C-002

システムレベル設計における通信仕様モデルの探索
Exploration of Communication Specification Models in System Level Design

木ノ嶋 崇 † **小林 憲貴** ‡ Nurul Azma Zakaria† **松本 倫子** † **吉田 紀彦** †
Takashi Kinoshima Kazutaka Kobayashi Nurul Azma Zakaria Noriko Matsumoto Norihiko Yoshida

1. はじめに

組み込みシステムの高度化，複雑化により，システムの設計が困難になっている．そのため，設計効率が問題となっている．そして，複数の組み込みシステムが通信を行うことで，システムを構築するようになったため，組み込みシステム間での通信の重要性が増大している．

組み込みシステムは HW と SW で構成されており，HW と SW の協調設計手法としてシステムレベル設計 [1][2] が考案された．システムレベル設計とは仕様から実装まで段階的に詳細化していくことにより，設計効率を向上させる設計手法である．しかし，例えば，システムレベル設計の方法論の一つである SpecC 方法論（図 1）では段階的詳細化の過程が詳細に定義されているが，アーキテクチャ探索と通信探索が切り離されていないため，通信の重要性が増大している現在では最適な探索を行うことが困難になっている．

そこで，本研究ではアーキテクチャ探索から通信探索を切り離し，通信探索の段階的詳細化の過程を考える [3][4]．以下，2. で通信探索を行う上での選択肢について述べ，3. で作成した通信モデルについて説明する．4. で作成したモデルのシミュレーション結果を述べ，最後に 5. で本研究のまとめと今後の課題を述べる．

2. 通信仕様

2.1 イベント・トリガ型，タイム・トリガ型

通信探索の段階的詳細化の過程において，様々な選択肢が考えられる．まず，大きく分けるとイベント・トリガ型とタイム・トリガ型の二つに分けられる．

イベント・トリガ型は送信要求に応じて通信を行う．複数の送信要求があった場合には衝突を回避するためにアービタが用いられる．アービタは複数の送信要求に対して，送信を行う順番を決定する処理を行う．また，受信先の振り分けにフィルタが用いられる．イベント・トリガ型は送信要求に応じた柔軟な通信が可能であるが，通信頻度が高い場合に送信待ちによる遅延が発生してしまう可能性がある．

タイム・トリガ型は送信を行う順番がスケジューラのスケジュールテーブルに定められている．そのため，スケジュールテーブルに沿った静的な通信が可能であるため，周期的な通信の場合や，リアルタイム性が求められる場合に適している．

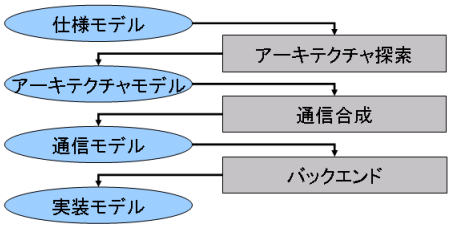

図 1: SpecC 方法論の設計プロセス

2.2 集中型，分散型

イベント・トリガ型，タイム・トリガ型のどちらも，さらに集中型と分散型に分けることができる．

集中型は一つのアービタやスケジューラで通信の管理を行う．分散型は各ノードにアービタの機能を持たせたり，スケジューラを配置することにより，通信の管理を行う．集中型は分散型より構造が簡単であり，低コストでの実装が可能であるが，分散型のほうが高速な通信が可能である．例えば，車載ネットワークについて言えば CAN はイベント・トリガ分散型，LIN はタイム・トリガ集中型，FlexRay はタイム・トリガ分散型である．

3. 通信モデルの探索

抽象的な通信モデルから段階的に詳細化していくモデル変換の過程を図 2 に示す．使用したモデルの記述には SpecC を用いた．SpecC による実装には Inter Design Technologies, Inc. の VisualSpec Version3 を使用した．なお，SystemC などでも同様な記述が可能である．

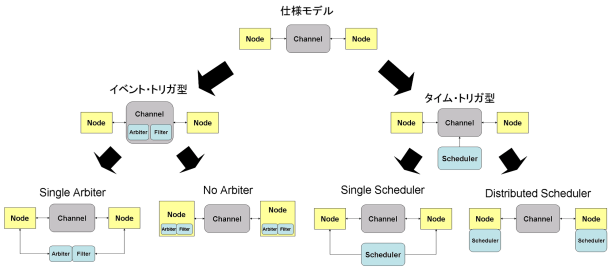

図 2: 通信モデルの探索

3.1 仕様モデル

まず，最も抽象的な仕様モデルは両側のノードがチャネルを介してデータの送受信を行うモデルである．SpecC コードの骨格を以下に記す．受信側はデータが送信されてくるのを待つ．送信側は送信元を区別する識別子 (ID) と送信したいデータを合わせてを送信し，チャネルを介して送信元に対応する識別子を持つ受信側にデータが送られる．

†埼玉大学，Saitama University
‡インターデザインテクノロジー，InterDesign Technologies, Inc.

送信側（Sender）

```
behavior Sender (I_send send) {
void main(void) {
  /*...*/
  send.send(S_data);                    // データ送信
};
```

受信側（Receiver）

```
behavior Receiver (I_receive receive) {
void main(void) {
  R_data = receive.receive(ID);      // データ受信
  /*...*/                            // 受信後の処理
};
```

チャネル（Channel）

```
channel Chnl (void) {
void send(DATA d) {
  /*...*/          // Sender からデータを受け取る
  notify(e);                              // receive へ
}
DATA receive(int id) {
  wait(e);                    // send の notify を待つ
  /*...*/        // 宛先の Receiver へデータを渡す
}
};
```

3.2 イベント・トリガ型，タイム・トリガ型

送信タイミングにより，イベント･トリガ型とタイム･トリガ型に変換することができる．モデル変換後の SpecC コードの追加，変更部分を以下に記す．イベント・トリガ型の場合，チャネルにアービタとフィルタの機能を持たせる．アービタはチャネルが現在使用中であるか把握しており，送信要求があったときにチャネルが使用中の場合，送信を待たせる処理を行う．フィルタは送られてきたデータが受信ノード宛のデータであるか判別し，受信するかどうかを判断する．

イベント・トリガ型 チャネル（Channel）

```
channel Event_Triggered (void) {
void send(DATA d) {
  arbiter.check();  // チャネルが使用中かチェック
  /*...*/          // Sender からデータを受け取る
  notify(e);                              // receive へ
}
DATA receive(int id) {
  do {
    wait(e);                  // send の notify を待つ
  }while(!filter.check(id));       // フィルタリング
  arbiter.reset();  // 受信完了時チャネルを空ける
  /*...*/                 // Receiver へデータを渡す
};
```

イベント・トリガ型 アービタ（Arbiter）

```
channel Arbiter (void) {
void check(void) {
  while(use != 0){      // チャネルが使用中の場合
    waitfor(1);          //チャネルが空くのを待つ
  }
  /*...*/          // チャネルが空いたらチャネルを
  use = 1;             // 使用中にし送信を開始する
}
void reset(void) {
  use = 0;    // チャネルを空ける
}
};
```

イベント・トリガ型 フィルタ（Filter）

```
channel Filter (in DATA d) {
int check(int id) {
  if(id == d.id) {          // 宛て先をチェックする
    return 1;                 // 自分宛の場合の処理
  } else {
    return 0;             // 自分宛でない場合の処理
  }
}
};
```

タイム・トリガ型の場合，スケジューラを追加する．スケジューラは最初にスケジュールテーブルを作成する．スケジューラは作成したスケジュールテーブルに従ってチャネルに対して現在のスロットを知らせる．チャネルはスケジューラから送られてきた現在のスロットに割り当てられている送信データに対して，宛て先の受信側へ送る処理を行う．

タイム・トリガ型 スケジューラ（Scheduler）

```
behavior Scheduler (out int slot_id) {
void init() {
  /*...*/             // スケジュールテーブルの作成
}
void main(void) {
  /*...*/
  slot_id = S_table[now_slot]; // 現在のスロットを
  /*...*/                                  // 指定
}
};
```

タイム・トリガ型 チャネル（Channel）

```
channel Time_Triggered (in int slot_id) {
void send(DATA d) {
  while(d.id != slot_id) {
    waitfor(1);      // 送信データに割り当てられた
  }                              // スロットまで待つ
  /*...*/          // Sender からデータを受け取る
  notify(e);                          // receive へ
}
DATA receive(int id) {
  wait(e);                  // send の notify を待つ
  /*...*/        // 宛先の Receiver へデータを渡す
}
};
```

3.3 集中型，分散型（イベント・トリガ型）

送受信の管理を行う構造により，イベント・トリガ型はさらに集中型と分散型に分けることができる．

集中型は一つのアービタとフィルタで通信の管理を行う．アービタとフィルタの基本的な処理は変わらないが，アービタは送信側ノードに対して送信の管理を行い．フィルタは受信ノードに対して受信の管理を行う．

分散型は各ノードにアービタとフィルタの機能を持たせる．同様にアービタとフィルタの基本的な処理は変わらないが，送信ノード自身がアービタの機能を持っているので自身で送信の管理を行う．受信の管理も同様にノード自身がフィルタの機能を持ち管理を行う．

Single-Arbiter 送信側（Sender）

```
behavior Sender (I_send send,
arbiter_check check_use) {
void main(void) {
  /*...*/
  check_use.check();         // アービトレーション
  send.send(S_data);                 // データ送信
}
};
```

Single-Arbiter 受信側（Receiver）

```
behavior Receiver (I_receive receive,
filter_check check_id) {
void main(void) {
  R_data = receive.receive(ID);    // データ受信
  if(check_id.fil_check(ID,R_data.id)) { // フィルタ
                                        // リング
    /*...*/          // 自分宛のデータの場合の処理
  }
}
};
```

No-Arbiter 送信側（Sender）

```
behavior Sender (I_send send) {
Arbiter arbiter_sender;          // アービタの機能
void main(void) {
  /*...*/
  arbiter_sender.main();    // アービトレーション
  send.send(S_data);                 // データ送信
}
};
```

No-Arbiter 受信側（Receiver）

```
behavior Receiver (I_receive receive) {
Filter filter_receiver(R_data, flag);  // フィルタの
void main(void) {                             // 機能
  R_data = receive.receive(ID);      // データ受信
  filter_receiver.main();          // フィルタリング
  if(flag == 1) {    // 自分宛の場合 flag=1 を返す
    /*...*/          // 自分宛のデータの場合の処理
  }
}
};
```

3.4 集中型，分散型（タイム・トリガ型）

イベント・トリガ型と同様にタイム・トリガ型も集中型と分散型に分けることができる．

集中型は一つのスケジューラで通信の管理を行う．スケジューラの機能は変更せず，各ノードがスケジューラから現在のスロットを受け取ることにより，スケジュールテーブルに従った通信を行う．

分散型は各ノードにスケジューラを配置して通信の管理を行う．各スケジューラは同一のスケジュールテーブルを持っており，それぞれのノードに現在のスロットを知らせることにより，スケジュールテーブルに従った通信を行う．

Single-Scheduler 送信側（Sender）

```
behavior Sender (in int slot_id, I_send send) {
void main(void) {
  /*...*/
  while(S_data.id != slot_id) {
    waitfor(1);      // 送信ノードに割り当てられた
  }                              // スロットまで待つ
  send.send(S_data);                 // データ送信
}
};
```

Single-Scheduler 受信側（Receiver）

```
behavior Receiver (I_receive receive) {
void main(void) {
  R_data = receive.receive(ID);       // データ受信
  if(R_data.id == ID) {
    /*...*/            // 自分宛のデータの場合の処理
  }
}
};
```

Distributed-Scheduler 送信側（Sender）

```
behavior Sender (in int slot_id_sender,
I_send send) {
void main(void) {
  /*...*/
  while(S_data.id != slot_id_sender) { // 各ノード
    waitfor(1);                  // のスケジューラから
  }                        // 現在のスロットを受け取る
  send.send(S_data);                   // データ送信
}
};
```

4. 実験

通信モデルの探索によって，作成した各モデルについて，具体的な例題システムを SpecC で記述し，動作確認を行った．例題システムは 2 つの送信ノードとそれぞれの送信ノードに対応する受信ノード 2 つとチャネルで構成される．

まず，仕様モデルにおいて，対応するノード間で正しく送受信が行われているか確認を行う．動作確認の結果を図 3 に示す．図はシミュレーション上の単位時間ごとの各ノードの処理，チャネルでの通信の様子を表している．

仕様モデルでは各送受信の組に専用のスロットが用意されているため，通信の衝突は起こらない．図 3 より，仕様モデルでの送受信が正しく行われていることが確認できる．

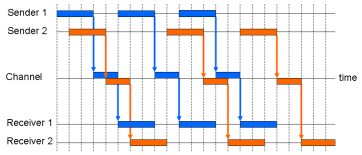

図 3: 仕様モデル

次に，イベント・トリガ型の動作確認の結果を図 4, タイム・トリガ型の結果を図 5 に示す．スケジュールテーブルはその時間に割り当てられているスロットを色で表している．それぞれ，アービトレーション，スケジューリングが正しく行われているか確認を行う．

イベント・トリガ型では，チャネルが使用中の場合にはチャネルが空くまで待っているのがわかる．タイム・トリガ型では，送信したいデータに割り当てられたスロットでなかった場合，割り当てられたスロットまで待っているのがわかる．よって，アービトレーション，スケジューリングが正しく行われていることが確認できる．

また，集中型，分散型についても同様に動作確認を行い，正しく動作していることが確認できた．

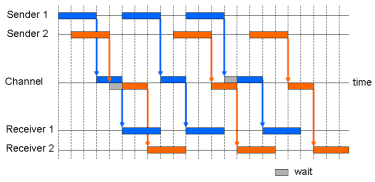

図 4: イベント・トリガ型

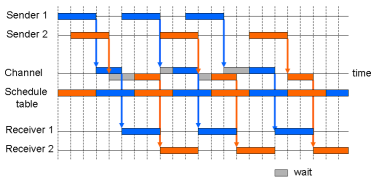

図 5: タイム・トリガ型

5. まとめ

通信システムの設計効率を向上させるために，通信仕様モデルの探索を行った．また，作成した各モデルについて動作確認を行い，モデル変換が正しく行われていることを示した．組み込みシステムネットワーク設計において，アーキテクチャ探索と同様に，通信方式の探索も段階的に進めることができるようになり，手戻りの削減や通信モデルの部品化の促進なども期待できる．現在は，モデル記述を SpecC からさらに抽象化して，実行可能 UML による記述に取り組んでいる．今後の課題としては，探索における見積もりの明確な評価基準を定める必要がある．

参考文献

[1] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, Dainiel 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, 富山宏之 訳, “SpecC 仕様記述言語と方法論”, CQ 出版社 (2000).

[3] Kazutaka Kobayashi, Takashi Shiraishi, Nurul Azma Zakaria, Ryosuke Yamasaki, Norihiko Yoshida, and Shuji Narazaki, “Exploration of Communication Models in Design of Distributed Embedded Systems”, IEEJ Transactions on Electrical and Electronic Engineering, accepted (May, 2007).

[4] 木ノ嶋崇, “システムレベル設計における通信仕様モデルの探索”, 埼玉大学卒業論文, 2007.