ソフトウェア工学 58－4
（１９８８．２．４）

データ駆動型テストの可能性について

田中　博明

（株）東芝　システム・ソフトウェア技術研究所

現在ソフトウェアのテスト技術は大きく静的テスト、機能テスト、構造テストの各分野で活発に研究開発がなされている。しかし、その個々の技術ではソフトウェアのエラーを完全に除去することはできず、これらを組み合わせて利用することが実際には行われている。

ところが、こういった組み合わせテストの技術は今まであまり活発に研究されてはいなかった。本報告ではいくつかのテスト技術を組み合わせて行うテストシステムを考え、その実現の一方法としてテスト関連データベースであるテストストレッジ、並びにデータ駆動型テストシステムを提案し、その利点について論じる。本システムを実現することにより各テスト実行単位は連動して動作することが可能となり、また今後の各ツール開発に即したテストの運用が可能になる。

"The Possibility of Data Driven Test System"
(in Japanese)

Hiroaki TANAKA

Systems and Software engineering laboratory,
Toshiba corporation
70 yanagi-cho saiwai-ku kawasaki,kanagawa, 210
Japan

Currently many researches and developments about software testing aim at static testing, functional testing and structured testing. But none of the techniques can find software errors completely. In fact all software developers use all of these techniques in combination. In this paper I propose test storage, a database for testing and a data driven test system that can combine any kind of testing methods efficiently, and discuss how it can be constructed and how it acts effectively for finding software errors.

1．はじめに

近年、ソフトウェア危機が叫ばれ、ソフトウェア生産性の向上と共にその品質、信頼性の向上が急務とされている。そしてエラーを除去してソフトウェアの信頼性を向上させるための手法として種々のテスト方法が考案され、それらは実際のソフトウェア生産の現場で利用されつつある。しかし、これらのテスト方法はそれ一つで全てのエラーを取ることはできず、いくつかのテストを組み合せて初めて効果的なエラーの除去が可能である。

本報告ではエラーを高精度で速く除去するテストの工程は多くのテストの組み合せで構成されているとの観点に立ち、それらが相補完して動作するにはテストに関連したデータを一元管理し、テストはそのデータに着目して起動される形式、いわゆるデータ駆動型のテストがより有効であることを示す。

2．テストツールの現状と問題

テスト作業には大きく分けて

- 静的テスト
- 機能テスト（動的テスト）
- 構造テスト（動的テスト）

の３つがあり、それらの特に動的テストを支援する基礎技術として

- 動的テスト実行の為の実行環境並びにユーザ・インターフェースの作成
- 再テスト支援とそれに必要なオペレーションの収集
- テストケース作成支援
- 動的テスト実行実績（カバレッジ）の測定ツール

といったテーマがある。これらは古くから提案され、ツール化されてきたテーマであるが未だ発展途上にあり、新しい考え方やその支援ツールが次々と提案されている。当社においてもテスト実行環境自動作成ツール[1] やテスト実行支援ツール[2] 、再テスト支援ツール[3] 、実行テストカバレッジ測定ツール[4] などいくつかのツールが作成されており、それぞれの用途に応じた使い方がなされている。

しかし、これら個々の技術なりツールはそれぞれ利点もあるが欠点も併せ持っており、実用的なエラーの除去を行う為にはこれらを組み合せて用いる必要がある。代表的な組み合せテストの例として機能テストと構造テストを組み合せて動的テストを行う場合を示す。

（１）プログラムと仕様が与えられる。

（２）仕様を見てそのプログラムが本来実行しなくてはならない機能を見付け、それに該当するテストケースを何本か作成してこれらについて動的テストを行う。（機能テスト）このとき（対象プログラムがそれ程大きくないならば）同時にパスカバレッジを収集しておく。

（３）プログラムには本来実行するべき機能の他に入力データの異常に対処する機能や特殊なオプションによるサービスなどの機能が備わっているのが一般的である。そこでソースコードを見て未だ通過していないパスの持つ意味を考え、これをテストケースに翻訳して動的テストを行う。（構造テスト、実際にはＣ１レベル程度を行う。）

（４）（３）のテストでカバレッジがそれ以上にならなくなっても(100% とは限らない。）仕様のある機能に対する処理をまるごと落としている可能性が存在する。これをチェックするため、今度は仕様書の細かい点にも着目し、また仕様書の裏を読むような形でテストケースを起こして実行テストを行う。（機能テスト、通常言うところのいじわるテスト）

このように構造テストと機能テストを組み合せて互いの特徴を生かしたエラーの除去が実現される。このほかにも静的テストと動的テストを組み合せたテストや機能テストの一部に構造テストの要素を取り入れた複合テスト、開発の他の段階で用いたテストの結果を利用するテストの形式なども提案され、エラー除去の可能性を高くしたりテストの回数や手間を減少させる効果を上げている。

こういったテストの複合領域の技術は今後も技術の進歩、ソフトウェア開発手法の変化に応じて大きく変化していくと思われる。そして今後はますますこういったテストの組み合せが一般化してくると考えられる。このときまず問題となってくるのはそれらテストの間でのデータの互換である。たとえば前記の組み合せテストにおいてもテスト実行のカバレッジ収集は２つのテストを通じて共通に行われなくてはならないし、構造テストの時に実行したテストのテストケースはそれに続くいじわるテストにおいて既に実行したことがわからないと、同じようなテストを再び実行してしまう可能性もある。一般的にはより多くのテストを組み合せた場合はより多くのデータが互換性を持つ必要がでてくると考えられる。

また、プログラムのデバッグ後の再テスト支援もまた問題となってくると思われる。現在再テストを支援するためテスト時の入力のログ収集とこれを用いたプログラムの半自動実行のツールが存在するが、これはそれまでに行った全てのテストの入力データを記憶するものであり、そのなかのどのデータを用いて再テストを実行すればよいかを示唆してくれるものではない。また、デバッグの結果修正されたプログラムとおそらくはその仕様に対してどのテストを実行するのが最も効果的かを示唆してくれるツールがない（こういったツールがもしあれば、おそらくそのツールはそれまでに行ってきた実行テストの入力データが仕様上

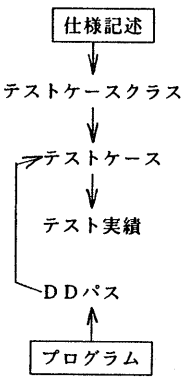
に占める位置、そのテストの実績としての通過パス、テストの結果を調べ、それらに対して総合的に判断を下すような形式となるであろう。）ため、再テスト時にもオペレータの高度な判断と幾つかの余分な（新たなエラーが発見されないことが証明できる）テスト実行を強いられている。また、そのテスト自体に見落としがあった場合、デバッグが新たなエラーを埋め込んでしまう可能性もある。

さらにはそれらのテストのどれから実行するのが最もエラー発見に効果的かを決定する戦術に欠けるという問題もある。前記の例では機能テストを２度実行しているが、この方法はあくまで実行テスト順序の可能性の一つであり、実際のテスト実行は担当者の流儀等に任されている。前述のデバッグ後のテストと同様、現段階でどのテストを行うことが可能であり、またどれが最も効果的かを示唆してくれるようなテストシステムがより望ましい。

3．テストストレッジ

これらの問題点のいくつかは仕様、プログラムをはじめテストケースやテスト実績など、テストに関連しまた他のテストツールが必要とする情報を一元管理するようなデータベースを作成し、そのデータフォーマットを明らかにして、各テストツールがそのフォーマットに基いてデータを入出力するすることで解決できる。そしてこれらの情報間に情報の修正が他の情報を無効にする、あるいはその対応関係を再確認する必要を生ぜしめるような依存関係を示す情報の関連を設け（変更依存グラフと名付ける。）、この関係を常に保つように管理する。報告者はこのデータベースにテストストレッジという名前を付けた。図１にテストストレッジの簡単な例として単一モジュール（サブルーチン）の仕様とプログラムに関するテストストレッジの変更依存グラフを示す。ここでテストケースとは一回の実行テストに必要な動作環境、入力データ、並びにその動作と出力の予測の組（の集合）である。また、テストケースクラスとはモジュールがだいたい同じ動作をするようなテストケースを生み出す動作環境並びに入力データの張る空間を仕切ったある領域（の集合）である。一般にモジュールの仕様に対してテストケースクラスは複数個生成され、その一つ一つのクラスに対してテストケースはさらに複数個生成される。

この図より、例えばテストケースはテストケースクラス、ＤＤパスのいずれかに修正があった場合更新され得るということがわかる。（ＤＤパス：条件分岐を内包しないプログラム上の処理の単位、decision-to-decision path ）



□ は独立データ（オペレータによって変更される）
その他は依存データ（テストシステムによっても変更される）

図１　テストストレッジ（変更依存グラフ）の例

4．データ駆動型テストシステム

テスト実行の順序に関する問題点ついては個々のテストをテストストレッジ中の変更依存関係に対してその依存関係の元となるほうの変更を監視していて、変更が起こった時起動するような形すなわちデータ駆動型テストシステムとしておくと解決することができる。

図２に図１で紹介したテストストレッジ上で動作するデータ駆動型テストシステムの構成を示す。ここで図の左側には主にテストストレッジの内容である情報の種類を、右側にはデータ駆動型テストの各実行単位がツールの形で配置されている。矢印はデータの流れを表している。テスト

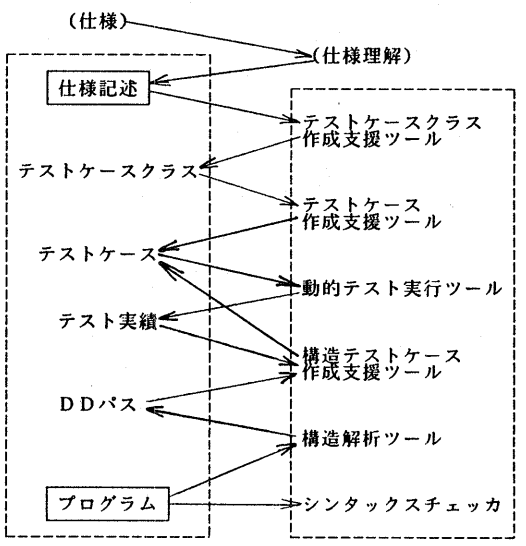


図２　データ駆動型テストシステムの例

実行単位へ入る矢印はその実行単位が変更を監視している、または処理に必要なテストレッジの内容であり、テスト実行単位から出ている矢印はそのテストストレッジの変更依存関係を（テスト動作などによって）修復することを意味している。

同時にテスト実行単位は新旧二つの内容を比べる必要があるため、テストストレッジの内容はテストケース、テスト実績を除いて二重とする。

図2の動作を説明する。

1） テストストレッジの生成直後はその全ての内容は空である。

2） モジュールの仕様がオペレータによって与えられるとテストストレッジはまずその仕様を理解しようとする。（この作業はテスト担当者自身によってなされるかもしれない。）この仕様の理解によってテストストレッジはモジュールの仕様を計算機に理解できる形で格納することができる。

3） テストケースクラス作成支援ツールは常に仕様記述が更新されたかどうかを見張っているが、2が終了した段階で仕様記述が更新されたことがわかる。（仕様記述の古い内容は空である。）従ってこのツールは動作を開始し、入力された仕様記述の新しくなった部分に対して対応するテストケースクラスを生成（更新）する。

4） テストケース作成支援ツールはテストケースクラスが更新されたかどうかを見張っている。3の出力である新しいテストケースクラスの内容はその更新された部分の各々に対応した複数のテストケース作成支援ツールを起動し、対応するテストケースを生成していく。

5） 動的テスト実行ツールはテストケースとそれに対応したテスト実績の通過パス実績を監視しており、テストケースに対して通過パス実績が無いか、あるいは通過パス実績の表記がＤＤパスと対応が取れなくなるとそのテストケース毎に起動される。その結果はテスト実績に格納され、この一連のテスト作業は完了する。動的テスト実行ツールはテストケースの更新のみならずテスト実績の記述もチェックすることでプログラムのデバッグに伴う再テストを自動的に実行できるようになっている。

ここまでは本テストシステムの主に機能テストに必要な各実行単位をテストケースクラス作成、テストケース作成、実行テストとひと続きの形で説明してきた。しかし、ここまでの内容だけだと通常の制御型テストシステムの動作とほとんど変わらない。（もちろん、同時に並行して動作する実行単位に関する記述はなされているが。）、ところがこのシステムの別の部分には構造テストに関連したこれと並行するもう一本のテストの流れが存在する。この流れについて6）以降で説明する。

6） 動的テスト実行ツールもそうだが、このテスト実行単位が起動するには仕様に対応したプログラムが必要である。シンタックスチェッカはプログラムが更新されるとその更新された部分に対して起動され、その部分の記述並びに他の（変更を受けていない）部分との繋がりをチェックする。

シンタックスチェッカは一般に次に述べる構造解析ツールの一部として実現されるが、その役割は少し違う。シンタックスチェッカの出力はテストストレッジに反映されない。シンタックスチェッカの出力はそのプログラムが文法に違反している、あるいはエラーである可能性の高い記述を含んでいる時は警告を出し、それらの記述を含んでいない場合は何も出力しない。普通シンタックスチェッカが警告を発するとオペレータは直ちにこれをエラーと判定する。

7） 同じくプログラムが更新（入力）された時、その更新された部分に対して構造解析ツールが起動され、その部分を含むＤＤパスを更新する。このＤＤパスはまたテスト実績の通過済みパスを記述するための単位でもある。

8） 構造テストケース作成支援ツールはＤＤパスとそれまでのテスト実績とを監視しており、通過実績を持たないＤＤパス（複数も可）に対してそれを通過するために必要な入力の条件を求めてテストストレッジのテストケースを更新する。この出力は5の動的テスト実行ツールにより仕様記述から求めたテストケースと同様に実行される。

このように個々のテスト実行単位がそれぞれ受持ちの変更依存関係を監視していて、常にその対応を修復するように動作するテストシステムはとりもなおさずデータ駆動型のシステムである。

テストシステムをデータ駆動型と捉えることはテストストレッジの使用とあいまってさらにいくつかの利点を生む。以下にこのデータ駆動型テストシステムの利点を列挙する。

1） テストストレッジに新たなデータ、考え方、テスト支援ツールを組み込むことが容易である。

従来の方法ではテストの方法について新しい考え方が生れた時その考えをツール化するためには入出力データとその考え方に沿ったテスト支援ツールを作成する必要があった。データ駆動型システムとしておけば新規に利用するデータの記憶構造、修正の依存関係とそれにアクセスするテスト支援ツールを作れば、そのテスト支援ツールはそれまでに作成したデータをそのまま利用できるし、そのツールを作ったあと、それをいつ起動させるかについて考える必要が無い。

2） テスト作業の並列性が記述し易い

データ駆動型のシステムならばあたり前であるが、個々のテスト作業のうち入出力データが互いに独立であるようなテストは並列に行っても問題ないということが自ずと理解できる。このことはテスト作業を多人数で行う場合やテスト自動実行とオペレータによるテスト実行の並行作業など、従来の技術では明確に与えることが困難であった並行作業を体系的に記述できるため、大きな意義を持つ。

3） オペレータによる作業とツールを用いた作業の切り分けが柔軟である。

例えばそれまで殆ど人手に頼っていたテスト作業の一部をテスト支援ツールによって支援しようとする場合、そのツールの入力と出力に相当するデータを新たにテストストレッジに構築し、その間の作業をツール化すればよい。オペレータはその残った部分についてテスト作業を実施する。従ってしなくてはならないと考えたテスト動作はとりあえず対応するテストストレッジを組み上げ、テスト作業をオペレータに任せながら並行して支援ツールの開発を行い、出来た所までを順次テストシステムに組み込むといった柔軟な運用が可能になってくる。

4） テストストレッジのデータの修正に伴うテスト動作の波及範囲が明確になる。

個々のテスト技術に着目していると（木を見て森を見ないの例え通り）プログラムなりデータなりが変更を受けた場合、あるいはプログラムの仕様が変更された場合その影響がどこに及び、またその影響の範囲内でエラーが発生しているかを調べるためのテストとしてどのようなものが必要になってくるかを明確に捉えることが難しい。テストストレッジにはこういった変更の波及範囲が明示されている（変更依存グラフの矢印を辿る事で実現する）ので、本システムにおいてはどのテスト実行単位が起動されるべきかは直ちにわかる。この効果は本章で紹介した再テストに関することだけではない。モジュールあるいはその仕様の変更はインターフェースの不一致を引き起こし、このモジュールを利用する他のモジュールのエラーをも引き起こす可能性がある。５章で説明するが、テストストレッジはインターフェースを介して互いに関連し合う複数のモジュール、プログラムの関連に対しても修正の依存関係を持たせることが可能であり、この機能を利用してモジュール修正に伴うエラーのモジュール間移転をもトレースすることが可能となる。

5．データ駆動型テストシステムの拡張

本章では３、４章で紹介したテストストレッジ並びにデータ駆動型テストシステムを複数モジュールから成るソフトウェアに対応できるよう拡張して、ソフトウェア開発の種々の側面でテストを支援出来るようにすることを考える。

データ駆動型テストシステムをその用途に応じて便宜上三つに分ける。

- 単一のモジュールを対象としたデータ駆動型テストシステム
- モジュール同士の（引数、戻り値、外部変数などを介した）リンクを対象とするデータ駆動型テストシステム
- 単体テスト→結合テスト　といったプログラム開発の縦のラインに着目したデータ駆動型テストシステム

このうち単一のモジュールを対象としたテストシステムは既に紹介したが、さらに実際的なテスト実行の概念を追加する。また４章でも紹介した通りデバッグ時に発生する仕様の修正の他モジュールへの波及についても記述する。また図１、２で紹介した例ではテスト実績の再利用については述べられていなかったが、これについてもテストストレッジ自体を再利用の対象として明確に位置付ける。

これらの点を考慮し、拡張したテストストレッジ（変更依存グラフ）の形態を図３に示す。

図１と図３を比較して増えている内容は他のモジュールに関する記述（テストストレッジ）とスタブ、並びにその動作実績である。また、図１ではテスト実績とだけ書かれていた内容が展開されていくつかの具体的な実績名があらわれている。

テストストレッジはプログラム単位に対してただ一つ作られる。テストストレッジはただ１組の機能仕様とプログラムの記述欄を持つ（更新のチェックを考えると２組）。その他にこの機能仕様の中にサブモジュールがあり利用しているのならばそのサブモジュールに関するテストストレ

ッジを入れ子またはポインタ付きの外部ストレッジの形で持つ。あるプログラム単位からライブラリ部品など別途作られた機能単位を参照する場合も該当するテストストレッジをポインタを通して参照する形式でテストストレッジどうしのリンクを取ることが可能である。

上位のプログラムから見たときそこに含まれる部分機能はある高度な処理を行う１ステートメントとして捕えることができる。本テストストレッジにおいても部分機能に関する記述はそのレベルのプログラムと同等の取り扱いをすることで上位のテストストレッジに組み込むことに成功している。実際、部分機能の修正は上位プログラムではＤＤパスの変化という形でとらえ、再テストを促す形式となっている。

スタブは従来実行環境の一部として捕えられ、テストストレッジのようなテスト実行の関係情報としては考えなかったのだが、ここでは特に単体テストと結合テストとの整合を考える必要からスタブを機能仕様と１対１に対応する、プログラムの影武者として捕えることにした。同時に上位モジュールから見たスタブの実行はその動作にオペレータの介入を要することを除いてあたかも実際にプログラムが存在するかのようにし、またその動作の履歴をも収集することにする。この履歴はスタブとして同じ動作を要求されたときオペレータの手を煩わさずにすませるためと、対応するプログラムが出来たとき、スタブ動作と同じことを実際のプログラムにさせることで簡易結合テストを行うために用いられる。

次にこの拡張テストストレッジ上で動作するデータ駆動型テストシステムを図４に示す。

ここで図中に日本語で記述してあるものは図３に説明したテストストレッジの内容、○で囲った番号はテストの処理単位を、また矢印は主に処理単位が参照し、修正するデータの流れを示している。処理単位については図の下部に凡例を載せている。

図４で新たに追加されたテスト実行単位はスタブ生成ツール、スタブ実行支援ツール、並びに簡易テストケース生成ツールの３つである。また、図２で既に紹介したテスト

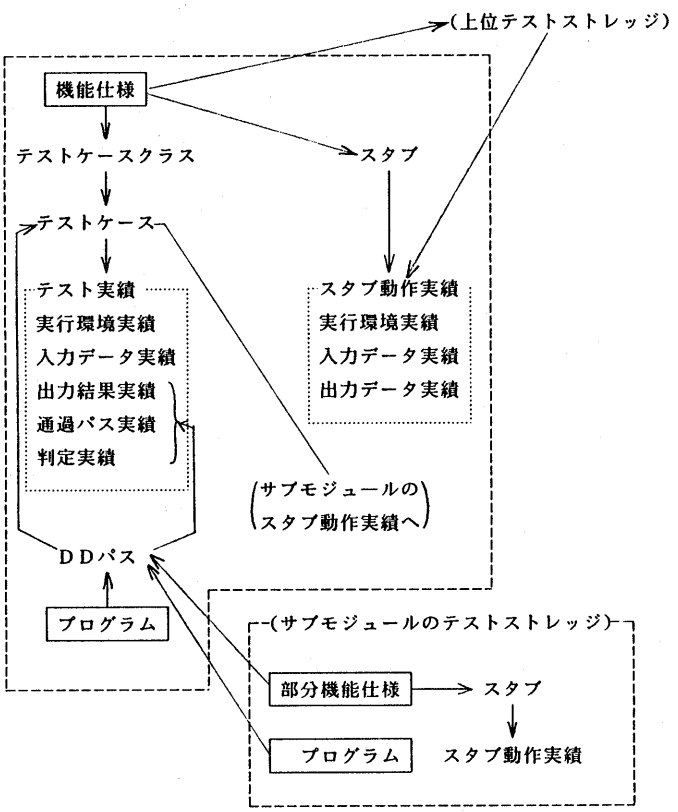


図３　拡張テストストレッジ（変更依存グラフ）

実行単位も機能がより明確になるに従って扱うデータが複雑になっている。以下にこれらの変更点について説明する。

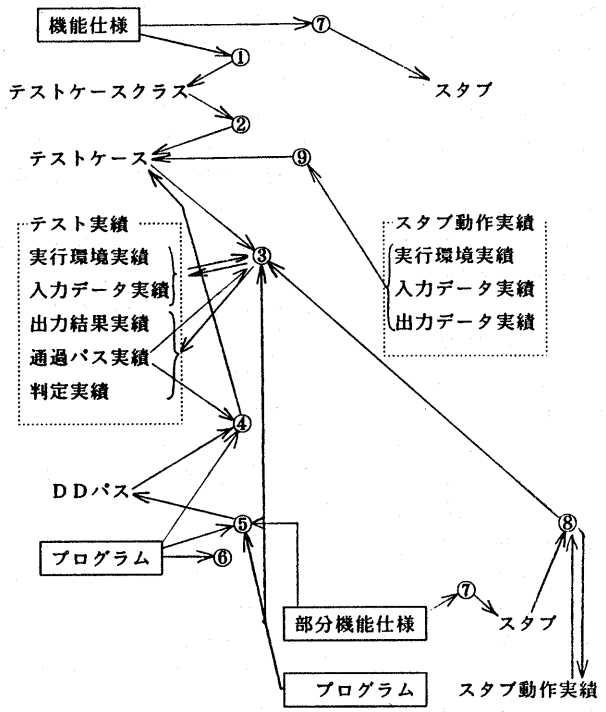
1）テスト実行支援ツールは新たにテスト実績の実行環境実績、入力データ実績を入力とする。これは一旦行ったテストをもう一度行うとき、オペレータに余分な負担をかけないため、そのときの状況を再現するために必要である。

またテスト実行支援ツールはテスト実績の通過パス実績とＤＤパスを入力とする。これはテストの実績を持つプログラムが修正を受けたとき、テスト実績のうち通過パスの一部に修正を受けた部分があるものについては再テストを行うために必要である。

さらにテスト実行支援ツールはサブモジュールのテストストレッジの部分機能仕様を参照し、またそのスタブを動作させるスタブ実行支援ツールを利用する。これらは各々結合テスト、単体テストを行うときそのサブモジュールの動作を決定するため利用される。スタブ実行支援ツールはテスト実行支援ツールからはサブモジュールとして見えるものであり、その間の矢印は他の矢印のような単純なデータの流れではない。

2）構造解析ツールは新旧のプログラムの他にサブモジュールの部分機能仕様並びにそのプログラムも入力とする。その結果サブモジュールの仕様、プログラムの更新の内容は逐一本テストストレッジに報告され、構造解析ツールによってＤＤパスに反映される。

3）スタブ生成ツールはサブモジュールの部分機能仕様



① テストケースクラス作成支援ツール
② テストケース作成支援ツール
③ テスト実行支援ツール
④ 構造テストケース作成支援ツール
⑤ 構造解析ツール
⑥ シンタックスチェッカ
⑦ スタブ生成ツール
⑧ スタブ実行支援ツール
⑨ 簡易結合テストケース生成ツール

図４　拡張テストストレッジに対応した
データ駆動型テストシステム

をもとにスタブを生成する。

4）スタブ実行支援ツールは1）でも説明したが、テスト実行支援ツールのサブモジュールとして動作する。そして未完成のサブモジュールの動作をオペレータの手を借りながら実行する。同時にその実績を覚えておいて、同じ動作のリクエストが来たら2回目からはオペレータの手を借りずに動作するようにしておく。

スタブ実行支援ツールをテスト実行支援ツールと一体にしなかったのは、この機能はサブモジュールが出来ると不要になるためとテスト実行支援ツールはその多くの動作を自動で実行できるのに対してこの機能は実行に際して頻繁にオペレータの手を借りるので単一の機能として扱いたくなかったからである。

5）簡易結合テストケース生成ツールはそのテストストレッジのプログラムが出来上がる前に上位モジュールの単体テストが行われていた場合、その実績通りにこのプログラムが動くかどうかを調べることで疑似的に結合テストに相当するテストを行うことを目的としている。但し、テストの実行はテスト実行支援ツールに任せ、ここではただその実績をテストケースに焼き直すことだけを行う。

次に5章の先頭で分けた三つのデータ駆動型テストシステムのうち、モジュール同士のリンクのテスト、プログラム開発のラインに沿ったテストが図3並びに図4の中にどのように反映されているかを見る。

図3の説明にあったようにテストストレッジは機能単位（ここではモジュールを意味する。モジュールは上位、下位を問わない。）毎に一つずつ作られる。そしてテストストレッジの間にはスタブの動作、プログラム、並びに部分機能仕様の変更を知るためのポインタを張ることができる。

あるモジュールの仕様またはプログラムが修正されたとき、そのモジュールを参照している他のモジュールの構造解析ツールは一斉にその変更を察知する（どのように変更されたかはここでは問わない）。そしてそのモジュールを参照しているＤＤパスに変更があったことを記す。これをもとにテスト実行支援ツールならびに構造テストケース作成支援ツールはそれぞれの立場からこのモジュールにとって動作の保証に必要と思われるテスト（もとの機能の確認と新規機能への対応テストケースの生成）を実行する。構造テストケース作成支援ツールの出力は新規のテストケースなので、これをもとにさらにテスト実行支援ツールが起動される。このプロセスはモジュール同士のリンクのテストも開発のラインに沿ったテストも同様である。但し開発のラインに沿ったテストで、スタブ動作実績を持つモジュールの仕様が更新された場合は、その実績の一部が無効になるため該当部分の消去が必要になってくる。

6．まとめと今後の課題

本報告では個々のテスト実行単位を連結動作させる一手法について紹介した。その特徴は1．テストストレッジは新たなテストの概念を柔軟に取り入れることができる、2．テスト作業の並列性を簡単に記述できる、3．オペレータによる作業とテストツールによる作業の親和性がよい、4．テストストレッジのデータ修正に伴うテスト動作の波及範囲が明確に捕えられる、といった点にある。テストストレッジの細い内容、テスト実行単位とその仕様については今後の考察によりさらに実際のテストに即した内容、体系的にまとまった内容にしていく必要があると考えている。今後はまずこのアイデアを評価する手段としてのプロトタイプの完成を目指す予定である。

具体的に今後調査検討すべき技術的点を以下に挙げる。

- テストストレッジの実際の構築方法
- プログラムのサイズに対するテストストレッジのサイズ、サイズの削減とそれによって不可能となるサービスの関係の調査
- プログラム開発関連のデータベースとのリンク方法
- テスト実行単位の待機方法と複数の処理単位の効果的運用方法
- デザインレビューのテストストレッジへの取り込み

こういった点も今後順次解決していく予定である。

参考文献

[1] 田中
Ｃ言語用テストベッドｍｋｓｔｕｂの開発
情報処理学会第３５回全国大会　3Y-1 s62

[2] 竹沢他
ソフトウェアテストと品質保証
東芝レビュー　1983 Vol.38 No11 pp974

[3] 熊谷他
テスト自動化ツールの実現
情報処理学会第３３回全国大会　4G-7 S61

[4] 中島他
Ｃプログラムのためのモジュールテスト環境
ソフトウェア工学研究会　45-4 S60