에이전트 기반 시스템 통합 및 자동화 방안*

이재호, 이기훈**

An Agent-Based System Integration and Automation Framework

Jaeho Lee, Kihoon Lee

요 약

하나의 에이전트로 해결하기 어려운 복잡한 문제를 해결하기 위한 방안으로 여러 에이전트들 간의 상호작용과 협동을 통하여 작업이 수행되는 멀티에이전트 시스템을 구현하는 것이 점차 보편화되고 있다. 멀티에이전트 시스템의 중요한 특징 중의 하나는 여러 에이전트들 간의 능력과 정보가 서로 공유된다는 점이다. 멀티에이전트 시스템에서는 한 에이전트의 처리 결과를 다른 에이전트가 이용할 수도 있으며, 그 결과 또한 다시 다른 에이전트에게 제공되기도 한다. 그러나 각각의 에이전트들은 서로 다른 환경 하에서 구축된 이형질성과 원격지에서 작동하는 분산성으로 인해 하나의 시스템으로 통합하기가 용이하지 않다. 이를 해결하기 위한 방법의 하나로 이형질의 에이전트들 간에 정보 교환을 위한 파일랩퍼(File Wrapper) 이용 방안이 있다. 한 에이전트의 처리 결과가 파일이라는 일반적인 형태로 존재한다고 할 때, 이러한 파일내의 정보를 추출하고 조합 및 가공하는 기능을 담당하는 것을 파일랩퍼(File Wrapper)라고 한다.

본 연구에서는 Java와 JavaCC를 이용하여 에이전트 스크립트 언어로 새롭게 구현된 파일랩퍼를 소개한다. 이 파일랩퍼는 파일에서 정보를 추출하기 위한 명령어들과 출력에 관련한 명령어들 그리고 작업의 효율성을 위한 명령어들로 구성된 텍스트 기반 언어로 구성되어 있다.

Key words : File Wrapper, Script

* 본 연구는 첨단정보기술 연구센터를 통하여 과학재단의 지원을 받았음.
** 서울시립대학교 대학원 전자전기컴퓨터공학부 및 첨단정보기술연구센터(AITrc)

1. 서론

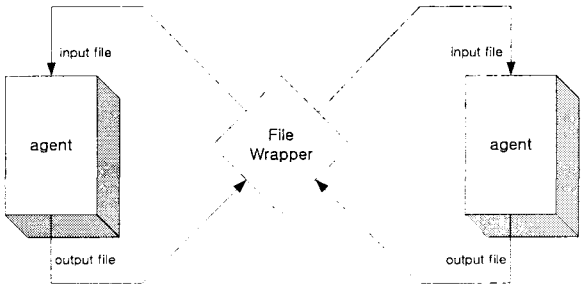
에이전트에 대한 정의는 간단히 누군가를 위하여 무엇인가를 대신 해주는 컴퓨터 프로그램을 통칭한다. 따라서 에이전트는 사용자가 보다 편리하게 컴퓨터를 이용하는 것을 가능하게 한다. 에이전트는 스스로 학습하고, 일을 처리하기 위한 일련의 계획을 생성하기도 한다. 이러한 에이전트 기술을 이용하여 사용자의 간단한 조작만으로도 에이전트 스스로가 주어진 문제를 해결하기도 한다. 그러나 복잡한 문제에 있어서 하나의 에이전트로는 해결하기 어려운 점이 있다. 그래서 여러 개의 에이전트들이 상호작용과 협동에 의해서 한 에이전트가 해결하기 어려운 복잡한 문제의 해결을 추구한다. 이를 멀티 에이전트 시스템이라 한다. 멀티 에이전트 시스템이 가지는 특징은 여러 에이전트들간의 능력과 정보가 서로 공유된다는 것이다. 그러나 각각의 에이전트들은 서로 다른 환경 하에서 구축된 이형질성(heterogeneity)과 원격지에서 작동하는 분산성(distribution)으로 인해 하나의 시스템으로 통합하기가 용이하지 않다.

이러한 에이전트들의 이형질성과 분산성을 해결하기 위한 한 방법으로 본 연구에서는 파일랩퍼(File Wrapper)를 사용한다. 각각의 에이전트가 처리 결과의 정보를 파일로 저장한다고 할 때 다른 에이전트가 그 정보를 이용할 경우 정보를 추출하고 조합하는 기능이 요구된다. 바로 그 기능을 담당하는 것이 파일랩퍼이다. 이를 위해 JavaCC와 Java를 사용하여 만든 텍스트 기반 스크립트 언어를 사용하여 해결하고자 한다.

2. 파일랩퍼(File Wrapper)의 개념

각각의 에이전트들은 저마다의 방식으로 정보를 저장하고 이용을 한다. 이러한 이질적인 정보를 에이전트가 사용하기 위해서는 정보의 추출과 조합이 필수적이다. 이를 위해서 에이전트 고유의 자료에서 추출하고자 하는 정보의 위치와 구조, 형식 등을 나타내는 규칙이 필요하며 이러한 규칙을 표현하고 실행하는 프로그램을 랩퍼(wrapper)라고 한다.

멀티 에이전트 시스템에서 한 에이전트의 처리 결과를 다른 에이전트가 이용할 수도 있으며 그 결과 또한 다시 다른 에이전트에게 제공되기도 한다. 한 에이전트의 처리 결과가 파일이라는 일반적인 형태로 존재한다고 할 때, 이러한 파일내의 정보를 추출하고 조합 및 가공하는 기능을 담당하는 것을 파일랩퍼(File Wrapper)라고 한다.



[그림 1] 파일랩퍼와 에이전트의 관계

3. 스크립트 언어의 어휘 규칙

- 스크립트 언어에서 사용하는 키워드(keyword)는 다음과 같고, 모두 예약어(reserved word)이다.

  echo get set find move goto bind while repeat if else try catch regexp integer float string to lines words characters

  위 문자들은 모두 대소문자를 구분하지 않는다

  아래 문자는 소문자로 써야 한다.

  $$exception $$lineIndex $$fieldIndex $$charIndex

- 특수 심볼(symbol)은 다음과 같다.

  + - * / ^ < <= > >= == ! != = && || , ;

  ( ) [ ] { } /* */

- 이외의 토큰으로 몇 가지가 있는데 아래와 같이 정의한다.

소문자와 대문자는 서로 다른 문자로 취급한다.

토 큰	정 의
#letter	(["-","_","A"-"Z","a"-"z"])
#digit	["0"-"9"]
#expn	["e","E"] (["+","-"])? (<digit>)+
#number0	(<digit>)+
#number1	(<digit>)+ "." (<digit>)* (<expn>)?
#number2	"." (<digit>)+ (<expn>)?
VAL_INTEGER	<number0>
VAL_FLOAT	(<number1> | <number2>)+
VAL_STRING	"\"" (~["\"","\n"] | "\"\"" | "\\\"")* "\""
IDENTIFIER	["_","A"-"Z","a"-"z"] (<letter> | <digit>)*
VARIABLE	"$" (<letter> | <digit> | ".")*
SYSVARIABLE	"$$" (<letter> | <digit> | ".")*

<표 1> 토큰의 정의

- 공백 문자(white space – \n, \t, \r, \f, " ")는 무시되는데, 토큰과 키워드는 가운데 공백 문자가 있어야 서로 분리된다.
- 주석(comment)은 자바(Java)에서처럼 /*와 */로 둘러싸거나 //을 사용한다.
- 변수명은 $로 시작하고 대소문자를 구별한다. 시스템 변수는 $$로 시작하고 4개의 예약어가 있다.

4. 스크립트 언어의 구조 및 설계

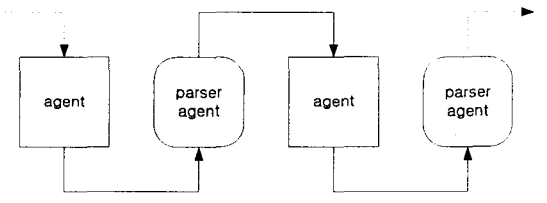
파일랩퍼는 에이전트의 처리 결과가 파일로 나타날 때 이 파일내의 정보를 추출하고 조합 및 가공하기 위한 역할을 한다. 파일랩퍼는 크게 파서(parser) 에이전트와 템플릿(template) 및 바인딩(binding) 관련 부분으로 나뉜다.

파서 에이전트는 파일랩퍼의 주요 부분이다. 에이전트간의 정보교환을 위해 에이전트의 처리 결과에 대한 정보의 위치와 형식 및 규칙 등을 정의한 스크립트 파일을 파서 에이전트가 해석을 하여 정보가 필요한 에이전트에게 넘긴다. 이때 사용하는 스크립트 언어는 JavaCC와 Java를 이용하여 만든 텍스트 기반 스크립트 언어이다. 파서 에이전트의 역할로 인해 서로 다른 이형질성의 에이전트라도 다른 에이전트로부터 결과를 넘겨 받아 내부에서 처리가 가능해진다. 또한 에이전트의 개발 및 보수에 있어서도 많은 부담을 덜어준다.

에이전트간의 정보교환의 실행 단계는 아래와 같다.

1. 현재 실행하고 있는 에이전트의 처리결과가 파일로 저장된다.
2. 파일로 저장된 정보를 파서 에이전트가 설정된 스크립트에 의해서 정보를 추출하고 조합하여 새로운 입력파일을 생성한다.
3. 파서 에이전트가 새로이 생성시킨 파일을 다음으로 실행될 에이전트의 입력으로 넘긴다.
4. 입력을 받은 에이전트가 수행된다.

만약 순차적으로 에이전트가 실행된다고 하면 위의 과정을 반복한다.



[그림 2] 에이전트의 정보 흐름

이 파서 에이전트에 쓰이는 파서를 BNF

(Backus-Naur Form) 문법 형태로 표현하면 다음과 같다.

```
program → statement-list
statement-list → statement
statement → assignment
          | block
          | echo-statement
          | get-statement
          | set-statement
          | move-statement
          | goto-statement
          | find-statement
          | bind-statement
          | while-statement
          | if-statement
          | trycatch-statement
assignment → VARIABLE = expression ;
expression → sum
sum → term PLUS term
    | term MINUS term
    | term
term → exponent MUL exponent
     | exponent DIV exponent
     | exponent
exponent → unary EXP exponent
         | unary
unary → MINUS element
      | element
element → VAL_FLOAT
        | VAL_INTEGER
        | VAL_STRING
        | PI
        | VARIABLE
        | SYSVARIABLE
        | function
        | ( sum )
function → IDENTIFIER ( param_list )
param_list → sum
           | param_list , sum
block → { statements }
echo-statement → ECHO expression-list ;
expression-list → expression
                | expression-list  expression
get-statement
      → GET KEY_INTEGER TO VARIABLE ;
       | GET KEY_FLOAT TO VARIABLE ;
       | GET KEY_STRING TO VARIABLE ;
set-statement
         → SET VARIABLE TO expression ;
move-statement
   → MOVE expression LINES ;
    | MOVE expression WORDS ;
    | MOVE expression CHARACTERS ;
    | MOVE expression REGEXP expression ;
goto-statement
        → GOTO expression LINES ;
         | GOTO expression WORDS ;
         | GOTO expression CHARACTERS ;
find-statement
            → FIND VAL_STRING ;
             | FIND REGEXP VAL_STRING ;
bind-statement
        → BIND expression TO IDENTIFIER ;
while-statement → repeat-stmt
                | while-stmt
repeat-stmt
       → REPEAT expression { statements }
while-stmt
        → WHILE ( condition ) { statements }
condition → cond_unit
cond_unit → cond_and OR cond_and
          | cond_and
cond_and → cond_element AND cond_element
         | cond_element
```

```
cond_element → sum EQ sum
             | sum NEQ sum
             | sum LT sum
             | sum LE sum
             | sum GT sum
             | sum GE sum
             | NOT sum
             | sum
             | ( cond_unit )
if-statement
→ IF ( condition ) block
| IF ( condition ) block ELSE block
| IF ( condition ) block else_if-list ELSE block
else_if-list → else_if-statement
             | else_if-list  else_if-statement
else_if-statement
             → ELSE IF ( condition ) block
trycatch-statement
             → TRY  block  CATCH  block
```

5. 스크립트 언어의 형식

명령어의 형식을 간략히 정리하면 아래와 같다.

명령어	형 식
echo	echo <*value* | *variable* | *escape char.*> [<...>];
get	get <*variable type*> to <*variable name*>;
set	set <*variable name*> to <*value* | *regular exp.*>;
find	find <*value*>; | find regexp <*regular exp.*>;
move	move <*value*> <lines | words | characters>; move <*value*> regexp <*regular exp.*>;
goto	goto <*value*> <lines | words | characters>;
bind	bind <*variable*> to <*variable*>;
while	while (<*condition*>) {...}
repeat	repeat <*condition*> {...}
if else	if (<*condition*>) {...} else {...}
try catch	try {...} catch {...}

<표 2> 스크립트 명령어 형식

- echo

값을 출력하고자 할 때 쓴다. 뒤에 나오는 인수(expression list)들을 표준 출력으로 내보낸다. 인수들은 공백으로 구분한다. $$exception은 예외 발생을 처리하기 위한 변수이다. 주로 catch문 내에서 쓰인다. $$lineIndex는 현재 커서가 몇 번째 라인에 위치한지를 나타낸다. $$fieldIndex는 라인 내에서 커서가 몇 번째 워드에 위치한지를 나타낸다. $$charIndex는 라인 내에서 커서가 몇 번째 문자에 위치한지를 나타낸다.

```
echo $a $b;       // 변수 a와 b의 값을 출력한다.
echo "Hello World!";// Hello World를 출력한다.
echo "\n";              // 줄 바꾸기
echo "Current Lines : " $$lineIndex "\n";
// 현재 라인의 위치를 출력한다.
echo "Current Words : " $$fieldIndex "\n";
echo "Current Characters : " $$charIndex
"\n";
echo $$exception;
// 예외 발생시 예외처리를 출력한다.
// catch문 내에서 쓰인다.
```

- get

get은 주로 move, goto, find와 같이 쓰이며, 그 명령어 다음에 위치한다. 현재 커서가 위치한 값(데이터 파일 내에서)을 구해 지정한 타입의 변수에 값을 가져온다. 예외가 발생할 수 있으므로 try catch문으로 감싼다.

```
get integer to $a;   // 정수형의 값을 변수 a로
가져온다.
```

```
get float to $b;        // 실수형의 값을 변수 b로 가져온다.
get string to $str;     // 문자열의 값을 변수 str로 가져온다.
```

- set

set은 아래 예처럼 변수에 값을 대입한다. 변수 타입은 값에 의해 자동으로 결정된다.

```
set $ival to 10;  // 변수 ival에 10을 대입한다.
set $fval to 12.34567;        // 변수 fval에 12.34567을 대입한다.
set $str2 to "Hello World!"; // 변수 str2에 문자열을 대입한다.
set $reg2 to "^[0-9]+";
// 변수 reg2에 정규 표현식 문자열을 대입한다.
set $ref2 to $ival;    // 변수 ref2에 변수 ival의 값을 대입한다.
```

- find

명령어 다음에 나오는 문자열을 찾거나 정규 표현식을 이용하여 해당 문자열을 찾을 때 쓰인다. 해당 문자열을 찾은 후 get을 이용해서 값을 가져온다. 문자열을 찾지 못하면 예외가 발생할 수 있으므로 try catch문으로 둘러싼다.

```
find "ten";          // ten이라는 문자열을 찾는다.
find regexp "^[0-9]+"; // 정규 표현식에 맞는 문자열을 찾는다.
find "$ival";      // $ival이라는 문자열을 찾는다.
```

- move

데이터 파일의 현재 커서의 위치를 이동시키기 위해 사용한다. 아래 goto와 달리 기존의 위치에서 주어진 값만큼 이동한다. line은 줄바꿈 문자(newline)로 구분하고 word는 공백으로 구분한다. 이 또한 예외가 발생할 수 있으므로 try catch문으로 감싼다.

```
move 4 lines; // 현 커서의 위치를 4줄 아래로 이동한다.
move 3 words; // 커서의 위치를 현 위치에서 3번째 단어로 이동한다.
```

- goto

이 또한 데이터 파일의 커서의 위치를 이동시키기 위해 사용한다. 단 위치 이동은 무조건 파일의 처음부터 시작한다. 절대 위치 이동이다. 일반적으로 try catch문으로 감싼다.

```
goto 5 lines; // 절대 위치로 이동. 파일의 5번째 줄로 이동한다.
goto 32 words; // 파일의 처음부터 32번째 단어로 이동한다.
goto 457 characters; // 파일의 처음부터 457번째 문자로 이동한다. 공백 제외.
goto $b words; // 또한 변수 사용 가능.
```

- bind

bind는 데이터 서버로 값을 보내는 역할을 한다. 첫번째 인수는 보낼 값이며 마지막 인수는 데이터 서버에 저장될 변수명이다.

```
bind $a to gradient;
// 변수 a의 값을 데이터 서버의 변수 gradient로
// 보낸다.
bind $a + 10 to gradient2;
// 변수 a의 값에 10을 더해 데이터 서버의 변수
// gradient2로 보낸다.
bind $a/3 to gradient3; // 사칙연산도 가능하다.
bind $a * sin(45) to gradient4;   // 삼각 함수도 가능하다.
```

- if else

일반적인 if else와 같다. 중첩된 if문도 가능하고, 조건식에는 수식도 가능하다.

```
if ($ival > 15)                    // 조건식
{
        // code here...
```

```
}
else
{
        // code here...
}
```

- while

이 명령어 또한 일반적인 while과 같다. 조건식에 수식도 가능하다.

```
while ($count <= 8)        // 조건식
{
        // code here...
}
```

- repeat

repeat문은 인수로 주어진 정수 만큼 무조건 반복한다. 예외 상황이 발생할 수 있으므로 try catch문이 필요하다. 수식도 또한 가능하다.

```
repeat 10                  // 반복횟수
{
        // code here...
}
```

- try catch

일반적인 try catch문과 같다. Exception 발생이 예상 될 경우 try catch문으로 감싼다. find, move, goto, repeat, get 등과 함께 쓰인다.

```
try
{
        // code here...
}
catch
{
        echo $$exception;      // 예외 발생시 예
```

6. 결론

본 연구는 파일랩퍼에 있어서 가장 중요한 부분인 파서 에이전트를 설계하고 구현하였다.

이를 이용하여 에이전트 시스템에 있어서 파일랩퍼를 구현하여 수행해 보았으며 스크립트 언어는 어렵지 않으며 이를 이용하여 손쉽게 스크립트 언어를 작성할 수 있다.

참고문헌

[1] 한국전자통신연구소 인공지능 연구실, 이형 분산 환경에서 에이전트들간의 이질성과 분산성을 극복하기 위한 멀티 에이전트 기반구조, 1996.

[2] 최중민, 준구조화된 정보소스에 대한 지식기반의 Wrapper 학습 에이전트, 2002.

[3] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hill, 1995.

[4] Christine Guilfoyle and Ellie Warnet, Intelligent Agents: the New Revolution in Software, p.214, Ovum Ltd, London, 1994.

[5] Lee Jaeho, An Agent-Oriented Approach to the Development of Distributed Software Systems, The Univ. of Seoul, Korea.