ソフトウェア工学 124－7
（1999．10．22）

スライシングを用いたフレームワークの再構成支援技法

施 暁波[†] 小野 康一[‡] 深澤 良彰[†]

[†] 早稲田大学 理工研究科 情報科学専攻
〒169-8555 東京都新宿区大久保 3-4-1

[‡] 日本アイ・ビー・エム株式会社 東京基礎研究所
〒242-8502 神奈川県大和市下鶴間 1623-14

フレームワークは、その利用において要件に合わせて改変されることがある。一方、その再利用性からフレームワークはアプリケーション開発の共通基盤でもあるので、改変の影響が既存のアプリケーションに及ぶことは避けなければならない。本論文では、プログラムのスライシングとプログラム変更に対する正当性検証を用いたフレームワークの再構成支援技法を提案する。本技法によって、フレームワーク改変の影響が波及しないように、そして元のフレームワークの再利用性をなるべく損なわないようにフレームワークの再構成が実現できる。

A framework reconstruction technique using program slicing

Xiaobo SHI[†] Kouichi ONO[‡] Yoshiaki FUKAZAWA[†]

[†]Department of Information and Computer Science
Waseda University
3-4-1, Okubo, Shinjuku-ku,
Tokyo 169-8555, Japan

[‡]Tokyo Reaerach Laboratory
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi,
Kanagawa, 242-8502, Japan

In some cases, a framework must be changed to satisfy its application requirements. On the other hand, it is also the common base of the application development because of its reusability. Therefore, the influence of the change must be localized in order not to spread to other existing applications. This paper proposes a technique which realizes the framework reconstruction by using program slicing and verification of correctness. Utilizing this technique, the high reusability of framework is kept as much as possible.

1 はじめに

フレームワークはオブジェクト指向によるソフトウェア再利用へのアプローチの一つであり、「抽象クラスの集合とそのインスタンス間の相互作用によって表現された、システムの全体または一部の再利用可能な設計」[1]と定義されている。

フレームワークを用いて新しいアプリケーションを開発する場合には、必要な抽象クラスを実装したり、仮想関数をオーバライドしたり、新たなクラスを付加したりする。その対象となるクラスをホットスポット (hot spot) と呼び、また、常に固定的な処理を行なう部分をフローズンスポット (frozen spot) と呼ぶ。

フレームワークに基づいてアプリケーションを開発する際に、ホットスポットだけでなく、フローズンスポットも変更する必要がしばしば生じる。その理由として、以下が挙げられる。

- フレームワークの開発者のドメイン分析不足で、フレームワークとして十分に洗練されていない。

- 開発するアプリケーションに最適なフレームワークが存在しないため、ドメインの類似するフレームワークを改造して使用する。

フレームワークは再利用されるソフトウェアであるため、複数のアプリケーションプログラムにおいて共有される。したがってフレームワーク自身の変更は、それらのアプリケーションプログラムの全てに影響を与える可能性がある。

フレームワークに変更を加える際には、そのフレームワークを使用している全てのアプリケーションプログラムに影響を与えないように行なう必要がある。本研究は、フローズンスポットの変更に対してアプリケーションプログラムに影響を与えないフレームワークの再構成手法の確立を目的としている。このために、元のフレームワークの意味を保った上で再利用性を損ねないように再構成する手法を提案する。

2 本研究の概要

2.1 対象となる問題

フレームワークを用いて開発するソフトウェアを以下ではアプリケーションと呼ぶ。個々のアプリケーションには仕様があり、それぞれの仕様を実現するために作成される個々のプログラムを個別の実装 (implementation) と呼ぶ。したがって実装は、フレームワークのホットスポットに埋め込まれる関数やクラスで構成される。

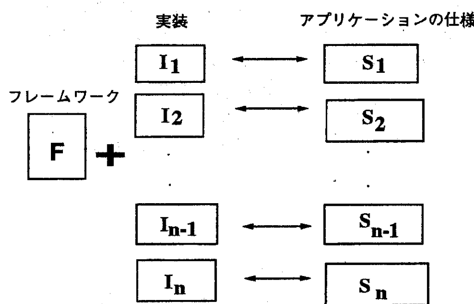

図 1: フレームワークとアプリケーション

図1のように、あるフレームワーク F をベースに、n個のアプリケーションが開発されているとする。それぞれのアプリケーションについては仕様 $S_1 \ldots S_n$ が与えられ、それに基づいて F に対する実装 $I_1 \ldots I_n$ が作成されている。保守担当者はフレームワーク F のソースコード, アプリケーションの仕様 $S_1 \ldots S_n$, 実装のソースコード $I_1 \ldots I_n$ を入手できると仮定する。

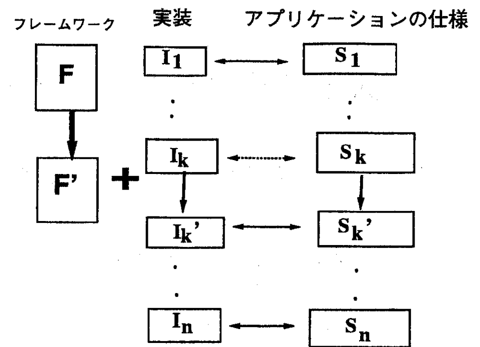

図 2: 一度変更されたフレームワーク

この状況において、アプリケーション A_k の仕様 S_k が S_k' に変更されたと仮定する。フレームワーク F が十分吟味して作られ、この変更を I_k の変更だけで吸収できるのであれば、他のアプリケーションへの影響は出ないし、またこうなっているべきである。しかし、I_k を変更するだけでは S_k' を満たすことができない場合には、F も変更しなくてはならない。つまり、F のフローズンスポットを変更する必要が生じる (図2)。今、フレームワーク F のフローズンスポットに対して S_k' だけを満たすように変更を加えたものを F' と呼ぶ。

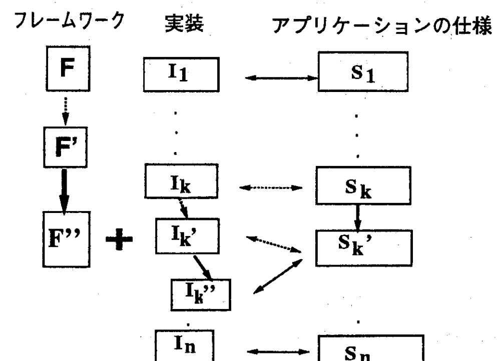

図 3: フレームワークの再構成の概要

図2において、$S_j(1 \leq j \leq n, j \neq k)$ は変更され

ないので、F を F' に変更してしまうと、A_k 以外のアプリケーションまで影響が波及し、仕様通りに動作しなくなることが考えられる。そこで、$F \rightarrow F'$ の変更の影響を及ぼさないようにするために、F' を再構成する。その結果作られたフレームワークを F'' とする。さらに、$F' \rightarrow F''$ の再構成に対応して、アプリケーションの仕様 S_k' を満たすように実装 I_k' から I_k'' への変更が必要となる (図3)。フレームワークの再構成 ($F' \rightarrow F''$) と、実装の変更 ($I_k' \rightarrow I_k''$) を自動化すると共に、その影響の範囲を最小限に抑えることを本研究において実現する。

2.2 スライシングを用いた再構成

元のフレームワークFのフローズンスポットにおける変更がメソッドMに対して行なわれ、その変更はメソッドの内容をmからm'に変えたとする。そこで、変更されたフレームワークF'の再構成を行なう。変更されたメソッドMに対応する仮想関数VMを新たに導入し、VMの内容をmとして定義し、そしてメソッドMの内容をVMへの呼出しと定義する。このような再構成を行い、仮想関数のままで実装をすれば、仕様変更のないアプリケーションの仕様を満たすことができる。また仕様変更のあったアプリケーション A_k についても、実装 $I_k' \rightarrow I_k''$ において、仮想関数VMのオーバライドを行ない、内容をm'に変更することによって、仕様を満たすことができる (図4)。

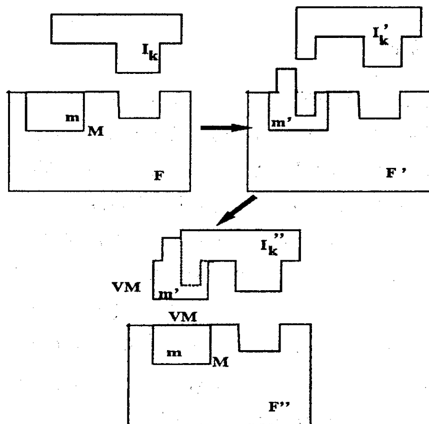

図 4: メソッド全体を仮想関数として再構成

このように、メソッド全体を仮想関数としてホットスポットに組み込まれることで、原理上はフレームワークの意味を変えずに再構成できる。しかし、このような方法は結果的に元のフローズンスポットの部分を減らすため、フレームワークを持っている再利用性を損ねてしまうという欠点がある。

これを改善するために、本研究では再構成の前にスライシングによるプログラム分割を行なう。再構成の対象となるメソッドMをスライス分割し、変更の影響を受けるスライスと受けないスライスに分類する。変更の影響を受けるスライスがその影響をさらにアプリケーションに波及し、アプリケーションの仕様を満たせなくなっているかどうかは検証によって確定させる。影響を波及させるスライスに対して、対応する仮想関数を新たに導入し、それ以外のスライスに対してはそのまま元のメソッドに残す。このように、スライシングを用いたメソッドの分割によって、仮想関数にするのをメソッド全体ではなく、必要最小限の部分にとどめておくことができる。すなわちフローズンスポットに残った部分が増え、再利用性をなるべく損なわずに済むというメリットが得られる。

2.3 プログラムの正当性の検証

メソッドのスライスがアプリケーションに影響を与えるかどうかは検証によって確定する。本稿では、検証履歴に基づいて、プログラム正当性検証技法 [2] を応用している。この検証においては、プログラムを抽象プログラム (Abstract Program; AP) で表現することとしている。APとは、プログラムと等価な論理式付き有限有向グラフである [3]。ただし、本手法では [2] において定義する表明つきAPを用いる。表明とは、プログラムのある状態 (表明つきAPにおいては節点で表現される) において成立しているべき条件のことである。プログラムの実行開始の状態 (入力節点) や実行終了の状態 (終了節点) に割り当てられる表明は、それぞれ入力表明、出力表明と呼ばれる。プログラム中の文は、表明つきAPにおいては矢印で表現される。したがって、表明つきAPの各矢印の文は、条件分岐における条件式 (テスト論理式) か、代入文に対応している。プログラムを表明つきAPで表現することで、プログラム構造は明確かつ一意に表現される。

本研究では、個々のアプリケーションに仕様が与えられ、変更前のプログラムが仕様を満たすことが、検証によって保証されているという仮定をおく。

図5に示すように、アプリケーションが仕様を満たしていることが以下の検証によって確認されているとする。各メソッドに入力表明と出力表明を割り当てる (ループ

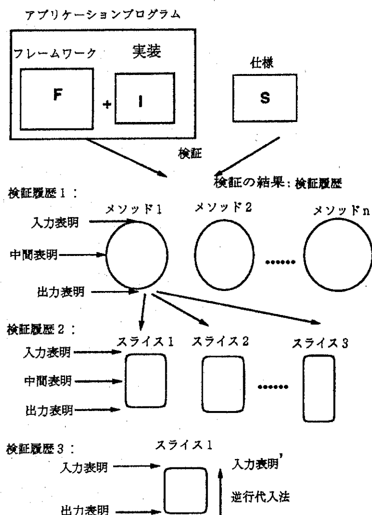

図 5: 検証と検証履歴

節点がある場合、中間表明も割り当てる)。入力表明と出力表明 (ループ節点がある場合、中間表明も) を合わせて検証条件と呼ぶ。また、各メソッドにスライス分割を適用し、個々のスライスに入力表明 A_{pre} と出力表明 A_{post} をメソッドの検証条件から割り当てる。 A_{pre} を前提として、スライス中の文を適用した結果が A_{post} を満たすことを導く。出力表明 A_{post} から逆行代入法により、 A_{post} が満たされるために入力節点で成立している必要がある条件 $A_{pre}*$ を求める。

検証の結果として、以上示した各メソッドの検証条件、各メソッドを分割したスライス、各スライスの検証条件、逆行代入法によって求めた条件 $A_{pre}*$ をそれぞれ得る。これらを総じて、検証履歴と呼び、フレームワークの変更以前に得られていると仮定する。

2.4 入出力

入力として、フレームワークのソースコード F と一度変更を加えた F', アプリケーション A_i の仕様 $S_i(1 \leq i \leq n)$ と変更された $S_k{}'$, そして A_i の実装 I_i のソースコードを用意する。スライスの検証に利用するため、フレームワーク変更前の検証履歴 H_k も入力とする。

出力は再構成されたフレームワークのソースコード F'' と、 F'' に対応して変更された実装 $I_k{}''$ である。

3 フレームワークの再構成技法

本手法はメソッドの分割、スライスの検証、フレームワークの再構成という 3 つの手順によって構成される。分割において検証対象をスライスに細分化し、検証において影響を与えるスライスを検出し、再構成において影響を与えるスライスに対して仮想関数を導入し影響の波及を回避する。図 6に手順の概略を示す。

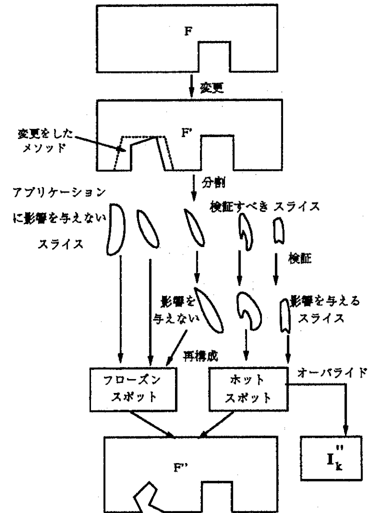

図 6: 本手法の手順

3.1 分割

変更前の検証履歴と変更後のメソッドを比較して変更の影響範囲を特定する。検証対象を比較する前にスライス分割技法を適用して比較対象を細分化し、変更の影響を受ける可能性のない部分を予め検証対象から除外する。

まず、変更されたメソッドをスライス群 S_n に分割する。ここでのスライスは通常の静的スライシングによって得られるスライスではなく、[2] で定義した表明つき AP の静的スライスである。区別のために S_n と表記する。スライス分割の基準変数は大域変数、クラスのメンバ変数、メソッドの返り値に含まれる変数を用いる。

分割されたスライス群 S_n の中から、フレームワークの変更の影響を受けないスライス群 $S_{n(noimp)}$ と受けている可能性のあるスライス群 $S_{n(imp)}$ に分ける。具体的な方法として、変更された文が代入文なら代入対象の変数に注目し、if,while 文なら文全体に注目し、注目される変数 / 文

を含まないスライスは $S_{n(noimp)}$ に分類する。それ以外のスライスは $S_{n(imp)}$ に分類する。

3.2 検証

$S_{n(imp)}$ に分類されているスライスは、変更の影響を受けている可能性があるだけで、確定していない。変更の影響の有無を検証によって確定させる。本稿では、検証履歴に基づくプログラム正当性検証技法 [2] を応用している。

$S_{n(imp)}$ のそれぞれのスライスに対して、入力表明 A_{pre} と出力表明 A_{post} を割り当てる。逆行代入法により、出力表明が満たされるために入力節点で成立している必要がある条件 $A_{pre}*$ を求める。検証履歴に残されている変更前のスライスの $A_{pre}, A_{pre}*$ に対して、

$$\text{変更後のスライスの}\mathrm{A_{pre}} \vdash \text{変更前のスライスの}\mathrm{A_{pre}}$$
$$\text{変更後のスライスの}\mathrm{A_{pre}*} \dashv \text{変更前のスライスの}\mathrm{A_{pre}*} \quad (1)$$

の定理証明を行なう。成立する場合は、変更後のスライスが変更前のスライスと同じ表明を満たしている。つまり、そのスライスが変更による影響を受けていない。厳密には履歴中のどのスライスと対応するのかを一般的に決めることはできないので、全てのスライスに対して検証する必要がある。履歴中のどのスライスに対しても上記の式が成立しない場合、変更による影響をそのスライスが受けていることが確定する。検証履歴はアプリケーションごとに存在する。したがって、全てのアプリケーションについて検証する必要がある。全てのアプリケーションについて検証が成功したスライスは、変更による影響を (少なくとも対象としているアプリケーション群において) 受けていないことが確定する。

3.3 再構成

検証の結果に基づいて、フレームワークの再構成とアプリケーションの実装の変更を行なう。

3.3.1 再構成対象となるスライスの決定

変更前のフレームワークのスライスと変更後のフレームワークのスライスとの対応関係を求める。スライスの対応関係を以下のように定義する:

スライス S に含まれるテスト論理式の集合を $Cond(S)$ で表す。変更前のスライス S と変更後のスライス S' との間に

$$Cond(S) \subseteq Cond(S') \quad (2)$$

が成立し、かつスライスの基準変数、基準節点が同じであれば、S と S' が対応しているとし、$Rel(S, S')$ で表す。以上の定義の元で、変更前後のスライス間で対応関係を求める。この対応関係と検証結果に基づいて、変更前のフレームワークから再構成の対象となるスライスを決定する。変更の影響を受けているスライスと対応関係を持つ変更前のスライスを再構成の対象とする。

3.3.2 仮想関数の導入

再構成の対象となるスライスに対応する仮想関数を新規導入する。個々のスライスに一つずつ仮想関数を対応づけると、フレームワークが断片化されてしまうので、スライスのグループ化を行なう。各アプリケーションにおける影響の有無が完全に一致するスライスを一つのグループとする。グループ単位に仮想関数を導入する。

また、再構成の対象とならなかったスライスは、一つの関数にまとめて新しいフローズンスポットとする。フローズンスポットと仮想関数を合わせて新しいメソッドを構成する。

スライシングの結果、一つの文が複数のスライスに含まれることがある。このため、フローズンスポットで一度実行された文がさらに仮想関数で重複実行される可能性がある。また、再構成した関数を連結して呼び出す場合には、文の実行順序が再構成前と変化する可能性がある。これらの文の実行形態の変化は、その文によってデータ依存を受ける変数に影響を与える。したがって、この変数に対する影響の回避を考える必要がある。元のスライスの基準変数 x を以下の 4 つのケースに分けて、対応方法を示す:

1. 基準変数 x が更新されない場合

   この場合は影響がないので、対応しなくて良い。

2. 基準変数 x がローカル変数の場合

   この場合はスライスの定義によって、x の初期化の文が必ず入るので、重複実行による影響がない。

3. 基準変数 x が元のメソッドの引数の場合

   この場合は仮想関数の中で x の初期化の文がないので、フローズンスポットの中で、x の初期値を記憶しておき、そして仮想関数で x に初期値を代入し、影響の波及を回避することができる。

4. 基準変数 x が大域変数あるいはメンバー変数の場合

この場合もまずフローズンスポットの中で、xの初期値を記憶しておく。仮想関数に仮引数x0を導入し、これを初期値で呼び出す。仮想関数の中では、xの更新が行なわれる文以前でxを参照している箇所をx0に置き換える。これにより、xが意図しない値に更新されるのを回避することができる。

3.3.3 仮想関数のオーバライド

仮想関数のままでは、仕様変更を行なっていないアプリケーションの仕様を満たすことはできるが、仕様変更があったアプリケーションにおいては、仕様に対応するように仮想関数をオーバライドしなくてはならない。具体的には、まず元のクラスのサブクラスを作り、仮想関数で呼び出している関数と同じ名前の関数を宣言し、影響を与える変更前のスライスに対応関係を持つ変更後の全てのスライスに合わせて関数をオーバライドする。そして、実装における関数の呼出し関係により、新しいサブクラスでオーバライドした関数が参照されるように変更を加える。

4 Unidrawにおける再構成の実行例

以下では、図形エディタフレームワークUnidrawを用いたソフトウェア開発を例に、Unidrawのフローズンスポットに対する変更が生じた場合の再構成を説明する。

Unidrawに基づいて、PunidrawとIdrawという二つの図形エディタアプリケーションが開発されているとする。Punidrawの仕様が図形オブジェクトの移動操作について変更され、キャンバスの中を自由に動けるという仕様から、縦か横のどちらか移動量が多かった方に寄せられるという仕様に変わったと仮定する。

図形オブジェクト移動操作に関するPunidrawの実装を図7に示す。関数`Tools`でUnidrawの`MoveTool`クラスのメンバ関数`InterpretManipulator`を呼び出し、さらに、その関数の中から`GraphicView`クラスのメンバ関数`InterpretManipulator`を呼び出す構造となっている。

前述の仕様変更は、Punidrawの実装プログラムにおいて、Unidrawのフローズンスポットのメンバ関数を呼び出す部分を変えるだけでは満たすことができない。Unidraw中で呼び出されているクラスのメンバ関数も変えなくてはいけないので、実際に呼び出されている`GraphicView`クラスの`InterpretManipulator`メンバ関数の中の

```
cmd = new MoveCmd(ed, fx1-fx0, fy1-fy0);
```

という行に分岐文を入れ、これを

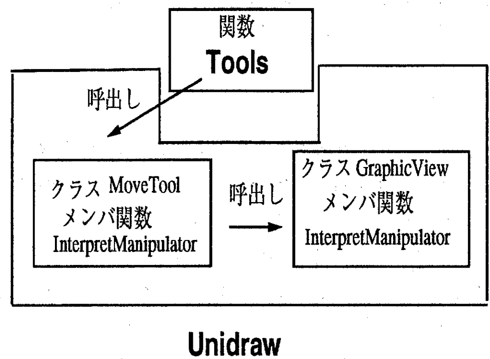

図 7: Punidraw(変更前)における呼出し関係

```
if (abs(fx1-fx0) >= abs(fy1-fy0)) {
    cmd = new MoveCmd(ed, fx1-fx0,0);
} else {
    cmd = new MoveCmd(ed,0, fy1-fy0);
}
```

と変更し、Punidrawの仕様変更に対応する。しかしこの変更により、Idrawにも影響を与えてしまう。この影響を回避するために、一度変更したUnidrawを再構成する必要がある。以下で、この`InterpretManipulator`関数の分割、検証、再構成を示す。また、Idrawのアプリケーションプログラムを仕様に対して検証を行なった結果として残した検証履歴から、メソッドの入力表明および出力表明、変更前のメソッドが分割された6つのスライスS_1, S_2, S_3, S_4, S_5, S_6、そして各スライスのA_{pre}と$A_{pre}*$がある。

4.1 分割

変更されたメソッドをスライス分割の結果、S_1', S_{2-1}', S_{2-2}', S_{3-1}', S_{3-2}', S_4', S_5', S_6'の8つのスライスが得られる。この8つのスライスのうち、変更された文を含むスライスはS_{2-1}', S_{2-2}', S_{3-1}', S_{3-2}'の4つである。よって、この4つのスライスをIdrawに影響を受ける可能性のあるスライス群$S_{n(imp)}$とし、それ以外のスライスS_1, S_4, S_5, S_6は影響を受けないスライス群$S_{n(noimp)}$とする。分割の結果は図8で示す。

4.2 検証

$S_{n(imp)}$中のスライスS_{2-1}', S_{2-2}', S_{3-1}', S_{3-2}'が検証の対象となる。個々のスライスに対して、入力表明A_{pre}と出力表明A_{post}を割り当てる。出力表明A_{post}か

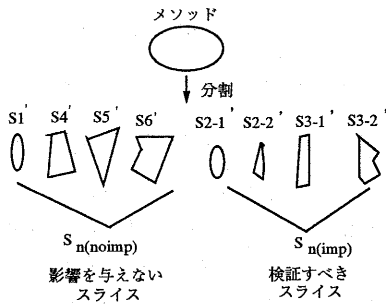

図 8: 分割の結果

ら逆行代入法によって、各スライスの入力節点で成立している必要のある条件 $A_{pre}*$ を求める。

検証履歴に残されている変更前の各スライスの A_{pre} と $A_{pre}*$ に対して、式1の定理証明を行なった結果、スライス S_{3-1}' のみが変更前のスライス S_3 と上記の関係を持ち、他の3つのスライスが変更前全てのスライスと関係を持たないという結果になる。よって、検証すべきスライス群 $S_{n(imp)}$ の中、S_{2-1}', S_{2-2}', S_{3-2}' が Idraw に影響を与え、S_{3-1}' が影響を与えないという結果になる。

4.3 再構成

4.3.1 再構成対象となるスライスの決定

スライスの対応関係は、$Rel(S_1, S_1')$, $Rel(S_2, S_{2-1}')$, $Rel(S_2, S_{2-2}')$, $Rel(S_3, S_{3-1}')$, $Rel(S_3, S_{3-2}')$, $Rel(S_4, S_4')$, $Rel(S_5, S_5')$, $Rel(S_6, S_6')$ となる。変更前のスライス S_{2-1}', S_{2-2}', S_{3-2}' が Idraw に影響を与えるという検証結果より、対応関係を持つ変更前のスライス S_2 と S_3 が Idaw に影響を与える結果になる。

4.3.2 仮想関数の導入

Idraw に影響を与えるスライス S_2, S_3 をグループ化し、対応する仮想関数を導入する。まず、GraphicView クラスの関数宣言部で以下のように新しい関数 newmove とそれを呼び出す仮想関数 virmove の宣言をする。

```
class GraphicView : public ComponentView {
public:
  Command* newmove(Manipulator*);

  virtual Command* virmove(Manipulator* m)
       {return newmove(m);}
};
```

次に、メンバ関数の定義部で newmove 関数の定義をする。関数 newmove の中身はスライス S_2, S_3 を合わせたものである。

最後に、変更前のスライスの中、S_2, S_3 以外の Idraw に影響を与えないスライス S_1, S_4, S_5, S_6 を合わせて、新しいフローズンスポットとして元のメソッドに残す。フローズンスポットと仮想関数を合わせて、新しいメソッドを構成する。

```
Command* GraphicView::InterpretManipulator {
        フローズンスポット部分
        (S1, S4, S5, S6 を合わせた部分)

        仮想関数 virmove;
        (S2 と S3 を合わせた部分)
}
```

4.3.3 仮想関数のオーバライド

アプリケーション Punidraw の実装において、以下のように Unidraw 中の元の GraphicView クラスのサブクラスとして newGraphicView を作り、新しく定義したメンバー関数 newmove をオーバライドする。

```
class newGraphicView : public GrachicView{
  Command* newmove(Manipulator* );
};
```

オーバライドする関数 newmove の内容は S_2, S_3 と対応関係を持ち、Idraw に影響を与えるスライス S_{2-1}', S_{2-2}', S_{3-1}', S_{3-2}' を重ね合わせものとなる。

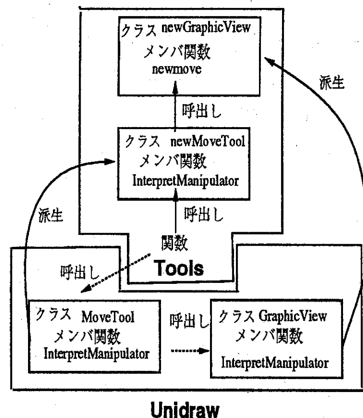

図 9: 変更後の呼出し関係

MoveTool クラスのサブクラス newMoveTool を作り、GraphicView クラスの InterpretManipulator メンバ関

数を呼ぶところを、newGraphicView クラスの newmove 関数を呼ぶようにオーバライドする (図 9)。

最後に、Punidraw の実装において、図形オブジェクトの移動操作に関する呼出しの部分を newMoveTool クラスを呼び出すように変更する。これで、newGraphicView クラスでオーバライドした newmove 関数が呼び出されることで、Punidraw の仕様変更を満たすことができる。

5 評価

再構成後のフレームワークにフローズンスポットとして残されているコードの量を計測し、再利用性を損ねない再構成手法としての有効性を評価した。

4 章で述べた Unidraw の再構成は以下のようになった。

- 変更前のメソッドのコード:39 行
- 変更後のメソッドのコード:44 行
- 再構成後のメソッドに残ったコード:22 行
- 再構成により新しく増加したコード:32 行
- Punidraw の実装の変更により増加したコード:59 行

フローズンスポットにあるメソッドの半分以上のコードが再構成の後もフローズンスポットに残り、再利用できていることが分かる。また再構成後のフレームワークは変更前より約 15 行増加したという結果になっている。再構成によって、もともと一つだった関数が複数の関数に分離した結果、文が重複していることが原因である。さらに、Punidraw の実装の変更によって、59 行も増加している。わずかな行の変更であっても、その影響を受ける部分は広いことがわかる。

6 まとめ

複数のアプリケーションの実装によって共有されているフレームワークが変更を受けた時に、その影響がアプリケーションの実装に波及しないように、再構成する手法について述べた。スライシングによるプログラム分割技法とプログラムの正当性検証技法を用いることでフレームワークの再利用性を損なわない再構成を行なうことができた。本手法をフレームワークの洗練のための支援技術と見なすこともできる。

本手法の適用による問題点として

- コードの断片化 (fragmentation) によるフレームワーク全体の見通しの悪化、複雑さの増大
- 仮想関数の機械的な追加による API の統一性の破壊
- 重複するコードの増加によるフレームワークの肥大化

などが考えられる。したがって、フレームワークのいかなる変更に対しても本手法による再構成を行なうのは適当ではない。フレームワークの本来の適用可能領域から大きく逸脱する変更については本手法による再構成を放棄すべきであると考えている。フレームワークに加えられた変更がその適用領域に与える影響を算出する手法を今後の課題にしたい。

参考文献

[1] Ralph E.Johnson, 中村宏明, 中山裕子, 吉田和樹: パターンとフレームワーク, 共立出版, 1999.

[2] 小野康一, 丸山勝久, 深澤良彰: プログラム変更に対する正当性検証技法と分割技法の適用, 信学論 D-I, Vol.J77-D-I, No.11, pp747-758, 1994.

[3] Z.Manna: "*Properties of Programs and the First-Order Predicate Calculus*", J. Assoc, Comput. Mach., Vol.16, No.2, pp.244-255, 1969.