2004－SLDM－117（8）
2004／12／1

リファクタリング技術を応用したHW/SW分割

山崎　亮介†　　吉田　紀彦††　　楢崎　修二†††

† 埼玉大学大学院理工学研究科
〒 338 8570 埼玉県さいたま市桜区下大久保 255
†† 埼玉大学工学部
〒 338 8570 埼玉県さいたま市桜区下大久保 255
††† 長崎大学工学部
〒 852 8521 長崎県長崎市文教町 1 14
E-mail: †ryosuke@ss.ics.saitama-u.ac.jp, ††yoshida@ics.saitama-u.ac.jp, †††narazaki@cs.cis.nagasaki-u.ac.jp

あらまし　システムレベル設計に対してオブジェクト指向技術を適用するためには，いくつかの課題を解決しなければならない．それら課題はオブジェクト指向技術の特徴であるデータ抽象化と動的結合に起因するものである．その特徴はシステムレベル設計に対しては弊害となる．本稿は，それら課題を明らかにし，HW/SW 分割を含めた解決方法を提案する．我々の提案手法はリファクタリング技術にもとづく．リファクタリングとは，システムの外部振る舞いを変えずに，システムの内部構造を変える手段である．システムレベル設計に提案手法を適用することで，前述の課題を解決することができた．また，HW/SW 分割の指針を示すことができた．

キーワード　リファクタリング, オブジェクト指向設計, システムレベル設計, HW/SW 分割

HW/SW Partitioning Using Refactoring Technique

Ryosuke YAMASAKI†, Norihiko YOSHIDA††, and Shuji NARAZAKI†††

† Graduate School of Science & Engineering, Saitama University
Shimoōkubo 255, Sakura-ku, Saitama-shi, Saitama, 338 8570 Japan
†† Faculty of Engineering, Saitama University
Shimoōkubo 255, Sakura-ku, Saitama-shi, Saitama, 338 8570 Japan
††† Faculty of Engineering, Nagasaki University
Bunkyo-machi 1 14, Nagasaki-shi, Nagasaki, 852 8521 Japan
E-mail: †ryosuke@ss.ics.saitama-u.ac.jp, ††yoshida@ics.saitama-u.ac.jp, †††narazaki@cs.cis.nagasaki-u.ac.jp

Abstract　To apply Object-Oriented techniques to System Level Design, we must solve some problems. Their problems are caused by data abstraction and dynamic binding that are peculiarities of Object-Oriented techniques. The peculiarities prevent applying Object-Oriented techniques to System Level Design. In this paper, we make clear their problems and propose a new approach including HW/SW partitioning. The approach we propose is based on Refactoring techniques. The Refactoring is the way of changing the internal structure of the system without changing the external behavior of the system. We could solve the above-mentioned problems using our approach. And we could show the policy of HW/SW partitioning.

Key words　Refactoring, Object-Oriented Design, System Level Design, HW/SW partitioning

1. まえがき

近年，システム LSI がますます複雑化，大規模化しており，その設計期間はさらに短縮化されている．このような状況の中，設計者の設計生産性を向上させるため，システムレベル設計という設計手法が考案され，設計言語としてシステムレベル言語がいくつか提案されている．これら設計手法や設計言語の登場により設計生産性は向上している．

更なる設計生産性の向上を目指し，最近ではオブジェクト指向をシステムレベル設計に応用する研究が進められている．オ

ブジェクト指向が備える高い抽象度によりオブジェクト間の結合を抽象化できるので，過去の設計資産の再利用が容易になる．ゆえに，設計者への設計負担の軽減が期待できる．現在は設計仕様に UML（Unified Modeling Language）が用いられつつあるが，設計仕様だけでなくシステムレベル設計全体に対してもオブジェクト指向の適用が望まれている．しかし，システムレベル設計はハードウェア（以下，HW）設計も設計対象としている．したがって，オブジェクト指向をシステムレベル設計に適用する場合，データ抽象化や動的結合といったオブジェクト指向の特徴が弊害となることがある．例えば，リコンフィギャラブル HW を除き，一般の HW ではオブジェクトの動的結合や動的生成は不可能である．また，オブジェクト指向の特徴である継承により機能拡張されたクラスは肥大化している．したがって，そのようなクラスを HW とソフトウェア（以下，SW）へ分割することは困難または不可能な場合がある．

本研究では，これらの課題を解決するためにリファクタリングという技術を応用する．リファクタリングとは，システムの外部振る舞いを保存したまま，システム内部を再構成する技術である．リファクタリングの本来の目的は部品化や再利用の簡便化，促進といったオブジェクト指向の恩恵を最大限に得ることである．しかし，我々はリファクタリングが備える外部振る舞いを保存したままシステム内部を再構成する能力に着目し，データ抽象化と動的結合に起因する課題を解決する．

本稿の構成は以下のとおりである．2. でオブジェクト指向をシステムレベル設計に適用する際の課題を明らかにし，関連研究とそこで提案されている解決方法を述べる．3. で，リファクタリングについて述べる．4. で我々が提案するそれら課題の解決方法を述べる．5. ではインターネットルータを設計例題として，我々の提案手法の適用と考察を述べる．6. で本稿のまとめと今後の課題，展望を述べる．

2. 関連研究と本研究の位置づけ

オブジェクト指向が備える抽象度の高さにより，SW アプリケーションの設計生産性は格段に向上している．近年，設計生産性の大幅な向上が強く望まれている HW 設計やシステムレベル設計に対して，SW 工学の成果であるオブジェクト指向を適用する研究が進められている．その適用対象であるシステムレベル設計は SW と同様に HW も設計対象としている．したがって，SW と HW の抽象度の差に起因するいくつかの課題を解決しなければ，システムレベル設計に対してオブジェクト指向を適用することは困難である．ここでは，それら課題を明らかにするとともに，解決方法を提案している関連研究と我々の提案手法との差異について述べる．

2.1 解決すべき課題

システムレベル設計の設計プロセスの 1 つに HW/SW 分割がある．そこでは設計面積制約，消費電力制約，実行時間制約など，多くの制約を満たすように設計仕様あるいは実行可能仕様を HW と SW へ分割しなければならない．ところが，オブジェクト指向を用いて設計仕様や実行可能仕様を作成すると，それら設計制約のうち，とくに設計面積制約と消費電力制約を満たすことが困難あるいは不可能となる以下のよう場合がある．

（1） 継承や合成による機能拡張によりクラスが肥大化している．

（2） 機能拡張を行っていないクラスであっても，クラス内のメソッドが長すぎるためにクラスが肥大化している．

（3） クラス内に使用されないメソッドやフィールドがある．

また，HW 特有の制約を満たすことができない場合を以下に挙げる．ただし，リコンフィギャラブル HW を除く．

（4） オーバーロードやポリモーフィズムによりオブジェクト間の接続を動的に変更できる．

（5） オブジェクトの動的生成が可能である．

2.2 関連研究

上記の問題について，すでにいくつかの論文で解決方法が提案されている．[1] では，(1) ～ (3) の場合のように大きすぎるオブジェクトがある場合，デレゲーションパターンを応用したオブジェクトデコンポジションを行う．ところが，この手法では分割粒度を自由に設定できない．[2] はシステムに最適な分割を行うために，クラスタベースで CDFG(Control Data Flow Graph) を作成する分割手法を提案している．この手法ではタスク，関数，ブロックなどの粗い粒度から，ステートメントや演算などの細かい粒度までを分割対象としている．したがって，様々な粒度で HW/SW 分割が可能である．しかし，この手法はオブジェクト指向を考慮していないため，とくに (1) を解決できない．[3] では，分割粒度をタスクとしている．この手法もオブジェクト指向を考慮していないため，とくに (1) を解決できない．また，分割粒度を自由に設定できない．

2.3 本研究の位置づけ

上記 5 つの問題を整理すると，次の 2 つにできる．

（1） 肥大化したクラス

（2） 動的結合

HW を意識する必要がない場合，つまりシステム全体を SW として実現する場合であれば，(1) と (2) は重大な問題とはならない．(2) はオブジェクト指向の特徴であり，柔軟な SW アプリケーション開発を促進するものである．ところが，システムレベル設計では HW も設計対象とするため，(2) は制限しなければならない．

(1) を解決するためには，必要なメソッドとフィールドのみを取り出す方法を考えなくてはならない．(2) を解決するためには，オブジェクト間の接続，つまりメソッドの入出力を指定し，オブジェクトの動的生成を禁止しなければならない．

この 2 点を解決するために，我々はメソッド呼び出しにもとづくコールグラフを作成する．図 1 にコールグラフを示す．必要なメソッドをクラスとして定義し，そのインスタンス，つまりオブジェクトをコールグラフのノードとする．したがって，メソッドが親クラスで定義されている場合であっても，そのメソッドを取り出すことができる．チャネルはノード間を固定的に接続する．これにより，必要なフィールドのみを取り出すことができ，オブジェクトの動的結合を制限することができる．

取り出すメソッド，つまりコールグラフのノードが大きい場合はそのノードを複数のノードに細分化する．コールグラフの

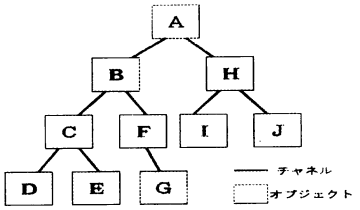


図 1　コールグラフ
Fig. 1　Call graph.

ノードが小さい場合はそのノードを他のノードとともに集約する．細分化と集約により様々な分割粒度を自由に設定できるので，システムに適切な HW/SW 分割が可能である．本研究ではコールグラフを作成するために，リファクタリング技術を応用する．

3. リファクタリング

[4] によると，リファクタリングとは「外部振る舞いを保存しながら，部品化や再利用が容易になるように SW の内部構造を変化させること」である．ゆえに，リファクタリングには部品化と再利用を妨げる様々な状況ごとに，それらを改善するための技法がある．例えば，重複したコードは部品化と再利用を妨げる要因の 1 つである．その状況を改善するために，リファクタリングには「メソッドの抽出」,「クラスの抽出」,「テンプレートメソッドの形成」といった技法がある．状況に応じて複数の技法を組み合わせて使用することもある．

これら技法は決まった手順に必ず従わなければならない．また，手順ごとに必ずテストを行わなければならない．これら規則を遵守することで，システムの外部振る舞いを変更せずにシステム内部を再構成することができる．したがって，リファクタリングの規則に従わずにコードを書き換えると，システムの外部振る舞いが変わる，あるいはバグが混入するかもしれない．

リファクタリングについて，その技法の 1 つである「メソッドの抽出」を例に説明する．プログラムコード内にひとまとめにできるコードの断片がある場合やメソッドが長く理解しづらい場合,「メソッドの抽出」を行う．例コードを図 2 (a) に，手順を以下に示す．

（1） 新たなメソッドを作成し，抽出するコードの意図に合わせて命名する．

（2） 抽出されるコードを元のメソッドから新たな抽出先のメソッドにコピーする．

（3） 抽出されるコードのうち，元のメソッドにおいてローカル変数への参照を探す．

（4） 抽出されるコード内だけでローカル変数が使われていることを調べる．

（5） 抽出されるコードがローカル変数を変更するかどうかを調べ，変更される変数が 1 つならば，抽出したコードを問い合わせメソッドとし，結果を変数に代入する．

（6） 抽出されるメソッドが読むローカル変数を，抽出先メソッドの引数とする．

（7） 全てのローカルスコープ変数に対する処置が終わったらコンパイルする．

（8） 抽出されたコード部分を抽出先メソッドへの呼び出しに置き換える．

（9） コンパイルしてテストする．

図 2 (a) のコードに対して,「メソッドの抽出」を行うことにより図 2 (b) のコードを得る．リファクタリングを行うことで，外部振る舞いを変えることなく，抽出されたメソッドの再利用がより容易になる．

```
void display(int t){
  int i;
  printHeader();

  //t 回 FC を表示する
  for(i=0;i<t;i++){
    System.out.print("FC");
  }
}
```

(a) Before Refactoring.

```
void display(int t){
  printHeader();
  multiDisplay(t);
}
void multiDisplay(int t){
  int i;
  for(i=0;i<t;i++){
    System.out.print("FC");
  }
}
```

(b) After Refactoring.

図 2　例コード 1
Fig. 2　Example code 1.

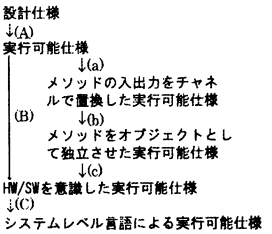


図 3　設計フロー
Fig. 3　Desgin flow.

4. 提案手法の詳細

図 3 に，我々が提案する設計手法を示す．まず，HW/SW を意識せずに UML を用いてシステムの設計仕様を作成する．次に，その設計仕様をもとにオブジェクト指向プログラミング言語を用いて実行可能仕様を作成する（図 3（A））．この実行可能仕様も HW/SW を意識しない設計仕様である．続いて，実行可能仕様を再構成することで HW/SW を意識した実行可能仕様とする（図 3（B））．最後に，HW のパーティションに割り当てられたオブジェクトをシステムレベル言語へ書き換える（図 3（C））．これらの作業はリファクタリング技術にもとづいているので，システムの外部振る舞いは常に保存されている．

4.1 で図 3（A），4.2 では図 3（B），4.3 で図 3（C）について説明する．

```
class XX{
  void methodA(){
    int a, b;
    b=methodB(a);
  }
  int methodB(int x){
    return x+1;
  }
}
```

図 4　例コード 2
Fig. 4　Example code 2.

```
class XX{
  void methodA(){
    Channel a, b;
    methodB(a, b);
  }
  void methodB(Channel x,
               Channel y){
    y.write(x.read()+1);
  }
}
```

(a) Rewrited code.

```
class Channel{
  int v;
  int read(){
    return v;
  }
  void write(int x){
    v=x;
  }
}
```

(b) Channel.

図 5　チャネルを用いたメソッドの接続
Fig. 5　Connecting methods with channel.

4.1　実行可能仕様の作成

設計仕様をもとに実行可能仕様を作成する（図 3（A））．この段階では HW/SW を区別しない．まず，ユースケースを洗い出し，ユースケース図を作成する．次に，各ユースケースを実現するために必要となるクラスを検討し，クラス図を作成する．続いて，ユースケース図，ユースケース，クラス図をもとにシーケンス図を作成する．通常，これらの作業は 1 度で完了することはなく，実行可能仕様を作成することができる詳細度となるまで繰り返す．また，各図の作成順は任意である．UML には他にもいくつかの図があるが，必要に応じてそれら図を作成する．設計仕様を作成した後，それをもとに実行可能仕様を作成する．

4.2　システム内部の再構成

実行可能仕様を再構成することで，HW/SW を意識した実行可能仕様を作成する（図 3（B））．再構成はリファクタリング技術にもとづく．再構成は 3 つのステップからなる．(a) で，メソッドの入出力を全てチャネルを用いて置き換える．(b) で，チャネルで接続されたメソッドをクラスとして定義し，メソッド呼び出しの起点を頂点とするコールグラフを作成する．(c) で，コールグラフに対してプロファイリング，集約，細分化を行う．以上の (a)，(b)，(c) それぞれについて，4.2.1，4.2.2，4.2.3 で詳述する．

4.2.1　チャネルを用いたメソッドの接続

オブジェクト指向の特徴である動的結合を制限するために，動的なインスタンス生成を制限し，オブジェクト間の接続を明確化する．以下に詳細を述べる．

インスタンス生成をコンストラクタ内だけに限り，システムを動作させる前に必要となるオブジェクトを全て生成し終えることで，動的なインスタンス生成を制限する．オブジェクト間の接続を明確化するためにチャネルを利用する．これらの手順はリファクタリングの 1 つ，「オブジェクトによるデータ値の置き換え」に従う．例コードを図 4 に，手順を以下に示す．

（1）　呼び出し先クラスの上位クラスでメソッドを定義している場合，呼び出し先クラスへ必要なメソッドとフィールドをリファクタリング技法の「メソッドの格下げ」，「フィールドの格下げ」，「階層の平坦化」を行って 1 まとめにする．

（2）　呼び出し先メソッドの引数リスト内のそれぞれの引数の型と，返り値の型を調べる．

（3）　上記の引数の型および返り値の型に合ったチャネルクラスを作成する．チャネルは読み出し用メソッドと書き込み用メソッドを備える．

（4）　チャネル型の引数を持つメソッドを新たに作成する．出力先もメソッドの引数とする．

（5）　オリジナルの呼び出し先メソッド内部をコピーし，新しく作成したメソッドへコピーする．

（6）　引数を使用している部分をチャネルにアクセスしてデータを取り出すように書き換える．

（7）　値を返している部分は，引数リストにある出力用チャネルへ書き込むように書き換える．

（8）　呼び出し元メソッド側で，オリジナルの呼び出し先メソッドを呼ぶために使用している変数を調べ，チャネルに置き換える．

（9）　オリジナルの呼び出し先メソッドを呼び出している部分を，新しく作成した呼び出し先メソッドを呼ぶように書き換える．

（10）　各ステップで，必ずコンパイルとテストを行う．

上記のリファクタリングを行うことで，システムを構成する最小限のメソッドを，チャネルを介する固定接続とすることができる．上記手順を図 4 の例コードへ適用した結果を図 5 (a)，(b) に示す．

このリファクタリングだけでは，チャネルおよびチャネルによって固定的に接続されたメソッドを追加しているため，その分クラスが肥大化している．したがって，その肥大化したクラスから必要なメソッドとフィールドのみを分取り出し，分離・独立させるリファクタリングを行わなくてはならない．4.2.2 でその方法を説明する．

4.2.2　メソッドのクラス化と独立

図 5 (a) のコードから必要なメソッドとフィールドを分離・独立させるために，リファクタリング技法の「フィールドの移動」，「メソッドの移動」，「クラスの抽出」，「局所的拡張の導入」を行う．手順は以下のとおりである．

（1）　チャネルで接続された全てのメソッドを調べる．

（2）　「クラスの抽出」を行い，メソッドに対応したクラスを定義する．

（3）　「フィールドの移動」を使い，定義したクラスへメソッ

```
class A{
  B objB;
  void A(){
    a=new Channel();
    b=new Channel();
    objB=new B(a,b);
  }
  void main(){
    objB.main();
  }
}
```

(a) Class A.

```
class B{
  Channel a,b;
  void B(Channel x,
         Channel y){
    a=x;
    b=y;
  }
  void main(){
    y.write(x.read()+1);
  }
}
```

(b) Class B.

図 6 クラス定義

Fig. 6 Class Definition.

```
class クラス名{
  /*外部接続用チャネル*/
  /*内部接続用チャネル*/
  /*内部オブジェクト*/
  コンストラクタ (外部チャネル){
    /*外部接続用チャネルと外部チャネルの接続*/
    /*内部チャネルの生成*/
    /*内部オブジェクトの生成と接続*/
  }
  void main(){
    /*動作定義*/
  }
}
```

図 7 クラスの内部構造

Fig. 7 Internal Structure of a class.

ドが使用するチャネルを移動する.

(4) 「メソッドの移動」を行い, 定義したクラスへメソッドを移動する.

(5) 合成による「局所的拡張の導入」を行い, 実際の処理を委譲する.

図 5 (a) のコードを上記の手順に従いリファクタリングした結果を図 6 (a), (b) に示す. このリファクタリングでコールグラフを作成する. コールグラフを構成するノードオブジェクトの内部構造を図 7 に示す. コールグラフは親ノードが子ノードの実行順を決定し. 左側から順にノードを実行する. 図 1 のコールグラフは A-B-C-D-E-F-G-H-I-J の順に実行する. ノードの main メソッド内の条件分岐により, ノードは実行されないこともある. 例えば, ノード B の main メソッド内で条件分岐する場合を考える. 条件式を評価したとき, それが真であれば C を実行し, 偽であれば F を実行するものとする. もしも条件式の評価が真であれば, F を実行しない. したがって G も実行しない.

4.2.3 プロファイリングと HW/SW 分割

コールグラフに対してプロファイリングを行う. オブジェクトの動作頻度, チャネルの被アクセス頻度, 実行に要する時間, 設計面積, 消費電力など, プロファイリングの項目は数多くある. これらをもとに HW/SW 分割を行う.

例えば, 動作頻度に注目した場合, 頻繁に動作するオブジェクトは HW へ割り当てるほうがよい. 逆に, しばしば動作するオブジェクトは SW へ割り当てるほうがよい. この動作頻度は相対的なものである. 実行に要する時間が時間制約を満たすのであれば, 頻繁に動作するオブジェクトを SW へ割り当てることも可能である. 設計面積制約を満たすのであれば, しばしば動作するオブジェクトを HW へ割り当てることも可能である.

この段階の実行可能仕様を構成する各オブジェクトについて, その粒度が粗い場合には細分化を行う. 細分化では 4.2.1 と 4.2.2 で行ったリファクタリングを繰り返す. 過度に細分化されたオブジェクトは集約する. 集約はリファクタリング技法の「メソッドのインライン化」,「フィールドの移動」,「クラスのインライン化」に従う. 細分化と集約により, 分割粒度を自由に決めることができ, より適切な HW/SW 分割が可能となる.

```
behavior A(){
  Channel a, b;
  B objB(a,b);
  void main(void){
    objB.main();
  }
};
```

(a) Behavior A.

```
behavior B(Channel x,
           Channel y){
  void main(void){
    y.write(x.read()+1);
  }
};
```

(b) Behavior B.

図 8 SpecC コード

Fig. 8 SpecC code.

4.3 システムレベル言語への書き換え

プロファイリングの結果, HW へ割り当てられたオブジェクトをシステムレベル言語を用いて書き換える. 例えば, 書き換え対象のシステムレベル言語が SpecC の場合は以下の手順に従う.

(1) behavior にクラスと同じ名前をつける.

(2) クラスのコンストラクタ引数を behavior の引数とする.

(3) クラス内の内部接続用チャネル, 内部オブジェクトを, behavior 内に同様に定義する.

(4) クラスの main メソッドを bihavior へコピーする.

図 6 (a), (b) を SpecC へ書き換えたコードを図 8 (a), (b) に示す.

システムレベル言語がサポートしていないデータ型を使用して実行可能仕様を作成している場合はデータ型を変更しなければならない. 例えば, Java のコレクションクラスである Vector や ArrayList は SpecC でサポートしていないため, 基本データ型の配列などへ変更しなければならない. 書き換えの対象とするシステムレベル言語が早期に決まっているのであれば, 実行可能仕様を再構成する際にデータ型の変更を行うほうがよい.

5. 例題設計

我々の提案手法の有効性を確かめるために, インターネットルータを例題として設計した. ルータは計算機上に仮想的に作成し, 経路表を作成する基本機能のみを実装している. ルーティングアルゴリズムは 3 種類ある. 距離ベクトル型, リンク

状態型，パスベクトル型である．これら3種類のアルゴリズムを実装した3つのルータを設計した．ルータ間で授受する情報は，それぞれのアルゴリズムに必要なものだけである．

設計仕様には一般に広く使われている UML を，実行可能仕様には Java を用いた．Java を用いる理由は2つある．1つはオブジェクト指向であること，もう1つは並行性を備えていることである．実際の HW は並列に動作するので，Java を用いることで実際のシステムに近いシミュレーションすることができる．書き換え対象のシステムレベル言語は，図7と構造が似ている behavior を持つ SpecC とした．

はじめに，ユースケース分析をおこない，「経路を計算する」，「経路情報を送受信する」，「次ホップを検索する」，「経路表を更新する」の4つのユースケースを得た．これらユースケースを実現するために，「経路探索」，「通信」，「経路表」の3つのクラスを定義した．次に，ユースケース図とクラス図をもとに，シーケンス図を作成した．続いて，設計仕様をもとに実行可能仕様を作成した．その後，4. に従い，実行可能仕様を再構成した．再構成の結果得られたコールグラフに対して，プロファイリングを行った．プロファイリングの項目は最も簡便なオブジェクトの動作頻度とした．図9 (a) にプロファイリング後のコールグラフの一部を示す．図9 (a) 中の「該当するものがあるか判定」オブジェクトは過度に細分化しているので，親オブジェクトへ集約した．その結果と図9 (a) 中の「アルゴリズムを実行する」オブジェクトの詳細を図9 (b) に示す．最後に，プロファイリングの結果をもとに HW のパーティションへ割り当てられたコールグラフのノードを SpecC で書き換えた．また，すべてのノードが HW のパーティションへ割り当てられた場合を想定し，コールグラフのノード全てを SpecC で書き換えた．

5.1 評価と考察

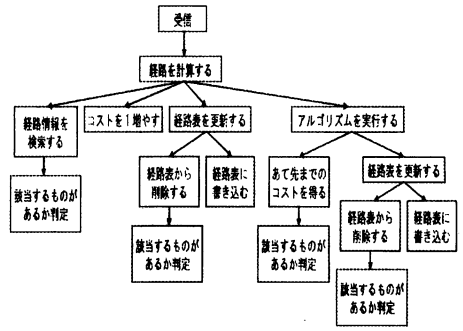
設計の上流では HW/SW を意識せず，通常の SW アプリケーションと同様に実行可能仕様を作成することができた．この実行可能仕様に対して提案手法を適用することで HW/SW を意識した実行可能仕様へと再構成することができた．現段階では手作業で再構成している．提案手法の自動化により設計者の設計負担を軽減することができ，設計生産性を向上させることができると考えている．

SpecC での書き換えの際，バグの混入はなかった．また，シミュレーションの結果，外部振る舞いが保存されていることが確認できた．ゆえに，図3に示しているように，提案手法を適用して段階的に抽象度を下げることで，抽象度の高さに起因する課題を解決できると考えている．

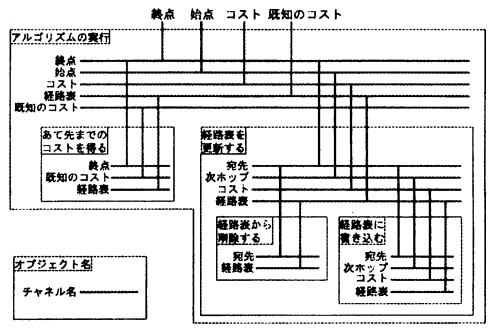
6. むすび

本稿はオブジェクト指向をシステムレベル設計に適用する際に生じる抽象度の高さに起因する課題を解決するために，リファクタリング技術を応用した設計手法を提案した．この設計手法を用いることで，HW/SW を意識しない設計仕様を HW/SW を意識した実行可能仕様へ書き換えることができた．また，システムレベル言語への書き換えは容易であった．

今後の課題として，提案手法による設計の自動化が第一に挙げられる．また，プロファイリングの項目，評価基準，評価方法を策定しなければならない．さらに，具体的なスケジューリング方法を考えなくてはならない．そのためには時間の概念をどの段階で取り入れるかを検討しなくてはならない．これら課題を解決することで，システムレベル設計全体に対してオブジェクト指向を適用することができるようになり，その結果，設計生産性を向上させることができると考えている．



(a) Call graph.



(b) Channels and Objects.

図9 結果の一部

Fig. 9 A part of result.

文献

[1] Carsten Schuls-key, Tommy Kuhn, and Wolfgang Rosenstiel, "A framework for system-level partitioning of object-orientedspecifications," Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI'2001), 2001.

[2] Nam-Hoon Kim, and Hyunchul Shin, "New partitioning techniques in hardware-software codesign," Journal of Korea Information Science Society, pp.94–99, Korea, Jan. 1998.

[3] Karam S. Chatha, and Ranga Vemuri, "An Iterative Algorithm for Hardware-Software Partitioning, Hardware Design Space Exploration and Scheduling," Design Automation for Embedded Systems Journal, Kluwer Academic Publishers, no.5, pp 281–293, Boston, Aug. 2000.

[4] マーチン・ファウラー，リファクタリング–プログラミングの体質改善テクニック，ピアソン・エデュケーション，東京，2000.