2009-ARC-181／2009-EMB-11（8）
2009/1/14

LTSA と SPIN を連携させたタスク設計の提案

藤倉 俊幸[†] 野中 哲[‡] 宇佐美 雅紀[†]

†イーソル株式会社 〒164-8721 東京都中野区本町 1-32-2
‡タオベアーズ合同会社 〒231-0023 横浜市 中区 山下町 1番地 シルクセンター 921
E-mail: †{t-fujikura, m-usami}@esol.co.jp, ‡nonaka@taobears.com

タスク分割および基本的な同期構造の設計および検証を LTSA でおこない．次に，検証済みの LTSA モデルを promela に変換し SPIN を使用してタスク詳細設計をおこなう手法を提案する．LTSA モデルは要求分析の結果作られる，シーケンス図や状態図から生成する．変数等を使用しない抽象度の高いタスク動作の検証を LTSA で実行する．SPIN ではグローバル変数や非同期通信を利用した詳細設計レベルの検証をおこなう．LTSA と SPIN の特徴を活かしたタスク設計法を提案する．

キーワード LTSA，SPIN，タスク設計

1. はじめに

ソフトウェアアーキテクチャには，モジュール構造を決定する静的な側面とタスク構造を決定する動的な側面が存在する．それぞれを静的アーキテクチャおよび動的アーキテクチャと呼ぶことにする．並列・並行動作を前提とする組込みソフトウェアでは動的アーキテクチャが重要である．高品質の組込みソフトウェアを効率よく開発するためには動的アーキテクチャの設計および検証を支援することが求められる．

一般的に複雑な処理をおこなうソフトウェアを開発する場合，まず静的アーキテクチャを決定してから動的アーキテクチャを決定することが多い．複雑な処理とは，デバイスによって初期化の手順が決められていたり，例外が発生した場合の対応方法が動作制約として指定されていたりする場合などである．静的アーキテクチャ決定では，最初にコンポーネント分割やクラス抽出をおこない，ソフトウェアを構成する静的構成要素を決定し，次に抽出された構成要素に処理を割り当てる．具体的には，特定の役割を担うクラスを抽出して，そのクラスに操作を割り当てる．この際には要求分析の結果得られた操作の実行順序を考慮し検証をおこなうことができる．ただし，各クラスが並列に動作するようなセマンテックスの下で検証がおこなわれる．

現実の実装環境における動作単位は静的構成要素ではなく，RTOS 等により作り出されるタスクあるいはスレッドである．従って，静的アーキテクチャ設計から動的アーキテクチャ設計に移る際には動作単位の変換が必要になる．また，動作単位が変更になるだけではなく，動作セマンティクスも要求と実装では異なる．要求段階では，実世界の動作をモデリングする必要があるため，動作は共通イベントや共通事象として扱われる．例えば，金槌で釘を打つ場合の「打つ」動作は金槌と釘のどちらに属するのか問うことはナンセンスである．一方，実装段階における動作は，関数呼び出しやメッセージ通信であり，送信側と受信側が明確に分かれる．

動作セマンテックスの違いは，各コンポーネントの動作を合成してシステム全体の動作を生成する場合の合成方法に対応させることができる．要求段階には同期合成が対応し，ソフトウェアの実装段階には非同期合成が対応する．このことは，モデル検査ツールを導入する際のツール選択に影響を与える．

本研究では，要求段階の振る舞いモデルを LTSA[1] で作成し初期の静的アーキテクチャを構築および検証する．次に，初期の静的アーキテクチャをタスクに分割し，分割されたタスクモデルを promela[2]の proctype で表現する．

単純なシステムの場合では直接 promela でタスク動作を記述できる．しかし，個々のデバイスに対する制約条件がある場合や，複数の動作シナリオを実装しなければならない場合は，まず役割ベースの静的モデリングをおこないモジュールを LTSA のプロセスにより表現し基本的な動作に矛盾がないことを確認した方が効率的である．更に，要求レベルのシステム全体の振る舞いモデルを検証に利用することで効率よく品質の高いソフトウェアの設計が可能となる．また，promela は関数呼び出し動作を表現するには向いていないため，静的構造をベースとして動作モデルを構築することが困難である．LTSA でタスク分割を実施することで静的構造を動的構造に組み替えることができるので関数呼び出し構造を promela で構築する必要がなくなる．

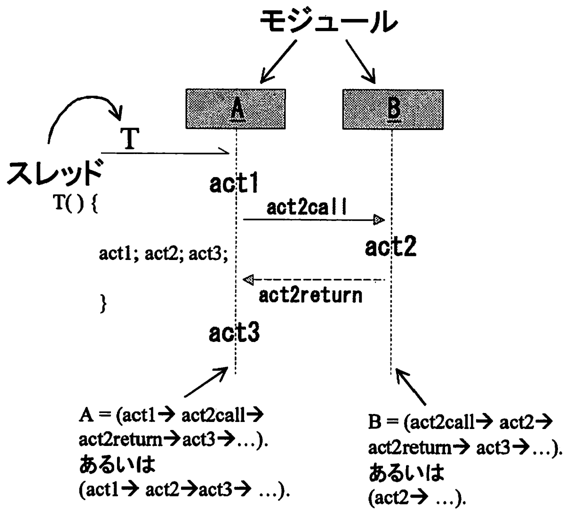


図 1. モデリングの関係

2. 提案手法

LTSAによるモジュール動作のモデリングとprolemaによるスレット動作のモデリングの関係を図 1に示す. LTSA による動作モデリングは動作共有を前提としておこなわれるため，モジュール A と B の間で act2call と act2return，または簡略化して act2 が同時に実行される様なモデリングになる．この形式は，要求分析段階で実世界の物の間の協調動作を記述するには都合が良いが，実際のソフトウェアの非同期な動きを記述するには向いていない．promela の proctype は，変数共有やランデブ，メッセージ通信をグローバル変数や通信チャンネルを使用してモデル化することができる．しかし，モデル化対象のスレッドの動きを act1; act2; act3 の様に明確に指定する必要がある．proctype で関数呼び出しを含んだ動作をモデリングする場合, return するまで呼び出し側を待たせなければならない．例えば，以下の様なモデリングが必要になる．

```
#define SYNC 0

mtype = {jts, rtn};
chan funCh = [SYNC] of {mtype};

proctype   fun()
{
mtype call;

funCh?call;
call==jts;
funCh!rtn;
}

proctype   caller()
{
mtype func;
funCh!jts;
funCh?func;
}

init{
	run fun();
	run caller();
}
```

関数 fun を proctype にすることが不自然であるし，関数の実行が終わるまで caller が待つことをモデル化しているが，そもそもスレッドは関数呼び出しで待つことはない．関数呼び出しで呼び出し側が待つと言うのは，静的なモデリングにおける振る舞い記述の都合上導入される概念であり実体のないものである．

promela の proctype はスレッドの動きを忠実にモデル化するべきであり，関数呼び出しを自然な形で取り込む必要がある．本研究では LTSA で振る舞い記述をおこない，実体のない待ちを取り除いたスレッドの動作パターンを生成し，それを promela で記述する．図 1 の場合にはスレッドの動作パターンは単純であるが，動作制約が増えると全てのパターンに対応することは困難になるため LTSA で可能なパターンを生成する方が効率的で正確である．

手順は以下の様になる．

1. LTSA による動作モデルの作成する．これを BaseModel とする．
2. BaseModel からプロジェクションによりスレッドの動作パターンを生成する[3].
3. スレッド動作パターンを promela に変換する．
4. BaseModel から検査式を生成して promela モデルを検証する．

BasModel の作成は，LTSA の教科書である文献[1]に従っておこなう．スレッドの動作パターン生成は，BaseModel からスレッドにアサインする処理を公開インタフェースとして指定して双模倣な状態マシンにミニマイズすることでおこなう．この処理は LTSA に実装されている．この際に，内部動作が tau として残る場合がある．その際は，公開するインタフェースが不

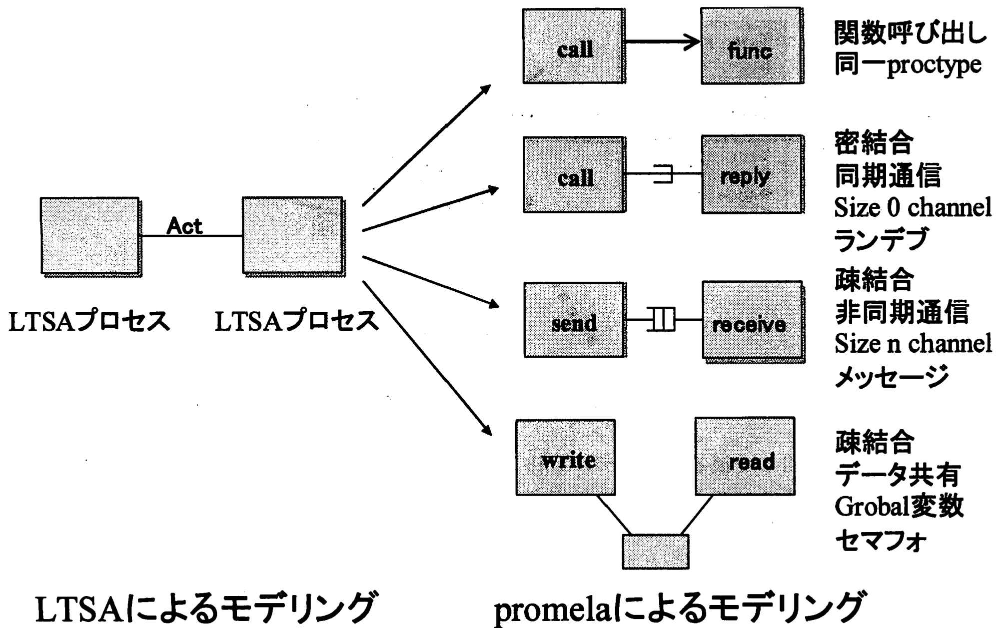
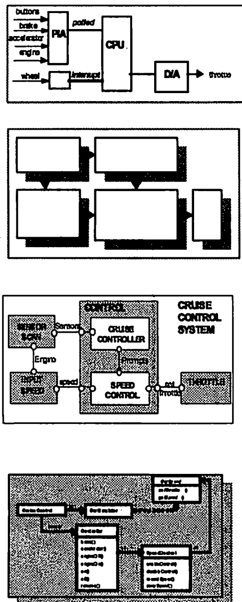


図 2. 同期構造の設計

十分であるので，tau が無くなるまで，必要なインタフェースを追加して再度ミニマイズ処理を実施する[3]. tau が無くなることで一つの閉じた動作モデルができあがる．これを promela に変換する．現状では promela 変換は手でおこなっているが単純な状態遷移系なので変換プログラムによって自動化することは容易である．

この段階で，先に示した関数呼び出しを必要としない proctype のテンプレートができあがる．その後は，関数呼び出し以外のスレッド間の相互作用を promela でモデル化し，BaseModel を検査式として検証をおこなうことでスレッド仕様が完成する．LTSA では共有



ハード構成
コンテキスト図

コンポーネント分け
インタフェース決め

役割によるモデリング
詳細化

モジュール設計
実装

■ 静的アーキテクチャ構築と動的アーキテクチャ構築を並行して実施する。

提案手法

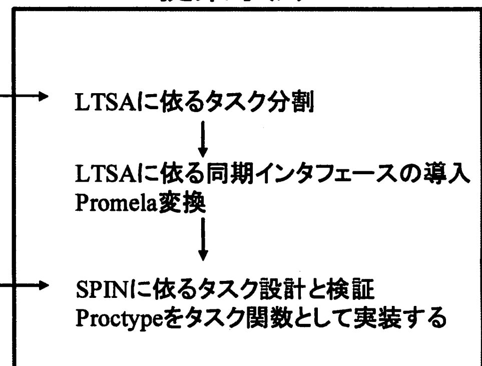


図 3. 文献[1]における設計の流れと本手法の関係

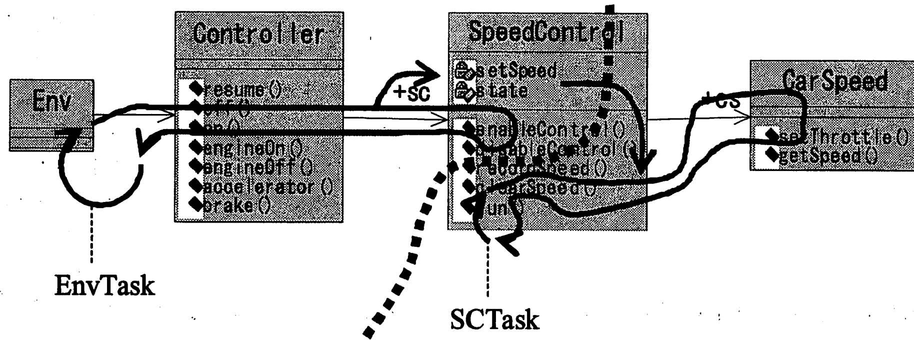


図 4. クルーズコントロールにおける静的構造と動的構造

動作としての相互作用を利用して振る舞いモデルを作成したが，promela では実際のソフトウェアに対応させた相互作用のモデル化が可能になる(図2)．関数呼び出しのみを LTSA でモデル化しておくことが本手法の特徴である．

3. 適用例

本手法の適用事例を示す．一つは，文献[1]のクルーズコントロールである．文献[1]では LTSA のモデルから Java プログラムを作成しているが LTSA モデルと Java ソースコードの間にはギャップがある．この手法に依ればそのギャップを埋めることができる．もう一つは，静的には単純だが動的な相互作用はクルーズコントロールよりも複雑でインタフェースの指定だけでは tau が残ってしまう事例である．

3.1. クルーズコントロール

文献[1]において設計は図 3 の左側に示したように進み静的構造を決定し Java のスレッドを二つ使用して動作させる．そのスレッドの動きをリバースすると図 4 の様になっている．ここでは，それぞれ EnvTask と SCTask と呼ぶことにする．それぞれのタスクは SpeedControl クラス内の setSpeed と state 変数によってインタフェースを取っている．それぞれのスレッドの promela テンプレートを得るために LTSA で以下の様にして状態マシンを生成させる．

||EnvTask = (BaseModel) @{accelerator, brake, clearSpeed, disableControl, enableControl, engineOff, engineOn, off, on, recordSpeed, resume}.

||SCTask = (BaseModel) @{setThrottle, speed,zoom}.

この事例の場合には，スレッド用の状態マシンを生成しても tau が残ることはないので EnvTask と SCTask 間インタフェースを追加する必要は無い．

EnvTask と SCTask を合成して BaseModel と動作を比較することでスレッドの設計検証をすることができる．この場合，LTSA モデルの speed アクションは Java では getSpeed 関数で実装されている．この関数を呼び出すタイミングは，クルーズコントロール開始後になっている．EnvTask と SCTask を合成すると getSpeed 関数の呼び出しが早すぎる問題点が検出される．Java コードでは，enableControl 関数の中で SCTask を生成しているのでこの問題は発生しない．LTAS モデルと Java コードの間には文献[1]で説明されていない実装が幾つかあるがそれらは EnvTask と SCTask の動作を比較することで説明することができる．

3.2. 動的な相互作用が必要な事例

図 5 に示したような二つのコンポーネントが相互作用をする場合を考える．二つのコンポーネント間は b と d を共有している．また，静的には二つのクラスで実装され，b と d の相互作用は関数呼び出しとし b の

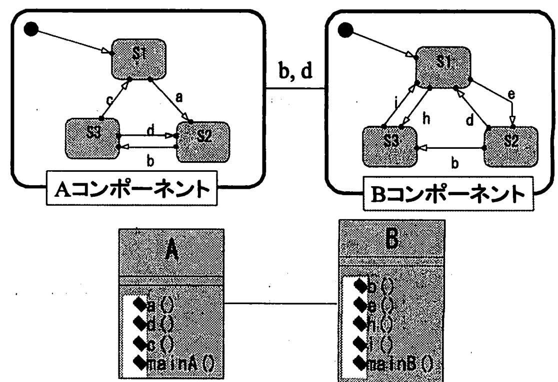


図 5. 動的な相互作用が必要な事例

実体はクラス B に有り、d の実体はクラス A にあるとする.

図 5 に示した状態マシンを LTSA でモデル化してコンポーネント A,B の要求レベルの動作モデル AReq, BReq とする.この二つを合成することで全体の動作モデル(BaseModel)を得る. 動的構造としてはスレッドを二つ使用して, それぞれ mainA と mainB で動作させることとして,それぞれで実行する関数を割り当てる. mainA では a, b, c を実行し, mainB では e, h, d, i を実行するものとする.

ここでの設定は, サンプルとして利用するためであり必然性は無い. 例えば, コンポーネント A,B は何らかのデバイスに対応していて, 各関数はそのデバイスを制御するためのものである. 関数 d は例外が起きた場合の処理で, コンポーネント A では状態 S2 からやり直すことになるがコンポーネント B では状態 S1 まで戻らなければならない. と言うような動作制約が課せられていると考えれば良い. そして, この制約を満足するような mainA と mainB の設計をしたい.

まず, LTSA でそれぞれの関数の proctype テンプレートを得るために

```
minimal ||MainA0 = (BaseModel) @{a, b, c}.
minimal ||MainB = (BaseModel) @{e, h, d, i}.
```

のようにモデル化する. この場合, 図 6 の様に tau が残ってしまう.

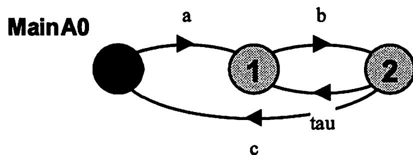


図 6. mainA テンプレート

これは、コンポーネント A の状態が、タスク mainA の知らない間にタスク mainB の関数 d の呼び出しによって勝手に状態 2 から 3 に遷移してしまうことを表している。このことから, mainA の状態を記述するためには d も考慮する必要があることが分かる. そこで, 以下の様に変更する. この様にすると tau の部分は d になり tau は無くなる.

```
minimal ||MainA = (BaseModel) @{a, b, c, d}.
```

mainB についても同様にモデル化することで, a と c を追加する必要があることが分かる. 最終的な状態マシンを図 7 に示す.

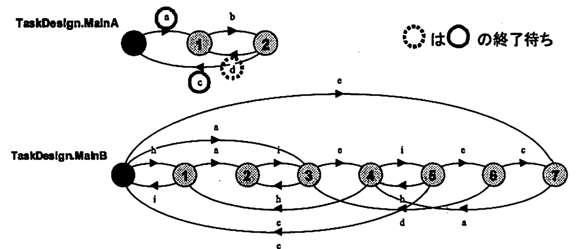


図 7. mainA と mainB の proctype テンプレート

このテンプレートから promela モデルを作成すると以下の様になる.

```
#define SYNC 0
mtype = {msg};
chan syncA = [SYNC] of {mtype};
chan syncC = [SYNC] of {mtype};
chan syncD = [SYNC] of {mtype};

active proctype main_a(){
s0:
        syncA ! msg;
        printf("a¥n");
s1:
        printf("b¥n");
s2:
        if
        :: syncD ? _ -> goto s1;
        :: syncC ! msg -> printf("c¥n");goto s0;
        fi;
};

active proctype main_b(){
s0:
        if
        :: printf("e¥n") -> goto s7;
        :: printf("h¥n");
        :: syncA ? _ -> goto s3;
        fi;
s1:
        if
        :: syncA ? _;
        :: printf("i¥n");goto s0;
```

```
        fi;
s2:
        printf("i¥n");
s3:
        if
        :: printf("e¥n");
        :: printf("h¥n") -> goto s2;
        fi;
s4:
        if
        :: printf("i¥n");
        :: syncC ? _ -> goto s1;
        fi;
s5:
        if
        :: printf("h¥n") -> goto s4;
        :: printf("e¥n");
        :: syncC ? _ -> goto s0;
        fi;
s6:
        if
        :: syncC ? _;
        :: syncD ! msg -> printf("d¥n");goto s3;
        fi;
s7:
        syncA ? _;
        goto s4;
};
```

このpromelaモデルをベースにして残りのタスク間相互作用を設計していく.関数呼び出しの部分をLTSAでモデル化することでpromelaではチャンネルとグローバル変数を利用したモデリングに専念することが可能となる.

4. まとめ

要求レベルの振る舞いから関数呼び出しパターンを解決する部分までをLTASでモデル化し，その後のタスク詳細設計をpromelaでおこなう手法について提案した.

LTSAからpromelaへの自動変換と，LTSAのpropertyを利用したsafetyチェック機能をpromelaのneverオートマトン等に変換する手法については今後の課題としたい.

文　献

[1] J. Magee, J. Kramer, Concurrency – State Model & Java Programs (2ed.), Wiley 2006.

[2] G. J. Holzmann, The SPIN Model Checker, Addison-Wesley 2004.

[3] 藤倉俊幸,"・オーバーラップ制御の設計と検証,"第四回システム検証の科学技術シンポジウム予稿集, pp.97-100, 2007. http://unit.aist.go.jp/cvs/symposium/verification2007/proceedings%204th%20symposium.pdf