrame != None)
  (overFrame, controlStack.topMostFrame) match {
    case (Some(oFrame), Some(tFrame)) if oFrame == tFrame =>
      overFrame = None
  }
else if(isStepReturn && controlStack.topMostFrame != None)
  (returnFrame, controlStack.topMostFrame) match {
    case (Some(rFrame), Some(tFrame)) if rFrame == tFrame =>
      returnFrame = tFrame.callerFrame
  }
}
```

表 4.7: updateVariable 函式

```
1  def udpateVariable(name: String,
2                     valueType: Any,
3                     valueValue: Any) {
4    var newValue = valueValue match {
5      case l: List[Any] => new ListValue(valueType, l)
6      case p => new PrimValue(valueType, p)
7    }
8
9    controlStack.topMostFrame match {
10     case Some(vdFrame) =>
11       vdFrame.variables.toArray.toList.
12       asInstanceOf[List[Variable]] find (_.getName == name)
13       match {
14         case Some(variable: Variable) =>
15           variable.setValue(valueValue)
16     case None =>
17           var newVariable: Variable =
18             new Variable(name, valueType, valueValue)
19           var vdVariables: List[Variable] =
20             vdFrame.variables.toArray.toList.
21               asInstanceOf[List[Variable]]
22           vdFrame.variables =
23            (newVariable::vdVariables).toSeq
24       }}}
```

4.2.3 接收偵錯命令與回傳偵錯事件

偵錯模式下，偵錯器需要接收從 RageDen 偵錯套件傳送過來的偵錯命令並執行。以下爲目前本系統定義的偵錯命令：

- `Resume`：回復目前暫停程式的執行。
- `Suspend`：暫停目前程式的執行。
- `StepInto`：執行至目前程式碼所在行的下一個函式呼叫。
- `StepOver`：執行至目前程式碼所在行的下一個函式呼叫，但不進入函式內部。
- `StepReturn`：執行至目前程式碼所在行直至回到呼叫它的函式下。
- `Terminate`：中止程序。
- `SetBreakpoint(n)`：在程式碼的第 n 行新增中斷點。
- `ClearBreakpoint(n)`：清除程式碼第 n 行的中斷點。

執行這些偵錯命令後，偵錯器會回傳偵錯事件給偵錯套件，通知套件目前程序的狀態，以下爲本系統定義的偵錯事件：

- `started`：通知套件目前程式已啓動。
- `resume X`：通知套件目前程式已回復之前的執行，`X` 代表回復的原因。若爲 `step` 表示一個 Step 的要求已開始操作；若爲 `client` 表示使用者呼叫的回復命令已執行。
- `suspended X`：通知套件目前程式已暫停，`X` 代表暫停的原因。若爲 `breakpoint` 表示遭遇到中斷點；若爲 `client` 表示使用者要求的暫停指令已執行；若爲 `step` 表示 Step 步驟已操作完畢。

- terminated：通知套件目前程式已停止。

4.3 實作 RageDen 偵錯套件

建構好 RageDen 執行平台之後，我們需要實作偵錯套件，將直譯器與偵錯器整合至 Eclipse。偵錯套件會實作 Eclipse 的偵錯框架，讓使用者透過偵錯視圖與偵錯器互動。我們的偵錯套件分成介面與非介面兩部分的實作，org.rageden.dubug.core 爲偵錯套件的核心實作，org.rageden.debug.ui 則爲偵錯套件的介面實作。首先我們需要建構啓動器以便執行或偵錯程式，接著需要實作偵錯模型，呈現目前程式的執行狀態。之後需設置中斷點機制，讓程式在進行偵錯時會因中斷點暫停至某處，以供使用者追蹤程式流程。最後再整合原始碼顯示，使 DSL 程式碼能對應至目前程式的執行狀態。以下將依序建立偵錯套件。

4.3.1 建構啓動器

首先在 org.rageden.debug.core 下新增一個表 4.8 的 launchConfigurationType 的延伸元件，以便建構啓動器。此啓動組態型態爲 RageDen 的啓動組態型態，意即只要是由 RGD 所描述的 DSL 語言，其程式碼均需由此啓動組態型態來啓動並進行執行或偵錯。

除此之外，還需要實作表 4.9 的啓動代理類別 RLaunchDelegate，此類別代理了啓動器的工作並啓動程式碼。RLaunchDelegate 繼承 LaunchConfigurationDelegate 類別，而 LaunchConfigurationDelegate 已實作 ILaunchConfigurationDelegate 大部分的功能，我們只需另外實作 launch 函式以啓動程式。首先從組態頁面取得取得程式碼的相對路徑 programPath，並藉此取得程式碼的 Resource 與使用者定義的 DSL 名稱，接著我們會新增 RProcess 模擬爲 RageDen 直譯暨偵錯器的程序。並讀入程式碼所屬的

表 4.8: org.eclipse.debug.core.launchConfigurationTypes 延伸元件

```
<launchConfigurationType
  delegate="org.rageden.debug.core.launch.RLaunchDelegate"
  id="rageden.launchType"
  modes="run,debug"
  name="RageDen Model"
  sourceLocatorId="regeden.sourceLocator"
  sourcePathComputerId="regeden.sourcePathComputer">
</launchConfigurationType>
```

DSL 名稱以及抽象語法樹以便執行程式碼。若以偵錯模式啓動程式，則我們需要額外新增一個偵錯目標來偵錯程序。

新增啓動組態型態後，接下來要宣告表 4.10 的 LaunchConfigurationTabGroups 延伸元件，提供一組啓動組態頁面的圖形介面，使用者可在上面編輯啓動組態的屬性。而在表 4.11 的 RTabGroup 內，我們需要實作 createTabs 函式，宣告需要加入哪些頁面。RMainTab 頁面擁有啓動程式需要的組態對話盒供使用者輸入並儲存屬性。而 SourceLookupTab 頁面則用來顯示與編輯原始碼路徑，此頁面將在原始碼查看時使用。

4.3.2 實作偵錯模型

建構完啓動器後，使用者即可在執行模式或偵錯模式下啓動程式碼。我們需要實作偵錯模型以便在偵錯模式下，呈現目前程序的 stack frames。如同節 3.4 的描述，偵錯器實作一個控制堆疊儲存 DSL 的函式呼叫順序，以下實作的偵錯模型均會奠基於控制堆疊內的資訊：

表 4.9: 啓動代理：RLaunchDelegate.java

```
1  public class ModelLaunchDelegate
2                    extends LaunchConfigurationDelegate {
3    public void launch(ILaunchConfiguration configuration,
4      String mode, ILaunch launch, IProgressMonitor monitor)
5                                        throws CoreException {
6        String program = configuration.getAttribute(
7        DebugCorePlugin.ATTR_RAGEDEN_PROGRAM, (String) null);
8      if (programPath == null)
9        abort("RageDen program unspecified.", null);
10     ResourceSet rs = new ResourceSetImpl();
11     Resource modelResource =
12       rs.getResource(URI.createURI(programPath), true);
13     String name = ExtensionManager.getDslName(programPath);
14     IProcess p =
15       new RProcess(name, launch, modelResource, null,
16                    mode.equals(ILaunchManager.DEBUG_MODE));
17     try { Thread.sleep(1000);
18     } catch (InterruptedException e1) { e1.printStackTrace(); }
19     if (mode.equals(ILaunchManager.DEBUG_MODE)) {
20       IDebugTarget target = new RDebugTarget(launch, p);
21       launch.addDebugTarget(target);
22     }}
23   ...
```

表 4.10: org.eclipse.debug.ui.launchConfigurationTabGroups 延伸元件

```
<launchConfigurationTabGroup
  class="org.rageden.debug.ui.launcher.RTabGroup"
  id="rageden.tabGroup"
  type="rageden.launchType">
</launchConfigurationTabGroup>
```

表 4.11: 啓動組態頁面組：RTabGroup.java

```
public void createTabs(ILaunchConfigurationDialog dialog, String
    mode) {
  setTabs(new ILaunchConfigurationTab[] {
    new RMainTab(),
    new SourceLookupTab(),
    new CommonTab()
  });
}
```

- RThread 實作 IThread 介面，並參照 ControlStack 實作下列函式：

  getFrames() — 由 ControlStack 的 getFrames 函式取得 ActivationRecord 串列。並依序新增 RStackFrame 串列，且個別指定串列內的 ActivationRecord 給每個相對應的 RStackFrame。

- RStackFrame 實作 IStackFrame 介面，並參照 ActivationRecord 實作下列函式：

  getVariables() — 由 ActivationRecord 內的 getVariables 函式取得變數 Variable 串列。並依序新增 RVariable 串列，且個別指定串列內的 Variable 給每個相對應的 RVariable。

  getLineNumber() — 由 ActivationRecord 內的 getLine 函式取得目前的程式位址。

  getName() — 由 ActivationRecord 的 getLName 函式取得名稱標籤。

- RlVariable 實作 IVariable 介面，並參照 RVariable 實作下列函式：

  getVaule() — 由 Variable 內的 getValue 函式取得數值資訊。並新增 RValue 串列，且指定 Value 給 RValue。

  getName() — 由 Variable 的 getName 函式取得變數的名稱標籤。

- RValue 實作 IValue 介面，並參照 Value 實作下列函式：

  getValueString() — 由 Value 的 getValueString 函式取得表達數值內容的字串。

  getVariable() - 由 Value的 getVariables 函式取得變數的串列。若變數串列存在元素，則表示此數值爲樹狀結構，例如陣列或串列等等。

表 4.12: org.eclipse.core.resources.markers 延伸元件

```
<extension point="org.eclipse.core.resources.markers"
  id="rageden.markerType.lineBreakpoint"
  name="RageDen Line Breakpoints">
  <super type=
    "org.eclipse.debug.core.lineBreakpointMarker" />
  <persistent value="true" />
</extension>
```

4.3.3 設置中斷點機制

中斷點機制讓使用者可以在程式碼內加入中斷點，以便在偵錯時，讓偵錯器得知執行至何處時需要中斷。中斷點的種類有許多種，例如監看變數的 watchpoint，或是具備某些條件或表達式的中斷點等等。本系統目前只實作一行最多存在一個的行中斷點。要設置中斷點機制，我們需要先定義表 4.12 的標記 (marker) 延伸元件來記錄中斷點位於編輯器何處的資訊。

接著我們新增表 4.13 的中斷點延伸元件，並實作表 4.14 的 `RLineBreakpoint` 繼承 `LineBreakpoint`。新增一個 `RLineBreakpoint` 時，同時也得新增一個標記來記錄中斷點的資訊。

實作 `RLineBreakpoint` 後，還需加入表 4.15 的觸發中斷點的動作以供使用者點擊，使用者才能自行新增或移除中斷點。當使用者在編輯器的尺規上開啓水平式彈出式選單時，會呼叫 `RulerToggleBreakpointActionDelegate` 來代理觸發中斷點的動作。`RulerToggleBreakpointActionDelegate` 則會新增一個觸發中斷點的 retargetable action。這個 retargetable action 會向正在使用的 view 或編輯器要求觸發中斷點的

表 4.13: org.eclipse.debug.core.breakpoints 延伸元件

```
<breakpoint
  class="org.rageden.debug.core.breakpoints.RLineBreakpoint"
  id="rageden.lineBreakpoint"
  markerType="rageden.markerType.lineBreakpoint"
  name="RageDen Line Breakpoint">
</breakpoint>
```

adapter，如果可用，則此 adapter 會當成代理來處理觸發中斷點的動作。我們新增一個表 4.16 的 adapter 延伸元件，並定義 REditorAdapterFactory，在屬於 ITextEditor 類別的編輯器下取得表 4.17 內 IToggleBreakpointsTarget 的 adapter，如果編輯器內的程式碼副檔名符合已生成的執行元件時，會回傳 RBreakpointAdapter。

表 4.18 的 toggleLineBreakpoints 函式爲 RBreakpointAdapter 內代理觸發中斷點的函式，我們先找出目前被選擇文字 ITextSelection 的起始行號並加一來當作 lineNumber，因爲 ITextSelection 的起始行號是以 0 爲起點，但使用者所看到的程式碼行號卻是從 1 開始。接著再取得 BreakpointManager，如果 BreakpointManager 儲存的中斷點串列中已有位於 lineNumber + 1 行號的中斷點，則我們移除此中斷點，若沒有位於 lineNumber + 1 行號的中斷點，則我們則於新增一個處於 lineNumber + 1 行號的 RLineBreakpoint。

4.3.4 整合原始碼顯示

最後實作原始碼查看以便加入程式碼高亮的功能。

表 4.14: 實作行中斷點：RLineBreakpoint.java

```
1  public class RLineBreakpoint extends LineBreakpoint
2      implements IModelEventListener {
3    public RLineBreakpoint(final IResource resource,
4      final int lineNumber) throws CoreException {
5      IWorkspaceRunnable runnable = new IWorkspaceRunnable() {
6        public void run(IProgressMonitor monitor)
7                                          throws CoreException {
8          IMarker marker = resource.createMarker(
9              "rageden.markerType.lineBreakpoint");
10         setMarker(marker);
11         marker.setAttribute(IBreakpoint.ENABLED,
12                             Boolean.TRUE);
13         marker.setAttribute(IMarker.LINE_NUMBER, lineNumber);
14         marker.setAttribute(IBreakpoint.ID,
15                             getModelIdentifier());
16         marker.setAttribute(
17           IMarker.MESSAGE,
18           "Line Breakpoint: " + resource.getName() +
19           " [line: " + lineNumber + "]");
20         }};
21     run(getMarkerRule(resource), runnable);
22   }
23   ...
```

表 4.15: org.eclipse.ui.popupMenus 延伸元件

```
<extension point="org.eclipse.ui.popupMenus">
  <viewerContribution
    id="rageden.editor.ruler"
    targetID="#TextRulerContext">
    <action
      class="org.eclipse.debug.ui.actions.
        RulerToggleBreakpointActionDelegate"
      id="org.rageden.editor.ruler.toggleBreakpointAction"
      label="Toggle Breakpoint"
      menubarPath="debug">
    </action>
  </viewerContribution>
</extension>
```

表 4.16: org.eclipse.core.runtime.adapters 延伸元件

```
<factory
  adaptableType="org.eclipse.ui.texteditor.ITextEditor"
  class=
    "org.rageden.debug.ui.breakpoints.REditorAdapterFactory">
  <adapter type=
    "org.eclipse.debug.ui.actions.IToggleBreakpointsTarget"
  />
</factory>
```

表 4.17: 實作 getAdapter(): REditorAdapterFactory.java

```
1  public Object getAdapter(Object adaptableObject,
2                           Class adapterType) {
3    if(adaptableObject instanceof ITextEditor) {
4      ITextEditor editorPart = (ITextEditor) adaptableObject;
5      IResource resource =
6        (IResource) editorPart.getEditorInput().
7          getAdapter(IResource.class);
8      if(resource != null) {
9        String extension = resource.getFileExtension();
10       if(extension != null &&
11         ExtensionManager.isDslExtensions(extension) &&
12         adapterType.equals(IToggleBreakpointsTarget.class))
13         return new RBreakpointAdapter();
14     }
15   }
16   return null;
17 }
```

表 4.18: 實作中斷點的 adapter: RBreakpointAdapter

```
1  public void toggleLineBreakpoints(IWorkbenchPart part,
2    ISelection selection) throws CoreException {
3    ITextEditor t = getEditor(part);
4    if (t != null) {
5      IResource resource = (IResource) t.getEditorInput().
6                      getAdapter(IResource.class);
7      ITextSelection textSelection = (ITextSelection) selection;
8      int lineNumber = textSelection.getStartLine();
9      IBreakpoint[] breakpoints =
10       DebugPlugin.getDefault().getBreakpointManager().
11       getBreakpoints(DebugCorePlugin.ID_RAGEDEN_DEBUG_MODEL);
12     for(int i = 0; i < breakpoints.length; i++) {
13       IBreakpoint b = breakpoints[i];
14       if(b instanceof ILineBreakpoint &&
15         resource.equals(b.getMarker().getResource()))
16         if(((ILineBreakpoint)b).getLineNumber()
17             == lineNumber + 1) {
18           b.delete();
19           return;}}
20   RLineBreakpoint lineBreakpoint =
21     new RLineBreakpoint(resource, lineNumber + 1);
22   DebugPlugin.getDefault().
23     getBreakpointManager().addBreakpoint(lineBreakpoint); }}
```

表 4.19: org.eclipse.debug.core.sourceLocators 延伸元件

```
<sourceLocator
  class="org.rageden.debug.core.sourcelookup.RSourceLookDirector"
  id="regedan.sourceLocator"
  name="RageDen Source Lookup Director">
</sourceLocator>
```

我們利用 Eclipse 偵錯框架提供的標準原始碼查看機制來完成本系統的原始碼查看功能。偵錯框架已實作了一個原始碼查看器 AbstractSourceLookupDirector，它可以沿著使用者搜尋路徑來搜尋資料夾下的檔案。唯一還沒實作的函式即是 initializeParticipants，此函式需要產生一組 ISourceLookupParticipants 來將 stack frames 對映至檔案名稱。

首先加入 regeden.sourceLocator 原始碼查看器至 launchConfigurationTypes 內的 sourceLocatorId 屬性，接著再新增表 4.19 的 SourceLocators 延伸元件。原始碼查看器需要實作表 4.20 的 RSourceLookDirector。它繼承自處理了大部分工作的 AbstractSourceLookupDirector，但仍需實作 initializeParticipants 函式以便初始化參與者的串列，並新增一個 RSourceLookupParticipant 至串列中。除此之外，還需要重新定義 getSourceElement 函式，當函式接收到一個 stack frame，且它繼承自 RStackFrame 時，則從 RStackFrame 的 getModelElement 函式取得此 stack frame 的抽象語法樹節點並回傳。表 4.21 的 RSourceLookupParticipant 則重新定義了 getSourceName 函式，如果它接收到 RStackFrame，會回傳 RStackFrame 的名稱標籤。

接著使用偵錯框架提供的原始碼搜尋計算器 (source path computer) 延伸元件來處理原始碼搜尋路徑。再度將表 4.22 內的 sourcePathComputerId 設爲 regeden.sourcePathComputer，並新增一個 SourcePathComputers 延伸元件。

表 4.20: 實作原始碼查看導引者：RSourceLookDirector.java

```
1  public class RSourceLookDirector
2    extends AbstractSourceLookupDirector {
3    public void initializeParticipants() {
4      addParticipants(new ISourceLookupParticipant[]{
5        new RSourceLookupParticipant()});
6    }
7    @Override
8    public Object getSourceElement(Object object) {
9      if (object instanceof RStackFrame)
10       return ((RStackFrame)object).
11                 getModelElement().eResource();
12     else if (object instanceof EObject)
13       return ((EObject) object).eResource();
14     return null;
15   }
16   @Override
17   public Object getSourceElement(IStackFrame stackFrame) {
18     return getSourceElement((Object) stackFrame);
19   }
20 }
```

表 4.21: 實作原始碼查看參與者：RSourceLookupParticipant.java

```
1 public class RSourceLookupParticipant
2   extends AbstractSourceLookupParticipant {
3   public String getSourceName(Object object)
4     throws CoreException {
5   if (object instanceof ModelStackFrame)
6     return ((RStackFrame)object).getName();
7   return null;
8   }
9 }
```

表 4.22: org.eclipse.debug.core.sourcePathComputers 延伸元件

```
1 <sourcePathComputer
2   class=
3     "org.rageden.debug.core.sourcelookup.
4      RSourcePathComputerDelegate"
5   id="regeden.sourcePathComputer">
6 </sourcePathComputer>
```

我們需實作表 4.23 的原始碼路徑計算器 RourcePathComputerDelegate，當一個屬於 ISourceContainer 的元素包含在原始檔內時，它會回傳一個單一元素來源的路徑。

接下來表 4.24 新增 debugModelPresentation 以便將來源元素對應至一個編輯器來顯示，

偵錯模型會透過 extension id 連接至偵錯模型顯示，對偵錯套件來說，偵錯元素介面 IDebugElement 需要知道自己屬於哪個偵錯模型，所以需在 RDebugElement 內實作 getModelIdentifier 函式，並回傳 rageden.debugModel。表 4.25 即爲偵錯模型顯示介面的實作，RDebugModelPresentation 實作 getEditorInput 與 getEditorId 函式以便將來源元素轉換成 editor input 與 id。

表 4.23: 實作原始碼路徑計算器代理：RSourcePathComputerDelegate.java

```
1  public ISourceContainer[] computeSourceContainers(
2    ILaunchConfiguration configuration,
3    IProgressMonitor monitor) throws CoreException {
4    String path = configuration.getAttribute(
5      DebugCorePlugin.ATTR_RAGEDEN_PROGRAM, (String)null);
6    ISourceContainer sourceContainer = null;
7    if (path != null) {
8      IResource resource =
9        ResourcesPlugin.getWorkspace().getRoot().
10       findMember(new Path(path));
11     if (resource != null) {
12       IContainer container = resource.getParent();
13       if (container.getType() == IResource.PROJECT)
14         sourceContainer =
15           new ProjectSourceContainer((IProject)container,
16                                      false);
17       else if (container.getType() == IResource.FOLDER)
18         sourceContainer =
19           new FolderSourceContainer(container, false);
20     }}
21   if (sourceContainer == null)
22     sourceContainer = new WorkspaceSourceContainer();
23   return new ISourceContainer[]{sourceContainer};
24 }
```

表 4.24: org.eclipse.debug.ui.debugModelPresentation 延伸元件

```
<debugModelPresentation
  class="org.rageden.debug.ui.
         presentation.RDebugModelPresentation"
  id="rageden.debugModel">
</debugModelPresentation>
```

表 4.25: 實作偵錯模型顯示：RDebugModelPresentation.java

```
1  public IEditorInput getEditorInput(Object element) {
2    if(element instanceof Resource)
3      return new FileEditorInput(
4        (IFile) ResourceUtil.getFile((Resource)element));
5    else if (element instanceof ILineBreakpoint)
6      return new FileEditorInput(
7        (IFile)((ILineBreakpoint)element).
8          getMarker().getResource());
9    return null;
10 }
11 public String getEditorId(IEditorInput input, Object element) {
12   if(element instanceof Resource &&
13       input instanceof IFileEditorInput)
14   try {
15     IFileEditorInput fileEditorInput =
16       (IFileEditorInput) input;
17     IEditorDescriptor editorDesc =
18       IDE.getEditorDescriptor(fileEditorInput.getFile(), true);
19     return editorDesc.getId();
20   }
21   catch (PartInitException e) { return null; }
22   return null;
23 }
```

第 五 章

系統範例

本章節將說明與示範如何定義一個 DSL，並藉由本系統自動生成執行與偵錯環境。首先我們使用 Xtext 來描述 DSL 的文法，生成相對應的編輯環境。接著再以 RGD 描述其自然語意及偵錯指令，生成執行與偵錯環境，協助使用者使用編輯器撰寫程式並偵錯與執行程式。在此我們以 Wu 等人 [31] 定義的具備函式呼叫的簡單機器人語言 — Robot 爲例，逐步闡述如何利用模型驅動技術快速生成 Robot 的執行與偵錯環境。

5.1 專案設置

RageDen 系統得在 Eclipse 平台下執行，並需安裝 Xtext 套件。除此之外，還需要安裝 Scala 語言的執行環境，以及 ScalaIDE 套件。最後再加入以下 RageDen 的五個套件：

- `org.rageden.rgd`
- `org.rageden.rgd.ui`
- `org.rageden.runtime`
- `org.rageden.rgd.debug.core`

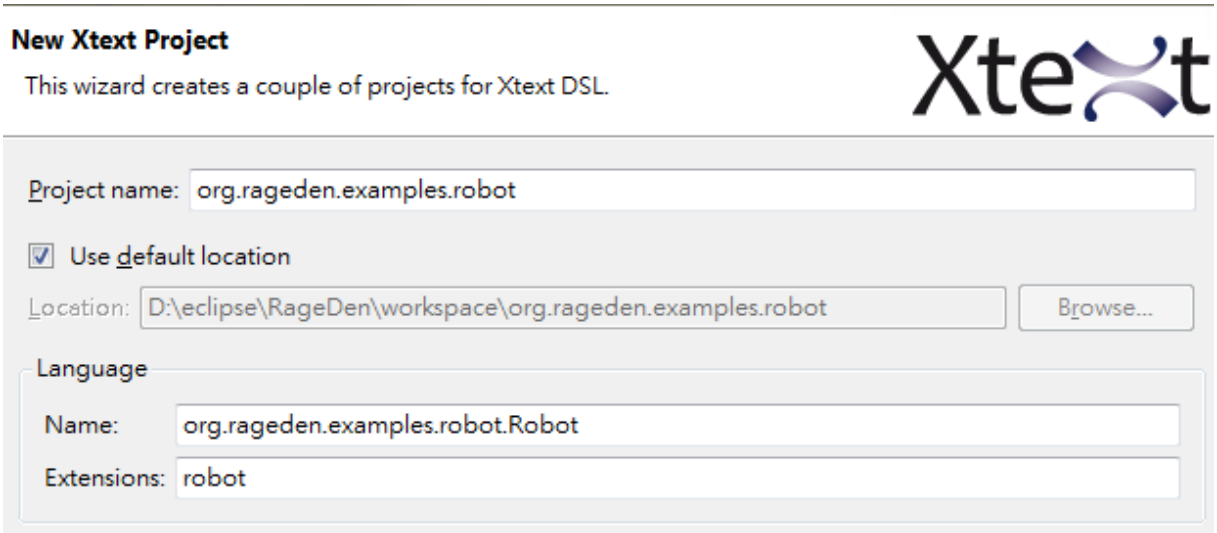

圖 5.1: 新增 Robot 的 Xtext 專案

- `org.rageden.rgd.debug.ui`

5.2 建立 Robot 的編輯環境

我們首先使用 Xtext 專案精靈，建立圖 5.1 的 Robot 專案。專案名稱爲 `org.rageden.examples.robot`，語言名稱爲 `org.rageden.examples.robot.Robot`，副檔名爲 robot。接著在 Robot.xtext 內加入表 3.1 的描述。再由 Robot.xtext 生成 Robot 的編輯環境，包括編輯器、解析器以及超模型等等，完成 Robot 的編輯環境的建立。

5.3 描述 Robot 自然語意及偵錯指令

建立好 Robot 的編輯環境之後，尚需描述 Robot 的自然語意及偵錯指令，以便生成執行與偵錯環境。首先新增如圖 5.2 的 Eclipse 套件專案，名稱爲 `org.rageden.examples.robot.execute`，並加入所需的套件。其中包括了執行 Scala 程式碼需要的函式庫、Robot 編輯器的核心套件，還有 RageDen 的執行平台。

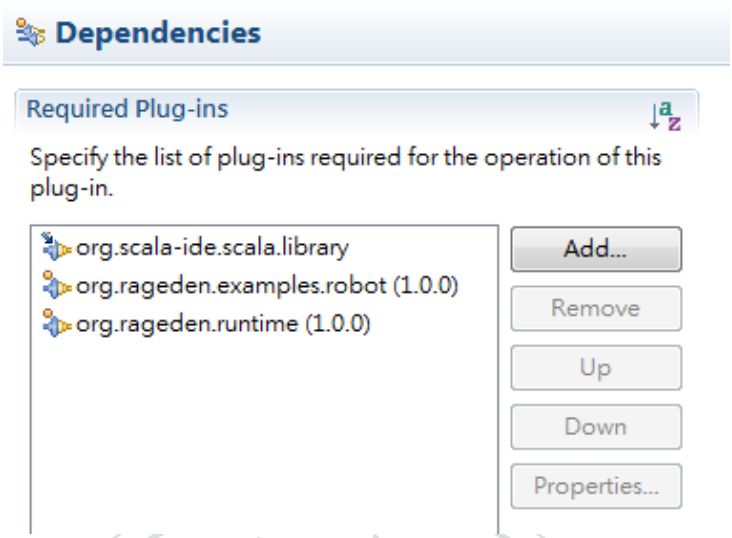

圖 5.2: Robot 需要的套件

接著在專案下新增一個名爲 model 的資料夾，並加入 main.rgd 與 robot.rgd 來描述語意。表 3.7 的 main.rgd 爲 RGD 的程式進入點。而表 5.1 的 robot.rgd 則爲自然語意的描述，一開始先匯入 Robot 的超模型，並宣告提取器。

接著宣告 `datatype Location = COORD of Int * Int` 資料型態來儲存機器人的座標，在表 5.2 的 `evalRobot` 規則集合會解譯整個 `Robot` 模型。首先比對參數是否爲 `Robot` 模型的根節點 `BOT`，並匹配 `Main` 包含的命令給 `cmds`。推論前提前，需要宣告 `StackFrame "main" @ main`，表示開始執行 main 的函式呼叫。然後呼叫 `update(COORD(0, 0))` 指示偵錯器目前機器人的座標爲 `(0, 0)`。接著呼叫 `evalCmds (cmds, COORD(0, 0), funcs)` 來解譯函式內的命令，其中 `cmds` 爲需要解譯的命令串列，`COORD(0, 0)` 爲目前機器人所在的座標，`funs` 則爲所以函式宣告的節點串列。最後加入 `endStackFrame`，表示此時已執行完函式呼叫並離開函式。

表 5.3 爲解譯 `Command` 模型串列的語意描述。我們希望以 `Command` 模型作爲執行一次 Step 的間隔。所以在每個解譯命令的 `evalCmd(cmd, coord, funcs)` 前均加入 `SuspendCheck cmd`，確保執行每個 `Command` 前，都會判斷是否需要暫停。而表5.4 則一一描述如何解譯每個命令。執行完一個命令之後，會獲得新的機器人座標 `coord2`，此

表 5.1: Robot.rgd: 提取器宣告

```
1  module org.rageden.examples.robot.execute.Robot
2
3  import org.rageden.examples.robot.robot.*
4
5  type Ident = String
6
7  extractor Robot = BOT of Function * List[Function]
8
9  extractor Function =
10   MAIN of List[Command]
11 | FUNC of Ident * List[Command]
12
13 extractor Command =
14   UP | DOWN | LEFT | RIGHT
15 | WHERE
16 | CALL of Ident
17 | SET of Int * Int
```

表 5.2: Robot.rgd: 解譯 Robot 模型

```
1 set evalRobot(Robot) {
2
3   rule StackFrame "main" @ main &
4        update(COORD(0, 0)) &
5        evalCmds(cmds, COORD(0, 0), funcs) &
6        endStackFrame
7        ---------------------------------
8        BOT(main as MAIN(cmds), funcs) => ()
9 }
```

時需要呼叫 update(coord2)，以便通知偵錯器更新目前函式下的變數。

當解譯至函式呼叫的節點時，需要表 5.5 的 evalCall 描述函式呼叫的語意。當欲呼叫的函式名稱與函式宣告內的函式名稱符合時，加入 StackFrame id @ funcs.first，指示此時進入函式呼叫。而在執行完函式內的命令後，同樣加入 endStackFrame 代表離開函式。

當機器人移動的時候，座標會改變，同時也需要呼叫表 5.6 的 update 規則集合，指示偵錯器更新目前函式下，區域變數的變值。我們在此規則內加入 Variable "x" : "int" = x 與 Variable "y" : "int" = y 兩個偵錯指令，通知偵錯器更新目前函式下的變數數值。

表 5.3: Robot.rgd: 解譯 Command 模型串列

```
1  set evalCmds(List[Command],
2               Location,
3               List[Function]): Location = {
4
5    axiom (Nil, coord,_) => coord
6
7    rule SuspendCheck cmd &
8         evalCmd(cmd, coord, funcs) => coord2 &
9         update(coord2) &
10        evalCmds(cmds, coord2, funcs) => coord3
11        ---------------------------------------
12        ((cmd::cmds), coord, funcs) => coord3
13 }
```

表 5.4: Robot.rgd: 解譯 Command 模型

```
1  set evalCmd(Command, Location, List[Function]): Location = {
2    rule y + 1 => y2
3         ---------------------------------------
4         (UP(), COORD(x, y),_) =>  COORD(x, y2)
5    rule y - 1 => y2
6         ---------------------------------------
7         (DOWN(), COORD(x, y),_) =>  COORD(x, y2)
8    rule x - 1 => x2
9         ---------------------------------------
10        (LEFT(), COORD(x, y),_) =>  COORD(x2, y)
11   rule x + 1 => x2
12        ---------------------------------------
13        (RIGHT(), COORD(x, y),_) =>  COORD(x2, y)
14   rule printr("( ") & printr(x) & printr(" ,") &
15        printr(y) & printrln(" )")
16        ---------------------------------------
17        (WHERE(), COORD(x, y),_) => COORD(x, y)
18   rule evalCall(id, coord, funcs, funcs) => coord2
19        ---------------------------------------
20        (CALL(id), coord, funcs) => coord2
21   axiom (SET(x, y),_,_) => COORD(x, y)
22 }
```

表 5.5: Robot.rgd: 解譯函式呼叫

```
1  set evalCall(Ident,
2               Location,
3               List[Function],
4               List[Function]): Location = {
5
6    rule id = fid &
7         StackFrame id @ funcs.first &
8         update(coord) &
9         evalCmds(cmds, coord, all) => coord2 &
10        endStackFrame
11        ------------------------------------------------
12        (id, coord, funcs as (FUNC(fid, cmds)::_), all)
13          => coord2
14
15   rule not id = fid &
16        evalCall(id, coord, rest, all) => coord2
17        ------------------------------------------------
18        (id, coord, FUNC(fid, cmds):: rest, all) => coord2
19
20   rule evalCall(id, coord, rest, all) => coord2
21        ----------------------------------------
22        (id, coord, MAIN(_):: rest, all) => coord2
23
24 }
```

表 5.6: Robot.rgd: 更新變數資訊

```
1 set update(Location) {
2   rule Variable "x" : "int" = x &
3        Variable "y" : "int" = y
4        ------------------------
5        COORD(x, y) => ()
6 }
```

5.4 生成執行元件並新增延伸元件

描述完 Robot 的語意後，如圖 5.3 在專案下新增一個 src-gen 的套件資料夾，RGD2Code 便會自動生成 Robot 的執行元件。此時需要新增圖 5.4 的 `org.rageden.runtime.execute` 的延伸元件，以便讓直譯與偵錯器通用框架能夠載入 Robot 執行元件。

5.5 執行與偵錯 Robot 程式碼

完成上述套件，我們即可在 Eclipse 執行與偵錯程式。圖 5.5 爲使用者實際操作情況，使用者除了可以新增與移除中斷點外，也可在偵錯模式下，由 Debug view 觀察程式目前的 stack frame，並執行 Step Into、Step Over、Step Return 操作來追蹤程式流程。由 Variables view 也可觀測目前 stack frame 下變數的數值。

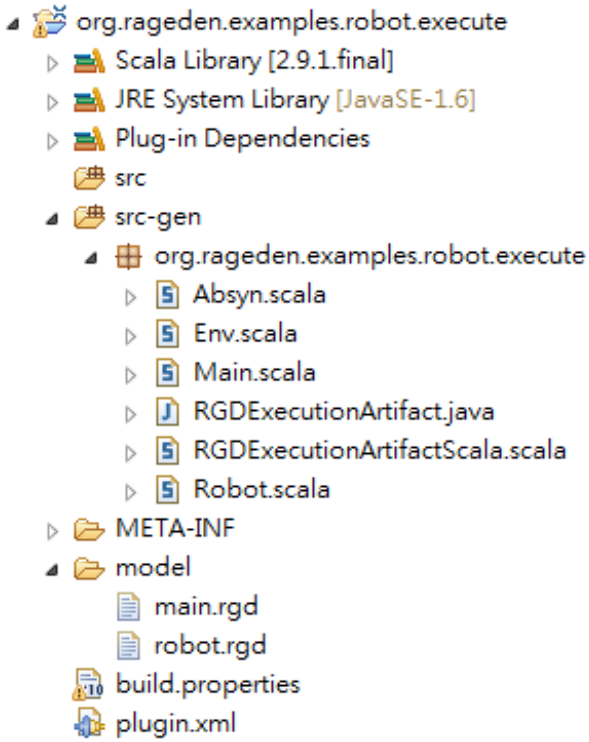

圖 5.3: 生成 Robot 執行元件

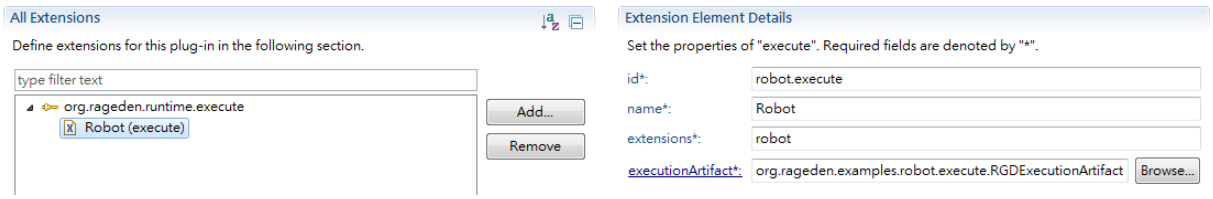

圖 5.4: 新增執行元件的延伸元件

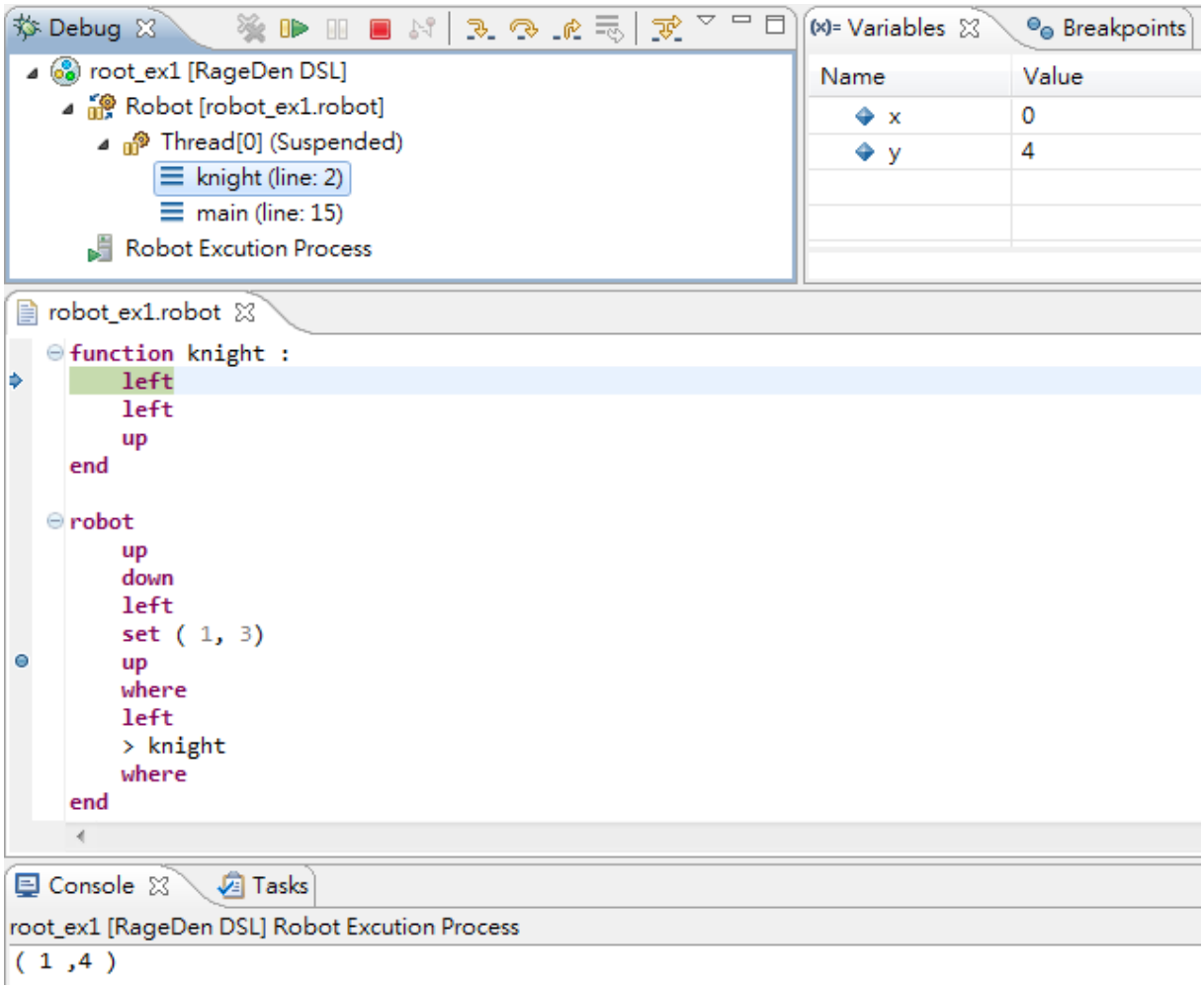

圖 5.5: 偵錯 ex1.robot

第 六 章

結論與未來展望

6.1 結論

相較於 DSL 的廣泛利用，相關的語言直譯器或偵錯器等輔助開發環境卻仍舊貧乏，然而針對特定語言從頭到尾量身打造其開發環境需要付出大量成本。若能降低建構輔助開發環境的難度，則能促使更多開發者爲 DSL 建構輔助開發環境，協助使用者撰寫程式。

因此本研究提出了一套名爲 RageDen 的 DSL 執行與偵錯環境生成架構，開發人員只要利用我們定義的 RGD 語言來描述語言的自然語意以及偵錯指令，即可快速生成執行與偵錯環境，並提供了完善的使用者圖形介面方便 DSL 使用者撰寫與偵錯程式。

6.2 未來展望

我們實作的 RageDan 系統目前已能達到快速生成 DSL 執行與偵錯環境的目的，但仍有部分可以改良與擴充實作與功能，以下將一一列出：

- 提高執行效率：我們雖然實作 RGD2Code，將自然語意轉換並生成執行元件，但並未最佳化程式碼。由於直譯元件的程式碼包含了許多 backtrack 的演算法，當使用者撰寫並執行輯運算複雜度太高的 DSL 程式碼時，執行元件需要大量呼叫重複且冗長的 backtrack 來進行解譯，造成執行效率低落。所以若最佳化直譯元件的程式碼，去除大部分非必要的 backtracking，將可提高執行效率。

- 引入 Java 函式庫：我們目前只提供了一些必要的函式庫供使用者在描述語意時呼叫。由於 RGD 生成的執行元件是在 JVM 上運行的 Scala 程式碼，若能直接整合 Java，讓使用者在描述語意時可以引用 Java 現有的函式庫以及相關類別，則可獲得極大的幫助。

- 支援多執行緒：由本系統的直譯與偵錯器通用框架目前僅支援單一執行緒的程式，意即使用者無法撰寫多工或多執行緒的程式。未來若支援多執行緒，則更能擴大 RageDen 的使用範圍。

- 擴充中斷點功能：我們的偵錯器目前僅支援以行號判斷的行中斷點。然而偵錯器中還有其他種類的中斷點，像是 watchpoint、method breakpoint、exception breakpoint 等等，而中斷點內也加入額外的表達式判斷，或是其他條件式的判斷等等。擴充這些中斷點功能，將會讓使用者能有更多方式來追蹤程式的執行流程。

- 增加數值的型態：本偵錯器目前已能呈現基本型態與陣列型態的數值，未來可以增加更複雜的結構型態實作，以便讓偵錯器能夠呈現更多種數值結構給使用者觀看。

參 考 文 獻

[1] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, 2000.

[2] O. M. G. Inc., "Omg model driven architecture." `http://www.omg.org/mda/`, 2011.

[3] E. Foundation, "Eclipse." `http://www.eclipse.org/`, 2012.

[4] D. Wright and C. Windatt, "Debug platform: The basics." `http://www.eclipse.org/eclipse/debug/documents/ec_2008/Debug_Tutorial_Basic_2008.ppt`, 2008.

[5] F. N. Hanne Riis Nielson, *Semantics with Applications: An Appetizer*. Springer, 2007.

[6] G. Winskel, *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[7] G. D. Plotkin, "A structural approach to operational semantics," *J. Log. Algebr. Program.*, vol. 60-61, pp. 17–139, 2004.

[8] G. Kahn, "Natural semantics," in *STACS* (F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, eds.), vol. 247 of *Lecture Notes in Computer Science*, pp. 22–39, Springer, 1987.

[9] R. S. Alfred V. Aho and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2006.

[10] "The lex & yacc page." `http://dinosaur.compilertools.net/`.

[11] "Relational meta-language and tools." `http://www.ida.liu.se/~pelab/rml/`, 2008.

[12] "k-framework." `http://code.google.com/p/k-framework/`, 2011.

[13] "Asf+sdf." `http://www.meta-environment.org/Meta-Environment/ASF%2bSDF`, 2010.

[14] "Stratego/xt." `http://strategoxt.org/`, 2011.

[15] "The txl programming language." `http://www.txl.ca/`, 2011.

[16] D. Prawitz, *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, 2006.

[17] T. Despeyroux, "Typol: A formalism to implement natural semantics," research report, INRIA Sophia-Antipolis, 1988.

[18] M. Pettersson, *Compiling Natural Semantics*, vol. 1549 of *Lecture Notes in Computer Science*. Springer, 1999.

[19] A. Blewitt, "Getting started with eclipse plug-ins: understanding extension points." `http://www.eclipsezone.com/eclipse/forums/t93753.html`, 2007.

[20] "Eclipse java development tools." `http://www.eclipse.org/jdt/`, 2012.

[21] "Plug-in development environment." `http://www.eclipse.org/pde/`, 2012.

[22] T. E. Foundation, "Xtext." `http://www.eclipse.org/Xtext/`, 2011.

[23] "Eclipse modeling framework project." `http://eclipse.org/modeling/emf/`, 2012.

[24] T. Parr, "Antlr parser generator." http://www.antlr.org/, 2012.

[25] T. E. Foundation, "Xtend." http://www.eclipse.org/Xtext/xtend/, 2011.

[26] "Dsl debugging framework." http://students.cis.uab.edu/wuh/DDF/index.html, 2008.

[27] "Eprovide." http://eprovide.sourceforge.net/, 2008.

[28] "Spoofax." http://strategoxt.org/Spoofax, 2011.

[29] R. T. Lindeman, L. C. L. Kats, and E. Visser, "Declaratively defining domain-specific language debuggers," in *Generative Programming and Component Engineering, 7th International Conference, GPCE 2011, Proceedings* (E. Denney and U. P. Schultz, eds.), ACM, 2011.

[30] M. Fowler, "Domain specific language." http://www.martinfowler.com/bliki/DomainSpecificLanguage.html, 2009.

[31] H. Wu, J. Gray, and M. Mernik, "Grammar-driven generation of domain-specific language debuggers," *Software : Practice and Experience*, vol. 38, pp. 1073–1103, 2008.

[32] D. A. Sadilek and G. Wachsmuth, "Prototyping visual interpreters and debuggers for domain-specific modelling languages," in *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, (Berlin, Heidelberg), pp. 63–78, Springer-Verlag, 2008.

[33] J. F. Andreas Blunk and D. A. Sadilek, "Modelling a debugger for an imperative voice controllanguage," in *Proceedings of the 14th international SDL conference on Design for motes and mobiles*, SDL'09, (Berlin, Heidelberg), pp. 149–164, Springer-Verlag, 2009.

[34] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism sdf," *SIGPLAN Not.*, vol. 24, pp. 43–75, November 1989.

[35] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik, "Weaving a debugging aspect into domain-specific language grammars," in *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pp. 1370–1374, ACM, 2005.

[36] G. Kiczales, J. Lamping, and etc., "Aspect-oriented programming," in *European Conference on Object-Oriented Programming(ECOOP)*, pp. 220–242, June 1997.

[37] M. T. Robin Milner and R. Harper, *The Definition of Standard ML*. The MIT Press, 1997.

[38] T. E. Foundation, "We have lift-off: The launching framework in eclipse." `http://www.eclipse.org/articles/Article-Launch-Framework/launch.html`.

[39] D. Wright and B. Freeman-Benson, "How to write an eclipse debugger." `http://www.eclipse.org/articles/Article-Debugger/how-to.html/`.

[40] M. Odersky, "The scala programming language." `http://www.scala-lang.org/`, 2012.

附錄A：RGD 文法規則

```
1
2  grammar org.rageden.rgd2code.RGD
3    with org.eclipse.xtext.common.Terminals
4
5  generate rGD "http://www.rageden.org/rgd2code/RGD"
6
7  Module:
8    'module' name=QualifiedName defs+=Definition*
9  ;
10 Definition returns Definition:
11   Import | TypeDef | ValDef | DataType | RuleSet | Extractor
12 ;
13 Import returns Import:
14   'import' importedNamespace=QualifiedNameWithWildCard
15 ;
16 TypeDef returns TypeDef:
17   'type' name=ID '=' type=TypeRef
18 ;
```

```
19 ValDef:
20   'val' name=ID '=' exp=Exp
21 ;
22 DataType :
23   'datatype' name=ID '=' alters=AlternativeSeq
24 ;
25 Extractor :
26   'extractor' name=ID '=' alters=AlternativeSeq
27 ;
28 AlternativeSeq:
29   seq+=Alternative ('|' seq+=Alternative)*
30 ;
31 Alternative:
32   name=ID ('of' args+=TypeRef ('*' args+=TypeRef)*)?
33 ;
34 Type returns Type:
35   TypeDef | DataType | Extractor
36 ;
37 TypeRef returns Type:
38   ListType | OptionType | TupleType | SimpleTypeRef
39 ;
40 ListType returns Type:
41   {ListType} 'List' '[' type=TypeRef ']'
42 ;
43 OptionType returns Type:
```

```
44    {OptionType} 'Option' '[' type=TypeRef ']'
45  ;
46  TupleType returns Type:
47    '(' TypeRef (
48      ({TupleType.types+=current} '*' types+=TypeRef)+
49    | ({TupleType.types+=current} ',' types+=TypeRef)+
50    ) ')'
51  ;
52  SimpleTypeRef returns Type:
53    {ReferencedType} referenced=[Type|QualifiedName]
54  | {RawType} primty=PrimitiveType
55  ;
56  enum PrimitiveType:
57    BYTE='Byte' | SHORT='Short' | INT='Int'
58  | LONG='Long' | CHAR='Char' | STRING='String'
59  | FLOAT='Float' | DOUBLE='Double' | BOOL='Boolean'
60  ;
61  TypeSeq returns TypeSeq:
62    {TypeSeq} (seq+=TypeRef (',' seq+=TypeRef)*)?
63  ;
64  RuleSet:
65    'set' name=ID '(' argstype=TypeSeq ')'
66      (':' returntype=TypeSeq '=' )? '{' (clauses+=Clause)* '}'
67  ;
68  Clause returns Clause:
```

```
69    Axiom | Rule
70  ;
71  Axiom:
72    'axiom' args=Pat (result=Result)?
73  ;
74  Rule:
75    'rule' premises+=DebugPremise ('&' premises+=DebugPremise)*
76      '-''-' + args=Pat (result=Result)?
77  ;
78  Result returns Exp:
79    '=>' ({FailExp} 'fail' | Exp)
80  ;
81  DebugPremise returns DebugPremise:
82    DebugInstruction | Premise
83  ;
84  Label returns Label:
85    ident=QualifiedName | StringLiteral
86  ;
87  DebugInstruction:
88    {ActivationRecordStart}
89      'StackFrame' label=Label '@' node=SimpleExp
90  | {ActivationRecordEnd} 'endStackFrame'
91  | {VariableUpdate}
92      'Variable' label=Label ':' type=Exp '=' value=Exp
93  | {SuspendCheck} 'SuspendCheck' node=SimpleExp
```

```
94  ;
95  Premise:
96    EqualPremise | NotPremise
97  | LetPremise   | CallPremise
98  ;
99  EqualPremise:
100   ident=ID '=' exp=Exp
101 ;
102 NotPremise:
103   ('not'|'!') notp=Premise
104 ;
105 LetPremise:
106   'let' pat=Pat '=' exp=Exp
107 ;
108 CallPremise:
109   OpCall | FunCall
110 ;
111 OpCall:
112   op=Operation '=>' outpat=Pat
113 ;
114 Operation returns Operation:
115   ({BinaryOp} left=SimpleExp operator=(BinOp|RelOp)
116 | {UnaryOp} operator=UnOp) right=SimpleExp
117 ;
118 BinOp:
```

```
119    '+' | '-' | '*' | '/' | '%'
120  ;
121  RelOp:
122    '>' | '>=' | '<' | '<=' | '==' | '!='
123  ;
124  UnOp: '-'
125  ;
126  FunCall:
127    ident=QualifiedName '(' argsexp=ExpSeq ')' ('=>' outpat=Pat)?
128  ;
129  Exp returns Exp :
130    TupleExp ({ConsExp.left=current} '::' right=Exp)?
131  ;
132  TupleExp returns Exp:
133    {TupleExp} '(' (tuple+=Exp ( ',' tuple+=Exp)*)? ')'
134  | StructExp
135  ;
136  StructExp returns Exp :
137    {StructExp} ident=QualifiedName
138      '(' (args+=Exp (',' args+=Exp)*)? ')'
139  | SimpleExp
140  ;
141  SimpleExp returns Exp :
142    {LiteralExp} literal=Literal
143  | {IdentExp} ident=QualifiedName
```

```
144 | {ListExp}'[' (list+=Exp (',' list+=Exp)*)? ']'
145 ;
146 ExpSeq returns Exp:
147   {ExpSeq} (seq+=Exp (',' seq+=Exp)* )?
148 ;
149 Pat returns Pat:
150   {BindingPat} id=ID 'as' pat=Pat
151 | ConsPat
152 ;
153 ConsPat returns Pat:
154   TuplePat ({ConsPat.left=current } '::' right=ConsPat)?
155 ;
156 TuplePat returns Pat:
157   {TuplePat} '(' (tuple+=Pat (',' tuple+=Pat)*)? ')'
158 | StructPat
159 ;
160 StructPat returns Pat:
161   {StructPat} ident=QualifiedName
162     '(' (args+=Pat (',' args+=Pat)*)? ')'
163 | SimplePat
164 ;
165 SimplePat returns Pat:
166   {WildPat}'_'
167 | {LiteralPat} literal=Literal
168 | {IdentPat} ident=QualifiedName
```

```
169 | {ListPat} '[' (list+=Pat (',' list+=Pat)*)? ']'
170 ;
171 Literal:
172   IntLiteral | FloatLiteral | BoolLiteral
173 | CharLiteral | StringLiteral | {NullLiteral} key='Null'
174 ;
175 IntLiteral returns IntLiteral:
176   value=INT
177 ;
178 FloatLiteral returns FloatLiteral:
179   value=FLOAT
180 ;
181 CharLiteral returns CharLiteral:
182   value=CHAR
183 ;
184 BoolLiteral:
185   value=BOOL
186 ;
187 StringLiteral returns StringLiteral:
188   value=RSTRING
189 ;
190 terminal FLOAT:
191   '-'?('0'..'9')+'.'+('0'..'9')+
192 ;
193 enum BOOL:
```

```
194   TRUE = 'true' | FALSE = 'false'
195 ;
196 terminal CHAR:
197   "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'"'|"'"|'\\') | !('\\'|'"') )
          "'"
198 ;
199 terminal RSTRING        :
200   '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"')
         )* '"'
201 ;
202 QualifiedName:
203   ID ('.' ID)*
204 ;
205 QualifiedNameWithWildCard:
206   QualifiedName '.*'?
207 ;
```