知識ベース管理システムｋａｐｐａ
－論理型言語でのテスト－

3M－7

小林正彦＊　　河村元共＊＊　　溝口徹夫＊＊＊
＊ビップ・システムズ（株），　＊＊（財）新世代コンピュータ技術開発機構，
＊＊＊三菱電機（株）

１．はじめに

Ｋａｐｐａの試作開発の一環として、ＰＳＩマシンでの論理型言語ＥＳＰによるソフトウェア開発における成果の機能検査について報告したい。

試作開発とは言いながら、相当量のプログラム開発を行なうわけであり、また従来のプログラム開発とは異なった観点が存在しうるので、出来上がったプログラムの機能実現の確認、信頼性の確保には、ある方針をもって臨まなければならないと言える。Ｋａｐｐａの試作開発におけるソフトウェア工学的見地からの報告を行なうのが本文の主旨である。

まず、開発環境と検査環境の事情を述べ、本開発におけるプログラム開発過程、検査過程について概説する。次に、検査の実際方法と発見された不具合の性質や検査方法及び開発環境の長所・短所について述べる。

２．開発環境と検査環境

Ｋａｐｐａは逐次型推論マシンＰＳＩのＯＳであるＳＩＭＰＯＳのもとで提供される論理型言語ＥＳＰで作成される。ＥＳＰは、論理型言語Ｐｒｏｌｏｇをベースにオブジェクト指向を導入した、知識表現言語系システム記述言語である。Ｐｒｏｌｏｇが言語仕様としてはプログラム・モジュール化機能を用意していないのに対して、ＥＳＰはクラスと呼ばれるモジュール単位でプログラムを記述することが出来る。また、言語の記述性の高さと柔軟性、および対話型システムである強力なデバッガを使用することでプロトタイピングも容易になる。本開発では、これらの機能およびオブジェクト指向の特徴である継承機能などを活かしプログラム開発の効率化を図っている。

検査においても、前述のような特徴はそのプログラム作成、あるいは検査そのものを容易なものとしている。特にモジュール化機能、ＳＩＭＰＯＳのデバッガは有用であった。

ＳＩＭＰＯＳのデバッガは、プログラムの対話的な実行が可能である。このことは、被検査コマンドを直接デバッガから実行し、使用方法の確認などのプロトタイピング的なことを行ない、検査プログラムの作成に役立った。また、前後の流れに関係なく単発的に実行できるコマンドは、プログラムを作成せずに検査することも出来た。さらにトレース中のオブジェクトの状態や複雑なデータ構造の中身を調べることの出来るインスペクタの機能のため、コマンドの実行状態や不具合の原因の発見が容易であった。

検査プログラム作成では被検査コマンドの実行とその結果を出力するために従来、図２．ａ）のようなプログラム記述を行なっていたものが、クラスを分けることにより図２．ｂ）のような記述となる。このような記述をすることにより、検査プログラムの流れもわかり易くなり、被検査コマンドのｓｕｃｃｅｓｓ（成功）、ｆａｉｌ（失敗）なども意識しなくても済むようになった。また、検査が進みその時点での各コマンドの実行結果を調べなくてもよくなった場合には、図２．ｃ）のようにコマンドの第一引数を変更し、検査対象を変えるだけで不要な出力を省くことが出来る。これは、検査プログラムの作成だけでなく、そのデバッグも容易にしている要因である。

```
a)    :get(Kappa,Table_name,Fmt,Pos,Read_id,Rec),
                                        被検査コマンド
      :putf(Window,Output_fmt,[Fmt,Pos,Read_id,Rec]),
                                        結果出力
      :get(Kappa,Table_name2,Fmt2,Pos2,Read_id2,Rec2),
      :putf(Window,Output_fmt,[Fmt2,Pos2,Read_id2,Rec2]).

b)    :get(#test_kappa,Table_name,Fmt,Pos,Read_id,Rec),
      :get(#test_kappa,Table_name2,Fmt2,Pos2,Read_id2,Rec2),

      class test_kappa has
      :get(Class,Table_name,Fmt,Pos,Read_id,Rec) :-
            (
              :get(Kappa,Table_name,Fmt,Pos,Read_id,Rec),!,
                  :putf(Window,Output_fmt,[Fmt,Pos,Read_id,Rec])
              ;
                  :putf(Window,"Fail -> get/6\n"),
                  :show_status(Obj,Window)
            );
      end.

c)    :get(Kappa,Table_name,Fmt,Pos,Read_id,Rec),
      :get(#test_kappa,Table_name2,Fmt2,Pos2,Read_id2,Rec2),
```

図２．検査プログラムの例

３．共有オブジェクトの利用

本開発では、カーネル部、インターフェイス部のほかにステータス・オブジェクト（ｋａｐｐａ＿ｓｔａｔｕｓ）というものを用意した。これは、Ｋａｐｐａ使用中に起きる様々な現象を一括して管理できるものである。Ｐｒｏｌｏｇのような論理型言語ではｆａｉｌしたときの情報を得ることが難しいが、オブジェクト指向を取り入れたＥＳＰでは、オブジェクトを使用することで容易にそれを実現できる。

例えば、エラーの起きた状態や、評価で行なう処理時間の測定などの情報を蓄え、それを使用者が参照できるようになっている。各モジュールは、図３に示すようなコマンドを使用するだけで、情報の蓄積や分類は、ｋａｐｐａ＿ｓｔａｔｕｓが行なう。

```
:set_status(Status,Manager,Method,Error_infor).
:set_status_path(Status,Manager,Method).
:set_status_time(Status,Manager,Time).
```

図３．ステータス・オブジェクトのメソッド

これによりコマンドの使用方法の誤りなどのミスの発見が容易になり、本開発の効率は向上した。

Knowledge Base Management System Kappa
-Testing in Logic Programming Language-

Masahiko KOBAYASHI*　Moto KAWAMURA**　Tetsuo MIZOGUCHI***
*BIP SYSTEMS Corp., **ICOT, ***Mitsubishi Electric Corp.

4．開発過程と検査法

一般的にプログラム開発過程は、図4に示すようなものである。

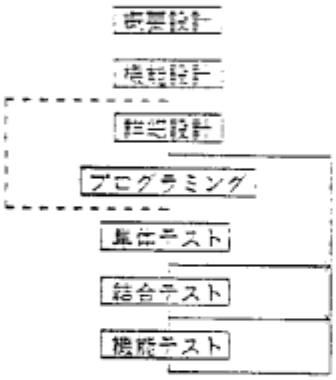

図4. プログラム開発過程

Ｋａｐｐａにおいても同様な過程を持つ。論理型言語ＥＳＰにおいても大規模なシステムを開発するには、従来の言語におけるような機能分割、モジュール分割プログラミングが必要となるためである。

また、本開発では詳細設計とプログラミングの差が少なくなっている（図4の破線）。この理由は、2節で述べた言語の記述性、プロトタイピングの容易さに基づくものと言える。

また、検査過程においては、単体・結合テストを各モジュール作成者、機能テストを検査要員が行なう形式を採っている。これは、本来あるべき形で当然とも言えるであろうが、本開発では検査対象のモジュールの内部処理について予備知識のない人間が機能テストを行なうことにより、要求された機能を正しく実現していることを確認している。

5．検査

Ｋａｐｐａの各モジュールは、ＥＳＰのプログラム（上位モジュールあるいは利用者）から利用できるコマンド（メソッド）提供している。従って、検査の実施方法は、下位レベルのモジュールに対しては上位レベルのモジュール、上位レベルのモジュールに対しては利用者が使用するのに近い形で実施する。ただし、検査という立場から、通常ありえないようなデータなどの異常系、正常系でも耐久テストなど通常使用しないような形でのテストも行なう。機能テストで発見される不具合のほとんどが、この2種類のテストで発見されている。理由としては、単体・結合テストを行なうモジュール作成者が、そのような異常データを予測していない、あるいは内部の処理内容を理解しているために、異常データを与えた場合にｆａｉｌすることのみを確認しているなどが考えられる。

発見された不具合の傾向としては、まずスペル・ミス、データ型のミス、各モジュール間におけるインターフェイスの不整合、修正ミスによる不具合などが挙げられる。

スペル・ミスに起因する不具合が発見しづらい理由としては、次のようなことが考えられる。まず、メソッドをやスロット名に誤りがある場合は、実行時に発見しやすい。しかし、変数に誤りがある場合のミスの発見は非常に困難なことが多い。変数の誤りには2つある。1つは、Ｔａｂｌｅ＿ｎａｍｅとｔａｂｌｅ＿ｎａｍｅのように先頭文字を小文字にしてしまう場合で、この時前者は変数、後者はａｔｏｍとなる。もう1つは、Ｔａｂｌｅ＿ｎａｍｅ，Ｔａｂｌｅ＿＿ｎａｍｅのようなどちらも違う変数と判断される場合である。

ＥＳＰでは、従来の言語と違い変数の型を意識しないで使用できる。このため、その時点では正常に動作するが、後にその変数を参照したときにエラーが発生する。従って、従来の言語ではコンパイル時やリンク時に発見できるエラーをＥＳＰでは発見しにくい。

また、インターフェイスの不整合については2つ理由が考えられる。1番目は、ダミーを用いてテストする際に、そのダミーと実際のモジュールの違いがもたらすものである。2番目としては、バージョン・アップに伴う仕様の変更などが、各モジュール作成者に正しく伝わっていないことがある。これは長期開発におけるバージョン管理、ドキュメンテーションの整理が正しくされていなかったことによるものである。

バージョン管理の不備は、修正ミスの原因にもなっている。誤って古いプログラムを修正したために、改修されていた不具合が再発することがあった。

6．デバッグ

3節で述べたように本開発では、ｋｅｒｎｅｌ内でエラーが発生した場合にそのエラーの情報をｋａｐｐａ＿ｓｔａｔｕｓというクラスに蓄え使用者にエラーの起こった状態を通知することが出来るようになっている。これは開発効率の向上につながったが、その情報をセットするメソッド名、あるいはその情報そのものの内容に誤りがあることがある。

メソッド名に誤りがあった場合は、実行時に発見できるが、エラー情報に誤りがある場合には、発見しづらい。デバッグ時に、その誤った情報により、プログラムの修正・デバッグを行なおうとするので、その効率は大きく低下してしまう。また、前者においてもエラー情報をセットするメソッド呼び出しに誤りがあり、そこでｆａｉｌした場合は、エラーの発生した下位モジュールではその情報がセットされず、バックトラックにより戻った上位モジュールで意味の不明な情報がセットされてしまう。これもまた、デバッグの効率の低下を引き起こすことになる。

また、論理型言語において、たびたび不具合を発生させる原因に、不必要なａｌｔｅｒｎａｔｉｖｅ（枝）が残っているためエラーが発生したときに正しくｆａｉｌしないということがある。

7．まとめ

ＥＳＰは新しい言語で、柔軟性があり、プログラムの記述が容易でプログラム開発の効率を上げることが出来る。しかし、プログラム書法がまだ未熟な部分もあり、それについて注意して置かなければならない。

また、バックトラックなどのＥＳＰの特徴も不必要な枝を残さないような注意をするなど、言語の持つ特性を熟知し設計を行なえば、開発効率の向上につながるであろう。

ＥＳＰのプログラムは、同じ処理をするのでも、1行違うだけで処理効率は著しく変化することがある。カット、ユニファイ、ｌｉｓｔの使い方などプログラム記述方法の確立、またその検査が、今後の課題となるであろう。

参考文献

[1] 溝口、横田、内田 "知識ベース管理システムＫａｐｐａ－データベースから知識ベースへ" 情報処理学会第35回全国大会、3N-3.1987