C_007

システムレベル設計におけるリファクタリング規則の定式化とカタログ化
Formulation and Cataloging of Refactoring Rules in System Level Design

木村 正裕† 小林 憲貴† 山崎 亮介† 吉田 紀彦†
Masahiro Kimura Kazutaka Kobayashi Ryosuke Yamazaki Norihiko Yoshida

1. はじめに

近年，携帯電話や情報端末など電子機器の小型化・高性能化に伴い，SoC（System-on-a-Chip）設計は高度化・複雑化している．これにより，従来の設計手法である低い抽象レベルからの設計では，特に，設計複雑度への対応が困難となってきた．そのため，設計抽象レベルを引き上げ，設計対象である部品数を減少させ，設計する必要が生じてきた．このように，急激に増大する設計の複雑さに対する解として，今日では，システムレベル設計 [1][2] と呼ばれる設計手法が確立されてきた．

システムレベル設計の概要を図1に示す．システムレベル設計とは，まず，システムの設計仕様から段階的に抽象レベルを下げていき，システム仕様をハードウェア側とソフトウェア側に分割する．そして，分割したハードウェア側とソフトウェア側の仕様を並行して詳細化し，最終的に一つのシステムとして統合するシステム設計手法である．

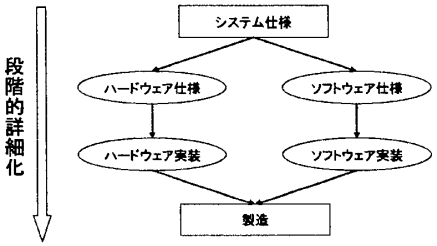


図 1: システムレベル設計の概要

システムレベル設計の問題点として，システムレベル設計では，所望の設計の機能，性能，消費電力，コスト等の厳しい制約条件が設計を複雑化しており，段階的詳細化におけるモデル変換手順は，現状では体系化されていない．そのため，この変換作業は設計者の経験や技術により手作業で行う必要があり，設計者ごとに設計効率に差が生じてしまう．特に，現状のモデル変換過程では，個々の変換作業における変換前と変換後のギャップが大きいため，設計仕様の機能や動作が変換後も不変という正当性の保証を示すことが難しい．また，現状では抽象レベルを引き上げたとはいえ，言語ベースで段階的詳細化による設計・検証をしているため，将来は言語レベルに依存しない，さらに抽象レベルを引き上げた UML（Unified Modeling Language：統一モデリング言語）ベースで設計・検証を可能とすることが望まれる．

そこで，本研究では，システムレベル設計における UML を用いた変換規則の定式化に主に焦点を当て，その正当性を保証する方法を考えた．UML での変換規則を定式化するには，まず，設計方法論が確立されている言語で定式化する必要があったため，設計方法論として SpecC 方法論 [1][2] を，規則の定式化‡にはリファクタリング [3] の技術を応用した．そして，言語ベースでの変換規則の正当性を検証した．具体的には，段階的詳細化の各モデル変換作業で現在見い出されている変換の指針 [2] を基に，それらのギャップを埋めるより詳細な規則を定式化した．その際，特に SpecC による実際の適用例§も交えつつ，その適用手順・方法を，リファクタリングカタログとしてまとめた [4]．

本研究の有益な点は，システム設計の初心者のみならず，経験者も段階的詳細化のモデル変換を容易に行えること，より微細な規則を定式化することにより，設計者は各モデル変換前後で挙動が保たれているかという正当性のチェックのギャップを埋め，そのチェックが容易になることである．また，定式化した規則は，今後，UML ベースで段階的詳細化による設計・検証を実現する場合の橋渡しになることも挙げられる．

以下，2.〜4. 節で SpecC とその設計方法論，ソフトウェア工学と SpecC におけるリファクタリング，本研究で定式化・カタログ化したリファクタリング規則の一覧を述べる．次に，5. 節で，規則の一覧の中から「PE（プロセス・エレメント）の割り付け」の実際の適用例を述べ，最後に 6. 節で結論として，本研究のまとめと今後の課題を述べる．

2. SpecC 方法論

SpecC[1][2] は C 言語を拡張したシステムレベル設計言語であり，構造や動作の階層・同期・例外処理・タイミング等をサポートしている．その他のシステムレベル設計言語には，C++を拡張した SystemC，ハードウェア記述言語である Verilog を拡張した SystemVerilog 等がある．

この SpecC に基づく設計方法論を SpecC 方法論 [1][2] という．SpecC 方法論は，システムレベル設計に基づいた設計手法であり，明確に定義された 4 つのモデルに区分されている．

以下に SpecC 方法論に基づく設計フローと各モデルの説明を示す．

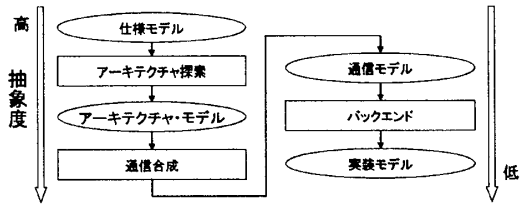


図 2: SpecC 方法論に基づく設計フロー

- 仕様モデル
  目的のシステム機能を持った機能記述．タイミングの概念を持たない．
- アーキテクチャ・モデル
  演算の順序をタイミングの見積りをベースとして定めた記述．通信は現時点では抽象レベル．
- 通信モデル
  通信をバス機能の部品モデルに詳細化し，演算と通信のタイミングが正確な記述．

†埼玉大学 Saitama University

‡SpecC には，SCE(System-on-Chip Environment) という半自動詳細化ツールが存在するが，これは内部でどのようにコード変換をしているのか詳細が分からない．本研究は，あくまで変換規則の定式化を目的としている．

§(株)Inter Design Technologies 社の組込みシステム統合開発ツール Visual Spec3.5[5][6] を用いた．

- 実装モデル
  RTL/IS でシステムのサイクル精度の記述.

3. リファクタリング

システムレベル設計におけるモデル変換手順を体系化するために，本研究では，リファクタリング [3] の技術を応用した．リファクタリングとは，ソフトウェア工学で提案されたモデル変換手法であり，外部から見たときの振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を再構成することである．具体的には，コードの再利用・保守を容易にするための様々な変換規則が定義されている．

4. SpecC におけるリファクタリング

SpecC 方法論におけるモデル変換過程では，設計仕様を表した仕様モデルから，本質的な挙動を保ちつつ，段階的に抽象レベルを下げていき，システム設計の最終的な結果である実装モデルへとモデル（コード）変換している．一方，リファクタリングでは，ソフトウェアの挙動を保ちつつ，コード変換を行い，ソフトウェアの内部構造を再構成させている．この二つはどちらも，変換前後で実質的な挙動を保っているため，リファクタリングを用いてシステムレベル設計，特に，SpecC 方法論におけるモデル変換規則を体系化することができる．

現在見い出されているモデル変換の指針 [2] を基に，本研究で定式化・カタログ化したリファクタリング規則 [4] を以下に示す．なお，これらの規則は，ソフトウェア工学のリファクタリング規則のように各々独立した規則として適用するのではなく，各モデルの変換過程で一連の流れとして適用する必要がある．

定式化・カタログ化したリファクタリング規則の一覧

- 仕様モデル
  - 通信と演算の分離（4 規則）
    1. 変数・イベントのカプセル化
    2. チャネルで置換
    3. 動作とチャネルの接続
    4. 通信の更新
- アーキテクチャ探索
  - PE（プロセス・エレメント）の割り付け（4 規則）
    1. 新たな階層の導入
    2. PE への割り付け・動作のグループ化
    3. 同期の挿入
    4. 通信の移動
  - 変数分割（4 規則）
    1. 変数を PE 内へ移動
    2. 通信チャネルの追加
    3. 変数へのアクセスの更新
    4. 通信と同期の最適化
  - スケジューリング（3 規則）
    1. 動作の階層の平坦化
    2. 動作の階層の逐次化
    3. 制御の最適化
  - チャネル分割（3 規則）
    1. 階層の追加
    2. チャネルのバスへの割り付け・通信のグループ化
    3. チャネルへのアクセスの更新
- 通信合成
  - プロトコルの挿入（3 規則）
    1. プロトコル・コードの挿入
    2. アプリケーション・レーヤの生成
    3. バス・チャネルの置換
  - プロトコルのインライン化（6 規則）
    1. プロトコル・レーヤの分割
    2. アプリケーション・レーヤの分割
    3. アプリケーション・レーヤの埋め込み
    4. 動作のポートへの接続
    5. 結線を Top レベルで定義
    6. 結線の引数の変更
- 動作合成・コンパイル
  - ハードウェア合成（1 規則）
  - ソフトウェア合成（1 規則）
  - インタフェース合成（1 規則）

5. 規則の適用例

ここでは，アーキテクチャ探索における「PE の割り付け」のリファクタリング規則について説明する．

5.1 PE（プロセス・エレメント）の割り付け

5.1.1 目的

- システム仕様をハードウェア側とソフトウェア側の仕様に分割する．
- 各 PE で実装される機能を決定し，以降の設計を効率よく行う．

5.1.2 リファクタリング規則の適用順序

初めに，見積りツール等を利用し，性能や必要メモリ量などの設計指標を決定する．その後，PE の割り付け方が決定した場合に以下の規則を適用する．

1. 新たな階層の導入
   動作の階層の TOP レベルに新たな階層（PE）を導入する．
2. PE への割り付け・動作のグループ化
   PE に動作を割り付け，仕様の動作に基づいて，割り付けた PE 内の動作をグループ化する．並列動作などの一部の動作を異なる PE に割り付けた場合，空白となった動作元の上位動作に新たな動作を挿入する．これは，元の仕様の階層構造を維持し，以降の設計を容易にするために行う．

そして，動作指標の見積りを行う．
動作を実行するコンポーネントに対して，選択された指標の見積り値をその動作にアノテートする（例：リーフ動作では，シミュレーション実行の際に，実行時間を正確なものとするために，適当な `wait()` 文が追加される）．

3. 同期の挿入
   並列動作などの一部の動作を異なる PE に割り付けた場合，分割した PE 間の動作の実行順序を保つために，同期通信チャネルを挿入する．その際，元々動作が存在していた側から先に通信を送るようにする．また，同期通信チャネルと接続される動作内の引数にこれらのインタフェースを指定する．なお，同期通信チャネルの実装は，空のメッセージでかまわない．
4. 通信の移動
   異なる PE に割り付けた動作間で使用している変数と通信チャネルを TOP レベルに移動させる．それに伴い，移動させた変数と通信チャネルに関連する下位動作の引数を更新する．その結果，PE 間の通信は変数と通信チャネルでのみ行われるようになる．また，システムの TOP レベルは PE と通信（変数・通信チャネル）のみ存在し，各々の PE は並列に動作するようになる．

5.1.3 SpecC コードの例

誌面の都合上，SpecC コードの例は 2.PE への割り付け・動作のグループ化，3. 同期の挿入についてのみ説明する．また，仕様モデル中の各リーフ動作，通信チャネルの実装コードはここでは割愛する．なお，コードの下線部分が追加・変更した箇所を示している．

まず，本研究で使用した動作モデル [1] を図 3 に示す．

動作モデルの説明

この動作モデルは，まず，動作 B1 が処理を行い，その出力である変数 v1 を用いて，動作 B2 と B3 が通信

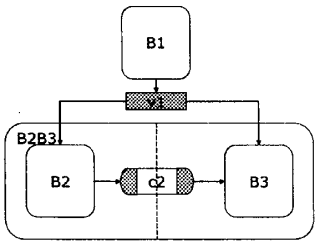
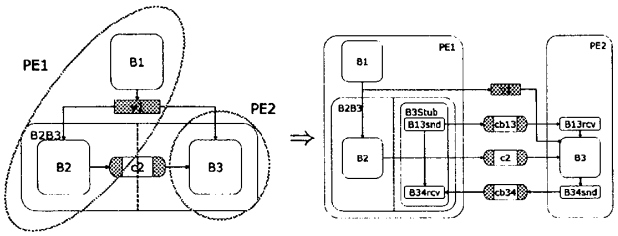


図 3: 本研究で使用した動作モデル

チャネル c2 を介して通信しながら，並列に動作しているモデルである．

図 3 に対して，PE の割り付けが図 4 の「PE の割り付け（前）」に示すように決定した場合を考える．先にも述べた通り，現状は図 4 の変換が 1 ステップで行われている．



PE の割り付け（前）　　PE の割り付け（後）

図 4: PE の割り付け（現状）

1. 新たな階層の導入（図 5）
   新たな階層として PE1，PE2 を導入する．また，TOP レベルに新たに作成した階層を追加する．

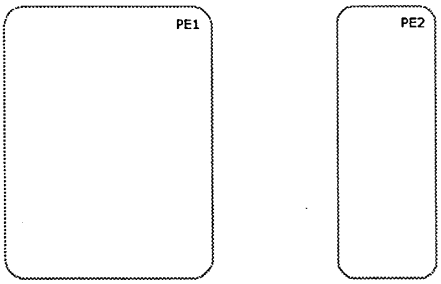


図 5: 新たな階層の導入

2. PE への割り付け・動作のグループ化（図 6）
   ここでは，PE1 に動作 B1 と B2B3 を割り付け，PE2 に動作 B3 を割り付ける．その際，並列動作の一部である動作 B3 を PE2 に割り付けたことにより，空白となった PE1 の上位動作 B2B3 に新たな動作 B3Stub を挿入する．
   まず，PE1 に動作 B1 と B2B3 を割り付ける．

PE1（変更前）
```
behavior PE1(){
  void main(void){
  }
};
```

⇓

動作の追加

PE1（変更後）
```
behavior PE1(){
```

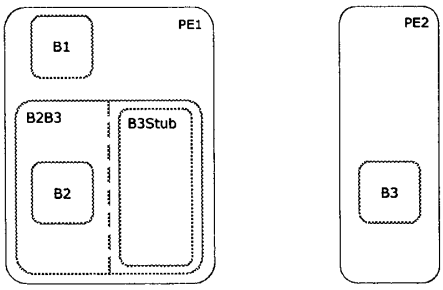


図 6: PE への割り付け・動作のグループ化

```
  B1 b1(v1);
  B2B3 b2b3(v1);
  void main(void){
    b1.main();
    b2b3.main();
  }
};
```

次に，PE2 に動作 B3 を割り付ける．

PE2（変更前）
```
behavior PE2(){
  void main(void){
  }
};
```

⇓

動作の追加

PE2（変更後）
```
behavior PE2(){
  B3 b3(v1, c2);
  void main(void){
    b3.main();
  }
};
```

ここで，新たな動作 B3Stub を定義する．

動作 B3Stub の定義
```
behavior B3Stub(){
  void main(void){
  }
};
```

そして，動作 B2B3 内の動作 B3 を動作 B3Stub で置き換える．ここでは，通信チャネル c2 がすでに割り付けられているが，これは，4. 通信の移動で TOP レベルに移動させる．

動作 B2B3（変更前）
```
behavior B2B3(in int v1){
  C2 c2;
  B2 b2(v1, c2);
  B3 b3(v1, c2);
  void main(void){
    par{
      b2.main();
      b3.main();
    }
  }
};
```

⇓

リファクタリングの基本規則 [3]
フィールド/メソッドの移動（参照の置き換え）

動作 B2B3（変更後）
```
behavior B2B3(in int v1){
```

```
  C2 c2;
  B2 b2(v1, c2);
  B3Stub b3stub();
  void main(void){
    par{
      b2.main();
      b3stub.main();
    }
  }
};
```

3. 同期の挿入（図 7）
動作 B3 を PE2 側へ割り付けたことにより，PE1 側の動作 B3Stub と PE2 側の動作 B3 の間で実行の同期を取る必要があるため，これらの動作中に同期通信チャネルを挿入する．

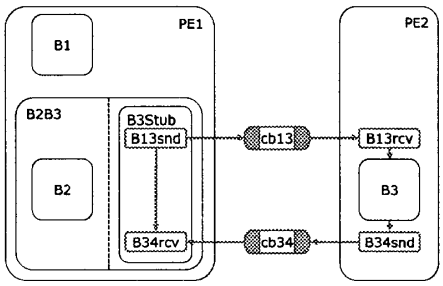


図 7: 同期の挿入

まず，同期通信チャネルを定義する（実装は空のメッセージを送受信）．

同期通信チャネルの定義 send 側

```
behavior BSnd(ISend ch){
  void main(void){
    ch.send(0, 0);
  }
};
```

同期通信チャネルの定義 receive 側

```
behavior BRcv(IRecv ch){
  void main(void){
    ch.recv(0, 0);
  }
};
```

そして，動作 B3Stub にこちら側から先に送信する同期通信チャネルを挿入する．

動作 B3Stub（変更前）

```
behavior B3Stub(){
  void main(void){
  }
};
```

⇓

動作の追加

動作 B3Stub（変更後）

```
behavior B3Stub(ISend cb13, IRecv cb34){
  BSnd b13snd(cb13);
  BRcv b34rcv(cb34);
  void main(void){
    b13snd.main();
    b34rcv.main();
  }
};
```

PE2 も同様（省略）．

4. 通信の移動（図 8）
変数 v1 と通信チャネル cb13，c2，cb34 を TOP レベルに移動させる．それに応じて，PE1 側は本来動作 B3 が存在していた B3Stub まで，PE2 側は PE2 まで下位動作の引数を更新する．

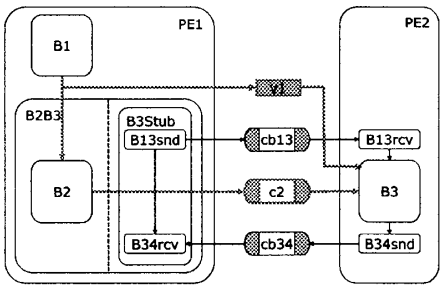


図 8: 通信の移動

6. 結論

本研究では，システムレベル設計において，現状では体系化されていない段階的詳細化におけるモデル変換規則を定式化し，その適用手順・方法をリファクタリングカタログとしてまとめた．結果として，約 30 個のリファクタリング規則を定式化できた．しかし，現状では，リファクタリング規則を定式化していない場合や，定式化が困難な場合がある．以下にその主な場合を示す．

- ライブラリ等からの部品挿入時
- 変数分割において，広域共有変数を専用の共有メモリ・コンポーネント上に割り当てる場合
- スケジューリングにおいて，動的スケジューリングを行う場合
- 互換性のないシステムを合成する際にトランスデューサを挿入する場合．また，そのインライン化を行う場合
- バックエンドにおける，主にハードウェア合成，（ハードウェア側）インタフェース合成を行う場合

なお，シミュレーション実行のためのタイミングの挿入に関しては，アーキテクチャ・コンポーネントに依存するため，現状ではリファクタリング規則として定式化していない．

今後の課題としては，上述のリファクタリング規則や新たなリファクタリング規則を定式化することが挙げられる．また，システムレベル設計言語向けに定式化したリファクタリングカタログを UML 向けに拡張させることも挙げられる．

参考文献

[1] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, Daniel D.Gajski 著，木下常雄 訳，“システム設計 SpecC による実現”，SpecC Technology Open Consortium, 2002.
[2] Daniel D.Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, Shuqing Zhao 著，木下常雄，冨山宏之 訳，“SpecC 仕様記述言語と方法論”，CQ 出版社，2000.
[3] Martin Fowler 著，児玉公信，友野晶夫，平澤章，梅澤真史 訳，“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，2003.
[4] 木村正裕，“システムレベル設計におけるリファクタリング規則の抽出とカタログ化”，埼玉大学卒業論文，2006.
[5] http://www.interdesigntech.co.jp/modules/tinyd0/
[6] 株式会社インターデザイン・テクノロジー，“VisualSpec Version3.5 チュートリアル”，株式会社インターデザイン・テクノロジー，2004.