ดีบักเกอร์ชาญฉลาดสำหรับภาษาโปรล็อก

INTELLIGENT PROLOG DEBUGGER

นายนัทธพงศ์ จึงธีรพานิช

นายชัยโรจน์ ศิษย์ธนานันท์

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2543

ปริญญานิพนธ์ปีการศึกษา 2543

ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง ดีบักเกอร์ชาญฉลาดสำหรับภาษาโปรล็อก

Intelligent Prolog Debugger

ผู้จัดทำ

1. นายนัทธพงศ์ จึงธีรพานิช รหัสประจำตัว 40010380
2. นายชัยโรจน์ ศิษย์ธนานันท์ รหัสประจำตัว 40010386

_____ อาจารย์ที่ปรึกษา

(ดร. วิศิษฏ์ หิรัญกิตติ)

สารบัญ

หน้าที่

สารบัญรูปภาพ

หน้าที่

บทที่ 1
บทนำ

เป็นที่ยอมรับดีว่า โปรแกรมที่เขียนขึ้นมักจะมีข้อผิดพลาดอยู่เสมอ จึงจำเป็นต้องมีขั้นตอนการตรวจสอบความถูกต้อง ซึ่งจุดประสงค์คือ การทำให้มั่นใจว่าโปรแกรมจะทำงานได้ถูกต้องเมื่อนำไปใช้งานจริง หากการตรวจสอบพบความผิดพลาดก็ต้องทำการแก้ไข ซึ่งขั้นตอนนี้คือการดีบัก (Debugging) นั่นเอง การดีบักเป็นขั้นตอนที่ยาก ดังจะเห็นได้จากการที่โปรแกรมเมอร์ต้องเสียเวลาไม่น้อยไปกับการดีบัก จึงเกิดแนวคิดที่จะสร้างระบบที่ช่วยในการดีบัก ในปัจจุบันได้มีตัวอย่างของระบบเหล่านี้ซึ่งสามารถช่วยโปรแกรมเมอร์ในการหาข้อผิดพลาดของโปรแกรม (Program Diagnosis) เทคนิคหนึ่งที่ประสบความสำเร็จในการหาข้อผิดพลาดคือ เทคนิค Algorithmic Program Debugging ดีบักเกอร์ที่ได้พัฒนาขึ้นจึงประยุกต์ใช้เทคนิคนี้

1.1 การตรวจสอบความถูกต้องของโปรแกรม (Program Verification and Validation)

"The understanding of the theory of a routine may be greatly aided by providing, at the time of construction, one or two statements concerning the state of the machine at well chosen points. ... In the extreme form of theoretical method, a watertight mathematical proof is provided for the assertions. In the extreme form of experimental method, the routine is tried out on the machine with a variety of initial conditions and is pronounced fit if the assertions hold in each case. Both methods have their weaknesses."

- A. M. Turing, Ferranti Mark I Programming Manual (1950)

วิธีที่สำคัญในการตรวจสอบความถูกต้องมี 2 วิธีคือ

- **การตรวจสอบความถูกต้องแบบสแตติก (Static Verification)**
  เป็นการตรวจสอบความถูกต้องโดยแสดงให้เห็นว่าโปรแกรมจะทำงานตามที่สเปกซิฟิเคชั่น (Specification) กำหนด โดยอาจแสดงอย่างเป็นทางการ ในรูปของบทพิสูจน์ (Proof) เหมือนการพิสูจน์ทฤษฎีทางคณิตศาสตร์ หรืออาจแสดงอย่างไม่เป็นทางการด้วยคำอธิบายที่มีเหตุผล เพื่ออ้างว่าโปรแกรมจะทำงานได้ถูกต้อง ทั้งนี้หากการพิสูจน์นั้นถูกต้องจริง ก็มั่นใจได้ว่าโปรแกรมจะทำงานตามสเปกซิฟิเคชั่น แต่ก็ไม่ได้หมายความว่าโปรแกรมจะทำงานถูกต้องตามความต้องการของผู้ใช้ เพราะสเปกซิฟิเคชั่นอาจจะมีความผิดพลาด หรือไม่สมบูรณ์ก็เป็นได้

- **การทดสอบโปรแกรม (Program Testing)**

  เป็นการตรวจสอบความถูกต้องโดยการทดสอบโปรแกรม คือทดลองป้อนอินพุตให้โปรแกรมประมวลผล หากโปรแกรมให้ผลลัพธ์ที่ไม่ถูกต้อง แสดงว่าโปรแกรมมีข้อผิดพลาดอยู่ แต่หากโปรแกรมให้ผลลัพธ์ถูกต้อง ยังไม่สามารถสรุปได้ว่าโปรแกรมนั้นปราศจากข้อผิดพลาด อาจจะมีอินพุตอื่นที่โปรแกรมทำงานไม่ถูกต้องก็เป็นได้

  สมมติฐานหนึ่งของการทดสอบโปรแกรม คือ ผู้ที่จะทำการทดสอบโปรแกรมจะต้องรู้ว่าผลลัพธ์ที่เกิดจากการประมวลผลของอินพุตที่ใช้ทดสอบคืออะไร หรืออย่างน้อยก็ต้องบอกได้ว่ามันผิดหรือไม่ ตัวอย่างเช่น หากเขียนโปรแกรมหาค่าแฟกตอเรียล อาจทำการทดสอบโปรแกรมด้วยอินพุต 0,1,2,5 ซึ่งทราบว่าผลลัพธ์ที่ควรจะได้คือ 1,1,2,120 ตามลำดับ แต่หากทดสอบด้วยอินพุต 10 ซึ่งไม่รู้ว่าผลลัพธ์คืออะไร เมื่อโปรแกรมประมวลผลออกมาเช่นได้ 3626800 ก็ไม่สามารถบอกได้ว่าผิดหรือไม่ สรุปก็คือ ในการทดสอบโปรแกรม จำเป็นต้องมี เซตของอินพุตและผลลัพธ์ที่ควรจะเป็น เซตนี้มักเรียกว่า ข้อมูลทดสอบ (Test Data) การที่ต้องรู้ผลลัพธ์ที่ควรจะเป็นนี้ หากพูดในทางลอจิกคือ ต้องมี Intended Model ของโปรแกรม ถึงแม้ว่าการทดสอบโปรแกรมจะไม่สามารถรับประกันความถูกต้องของโปรแกรมได้ การที่มีข้อมูลทดสอบที่เหมาะสมและมีปริมาณมากพอ ก็ช่วยให้มั่นใจความถูกต้องได้ในระดับหนึ่ง

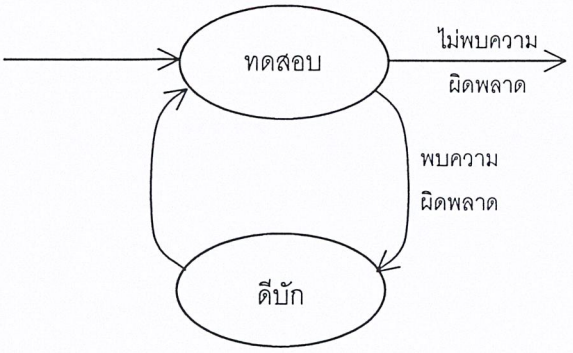
ในการพัฒนาโปรแกรมจริงๆ มักจะไม่สามารถจะกำหนดสเปกซิฟิเคชั่นที่ถูกต้องและสมบูรณ์ได้ กล่าวอีกนัยหนึ่งสเปกซิฟิเคชั่นก็อาจจะมีข้อผิดพลาดได้เช่นเดียวกับโปรแกรม สิ่งนี้เป็นปัญหาสำคัญของการทำการตรวจสอบความถูกต้องแบบสแตติก ซึ่งทำให้วิธีนี้ไม่นิยมใช้โดยเฉพาะอย่างยิ่งกับโปรแกรมขนาดใหญ่ การตรวจสอบความถูกต้องด้วยการทดสอบเป็นที่แพร่หลายมากกว่าเพราะทำได้ง่าย ถึงแม้จะต้องยอมรับว่าการทดสอบไม่สามารถรับประกันความถูกต้อง

1.2 การดีบักและดีบักเกอร์ (Debugging and Debugger)

เมื่อการทดสอบโปรแกรม (Testing) พบอินพุตที่โปรแกรมให้ผลลัพธ์ไม่ถูก แสดงว่าโปรแกรมมีข้อผิดพลาด หรือที่เรียกว่า บัก (Bug) เป็นหน้าที่ของโปรแกรมเมอร์ที่จะทำการกำจัดข้อผิดพลาดนั้นออกจากโปรแกรม จึงเป็นที่มาของกระบวนการที่เรียกว่า การดีบัก (Debugging) จึงนิยามการดีบักได้ดังนี้

การดีบัก (Debugging) เป็นกระบวนการที่ทำเมื่อการทดสอบพบว่าโปรแกรมมีข้อผิดพลาด (Bug) จุดมุ่งหมายของการดีบักก็คือ การกำจัดข้อผิดพลาด เพื่อให้โปรแกรมทำงานได้ถูกต้อง

จึงเห็นได้ว่า การทดสอบ (Testing) และการดีบัก (Debugging) เป็นกระบวนการที่ทำต่อเนื่องกัน และวนเป็นวัฏจักร (Cycle) ดังแสดงในรูปที่ 1-1 เมื่อการทดสอบกับอินพุตจำนวนหนึ่งพบความผิดพลาด จึงเข้าสู่กระบวนการดีบัก และเมื่อทำการกำจัดข้อผิดพลาดออกไปแล้ว ต้องกลับไปทำการทดสอบใหม่ว่ายังมีข้อผิดพลาดอีกหรือไม่ ทั้งนี้การทดสอบควรทำกับอินพุตที่ทดสอบไปแล้วด้วย เพื่อให้มั่นใจว่าการดีบักได้แก้ไขข้อผิดพลาดจริง และไม่ได้สร้างข้อผิดพลาดอื่นๆ เพิ่ม



รูปที่ 1-1 การทดสอบและการดีบัก

ดีบักเกอร์ (Debugger) มีความหมายกว้างๆ ถึงบุคคล โปรแกรม หรือสิ่งที่ดำเนินการหรือมีส่วนช่วยในการดีบัก แต่เมื่อ พูดถึงดีบักเกอร์มักจะหมายถึงโปรแกรมที่ช่วยในการดีบัก

เมื่อโปรแกรมเมอร์ทราบว่าโปรแกรมมีข้อผิดพลาด การที่จะกำจัดข้อผิดพลาดนั้น โปรแกรมเมอร์จะหาว่าข้อผิดพลาดอยู่บริเวณใด เช่นในฟังก์ชั่น (Function) อะไรภายในโปรแกรม แล้วโปรแกรมเมอร์จะทำการวิเคราะห์บริเวณที่คิดว่าผิดพลาดเพื่อหาสาเหตุ แล้วทำการออกแบบการแก้ไข จึงสามารถสรุปได้ว่ากระบวนการดีบักประกอบด้วยขั้นตอนหลักๆ 2 ขั้นตอนคือ

1) **การวินิจฉัยข้อผิดพลาด (Diagnosis, Bug Localization)** คือขั้นตอนการหาว่าข้อผิดพลาดอยู่ที่บริเวณใดในโปรแกรม
2) **การแก้ไขข้อผิดพลาด (Correction)** คือขั้นตอนการวิเคราะห์ แล้วทำการแก้ไขข้อผิดพลาด

ใน IDE (Integrated Development Environment) ของภาษาคอมพิวเตอร์ในปัจจุบัน จะมีเครื่องมือช่วยการดีบัก (Debugging Tools) ซึ่งส่วนใหญ่ทำหน้าที่เป็นเทรเซอร์ (Tracer) คือช่วยให้โปรแกรมเมอร์ดูการทำงานทีละขั้นของโปรแกรม โดยสามารถดูค่าตัวแปรได้ (Variable watching) จะเห็นว่าเครื่องมือเหล่านี้ไม่ได้ทำการหาข้อผิดพลาดเอง มันทำหน้าที่เพียงอำนวยความสะดวกโปรแกรมเมอร์ในการหาข้อผิดพลาด (Diagnosis)

1.3 การดีบักแบบอัตโนมัติ (Automated Debugging)

แต่เดิมจุดมุ่งหมายในการศึกษาการดีบักแบบอัตโนมัติ (Automated Debugging) คือการสร้างระบบที่สามารถดีบักได้ด้วยตัวของมันเอง (Fully Automatic) จุดมุ่งหมายนี้ยังไม่เคยสำเร็จ และคงจะไม่สำเร็จได้ในอนาคตอันใกล้ ปัจจุบันจุดมุ่งหมายได้เปลี่ยนไปเป็นการสร้างระบบแบบกึ่งอัตโนมัติ (Semi Automatic) และมีลักษณะโต้ตอบกับผู้ใช้ (Interactive) นั่นคือระบบจะทำหน้าที่เหมือนเป็นผู้ช่วย (Assistant) ของโปรแกรมเมอร์ ตัวอย่างของระบบประเภทนี้ได้แก่

- Algorithmic Program Debugging (APD), Prolog, Yale University
- Annalyzer, Ada, Stanford University
- Aspect, CLU, MIT
- Debussi, Lisp, MIT

- Program error-locating assistant system (PELAS), Pascal, Oakland University
- Rational Debugging (RD), Prolog, University of Lisbon

ระบบเหล่านี้จะช่วยโปรแกรมเมอร์ในการหาข้อผิดพลาด นั่นคือทำหน้าที่เป็นตัววินิจฉัย (Diagnoser) ในปัจจุบันยังไม่มีระบบดีบักเกอร์ใดสามารถทำการแก้ไขโปรแกรม (Correction) ได้อย่างเป็นที่น่าพอใจ แสดงให้เห็นว่าการจะให้คอมพิวเตอร์แก้ไขข้อผิดพลาด ยากกว่าการหาข้อผิดพลาดมาก การแก้ข้อผิดพลาดต้องมีทั้งการเปลี่ยนแปลงโปรแกรมในส่วนที่ผิด และเพิ่มเติมส่วนที่ขาดหาย การทำให้ดีบักเกอร์แก้ไขโปรแกรมได้ยากกว่าการสังเคราะห์โปรแกรมแบบอัตโนมัติ (Automatic Program Synthesizing) หากสามารถสร้างตัวแก้ข้อผิดพลาดได้สำเร็จ แล้วให้โปรแกรมเปล่าไปทำการแก้ข้อผิดพลาด (Correct) ถ้าดีบักเกอร์สามารถแก้ข้อผิดพลาด จนได้โปรแกรมที่ถูกต้อง แสดงว่าสามารถสังเคราะห์โปรแกรมได้

ข้อสังเกตหนึ่งคือ ในชีวิตจริงพบว่าโดยส่วนใหญ่โปรแกรมเมอร์จะเสียเวลากับการหาข้อผิดพลาดมากกว่าการแก้ไข เช่น ในโปรแกรมที่มีขนาดใหญ่ เมื่อการทดสอบโปรแกรมพบข้อผิดพลาด โปรแกรมเมอร์อาจเสียเวลาเป็นวันในการหาว่าข้อผิดพลาดอยู่ที่ไหน แต่ส่วนใหญ่เมื่อพบข้อผิดพลาดแล้ว โปรแกรมเมอร์มักแก้ไขได้โดยใช้เวลาไม่นาน ตัวอย่างนี้แสดงให้เห็นว่า ถึงแม้จะไม่สามารถสร้างดีบักเกอร์ที่ช่วยแก้ไขข้อผิดพลาด หากดีบักเกอร์สามารถช่วยในการหาข้อผิดพลาดได้อย่างมีประสิทธิภาพ ก็มีประโยชน์ไม่น้อยเลยทีเดียว

เทคนิคที่ใช้ในการพัฒนาตัววินิจฉัยมี 2 เทคนิคหลักๆ ซึ่งต่างกันที่ข้อมูลที่ใช้ในการวิเคราะห์ในการหาข้อผิดพลาด

- **เทคนิคแบบสแตติก (Static Technique)** เทคนิคนี้จะใช้ซอร์สโค้ด (source code) ของโปรแกรมในการวิเคราะห์
- **เทคนิคแบบไดนามิก (Dynamic Technique)** เทคนิคนี้จะใช้ Execution Traces ของโปรแกรมในการวิเคราะห์

ในการหาข้อผิดพลาด โปรแกรมเมอร์มักจะใช้ทั้ง 2 เทคนิคควบคู่กันไป ตัวอย่างเช่น ขั้นแรกโปรแกรมเมอร์มักดูจากซอร์สโค้ดก่อน ถ้าโปรแกรมขนาดไม่ใหญ่ และความผิดพลาดค่อนข้างชัดเจน โปรแกรมเมอร์ก็มักพบข้อผิดพลาดได้ แต่หากไม่พบหรือเกิดความไม่แน่ใจว่าเป็นจุดที่ผิดพลาดหรือไม่ โปรแกรมเมอร์อาจแทรกคำสั่งที่แสดงสถานะของการประมวลผล เช่นใช้คำสั่ง print เพื่อแสดงค่าของตัวแปร หรือโปรแกรมเมอร์อาจใช้เทรเซอร์เพื่อดูการทำงานของโปรแกรม ซึ่งก็คือการใช้เทคนิคแบบไดนามิกนั่นเอง

1.4 เทคนิค Algorithmic Program Debugging

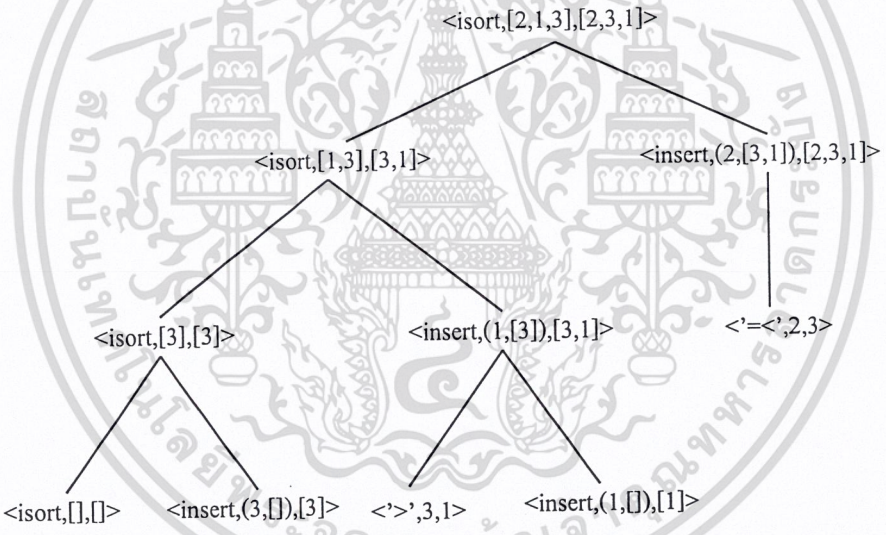
เทคนิค Algorithmic Program Debugging (APD) ซึ่งคิดค้นโดยศาสตราจารย์ Ehud Y. Shapiro เป็นเทคนิคที่มีประสิทธิภาพและเป็นที่ยอมรับ เห็นได้จากการที่มีผู้นำไปขยายต่ออย่างกว้างขวาง เทคนิคนี้ถูกออกแบบสำหรับภาษาที่มีลักษณะเป็นโปรซีเยอร์ (Procedure) โดยผู้คิดค้นได้ทำการสร้างตัววินิจฉัย

สำหรับภาษาโปรล็อก แต่ก็สามารถไปประยุกต์ใช้กับภาษาอื่นๆ ได้ เทคนิคนี้เป็นแบบไดนามิก โดยทำการวิเคราะห์จากขั้นตอนของการเอ็กซิคิว ในรูปแบบของทรีที่เรียกว่า คอมพิวเตชั่นทรี (Computation Tree)

คอมพิวเตชั่นทรี จะแสดงการเรียกโปรซีเยอร์ โดยแต่ละโหนดในทรี สามารถเขียนในรูปของลำดับ <p,x,y> เมื่อ p คือชื่อโปรซีเยอร์, x คืออินพุต และ y คือเอาท์พุต ซึ่งมีความหมายว่า โปรซีเยอร์ p เมื่อประมวลผลด้วยอินพุต x จะได้เอาท์พุตเป็น y; โหนดลูก (Child nodes) ของโหนด <p,x,y> ก็คือโหนดทั้งหมดของการเรียกโปรซีเยอร์โดย p ในการประมวลผลอินพุต x

```
isort([X|Xs],Ys) :- isort(Xs,Zs),insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[Y|Zs]) :- Y>X,insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :- X=<Y.
insert(X,[],[X]).
```

โปรแกรมที่ 1-1 โปรแกรม insertion sort ที่มีข้อผิดพลาด



รูปที่ 1-2 คอมพิวเตชันทรีของการเรียกโปรซีเยอร์ isort ด้วยอินพุต [2,1,3]

รูปที่ 1-2 แสดงตัวอย่างของคอมพิวเตชั่นทรีของการเรียกโปรซีเยอร์ isort ในโปรแกรมที่ 1-1 ด้วยอินพุต [2,1,3] จะเห็นว่าโปรแกรมให้เอาท์พุตที่ไม่ถูกต้อง การทำงานของตัววินิจฉัยก็คือ การทราเวิร์สเข้าไปในทรีเพื่อหาจุดที่ผิดพลาด อาจแสดงตัวอย่างได้ดังนี้ ขั้นแรกตัววินิจฉัยจะหาว่าโหนดลูกของ <isort,[2,1,3],[2,3,1]> โหนดใดไม่ถูกต้องโดยการถามโปรแกรมเมอร์ ซึ่งพบว่า <isort,[1,3],[3,1]> ผิด แสดงว่าข้อผิดพลาดมีอยู่ในการประมวลผลโหนดนี้ ตัววินิจฉัยจะทราเวิร์สเข้าไปต่อแล้วพบว่าโหนด <insert,(1,[3]),[3,1]> ผิด แต่จากการทราเวิร์สเข้าไปแล้วถามโปรแกรมเมอร์ก็พบว่าโหนดลูกคือ <'>',3,1> และ <insert,(1,[]),[1]> ถูกต้อง ซึ่งแสดงว่าโหนด <insert,(1,[3]),[3,1]> ผิดทั้งๆ ที่ โปรซีเยอร์ที่ถูกเรียกให้เอาท์พุตถูกต้อง จึงสรุปได้ว่ามีข้อผิดพลาดอยู่ในโปรแกรมส่วนที่ประมวลผล insert ด้วยอินพุต (1,[3]) ซึ่งในที่นี้ก็คือ คลอส insert(X,[Y|Ys],[Y|Zs]) :- Y>X, insert(X,Ys,Zs).

นอกจากการทราเวิร์สทรีแบบบนลงล่าง (Top-Down) อาจมีการปรับปรุงให้ทราเวิร์สแบบอื่นก็ได้เช่น แบบล่างขึ้นบน (Bottom-Up) ทั้งนี้การเลือกที่จะทราเวิร์สแบบใดมีผลต่อจำนวนคำถาม จะเห็นได้ว่าการทำงานของตัววินิจฉัยนี้เข้าใจได้ง่าย โปรแกรมเมอร์ก็สามารถใช้งานได้ง่ายโดยเพียงตอบคำถามว่าใช่หรือไม่ใช่ (Yes-No) เท่านั้น และเนื่องจากเทคนิค APD นี้เป็นเทคนิคแบบไดนามิกทั้งหมด จึงไม่จำเป็นต้องมีการเขียนสเปกซิฟิเคชั่น โปรแกรมเมอร์เพียงมี อินเทนด์เดด โมเดล ของโปรแกรมก็พอ

1.5 ภาษาโปรล็อก (Prolog)

ระบบที่พัฒนาขึ้นมานี้ ใช้สำหรับดีบักโปรแกรมที่เขียนด้วยภาษาโปรล็อก (Prolog as the target language) และตัวระบบเองก็เขียนขึ้นมาด้วยภาษาโปรล็อก (Prolog as the implementation language)

ทำไมจึงต้องสร้างดีบักเกอร์สำหรับภาษาโปรล็อก ?

ดีบักเกอร์ที่พัฒนานั้นใช้ได้กับส่วนหนึ่งของภาษาโปรล็อกเท่านั้นคือ เพียวโปรล็อก (Pure Prolog) เพราะภาษาเพียวโปรล็อกมีโครงสร้างที่แน่นอน ไม่ซับซ้อนทำให้การพัฒนาดีบักเกอร์ง่ายขึ้น และข้อดีอีกข้อหนึ่งของเพียวโปรล็อก ก็คือไม่มีไซด์-เอฟเฟค (Side-effect) ดังเช่นในภาษา Pascal หรือ C เนื่องจากการจัดการกับไซด์-เอฟเฟค ต้องใช้เทคนิคที่ต่างออกไป

ทำไมต้องพัฒนาดีบักเกอร์ด้วยภาษาโปรล็อก ?

เนื่องจากดีบักเกอร์เป็นเมตาโปรแกรม คือเป็นโปรแกรมที่จัดการกับโปรแกรมอื่น ภาษาที่ใช้สร้างดีบักเกอร์จึงควรสนับสนุนการพัฒนาเมตาโปรแกรม ภาษาโปรล็อกเป็นภาษาที่ไม่แบ่งแยกระหว่างโปรแกรมกับข้อมูลอย่างชัดเจนเหมือนภาษาทั่วๆ ไป ทำให้สามารถจัดการโปรแกรมได้เหมือนข้อมูลเช่น แก้ไข ลบ หรือ เพิ่ม ส่วนของโปรแกรม ระหว่างการทำงานได้ นั่นคือโปรล็อกเหมาะสมกับการสร้างเมตาโปรแกรม อีกทั้งโปรล็อกยังมีลักษณะที่เป็นอินเตอร์พรีทเตอร์ จึงช่วยให้การสร้างดีบักเกอร์ ที่ทำการดีบักโปรแกรมภาษาโปรล็อกเองง่ายขึ้นมาก เพราะสามารถใช้อินเตอร์พรีทเตอร์ของโปรล็อกเอง เพื่อช่วยในการดีบักได้

1.6 วัตถุประสงค์ของปริญญานิพนธ์

1. เข้าใจการทำงานของภาษาโปรล็อก และความผิดพลาดแบบต่างๆ ที่อาจเกิดขึ้นกับโปรแกรมที่เขียนด้วยภาษาโปรล็อก
2. เข้าใจทฤษฎีการดีบัก และแนวทางการพัฒนาดีบักเกอร์ให้มีความชาญฉลาด
3. สามารถพัฒนาดีบักเกอร์ที่ช่วยโปรแกรมเมอร์ในการดีบักโปรแกรม

1.7 ขอบเขตของปริญญานิพนธ์

1. สร้างอินเตอร์พรีทเตอร์สำหรับภาษาเพียวโปรล็อก (Pure Prolog) ที่สามารถเก็บคอมพิวเตชัน ทรี (Computation Tree) ได้

2. สร้างตัววินิจฉัยโปรแกรมที่มีความผิดพลาดแบบ
   - โปรแกรมให้ผลลัพธ์ไม่ถูกต้อง (Incorrect Solutions)
   - โปรแกรมไม่สามารถหาผลลัพธ์ได้ (Finite Failure)
3. สามารถใช้ Specifications และ Properties ของโปรแกรมเพื่อช่วยในการดีบักได้
4. พัฒนา IDE (Integrated Development Environment) เพื่ออำนวยความสะดวกผู้ใช้ในการพัฒนาโปรแกรม และใช้งานดีบักเกอร์

1.8 วิธีการดำเนินงาน

1. ศึกษาทฤษฎีลอจิกโปรแกรมมิ่ง (Logic Programming) และโปรล็อก (Prolog)
2. ศึกษาทฤษฎีการดีบักโดยใช้เทคนิค Algorithmic Program Debugging
3. ออกแบบและสร้างดีบักเกอร์ตามทฤษฎีที่ได้ทำการศึกษา
4. พัฒนาดีบักเกอร์ที่สร้างให้มีความสะดวกในการใช้งาน และเพิ่มความฉลาดในการดีบักให้กับโปรแกรม
5. ทดสอบโปรแกรมตามสภาพแวดล้อมและความผิดพลาดในรูปแบบต่างๆ
6. สรุปการทำงานของโปรแกรมว่าตรงตามเป้าหมายหรือไม่ และศึกษาแนวทางการปรับปรุงดีบักเกอร์

1.9 เนื้อหาโดยรวม (Outline)

ในบทที่ 2 อธิบายถึงทฤษฎีพื้นฐานของลอจิกโปรแกรมมิ่ง (Logic Programming) และโปรล็อก (Prolog) และในบทนี้ยังได้อธิบายถึงการสร้างเมตาอินเตอร์พรีทเตอร์ (Meta-Interpreter) ซึ่งใช้ในการอินเตอร์พรีทโปรแกรมโปรล็อกที่จะทำการดีบัก เพื่อสร้างคอมพิวเตชั่นทรีสำหรับการดีบัก

ทฤษฎีและอัลกอริทึมของตัววินิจฉัยตามเทคนิค APD อธิบายโดยละเอียดในบทที่ 3

บทที่ 4 อธิบายการกำหนด สเปกซิฟิเคชั่น และ พร็อพเพอร์ตี้ (property) ของโปรแกรม ซึ่งดีบักเกอร์สามารถใช้ในการตรวจว่าผลลัพธ์ถูกต้องหรือไม่ โดยไม่จำเป็นต้องถามผู้ใช้ พร้อมทั้งเสนอแนวคิดของการดีบักว่าเป็นการสื่อสารระหว่าง แพรกมาทิส เอเจนต์ (Pragmatist Agent) และ ออราเคิล เอเจนต์ (Oracle Agent)

บทที่ 2
ลอจิกโปรแกรมมิ่งและภาษาโปรล็อก

2.1 บทนำ

ดีบักเกอร์ที่พัฒนาขึ้นใช้สำหรับดีบักโปรแกรมภาษาโปรล็อก และตัวดีบักเกอร์เองก็ยังเขียนด้วยภาษาโปรล็อก ดังนั้นจึงจำเป็นต้องศึกษาทฤษฎีและการทำงานภายในภาษาโปรล็อกโดยละเอียด และเนื่องจากภาษาโปรล็อกถูกสร้างขึ้นบนทฤษฎีลอจิกโปรแกรมมิ่ง ในบทนี้จึงอธิบายพื้นฐานของลอจิกโปรแกรมมิ่งไว้ด้วย เนื้อหาในบทนี้ไม่เพียงพอสำหรับผู้ที่ต้องการศึกษาลอจิกโปรแกรมมิ่งและโปรล็อก ผู้ที่สนใจสามารถหาข้อมูลเพิ่มเติมจากบรรณานุกรมในท้ายเล่ม

นอกจากนี้ ในท้ายบทยังอธิบายการเขียนเมตาโปรแกรมในโปรล็อก พร้อมทั้งการสร้างอินเตอร์พรีทเตอร์สำหรับภาษาโปรล็อก ซึ่งถือว่าเป็นพื้นฐานสำคัญของการสร้างดีบักเกอร์

2.2 ลอจิกโปรแกรม (Logic Programs)

ลอจิกโปรแกรม คือเซตของคลอสจำกัด (definite clause) ซึ่งมีรูปแบบประโยคดังต่อไปนี้

$$A \leftarrow B1, B2, \ldots, Bk \quad k \geq 0$$

เมื่อ A และ B เป็นอะตอมทางลอจิก (logical atom) ซึ่งถูกเรียกว่าเป้าหมายหรือโกล (goal) จากประโยคข้างต้น มีความหมายว่า การที่จะทำให้โกล A สำเร็จได้ เกิดจากการทำโกล B ทุกๆ อัน เรียก A ว่าเป็นหัว (head) ของคลอส และ B ว่าเป็นตัว (body) ของคลอส หากว่ารูปแบบข้างต้นไม่มี B นั่นหมายความว่า A เป็นจริงโดยไม่มีเงื่อนไข

ตัวอย่างของโปรแกรมสำหรับการเรียงลำดับแบบ insertion sort แสดงในโปรแกรม 2-1

```
isort([X|Xs],Ys):-isort(Xs,Zs),insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[X,Y|Ys]):-X=<Y.
insert(X,[Y|Ys],[Y|Zs]):- Y>X,insert(X,Ys,Zs).
insert(X,[],[X]).
```

โปรแกรม 2-1 การเรียงลำดับแบบ insertion sort

2.2.1 การประมวลผล (Computation)

การทำงานของลอจิกโปรแกรม P สามารถอธิบายได้ดังนี้ การทำงานเริ่มที่โกลเริ่มต้น A ซึ่งมีผลลัพธ์ได้ 2 กรณี คือ สำเร็จ (success) และ ล้มเหลว (failure) ถ้าหากการทำงานสำเร็จ แล้วค่าสุดท้ายของตัว

แปรใน A คือเอาท์พุตของการทำงาน โกลที่ได้มาสามารถมีได้หลายค่าจากหลายการประมวลผลที่ให้เอาท์พุตที่แตกต่างกัน

การทำงานจะประมวลผลผ่าน การลดรูปโกลแบบนอนดิเทอร์มินิสติก (nondeterministic goal reduction) ในแต่ละขั้นจะมีโกลปัจจุบันเป็น $A_1,A_2,\ldots,A_n$ แล้วจึงทำการเลือกคลอส $A' \leftarrow B_1,B_2,\ldots,B_k$ ในโปรแกรม P แบบนอนดิเทอร์มินิสติก โดยหัวของคลอส A' ถูกยูนิฟายกับ A_1 ด้วยการแทนที่ θ และโกลที่ผ่านการลดรูปแล้วคือ $(B_1,B_2,\ldots,B_k,A_2,\ldots,A_n)\theta$ การประมวลผลจะสิ้นสุดลงเมื่อโกลปัจจุบันว่างเปล่า

ก่อนที่จะเข้าใจในเนื้อหาของ ลอจิกโปรแกรม ต้องทำความเข้าใจคำจำกัดความบางอย่างที่ปรากฏในทฤษฎีของลอจิกโปรแกรมก่อน

เทอม (term) อาจเป็นค่าคงที่ ตัวแปร หรือ เทอมประกอบ (compound term) ค่าคงที่นั้น รวมไปถึงจำนวนเต็ม (integer) และ อะตอม (atom) อะตอมสามารถเขียนแทนด้วยสตริงซึ่งขึ้นต้นด้วยอักษรพิมพ์เล็ก ส่วนตัวแปรนั้นจะให้ใช้อักขระเริ่มต้นเป็นอักษรตัวใหญ่ ถ้าหากตัวแปรอ้างถึงเพียงสิ่งเดียว มันก็ไม่จำเป็นต้องมีชื่อ และสามารถใช้สัญลักษณ์ "_" (anonymous) ได้

เทอมประกอบ (compound term) ประกอบด้วยฟังก์เตอร์ (functor) และลำดับของเทอมซึ่งเรียกว่าอาร์กิวเมนท์ (argument) ฟังก์เตอร์สามารถแบ่งแยกด้วยชื่อและจำนวนอาร์กิวเมนท์ของมัน อะตอมคือฟังก์เตอร์ที่มีจำนวนอาร์กิวเมนท์เป็นศูนย์

การแทนที่ (substitution) คือเซตจำกัดของคู่ลำดับ มีรูปแบบคือ $X \rightarrow t$ เมื่อ X เป็นตัวแปร และ t เป็นเทอม สำหรับการแทนที่ $\theta=\{ X_1 \rightarrow t_1, X_2 \rightarrow t_2,\ldots, X_n \rightarrow t_n \}$ และเทอม s, $s\theta$ จะแทนผลลัพธ์ของการแทนที่ ซึ่งได้จากการแทนทุกๆ ตัวแปร X_i ใน s ด้วย t_i เมื่อ $1 \leq i \leq n$;เทอม $s\theta$ จะเรียกว่าเป็นอินสแตนซ์ (instance) ของ s

การแทนที่ θ จะเรียกว่า ยูนิฟายเออร์ (unifier) สำหรับเทอม s1 และ s2 ถ้า $s1\theta = s2\theta$ ซึ่งการแทนที่จะเรียกว่าเป็น most general unifier(mgu) ของ s1 และ s2 ถ้าสำหรับทุกๆ ยูนิฟายเออร์ θ_1 ของ s_1 และ s_2 , $s_1\theta_1$ เป็นอินสแตนซ์ของ $s_1\theta$ ถ้าสองเทอมใดๆ มียูนิฟายเออร์แล้ว เทอมเหล่านั้นจะมี mgu เพียงหนึ่งเดียว

ให้คำจำกัดความของการประมวลผลลอจิกโปรแกรมดังนี้

ให้ $N=A_1,A_2,\ldots,A_m$, $m\geq 0$ เป็นโกล และ $C=A\leftarrow B_1,\ldots B_k$, $k\geq 0$ เป็นคลอสซึ่ง A และ A_1 มียูนิฟายเออร์เป็น θ แล้ว $N'=(B_1,\ldots B_k,A_2,\ldots,A_m)\theta$ จะเป็นโกลที่ derive มาจาก N และ C ด้วยการแทนที่ θ และโกล $A_j\theta$ ของ N' ก็กล่าวได้ว่า derive จาก A_j ใน N และกล่าวได้ว่าโกล $B_j\theta$ ของ N' ถูกเรียก (invoke) มาจาก A_1 และ C

ให้ P เป็นลอจิกโปรแกรม และ N เป็นโกล Derivation ของ N จาก P คือลำดับของ $\langle N_i,C_i,\theta_i \rangle$, i=0,1,...เมื่อ N_i เป็นโกลและ Ci เป็นคลอสใน P กับสัญลักษณ์ใหม่ที่ไม่เคยเกิดขึ้นก่อนหน้านี้ในการสืบทอด θ_i เป็นการแทน $N_0= N$ และ N_{i+1} เป็นการสืบทอดมาจาก N_i และ C_i ซึ่งแทนด้วย θ_i ที่ $i \geq 0$

Derivation ของ N จาก P จะเรียกว่า รีฟิวเตชัน (refutation) ของ N จาก P ถ้า N_l เป็น □ (โกลที่ว่างเปล่า) สำหรับบาง $l \geq 0$ derivation นั้นจะจำกัดและ มีความยาวเป็น l ถ้าหากมีรีฟิวเตชันของโกล A จากโปรแกรม P จะกล่าวได้ว่า P สำเร็จ (succeed) บน A

ต่อไปจะเป็นการแสดงการรีฟิวเตชันของโกล isort([2,1],X) จากโปรแกรม 2-1

```
<isort([2,1],L),
      (isort([X|Xs],Ys) <- isort(Xs,Zs),insert(X,Zs,Ys)).
      {X->2,Xs->[1],L->Ys}>
<isort([1],Zs),insert(2,Zs,Ys)),
      (isort([X1|Xs1],Ys1)<-isort(Xs1,Zs1),insert(X1,Zs1,Ys1))
      {X1->1,Xs1->[],Zs->Ys1}>
<isort[],Zs1),insert(1,Zs1,Zs),insert(2,Zs,Ys)),
      isort([],[]),
      {Zs1->[]}>
<(insert(1,[],Zs),insert(2,Zs,Ys)),
insert(X2,[],[X2],
{Zs->[X2],X2->1}>
<insert(2,[1],Ys)),
(insert(X3,[Y3|Y3s],[Y3|Z3s])<-X3>Y3,insert(X3,Y3s,Z3s)),
{X3->2,Y3->1,Y3s->[],Ys->[1]Z3s]}>
<(2>1,insert(2,[],Z3s)),2>1{}>
<insert(2,[],Z3s),insert(X4,[],[X4]),{Z3s->[2],X4->2})>
<□,□,{}>
```

รูปที่ 2-1 ตัวอย่างการทำรีฟิวเตชัน

อีกวิธีหนึ่งของการอธิบายความสำเร็จของการทำงานของลอจิกโปรแกรมคือ การอธิบายผ่านทางรีฟิวเตชันทรี (refutation tree) ในรีฟิวเตชันทรีโหนดจะแสดงโกลที่ถูก instantiate ด้วยค่าสุดท้าย กิ่ง(arc) จะแสดงการ invocation รีฟิวเตชันของ isort([2,1],L) ในรูปที่ 2-1 สามารถเขียนเป็นรีฟิวเตชันทรีได้ดังรูปที่ 2-2 ความลึกของย่อหน้า คือความลึกของทรี

```
isort([2, 1],[ 1,2])
      isort([1],[1])
            isort([],[])
            insert(1,[],[1])
      insert(2,[1],[1,2])
            2>1
            insert(2,[],[2])
```

รูปที่ 2-2 ตัวอย่างของรีฟิวเตชันทรี

2.2.2 ซีแมนติก (Semantics)

นิยามซีแมนติกของลอจิกโปรแกรม ซึ่งเป็นกรณีพิเศษของซีแมนติกมาตรฐานเชิงโมเดล (standard model-theoretic semantics) ของลอจิกลำดับที่หนึ่ง (first order logic) โมเดล (model) คือเซต

ของโกลที่ปราศจากตัวแปร (variable-free goal) การแทนที่ θ จะทำให้ โกล A1,A2,...,An เป็นจริง (satisfy) ใน M ถ้า $A_i\theta$ เป็นสมาชิกของ M สำหรับทุกๆ $1 \leq i \leq n$

นิยามที่ 2.1 : คลอส A ←B1,B2,...,Bn คลอบคลุม(Cover) A' ใน M ถ้ามี θ ซึ่งยูนิฟาย A ด้วย A' และทำให้ B1,B2,...,Bn เป็นจริงเมื่อเทียบกับ M

คลอสใดๆ จะเป็นจริงใน M ถ้าทุกๆ โกลที่ปราศจากตัวแปรซึ่งคลอสนั่นครอบคลุมใน M อยู่ใน M และคลอสจะเป็นเท็จหากไม่เป็นดังที่กล่าว โปรแกรม P จะเป็นจริงใน M ถ้าทุกๆ คลอสใน P เป็นจริงใน M ในที่นี้จะถือว่าคำว่าถูกต้อง (correct) มีความหมายเหมือนกับคำว่าจริง (true) และคำว่าไม่ถูกต้อง (incorrect) มีความหมายเหมือนกับคำว่าเท็จ (false)

2.3 ภาษาโปรล็อก (Prolog)

ภาษาโปรล็อกเป็นภาษาทางลอจิกโปรแกรม โดยโปรล็อกขยายความสามารถของเพียวโปรล็อกออกมาอีกเพื่อประสิทธิภาพและความลึกซึ้ง ต่อไปในหัวข้อนี้ จะอธิบายเกี่ยวกับว่า ทำอย่างไรจึงจะอิมพลีเมนท์ (implement) นอนดีเทอร์มินิซึม (nondeterminism) ของลอจิกโปรแกรมใน ซีเควนเชียล โปรล็อก และศึกษาส่วนขยายบางส่วนของภาษาโปรล็อกที่ถูกใช้ในโปรแกรมและการอธิบายระบบ

2.3.1 กลไกการเอ็กซิคิวและการแบ็คแทรกค์ (The execution and backtracking mechanism)

การเอ็กซิคิวของโปรล็อกเป็นการจำลองของการทำงานแบบนอนดีเทอร์มินิสติกซึ่งอธิบายในหัวข้อที่ 2.2 แทนที่จะทำการเลือกคลอสถัดไปแบบนอนดีเทอร์มินิสติก โปรล็อกจะพิจารณาคลอสที่สามารถยูนิฟายได้ตามลำดับที่พบในโปรแกรม เมื่อไม่สามารถหาได้ มันจะทำการแบ็คแทรกซ์กลับไปที่จุดที่เป็นทางเลือกล่าสุด

การทำงานแบบ Sequential ของโปรล็อกที่กล่าวไปถูกต้อง (Correct) แต่ไม่ครบถ้วน (Incomplete) เนื่องจากอาจเกิดการประมวลผลที่ไม่สิ้นสุด (Infinite Computation) ทำให้ไม่สามารถหารีฟิวเตชันได้

ลอจิกโปรแกรมกับการพิสูจน์โปรซีเยอร์แบบลำดับที่กล่าวไปก็คือ ภาษาเพียวโปรล็อก (Pure Prolog) ส่วนขยายที่จะอธิบายต่อไปไม่อยู่ในเพียวโปรล็อก

2.3.2 การควบคุม (Control)

ลำดับของโกลในคลอสจะเป็นตัวกำหนดลำดับของการประมวลผล ลำดับของคลอสในโปรแกรมจะเป็นตัวกำหนดลำดับในการเลือกคลอส โปรล็อกมีสิ่งอำนวยความสะดวกสำหรับการควบคุมลำดับได้แก่สัญลักษณ์คัท (cut) " ! "

คัทของภาษาโปรแกรมภาษาโปรล็อกเปรียบเสมือนคำสั่ง "go to" ซึ่งเป็นการควบคุมระดับต่ำ การใช้คัทอาจทำให้โปรแกรมไม่มีโครงสร้าง และเข้าใจยาก คล้ายกับ " go to" ที่สามารถใช้อิมพลีเมนท์การควบคุมระดับสูงเช่น "while" และ "repeat" ลูป ในการแปลงโปรแกรมภาษา คัทสามารถใช้นิยามการควบคุมระดับสูงของเพรดดิเคทภาษาโปรล็อกได้

โครงสร้างหนึ่งที่เป็นการนิเสธ (negation) ได้แก่ "not" โปรซีเยอร์ not(A) จะเป็นโกลที่สำเร็จ ก็ต่อเมื่อ A ล้มเหลว โดย "not" จะสามารถนิยามได้จากคัทดังต่อไปนี้

```
not(P) <- P,!,fail.
not(P).
```

เมื่อ "fail" ที่เป็นโกลที่ล้มเหลวเสมอ โปรแกรม not จะใช้ตัวแปรเมตา (meta-variable) ซึ่งเป็นคุณสมบัติของโปรล็อกที่อนุญาตให้โกลถูกพิจารณาอย่างไดนามิก (dynamic) ในขณะประมวลผล

เครื่องหมายต่อไปของโปรล็อกคือ การใช้สัญลักษณ์ " ; " แทนคำสั่ง or พิจารณาโปรแกรมต่อไปนี้

```
grandfather(X,Y)<-father(X,Z),(father(Z,Y);mother(Z,Y)).
```

จะมีความหมายของการทำงานเหมือนกับโปรแกรมต่อไปนี้

```
grandfather(X,Y)<-father(X,Z),father(Z,Y).
grandfather(X,Y)<-father(X,Z),mother(Z,Y).
```

แต่โปรแกรมแรกจะมีประสิทธิภาพมากกว่า เพราะมันจะไม่ทำการคิด father(X,Z) เพื่อที่จะทำ mother (Z,Y) ซึ่งคำสั่ง or สามารถแสดงการแปลงได้เพราะถ้าโปรแกรมไม่มีคัท มันสามารถลบไปเป็นเพรดดิเคทใหม่ได้เช่น

```
grandfather(X,Y) <-father(X,Y),parent(Z,Y).

parent(X,Y)<-father(X,Y).
parent(X,Y)<-mother(X,Y).
```

การควบคุมอื่นๆที่มีประโยชน์ในภาษาโปรล็อกได้แก่ if-then-else คือ p->q;r คือหาก p สำเร็จให้ไปทำ q มิเช่นนั้นจะทำ r

2.3.3 ไซด์-เอฟเฟค (Side-effects)

โปรล็อกเป็นภาษาฟังก์ชันโดยธรรมชาติของมัน และมีหลายแอพพลิเคชั่น ซึ่งต้องการโครงสร้างข้อมูลเบื้องต้นเช่น สแตกและคิว สามารถอิมพลีเมนท์อย่างมีประสิทธิภาพโดยไม่มีไซด์-เอฟเฟค แต่อย่างไรก็ตาม บางครั้งมันก็จำเป็นในการพัฒนาสเตท (state) ของระบบ โปรล็อกมีสองเพรดดิเคท คือ assert และ retract ซึ่งทำงานกับไซด์-เอฟเฟค กับฐานข้อมูล เช่น เมื่อโปรแกรมถูกเอ็กซิคิว มีการเรียก assert(X) คือเพิ่มคลอส X ไปที่โปรแกรม (การ assert มีสองแบบคือ asserta คือการเพิ่มเข้าไปเป็นคลอสแรกของฐานข้อมูล และ assertz คือเพิ่มเข้าไปเป็นคลอสสุดท้ายของฐานข้อมูล) ส่วน retract(X) เป็นการลบคลอส X ออกจากฐานข้อมูล

ไซด์-เอฟเฟค ถูกใช้โดยการพัฒนาระบบเพื่อพัฒนาโปรแกรมในการตีความหมาย หรือการดีบัก มันสามารถนำมาใช้ในการบันทึกคำตอบของการไควรี่ของผู้ใช้ได้ทำให้ไม่ต้องถามคำถามเดิมซ้ำสองครั้ง

2.3.4 เพรดดิเคทลำดับที่สอง (Second order predicates)

เพรดดิเคทลำดับที่สองสามารถมองโกลเป็นข้อมูล ตัวอย่างของเพรดดิเคทประเภทนี้ได้แก่ bagof โกล bagof(X,P,S) ให้ค่า S เป็นลิสของอินสแตนซ์ของ X สำหรับแต่ละการประมวลผลของโกล P ตัวอย่าง เช่น

```
?-bagof((X-Y),(member(X,[1,2,3]),member(Y,[a,b,c,d])),S).

S = [1-a,1-b,1-c,1-d,2-a,2-b,2-c,2-d,3-a,3-b,3-c,3-d]
```

2.3.5 เมตาโปรแกรมมิ่ง (Meta-programming)

ดีบักเกอร์จัดว่าเป็นเมตาโปรแกรม(Meta-program) คือเป็นโปรแกรมที่สามารถจัดการโปรแกรมอื่น เช่น การทำการเคราะห์ ควบคุมการทำงานของโปรแกรม หรือแก้ไขและเพิ่มเติมส่วนต่างๆของโปรแกรมได้ พูดอีกนัยหนึ่งก็คือ เมตาโปรแกรมมองโปรแกรมเป็นข้อมูลได้

เมตาโปรแกรมจะประกอบด้วย 2 ระดับ ระดับแรกคือ ระดับเมตา (meta level) ซึ่งทำการควบคุมอยู่เหนือโปรแกรมอีกระดับหนึ่งที่เรียกว่า ระดับวัตถุ (object level) ในการดีบักโปรแกรม ส่วนของระดับวัตถุคือส่วนของโค้ดที่ต้องการทำการดีบัก ส่วนของระดับเมตาก็คือ ตัววินิจฉัยข้อผิดพลาด หรือ อินเตอร์พรีทเตอร์เป็นต้น

เมตาอินเตอร์พรีทเตอร์ (Meta-interpreter)

ในการที่จะทำการดีบักโปรแกรมได้ จะต้องเข้าใจว่าโปรแกรมที่ให้ผลลัพธ์ผิดพลาดมีการทำงานอย่างไร จึงต้องรู้ในแต่ละขั้นตอนของการทำงาน จึงจำเป็นต้องมีตัวอินเตอร์พรีทเตอร์เพื่อทำการอินเตอร์พรีทและเก็บข้อมูลระหว่างการอินเตอร์พรีท ตัวอย่างของเมตาอินเตอร์พรีทเตอร์อย่างง่ายๆ สำหรับเพียว โปรล็อก ที่เรียกว่า Vanilla meta-interpreter แสดงดังรูป 2-3

```
solve(true).
solve(A&B) :-
     solve(A),solve(B).
solve(A) :-
     hyp(A<-B),solve(B).
```

รูปที่ 2-3 Vanilla Meta-Interpreter

```
hyp(p<-q&c).
hyp(q<-a&b).
hyp(a<-true).
hyp(b<-true).
hyp(c<-true).
```

รูปที่ 2-4 โค้ดของโปรแกรมที่จะใช้ในการอินเตอร์พรีท

```
solve(true,[[]]).
solve(A&B,Ls) :-
      solve(A,L1s),solve(B,L2s),
      append(L1s,L2s,Ls).
solve(A,[tree(A,[])]) :-
      builtin(A),!,A.
solve(A,[tree(A,Ls)]) :-
      hyp(A<-B),solve(B,Ls).
```

รูปที่ 2-5 เมตาอินเตอร์พรีทเตอร์ที่สามารถเก็บคอมพิวเตชั่นทรีได้

เมื่อมีการไควรี่, อินเตอร์พรีทเตอร์จะทำการอินเตอร์พรีทกับโค้ด ของโปรแกรมที่เก็บอยู่ใน เพรดดิเคท hyp (ดูรูปที่2-4) จะเห็นว่า เมตาอินเตอร์พรีทเตอร์ตัวนี้ ไม่มีการเก็บข้อมูลของการอินเตอร์พรีท

รูปที่ 2-5 แสดงโค้ดของเมตาอินเตอร์พรีทเตอร์ที่สามารถเก็บ คอมพิวเตชั่นทรี ซึ่งแสดงการพิสูจน์โปรแกรมได้ ข้อแตกต่างอีกอย่างหนึ่งของอินเตอร์พรีทเตอร์นี้คือ มีการแยกการจัดการ Built-in predicate โดยส่งไปให้โปรล็อก อินเตอร์พรีทเตอร์ (Prolog Interpreter) เพราะเพรดดิเคทเหล่านี้ไม่ได้ถูกนิยามไว้ใน hyp ตัวอย่างของการใช้งานของอินเตอร์พรีทเตอร์ตัวนี้เป็นดังนี้

```
| ?- solve(p,Proof).
Proof = [tree(p,[tree(q,[tree(a,[]),tree(b,[])]),tree(c,[])])]
```

บทที่ 3
การพัฒนาตัววินิจฉัยโดยใช้
Algorithmic Program Debugging

3.1 บทนำ

การพัฒนาตัววินิจฉัย (Diagnoser) ในที่นี้มีการประยุกต์ใช้เทคนิค Algorithmic Program Debugging (APD) ซึ่งคิดโดยศาสตราจารย์ Ehud Y. Shapiro APD เป็นเทคนิคการดีบักแบบตอบโต้ (Interactive) ดีบักเกอร์จะเป็นผู้ช่วยของผู้ใช้ในการหาข้อผิดพลาดในโปรแกรม โดยการวิเคราะห์จากโปรแกรมแล้วถามผู้ใช้ เพื่อนำข้อมูลมาสรุปได้ว่าข้อผิดพลาดของโปรแกรมอยู่ที่ใด เทคนิคนี้จะมองโปรแกรมเป็นเซตของโปรซีเยอร์ มีทฤษฎีว่าการที่โปรแกรมผิดพลาดเกิดจากการที่มีโปรซีเยอร์ที่ประพฤติตัวผิดจากที่ผู้ใช้คิด ตัววินิจฉัยจะช่วยผู้ใช้หาโปรซีเยอร์เหล่านั้น ในกรณีของโปรแกรมภาษาโปรล็อก โปรซีเยอร์ก็คือคลอส (clause) นั่นเอง

3.2 สมมติฐานเกี่ยวกับภาษาโปรแกรม

กลไกการทำงานของภาษาจะต้องมีลักษณะเป็นการเรียกโปรซีเยอร์ แต่จะไม่สนการทำงานภายในโปรซีเยอร์นั้นๆ อัลกอลิทึมจะสนใจเพียงว่าโปรซีเยอร์ใดเรียกใช้โปรซีเยอร์ตัวใด มีอินพุตและเอาท์พุตเป็นอะไร ภาษาที่ถูกต้องตามโมเดล (model) ดังกล่าว ได้แก่ ภาษาเพียวลิสป์ (pure Lisp) ภาษาเพียวโปรล็อก (pure Prolog)

จะอธิบายสมมติฐานดังกล่าวอย่างมีหลักการได้คือ โปรแกรมเป็นเซตของโปรซีเยอร์ ซึ่งโปรซีเยอร์ถูกอธิบายด้วย ชื่อ จำนวนสมาชิก และ โค้ด(code) ของโปรซีเยอร์เหล่านั้น โดยในภาษาเหล่านั้น จะไม่สนใจว่าโค้ดของมันเป็นอย่างไร แต่จะสมมติว่าภาษานั้นมีอินเตอร์พรีทเตอร์ซึ่งสามารถแปลงโค้ดไปเป็นพฤติกรรม (behaviors) สำหรับโปรซีเยอร์นั้น โดยอธิบายได้จากนิยามของ top level trace ดังนี้

top level trace ของโปรซีเยอร์ **p** ที่มีอินพุตเป็น x และให้ผลลัพธ์เป็น y แล้ว สามารถแสดงเป็นลำดับได้ว่า

{<p1,x1,y1>, <p2,x2,y2>, ...,<pk,xk,yk>}

ซึ่ง **p** เป็นชื่อของโปรซีเยอร์ ส่วน x และ y เป็นเวกเตอร์บนโดเมน **D** การอินเตอร์พรีทดังกล่าวตามนิยามของ top level trace จะเป็นดังนี้คือ ถ้าเซตดังกล่าวที่ได้เป็นเซตว่าง โปรซีเยอร์ **p** ที่มีอินพุตเป็น x จะให้ค่าผลลัพธ์เป็น y โดยไม่ต้องมีการเรียกโปรซีเยอร์ แต่ถ้าหากว่าเซตที่ได้มิใช่เซตว่าง นั่นคือ โปรซีเยอร์ **p** ที่มีอินพุตเป็น **x** จะมาสามารถหาค่าได้โดยการเรียกโปรซีเยอร์ **p1** โดยใช้อินพุต x1 และจะให้ผลลัพธ์

เป็น y1 และเมื่อโปรซีเยอร์ **p** เรียกโปรซีเยอร์ **p2** ที่ใช้อินพุต x2 จะให้ผลลัพธ์เป็น y2 เป็นเช่นนี้ไปเรื่อยๆ เมื่อเสร็จสิ้นการเรียกโปรซีเยอร์สุดท้าย โปรซีเยอร์ **p** จะให้ผลลัพธ์เป็น y

ข้อบังคับของ top level trace ที่มีความเป็นลำดับคือ จะต้องทำการอินเตอร์พรีทตามลำดับดังกล่าว แต่ข้อบังคับนี้สามารถยืดหยุ่นได้หากใช้จัดการกับภาษาโปรแกรมที่มีการทำงานพร้อมๆ กัน (Concurrent Programming Language) ภาษาโปรแกรมสามารถเป็นนอนดิเทอร์มินิสติก (nondeterministic) คือในกรณีของแต่ละโปรซีเยอร์ หนึ่งๆ ก็สามารถมีได้หลาย top level trace ด้วย

จะอธิบายการทำงานของโปรแกรมผ่าน คอมพิวเตชันทรี (computation tree) โดยให้ **P** เป็นโปรแกรม พาเชียล คอมพิวเตชันทรี (partial computation tree) ของโปรแกรม **P** เป็นทรีที่มีลำดับและมีรูท (root) ในแต่ละโหนดแสดงได้เป็น <p,x,y> โดยที่ p เป็นชื่อของโปรซีเยอร์ x และ y เป็นเวกเตอร์บนโดเมน **D** และทุกๆ โหนด <p,x,y> ใน T ที่มีโหนดลูกเป็น S จะได้ว่า S เป็น top level trace ของ <p,x,y> ที่มี p เป็นโปรซีเยอร์ในโปรแกรม **P** จะได้ว่า T จะเป็น คอมพรีท คอมพิวเตชันทรี (complete computation tree) ของโปรแกรม **P** ถ้ามันเป็นพาเชียล คอมพิวเตชัน ทรี ของ **P** ที่โหนดล่างสุดไม่มีการเรียกโปรซีเยอร์อื่นอีก คอมพรีท คอมพิวเตชัน ทรี แสดงให้เห็นการทำงานที่สำเร็จของโปรแกรม และมีลักษณะเช่นเดียวกับ รีฟิวเตชัน ทรี (refutation tree) ในลอจิกโปรแกรม

จะได้ว่า T เป็น รีชเอเบิล คอมพิวเตชัน ทรี (reachable computation tree) ของโปรแกรม **P** ถ้ามันเป็นพาเชียล คอมพิวเตชัน ทรี ของโปรแกรม **P** ที่ทุกๆ พาท (path) ใน T ที่มีทุกๆ ซับทรีที่โหนดทุกโหนดทางซ้ายยกเว้นโหนดสุดท้ายเป็นคอมพรีท คอมพิวเตชัน ทรี ซึ่งรีชเอเบิล ทรี แสดงถึงการประมวลผลที่ยังไม่เสร็จของอินเตอร์พรีทเตอร์

จะได้ว่า T เป็น อินฟีนิท รีชเอเบิล คอมพิวเตชัน ทรี (infinite computation tree) ของโปรแกรม **P** ถ้ามันเป็น รีชเอเบิล คอมพิวเตชัน ทรี ของโปรแกรม **P** ที่มีพาทไม่สิ้นสุด ซึ่งอินฟีนิท รีชเอเบิล ทรี แสดงถึงการประมวลผลที่ไม่สิ้นสุดของอินเตอร์พรีทเตอร์

จะพบว่าหากโปรแกรม **P** ไม่มี อินฟีนิท รีชเอเบิล คอมพิวเตชัน ทรีที่มีรูทเป็น <p,x,y> แล้วจะมีรีชเอเบิล คอมพิวเตชัน ทรี ที่มีจำนวนจำกัด ทำให้โปรแกรมมีการทำงานที่สามารถสิ้นสุดได้

จะถือว่าภาษาโปรแกรมที่มีอินเตอร์พรีทเตอร์ดังเช่นการทำงานดังกล่าวได้ต้องมีลักษณะดังนี้ คือ เมื่อมีการเรียกโปรซีเยอร์ ตัวอินเตอร์พรีทเตอร์จะดำเนินการประมวลผล รวมถึงเรียกตัวเองจากการร้องขอของโปรซีเยอร์อื่นๆ และส่งค่าผลลัพธ์กลับออกมา ถือว่าคอมพิวเตชันทรีของโปรแกรมจะแทนพฤติกรรมที่เป็นไปได้ที่แท้จริงของตัวอินเตอร์พรีทเตอร์ เช่น ให้โปรแกรม **P** และมีการเรียกโปรซีเยอร์ <p,x> ถือว่าอินเตอร์พรีทเตอร์จะตอบสนองดังนี้คือ จะไดเวิร์จ (diverge) เมื่อโปรแกรม **P** มี อินฟีนิท รีชเอเบิล คอมพิวเตชัน ทรี ที่มีรูทเป็น <p,x,y> ในบางเอาท์พุต y แต่หากไม่มีอินฟีนิต รีชเอเบิล คอมพิวเตชัน ทรี มันอาจจะมีคำตอบหลายคำตอบที่ให้ค่า y ออกมาสำหรับโปรแกรม **P** ที่มีคอมพรีท คอมพิวเตชัน ทรี ที่ <p,x,y> เป็นรูท หากมีค่า y จริงมันจะให้ค่า y ออกมาตัวหนึ่ง มิเช่นนั้น มันจะไม่ให้ผลลัพธ์ออกมา

ให้คำจำกัดความของความหมายของโปรแกรมที่ใช้เหมือนโมเดลของทฤษฎีทางลอจิกโปรแกรม ให้ **M** เป็นเซตของการเรียกโปรซีเยอร์ <**p**,x,y> โดย **p** คือชื่อของโปรซีเยอร์ที่มีตัวแปรอินพุต n ค่า และ มีตัวแปรเอาท์พุต m ค่า x อยู่ในโดเมน $\mathbf{D}^n$ และ y อยู่ในโดเมน $\mathbf{D}^m$ ถือว่าสำหรับทุก **p** และ x ที่มีจำนวน y จำกัดซึ่ง <**p**,x,y> อยู่ใน **M** จะสามารถกล่าวได้ว่า y เป็นเอาท์พุตที่ถูกต้องในการเรียกโปรซีเยอร์ <**p**,x> ใน **M** ถ้า <**p**,x,y> อยู่ใน **M** ด้วยหรือในกรณีที่ y ไม่ให้ผลลัพธ์ออกมา ถ้าไม่มี y' ที่ <**p**,x,y'> อยู่ใน **M** นั่นคือ จริงๆแล้ว คำตอบที่ได้จากการเรียกโปรซีเยอร์ **p** ไม่มีคำตอบ นอกนั้น จะถือว่า y เป็นเอาท์พุตที่ไม่ถูกต้องใน **M**

โปรแกรม **P** จะพาเชียลลี่ คอเรค (partially correct) ใน **M** ถ้ารูทของทุกๆ คอมพรีท คอมพิวเตชัน ทรี ของโปรแกรม **P** อยู่ใน **M** โปรแกรม **P** จะ คอมพรีท ใน **M** ถ้าทุกๆโปรซีเยอร์ใน **M** เป็นรูทของ คอมพรีท คอมพิวเตชัน ทรีของโปรแกรม **P**

โปรแกรม **P** จะเทอมิเนทเสมอ (everywhere terminating) ถ้าตัวมันไม่มีอินฟีนิต รีชเอเบิล คอมพิวเตชัน ทรี ไม่เช่นนั้น จะถือว่ามัน ไดเวิร์จ และโปรแกรมจะโทโทลลี่ คอเรค (totally correct) ถ้ามันมี พาเชียลลี่ คอเรค คอมพรีท และ เทอมิเนทเสมอ

ใช้คำสำหรับแสดงความซับซ้อนของการทำงานของอินเตอร์พรีทเตอร์ได้ 2 แบบด้วยกัน คือ ความยาว (length) และ ความลึก (depth) โดยนิยามของความยาว คือ จำนวนของการเรียกโปรซีเยอร์ระหว่างการทำงาน และความลึกคือ ความลึกสูงสุดของการเรียกโปรซีเยอร์ ความยาวของการทำงานคือจำนวนของโหนดในคอมพิวเตชัน ทรี และ ความลึกของการทำงานคือ ความลึกของคอมพิวเตชัน ทรี บางครั้ง จะพิจารณาการแตกกิ่งของการทำงาน (branching) โดยดูจากการแตกกิ่งสูงสุดของคอมพิวเตชัน ทรี

อัลกอริทึมการวินิจฉัยเป็นการจำลองการทำงานของโปรแกรมที่จะทำการดีบัก ฉะนั้นจึงสามารถสร้างตัวจำลองการอินเตอร์พรีทของภาษานั้นได้ และทำการขยายความสามารถ โดยเพิ่มคำสั่งบางอย่างเข้าไปเพื่อทำให้ทำการเก็บค่าบางอย่างของการเรียกโปรซีเยอร์ในระหว่างการทำงานได้

สามารถสรุปคุณสมบัติของโปรแกรมดังที่ได้กล่าวมาได้ดังนี้

- โปรแกรมเป็นเซตของโปรซีเยอร์
- โปรซีเยอร์ประกอบด้วย ชื่อของโปรซีเยอร์, อินพุต, เอาท์พุต และ โค้ด
- โค้ดของโปรซีเยอร์ พิจารณาจากเซตของพฤติกรรมที่เป็นไปได้ทั้งหมดของโปรซีเยอร์ ซึ่งจะทำการอธิบายผ่าน top level trace
- การทำงานของโปรแกรม จะอธิบายผ่าน คอมพิวเตชัน ทรี ที่สร้างจาก top level trace ของโปรซีเยอร์

3.3 การวินิจฉัยโปรแกรมที่ให้ผลลัพธ์ไม่ถูกต้อง (Diagnosing Incorrect Solutions)

3.3.1 ความถูกต้อง (Correctness)

ถ้าโปรแกรมพาเชียลลี่ คอเรค แล้วทุกๆโปรแกรมย่อยของมันจะพาเชียลลี่ คอเรคด้วย โดยที่คอมพิวเตชัน ทรี ของ โปรแกรมย่อยจะเป็นสับเซต (subset) ของคอมพิวเตชันทรีของโปรแกรม แต่ในทางกลับ

กัน มันก็ไม่จริงเสมอไป คือหากโปรแกรมย่อยของมัน เป็นพาเชียลลี่ คอเรคแล้ว เมื่อนำมารวมเป็นโปรแกรมใหญ่ก็ไม่จำเป็นต้องเป็นพาเชียลลี่ คอเรคด้วย ดังตัวอย่างเช่น โปรแกรมย่อย **P'** ของโปรแกรม **P** ที่โปรแกรม **P** เป็น พาเชียลลี่ คอเรค สามารถแก้โปรแกรมย่อย **P'** ของโปรแกรม **P** ให้ **P'** เป็น พาเชียลลี่ คอเรค แต่ โปรแกรม **P** ไม่เป็น พาเชียลลี่ คอเรคดังตัวอย่างโปรแกรมการคูณดังต่อไปนี้

```
times(0,Y,0).
times(X,0,Z) <- plus(X,0,Z).
times(s(X),Y,Z)<- times(X,Y,U),plus(U,Y,Z).

plus(0,Y,Y).
plus(X,0,Z) <- plus(X,0,Z ).
plus(s(X),Y,s(Z)) <- plus(X,Y,Z).
```

โปรแกรมนี้เป็น พาเชียลลี่ คอเรค เมื่อเปรียบเทียบกับการอินเตอร์พรีตมาตรฐานของการคูณ และการบวก เมื่อการคำนวนบนโปรซีเยอร์ times(X,Y,Z) สิ้นสุด(โดยที่ Y ไม่เท่ากับ 0) ผลของค่า Z คือ ค่าจากการคูณ X ด้วย Y อย่างไรก็ตาม สามารถปรับปรุงให้โปรซีเยอร์ plus(X,Y,Z) ให้ผลลัพธ์ใน Z เป็นค่าจาก X ถ้า Y มีค่าเป็น 0 การปรับปรุงนี้ยังคงรักษาความเป็น พาเชียลลี่ คอเรค ของโปรแกรม บวกไว้ แต่ละเมิดการเป็นพาเชียลลี่ คอเรค ของโปรแกรมทั้งหมด นั่นคือ เมื่อทำการเปลี่ยนแปลงแล้ว times(1,0,Z) จะให้ค่า Z เป็น 1 แทนที่ก่อนหน้านี้ ไม่ให้ผลลัพธ์

จากตัวอย่างดังกล่าวที่แสดงให้เห็นว่า คุณสมบัติของความเป็นพาเชียล คอเรค ไม่สามารถใช้กับโปรซีเยอร์ได้ดังนั้น จึงจำเป็นต้องหานิยามสำหรับการหาความผิดพลาด คำจำกัดความที่จะเสนอต่อไปนี้เป็นแสดงให้เห็นว่า โปรซีเยอร์ **p** จะกล่าวว่าถูกต้องจากการอินเตอร์พรีทตามโมเดล **M** ถ้า ทุกการเรียกโปรซีเยอร์ โดย **p** ให้ผลลัพธ์ที่ถูกต้องตามโมเดล **M** แล้ว **p** จะให้ผลลัพธ์ที่ถูกต้องใน **M**

นิยามที่ 3.1 : ให้โปรซีเยอร์ **p** ซึ่งแทนการเรียกใช้โปรซีเยอร์ **p** ด้วย <p,x,y> เมื่อเทียบกับโมเดล **M** ถ้าหาก <p,x,y> มี top level trace เป็นสับเซตในโมเดล **M** แล้ว จะกล่าวได้ว่า

โปรซีเยอร์ **p** จะถูกต้องในโมเดล **M** ถ้า ทุกๆการเรียกใช้โปรซีเยอร์ <p,x,y> ซึ่งแทนได้ด้วย **p** เมื่อเทียบกับโมเดล **M** อยู่ในโมเดล **M** มิเช่นนั้น จะถือว่า **p** ไม่ถูกต้องในโมเดล **M**

ถ้าโปรซีเยอร์ **p** ไม่ถูกต้องใน **M** แล้ว มันจะมี top level trace ใน **M** สำหรับบางการเรียกโปรซีเยอร์<p,x,y> ที่ไม่ได้อยู่ใน **M** ซึ่ง top level trace ถูกเรียกว่า ตัวอย่างแย้ง (counterexample) ความถูกต้องของ **p** ใน โมเดล **M**

ต่อไปจะแสดงให้เห็นว่า ถ้า ทุกๆโปรซีเยอร์ในโปรแกรมถูกต้องแล้ว โปรแกรมจะเป็น พาเชียลลี่ คอเรค แต่ในทางตรงข้ามไม่จริงเสมอไป นั่นคือ หากโปรแกรมเป็นพาเชียลลี่ คอเรคแล้ว ไม่ได้หมายความว่าโปรซีเยอร์ในโปรแกรมนั้นถูกต้อง นั่นคือทุกๆโปรแกรมที่ไม่ให้ผลลัพธ์ เป็นพาเชียลลี่ คอเรค แต่โปรแกรมดังกล่าวอาจไม่ถูกต้องก็ได้ ดังตัวอย่างต่อไปนี้

```
f(X,Y)<- integer(2*Y),f(2*X,Y).
```

จะเห็นได้ว่า โปรแกรมดังกล่าวไม่ถูกต้องในการอินเตอร์พรีตตามโมเดล **M** ={ <f,X,1> | X is an integer}, เพราะว่า <f,0.5,1> ไม่ได้อยู่ในโมเดล **M** แต่ top level trace { <f,1,1> } อยู่ในโมเดล **M** ทฤษฎีบทที่ 3.1 แสดงให้เห็นว่า หากโปรแกรมมีการทำงานที่สิ้นสุด และให้ผลลัพธ์ที่ไม่ถูกต้องแล้วแสดงว่าโปรแกรมนั้นประกอบด้วยโปรซีเยอร์ที่ไม่ถูกต้อง

ทฤษฎีบทที่ 3.1 : ให้ **P** เป็นโปรแกรม และ **M** เป็นโมเดลของการอินเตอร์พรีต ถ้าโปรแกรม **P** นั้นไม่เป็นพาเชียลลี่ คอเรค เมื่อเทียบกับโมเดล **M** แล้วโปรแกรม **P** จะประกอบด้วย โปรซีเยอร์ที่ไม่ถูกต้องในโมเดล **M**

พิสูจน์ทฤษฎีบทที่ 3.1 : ให้ **p** เป็นโปรซีเยอร์ในโปรแกรม **P** ที่มีอินพุตเป็น x และให้ผลลัพธ์ที่ผิดของ y ออกมา จะพิจารณา คอมพิวเตชัน ทรี ของ <p,x,y> โดยพิจารณาโหนดแรกที่ <q,u,v> โดยทราเวิร์สแบบโพส-ออเดอร์ (post-order traversal) ของทรีแล้วพบว่าไม่ได้อยู่ในโมเดล **M** และเนื่องจากโหนดดังกล่าวไม่ได้อยู่ในโมเดล **M** แต่โหนดลูกทั้งหมดของโหนดนั้นอยู่ในโมเดล **M** ดังนั้นเมื่อพิจารณา **q** จะเห็นได้ว่าการเรียกโปรซีเยอร์ <q,u,v> ที่มีโหนดลูกทั้งหมดอยู่ใน **M** นั้น <q,u,v> ไม่ได้อยู่ในโมเดล **M** นั่นแสดงว่า **q** ไม่ถูกต้องในโมเดล **M**

3.3.2 อัลกอริทึมแบบ Single-Stepping ที่ใช้วินิจฉัยโปรซีเยอร์ที่ผิดพลาด

การพิสูจน์ในทฤษฎีบทที่ 3.1 เสนอให้เห็นถึงอัลกอริทึมสำหรับตรวจหาโปรซีเยอร์ที่ผิดพลาดในโปรแกรม **P** ที่ไม่เป็นพาเชียลลี่ คอเรค การใช้เทคนิคซิงเกิล-สเตประหว่างการทำงานในขณะเรียกโปรซีเยอร์ โปรซีเยอร์แรกที่ให้ผลลัพธ์ที่ผิดพลาดคือโปรซีเยอร์ที่ผิด

จะทำให้อัลกอริทึมดังกล่าวเป็นแบบแผน และเพื่อแสดงว่าอัลกอริทึมดังกล่าวถูกต้อง อัลกอริทึมจะใช้ กราวด์ ออราเคิล (ground oracle) สำหรับ **M** ซึ่งเป็นเครื่องมือดังกล่าว โดยที่อินพุต <p,x,y> จะให้ผลลัพธ์เป็น yes หาก <p,x,y> อยู่ใน **M** และเป็น no หากมิได้อยู่ใน **M**

อัลกอริทึม 1 : การสืบหาความผิดพลาดของโปรซีเยอร์ โดย ซิงเกิล-สเตปปิ้ง
อินพุต : โปรซีเยอร์ **p** ใน **P** และมีอินพุต x ซึ่ง **p** ที่มีอินพุต x ให้ผลลัพธ์ออกมาเป็น y และไม่ถูกต้องใน **M**
เอาท์พุต : โปรซีเยอร์ <q,u,v> ไม่ได้อยู่ใน **M** ซึ่งแทน <q,u,v> ด้วย **q**
อัลกอริทึม : จำลองการเอ็กซิคิว (execution) ของโปรซีเยอร์ **p** ที่มีอินพุต x ให้ผลลัพธ์เป็น y เมื่อการเรียกโปรซีเยอร์ <q,v> ให้ผลลัพธ์เป็น v ใช้กราวด์ ออราเคิลในการตรวจสอบ นั่นคือ <q,u,v> จะอยู่ใน **M** หาก <q,u,v> สิ้นสุดการทำงาน และไม่ได้ให้ผลลัพธ์ออกมา

อัลกอริทึม 1 ถูกต้องเพราะว่าลำดับของการเรียกโปรซีเยอร์ที่ให้ผลในการทำงานเป็นแบบการ

ทราเวิร์สแบบโพส-ออเดอร์ ในทรี พิจารณาโหนดแรก <q,u,v> ของทรีในการโพส-ออเดอร์ ซึ่งกราวด์ ออราเคิล ให้ผลลัพธ์ว่า no โดยนิยามของอัลกอริทึม 1 ลูกทั้งหมดของโหนดนี้ได้ถูกตรวจสอบแล้วว่าถูกต้อง ดังนั้นจะพบว่า **q** เป็นโปรซีเยอร์ที่ไม่ถูกต้องใน **M**

3.3.3 การพัฒนาด้วยภาษาโปรล็อก

ความสัมพันธ์ในการบรรยายของ ตอนนี้เกี่ยวข้องกับลอจิก โปรแกรม ซึ่ง top level trace ของโปรซีเยอร์ จะมีลักษณะเช่นเดียวกับบอดี้ (body) ของกราวด์ อินสแตนซ์ (ground instance) ของคลอส (clause) โดยที่ความถูกต้องของโปรแกรม **P** ใน **M** เป็นโมเดลทางทฤษฎี (model-theoretic) ของความถูกต้องใน **M** คลอสจะถือว่าไม่ถูกต้องใน **M** ก็ต่อเมื่อ มันแทนเป้าหมาย (goal) ที่ไม่อยู่ใน **M** และตัวอย่างแย้งของความถูกต้องของโปรแกรม **P** เป็นตัวอย่างของความผิดพลาดของคลอสในโปรแกรม **P** ในกรณีของลอจิก โปรแกรม อัลกอริทึม 1 ได้สรุปการหาความผิดพลาดของคลอสในโปรแกรม **P** โดยตรวจหาความผิดพลาดที่เกิดที่โปรแกรม **P** โปรแกรมต่อไปนี้จะเป็นการอิมพลีเมนท์อัลกอริทึม 1

```
fp((A,B),X) <- !,
      fp(A,Xa),
      ( Xa=ok ->fp(B,X); X=Xa ).
fp(A,X) <-
      system(A) -> A, X=ok ;
      clause(A,B),fp(B,Xb),
      ( Xb \= ok -> X=Xb;
        query(forall,A,true) -> X=ok ; X=(A->B)).
```

โปรแกรม fp ข้างต้นได้ตรวจหาคลอสที่ผิดพลาดด้วยภาษาโปรล็อก เนื่องจากความผิดพลาดของเป้าหมาย ซึ่งขยายความสามารถมาจากโปรแกรม solve ที่จะแสดงต่อไปนี้

```
solve(true).
solve((A,B))<- solve(A), solve(B).
solve(A) <- clause(A,B), solve(B).
```

fp(A,X) คำนวนความสัมพันธ์ของที่ว่า A สามารถหาคำตอบได้ตามเป้าหมาย ถ้า A เกิดความผิดพลาดแล้ว X จะเป็นตัวอย่างที่ผิดพลาดของคลอสที่ใช้ในการหาผลลัพธ์ของ A มิเช่นนั้น X จะมีค่าเป็น ok

โปรซีเยอร์ fp ประกอบด้วย 2 คลอส โดย คลอสแรกเกี่ยวข้องกับเป้าหมายที่มีตัวเชื่อมและ (conjunctive goal)โดยมันจะให้ผลลัพธ์เป็น(A'<-B')ถ้าการเรียกตัวเองบนทุกๆตัวเชื่อมและ ให้ผลลัพธ์กลับมาเป็น (A'<-B') ละให้ผลลัพธ์เป็น ok หากไม่มีการให้ผลลัพธ์ที่ผิดพลาดกลับมา คลอสที่สองเกี่ยวข้องกับโกลหน่วย (unit goal) คือถ้า A เป็นเพรดิเคท (predicate) ของระบบที่ทำการเอ็กซิคิว และให้ผลลัพธ์กลับมาเป็น ok ถ้า A สำเร็จ และให้ค่าเป็น (A'<-B')เมื่อเกิดความผิดพลาด ถ้าการเรียงตัวองของ fp บนตัวของคลอสที่ร้องขอมันจะให้ผลลัพธ์ที่เป็นความผิดพลาดกลับมา นั่นคือ (A'<-B')ไม่เช่นนั้น แสดงว่าไม่มีความผิดพลาด มันจะให้ผลลัพธ์เป็น ok ออกมา

โกลที่ผ่านการหาคำตอบแล้วจะยังคงประกอบด้วยตัวแปร ซึ่งถูกตีความว่า universally quantified การอิมพลีเมนท์ของไควรี่ (query) สามารถเป็นได้ทั้งการไควรี่จากผู้ใช้ หรือ universal query ก็ได้ ในการอิมพลีเมนท์นี้ query(forall,A,V) จะให้ผลลัพธ์เป็น V=true หากทุกๆตัวอย่างของ A เป็นจริง แต่ถ้าหากมีความผิดพลาด V จะมีค่าเป็น false

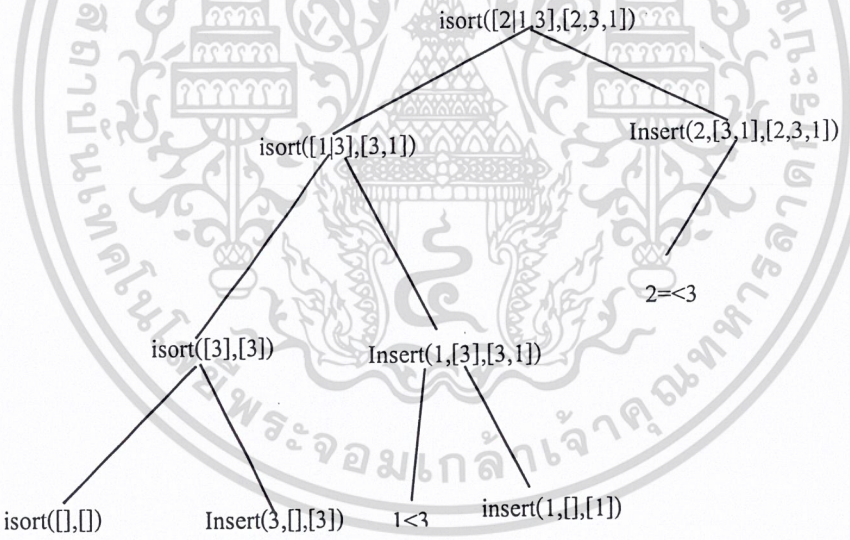
มาพิจารณาโปรแกรมการเรียกลำดับแบบ insertion sort และดูพฤติกรรมของ fp

```
isort([X|Xs],Ys):-isort(Xs,Zs),insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[Y|Zs]):- Y>X,insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]):-X=<Y.
insert(X,[],[X]).
```

เมื่อผู้ใช้ทำการไควรี่(query) โปรแกรม isort นี้ด้วยคำสั่ง isort ด้วยอินพุต [2,1,3] ผลที่ได้จะเป็นดังนี้

```
| ?- isort([2,1,3],X).
X = [2,3,1]
```

ซึ่งจะเห็นว่าผลลัพธ์ที่ได้ผิดพลาด สามารถแสดง คอมพิวเตชัน ทรีได้ดังรูปที่ 3-1



รูปที่ 3-1 คอมพิวเตชันทรีของโปรแกรม insertion sort ที่ผิดพลาด

เมื่อ นำ fp มาใช้กับ isort[2,1,3],[2,3,1]) โปรแกรม fp จะทำการถามผู้ใช้ดังต่อไปนี้

```
| ?- fp(isort([2,1,3],[2,3,1]),C).
query: isort([],[])? y.
query: insert(3,[],[3])? y.
query: isort([3],[3])? y.
query: insert(1,[],[1])? y.
query: insert(1,[3],[3,1])? n.
C = insert(1,[3],[3,1]) <- 3 >1, insert(1,[],[1]))
```

fp ให้คลอสที่ทำให้ผลลัพธ์ผิดพลาดออกมา ในคลอสแรกของ insert จากการพิจารณาจะเห็นได้ว่า ถ้าเปลี่ยน เครื่องหมาย > เป็น < แล้วทำการเรียกโปรแกรม isort อีกครั้ง จะได้ว่า

```
| ?- isort([2,1,3],X).
X = [1,2,3]
```

ซึ่งเป็นคำตอบที่ถูกต้อง

3.3.4 ขอบเขตล่างของจำนวนการถามคำถาม

แทนค่าของอัลกอริทึมในสองมิติด้วยกันคือ ความซับซ้อนของการทำงาน ซึ่งเป็นผลจากทรัพยากรของเครื่องที่ถูกใช้ และความซับซ้อนของการไควรี่ ซึ่งเป็นผลจากจำนวน และ ชนิดของการไควรี่กับออราเคิลขณะที่มันทำงาน โดยทั่วไป ผู้ที่ทำหน้าที่ออราเคิลก็คือผู้ใช้ เพราะฉะนั้น จะเน้นไปในการลดความซับซ้อนของการไควรี่หาความผิดพลาด

ความยาวและความลึกของการทำงานของการตรวจหาความผิดพลาดวัดได้จากฟังก์ชันของความซับซ้อนของการทำงานในการหาความผิดพลาด จะไม่สนใจค่าของการไควรี่ เนื่องจาก แยกการวิเคราะห์ของการไควรี่ ออกจากส่วนการหาความผิดพลาด

เมื่อใช้การวัดดังกล่าว กรณีที่แย่สุด (worst-case) ของความยาว และ ความลึกของอัลกอริทึม การหาความผิดพลาดแบบ ชิงเกิล-สเตป เมื่อการหาความผิดพลาดของการทำงานของโปรซีเยอร์ p ที่มีอินพุตเป็น x ซึ่งให้ผลลัพธ์เป็น y เป็นเชิงเส้นในความยาวและความลึกโดยลำดับของการผิดพลาดของการทำงานจำนวนสูงสุดของการไควรี่ ออราเคิลแสดงเป็นขอบเขตโดยความยาวของการทำงาน

การตอบคำถามหลายๆ ไควรี่จะเกิดขึ้นมากถ้าหากการทำงานมีความยาว กรณีที่แย่ที่สุดของจำนวนการไควรี่ในการหาความผิดพลาดเป็นลำดับของฟังก์ชันลอกาลิทึมของการเรียกโปรซีเยอร์ที่ใช้ในการทำงาน ในส่วนนี้ จะทำการพัฒนาอัลกอริทึมนี้ เพื่อให้มีประสิทธิภาพ

การพิสูจน์ขอบเขตล่างเป็นการอ้างอิงถึงทฤษฎี information-theoretic adversary argument โดยมันเป็นแนวความคิดพื้นฐานที่ว่า การไควรี่ทั้งหมดสามารถบอก ได้ถึงการมีอยู่ของความผิดพลาดของของส่วนประกอบของคอมพิวเตชันทรี

ข้อผิดพลาดในทางกลับกัน adversary bug สามารถซ่อนอยู่ในส่วนประกอบของทรี วิธีที่ดีที่สุดที่จะป้องกันความผิดพลาดคือการทำการไควรี่โดยแบ่งทรีออกเป็นสองส่วนเท่าๆกัน ผลของแต่ละการไควรี่จะทำให้มีการตรวจหาความผิดพลาดในพื้นที่ๆแคบลงและสามารถตรวจหาความผิดพลาดได้มากที่สุดเป็น $\log_2 n$

สมมติว่าอัลกอริทึมของการหาความผิดพลาสำหรับโปรซีเยอร์ที่ไม่ถูกต้องที่ใช้ กราวด์ออราเคิลสำหรับ M เมื่อนำมาใช้กับการทำงานของโปรซีเยอร์ p ที่มีอินพุต x และให้ผลลัพธ์ที่ไม่ถูกต้องเป็น y ใน M และจะให้ <q,u,v> ที่ไม่อยู่ใน M ออกมา

ทฤษฎีบทที่ 3.2: ให้ DA เป็นอัลกอริทึมการหาความผิดพลาดสำหรับโปรซีเยอร์ที่ไม่ถูกต้อง จะมีโปรแกรม **P** ซึ่งมีบาง n ที่มีอินเตอร์พรีทเตชัน **M** และ <p,x,y> ซึ่งมีความยาวของ คอมพิวเตชัน ทรี น้อยกว่า หรือเท่ากับ n สำหรับ DA สำหรับการเรียกโปรซีเยอร์ <p,x,y> จะไควรี่น้อยกว่า $\log_2 n$

พิสูจน์ทฤษฎีบทที่ 3.2 : แสดงรายละเอียดของโปรแกรม ซึ่งเป็นวิธีขัดแย้งสำหรับการหาความผิดพลาด ซึ่งสามารถทำการอิมพลีเมนท์ได้ พิจารณาโปรแกรมข้างล่างนี้

p(0).

p(s(X)) <- **p**(X).

โปรแกรมจะตรวจสอบว่าอินพุตของมันเป็นสตริงที่ประกอบด้วย 0 และมีแอพพริเคชันที่สืบทอดมาจากฟังก์ชันของs หรือไม่

กำหนดการอินเตอร์พรีท **M** ซึ่งมีสตริงที่มีความยาวน้อยกว่า k ที่ k มากกว่าหรือเท่ากับศูนย์ สำหรับอัลกอริทึมการหาความผิดพลาดเพื่อหาตัวอย่างแย้งของโปรแกรม มันจำเป็นต้องหาค่า k ที่แน่นอน สำหรับ $p(s^k(0))$ อยู่ใน M แต่ $p(s^k+1(0))$ไม่อยู่

เพราะว่าคำตอบที่เป็นบวกของการไควรี่ p(s^k+1(0))บังคับให้ ต้องทำการเลือก k>i และคำตอบที่เป็นลบจะบังคับให้ เลือกการไควรี่ที่ $k \leq i$ โดยมันจะเป็นไปตามทุกๆอินพุต X ที่มีขนาด n และทุกการไควรี่ DA สามารถซ่อน k ได้ ดังนั้น DA จะจำเป็นต้องทำการ ไควรี่น้อยกว่า $\log_2 n$ เพื่อทำการค้นหา

3.3.5 Divide-and-query : อัลกอริทึมที่ลดการถามคำถาม

ขอบเขตล่าง พิสูจน์ให้เห็นถึงความคิดในการปรับปรุง วิธีการไควรี่แบบ ซิงเกิล-สเตป เมื่อไควรี่โหนด <q,u,v> ในคอมพิวเตชัน ทรี ซึ่งจะเป็นการแบ่งทรี ออกเป็นส่วนประกอบสองส่วนอย่างหยาบๆ ถ้า <q,u,v> อยู่ใน **M** แล้ว เว้นแต่ว่า รากของซับทรีอยู่ที่โหนดนั้น และมีการเรียกซ้ำ ไม่เช่นนั้น จะใช้อัลกอริทึมดังกล่าวเรียกจะเป็นการเรียกซ้ำในสับทรี

พัฒนาอัลกอริทึมดังกล่าวเพื่อสนับสนุนเทคนิคการไควรี่ดังกล่าว ซึ่ง ความยาวและความลึกของการไควรี่จะแปรผันเป็นเส้นตรงกับความยาวและความลึกของทรี ต่อไป จะเป็นวิธีการแบ่งทรีตามที่ได้กล่าวมา

ให้ **M'** เป็นสับเซตของ **M** และเมื่อพิจารณาคอมพิวเตชัน ทรี ของ **p** ที่มีอินพุต x และให้ผลลัพธ์กลับมาเป็น y น้ำหนัก (weight) ของการเรียกโปรซีเยอร์ <p,x,y> มีเกณฑ์ที่ใช้ในการคำนวนจาก M'เป็นดังต่อไปนี้

ถ้า <p,x,y> อยู่ใน **M'** น้ำหนักของมันจะมีค่าเป็น 0 แต่หากไม่อยู่ใน **M'** แล้ว ถ้า <p,x,y> เป็นใบ (leaf) น้ำหนักจะมีค่าเป็น 1 หากนอกเหนือจากนี้ น้ำหนักของ <p,x,y> จะเป็นผลบวกของน้ำหนักของโหนดลูกของมันบวกด้วย1

ให้ T เป็นคอมพิวเตชัน ทรีซึ่งมีค่าน้ำหนัก **M'** เป็น w กำหนดให้โหนดกลางของทรีเป็นเป็นโหนดซ้ายที่หนักที่สุด (leftmost heaviest node) ในทรี ที่มีน้ำหนักของเกณฑ์ที่ใช้ในการคำนวน $M' \leq [w/2]$ ให้คอมพิวเตชัน ทรีซึ่งมีน้ำหนัก w ทีมีเกณฑ์ที่ใช้ในการคำนวน **M'** สามารถคำนวนหาโหนดกลาง

ได้ในการแปรผันเชิงเส้นตรงระหว่างความยาว และ w โดยเรียกโปรซีเยอร์ซ้ำโดยใช้โปรซีเยอร์ fpm กับรูทของทรี

โปรซีเยอร์ fpm จะทำการคำนวนโหนดกลางของทรี และ น้ำหนักของมัน มันจะทำการคำนวนน้ำหนักและตัวบ่งชี้ (identify) ของโหนดตลอดเวลาที่มันเป็นครึ่งล่าง (lower half) ของทรี ตัวอย่างเช่นเมื่อทำการทราเวิร์สโหนดซึ่งน้ำหนักน้อยกว่าครึ่งหนึ่งของน้ำหนักของทรี แล้วให้ค่าที่มีโหนดน้ำหนักมากสุดและส่งผลกลับโดยโหนดลูกและน้ำหนักของมันตราบเท่าที่มันยังให้โหนดครึ่งบนของทรีตัวอย่างเช่น ทราเวิร์สโหนดซึ่งน้ำหนักมากกว่าครึ่งของน้ำหนักของทรี การทำงานดังกล่าวเป็นดังนี้ ที่อินพุต <**p**,x,t> และ W เป็นการค้นหาของมันในคอมพิวเตชัน ทรี แบบโพสออเดอร์ เพื่อตัดบางโหนดใน **M'** สำหรับโหนด **A** ที่ถูกค้นหาจากการทำงานของ Wa น้ำหนักที่ใช้เป็นเกณฑ์ในการคำนวน **M'** ของโหนด และ น้ำหนักและตัวบ่งชี้ของโหนดที่หนักที่สุดเป็น(**B**,Wb) ส่งค่ากลับโดยการเรียกซ้ำตัวเองจากเพรดิเคท fpm ที่โหนดลูกของ **A'** ถ้า Wa>[W/2] แล้วให้ผลคืนกลับมาเป็น (**B**,Wb) ไม่เช่นนั้นมันจะให้ผลลัพธ์เป็น (**A**,Wa) โปรซีเยอร์ fpm อธิบายสมมติฐานให้กับคอมพิวเตชัน ทรี ในการกระทำสิ่งใดคอมพิวเตชัน ทรีจะไม่ถูกให้แต่จะถูกทำงานโดยอินเตอร์พรีทเตอร์ และ fpm เป็นอาร์กิวเมนเตชัน ของการอินเตอร์พรีทนั้น ดังเช่นการนำมาใช้งานในภาษาโปรล็อกดังที่จะได้กล่าวต่อไป

การนำอัลกอริทึมของการดีไวด์-แอนด์-ไควรี่มาใช้ในโปรซีเยอร์นี้แสดงได้ดังอัลกอริทึม 2

ทฤษฎีที่บทที่ 3.3: ให้ **P** เป็นโปรแกรมและ **M** แทนการอินเตอร์พรีท ถ้าโปรซีเยอร์ **p** ใน **P** ได้มีการทำงานบนอินพุต x ของความยาว **n** มีความลึก **d** มีจำนวนกิ่งเป็น **b** และให้ผลลัพธ์ออกมาที่ $y \neq \perp$ ไม่ถูกต้องใน **M** แล้วการทำงานของอัลกอริทึม 2 ประยุกต์ใช้กับการเรียกโปรซีเยอร์ <**p**,x,y> และ **M'**={} มีความยาวเท่ากับ cn สำหรับค่าคงที่ที่ c>0 ความลึกเป็น d+1 มีจำนวนการไควรี่เป็นblog n และให้โปรซีเยอร์ <**q**,u,v> ที่ไม่ได้อยู่ใน **M** ซึ่ง **q** แทนการเรียกโปรซีเยอร์ <**q**,u,v> ใน **M**

อัลกอริทึม 2: อัลกอริทึมการหาโปรซีเยอร์ที่ผิดพลาดโดยวิธี ดีไวด์-แอนด์-ไควรี่

อินพุต: โปรซีเยอร์ **p** ใน **P** และมีอินพุต x ซึ่ง ให้ผลลัพธ์ออกมาเป็น y :ซึ่งไม่ถูกต้องใน **M** และ $\mathbf{M' \subseteq M}$

เอาท์พุต: โปรซีเยอร์ <**q**,u,v> ที่ไม่ได้อยู่ใน **M** ซึ่ง **q** แทนการเรียกโปรซีเยอร์ <**q**,u,v> ใน **M**

อัลกอริทึม: ทำการจำลองการเอ็กซิคิวต์ของโปรซีเยอร์ **p** บนอินพุต x ให้ผลลัพธ์เป็น y การคำนวน w น้ำหนักของเกณฑ์ที่ใช้ในการคำนวน **M'** ของคอมพิวเตชัน ทรี แล้วการเรียกรีเคอซีฟของ fp ด้วยอินพุต <**p**,x,y> ค่าน้ำหนัก w และเกณฑ์ที่ใช้ **M'** การทำงานเป็นดังนี้ ถ้า w=1 แล้ว fp จะให้ <**p**,x,y>หากเป็นกรณีอื่น ต้องใช้ fpm โดยกำหนดว่าทำการหาโหนดที่มีน้ำหนักที่สุด <**q**,u,v> ในทรีของการเรียกโปรซีเยอร์ <**p**,x,y> ซึ่งมีน้ำหนักเป็น wq มีเกณฑ์การตัดสิน **M'** ที่น้อยกว่าหรือเท่ากับ [w/2] มันจะไควรี่ กราวด์ ออราเคิลถ้า <**q**,u,v> อยู่ใน **M**

หากคำตอบของการไควรี่เป็น yes แล้ว fp จะเรียกตัวเองแบบรีเคอซีฟด้วย <**p**,x,y> w-wq และ $\mathbf{M' \cup \{<q,u,v>\}}$ ถ้าหากออราเคิลตอบ no fp จะเรียกตัวเองด้วย <**q**,u,v> น้ำหนักเป็น wq และ **M'**

เลมม่าที่ 3.1: ภายในเงื่อนไขของทฤษฎีบทที่ 3.3 ถ้าอัลกอริทึมสิ้นสุดและให้ผลลัพธ์กลับเป็น <**p**,x,y> แล้ว **p** จะคลอบคลุม <**p**,x,y> ใน **M** แต่ <**p**,x,y> ไม่ได้อยู่ใน **M**

พิสูจน์ เลมม่าที่ 3.1 :สังเกตว่าถ้า ip ถูกเรียกด้วย <**p**,x,y> แล้ว <**p**,x,y> ไม่ได้อยู่ใน **M** นี่เป็นเงื่อนไขของอินพุตของอัลกอริทึมและถูกสงวนไว้โดยการเรียก fp ซึ่งโปรซีเยอร์ fp จะให้ <**p**,x,y> กลับมาเท่านั้น ถ้าหากมันถูกเรียกด้วย w=1 ซึ่ง แสดงความหมายว่าโหนดลูกของ <**p**,x,y> ในคอมพิวเตชัน ทรี มีค่าเป็นศูนย์ หรือในอีกแง่นึงก็คือ โหนดลูกทั้งหมดของมันอยู่ใน **M**

เลมม่าที่ 3.2: ในเงื่อนไขของทฤษฎีบทที่ 3.4 การทำงานของอัลกอริทึมที่ 1ที่นำมาใช้กับ <p,x,y> และ **M'**={} มีความยาว O(n) ความลึก d+1 และทำการไควรี่ blog n ครั้ง

พิสูจน์ เลมม่าที่ 3.2: จะแสดงว่าแต่ละการร้องขอโปรซีเยอร์ fp ขนาดของคอมพิวเตชัน ทรี ลดลงเป็นสัดส่วนกับ 1/2b ถ้าออราเคิลตอบ no ในการไควรี่ที่ทำโดย fp แล้ว fp จะทำการเรียกตัวเองแบบรีเคอซีฟ ด้วยสับทรีที่มีรูทเป็นโหนดที่ถูกไควรี่ ขนาดของสับทรีนี้มีขนาดมากสุด 1/2 ของขนาดของทรีเดิม ถ้าหากออราเคิลตอบ yes แล้วโหนดนั้นจะถูกเพิ่มเข้าไปใน **M'** ในการทำงานครั้งต่อไป น้ำหนักของโหนดนี้จะเป็นศูนย์ ซึ่งลดลงจากน้ำหนักของคอมพิวเตชัน ทรีในสัดส่วนของ 1/2b ให้ I(n) เป็นจำนวนของการเรียกซ้ำตัวเองของอัลกอริทึมบนทรีที่มีน้ำหนัก n ความสัมพันธ์ในขอบเขตของ I(n) เป็นดังนี้

$I(n) \leq 1$

$I(n) \leq 1+I(n(2b-1)/2b)$

สามารถยืนยันโดยการอุปนัยว่า I(n)=blog2 n ซึ่งทดแทนอสมการนี้ได้ โดยอสมการแรก เพราะว่า blog2 1เท่ากับ 0 ซึ่งน้อยกว่าหรือเท่ากับ 1 ขั้นต่อไปของการอุปนัยคือ พิสูจน์ว่าอสมการ

$\text{blog2 } n \leq 1+\log_2(n(2b-1)/2b)$

สามารถลดเหลืออสมการ $\log_2(2b-1)/2b \geq -1$ สำหรับ $b \geq 1$

ความยาวของแต่ละการเรียกตัวเองเป็นเส้นตรงของขนาดที่ยังคงอยู่ของคอมพิวเตชัน ทรี ดังนั้นความยาวรวมของการทำงานจะสามารถแทนได้ดังนี้

$L(1) \leq k$

$L(n) \leq kn + L(n(2b-1)/2b)$

สำหรับ k>0 ซึ่งถูกแทนด้วย $L(n) \leq cn$ สำหรับ $c \geq 2b$

พิสูจน์ทฤษฎีบทที่ 3.4: ทฤษฎีบทที่ 3.4 ว่าตามเลมม่าที่3.1 และ 3.2 โดยเลมม่าที่3.1คือถ้าอัลกอริทึมสิ้นสุดแล้วมันจะให้ <**q**,u,v> ที่ไม่อยู่ใน **M** ซึ่ง **q** คลอบคลุม <**q**,u,v> ใน **M** โดยเลมม่าที่ 3.1เมื่ออัลกอริทึมสิ้นสุด และความยาวและความลึกของมันจะถูกต้องการ

อยากจะแสดงให้เห็นผลที่ตามมาของทฤษฎี ถ้าความยาวของการทำงานของโปรแกรมที่ถูกตรวจหาความผิดพลาดเป็นพหุนาม (polynomial) ในขนาดของอินพุตของมัน และจำนวนของการไควรี่ที่ใช้วิธี

การตรวจหาความผิดพลาดโดยอัลกอริทึม ดีไวด์-แอนด์-ไควรี่อยู่ในรูปของลอการิทึมของอินพุตของมันที่เกิดความผิดพลาดในการทำงานที่ค่าคงที่แปรตามกำลังของพหุนาม ความผิดพลาดซึ่งยากที่จะตรวจพบจะแสดงตัวเองเฉพาะมีอินพุตขนาดใหญ่ การไควรี่ที่ซับซ้อนของอัลกอริทึมดีไวด์-แอนด์-ไควรี่ แสดงให้เห็นถึงจำนวนของการไควรี่ที่จะมีในการหาความผิดพลาดซึ่งสามารถทำได้สำหรับโปรแกรมที่รันในเวลาที่เป็นพหุนาม

3.3.6 การพัฒนาอัลกอริทึม Divide-and-Query ด้วยภาษาโปรล็อก

ต่อไปนี้จะเป็นตัวอย่างการอิมพลีเมนท์ โปรซีเยอร์ fpm

```
fpm(((A,B),Wab),M,W)<-!
      fpm((A,Wa),(Ma,Wma),W),fpm((B,Wb),(Mb,Wmb),W),
      Wab is Wa+Wb,
      (Wma>=Wmb->M=(Ma,Wma);M=(Mb,Wmb)).
fpm((A,0),(true,0),W)<-
      system(A),!,A;fact(A,true).
fpm((A,Wa),M,W)<-
      clause(A,B),fpm((B,Wb),Mb,W),
      Wa is Wa+1,
      (Wa>(W+1)/2-> M=Mb;M=((A<-B),Wa)).
```

คลอสแรกเป็นการคำนวนหาโหนดที่มีค่าน้ำหนักสูงสุดซึ่งได้กลับออกมาจากการเรียกรีเคอซีฟจากโหนดลูกและน้ำหนักรวมของโหนด คลอสที่สองเป็นการตัดเป้าหมายซึ่งอยู่ใน M' หรือเป็นเพรดิเคทระบบ คลอสที่สามเป็นการหาคำตอบของเป้าหมายเดี่ยว และ จะเลือกเอาว่าอยู่ในครึ่งล่างหรือครึ่งบนของคอมพิวเตชัน ทรี และส่งผลลัพธ์ออกมาตามนั้น

โปรแกรมต่อไปที่จะแสดง เป็นการนำเอาอัลกอริทึมที่ 2 มาอิมพลีเมนท์โดยตรงด้วยภาษาโปรล็อก

```
false_solution(A) <-
      writelv(['Error: wrong solution ',A,. diagnosing…']), nl,
      fpm((A,W),_,0),% finds W,the length of the computatiob
      fp(A,W,X)-> handle_error('false clause',X);
      write('!Illegal call to fp'), nl.

fp(A,Wa,X)<-
      fpm((A,Wa),((P<-Q),Wm),Wa),
      (Wa=1 -> X=(P<-Q);
      query(forall,P,true) -> Wa1 is Wa-Wm, fp(A,Wa1,X) ;
      fp(P,Wm,X) ).
```

โปรซีเยอร์ fp ถูกอธิบายไว้ด้านบนแล้ว ในการอิมพลีเมนท์ สมมติให้ M' เป็นตัวแทนของกลุ่มของคลอส สำหรับทุกๆเป้าหมายที่อยู่ใน M ที่มีคลอส fact(A,true) อยู่ในฐานข้อมูลและสมมติว่าผลลัพธ์ของระบบถูกต้อง

ข้อแตกต่างระหว่างอัลกอริทึมที่ 2 กับ การอิมพลีเมนท์คือ fp จะให้คลอสตัวอย่างที่ผิดออกมาแทนที่จะเป็นเป้าหมายที่ผิดที่แทนด้วยคลอส และหาผลลัพธ์แจ้งให้ผู้ดีบักทราบ
พิจารณาโปรแกรมสำหรับดีบักของโปรแกรมเรียงลำดับตัวเลขแบบ insertion sort ดังต่อไปนี้

```
isort([X|Xs],Ys)<-isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[X,Y|Ys])<-X≤Y.
insert(X,[Y|Ys],[Y|Zs])<-insert(X,Ys,Zs).
insert(X,[],[X]).
```

หาก นำซิงเกิล-สเต็ป มาใช้ในการหาความผิดพลาดจากการไควรี่isort([2,1,4,3,5,6],[6,4,5,2,3,1])มันจะทำการไควรี่ทั้งหมด 16 ครั้ง ในคอมพิวเตชัน ทรี ที่มีความยาว 17 แต่หากใช้วิธีของ ดีไวด์-แอนด์-ไควรี่ จะมีการไควรี่เพียง4ครั้ง โดยแสดงการไควรี่ได้ดังต่อไปนี้

```
?-fp(isort([4,1,2,3,5,6],[1,2,3,5,4,6]),17,C).
Query: isort([2,3,5,6],[2,3,5,6])? y.
Query: insert(4,[2,3,5,6],[2,3,5,4,6])? n.
Query: insert(4,[2,5,6],[5,4,6])? n.
Query: insert(4,[6],[4,6])?y.
C=insert(4,[5,6],[5,4,6])<-insert(4,[6],[4,6])
```

ทฤษฎีบทที่ 3.3 แสดงให้เห็นว่าความยาวและความลึกของการทำงานของการหาความผิดพลาดเป็นสมการเชิงเส้นของการทำงานที่ผิดพลาด มีความสงสัยอยู่ว่าการอ้างคล้ายๆอย่างนี้ไม่สามารถใช้ได้ในเวลาของการทำงานของโปรแกรม เพราะว่าวิธีการในการแบ่งไม่ได้คำนึงถึงจำนวนของการแบ็กแทรค(backtrack) ที่จำเป็นในโครงสร้างของโปรแกรมในแต่ละส่วนที่ต่างๆกันในคอมพิวเตชัน ทรีแต่ ขนาดสุดท้ายมีเพียงขนาดเดียว อย่างไรก็ตามขอบเขตของจำนวนการไควรี่มีค่าเป็น n log n คือ จำนวนของขอบเขตของการเรียกซ้ำมีค่าคาม log ของความยาว และ จำนวนครั้งของการทำงานในการเรียกซ้ำมีทั้งหมด n ครั้ง0

3.4 การวินิจฉัยโปรแกรมที่ไม่สามารถหาผลลัพธ์ (Diagnosing Finite Failure)

สำหรับโปรแกรมที่สามารถทำงานจนสิ้นสุดได้ ถ้าหากให้ y เป็นผลลัพธ์ที่ถูกต้องสำหรับโปรซีเยอร์ **p** ที่มีอินพุตเป็น x แต่การทำงานของโปรซีเยอร์ **p** ด้วยอินพุต x สิ้นสุดลงแล้วให้ผลลัพธ์ที่แตกต่างจาก y แสดงว่าผลลัพธ์ไม่ถูกต้อง ซึ่งวิธีการในการตรวจหากความผิดพลาดได้กล่าวไปในหัวข้อที่แล้ว สำหรับโปรแกรมที่ไม่สิ้นสุดการทำงาน อาจเกิดจากการที่ทุกๆ การทำงานของโปรซีเยอร์ **p** ในอินพุต x ทำงานสิ้นสุด และให้ผลลัพธ์ออกมาถูกต้องใน **M** แต่ไม่ให้ผลลัพธ์ของการทำงาน y ออกมาซึ่งกล่าวได้ว่าโปรแกรมทำงานผิดพลาดแบบ finite failure บน <p,x,y> ซึ่งความผิดพลาดแบบนี้จำเป็นต้องใช้การปฏิบัติเป็นพิเศษสำหรับโปรแกรมที่ทำงานไม่สิ้นสุด หัวข้อนี้ จะกล่าวถึงการพัฒนาวิธีการสำหรับการหาความผิดพลาดแบบ finite failure

3.4.1 ความสมบูรณ์ (Completeness)

โปรแกรม **P** กล่าวได้ว่ามีความสมบูรณ์ใน **M** ถ้าสำหรับทุกๆ<**p**,x,y> ใน **M** มีการทำงานของโปรซีเยอร์ **p** บนตัวแปร x ที่ให้ผลลัพธ์เป็น y ถ้าหากโปรแกรมมีความผิดพลาดแบบไฟไนท์เฟล ในการเรียกโปรซีเยอร์ใน **M** แล้ว มันจะถือว่าโปรแกรมไม่สมบูรณ์ใน **M** ด้วย

กำหนดให้ **p** เป็นโปรซีเยอร์ที่สมบูรณ์ที่ขึ้นกับ **M** ถ้าสำหรับทุกๆ <**p**,x,y> ให้ **p** แทน <**p**,x,y> ซึ่งขึ้นกับ **M** หากเป็นกรณีอื่น จะกล่าวได้ว่า มันไม่สมบูรณ์ ถ้าโปรซีเยอร์ **p** ไม่สมบูรณ์ คือไม่ครอบคลุม <**p**,x,y> และแล้ว **p** จำเป็นต้องปรับปรุงซึ่งทางเดียวที่จะทำให้โปรซีเยอร์ **p** ที่มีอินพุต x ให้ผลลัพธ์กลับมาเป็น y คือต้องมีโปรซีเยอร์บางโปรซีเยอร์ที่อยู่ใน <**p**,x> ให้ผลลัพธ์ที่ไม่ถูกต้องใน **M**

ทฤษฎีบทที่ 3.4 : ให้ **P** เป็นโปรแกรม และ **M** เป็นการอินเตอร์พรีท ถ้า **P** ไฟไน์เฟล ในการเรียกโปรซีเยอร์ <**p**,x,y> ใน **M** แล้วโปรแกรม **P** จะประกอบด้วยโปรซีเยอร์ที่ไม่สมบูรณ์ใน **M**

พิสูจน์ ทฤษฎีบทที่ 3.4: แสดงว่าถ้ามี <**p**,x,y> ใน **M** สำหรับทุกๆ รีซเอเบิล คอมพิวเตชัน ทรี ของโปรแกรม **P** ที่จำกัดที่มีรูทที่ <**p**,x,y> และไม่มีคอมพรีท คอมพิวเตชัน ทรี แล้ว **P** จะมีโปรซีเยอร์ที่ ไม่สมบูรณ์อยู่ พิสูจน์โดยการอุปนัย (induction) บน d ซึ่งเป็นความลึกสูงสุดของทุกๆ รีซเอเบิล คอมพิวเตชัน ทรีที่มีรูทอยู่ที่ <**p**,x,y> ดังนี้

ถ้า d เท่ากับ 1 แล้ว p ไม่มี top level trace สำหรับ <**p**,x,y> ดังนั้น มันจึงไม่ครอบคลุม <**p**,x,y> และเนื่องจาก <**p**,x,y> อยู่ใน **M** ดังนั้น **p** จึงไม่สมบูรณ์

สมมติว่าการอ้างสำหรับ d-1 ที่ d มากกว่า 1 เป็นความลึกสูงสุดของทุกๆ รีซเอเบิล คอมพิวเตชัน ทรี ที่มีรูทอยู่ที่ <**p**,x,y> ถ้าไม่มี top level trace ของ <**p**,x,y> อยู่ใน **M** แล้ว **p** ไม่ครอบคลุม <p,x,y> และ **p** จะทำให้การอ้างเป็นจริง หากเป็นกรณีอื่น พิจารณา <p,x,y> ใน top level trace มันจะมีอย่างน้อยหนึ่งโปรซีเยอร์ <**q**,u,v> ที่ไม่มีอยู่ในคอมพิวเตชัน ทรี ไม่เช่นนั้น จะมีคอมพรีท คอมพิวเตชัน ทรี สำหรับ <**p**,x,y> ซึ่งขัดแย้งกับสมมติฐานในตอนแรก

ดังนั้น ความลึกสูงสุดของแต่ละรีซเอเบิล คอมพิวเตชัน ทรีของ <**q**,u,v> เป็น d-1 โดยสมมติฐานความลึกสูงสุดของแต่ละรีซเอเบิล คอมพิวเตชัน ทรี ที่มีรูทอยู่ที่ <p,x,y> มีค่าเป็น d ดังนั้น สมมติฐานของการอ้าง ที่นำมาใช้กับ <q,u,v> และ d-1 โดยการอุปนัยสมมติฐาน **P** จะมีโปรซีเยอร์ที่ไม่สมบูรณ์ประกอบอยู่

3.4.2 อัลกอริทึมที่ใช้วินิจฉัยโปรซีเยอร์ที่ไม่สมบูรณ์

จากการพิสูจน์ทฤษฎีบทที่ 3.1 การพิสูจน์ทฤษฎีบทที่ 3.4 เสนอให้เห็นถึงอัลกอริทึมในการหาโปรซีเยอร์ที่ไม่สมบูรณ์ อัลกอริทึมนั้นใช้เอ็กซิสเทนเชียล ไควรี่ (existential query)ในการหาโปรซีเยอร์ โดยที่เอ็กซิสเทนเชียล ไควรี่ คือคู่ลำดับ <**p**,x> ซึ่งคำตอบของเอ็กซิสเทนเชียล ไควรี่ <p,x> ในการอินเตอร์พรีท **M** เป็นเซตของ y ที่ <**p**,x,y> อยู่ใน **M** โดยในสมมติฐานของการอินเตอร์พรีทที่ว่า เซตดังกล่าวเป็นเซตจำกัด

อัลกอริทึมนี้จะใช้ออราเคิล คอมพิวเตชัน (oracle computation) คือจะทำการถามออราเคิลให้ตอบคำถามของการเอ็กซิสเทนเชียล ไควรี่ โดยออราเคิล คอมพิวเตชันของ <p,x> เป็นการทำงานในทุกๆการเรียกโปรซีเยอร์ <q,u> ที่เป็นโปรซีเยอร์ใน <p,x> ซึ่งจำลองโดยการเรียก <q,u> ไปที่เอ็กซิสเทนเชียล ออราเคิลสำหรับ **M** ตามทางเลือกที่ได้ค่า v จากเซตที่ออราเคิลส่งค่ากลับมา

อัลกอริทึม 3: การหาโปรซีเยอร์ที่ไม่สมบูรณ์
อินพุต: <p,x,y> ใน **M** ที่ **p** เกิดไฟไนท์ เฟล
เอาท์พุต: <q,u,v> ใน **M** ที่ไม่ครอบคลุมโดย **q**
อัลกอริทึม: อัลกอริทึมจะเรียกรีเคอร์ซีฟโปรซีเยอร์ ip ด้วยอินพุต <p,x,y>
โปรซีเยอร์ ip จะทำงานดังนี้คือ ที่อินพุต <p,x,y> จะพยายามสร้างการจำลองของออราเคิลของการเรียกโปรซีเยอร์ <p,x> ซึ่งให้ผลลัพธ์ y โดยใช้ เอ็กซิสเทนเชียล ไควรี่ และขณะเดียวกันก็ทำการเก็บ top level trace ซึ่งเป็นลักษณะของการทำงาน ถ้ามันล้มเหลวในการทำงาน ip จะส่ง <p,x,y> ถ้ามันทำงานสำเร็จ มันจะทำการค้นหาเข้าไปใน top level trace ของ <q,u,v> ซึ่งอยู่ในโปรแกรม **P** ที่เกิดไฟไนท์ เฟล แล้วทำการเรียกตัวเองแบบรีเคอร์ซีฟด้วย <q,u,v> และส่งผลลัพธ์ของการรีเคอร์ซีฟกลับมา

ทฤษฎีบทที่ 3.5 : ให้ P เป็นโปรแกรม และ <p,x,y> เป็นการเรียกโปรซีเยอร์ **p** ด้วย x และให้ผลลัพธ์เป็น y ในโมเดล **M** ที่ **P** เกิดไฟไนท์ เฟล สมมติว่าแต่ละรีชเอเบิล คอมพิวเตชัน ทรีของ <p,x,y> มีความยาวทั้งหมดเป็น n และมีความลึกทั้งหมดเป็น d และขนาดสูงสุดของการรวมกันของ top level trace ในแต่ละ <q,u,v> ในแต่ละรีชเอเบิล คอมพิวเตชัน ทรีที่มีรูทอยู่ที่ <p,x,y> สำหรับแต่ละ z มีค่าเป็น b

ดังนั้น การทำงานของอัลกอริทึม 3 ที่ใช้ใน <p,x,y> มีความยาวทั้งหมดเป็น dn+1 ความลึกทั้งหมดเป็น d+1 จะมีการทำการไควรี่ทั้งหมด b(d-1)+1 ครั้ง และ ให้ผลลัพธ์ <q,u,v> ที่อยู่ใน **M** ที่ไม่ครอบคลุมโดย **q**

พิสูจน์ ทฤษฎีบทที่ 3.5 : การพิสูจน์ว่า ip สิ้นสุดการทำงานโดยตามอัลกอริทึมการวิเคราะห์ความซับซ้อน โดยจะเห็นได้ว่า หาก ip สิ้นสุดการทำงาน มันจะให้ <q,u,v> ที่อยู่ใน **M** ออกมาซึ่ง **q** ไม่ครอบคลุม <q,u,v>

สมมติว่า ip ถูกเรียกโดย <p,x,y> เพราะว่าโปรแกรม **P** มี อินฟินิต รีชเอเบิล คอมพิวเตชัน ทรี ที่มีรูทอยู่ที่ <p,x,y> โดยสมมติฐานของ ซึ่งมันจะประกอบด้วยรีชเอเบิล คอมพิวเตชัน ทรีที่มีรูทอยู่ที่ <p,x,y> ที่ไม่สิ้นสุดเท่านั้น ให้ n เป็นความยาวสูงสุด และ d เป็นความลึกสูงสุดของแต่ละทรี

พิสูจน์โดยการอุปนัยบน d ที่มีความลึกของการทำงานของ ip ทั้งหมดเป็น d+1 มีความยาวทั้งหมดเป็น dn+1 และ จำนวนของการไควรี่ทั้งหมดของการทำงานเป็น b(d-1) หาก d=1 หมายความว่าไม่มี top level trace ของ <p,x,y> ดังนั้น ip ไม่ต้องทำการไควรี่ใดๆเลย ดังนั้น ทั้งความลึกและความยาวของการทำงานของ ip เป็น d+1=dn+1=2 และ จำนวนการไควรี่จะเป็น b(d-1)=b(1-1)=0

สมมติว่าสำหรับการอ้าง d-1 ที่ d>1 ในกรณีของ ip ที่พยายามจำลองการทำงานของการเรียกโปรซีเยอร์ <**p**,x> ซึ่งให้ผลลัพธ์กลับมาเป็น y เพื่อที่จะทำสิ่งนั้น มันจะทำการไควรี่ b ครั้ง โดยสมมติฐานคือ ผลรวมของขนาดของแต่ละ top level trace ของแต่ละการเรียกโปรซีเยอร์ ใน รีซเอเบิล คอมพิวเตชัน ทรี ของ <**p**,x,y> เป็น b

จากขั้นดังกล่าว ip จะให้ผลลัพธ์ออกมาหรือไม่ก็ทำการเรียกตัวเองด้วย <**q**,u,v> ซึ่งเกิดไฟไนท์เฟล โดยสมมติฐานที่ว่า ความลึกของการทำงานของโปรซีเยอร์ **p** ที่อินพุตเป็น x มีค่าเป็น d และ ความยาวของมันเป็น n ความลึกของแต่ละการทำงานของโปรซีเยอร์ ใน top level trace ทั้งหมดเป็น d-1 และผลรวมสูงสุดของความยาวทั้งหมดเป็น n-1

โดยการอุปนัยสมมติฐานของจำนวนการไควรี่ ของการทำงาน ip บน <q,u> ทั้งหมดเป็น b(d-2) ครั้ง ดังนั้น จำนวนของการไควรี่ของการทำ ip บน <**p**,x> จะเป็น b(d-2)+b=b(d-1}ซึ่งทำให้คำกล่าวอ้างดังกล่าวได้ถูกพิสูจน์

3.4.3 การพัฒนาด้วยภาษาโปรล็อก

ต่อไปจะเป็นการนำภาษาโปรล็อกมาทำการอิมพลีเมนท์อัลกอริทึม 3 โดยกล่าวได้ว่า A เป็นความผิดพลาดโดยทันที (immediately fails) ใน **P** ถ้าไม่มีคลอสที่ A'←B' ใน **P** ซึ่ง A' ยูนิฟาย (unify) ด้วย A

โดยเป้าหมาย A จะเกิดไฟไนท์ เฟล ในโปรแกรม **P** ถ้าทุกการทำงานของ **P** บน A สิ้นสุด และในแต่ละการทำงานจะต้องมีอย่างน้อยหนึ่งเป้าหมายที่ผิดพลาดโดยทันที จากเรื่องลอจิกโปรแกรม ทฤษฎีที่ 3.4 กล่าวไว้ว่า ถ้าโปรแกรม **P** เกิดไฟไนท์ เฟล บนเป้าหมาย A ใน **M** แล้วมันจะมีเป้าหมาย B ใน **M** ซึ่งไม่มีคลอสใน P ที่คลอบคลุม B โปรแกรมที่จะแสดงต่อไปเป็นโปรแกรมที่สามารถตรวจหาเป้าหมายซึ่งทำการอิมพลีเมนท์มาจากอัลกอริทึมที่ 3 ดังนี้

```
ip((A,B),X) <- !,
      (A->ip(B,X);ip(A,X)).
ip(A,X)<-
      clause(A,B),satisfiable(B)->ip(B,X);X=A.

satisfiable((A,B))<-!,
      query(exists,A,true),satisfiable(B).
satisfiable(A)<-
      query(exists,A,true).
```

โปรซีเยอร์ ip(A,X) ทำงานด้วยความสัมพันธ์ที่ว่า ถ้า A เป็นเป้าหมายที่ผิดพลาดแบบไฟไนท์เฟลแล้ว X จะเป็นเป้าหมายที่ถูกต้องที่ไม่ถูกครอบคลุมนั่นคือ X เป็นเป้าหมายที่หายไปนั่นเองโดยเป้าหมายดังกล่าวจะถูกพบได้โดยการหาจากส่วนที่เกิดความผิดพลาดในการทำงาน บนสมมติฐานที่ว่าเป้าหมายดั้งเดิมที่ ip ถูกต้องและเกิดไฟไนท์เฟล การทำ ip ลงไปในส่วนนี้จะพบกับเป้าหมายที่ผิดพลาดโดยทันทีซึ่งเป้าหมายของ clause(A,B)จะล้มเหลวหรือไม่ถูกครอบคลุมในเป้าหมาย ซึ่งการเรียกโปรซีเยอร์ satistfiable(B) จะล้มเหลวในแต่ละการแก้ปัญหาของ clause(A,B) แต่ละ A ไม่ถูกครอบคลุม และได้เอาท์พุตโดย ip

โปรซีเยอร์ query(exits,A,V) ทำการไควรี่ผู้ใช้สำหรับตัวอย่างที่ถูกต้องของ A มันจะส่งค่าตัวอย่างกลับอย่างไม่สิ้นสุดซึ่ง V=true ถ้าแต่ละตัวอย่างมีค่า (เช่นให้คำตอบแรกจากผู้ใช้และแบ็คแทรคกลับถ้าจำเป็น) และให้ค่า V=false หากไม่เป็นเช่นนั้น ซึ่งคล้ายกับอินเตอร์พรีทเตอร์ระดับสูงของโปรล็อก ผู้ใช้ซึ่งตอบการไควรี่จะให้คำตอบทั้งหมดของตัวอย่างที่ถูกต้องของเป้าหมาย และจะจบด้วย no ถ้าระบบรู้ว่าเป้าหมายที่แน่นอนมีที่สิ้นสุด มันจะไม่ถามอีก ดังนั้น ถ้าเป้าหมายที่ถูกไควรี่เป็นเพรดดิเคทของระบบซึ่ง X>Y และ X≤Y ดังในตัวอย่างข้างต้น แล้วจะไควรี่ปัญหาตรงๆเพื่อแก้ปัญหา แทนที่จะเป็นการถามจากผู้ใช้

จะแสดงพฤติกรรมของ ip ในโปรแกรมการเรียงตัวเลขแบบ insertion sort ดังนี้ ในส่วนของ isort และ insert มีความสิ้นสุด ดังนั้นมีเพียงคำตอบเดียวที่จำเป็นในการ เอ็กซิสเทนเชียล ไควรี่

```
isort([X|Xs],Ys)<-isort(Xs,Zs),insert(X,Zs,Ys).
isort([],[]).

insert(X,[Y|Ys],[Y|Zs]) <- X>Y,insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys])<- X≤Y.
```

หากทำการไควรี่ตัวอย่างข้างต้น จะได้ผลเป็นดังนี้

```
| ?- isort([3,2,1],X).
no
```

ดังนั้น จะทำการเรียก ip บน isort([3,2,1],[1,2,3]),

```
| ?- ip(isort([3,2,1],[1,2,3]),X).
query: isort([2,1],X)? y.
which X? [1,2]
query:insert(3,[1,2],[1,2,3])? y.
query:isort([1],X)? y.
which X? [1]
query: insert(2,[1],[1,2])?y.
query:isort([],X)? y.
which X?[].
query:insert(1,[],[1])?y.
X=insert(1,[],[1])
```

และมันจะพบว่า insert(1,[],[1]) ไม่ถูกครอบคลุม พิจารณาคลอส2คลอสของการอินเซิร์ทและจะเห็นว่าคลอสทั้งสองไม่คลอบคลุมเป้าหมายนี้เลยเพราะว่าหัวของมันไม่ได้ยูนิฟายกับตัวมัน เพราะว่ามันคาดหวังว่าลิสที่ถูกอินเซิร์ทต้องไม่ว่างเปล่า ซึ่งก็คือคลอส insert(X,[],X) ขาดหายไป

การปรับปรุงความซับซ้อนของการไควรี่ (query-complexity) ของ ip สามารถได้รับโดยการทำการค้นหา top level trace ซึ่งอยู่ใน M กับการหาเป้าหมายที่ผิดพลาดไปด้วยไปพร้อมๆ กัน ในกรณีที่ดีที่สุด การปรับปรุงนี้สามารถเป็นแฟคเตอร์ของ b โดย b คือจำนวนสูงสุดของเป้าหมายในบอดี้ของคลอสใน

โปรแกรมที่ทำการหาความผิดพลาด ในกรณีที่แย่ที่สุด การไควรี่แบบซับซ้อนจะยังคงเหมือนเดิม โปรแกรมต่อไปนี้จะแสดงการปรับปรุงและจะใช้เป็นโปรแกรมในการพัฒนาระบบ

```
missing_solution(A)<-
      writel(['error: missing solution',A,'.diagnosing…']),nl,
      query(exists,A,true),\+solve(A,true)->
            ip(A,X),handle_error('uncovered atom',X);
      write('!Illlegal call to ip'),nl.

ip(A,X)<-
      clause(A,B),ip1(B,X)->true;X=A.
ip1((A,B),X)<-!,
      ( query(exists,A,true),(A,ip1(B,X);\+A,ip(A,X))).
ip1(A,X)<-
      query(exists,A,true),(A->break(ip1(A,X));ip(A,X)).
```

หากนำโปรแกรมข้างต้นมาทำการไควรี่ด้วยโปรแกรมการเรียงลำดับแบบอินเซิร์ทอีกครั้งจะได้ผลดังนี้

```
      | ?- ip(isort([3,2,1],[1,2,3],X).

      query:isort([2,1],X)?y.
      which X? [1,2].
      query:isort([1],X)?y.
      which X?[1].
      query:isort([],X)?y.
      which X?[].
      query:insert(1,[],[1])?y.

      X=insert(1,[],[1])
```

จะเห็นได้ว่า มีการไควรี่เพียง 4 ครั้งก็สามารถหาเป้าหมายที่ไม่ถูกครอบคลุมได้ เปรียบเทียบกับ ip เดิมคือใช้ถึง 6 ครั้ง

3.5 การวินิจฉัยโปรแกรมที่ไม่สิ้นสุดการทำงาน (Diagnosing Nontermination)

กับคำถามที่ว่า จะสามารถตรวจหาความผิดพลาดจากการที่โปรแกรมทำงานไม่สิ้นสุดได้อย่างไร แล้วจะรู้ว่าโปรแกรมทำงานไม่สิ้นสุดก็ต่อเมื่อ ต้องรอไปเป็นเวลาเท่าใด คำถามเหล่านี้โปรแกรมเมอร์ไม่สามารถคาดเดาคำตอบได้จากการดีบักโปรแกรม โปรแกรมที่ทำงานไม่สิ้นสุดอาจเกิดจากการที่เนื้อที่ที่จัดสรรถูกใช้หมดในการทำงานโปรแกรมหรือความอดทนในการรอของผู้ใช้สิ้นสุดลง

แต่ถึงแม้ว่าถ้าการทำงานใช้ทรัพยากรหมดมันก็ไม่ได้หมายความว่ามันทำงานไม่สิ้นสุด อาจเป็นเพราะว่าโปรแกรมมีประสิทธิภาพไม่เพียงพอ ความอดทนของผู้ใช้ไม่เพียงพอ หรือคอมพิวเตอร์มีขนาดใหญ่ไม่เพียงพอก็ได้ โดยจริงๆแล้ว อัลกอริทึมการหาความผิดพลาดที่จะอธิบายต่อไปนี้มันอาจล้มเหลวในการตรวจหาความผิดพลาดในโปรแกรมที่ใช้ทรัพยากรหมด และในแต่ละกรณีมันต้องขึ้นกับโปรแกรมเมอร์ในการตัดสินใจถึงสาเหตุของสิ่งที่เกิดขึ้นเอง

3.5.1 การสิ้นสุดการทำงาน (Termination)

งานในการหาความผิดพลาดจากการที่โปรแกรมไม่สิ้นสุดโดยการใช้เครื่องมือมีประโยชน์มาก ในการที่พิสูจน์ว่าโปรแกรมสิ้นสุดการทำงาน ให้ S เป็นเซตที่ไม่ใช่เซตว่าง ให้ เวล ฟาว ออเดอร์ลิ่ง (well found ordering) > บน S เป็น สตริก พาเชียล ออเดอร์ลิ่ง (strict partial ordering) บน S ที่ไม่มีลำดับอนันต์ที่ลดลง (infinite descending sequence) จะมี > ที่เป็นความสัมพันธ์แบบทวิภาค (binary) บน S ซึ่งมีคุณสมบัติการถ่ายทอด (transitive) อสมมาตร(asymetric) และไม่สะท้อน (irreflexive) และไม่มีลำดับอนันต์ x1,x2, ... ซึ่งเป็นสมาชิกของ S จะมี x1>x2>...

เลมม่าที่ 3.3: โปรแกรม **P** จะสิ้นสุดเสมอ (everywhere terminating) ก็ต่อเมื่อมีเวล ฟาวด์ ออเดอร์ลิ่ง > บนเซตของการเรียกโปรซีเยอร์ซึ่งสำหรับทุกๆการทำงานของโปรแกรม **P** ที่ <**p**,x> เรียก <**q**,u> นั่นคือกรณีที่ว่า <**p**,x> > <**q**,u>

พิสูจน์เลมม่าที่ 3.3: ถ้าหากมี เวล ฟาวด์ ออเดอร์ลิ่ง แล้ว ความลึกของแต่ละการทำงานของ **P** มีความจำกัด ดังนั้นการทำงานจึงสิ้นสุด

สมมติว่าทุกๆการทำงานของ **P** สิ้นสุดลง ให้ d(<**p**,x>)เป็นความลึกสูงสุดของทุกการทำงานของ **p** บน x อธิบายลำดับของ > ได้เป็น <**p**,x> > <**q**,u> ก็ต่อเมื่อ d(<**p**,x>) > d(<**q**,u>) มันจะเห็นได้ง่ายว่าลำดับที่อธิบายเป็น เวล ฟาวด์

ดังในความผิดพลาดสองแบบก่อนหน้านี้ จะอธิบายคุณสมบัติของโปรซีเยอร์สำหรับการจะรู้ได้ว่าโปรแกรมนั้นไดเวิร์จ แสดงว่ามันประกอบด้วยโปรซีเยอร์กับคุณสมบัติดังกล่าวนั้น อย่างไรก็ตามความจริงที่ว่า โปรซีเยอร์ **p** จะทำการเรียกโดยฝ่าฝืน เวล ฟาวด์ ออเดอร์ลิ่ง ก็ไม่ได้หมายความว่าโค๊ดของ **p** ผิด พิจารณาโปรแกรมการเรียกลำดับแบบควิ๊กซอร์ทที่นำมาดีบักดังต่อไปนี้

```
qsort([X|Xs],Ys)<-
          partition(Xs,X,Xs1,Xs2),
          qsort(Xs1,Ys1),qsort(Xs2,Ys2),
          append(Ys1,[X|Ys2],Ys).
qsort([],[]).

partition([X|Xs],Y,Xs1,[X|Xs2])<-
     Y<X,partition(Xs,Y,Xs1,Xs2).
partition([X|Xs],Y,[X|Xs1],Xs2)<-
     X≤X,partition(Xs,Y,Xs1,Xs2).
partition([],X,[X],[]).

append([X|Xs],Ys,[X|Zs]) <- append(Xs,Ys,Zs).
append([],Xs,Xs).
```

การทำงานของโปรแกรมบนเป้าหมาย qsort([1,2,1,3],X) ไม่สิ้นสุด ซึ่งก็คือ qsort([1,1],Ys) อยู่ในลูป ซึ่งเซกเมนท์เริ่มต้นของสแต็กของการทำงานเป็นดังนี้

```
qsort([1,2,1,3],Ys)
qsort([1,1],Ys)
qsort([1,1],Ys)
qsort([1,1],Ys)
...
```

อย่างไรก็ตาม ปัญหานี้ไม่ได้อยู่ในโปรซีเยอร์ qsort หากแต่อยู่ในโปรซีเยอร์ partition เพราะมันจะให้ลิสของเอาท์พุตยาวกว่าลิสของอินพุตเช่นหากไควรี่

```
| ? partition([1],1,Xs,Ys).
Xs=[1,1],
Ys=[]
```

ถ้าพิจารณาโค้ดของการ partition (หรือหาความผิดพลาดของ partition([1],1,[1,1],[]) โดยใช้ fp) จะพบว่ามันผิดคือควรจะเป็น partition([],X,[],[]) จึงจะถูกตามนิยาม

นิยามที่ 3.2 : ให้ **M** เป็นการอินเตอร์พรีต และ > เป็น เวล ฟานด์ ออเดอร์ลิ่ง บนการเรียกโปรซีเยอร์ โปรซีเยอร์ **p** จะกล่าวได้ว่าไดเวิร์จกับ > และ **M** ถ้ามันมี <**p**,x,y> กับ top level trace S ซึ่ง

1. มี <**q**,u,v> ใน S ที่ <**p**,x> ไม่เป็นเวล ฟาวด์ ออเดอร์ลิ่งของ <**q**,u>
2. ทุกๆการเรียกโปรซีเยอร์ใน S ก่อนหน้านี้ <**q**,u,v> อยู่ใน **M**

จำไว้ว่า การรู้ เวล ฟาวด์ ออเดอร์ลิ่งโดยตัวมันเองไม่เพียงพอที่จะตรวจหาความผิดพลาดในความผิดพลาดจากการทำงานที่ไม่สิ้นสุด จำเป็นต้องรู้เจตนาของการอินเตอร์พรีตให้ดี รู้ผลลัพธ์ที่แท้จริงของการเรียกโปรซีเยอร์ที่ทำงานก่อนที่จะเกิดการละเมิดลำดับที่ถูกต้อง

กล่าวได้ว่าโปรซีเยอร์จะติดลูป หากการเรียกตัวเองด้วยอินพุตเดิมถูกเรียก ซึ่งโปรซีเยอร์ที่ติดลูปถือเป็นไดเวิร์จ

ทฤษฎีบทที่ 3.5: ให้ **P** เป็นโปรแกรม **M** เป็นการอินเตอร์พรีต และ > เป็นเวล ฟาวด์ ออเดอร์ลิ่ง บนการเรียกโปรซีเยอร์ ถ้า **P** ไดเวิร์จแล้วมันจะมีโปรซีเยอร์ที่ไม่ถูกต้องใน **M** หรือโปรซีเยอร์ที่ไดเวิร์จกับ > และ **M**

พิสูจน์ทฤษฎีบทที่ 3.5: สมมติว่าการทำงานของโปรซีเยอร์ **p** บนอินพุต x ไม่สิ้นสุด โดยสมมติฐานของภาษาที่ใช้ในการโปรแกรมที่ได้กล่าวในหัวข้อ 3.2 คือมี รีซเอเบิล คอมพิวเตชัน ทรี กับ ส่วนของการเรียกโปรซีเยอร์ที่อนันต์ ซึ่งส่วนดังกล่าวต้องประกอบด้วยการเรียกโปรซีเยอร์สองส่วนตามกันมาคือ <**p**,x> และ <**q**,u> ซึ่ง <**p**,x> ไม่เป็นเวล ฟาวด์ ออเดอร์ของ <**q**,u> พิจารณาการเรียกโปรซีเยอร์ทั้งหมดถ้าแต่ละการเรียกโปรซีเยอร์ของ **p** ด้วย x ถูกกระทำก่อนการเรียก <**q**,u> ซึ่งแต่ละการคืนค่าของมันให้เอาท์พุตที่ไม่ถูกต้อง แล้ว โปรแกรม **P** จะไม่เป็น พาเชียล คอเรค และจากทฤษฎีบทที่ 3.2 โปรแกรม **P** จะประกอบด้วยโปรซีเยอร์ที่ไม่ถูกต้อง หากเป็นในกรณีอื่น จะมี top level trace ของ **p** บนอินพุต x ซึ่งมี <**q**,u,v>

สำหรับบาง v ดังนั้นทุกๆการเรียกโปรซีเยอร์ <q,u,v> ก่อนหน้าในเส้นทางอยู่ใน **M** และโดยนิยามแล้ว **p** จะไดเวิร์จที่ > และ **M**

3.5.2 อัลกอริทึมที่ใช้วินิจฉัยโปรซีเยอร์ที่ไดเวิร์จ

ในการตรวจหาความผิดพลาดมีพื้นฐานอยู่บนสมมติฐานที่ว่า ถ้าโปรแกรมเมอร์ไม่สามารถอธิบายเวล ฟาวด์ ยอเดอร์ลิ่งได้อย่างชัดเจน เขาต้องมีลำดับการทำงานอยู่ในใจตอนที่เขียนโปรแกรมอยู่แล้ว หรือเมื่อเกิดลูปหรือการทำงานที่ไดเวิร์จ สามารถตัดสินใจว่าการเรียกโปรซีเยอร์ใดเกิดความผิดพลาด สมมติฐานหลังจากนี้เป็นส่วนเล็กๆอันหนึ่ง และต้องถือว่าช่วยโปรแกรมเมอร์ให้สามารถทำการดีบักโปรแกรมได้

อัลกอริทึมที่ 4 ที่จะกล่าวต่อไปนี้ต้องการกราวด์ ออราเคิล สำหรับ M และ ออราเคิลสำหรับ > ซึ่งจะแบ่งว่าสามารถตอบคำถามการไควรี่ในรูปแบบที่ว่า <p,x> > <q,u>? สำหรับทุกๆโปรซีเยอร์ **p** และ **q** ใน **P** มันสมมติว่าทุกๆการทำงานของโปรซีเยอร์ **p** บนอินพุต x มีขอบเขตซึ่งถูกกำหนดไว้ d ในความลึกของมันซึ่งไม่สามารถละเมิดได้

อัลกอริทึมที่ 4: การหาโปรซีเยอร์ที่ไดเวิร์จ
อินพุต: โปรซีเยอร์ **p** ในโปรแกรม **P** และมีอินพุตเป็น x และ จำนวนเต็มที่ d>0 ซึ่งเป็นความลึกของการทำงานของโปรซีเยอร์ **p** บนอินพุต x ที่ละเมิด d
เอาท์พุต: <**q**,u,v> ไม่ได้อยู่ใน **M** ซึ่ง **q** ที่มีอินพุตเป็น u ให้เอาท์พุต v และมี2การเรียกโปรซีเยอร์ <**q**,u>, <**r**,w> ซึ่งละเมิด เวล ฟาวด์ ออเดอร์ลิ่ง หรือกล่าวว่าไม่พบการไดเวิร์จ
อัลกอริทึม: อัลกอริทึม จำลองการเรียกโปรซีเยอร์ **p** ด้วยอินพุต x เมื่อความลึกของการทำงานเกิน d มันจะยกเลิกการทำงานและส่งค่าที่คงอยู่ในสแต็กของการเรียกโปรซีเยอร์ปัจจุบันกลับมา อัลกอริทึมจะพิจารณาสแต็กสำหรับ2การเรียกโปรซีเยอร์ที่ตามกันมาคือ <**p**,x>,<**q**,u> ซึ่ง <**p**,x> ไม่เป็นเวล ฟาวด์ ออเดอร์ลิ่งของ <**q**,u> หากมันพบการเรียกโปรซีเยอร์ดังกล่าว มันจะทำการค้นหา โดยใช้ กราวด์ออราเคิลสำหรับการเรียกโปรซีเยอร์ **p** ด้วยอินพุต x ก่อนการเรียก <**q**,u> ซึ่งให้เอาท์พุตเป็น v ที่ไม่ถูกต้องใน **M** ถ้าหากพบโปรซีเยอร์ดังกล่าว อัลกอริทึมนี้จะทำการเรียก fp จากอัลกอริทึมที่ 2 ด้วย <q,u,v> และให้ผลลัพธ์ของ fp ไม่เช่นนั้น อัลกอริทึมจะให้ <**p**,x>,<**q**,u> ถ้าหากพบว่าไม่ได้ละเมิด เวล ฟาวด์ ออเดอร์ลิ่ง อัลกอริทึมจะส่งผลกลับมาว่า ไม่พบการไดเวิร์จ

3.5.3 การอิมพลีเมนท์ด้วยภาษาโปรล็อก

ได้อธิบายอินเตอร์พรีเตอร์ภาษาโปรล็อกที่รับอินพุต เป้าหมาย และ ขอบเขตความลึก และจะให้ผลลัพธ์กลับมาเป็น true และเป้าหมายตัวอย่างของ A ถ้ามันสามารถแก้ปัญหาได้สำเร็จและไม่เกินขอบเขตความลึกที่กำหนดให้ หรือจะส่งค่าสแต็กของเป้าหมายที่มีความลึกเป็น d ถ้าหากมีการทำงานเกินขอบเขตที่ตั้งไว้

พิจารณาโปรแกรมอินเตอร์พรีเตอร์ที่มีการกำหนดขอบเขตความลึกดังต่อไปนี้

```
solve(true,D,true)<-!
solve(A,0,(overflow,[]))<-!.
solve((A,B),D,S)<-!,
      solve(A,D,Sa),
      (Sa=true -> solve(B,D,Sb),s=sb;s=sa).
solve(A,D,Sa)<-
      system(A)->A,Sa=true;
      D1 is D-1,
      clause(A,B), solve(B,D1,Sb),
      (Sb=true -> sa = true;
      Sb=(overflow,S)->Sa=(overflow,[A|S])).
```

พิจารณาโปรแกรมการเรียงลำดับแบบอินเซิร์ทต่อไปนี้

```
      isort([X|Xs],Ys)<- isort(Xs,Zs),insert(X,Zs,Ys).
      isort([],[]).

      insert(X,[Y|Ys],[X,Y|Ys]) <- X ≤ Y.
      insert(X,[Y|Ys],Zs) <- insert(X,Ys,Ws),insert(Y,Ws,Zs).
      insert(X,[],[X]).
```

เมื่อนำโปรแกรมการเรียงลำดับแบบอินเซิร์ทมาทำการไดวรี่ด้วยอินพุต[2,1,3] โปรแกรมจะทำงานไม่สิ้นสุด หาก ทำการไดวรี่ isort([2,1,3],X) ด้วย solve ซึ่งมีขอบเขตความลึกเป็น 6 จะได้ผลลัพธ์ปืนดังนี้

```
      | ? solve(isort([2,1,3],X),6,S).
      S = (overflow,[isort([2,1,3],X),isort([1,3],[1,3]),
            insert(1,[3],[1,3],X),insert(3,[1],[1,3]),
            insert(1,[3],[1,3]),insert(3,[1],[1,3])]),
      X = X
```

จะเห็นได้ว่าโปรแกรมที่ได้มีการติดลูป เนื่องจากอัลกอริทึมการตรวจหาความผิดพลาดจำเป็นต้องตรวจผลของการเรียกโปรซีเยอร์ทางซ้ายของการไดเวิร์จว่าถูกต้อง ก่อนจะสรุปได้ว่าคลอสดังกล่าวเกิดการไดเวิร์จ มันจะมีประสิทธิภาพมากถ้าจะเก็บผลระหว่างการทำงาน และส่งผลลัพธ์กลับเมื่อ stack overflow แทนที่จะทำการคำนวนอีกครั้ง ถ้าต้องการให้เป็นดังที่กล่าวมา คลอสสุดท้ายของ solve ก็ควรเป็นดังนี้

```
      Sb = (overflow,S) -> Sa = (overflow,[(A<-B)|S]).
```

โปรแกรมต่อไปจะเป็นการอิมพลีเมนท์อัลกอริทึมที่ 4 โดยใช้เทคนิคการหาแบบเชิงเส้น

```
stack_overflow(P,S)<-
      write(['error: stack overflow on 'P'. diagnosing…']),nl,
      (find_loop(S,Sloop)-> check_segment(Sloop);
      check_segment(S)).
```

```
find_loop([(P<-Q)|S],Sloop) <-
      looping_segment((P<-Q),S,S1) -> Sloop = [(P<-Q)|S1];
      find_loop(S,Sloop).

looping_segment((P <- Q),[(P1<-Q1)|S],[(P1<-Q1)|Sl])<-
      same_goal(P,P1) -> writel([P,'is looping.']), nl, Sl=[];
      looping_segment((P<-Q),S,Sl).

check_segment([(P<-Q),(P1<-Q1)|S])<-
      query(legal_call,(P,P1),true) ->
      check_segment([(P1<-Q1)|S]);
      false_subgoal(P,Q,P1,Q1) -> false_solution(Q1);
      handle_error('diverging clause',(P<-Q)).

false_subgoal(P,(Q1,Q2),P1,Q)<-
      Q1≠P1 ,
      (query(forall,Q1,false) -> Q = Q1;
      false_subgoal(P,Q2,P1,Q)).
```

บทที่ 4
โครงสร้างทางลอจิกของดีบักเกอร์

4.1 บทนำ

จากเนื้อหาที่ได้กล่าวมาแล้วก่อนหน้านี้ สามารถนำมาอิมพลีเมนท์กับโปรแกรมภาษาโปรล๊อกให้สามารถทำการวินิจฉัยได้ หากต้องการพัฒนาโปรแกรมให้มีความฉลาดมากขึ้น ต้องทำให้ดีบักเกอร์สามารถเรียนรู้ และลดการไควรี่จากผู้ใช้ลง ซึ่งในบทนี้จะกล่าวถึงการเรียนรู้ของดีบักเกอร์ เพื่อพัฒนาให้ดีบักเกอร์เป็นแพรกมาทิสท์ (Pragmatist) โดยจะพูดถึงรายละเอียดในโครงสร้างของโปรแกรม ซึ่งอยู่บนพื้นฐานของวิธีการเชิงวิทยาศาสตร์ (scientific method) และจะกล่าวถึงโครงสร้างของโปรแกรมในส่วนของออราเคิล เอเจนต์ (Oracle Agent)

4.2 ภาพรวมของโครงสร้างแพรกมาทิสท์ (Overview of the PRAGMATIST Framework)

แพรกมาทิสท์เป็นโครงสร้างทางลอจิกสำหรับการพัฒนาโปรแกรม โดยการอิมพรีเมนท์ผ่านทางลอจิกโปรแกรมมิ่งในระดับเมตา ในโครงสร้างแพรกมาทิสท์มีการใช้กระบวนการทางวิทยาศาสตร์ (Scientific Method) ซึ่งแบ่งได้เป็น 3 ขั้นตอนหลักๆ

ตั้งสมมติฐาน (proposing hypothesis)
คาดเดาสาเหตุ (deriving predictions)
ทดสอบหาข้อขัดแย้ง (testing these against observations)

หากนำเอาทฤษฎีดังกล่าวมาทำการพัฒนาโปรแกรม ตามขั้นตอนจะได้ดังนี้

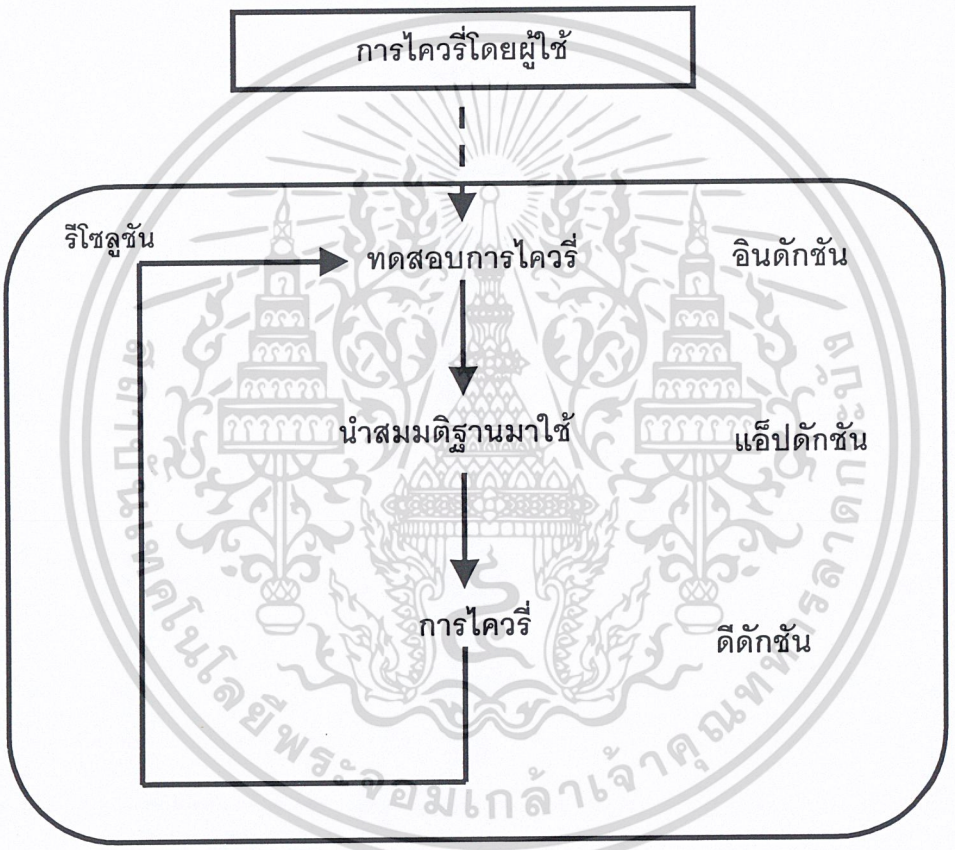
ตั้งคลอสของโปรแกรม (proposing program clauses)
คำนวณคำตอบของมัน (computing answers from them)
ทดสอบคำตอบที่ขัดแย้ง (testing these against intended answers)

ขั้นตอนทั้ง 3 ขั้น เป็นกระบวนการทางลอจิกคือ

แอ็ปดักชัน (abduction)
ดีดักชัน (deduction)
อินดักชัน (induction)

การทำงานทางลอจิกที่กล่าวมาทั้ง 3 นี้ให้สันนิษฐานว่าเป็นการอิมพลีเมนท์ที่ถูกต้อง ดังนั้นคำตอบที่ไม่ตรงกันระหว่างการดีดักชัน และอินดักชัน ต้องให้ถือว่าเป็นข้อบกพร่องของสมมติฐานเท่านั้น ซึ่งเป้าหมายพื้นฐานของการทดสอบคำตอบเป็นการทดสอบสมมติฐาน

ขั้นตอนที่ถูกต้องที่ประกอบขึ้นมาจากนิยามที่ได้กำหนดไว้ ดังนั้น ขั้นตอนจะประกอบด้วย 3 ขั้นด้วยกัน คือการไควรี่เพื่อการทดสอบ (induction) การนำสมมติฐานมาใช้ (abduction) และการได้มาซึ่งผลลัพธ์ (deduction) ซึ่งดังแสดงในรูปที่ 4-1



รูปที่ 4-1 ขั้นตอน 3 ขั้นของการทำงานของแพรกมาทิสท์

4.3 เอเจนต์ที่ใช้ในโครงสร้างแพรกมาทิสท์ (The Agents used in PRAGMATIST)

แพรกมาทิสท์ปฏิบัติกับโปรแกรมเหมือนสมมติฐานซึ่งได้มาจากกระบวนการของการแอ็ปดักชัน มุมมองนี้สามารถสนับสนุนกระบวนการของลอจิก ตัวอย่างเช่น การหาคำตอบของโปรแกรมจากสเป็กซิฟิเคชัน (specification) ของโปรแกรม ในทางปฏิบัติสามารถผิดพลาดได้ ดังนั้น ควรสร้างสมมติฐานเพียงหนึ่งอันเพื่อแทนสิ่งที่ต้องการ อย่างไรก็ตามสมมติฐานสามารถคาดเดาได้จากความต้องการ เซตของการคาดเดาทั้งหมดสามารถนิยามได้ว่าเป็น แพรกมาติก มีนนิ่ง (pragmatic meaning)

เพราะโดยทั่วไปแล้วโปรแกรมจะมีโครงสร้างที่ซับซ้อน ไม่สามารถสังเกตสิ่งที่ถูกกระทำได้โดยตรง นั่นคือ วิธีการในการเปรียบเทียบโดยตรงกับสิ่งที่ รู้ว่ามันถูกต้องว่า จะทำนายสิ่งเหล่านั้นได้อย่างไร อย่างไรก็ตามหากสามารถรู้ได้ก็เป็นสิ่งดี การทำนายเหล่านั้นเป็นคำตอบของการทำงานของมันในเงื่อนไขของการคาดคะเนความถูกต้องของการเอ็กซิคิว

โครงสร้างของแพรกมาทิสท์ประกอบด้วยเอเจนต์สำคัญสองส่วนด้วยกันคือ โปรแกรมเอเจนต์ และออราเคิลเอเจนต์ โปรแกรมเอเจนต์เป็นส่วนที่มีหน้าที่สำหรับเข้าไปใน โปรแกรมระดับออปเจกต์ (object-level program) ที่ถูกพัฒนาและสมมติฐานที่ประกอบขึ้นมา แล้วทำการดีดิวซ์ให้ได้การคาดคะเนน ส่วนออราเคิลเอเจนต์ จะใช้ในการทดสอบการคาดคะเน

โปรแกรมเอเจนต์เป็นโปรแกรมที่ทำงานอย่างตรงไปตรงมาไม่สามารถคิดอย่างมีเหตุผลมากนัก คือมีความจำกัดในการดีดิวซ์ผลลัพธ์ โดยเหตุผลของเอเจนต์ที่ใช้ในการคิดนั้นเป็นเซตของแอ็กเซียม (axiom) ซึ่งก็เปรียบเสมือนทฤษฎีพื้นฐานของเอเจนต์

นิยามที่ 4.1 : นาอีฟ เอเจนต์ (naïve agent) คือ <A,I,T> เมื่อ **A** เป็นชื่อของเอเจนต์, **I** เป็นเซตของอินเฟอร์เรนส์ รูล (inference rule) และ **T** เป็นทฤษฎี

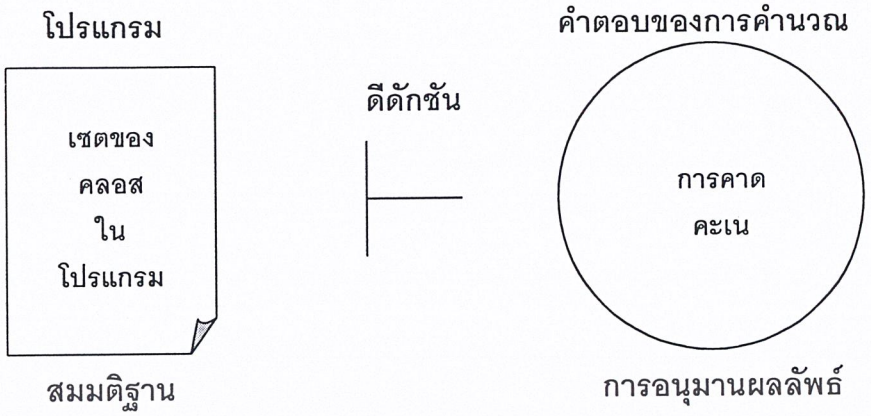
ผลลัพธ์ของ **T** อาจได้รับโดยการประยุกต์อินเฟอร์เรนส์ รูล **I** ซึ่งจะคาดคะเนว่าเหมาะสมและสมบูรณ์สำหรับเป้าหมาย เวลาเดียวกัน **I** และ **T** จะประกอบด้วย อินเฟอร์เรนส์ ซิสเต็ม (inference system) และดังนั้น นาอีฟ เอเจนต์เป็นส่วนสำคัญของอินเฟอร์เรนส์ ซิสเต็ม

ความหมายสำหรับ **T** สามารถได้รับโดยการให้ **I** อนุมานผลลัพธ์ของมันซึ่ง ได้นิยามไว้ตอนต้นประกอบเป็นความหมายในความรู้ของแพรกมาทิสท์ โดย **I** จะปฏิบัติตัวเหมือนเป็นอินเตอร์พรีตเตอร์ ซึ่งเอเจนต์หนึ่งสามารถสื่อสารกับเอเจนต์อื่นได้ เพื่อสามารถแลกเปลี่ยนความรู้ระหว่างกัน

ในระบบดีบักเกอร์มีเอเจนต์สองตัวดังที่ได้กล่าวไปแล้วซึ่งมีหน้าที่สร้างและทดสอบการคาดคะเน ซึ่งเครื่องมือนั้นจำเป็นต้องได้ผลลัพธ์ของการทดสอบ โดยจะใช้คำเรียกเอเจนต์ว่าแพรกมาทิสท์สำหรับเอเจนต์ที่รู้จักเหตุผล ไม่ใช่แค่เพียงสร้างการคาดคะเน แต่ต้องพิจารณาความเชื่อของมันว่าสิ่งไหนถูกและสิ่งไหนผิด มันจึงจำเป็นต้องใช้เมตาโปรแกรม ดังนั้นแต่ละการคาดคะเนจะต้องมีคำอธิบาย โดยจำเป็นแล้ว เซตของสมมติฐานได้รับมาจากการคาดคะเน โดยแพรกมาทิสท์จะใช้เหตุผลที่มีในการปฏิเสธการคาดคะเนที่เป็นคลอสที่ผิดพลาด

4.3.1 นาอีฟ โปรแกรม เอเจนต์ (The Naïve 'Program' Agent)

หากเปรียบแพรกมาทิสท์ เอเจนต์เป็นเสมือนมนุษย์ ที่สามารถคิดอย่างมีเหตุผล ที่สามารถอนุมานหรือ นิรนัย ผลลัพธ์ได้จากข้อมูลที่ได้มา ก็สามารถเปรียบนาอีฟ เอเจนต์ได้กับคอมพิวเตอร์ที่ทำงานทั่วๆไปคือสามารถทำงานให้ได้คำตอบตามที่มีคำสั่งมา แต่ไม่สามารถคิดหาเหตุผลที่ลึกซึ้งกว่านั้นได้ แสดงโปรแกรม(สมมติฐาน) และการคาดคะเนให้เห็นภาพดังรูปที่ 4-2



รูปที่ 4-2 โปรแกรม และ การคาดคะเนของนาอีฟ เอเจนต์

นาอีฟโปรแกรมเอเจนต์ที่สร้างขึ้นมามีความสามารถในการอินเตอร์พรีทได้ คือมันจะทำตัวเองเป็นอินเตอร์พรีทเตอร์ที่มีหน้าที่ให้ผลลัพธ์จากการไควรี่ของผู้ใช้ นั่นคือมันจะทำงานอย่างตรงไปตรงมาโดยนำการไควรี่ของผู้ใช้มาทำงานและให้ผลลัพธ์ออกไปโดยไม่มีการคิดด้วยเหตุผลที่ลึกซึ้งและเนื่องจากไม่สามารถทำการคิดอย่างมีเหตุผลมากนักมันจึงถูกเรียกว่า นาอีฟโปรแกรมเอเจนต์ แต่ จะเพิ่มความสามารถในการดีบักให้กับมัน โดยอาจกล่าวได้ว่า นาอีฟโปรแกรมเอเจนต์ที่สามารถดีบักได้ ก็คือแพรกมาทิสท์ เอเจนต์

4.3.2 นาอีฟ ออราเคิล เอเจนต์ (The Naïve 'Oracle' Agent)

Shapiro ได้สมมติเอเจนต์ตัวหนึ่งขึ้นมา โดยให้ชื่อมันว่า ออราเคิล โดยต้องการให้เป็นเสมือนผู้รู้ที่สามารถตัดสินใจว่าการทำงานของโปรแกรมถูกต้องหรือไม่ การทดสอบแต่ละคำตอบที่ถูกคำนวณเป็นกระบวนการของการอินดักชันซึ่งเกี่ยวข้องกับการเลือกตัวอย่างจากกลุ่มตัวอย่าง

ถึงแม้กลไกภายในสำหรับออราเคิลจะเป็นสิ่งไม่สำคัญในโครงสร้างทั้งหมดของการดีบัก ในที่นี้จะแสดงแหล่งของความรู้ที่สามารถบรรจุลงในออราเคิล 3 แหล่งด้วยกันคือ อินเทนด์เดด โมเดล, พร็อพเพอร์ตี้ และสเปกซิฟิเคชัน

4.3.2.1 อินเทนด์เดด โมเดล (Intended Models)

อินเทนด์เดดโมเดลเป็นเซตของกราวด์อะตอมที่ถูกต้องแน่นอน แต่แนวความคิดนี้มีข้อจำกัด ไม่เพียงแต่ในโปรแกรมการดีบักเท่านั้น แต่รวมถึงสามัญสำนึกทั่วไปด้วย เพราะการจดจำเซตที่มีขนาดมหาศาลนี้ไม่น่าจะเป็นวิธีที่ผู้ใช้เก็บอินเทนด์เดดโมเดล ถ้าจะพูดให้ถูกต้องคืออินเทนด์เดดโมเดลควรแสดงเป็น เซตของคำตอบทั้งหมดที่ได้มาจากสมมติฐานของผู้ใช้ ดังนั้น ถ้าสมมติฐานของผู้ใช้มีการเปลี่ยนแปลงในภายหลัง อินเทนด์เดดโมเดลก็จะเปลี่ยนด้วย ถึงแม้ว่าระหว่างนั้นโปรแกรมจะยังคงเหมือนเดิมก็ตาม ยิ่งกว่านั้นในแนวทางของทฤษฎี-โมเดล (model-theoretic) มันไม่เหมาะสมอย่างยิ่งกับคอนเซ็ป

ของตรวจสอบสมมติฐาน ความผิดพลาด และ ข้อขัดแย้งในวิธีการเชิงวิทยาศาสตร์ สำหรับการสืบสวนเชิงวิทยาศาสตร์ ค่าของสมมติฐานสามารถพิจารณาจากฟังก์ชันของค่าที่กำหนดในการดีดิวซ์จากการคาดเดาได้เท่านั้น

ในที่นี้ ความคิดของอินเทนด์เดดโมเดลเป็นเหมือนการพิสูจน์ทฤษฎีอย่างหนึ่ง ซึ่งมันเป็นเซตของกราวด์อะตอมที่สามารถดีดิวซ์ได้จากความตั้งใจที่ผู้ใช้ต้องการสุดท้ายจากโปรแกรมขณะทำการพัฒนา

นิยามที่ 4.2 : ให้อินเทนด์เดดโมเดล **MI** เป็นเซตของกราวด์อะตอมที่สามารถทำการดีดิวซ์ได้จากสมมติฐานของผู้ใช้ในคำตอบของการทำงาน

จากตัวอย่างต่อไปนี้ สมมติว่าโปรแกรมในระดับออปเจกต์ เป็นโปรแกรม append

```
append([],Z,Z).
append([U|X],Y,[U|Z]) <- append(X,Y,Z)
```

แล้วจะมี **MI** เป็นเซตดังนี้

```
{append([],[],[]),append([a],[b],[a,b]),
append([turing],[peirce,popper],[turing,peirce,popper]),
append([1],[],[1]),… }
```

ตัวอย่างง่ายๆ ที่แสดงเพียงพอที่จะชี้ให้เห็นถึงข้อจำกัดของทฤษฎีโมเดล ประการแรก ตัวอย่างดังกล่าวคือลอจิกโปรแกรมที่ใช้สม่ำเสมอในการต่อลิส แต่มันคำนวณจำนวนของคำตอบที่แน่นอนไม่สิ้นสุดที่ผู้ใช้ไม่ได้ต้องการ เช่นหากทำการ append([],3,3) ความผิดพลาดนี้คือโปรแกรมไม่สามารถแสดงถึงความต้องการอย่างที่โปรแกรมได้ประกาศไว้

ประการที่สอง โปรแกรมข้างต้นประกอบด้วยลิสที่ถูกสร้างจากลิสว่างและลิส-คอนสตัคเตอร์ (list-constructor) เท่านั้น ซึ่งหากทำการ append([],[[]],[[]]) และนั่นหมายความว่าไม่มีค่าคงที่หรือสัญลักษณ์ของฟังก์ชัน ต้องทำการแทนที่โมเดลข้างบนให้สามารถเขียนทั้งโปรแกรมและไควรี่ ความต้องการ และด้วยเหตุนั้น ต้องยอมรับอะตอมที่แสดงใน MI อย่างไรก็ตาม มันไม่มีเหตุผลในการจำกัดความอิสระของโปรแกรมเมอร์ในการไควรี่ได้ ในทางปฏิบัติ เห็นชัดว่าได้ถูกกำหนด ซึ่งมุมมองนี้ ตรงกับวิธีการเชิงวิทยาศาสตร์โดยทั่วไป

การที่ไม่สามารถควบคุมได้อย่างเต็มที่ภายในกฎเกณฑ์ทางวิธีการเชิงวิทยาศาสตร์เป็นส่วนพื้นฐานที่ทำให้เกิดความผิดพลาด เมื่อความผิดปรกติใหม่ถูกพบ ต้องหาคำมาอธิบายเช่นเดียวกับการการดีบักโปรแกรม ภาษาที่ใช้อินเทนด์เดดโมเดลจะดีกว่าโปรแกรมปรกติ โดยเฉพาะหากมีคลอสบางคลอสขาดหายไป เช่นเดียวกับคลอสใหม่ถูกเพิ่มเข้าไป ภาษาจะจำเป็นต้องขยายลำดับการอธิบายสิ่งที่เพิ่มเหล่านั้น

4.3.2.2 โปรแกรม พร็อพเพอร์ตี้ (Program Properties)

นอกจากอินเทนด์เดดโมเดล โปรแกรมพร็อพเพอร์ตี้สามารถนำไปใช้ในการสำรวจความผิดพลาดได้ มันจำเป็นสำหรับการรวมข้อบังคับของโปรแกรมซึ่งอยู่ในรูปประโยคปิด A $\longrightarrow$ F เมื่อ A เป็นอะตอมและ F เป็นฟอร์มูล่า ให้กราวด์อะตอม q=Aθ เพื่อทำการทดสอบ ออราเคิลจะลองพิสูจน์ Fθ จากแอ็กเซียมของพร็อพเพอร์ตี้ ถ้า Fθ ไม่สามารถพิสูจน์ได้แล้วแสดงว่า q ล้มเหลว อย่างไรก็ตาม ถ้า Fθ สามารถพิสูจน์ได้ แล้ว ผลจากการทดสอบจะไม่ได้ความรู้จาก q เซตของแอ็กเซียมถูกใช้นิยามพร็อพเพอร์ตี้ ถูกสมมติว่าถูกต้องและสมบูรณ์ เช่น ปราศจากความผิดพลาดเป็นต้น

ตัวอย่างของพร็อพเพอร์ตี้ของการ append(X,Y,Z) เป็นดังนี้

```
append(X,Y,Z) →
      length(X,J) ∧ length(Y,K) ∧
      length(Z,L) ∧ plus(J,K,L).

length([],0).
length([X,Y],N) ← length(Y,M) ∧ plus(M,1,N).
```

จากโปรแกรมดังกล่าวจะเห็นว่าการนำลิสของ X มาต่อกับลิสของ Y ได้เป็นลิสของ Z ซึ่งความยาวของ Z ได้จากการนำความยาวของลิส X มาบวกกับความยาวของลิส Y โดยหากออราเคิลมีพร็อพเพอร์ตี้ดังกล่าวแล้ว ออราเคิลจะไม่ถามคำถามบางคำถามที่ขัดแย้งกับข้อบังคับนี้เช่นหากออราเคิลพบว่าการต่อลิสของ X กับ Y ที่ได้ผลลัพธ์เป็น Z แล้วความยาวของ Z ได้ไม่เท่ากับความยาวของ X รวมกับ Y ออราเคิลจะถือว่าคลอสดังกล่าวผิดโดยไม่ทำการถามผู้ใช้เลย

4.3.2.3 โปรแกรมสเปกซิฟิเคชัน (Program Specifications)

โปรแกรมพร็อพเพอร์ตี้เป็นการให้ข้อมูลบางส่วนในอะตอมที่ถูกทดสอบ เปรียบเสมือนการรวมข้อบังคับของโปรแกรม มันสามารถปฏิเสธอะตอมได้แต่ไม่สามารถพิสูจน์มันได้ ตรงกันข้ามโปรแกรมสเปกซิฟิเคชันสามารถบอกได้มากกว่า สเปกซิฟิเคชันเป็นประโยคปิดที่อยู่ในรูป A$\leftrightarrow$F เมื่อ A เป็นอะตอม และ F ฟอร์มูล่า เมื่อให้กราวด์อะตอม q=Aθ เพื่อทำการทดสอบ ออราเคิลจะลองพิสูจน์ Fθ จากแอ็กเซียมของสเปกซิฟิเคชัน ถ้า Fθ ไม่สามารถพิสูจน์ได้แล้วแสดงว่า q ล้มเหลว อย่างไรก็ตาม ถ้า Fθ สามารถพิสูจน์ได้แล้ว แสดงว่า q เป็นจริง

ตัวอย่างต่อไปนี้เป็นสเปกซิฟิเคชั่นของการเรียงลำดับตัวเลข

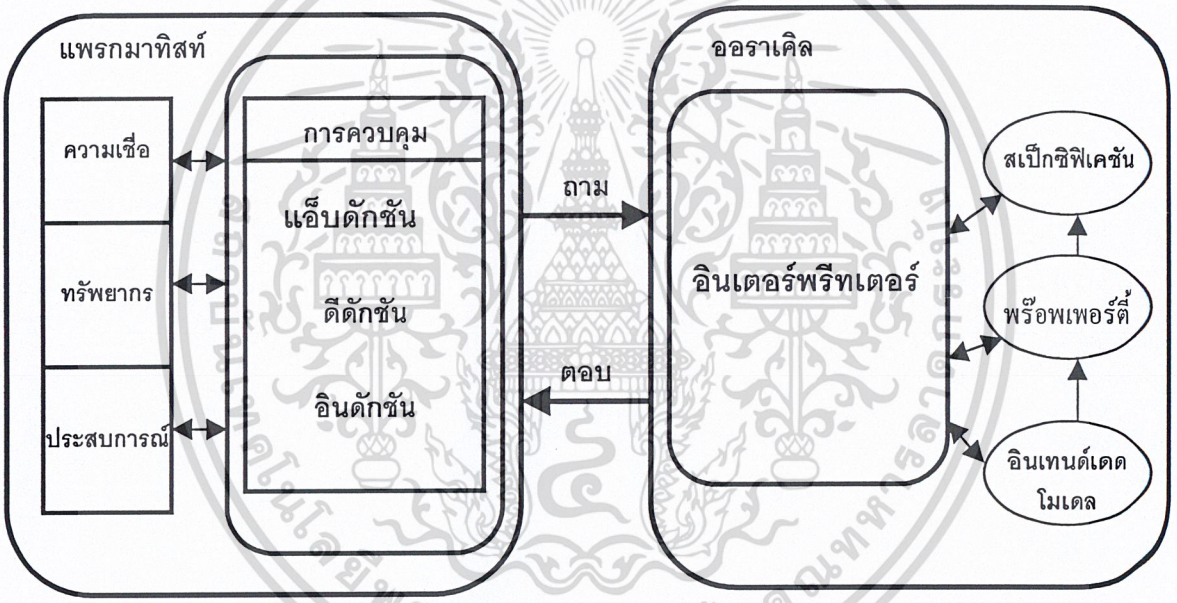
```
sort(X,Y) ↔ permutation(X,Y) ∧ ordered(Y)
```

สเปกซิฟิเคชั่นด้านบนสามารถอธิบายได้ว่า การเรียงลำดับตัวเลขของ X จะได้ผลลัพธ์เป็น Y ก็ต่อเมื่อ X เป็น permutation หนึ่งของ Y และ Y เป็นลิสต์ของตัวเลขที่เรียงแล้ว

4.3.3 การสื่อสารระหว่างแพรกมาทิสท์กับออราเคิล

ดังที่ได้กล่าวไปแล้วว่า แพรกมาทิสท์นั้นจะทำการดีบักโดยหากพบข้อสงสัยในคลอสหนึ่งคลอสใดจะทำการถามไปยังออราเคิลเอเจนต์ ซึ่งการทดสอบผลลัพธ์ของคำตอบที่ได้จากการทำงาน จำเป็นต้องใช้การสื่อสารระหว่างแพรกมาทิสท์และออราเคิล ซึ่งผลของการตรวจสอบควรเป็นประสบการณ์ที่แพรกมาทิสท์ได้รับ และ สามารถเก็บประสบการณ์เหล่านั้นได้ ขั้นตอนของการสื่อสารภายในออราเคิล ออราเคิลจะทำการดูในส่วนของสะเปกซิฟิเคชั่น แล้วจึงดูส่วนของพร๊อพเพอร์ตี้ สุดท้ายจึงทำการดูที่อินเทนด์เดดโมเดล หากดูทั้งสามส่วนดังกล่าวแล้วไม่พบสิ่งที่ต้องการรู้ ออราเคิลก็จะให้ผู้ใช้ทำการตอบคำถามนั้น ในส่วนของแพรกมาทิสท์นั้น ภายในแพรกมาทิสท์เองจะมีการคิดตามวิธีการเชิงวิทยาศาสตร์คือการแอ็บดักชัน ดีดักชัน และ อินดักชัน จากความเชื่อ ทรัพยากร และ ประสบการณ์ ของมันที่มีอยู่

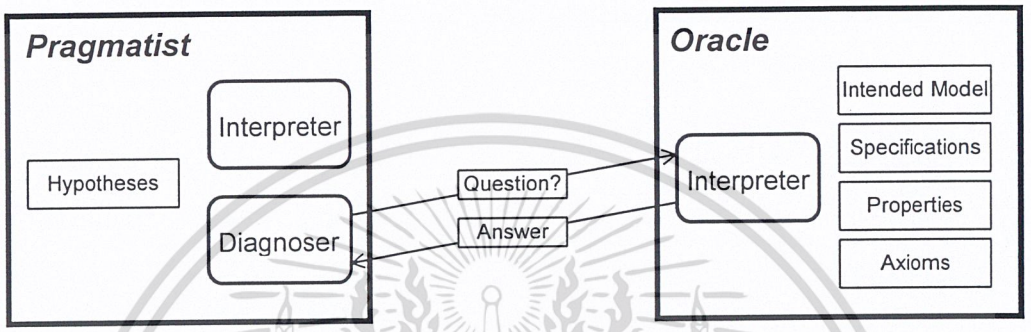
สามารถแสดงการสื่อสารระหว่างเอเจนต์ได้ดังต่อไปนี้



รูปที่ 4-3 การสื่อสารระหว่างแพรกมาทิสท์ กับ ออราเคิล

บทที่ 5
รายละเอียดของระบบดีบักเกอร์

5.1 โครงสร้างของระบบดีบักเกอร์



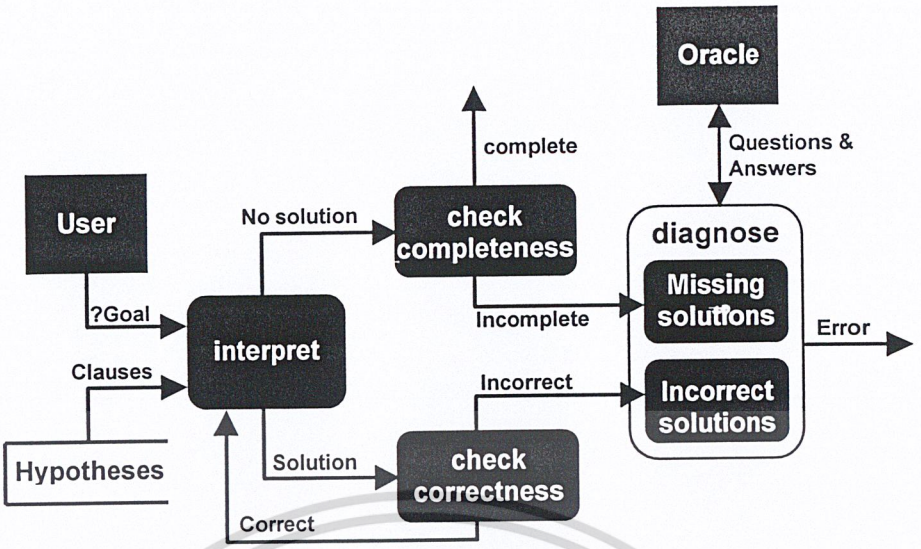
รูปที่ 5-1 โครงสร้างของระบบดีบักเกอร์

ระบบดีบักเกอร์ประกอบด้วยเอเจนต์ 2 เอเจนต์ที่มีการติดต่อกันดังรูปที่ 5-1 แพรกมาทิสท์เอเจนต์ (Pragmatist Agent) ทำหน้าที่ตรวจสอบหาข้อผิดพลาดของสมมติฐาน (Hypotheses) ซึ่งก็คือโปรแกรมที่จะทำการดีบักนั่นเอง โดยในการวินิจฉัยข้อผิดพลาด แพรกมาทิสท์จะมีการถามคำถาม (Querying) ไปยังออราเคิลเอเจนต์ (Oracle Agent) ออราเคิลจะตอบคำถามโดยพิจารณาจาก Specification, Properties และ Intended Model ของโปรแกรม

ในระบบดีบักเกอร์ เอเจนต์ทั้งสองไม่สามารถทำงานอย่างอัตโนมัติทั้งหมด ในแพรกมาทิสท์เอเจนต์ผู้ใช้จะเป็นผู้กำหนดข้อมูลทดสอบ (Test data) สำหรับการทดสอบโปรแกรม เนื่องจากแพรกมาทิสท์ยังไม่มีตัวสร้างข้อมูลทดสอบ (Test data generator) ในออราเคิลเอเจนต์ หากคำตอบของคำถามไม่สามารถหาได้จาก Specification, Properties และ Intended Model ที่เก็บอยู่ในเครื่อง ก็ต้องเป็นหน้าที่ของผู้ใช้ในการตอบคำถาม นั่นคือใช้ Intended Model ที่อยู่ในความคิดของผู้ใช้

5.2 แพรกมาทิสท์เอเจนต์ (Pragmatist Agent)

รูปที่ 5-2 แสดง Data Flow Diagram ของการทำงานภายในแพรกมาทิสท์เอเจนต์ ซึ่งสามารถสรุปได้ดังนี้ ขั้นแรกผู้ใช้จะกำหนดโกล (Goal) ที่ใช้ในการทดสอบโปรแกรม อินเตอร์พรีทเตอร์จะทำการอินเตอร์พรีทโกลกับสมมติฐาน(ซึ่งก็คือโปรแกรมที่จะทำการดีบัก) เมื่อได้ผลลัพธ์ผู้ใช้ก็จะทำการตรวจสอบหากผลลัพธ์ถูกต้องจะกลับไปอินเตอร์พรีทหาผลลัพธ์อื่นต่อ แต่หากผลลัพธ์ผิดจึงเข้าสู่ขั้นตอนการวินิจฉัยข้อผิดพลาด ในกรณีที่อินเตอร์พรีทเตอร์ไม่สามารถหาผลลัพธ์ได้อีกแล้ว ผู้ใช้จะทำการตรวจสอบว่าผลลัพธ์ที่ได้ครบถ้วนแล้วหรือไม่ ถ้ามีผลลัพธ์ใดหายไป ตัววินิจฉัยจะหาข้อผิดพลาดภายในสมมติฐาน



รูปที่ 5-2 Data Flow Diagram ของการทำงานภายในแพรกมาทิสท์เอเจนต์

ความรู้หรือข้อมูลที่แพรกมาทิสท์มีคือ สมมติฐาน(Hypotheses) สมมติฐานของแพรกมาทิสท์ก็คือโปรแกรมที่จะทำการดีบัก โปรแกรมคือเซตของคลอส(Clause) แต่ละคลอสจะถูกเก็บเป็นข้อมูลอยู่ในรูป hyp(X) เมื่อ X คือคลอส ตัวอย่างเช่นโปรแกรมที่ 5-1 เป็นสมมติฐานของโปรแกรม Subset

```
hyp( subset([],_) ).
hyp( (subset([X|Xs],Ys):-member(X,Ys),subset(Xs,Ys)) ).
hyp( member(X,[X|_]) ).
hyp( (member(X,[_|Xs]):-member(X,Xs)) ).
```

โปรแกรมที่ 5-1 ตัวอย่างของสมมติฐาน

ตัวประมวลผลของออราเคิลประกอบด้วย อินเตอร์พรีทเตอร์(Interpreter) และตัววินิจฉัยข้อผิดพลาด (Diagnoser) ต่อไปจะอธิบายรายละเอียดของส่วนประกอบเหล่านี้

- อินเตอร์พรีทเตอร์ (Interpreter)

หน้าที่ของอินเตอร์พรีทเตอร์คือ ใช้อินเตอร์พรีทสมมติฐานหาผลลัพธ์เพื่อให้ผู้ใช้ตรวจสอบว่าถูกต้องหรือไม่ โดยระหว่างการอินเตอร์พรีท อินเตอร์พรีทเตอร์จะเก็บคอมพิวเตชันทรี (Computation Tree) ไว้ด้วย คอมพิวเตชันทรีจะเป็นข้อมูลที่ใช้ในการหาข้อผิดพลาดโดยตัววินิจฉัย โปรแกรมที่ 5-2 แสดงโค้ดโปรแกรมของอินเตอร์พรีทเตอร์ที่สามารถเก็บคอมพิวเตชันทรีได้

```
%demo(pragmatist,+Goal,-TreeList)
demo(pragmatist,A,[true(A,[])]) :-
      builtin(A),!,A.
demo(pragmatist,(A,B),Ts) :- !,
      demo(pragmatist,A,As),
      demo(pragmatist,B,Bs),
      append(As,Bs,Ts).
```

```
demo(pragmatist,A,[true(A,Ts)]) :-
      hyp(C),
      (C=(A:-B)->demo(pragmatist,B,Ts);A=C,Ts=[]).
```

โปรแกรมที่ 5-2 อินเตอร์พรีทเตอร์ที่สามารถเก็บคอมพิวเตชันทรีได้

โปรแกรมที่จะทำการดีบักอาจมีการเรียกใช้เพรดดิเคทภายใน (built-in predicates) ซึ่งเป็นเพรดดิเคทที่ถูกนิยามไว้แล้วภายในโปรล็อก ตัวอย่างเช่น <, =, is หรือ fail เป็นต้น อินเตอร์พรีทเตอร์จะต้องรู้ว่าเพรดดิเคทใดเป็นเพรดดิเคทภายใน เพื่อจะแยกจัดการกับเพรดดิเคทเหล่านี้ วิธีง่ายๆ วิธีหนึ่งก็คือส่งให้อินเตอร์พรีทเตอร์ของโปรล็อกทำการอินเตอร์พรีทให้ ซึ่งทำให้ไม่ต้องกำหนดนิยามของเพรดดิเคทภายในให้กับอินเตอร์พรีทเตอร์ที่สร้างขึ้น ดังนั้นอินเตอร์พรีทเตอร์จะต้องเก็บรายชื่อของเพรดดิเคทเหล่านี้ ตัวอย่างเช่นโปรแกรมที่ 5-3 ซึ่งเก็บอยู่ในรูป builtin(X)

```
builtin(true).          builtin(fail).
builtin(_ = _).         builtin(_ \= _).
builtin(_ < _).         builtin(_ =< _).
builtin(_ > _).         builtin(_ >= _).
builtin(_ is _).
```

โปรแกรมที่ 5-3 ตัวอย่างการเก็บเพรดดิเคทภายใน (Built-in Predicates)

เมื่อทำการเรียกใช้อินเตอร์พรีทเตอร์กับโกล p1, p2, ..., pn จะได้เป็นลิสต์ [t1, t2, ..., tn] เมื่อ ti เป็นคอมพิวเตชันทรีของ pi โดย ti = true(pi, Cs) เมื่อ Cs เป็นลิสต์ของคอมพิวเตชันทรีของโหนดลูก ตัวอย่างการใช้อินเตอร์พรีทเตอร์เป็นดังนี้

```
| ?- demo(pragmatist,subset([2],[1,2]),Ts).
Ts = [true(subset([2],[1,2]),[true(member(2,[1,2]),[true
(member(2,[2]),[])]),true(subset([],[1,2]),[])])]
```

อินเตอร์พรีทเตอร์ที่สร้างขึ้นจะสำเร็จ (Success) ก็ต่อเมื่อโกลสามารถอินเตอร์พรีทจากสมมติฐานได้ แต่มีข้อยกเว้นคือในกรณีที่เกิดการประมวลผลที่ไม่สิ้นสุด(Infinite Computation) อินเตอร์พรีทเตอร์จะไม่สามารถหาคำตอบได้ วิธีการหนึ่งสำหรับแก้ปัญหานี้คือ การกำหนดระดับสแต็ก(Stack) สูงสุด เมื่อการประมวลผลใช้สแต็กเกินกว่าระดับที่กำหนด อินเตอร์พรีทเตอร์จะหยุดการทำงาน และให้คอมพิวเตชันทรีที่เก็บได้จนกระทั่งเกิด Stack Overflow โปรแกรมที่ 5-4 แสดงโค้ดของอินเตอร์พรีทเตอร์ที่สามารถกำหนดระดับของสแต็กได้

```
%demo_tree(pragmatist,+Goal,-TreeList,+Depth,-OverflowFlag)
demo_tree(pragmatist,A,[overflow],0,overflow) :- !.
demo_tree(pragmatist,A,[true(A,[])],D,yes) :-
   builtin(A),!,A.
demo_tree(pragmatist,(A,B),Ts,D,OV) :- !,
```

```
    demo_tree(pragmatist,A,As,D,OVA),
    (     OVA=overflow
          ->
          OV=OVA,Ts=As
          ;
          demo_tree(pragmatist,B,Bs,D,OV),
          append(As,Bs,Ts)
    ).
demo_tree(pragmatist,A,[true(A,Ts)],D,OV) :-
    hyp(C),
    (     C=(A:-B)
          ->
          D1 is D-1,
          demo_tree(pragmatist,B,Ts,D1,OV)
          ;
          A=C,Ts=[],OV=yes
    ).
```

โปรแกรมที่ 5-4 อินเตอร์พรีทเตอร์ที่สามารถกำหนดระดับของสแต็กได้

- **ตัววินิจฉัยข้อผิดพลาด (Diagnoser)**

แพรกมาทิสท์มีตัววินิจฉัยสำหรับโปรแกรมที่มีความผิดพลาดใน 2 แบบคือ ก) ผลลัพธ์ไม่ถูกต้อง (Incorrect Solutions) ข) ไม่สามารถหาผลลัพธ์ได้ (Finite Failure) ต่อไปจะแสดงรายละเอียดของตัววินิจฉัยข้อผิดพลาดแต่ละแบบ

ก) ตัววินิจฉัยข้อผิดพลาดสำหรับความผิดพลาดแบบผลลัพธ์ไม่ถูกต้อง

การวินิจฉัยความผิดพลาดแบบนี้ ตัววินิจฉัยจะวิเคราะห์จากคอมพิวเตชันทรีของการประมวลผลอินพุตที่ให้ผลลัพธ์ผิดพลาด โดยตัววินิจฉัยจะทำการค้นหา(Search) เข้าไปในคอมพิวเตชันทรีเพื่อหาโหนดที่ผิดพลาด เราพบว่าวิธีการค้นหามีผลต่อจำนวนคำถามที่ต้องถามออราเคิล หากเราต้องการสร้างตัววินิจฉัยข้อผิดพลาดที่มีความอัตโนมัติ ตัววินิจฉัยควรจะถามผู้ใช้ให้น้อยที่สุด (Query Minimization) ปัจจัยที่มีผลต่อจำนวนคำถามไม่ใช้มีเพียงวิธีการค้นหาเท่านั้น หากวิเคราะห์ให้ละเอียดขึ้นจะพบว่า ปัจจัยที่มีผลต่อจำนวนคำถามคือ

1) ความซับซ้อนของการประมวลผล (Computational Complexity)

   ความซับซ้อนของการประมวลผลสามารถสังเกตได้จากความซับซ้อนของคอมพิวเตชันทรี ตัววัดความซับซ้อนที่สำคัญมี 3 ตัวคือ

   - Length : จำนวนของการเรียกโปรซิเยอร์ทั้งหมดในการประมวลผล ซึ่งก็คือจำนวนโหนดในคอมพิวเตชันทรี
   - Depth : ความลึกสูงสุดของการเรียกโปรซิเยอร์ ซึ่งก็คือความสูงของคอมพิวเตชัน
   - Branching : จำนวนการแตกกิ่งสูงสุดของคอมพิวเตชันทรี

2) ตำแหน่งของข้อผิดพลาด (Location of bugs)

3) วิธีการค้นหา (Search Strategies)

ต่อไปจะศึกษาการทำงานและการสร้างตัววินิจฉัยที่มีการค้นหาแบบต่างๆ โดยใช้โปรแกรมที่ 5-5 เป็นตัวอย่าง โปรแกรมนี้เป็นโปรแกรมเรียงตัวเลขแบบ Quick Sort ซึ่งมีข้อผิดพลาดที่คลอส part(V, [P1|Ps],Ls,Gs) ทำให้ผลลัพธ์ของโปรแกรมไม่ถูกต้อง

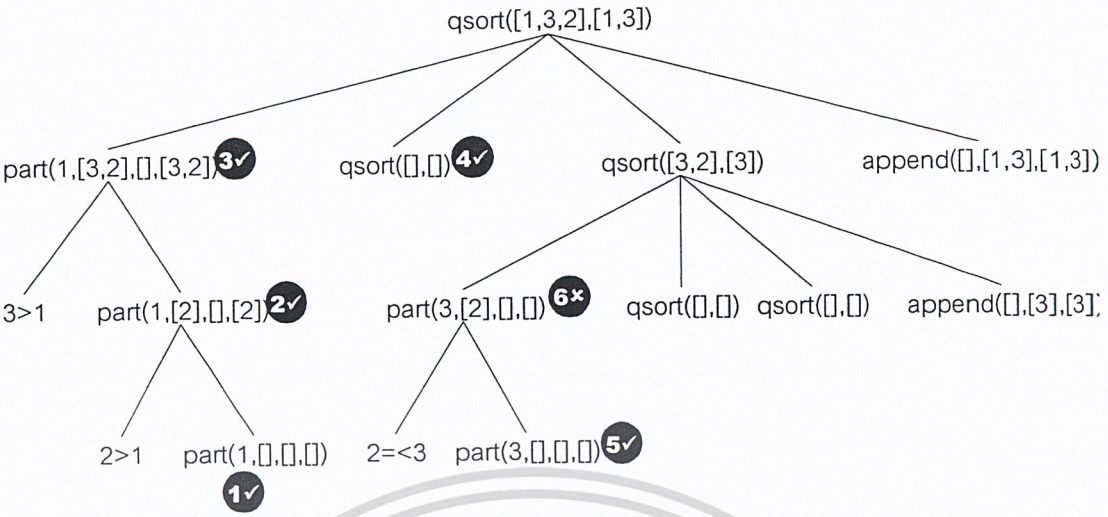
```
qsort([],[]).
qsort([P1|Ps],Qs) :-
      part(P1,Ps,Ls,Gs),
      qsort(Ls,L1s),
      qsort(Gs,G1s),
      append(L1s,[P1|G1s],Qs).
part(V,[],[],[]).
part(V,[P1|Ps],Ls,[P1|Gs]) :-
      P1>V,
      part(V,Ps,Ls,Gs).
part(V,[P1|Ps],Ls,Gs) :-
      P1=<V,
      part(V,Ps,Ls,Gs).
append([],Ls,Ls).
append([P1|Ps],Qs,[P1|Ls]) :-
      append(Ps,Qs,Ls).
```

โปรแกรมที่ 5-5 โปรแกรม Quick Sort ที่มีข้อผิดพลาด

1) การค้นหาแบบ Bottom-Up (Bottom-Up Search Strategy)

วิธีการค้นหาแบบ Bottom-Up อัลกอริทึมจะทำการ post-order traverse เข้าไปในคอมพิวเตชันทรี โดยในแต่ละโหนด <p,x,y> ที่ผ่าน ตัววินิจฉัยจะถามออราเคิลว่าถูกต้องหรือไม่ หากถูกต้องก็ให้ traverse ต่อไป หากผิดสรุปได้ว่าข้อผิดพลาดอยู่ใน p เนื่องจากการ traverse เป็นแบบ post-order โหนดลูกทั้งหมดของ <p,x,y> ได้ผ่านการตรวจสอบแล้วว่าถูกต้อง การที่ <p,x,y> ไม่ถูกต้องแสดงให้เห็นว่านิยามของ p มีข้อผิดพลาด

รูปที่ 5-3 แสดงลำดับการถามคำถามระหว่างการค้นหาแบบ Bottom-Up ในคอมพิวเตชันทรีของการประมวลผล qsort ด้วยอินพุต [1,3,2]



รูปที่ 5-3 ลำดับการถามคำถามของตัววินิจฉัยที่ค้นหาแบบ Bottom-Up

```
%fp(+TreeList,-ErrorClause)
fp([H|Ts],X) :-
      fp(H,X),!;
      fp(Ts,X).
fp(true(A,_),X) :-
      builtin(A),!,fail.
fp(true(A,Bs),X) :-
      fp(Bs,X),!;
      query(forall,A,no),
      treelist_conj(Bs,Body),X=(A:-Body).
```

โปรแกรมที่ 5-6 ตัววินิจฉัยข้อผิดพลาดที่ค้นหาแบบ Bottom-Up

โค้ดของตัววินิจฉัยข้อผิดพลาดแบบนี้แสดงในโปรแกรมที่ 5-6 ตัวอย่างการใช้งานเป็นดังนี้

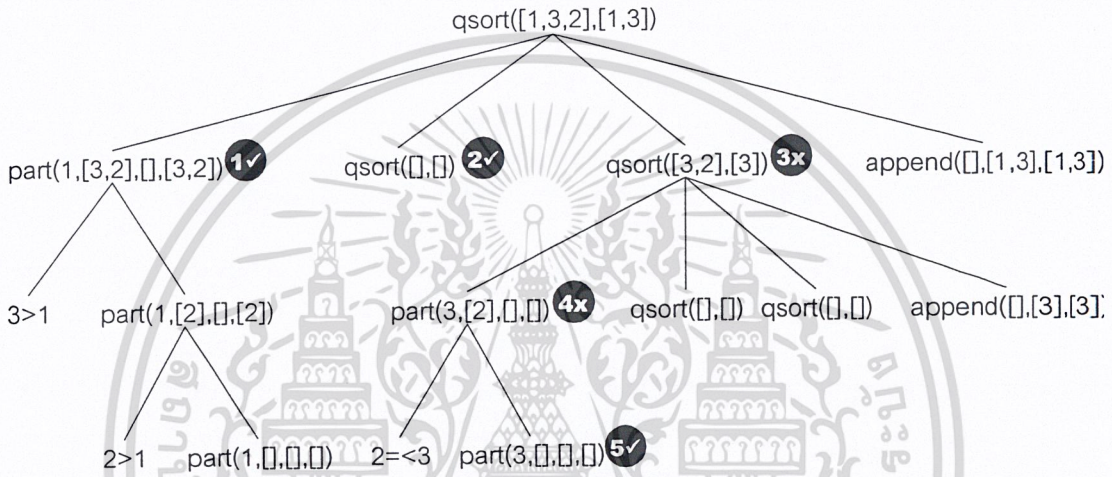
```
| ?- fp(qsort([1,3,2],Ls),Error).

part(1,[],[],[]) Valid? y
part(1,[2],[],[2]) Valid? y
part(1,[3,2],[],[3,2]) Valid? y
qsort([],[]) Valid? y
part(3,[],[],[]) Valid? y
part(3,[2],[],[]) Valid? n

Ls = [1,3]
Error = part(3,[2],[],[]) :- 2=<3, part(3,[],[],[]).
```

Query Complexity จำนวนคำถามของการค้นหาแบบ Bottom-Up ขึ้นอยู่กับความยาวของการประมวลผล (length) และตำแหน่งของบัก กรณี Worst-case เกิดขึ้นเมื่อ ข้อผิดพลาดอยู่ที่โหนดรูท (root node) ซึ่งทำให้จำนวนการถามคำถามเป็น n-1 เมื่อ n คือความยาวของการประมวลผล

2) การ ค้นหา แบบ Top-Down (Top-Down Search Strategy)

ตรงกันข้ามกับแบบ Bottom-Up การค้นหาแบบ Top-Down จะเริ่มจาก root node โดยทำการถามโหนดลูกของ root node เพื่อหาว่าโหนดใดไม่ถูกต้อง เช่นหากพบว่าโหนดลูก <p,x,y> ไม่ถูกต้อง ตัววินิจฉัยจะทำการ recursive ไปยัง sub-tree ที่มี <p,x,y> เป็น root node แต่หากไม่พบว่าโหนดใดผิดพลาดก็แสดงว่าโปรซิเยอร์ที่ root node มีข้อผิดพลาด

รูปที่ 5-4 แสดงลำดับการถามคำถามระหว่างการค้นหาแบบ Top-Down ในคอมพิวเตชันทรีของการประมวล qsort ด้วยอินพุต [1,3,2]



รูปที่ 5-4 ลำดับการถามคำถามของตัววินิจฉัยที่ค้นหาแบบ Top-Down

```
%fp(+TreeList,-ErrorClause)
fp([T1|Ts],C) :-
      fp(T1,C),!;
      fp(Ts,C).
fp(true(A,_),C) :-
      builtin(A),!,fail.
fp(true(A,Bs),C) :-
      query(forall,A,no),
      (
      fp(Bs,C),!;
      treelist_conj(Bs,Body),C=(A:-Body)
      ).
```

โปรแกรมที่ 5-7 ตัววินิจฉัยข้อผิดพลาดที่ค้นหาแบบ Top-Down

โค้ดของตัววินิจฉัยข้อผิดพลาดแบบนี้แสดงในโปรแกรมที่ 5-7 ตัวอย่างการใช้งานเป็นดังนี้

```
| ?- fp(qsort([1,3,2],Ls),Error).

part(1,[3,2],[],[3,2]) Valid? y
qsort([],[]) Valid? y
qsort([3,2],[3]) Valid? n
```

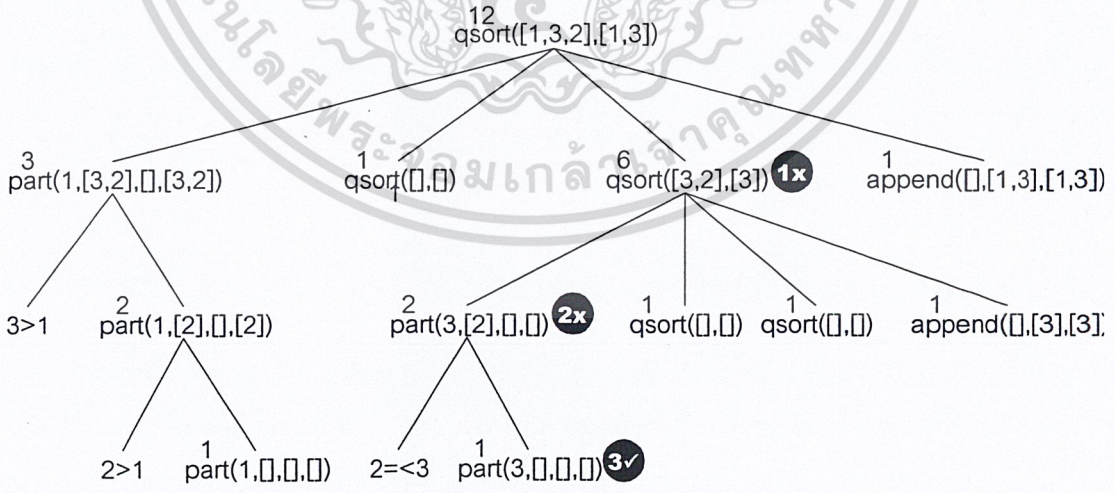
```
part(3,[2],[],[]) Valid? n
part(3,[],[],[]) Valid? y

Ls = [1,3]
Error = part(3,[2],[],[]) :- 2=<3, part(3,[],[],[]).
```

Query Complexity จำนวนคำถามของการค้นหาแบบ Top-Down ขึ้นอยู่กับลักษณะของคอมพิวเตชันทรี และตำแหน่งของบัก ถ้าทรีมีความลึก b และ Branching d แล้วในกรณี Worst-case จำนวนคำถามสูงสุดจะเป็น b*(d-1) ซึ่งจะเกิดขึ้นเมื่อข้อผิดพลาดอยู่ลึกสุด ถ้าทรีมีความสมดุลค่า d จะใกล้เคียง $\log_b n$ เมื่อ n คือความยาวของการประมวลผล ทำให้วิธีนี้ถามน้อยกว่าวิธี Bottom-Up แต่ถ้าทรีมีลักษณะเป็นเส้น จำนวนคำถามอาจจะมากถึง n

3) การค้นหาแบบ Divide-and-Query (Divide-and-Query Search Strategy)

แนวคิดคือ จะเลือกถามโหนดที่เมื่อถามแล้วจะตัดทรีออกไปได้ครึ่งหนึ่ง แล้วทำซ้ำจนกว่าจะเหลือเพียงโหนดเดียวซึ่งโหนดนั้นผิดพลาด วิธีการคือ ขั้นแรกจะเลือกโหนด M ที่มีน้ำหนัก(weight) เท่ากับหรือใกล้เคียงที่สุดกับ ½ ของน้ำหนักของ root node (น้ำหนักของโหนดใดๆ คือจำนวนโหนดที่อยู่ใน sub-tree ที่โหนดนั้น) แล้วทำการถามผู้ใช้ว่าโหนด M ถูกต้องหรือไม่ ถ้าผิดแสดงว่าความผิดพลาดอยู่ใน sub-tree ที่โหนดนั้น จึงให้ recursive ไปใน sub-tree แต่ถ้า M ถูกต้อง แสดงว่าไม่มีความผิดพลาดอยู่ จึงตัด sub-tree นั้นออกไปได้ แล้วทำซ้ำกับทรีที่เหลือ

รูปที่ 5-5 แสดงลำดับการถามคำถามระหว่างการค้นหาแบบ Divide-and-query ในคอมพิวเตชันทรีของการประมวล qsort ด้วยอินพุต [1,3,2]



รูปที่ 5-5 ลำดับการถามคำถามของตัววินิจฉัยที่ค้นหาแบบ Divide-and-Query

```
fpm(V,Ts,weight(WA,WBs),Ts,weight(WA,WBs),[],[]) :-
      WA=<V,!.
fpm(V,true(A,Bs),weight(WA,WBs),M,WM,[true(A,R1s)],[weight
(WA1,WR1s)]) :-
```

```
      fpm(V,Bs,WBs,M,WM,R1s,WR1s),
      WM=weight(WM1,WMs),
      WA1 is WA-WM1.
fpm(V,[],[],[],weight(0,[]),[],[]).
fpm(V,[H|Ts],[WH|WTs],M,WM,Rs,WRs) :-
      fpm(V,H,WH,MH,weight(WMH1,WMHs),RHs,WRHs),
      fpm(V,Ts,WTs,MT,weight(WMT1,WMTs),RTs,WRTs),
      (     WMH1>=WMT1
      ->    M=MH,WM=weight(WMH1,WMHs),
            append(RHs,Ts,Rs),append(WRHs,WTs,WRs)
      ;     M=MT,WM=weight(WMT1,WMTs),
            Rs=[H|RTs],WRs=[WH|WRTs]
      ).
%fp(+TreeList,+WeightTreeList,-Error)
fp([true(A,_)],[weight(1,_)],A) :- !.
fp(T,[weight(WA,WBs)],X) :-
      V is int((WA+1)/2),
      fpm(V,T,[weight(WA,WBs)],M,WM,R,WR),
      M=true(A,Bs),
      query(forall,A,YN),
      (YN=no->fp([M],[WM],X);fp1(R,WR,X)).
%fp(+TreeList,-Error)
fp(T,X) :-
      tree_weight(T,WT,_),
      fp(T,WT,X).
%tree_weight(+TreeList,-WeightTreeList,-Weight)
tree_weight([],[],0).
tree_weight([T1|Ts],[WT1|WTs],V) :-
      tree_weight(T1,WT1,V1),
      tree_weight(Ts,WTs,Vs),
      V is V1+Vs.
tree_weight(true(A,_),weight(0,[]),0) :-
      builtin(A),!.
tree_weight(true(A,Bs),weight(WA,WBs),WA) :-
      tree_weight(Bs,WBs,VB),
      WA is VB+1.
```

โปรแกรมที่ 5-8 ตัววินิจฉัยข้อผิดพลาดที่ค้นหาแบบ Divide-and-Query

โค้ดของตัววินิจฉัยข้อผิดพลาดแบบนี้แสดงในโปรแกรมที่ 5-8 ตัวอย่างการใช้งานเป็นดังนี้

```
| ?- fp(qsort([1,3,2],Ls),Error).

qsort([3,2],[3]) Valid? n
part(3,[2],[],[]) Valid? n
part(3,[],[],[]) Valid? y

Ls = [1,3]
Error = part(3,[2],[],[]) :- 2=<3, part(3,[],[],[]).
```

Query Complexity สำหรับ worst-case วิธีนี้จะดีกว่าทั้งสองวิธีแรก ถ้าทรีสมดุลจำนวนคำถามจะพอๆ กับวิธี Top-Down ค่า branching มีผลต่อจำนวนคำถาม เพราะถ้ามีค่ามากจะทำให้การแบ่งทรีไม่ครึ่งจริง Query Complexity ของวิธีคือ O(b*log n)

ข) ตัววินิจฉัยข้อผิดพลาดสำหรับความผิดพลาดแบบไม่สามารถหาผลลัพธ์ได้

การสร้างตัววินิจฉัยสำหรับความผิดพลาดแบบนี้ยากกว่าความผิดพลาดแบบผลลัพธ์ไม่ถูกต้อง ความผิดพลาดแบบนี้สังเกตได้จากการที่อินเตอร์พรีทเตอร์ไม่สามารถหาผลลัพธ์ได้ หรือหาผลลัพธ์ได้ไม่ครบในโปรแกรมที่ไม่มีการใช้ not สาเหตุของความผิดพลาดเกิดจากการที่โปรแกรมขาดคลอสบางคลอสซึ่งจะต้องใช้ในการอินเตอร์พรีทผลลัพธ์ที่ควรจะได้ พิจารณาตัวอย่าง

```
clever(X) :- like(X,Y), good(Y).
like(john,mary).
like(john,betty).
good(lucy).
```

อินเตอร์พรีทเตอร์ไม่สามารถหาผลลัพธ์ของโกล clever(john) ได้ สาเหตุอาจเกิดจากนิยามของ like (john,X) หรือ good(Y) ไม่ครบ หรืออาจจะเกิดจากนิยามของ clever(john,X) ไม่สมบูรณ์ก็เป็นได้ ทั้งนี้ขึ้นอยู่กับความหมายของโปรแกรมในความคิดของผู้เขียนโปรแกรม

การที่คลอสที่จำเป็นต้องใช้หาไม่พบ อาจเกิดจากคลอสนิยามไว้ไม่ถูกต้อง ตัวอย่างเช่น

```
difference(X,Y,Z) :- X>=Y, Z is X-Y.
difference(X,Y,Z) :- Y<X, Z is Y-X.
```

โกล difference(1,2,Z) ไม่สามารถอินเตอร์พรีทได้ จากโปรแกรมสามารถสรุปได้ว่า difference(1,2,Z) ไม่ได้ถูกนิยามไว้ แต่เมื่อโปรแกรมเมอร์วิเคราะห์แล้ว จะพบว่าคลอสที่ 2 นิยามไว้ผิดคือ Y<X ควรจะเป็น X<Y

```
%ip(+Goal,-Error)
ip(A,C) :-
      builtin(A),!,fail.
ip((A,B),X) :- !,
      (demo(pragmatist,A),!,ip(B,X);
      ip(A,X)).
ip(A,X) :-
      hyp((A:-B)),
      query(exist,B,YN),YN=yes,!,
      ip(B,X);
      X=A.
```

โปรแกรมที่ 5-9 ตัววินิจฉัยสำหรับความผิดพลาดแบบหาผลลัพธ์ไม่ได้

```
%ip(+Goal,-Error)
ip(A,C) :-
      builtin(A),!,fail.
ip((A,B),C) :- !,
```

```
        (     demo(pragmatist,A),
              ip(B,C),!
        ;
              ip(A,C)
        ).
ip(A,C) :-
        findall(A,demo(pragmatist,A),As),
        freeze(A,A1),not_exist(member(A1,As)),
        query(comp,(A,As),no),
        (     hyp((A:-B)),
              ip(B,C),!
        ;
              C=A
        ).
```

โปรแกรมที่ 5-10 ตัววินิจฉัยสำหรับความผิดพลาดแบบหาผลลัพธ์ไม่ได้ (ปรับปรุง)

```
clever(X) :- like(X,Y),good(Y).
like(X,Y) :- person(Y),cute(Y).
person(john).    person(mary).
person(betty).   person(spook).
cute(mary).      cute(betty).
cute(lucy).
good(lucy).
```

โปรแกรมที่ 5-11 ตัวอย่างโปรแกรมที่ไม่สมบูรณ์

โปรแกรมที่ 5-9 แสดงตัววินิจฉัยที่สามารถหาคลอสที่ขาดหายได้ ตัววินิจฉัยนี้จะมีการถามคำถามประเภท Existential Questions คือจะถามว่าโกลควรจะมีผลลัพธ์หรือไม่ ถ้ามีให้ป้อนผลลัพธ์หนึ่งกับตัววินิจฉัย(โดยการทำให้โกลนั้นเป็นกราวด์) ตัวอย่างการใช้งานตัววินิจฉัยกับโปรแกรมที่ 5-11 เป็นดังนี้

```
| ?- ip(clever(john),Err).

Enter a true ground instance of like(john,A),good(A)
if there is such, or "no" otherwise
|: like(john,lucy),good(lucy).

Enter a true ground instance of person(lucy),cute(lucy)
if there is such, or "no" otherwise
|: person(lucy),cute(lucy).

Err = person(lucy)
```

ความยุ่งยากของตัววินิจฉัยนี้คือ การที่ต้องบอกผลลัพธ์ให้ตัววินิจฉัย ซึ่งต่างจากตัววินิจฉัยสำหรับความผิดพลาดแบบผลลัพธ์ไม่ถูกต้องซึ่งถามเพียงคำถามประเภทใช่/ไม่ใช่ วิธีการหนึ่งในการปรับปรุงให้ตัววินิจฉัยถามคำถามใช้/ไม่ใช้แสดงในโปรแกรมที่ 5-10 ตัววินิจฉัยนี้จะถามคำถามประเภท Completeness Questions คือตัววินิจฉัยจะไปหาคำตอบทั้งหมดที่สามารถจะหาได้จากโปรแกรม แล้วถามออราเคิลว่าผลลัพธ์ที่ได้ครบถ้วนหรือไม่ ออราเคิลเพียงตอบว่าครบหรือไม่ครบเท่านั้น ตัวอย่างเช่น

```
| ?- ip(clever(john),Err).

All true ground instances of clever(john)
are in [] Complete? n
All true ground instances of good(mary)
are in [] Complete? y
All true ground instances of good(betty)
are in [] Complete? y
All true ground instances of like(john,_)
are in [like(john,mary),like(john,betty)] Complete? n
All true ground instances of cute(john)
are in [] Complete? y
All true ground instances of cute(spook)
are in [] Complete? y
All true ground instances of person(_)
are  in  [person(john),person(mary),person(betty),person(spook)]
Complete? n

Err = person(_)
```

5.3 ออราเคิลเอเจนต์ (Oracle Agent)

ความรู้ของออราเคิลประกอบไปด้วย intended model, specifications, properties และ axioms ความรู้ในส่วนของ specifications, properties และ axioms จะเก็บในฐานข้อมูลด้วยเพรดดิเคท specification, property และ axiom ตามลำดับ โปรแกรมที่ 5-12 แสดงตัวอย่าง แต่ในส่วน intended model ไม่สามารถเก็บในฐานข้อมูลได้ทั้งหมด เพราะ intended model มีขนาดมหาศาล เนื่องจากเราถือว่าผู้ใช้มี intended model ของโปรแกรมอยู่ ดังนั้นเมื่อระบบมีความต้องการใช้ intended model ระบบจะถามไปยังผู้ใช้ ในขณะนั่นผู้ใช้ก็ทำหน้าที่เป็นออราเคิลด้วย แต่เพื่อไม่ให้ถามข้อมูล intended model กับผู้ใช้ซ้ำ ระบบจะเก็บข้อมูลของ intended model ที่ถามได้จากผู้ใช้ในฐานข้อมูลภายใต้เพรดดิเคท intmodel ดังตัวอย่างในโปรแกรมที่ 5-13

```
specification( (sort([A],Bs)<=>Bs=[A]) ).
property( (sort(Xs,Ys)=>length(Xs,N),length(Ys,N)) ).
property( (insert(X,Xs,Ys)=>length(Xs,N),length(Ys,s(N))) ).
axiom( (length([],0) :- true) ).
axiom( (length([H|Ts],s(N1)) :- length(Ts,N1)) ).
```

โปรแกรมที่ 5-12 ตัวอย่างการเก็บ Specifications, Properties และ Axioms

```
intmodel(sort([],[]) - yes).
intmodel(insert(1,[3],[3]) - no).
intmodel(sort([1,4],[1,4]) - yes).
intmodel(insert(2,[],[]) - no).
intmodel(insert(1,[],[1]) - yes).
```

โปรแกรมที่ 5-13 ตัวอย่างการเก็บ Intended model ที่ถามจากผู้ใช้

ตัวประมวลผลของออราเคิลมีเพียงตัวเดียวคืออินเตอร์พรีทเตอร์ อินเตอร์พรีทเตอร์ของออราเคิลใช้ในการอินเตอร์พรีทหาคำตอบให้กับคำถามของแพรกมาทิสท์ คำถามที่แพรกมาทิสท์ถามมี 3 ประเภทคือ Universal Questions, Existential Questions และ Completeness Questions อินเตอร์พรีทเตอร์จะใช้ความรู้ของออราเคิลในการตอบคำถาม โดยจะพยายามหาคำตอบโดยใช้ specifications และ properties ก่อน หากไม่สามารถใช้ได้จึงใช้ข้อมูล intended model ที่เก็บในฐานข้อมูล แต่หากไม่มีอินเตอร์พรีทเตอร์จะส่งคำถามไปให้ผู้ใช้ตอบ ในระบบดีบักเกอร์ที่สร้างขึ้นอินเตอร์พรีทเตอร์ยังมีความสามารถจำกัด โดยอินเตอร์พรีทเตอร์สามารถใช้ความรู้ในฐานข้อมูลในการหาคำตอบให้กับคำถามประเภท Universal Questions และ Existential Questions เท่านั้น และยังมีข้อแม้ว่าคำถามต้องไม่ติดตัวแปร(นั่นคือเป็นกราวด์) คำถามประเภท Completeness Questions และคำถามที่ไม่กราวด์จะถูกส่งไปให้ผู้ใช้ตอบแทน โปรแกรมที่ 5-14 แสดงอินเตอร์พรีทเตอร์ของออราเคิล

```
demo(oracle,exist,A,YN,Source) :-
      ground(A),!,
      (
      demo(specification,A,YNSpec)->YN=YNSpec,Source=specification;
      demo(property,A,YNProp)->YN=YNProp,Source=property;
      demo(intmodel,A,YNInt)->YN=YNInt,Source=intmodel;
      demo(user,exist,A,YNUser)->YN=YNUser,Source=user
      ).
demo(oracle,exist,A,YN,user) :-
      demo(user,exist,A,YN).

demo(oracle,forall,A,YN,Source) :-
      ground(A),!,
      (
      demo(specification,A,YNSpec)->YN=YNSpec,Source=specification;
      demo(property,A,YNProp)->YN=YNProp,Source=property;
      demo(intmodel,A,YNInt)->YN=YNInt,Source=intmodel;
      demo(user,forall,A,YNUser)->YN=YNUser,Source=user
      ).
demo(oracle,forall,A,YN,user) :-
      demo(user,forall,A,YN).

demo(oracle,comp,(A,As),YN,user) :-
      demo(user,comp,(A,As),YN).

demo(specification,A,YN) :-
      specification( (A<=>S) ),
      demo(axiom,S,YN).
demo(property,A,no) :-
      property( (A=>F) ),
      demo(axiom,F,no).
demo(intmodel,A,YN) :-
      intmodel(A-YN).

demo(axiom,A,YN) :-
      demo(axiom,A)->YN=yes;YN=no.
demo(axiom,true).
demo(axiom,(A,B)) :- !,
      demo(axiom,A),
      demo(axiom,B).
demo(axiom,A) :-
```

```
      builtin(A),!,A.
demo(axiom,A) :-
      axiom((A:-B)),
      demo(axiom,B).

demo(user,exist,A,YN) :-
      query(oracle,user,exist,A,YN),
      (ground(A)->assert(intmodel(A-YN));true).
demo(user,forall,A,YN) :-
      query(oracle,user,forall,A,YN),
      (ground(A)->assert(intmodel(A-YN));true).
demo(user,comp,(A,As),YN) :-
      query(oracle,user,comp,(A,As),YN).
```

โปรแกรมที่ 5-14 อินเตอร์พรีทเตอร์ของออราคิล

บทที่ 6
บทสรุป

6.1 บทสรุป

ดีบักเกอร์ที่สร้างขึ้นใช้สำหรับดีบักโปรแกรมที่เขียนด้วยภาษาเพียวโปรล็อก (Pure Prolog) ดีบักเกอร์ทำหน้าที่เป็นตัววินิจฉัยซึ่งทำงานแบบตอบโต้กับโปรแกรมเมอร์เพื่อหาข้อผิดพลาดในโปรแกรม ตัววินิจฉัยพัฒนาขึ้นโดยใช้เทคนิค Algorithmic Program Debugging ซึ่งเป็นเทคนิคแบบไดนามิกล้วนๆ ใช้คอมพิวเตชันทรี (Computation Tree) ในการวิเคราะห์

นอกจากนี้เพื่อให้ดีบักเกอร์มีความอัตโนมัติ (ซึ่งทำให้ดีบักเกอร์ดูมีความฉลาดมากขึ้น) จึงมีการทำ Query Minimization เพื่อลดการไควรี่ไปยังผู้ใช้ในระหว่างการวินิจฉัย โดยได้ทำใน 2 แนวทางคือ 1) ปรับปรุงอัลกอริทึมการวินิจฉัยให้มีการไควรี่น้อยลง นั่นคือ ลด Query Complexity ของอัลกอริทึม 2) สามารถใช้ Specifications และ Properties ที่ผู้ใช้กำหนด เพื่อช่วยในการวินิจฉัยได้

เพื่อให้การใช้งานดีบักเกอร์สะดวกขึ้น จึงได้พัฒนา IDE ที่มีส่วนติดต่อผู้ใช้เป็นแบบกราฟิค ซึ่งทำให้ผู้ใช้สามารถพัฒนาโปรแกรม และดีบักโปรแกรมได้ง่ายขึ้น อีกทั้งผู้ใช้ยังสามารถดูคอมพิวเตชันทรีของการประมวลผลได้อีกด้วย

6.2 แนวทางการพัฒนาต่อ

1. สังเกตว่าเทคนิค APD หาข้อผิดพลาดโดยใช้คอมพิวเตชันทรีเท่านั้น ไม่ได้เข้าไปวิเคราะห์ซอร์สโค้ดของโปรแกรมเลยแม้แต่น้อย ซึ่งต่างจากการดีบักของคนที่ใช้ข้อมูลทั้งสองอย่าง หากสามารถปรับปรุงเทคนิคให้สามารถใช้ซอร์สโค้ดมาช่วยวิเคราะห์ด้วย น่าจะทำให้การถามผู้ใช้ลดลงมาก
2. การขยายให้ดีบักเกอร์ทำงานกับภาษาที่กว้างขึ้น เช่น Full Prolog หรือภาษาแบบ Procedural ทั่วๆ ไปเช่น ปาสคาล หรือ C เป็นต้น ภาษาอีกประเภทหนึ่งซึ่งดีบักยากคือ ภาษาแบบ Concurrent (Concurrent Programming Language) เช่น ภาษา PARLOG หรือ ภาษา Java ซึ่งสามารถเขียนโปรแกรมแบบ Multithreading ได้ ปัจจุบันยังไม่มีดีบักเกอร์สำหรับภาษาประเภทนี้ที่ใช้งานได้อย่างมีประสิทธิภาพ
3. การทำให้ดีบักเกอร์สามารถแก้ไขข้อผิดพลาด (Correction) ได้เอง การศึกษาด้านนี้มีความเกี่ยวข้องกับการทำ Program Synthesis
4. การพัฒนา IDE ให้สมบูรณ์ยิ่งขึ้น เพื่อเป็นระบบที่ผู้ใช้สามารถพัฒนาโปรแกรมและใช้งานดีบักเกอร์ได้สะดวก

ภาคผนวก ก
ตัวอย่างการวินิจฉัยข้อผิดพลาด

ตัวอย่างที่ 1 โปรแกรม insertion sort ที่ให้ผลลัพธ์ไม่ถูกต้อง

```
sort([A|B],C) :-
      sort(B,D),
      insert(A,D,C).
sort([],[]).

insert(A,[B|C],[B|D]) :-
      A > B,
      insert(A,C,D).
insert(A,[B|C],[B|C]) :-
      A =< B.
insert(A,[],[A]).
```

เมื่อใช้ตัววินิจฉัยแบบ top-down

```
?- demo(sort([4,2,3,5,1],Ls),Ts),fp(Ts,Err).
sort([4,2,3,5,1],[1,5]) Valid? n
sort([2,3,5,1],[1,5]) Valid? n
sort([3,5,1],[1,5]) Valid? n
sort([5,1],[1,5]) Valid? y
insert(3,[1,5],[1,5]) Valid? n
insert(3,[5],[5]) Valid? n
Err = insert(3,[5],[5]) :- 3 =< 5
```

เมื่อใช้ตัววินิจฉัยแบบ divide-and-query

```
?- demo(sort([4,2,3,5,1],Ls),Ts),fp(Ts,Err).
sort([5,1],[1,5]) Valid? y
sort([3,5,1],[1,5]) Valid? n
insert(3,[1,5],[1,5]) Valid? n
insert(3,[5],[5]) Valid? n
Err = insert(3,[5],[5]) :- 3 =< 5
```

ตัวอย่างที่ 2 โปรแกรม quick sort ที่ให้ผลลัพธ์ไม่ถูกต้อง

```
sort([A|B],C) :-
      partition(A,B,D,E),
      sort(E,F),
      sort(D,G),
      append(G,[A|F],C).
sort([],[]).
```

```
partition(A,[B|C],[B|D],E) :-
      B =< A,
      partition(A,C,D,E).
partition(A,[B|C],D,[B|D]) :-
      B > A,
      partition(A,C,D,_).
partition(_,[],[],[]).
append([],A,A).
append([A|B],C,[A|D]) :-
      append(B,C,D).
```

เมื่อใช้ตัววินิจฉัยแบบ top-down

```
?- demo(sort([4,2,3,5,1],Ls),Ts),fp(Ts,Err).
sort([4,2,3,5,1],[1,2,1,3,4,1,5]) Valid? n
partition(4,[2,3,5,1],[2,3,1],[5,1]) Valid? n
partition(4,[3,5,1],[3,1],[5,1]) Valid? n
partition(4,[5,1],[1],[5,1]) Valid? n
partition(4,[1],[1],[]) Valid? y
Err = partition(4,[5,1],[1],[5,1]) :- 5 > 4,partition(4,[1],[1],[])
```

เมื่อใช้ตัววินิจฉัยแบบ divide-and-query

```
?- demo(sort([4,2,3,5,1],Ls),Ts),fp(Ts,Err).
sort([2,3,1],[1,2,1,3]) Valid? n
sort([3,1],[1,3]) Valid? y
sort([1],[1]) Valid? y
partition(2,[3,1],[1],[3,1]) Valid? n
partition(2,[1],[1],[]) Valid? y
Err = partition(2,[3,1],[1],[3,1]) :- 3 > 2,partition(2,[1],[1],[])
```

ตัวอย่างที่ 3 โปรแกรม matrix multiplication ที่ให้ผลลัพธ์ไม่ถูกต้อง

```
vector_mult([],[],[]).
vector_mult([A|B],[C|D],[E|F]) :-
      E is A * C,
      vector_mult(B,D,F).
vector_sum([_],0).
vector_sum([A|B],C) :-
      vector_sum(B,D),
      C is D + A.
vector_dot(A,B,C) :-
      vector_mult(A,B,D),
      vector_sum(D,C).
matrix_trans([[]|_],[]).
matrix_trans(A,[B|C]) :- matrix_column(A,B,D),matrix_trans
(D,C).
matrix_column([],[],[]).
matrix_column([[A|B]|C],[A|D],[B|E]) :-
      matrix_column(C,D,E).
matrix_mult(A,B,C) :-
```

```
        matrix_trans(B,D),
        matrix_mult1(A,D,C).
matrix_mult1([],_,[]).
matrix_mult1([A|B],C,[D|E]) :-
        vect_dot_mat(A,C,D),
        matrix_mult1(B,C,E).
vect_dot_mat(_,[],[]).
vect_dot_mat(A,[B|C],[D|E]) :-
        vector_dot(A,B,D),
        vect_dot_mat(A,C,E).
```

เมื่อใช้ตัววินิจฉัยแบบ top-down

```
?- demo(matrix_mult([[1,2],[3,4]],[[4,3],[2,1]],Rs),Ts),fp(Ts,Err).
matrix_mult([[1,2],[3,4]],[[4,3],[2,1]],[[4,3],[12,9]]) Valid? n
matrix_trans([[4,3],[2,1]],[[4,2],[3,1]]) Valid? y
matrix_mult1([[1,2],[3,4]],[[4,2],[3,1]],[[4,3],[12,9]]) Valid? n
vect_dot_mat([1,2],[[4,2],[3,1]],[4,3]) Valid? n
vector_dot([1,2],[4,2],4) Valid? n
vector_mult([1,2],[4,2],[4,4]) Valid? y
vector_sum([4,4],4) Valid? n
vector_sum([4],0) Valid? n
Err = vector_sum([4],0)
```

เมื่อใช้ตัววินิจฉัยแบบ divide-and-query

```
?- demo(matrix_mult([[1,2],[3,4]],[[4,3],[2,1]],Rs),Ts),fp(Ts,Err).
matrix_mult1([[3,4]],[[4,2],[3,1]],[[12,9]]) Valid? n
vect_dot_mat([3,4],[[3,1]],[9]) Valid? n
vector_mult([3,4],[3,1],[9,4]) Valid? y
vector_dot([3,4],[3,1],9) Valid? n
vector_sum([9,4],9) Valid? n
vector_sum([4],0) Valid? n
Err = vector_sum([4],0)
```

ภาคผนวก ข
โค้ดโปรแกรม

```
main:-
        wfcreate(font,'Terminal',12,0),
        init,
        agent,
        shell,
        close_system.

init :-
        assert(hyp([])),retract(hyp([])),
        assert(specification([])),retract(specification([])),
        assert(property([])),retract(property([])),
        assert(axiom([])),retract(axiom([])),
        assert(intmodel([])),retract(intmodel([])),

        assert(stack_depth(20)).

shell :-
        talk(`~M~J<<Welcome to Intelligent Prolog Debugging System>>~M~J`),
        talk(program,`Enter a goal to query`),
        talk(`~M~J`),

        user_input(Term),
        (
                Term=exit,!
        ;
                query(user,program,Term,_),
                shell
        ).

%%%%%%%% Agent Querying %%%%%%%%
query(program,user,Term,YN) :-
        %talk(program,Term),
        message_box(yesno,Term,YN).
        %talk(user,YN).

query(user,program,Term,YN) :-
        talk(program,`Querying ...`),

        stack_depth(Depth),
        demo(program,Term,YN,Proofs,Depth),
        (
                YN=yes,
                talk(program,Term),

                list_string([Term,`~M~JDiagnose?`],MsgDiag),
                query(program,user,MsgDiag,YNDiag),
                (       YNDiag=yes,
                        talk(`~M~J`),
                        talk(program,`Diagnosing the incorrect solution ...`),
                        (       fp(Proofs,Err)
                                ->show_error(Err)
                                ;talk(program,`No error found.`)
                        ),
                        show_tree(Proofs)
                ;
                        YNDiag=no,
                        show_tree(Proofs),
                        query(program,user,`Find more solutions?`,YNMore),
                        YNMore=no
                )
```

```
    ;
            YN=no,!,
            talk(program,`No solution!`),
            query(program,user,`No solution!~M~JDiagnose?`,YNMiss),
            (       YNMiss=yes,
                    talk(`~M~J`),
                    talk(program,`Diagnosing the missing solution ...`),
                    (       ip(Term,Err)
                            ->show_error(Err)
                            ;talk(program,`No error found.`)
                    )
            ;
                    YNMiss=no
            )
    ;
            YN=overflow,!,
            talk(program,`Stack overflow!`),
            show_tree(Proofs)
    ).

show_tree(Proofs) :-
    query(program,user,`Show the computation tree?`,YNTree),
    (       YNTree=yes,
            tree,
            write_tree(0,Proofs,TreeStr),
            wtext((tree,8005),TreeStr)
    ;
            YNTree=no
    ).


show_error(Err) :-
    Err=incorrect(ErrClause)
    ->
    talk(program,[`The incorrect clause is `,ErrClause,`~M~J`])
    ;
    Err=incomplete(ErrClause),
    talk(program,[`The missing clause is `,ErrClause,`~M~J`]).

talk([H|T]) :- !,
    list_string([H|T],Str),
    talk(Str).
talk(Term) :-
    wtext((agent,8000),StrOld),
    term_string(Term,StrTerm),
    cat([StrOld,`~M~J`,StrTerm],StrNew,_),
    wtext((agent,8000),StrNew),
    wedtsel((agent,8000),64000,64000).

talk(From,[H|T]) :- !,
    list_string([H|T],Str),
    talk(From,Str).
talk(From,Term) :-
    wtext((agent,8000),StrOld),

    term_string(From,StrFrom),
    term_string(Term,StrTerm),

    cat([StrOld,`~M~J`,StrFrom,` : `,StrTerm],StrNew,_),
    wtext((agent,8000),StrNew),
    wedtsel((agent,8000),64000,64000).

user_input(Term) :-
    retractall(user_input_data(_)),
    user_input1(Term).

user_input1(Term) :-
    user_input_data(Str),
    string_term(Str,Term),!.

user_input1(Term) :- user_input1(Term).
```

```
%%%%%%%% General Knowledge %%%%%%%%
:-op(1200,xfx,=>).
:-op(1200,xfx,<=>).

%---------- Builtin predicates -------------
builtin(true).
builtin(fail).
builtin(_ = _).
builtin(_ \= _).
builtin(_ == _).
builtin(_ \== _).
builtin(_ < _).
builtin(_ =< _).
builtin(_ > _).
builtin(_ >= _).
builtin(_ is _).

%%%%%%%% Program Agent %%%%%%%%
%------------- Demo ------------
demo(program,A,YN) :-
       demo(program,A),YN=yes;
       YN=no.

demo(program,A) :-
       builtin(A),!,A.
demo(program,(A,B)) :- !,
       demo(program,A),
       demo(program,B).
demo(program,not(A)) :- !,
       demo(program,A,YN),!,YN=no.
demo(program,A) :-
       hyp(C),
       (C=(A:-B)->demo(program,B);A=C).

%------- Demo that collects proof ------
demo(program,A,YN,Ts) :-
       demo_tree(program,A,Ts),YN=yes;
       YN=no,Ts=[fail(A,[])].

demo_tree(program,A,[true(A,[])]) :-
       builtin(A),!,A.
demo_tree(program,(A,B),Ts) :- !,
       demo_tree(program,A,As),
       demo_tree(program,B,Bs),
       append(As,Bs,Ts).
demo_tree(program,not(A),Ts) :- !,
       demo(program,A,YN,Ts),!,YN=no.
demo_tree(program,A,[true(A,Ts)]) :-
       hyp(C),
       (C=(A:-B)->demo(program,B,Ts);A=C,Ts=[]).

%------- Demo that collects proof and overflow check ------
demo(program,A,YN,Ts,Depth) :-
       demo_tree(program,A,Ts,Depth,OV),YN=OV;
       YN=no,Ts=[fail(A,[])].

demo_tree(program,A,[overflow],0,overflow) :- !.

demo_tree(program,A,[true(A,[])],D,yes) :-
       builtin(A),!,A.

demo_tree(program,(A,B),Ts,D,OV) :- !,
       demo_tree(program,A,As,D,OVA),
       (      OVA=overflow
              ->
              OV=OVA,Ts=As
              ;
              demo_tree(program,B,Bs,D,OV),
              append(As,Bs,Ts)
```

```
        ).

demo_tree(program,not(A),Ts,D,OV) :- !,
        demo(program,A,YN,TAs,D),!,
        (       YN=overflow -> OV=YN,Ts=[overflow]
                ;
                YN=no -> OV=yes,Ts=TAs
        ).

demo_tree(program,A,[true(A,Ts)],D,OV) :-
        hyp(C),
        (       C=(A:-B)
        ->      D1 is D-1,
                demo_tree(program,B,Ts,D1,OV)
        ;       A=C,Ts=[],OV=yes
        ).

%------ Diagnose incorrect solution -----
fp([T1|Ts],C) :-
        fp(T1,C),!;
        fp(Ts,C).

fp(true(A,_),C) :-
        builtin(A),!,fail.
fp(fail(A,_),C) :-
        ip(A,C).
fp(true(A,Bs),C) :-
        query(program,oracle,forall,A,no),
        (
        fp(Bs,C),!;
        treelist_conj(Bs,Body),C=incorrect((A:-Body))
        ).
treelist_conj([],true) :- !.
treelist_conj([true(A,_)],A) :- !.
treelist_conj([fail(A,_)],not(A)) :- !.
treelist_conj([T1|Ts],(C1,Cs)) :-
        treelist_conj([T1],C1),
        treelist_conj(Ts,Cs).

%------- Diagnose finite failure --------------
ip(A,C) :-
        builtin(A),!,fail.
ip(not(A),C) :- !,
        demo(program,A,yes,Ts),!,fp(Ts,C).
ip((A,B),C) :- !,
        (       demo(program,A),
                ip(B,C),!
        ;
                ip(A,C)
        ).
ip(A,C) :-
        findall(A,demo(program,A),As),
        freeze(A,A1),not_exist(member(A1,As)),
        query(program,oracle,comp,(A,As),no),
        (       hyp((A:-B)),
                ip(B,C),!
        ;
                C=incomplete(A)
        ).

%--------- Hypotheses ----------
%hyp( (p :- a,b) ).
%hyp( (a :- c,d) ).
%hyp( (b :- true) ).
%hyp( (c :- true) ).
%hyp( (d :- true) ).
%hyp( (u :- v) ).
%hyp( (v :- w) ).
%hyp( (w :- u) ).
```

```
%hyp( (sort([X|Xs],Ys) :- sort(Xs,Zs),insert(X,Zs,Ys)) ).
%hyp( (sort([],[]) :- true) ).
%hyp( (insert(X,[Y|Ys],[Y|Zs]) :- (X>Y),insert(X,Ys,Zs)) ).
%hyp( (insert(X,[Y|Ys],[Y|Ys]) :- (X=<Y)) ).
%hyp( (insert(X,[],[X]) :- true) ).

%hyp( (diff(X,Ys,Zs) :- memb(X,Ys) , not(memb(X,Zs))) ).
%hyp( (diff(X,Ys,Zs) :- memb(X,Zs) , not(memb(X,Ys))) ).
%hyp( (memb(X,[X|_]) :- true) ).
%hyp( (memb(X,[_|Ys]) :- memb(X,Ys)) ).

query(program,oracle,forall,Term,YN) :-
       talk(program,[Term,` Valid?`]),
       demo(oracle,forall,Term,YNValid,Source),
       talk(oracle,[YNValid,` -from `,Source]),
       YN=YNValid.

query(program,oracle,comp,(A,[]),YNComp) :- !,
       talk(program,[A,` Exist?`]),
       demo(oracle,exist,A,YNExist,Source),
       talk(oracle,[YNExist,` -from `,Source]),
       (YNExist=yes->YNComp=no;YNComp=yes).

query(program,oracle,comp,(A,As),YN) :-
       list_string([`The solutions of `,A,` are `,As,` Complete?`],MsgComp),
       talk(program,MsgComp),
       demo(oracle,comp,(A,As),YNComp,Source),
       talk(oracle,[YNComp,` -from `,Source]),
       YN=YNComp.


%%%%%%%% Oracle Agent %%%%%%%%
%---------- Demo -------------
demo(oracle,exist,A,YN,Source) :-
       ground(A),!,
       (
       demo(specification,A,YNSpec)->YN=YNSpec,Source=specification;
       demo(property,A,YNProp)->YN=YNProp,Source=property;
       demo(intmodel,A,YNInt)->YN=YNInt,Source=intmodel;
       demo(user,exist,A,YNUser)->YN=YNUser,Source=user
       ).
demo(oracle,exist,A,YN,user) :-
       demo(user,exist,A,YN).

demo(oracle,forall,A,YN,Source) :-
       ground(A),!,
       (
       demo(specification,A,YNSpec)->YN=YNSpec,Source=specification;
       demo(property,A,YNProp)->YN=YNProp,Source=property;
       demo(intmodel,A,YNInt)->YN=YNInt,Source=intmodel;
       demo(user,forall,A,YNUser)->YN=YNUser,Source=user
       ).
demo(oracle,forall,A,YN,user) :-
       demo(user,forall,A,YN).

demo(oracle,comp,(A,As),YN,user) :-
       demo(user,comp,(A,As),YN).

demo(specification,A,YN) :-
       specification( (A<=>S) ),
       demo(axiom,S,YN).
demo(property,A,no) :-
       property( (A=>F) ),
       demo(axiom,F,no).
demo(intmodel,A,YN) :-
       intmodel(A-YN).
demo(axiom,A,YN) :-
       demo(axiom,A)->YN=yes;YN=no.
demo(axiom,true).
demo(axiom,(A,B)) :- !,
```

```
        demo(axiom,A),
        demo(axiom,B).
demo(axiom,A) :-
        builtin(A),!,A.
demo(axiom,A) :-
        axiom((A:-B)),
        demo(axiom,B).

demo(user,exist,A,YN) :-
        query(oracle,user,exist,A,YN),
        (ground(A)->assert(intmodel(A-YN));true).
demo(user,forall,A,YN) :-
        query(oracle,user,forall,A,YN),
        (ground(A)->assert(intmodel(A-YN));true).
demo(user,comp,(A,As),YN) :-
        query(oracle,user,comp,(A,As),YN).

%---------- Specification -------------
%specification([]).

%---------- Property -------------
%property([]).
%property( (sort(Xs,Ys) => length(Xs,N),length(Ys,N)) ).
%property( (insert(X,Xs,Ys) => length(Xs,N),length(Ys,s(N))) ).

%---------- Axiom -------------
%axiom( (length([],0) :- true) ).
%axiom( (length([H|Ts],s(N1)) :- length(Ts,N1)) ).

query(oracle,user,forall,Term,YN) :-
        list_string([Term,`~M~JValid?`],MsgTerm),
        message_box(yesno,MsgTerm,YN).

query(oracle,user,exist,Term,YN) :-
        list_string([Term,`~M~JExist?`],MsgTerm),
        message_box(yesno,MsgTerm,YN).

query(oracle,user,comp,(A,As),YN) :-
        list_string([`The solutions of `,A,` are~M~J`,As,`~M~JComplete?`],MsgComp),
        message_box(yesno,MsgComp,YN).


%%%%%%%% Agent %%%%%%%%
agent :-
    _S1 = [dlg_ownedbyprolog,ws_sysmenu,ws_minimizebox,ws_caption],
    _S2 = [ws_child,bs_groupbox,ws_visible],
    _S3 = [ws_child,ws_visible,ws_tabstop,ws_border,es_left,es_multiline,
ws_hscroll,ws_vscroll,es_autohscroll,es_autohscroll,es_readonly],
    _S4 = [ws_child,ws_visible,ws_tabstop,ws_border,es_left,
es_autohscroll,es_autovscroll,es_wantreturn],
    _S5 = [ws_child,ws_visible,ws_tabstop,bs_pushbutton],
    wdcreate(  agent, `Agent Communication`,  44,  73, 702, 440, _S1 ),
    wccreate( (agent,11000), button, `Agent's Communication Room`, 5,5,690,410,_S2 ),
    wccreate( (agent,8000), edit,   ``, 15, 110, 670, 215, _S3 ),%chat edit
    wccreate( (agent,8001), edit,   ``, 30, 360, 590,  25, _S4 ),%query edit
    wccreate( (agent,11001), button, `View`,                    10,  35, 325,  70, _S2 ),
    wccreate( (agent,1000), button, `Program Agent`,            15,  65, 150,  30, _S5 ),
    wccreate( (agent,1001), button, `Oracle Agent`,            170,  65, 160,  30, _S5 ),
    wccreate( (agent,1002), button, `Query`,                625, 360, 50,   25, _S5 ) ,
    wccreate( (agent,11002), button, ` User`,                   15, 330, 665,  70, _S2 ),
    window_handler(agent,debug_handler),

    wfont((agent,8000),font),
    wfont((agent,8001),font),

    wshow(agent,1).

debug_handler((agent,1000),msg_button,_,_):-
    program.


debug_handler((agent,1001),msg_button,_,_):-
```

```
        wccreate( (oracle,8003),  edit,    ``, 320, 135, 295,  25, _S5 ),%add Spec
        wccreate( (oracle,1003),  button, `Add`, 620, 135,  80,  25, _S4 ),
        wccreate( (oracle,1004),  button, `Remove`, 705, 135,  80,  25, _S4 ),
        wccreate( (oracle,1008),  button, `Remove`, 705, 305,  80,  25, _S4 ),
        wccreate( (oracle,1009),  button, `Add`, 620, 305,  80,  25, _S4 ),
        wccreate( (oracle,1010),  button, `Add`, 620, 475,  80,  25, _S4 ),
        wccreate( (oracle,1011),  button, `Remove`, 705, 475,  80,  25, _S4 ),
        wccreate( (oracle,8004),  edit,    ``, 320, 305, 295,  25, _S5 ),%add Prop
        wccreate( (oracle,8005),  edit,    ``, 320, 475, 295,  25, _S5 ),%add Axiam
        wccreate( (oracle,8006),  edit,    ``, 10, 35, 290,  400, _S3 ),%edit Intended
        wccreate( (oracle,1012),  button, `Clear Intended`,  50,  450,200,  25, _S4 ),
        window_handler(oracle,debug_handler),
        wfont((oracle,8000),font),

        wfont((oracle,8001),font),
        wfont((oracle,8002),font),
        wfont((oracle,8003),font),
        wfont((oracle,8004),font),
        wfont((oracle,8005),font),
        wfont((oracle,8006),font),

        wshow(oracle,1),
        update_oracle.

    debug_handler((oracle,1000),msg_button,_,_) :-
          opnbox('Open',[('Source','*.pl'),('Text','*.txt')],'*.*','pl',[F1|Fs]),
          open(F1,read),
          see(F1),
          read_file1,
          seen,
          close(F1),
          update_oracle.

    debug_handler((oracle,1001),msg_button,_,_):-
          savbox(`Save`,[(`Source`,`*.pl`),(`Text`,`*.txt`)],`*.*`,`pl`,[F1|Fs]),
          open(F1,write),
          tell(F1),
          write_file1(specification),
          write_file1(property),
          write_file1(axiom),
          write_file1(intmodel),
          told,
          close(F1).

    update_oracle :-
          write_file(specification) ~> Spec,
          write_file(property) ~> Prop,
          write_file(axiom) ~> Axiom,
          write_file(intmodel) ~> Int,
           wtext((oracle,8000),Spec),
          wtext((oracle,8001),Prop),
          wtext((oracle,8002),Axiom),
          wtext((oracle,8006),Int).

    debug_handler((oracle,1003),msg_button,_,_):-
          wtext((oracle,8003),Input),
          cat([Input,` `],Input1,_),
          read(Term1) <~ Input1 ,
          assert(specification(Term1)),
          write_file(specification)~>String,
          wtext((oracle,8000),String).

    debug_handler((oracle,1004),msg_button,_,_):-
          wtext((oracle,8003),Input),
          cat([Input,` `],Input1,_),
          read(Term1) <~ Input1 ,
          retract(specification(Term1)),
          write_file(property)~>String,
          wtext((oracle,8000),String).

    debug_handler((oracle,1009),msg_button,_,_):-
```

```
    wfont((program,8000),font),
    wfont((program,8001),font),

    wshow(program,1),
    update_program.

debug_handler((program,10012),msg_button,_,_):-
        retractall(stack_depth(N)),
        wtext((program,8001),Stadpt),
        string_term(Stadpt,Termsta),
        assert(stack_depth(Termsta)).

debug_handler((program,1002),msg_button,_,_):-
        retractall(hyp(_)),
        wtext((program,8000),Old),
        cat([Old,` `],Old1,_),
        read_file(hyp) <~ Old1,
        update_program.

debug_handler((program,10010),msg_button,_,_):-
        retractall(hyp(X)),
        wtext((program,8000),``).

debug_handler((program,1008),msg_button,_,_) :-
        opnbox('Open',[('Source','*.pl'),('Text','*.txt')],'*.*','pl',[F1|Fs]),
        open(F1,read),
        see(F1),
        read_file(hyp),
        seen,
        close(F1),
        update_program.

debug_handler((program,1009),msg_button,_,_):-
        savbox(`Save`,[(`Source`,`*.pl`),(`Text`,`*.txt`)],`*.*`,`pl`,[F1|Fs]),
        open(F1,write),
        tell(F1),
        write_file(hyp),
        told,
        close(F1).

update_program :-
        write_file(hyp) ~> Hyp,
         wtext((program,8000),Hyp),
        stack_depth(N),
        term_string(N,Str),
        wtext((program,8001),Str).


%%%%%%%% Utilities %%%%%%%%

read_file(Pred) :-
        repeat,
        read(Term),
        (       Term = end_of_file,!;
                assert(Pred(Term)),fail
        ).
read_file1 :-
        repeat,
        read(Term),
        (       Term = end_of_file,!;
                assert(Term),fail
        ).
write_file(Pred) :-
        Pred(Term),
        pretty_print(Term),
        write(`.`),nl,
        fail.
write_file(Pred) :- !.

write_file1(Pred) :-
        Pred(Term),
```

```
        pretty_print(Pred(Term)),
        write(`.`),nl,
        fail.
write_file1(Pred) :- !.

string_term(Str,Term) :-
        cat([Str,` `],Str1,_),
        read(Term) <~ Str1.

term_string(Term,Str) :-
        pretty_write(Term) ~> Str.

pretty_write(Term) :-
        vars(Term,Vs),
        ewrite(Term,Vs).
pretty_print(Term) :-
        vars(Term,Vs),
        eprint(Term,Vs).

list_string([],``).
list_string([H|T],Str) :-
        term_string(H,StrH),
        list_string(T,StrT),
        cat([StrH,StrT],Str,_).

not_exist(P) :-
        P,!,fail.
not_exist(P).

freeze(Term,Ground) :-
        copy_term(Term,Ground),
        numbervars(Ground,0,_).

%%%%%%%% Close System %%%%%%%%

debug_handler(agent,msg_close,_,_):-save_windows,close_system.
debug_handler(program,msg_close,_,_):-wclose(program).
debug_handler(oracle,msg_close,_,_):-wclose(oracle).

close_system :-
        assert(user_input_data(`exit.`)),
        wshow(agent,0).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

บรรณานุกรม

[1]. Hirankitti V. : "Applying Scientific Method to Program Debugging", Ph.D. Thesis, Imperial College, 1998.

[2]. Shapiro E. : "*Algorithmic Program Debugging*", MIT Press , 1983.

[3]. Van Le T. : "*Techniques of Prolog Programming with implementation of logical negation and quantified goals*", John Wiley & Sons, 1993.

[4]. Sterling L. and Shapiro E. : "*The Art of Prolog*", MIT Press, 1994.

[5]. Sommerville I., : "*Software Engineering*", Addison-Wesley, 2001.

[6]. Chang C. and Lee R. : "*Symbolic Logic and Mechanical Theorem Proving*", Academic Press.

[7]. Kowalski R. : "*Logic for Problem Solving*", North-Holland.

[8]. Genesereth M. R., and Nilsson N. J. : "*Logical Foundations of Artificial Intelligence*", Morgan Kaufmann.

[9]. Hogger C. J. : "*Essentials of Logic Programming*", Clarendon Press, 1990.

[10]. Genesereth M. R. and Ginsberg M. L. : "*Logic programming*" , CACM 28 , 1985.

[11]. Fritzson P., Shahmehri N., Kamikar M. and Gyimothy T. : "*Generalized Algorithmic Debugging and Testing*" , ACM Letters on Programming Languages and Systems 1 , 1992.

[12]. Rich C. and Waters R. C. : "*Artificial Intelligence and Software Engineering*" , AI in 1980 and beyond , MIT Press.