Vol.2011-SLDM-149 No.11
Vol.2011-EMB-20 No.11
2011/3/18

マルチプロセッサ対応RTOS向けテストプログラム生成ツールにおけるプロセッサ間同期の実現

浅見侑太[†1]　鴫原一人[†2]　本田晋也[†2]
山本晋一郎[†1]　高田広章[†2]

我々は，マルチプロセッサ対応 RTOS が提供する API の信頼性を検証するためのテストプログラムにおいて，必要となるプロセッサ間同期処理を生成する機能を実装した．同期処理を行う状況を同期パターンとして 10 種類にまとめ，同期処理を行う方法を同期メカニズムとして 4 種類にまとめた．そして，開発された API のテスト 2,542 件を全て通過することを確認した．

Support for Inter-processor Synchronization in the Test Program Generator for Multiprocessor RTOS

Yuta ASAMI,[†1] Kazuto SHIGIHARA,[†2]
Shinya HONDA,[†2] Shinichiro YAMAMOTO[†1]
and Hiroaki TAKADA[†2]

We implemented support for inter-processor synchronization needed to verify the reliability of the API offered by multiprocessor RTOS. We identified 10 different synchronization patterns and 4 different synchronization mechanisms. We implemented 2,542 different test scenarios for testing the API of a multiprocessor RTOS and confirmed their successful execution.

†1 愛知県立大学
Aichi Prefectural University
†2 名古屋大学
Nagoya University

1. はじめに

名古屋大学附属組込みシステム研究センター (NCES)[1)] では，コンソーシアム型共同研究として，複数の企業と団体の協力を得て TOPPERS プロジェクト[2)] で開発されている RTOS の検証環境の開発を進めている．RTOS の品質を保証するために様々なテストが考えられるが，コンソーシアム型共同研究では，RTOS がユーザに提供する API が仕様書通りに正しく振舞うことをテストする API テストを開発している．API の実装は RTOS のソースコードの大部分を占めており，RTOS が保証するべき最も重要な品質であるといえる．API テスト開発の第一段階として，シングルプロセッサ対応 RTOS の TOPPERS/ASP カーネル (以下，ASP カーネル) に対して API テストのテストスイートとテストプログラム生成ツール TTG(TOPPERS Test Generator)[3)] を開発した．TTG でテストプログラムの生成を自動化することにより，テスト実施の効率化を確認できた[4)]．

API テスト開発の第二段階として，ASP カーネルをマルチプロセッサ向けに拡張した TOPPERS/FMP カーネル (以下，FMP カーネル) に対する API テストの開発と実施を進めている．それに伴い，FMP カーネル向けのテストプログラムの生成を可能とするよう，TTG を拡張した．マルチプロセッサ用のテストプログラムにおいては，様々な種類のプロセッサ間同期処理を適切に用いる必要がある．同期処理の設計や使用は非常に複雑であるので，TTG により，正しくプロセッサ間同期処理が行われているテストプログラムが生成できれば，シングルプロセッサ用のテストプログラム生成以上に，テストスイート開発者の負担を減らすことができる．

本論文では，TTG のプロセッサ間同期処理の生成機能の設計について述べる．設計にあたっては，マルチプロセッサ向けのテストプログラムを分析することで，同期の状況 (同期パターン) と手法 (同期メカニズム) に分けてそれぞれにどのようなバリエーションが存在するか整理した．同期パターンを 10 種類，同期メカニズムを 4 種類にまとめ，TTG に実装した．結果として，FMP カーネル用の 2,542 件のテストケース全てを正しくテストプログラムとして実現することができた．本機能は FMP カーネルのための機能として設計したが，FMP カーネル特有の機能は使用しておらず，他のマルチプロセッサ対応 RTOS に対しても適応できる．

本研究は，「OJL による最先端技術適応能力を持つ IT 人材育成拠点の形成」[5)6)] の OJL(On the Job Learning) の開発テーマとして実施した．

本論文の構成は，次の通りである．2 章で TTG と入出力ファイルの形式について述べ，

© 2011 Information Processing Society of Japan

3 章で設計したプロセッサ間同期について述べる．そして 4 章で結果の評価を述べる．

2. TOPPERS Test Generator

2.1 TTG の概要

API テストでは，TOPPERS 新世代カーネル統合仕様書[7] から手作業によって確認すべきテストケースの抽出を行い，テストシナリオとして記述する．TTG は，このテストシナリオを入力として読み込んでテストプログラムを生成する．複数のテストシナリオをまとめて読み込み，1 つのテストプログラムとして生成することも可能である．

2.2 テストシナリオの形式

API におけるテストケースの例を，**図 1** に示す．テストシナリオは，仕様から抽出したこのようなテストケースを具体化したものである．API テストでは，API 発行によるシステム状態の変化を確認する．具体的には，テストプログラムにて，API 発行前のシステム状態 (前状態) を実現し，その状態でテスト対象となる API を発行 (処理) し，API 発行後のシステム状態 (後状態) を確認する．このようにテストプログラムの実行内容を「前状態」,「処理」,「後状態」の 3 つの項目に分けて表現したものを，API テストのテストシナリオとする．処理は複数記述することができるが，その場合は，それぞれに対応する後状態を記述する必要がある．

図 1 のテストケースを，テストシナリオで表現したものを**図 2** に示す．プロセッサ 1 には TASK1，プロセッサ 2 には TASK2 が存在して，前状態は，TASK1 が実行されている状態 (実行状態)，TASK2 が起床待ち状態である．処理は，TASK1 が TASK2 に対して wup_tsk を発行し，戻り値としてエラーコード E_OK を受け取ることである．処理の結果，TASK2 が実行状態となることを，後状態で定義している．

テストシナリオを形式的に記述するために，TESRY(TEst Scenario for Rtos by Yaml) 記法[3] という独自の記法が考案された．TESRY 記法は，テストシナリオを YAML 形式[8] で記述したものである．YAML を採用した理由は，テストシナリオをデータ構造で階層的に表現できること，改行とインデントによって構造を表すので可読性が高いこと，などである．TESRY 記法では，システム状態を，前状態として “pre_condition” 内に，後状態として “post_condition” 内に記述し，発行する API とその引数と戻り値を “do” 内に記述する．なお，前状態と後状態とで変化しないパラメータは省略可能とする．TESRY 記法によって記述されたテストシナリオは TESRY データと呼称する．pre_condition，post_condition には，テスト対象とする ASP カーネルおよび FMP カーネルに存在する全カーネルオブジェクト，および CPU ロック状態やディスパッチ禁止状態などのシステム状態が記述可能である．do には，テスト対象とする API を，発行する処理単位 (id)，引数を含む実行コード

```
他プロセッサの起床待ち状態のタスクに
対して呼び出した場合，対象タスクが
実行状態になること．
```

図 1 wup_tsk のテストケース例
Fig. 1 Example of a Test Case for wup_tsk

```
前状態
  プロセッサ 1 の TASK1 が実行状態
  プロセッサ 2 の TASK2 が起床待ち状態
処理
  TASK1 が wup_tsk(TASK2) を発行し，
  エラーコードとして E_OK が返る
後状態
  TASK2 が実行状態となる
```

図 2 wup_tsk のテストシナリオ例
Fig. 2 Example of a Test Scenario for wup_tsk

```
pre_condition:
  TASK1:
    type   : TASK
    tskstat: running
    prcid  : 1
  TASK2:
    type   : TASK
    tskstat: waiting
    wobjid:  sleep
    prcid  : 2
do:
    id     : TASK1
    syscall: wup_tsk(TASK2)
    ercd   : E_OK
post_condition:
  TASK2:
    tskstat: running
```

図 3 wup_tsk の TESRY データ例
Fig. 3 Example of The TESRY Notation for wup_tsk

```
01: void main_task(){
02:   mact_tsk(TASK2, 2);
03:   state_sync(TASK2, TTS_WAI);
04:   mact_tsk(TASK1, 1);
05:   slp_tsk();
06:
07:   ter_tsk(TASK1);
08:   mig_tsk(TSK_SELF, 2);
09:   ter_tsk(TASK2);
10:   mig_tsk(TSK_SELF, 1);
11: }
12:
13: void task1(){
14:   ER ercd;
15:   T_ALT_RTSK rtsk;
16:
17:   alt_ref_tsk(TASK2, &rtsk);
18:   assert(rtsk.tskstat == TTS_WAI);
19:
20:   ercd = wup_tsk(TASK2);
21:   check_ercd(ercd, E_OK);
22:   barrier_sync(1, 2);
23:
24:   alt_ref_tsk(TASK2, &rtsk);
25:   assert(rtsk.tskstat == TTS_RUN);
26:   check_point(1, 1);
27:   wup_tsk(MAIN_TASK);
28: }
29:
30: void task2(){
31:   slp_tsk();
32:
33:   barrier_sync(1, 2);
34:   check_point_sync(1, 1);
35:   finish_sync();
36: }
```

図 4 wup_tsk のテストプログラム例
Fig. 4 Example of a Test Program for wup_tsk

© 2011 Information Processing Society of Japan

(syscall)，戻り値 (ercd) に分けて記述する．処理単位とは，タスクや割込みハンドラなどのプログラムに対応付けられるオブジェクトのことである．図 2 のテストシナリオを TESRY 記法によって記述したものを**図 3** に示す．

2.3 テストプログラムの構造

テストプログラムには，TESRY データで指定されたタスクとは別に，前状態の実現や各システム状態のチェック用のタスク (管理タスク) を用意している．管理タスクの優先度は，最も高い値に設定してある．管理タスクの存在するプロセッサをメインプロセッサ，それ以外のプロセッサをサブプロセッサと呼ぶ．

システム状態の確認にはカーネルに用意されているオブジェクト状態参照 API（以下，状態参照 API）ではなく，API テスト用の独自の状態参照関数 (代替 API) を用いた．これは，状態参照 API では確認できないカーネルオブジェクトの状態があることや，システム状態によっては状態参照 API 自体を発行できないことがあるためである．また，システム状態の確認に代替 API を用いることで，状態参照 API 自体のテストも TTG で生成できるようになるというメリットもある．このほか，想定した順序でタスクが実行されることを確認するために，テストプログラム中にはチェックポイントを設けている．チェックポイントには，実行されるべき順番に連続した番号を付与しておき，想定外の順序でチェックポイントを通過した場合，エラーを出力する．

以下に，1 つのテストシナリオを実現するテストプログラムの流れを述べる．複数の TESRY データをまとめて読み込んだ場合は，以下の処理を TESRY データの数だけ繰り返す．

(1) **前状態の実現と確認**

管理タスクは，テストに必要なシステム状態の設定を行い，前状態を実現する．ただし，必要であればサブプロセッサ側でもシステム状態の設定を行う．そして，前状態の実現後，メインプロセッサで実行状態となっている処理単位から，実現したシステム状態が前状態に記述されている内容と一致することの確認を行う．確認の結果，TESRY データで指定されたシステム状態と異なる状態であった場合は，エラーを出力する．なお，メインプロセッサに実行状態の処理単位が存在しない場合は，管理タスクからシステム状態の確認を行う．

(2) **処理の実行**

TESRY データの処理 (do) として記述された API を，指定された処理単位から実行する．戻り値が指定されていれば，指定された戻り値と一致しているかの確認を行う．

(3) **後状態の確認**

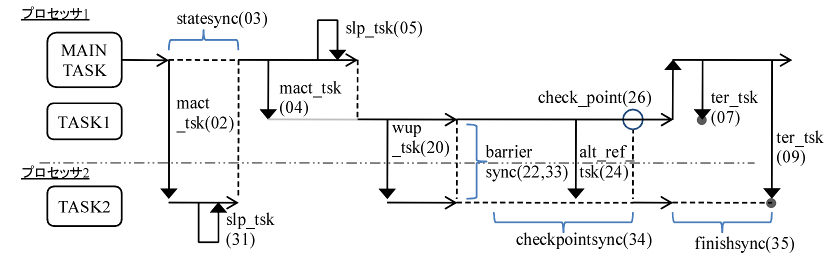

図 5 プログラムの処理フロー
Fig. 5 Processing flow of program

メインプロセッサで実行状態となっている処理単位から，システム状態が後状態に記述されている内容と一致するかを確認する．前状態の場合と同様に，TESRY データで指定されたシステム状態と異なる状態であった場合はエラーを出力する．また，メインプロセッサに実行状態の処理単位が存在しない場合は，管理タスクからシステム状態の確認を行う．

(4) **終了処理**

複数の処理 (do) が存在する TESRY データであれば，上述の処理の実行と後状態の確認を繰り返す．そして，最後の後状態では，システム状態の確認後に，各々のプロセッサで共通の終了処理を行う．これは，ディスパッチ禁止フラグの解除や割込み優先度マスクの解除など，そのプロセッサの状態を初期設定に戻すことである．最後に，管理タスクから全タスクをまとめて終了させる．

図 3 から生成されるテストプログラムを**図 4** に示す．ただし，メインプロセッサはプロセッサ 1 であり，システム状態の確認処理は TASK2 の状態チェックのみ行うように簡略化してある．また，チェックポイントは後述する同期処理のために必要なものしか記載していない．管理タスク (main_task) 以外のタスクは最初は休止状態であり，管理タスクから mact_tsk を用いて各タスクを指定プロセッサに割り付けて起動させる (L02–L04)．TASK2 は自ら slp_tsk を発行することによって起床待ち状態となる (L31)．そして，管理タスクを slp_tsk によって起床待ち状態として，TASK1 を実行状態にする (L05)．これによって前状態が実現されるので，メインプロセッサで実行中の TASK1 から状態の確認を行う (L17–L18)．その上で，処理 (do) に記載されている API を実行し，戻り値を確認する (L20–L21)．そして，後状態のシステム状態を確認する (L24–L25)．最後に，TASK1 から wup_tsk を発行し

ⓒ 2011 Information Processing Society of Japan

Vol.2011-SLDM-149 No.11
Vol.2011-EMB-20 No.11
2011/3/18

て管理タスクを起動し，管理タスクから ter_tsk によって TASK1 と TASK2 を休止状態にさせる (L07–L09)．ter_tsk は呼び出し元と同じプロセッサに所属するタスクにのみ発行可能であるので，mig_tsk によって管理タスクの割り付けプロセッサを移動させて行う．次のテストシナリオに備え，管理タスクはメインプロセッサに戻す (L10)．このプログラムには，4 回の同期処理が必要となる (L03,L22,L33–L35)．**図 5** に同期処理のフローを示す．括弧内の番号は行数である．これらの同期処理がなければタスクは次々と処理を進めてしまい，実行タイミングによってはエラーとなる可能性がある．同期処理の詳細は 3 章で述べる．

3. プロセッサ間同期の設計

3.1 プロセッサ間同期の必要性

マルチプロセッサ対応 RTOS 向けのテストプログラムでは，複数のプロセッサでタスクが動作するので，プロセッサ間の同期が必要である．すなわち，テストシナリオで指定した条件通りに動作するようにプロセッサごとの実行タイミングを合わせる必要がある．しかし，ハードウェアの性質上，バスの衝突や組み合わせるテストシナリオの順序によるキャッシュ内容の差異によって，実行タイミングに決定性はない．

テストシナリオでは，前状態，処理，後状態の間で同期が行われることを定義しており，テストプログラムで実現する必要がある．このような同期をコンディション間同期と呼ぶ．さらに，前状態が実現されてから前状態の確認を行う必要があるなど，コンディション内でも同期が必要となる．たとえば図 4 では，TASK2 が自身を対象とした slp_tsk の発行によって待ち状態に遷移するまで，TASK1 はシステム状態の確認を始めてはならない．このようなコンディション内で必要となる同期をコンディション内同期と呼ぶ．

3.2 設 計 方 針

テストプログラムにおいて，どのような状況で同期を行う必要があるのか検討を行った．そして，同期を行う状況によって分類した同期処理を同期パターンと呼ぶこととし，同期パターンの具体的な実現方法を考案した．状況によって利用可能な同期の実装は変わってくるので，複数の実装方法を用意している．同期の実装方法を同期メカニズムと呼ぶ．

RTOS は様々なターゲットシステムの上で動作するので，テストプログラムも多くのターゲットで動作することが求められる．このため，同期メカニズムは汎用性のある実装を目指すこととした．

3.3 同期パターン

同期パターンの分類結果と，その同期パターンを実現する同期メカニズムの対応表を**表 1** に示す．本節では，これらの同期パターンのうち，それぞれ異なる同期メカニズムによって実現されている StateSync, LastCheckedSync, ConditionSync, LoopTerminated について説明する．

表 1 プロセッサ間同期の設計
Table 1 Design of Inter-processor Synchronization

同期メカニズム	同期パターン
CheckPointSyncFunc	ActiveSync, LockSync, TimeSync, ExecSequenceSync, DoStartSync, LastCheckedSync
StateSyncFunc	StateSync, DoFinishSync
BarrierSyncFunc	ConditionSync
FinishSyncFunc	LoopTerminated

(1) **StateSync**

前状態において，サブプロセッサのタスクの待ち状態への状態遷移を待つコンディション内同期である．前状態においてサブプロセッサ側に待ち状態のタスクが存在する場合，管理タスクからそのタスクを起動し，対象タスク自身によって指定された状態へ遷移させる．たとえば起床待ち状態を指定された場合には，管理タスクから対象タスクを起動してから，対象タスクが自身に対して slp_tsk(待ち状態へ遷移させる API) を発行する．このとき，対象タスクが待ち状態へ遷移を終えてから管理タスクの処理を進める必要がある．

(2) **LastCheckedSync**

最後の後状態において，サブプロセッサ側でシステム状態の確認終了を待つコンディション内同期である．最後の後状態では，システム状態の確認後に，CPU ロック状態の解除やディスパッチ禁止フラグのクリアなど，各プロセッサごとに共通終了処理を行う．システム状態の確認が終わらなれければ終了処理を始められないので，サブプロセッサでは，メインプロセッサのシステム状態確認処理を待つ必要がある．

(3) **ConditionSync**

前状態，処理，後状態の終了時に複数プロセッサに実行状態の処理単位が存在する場合のコンディション間同期である．各コンディション終了時における全プロセッサの実行タイミングを同期することで，実行状態の処理単位がコンディションを超えて先に処理を進めないようにする．

(4) **LoopTerminated**

最後の後状態において，実行状態のタスクを終了させないように待たせておくコンディション内同期である．各プロセッサの共通終了処理が終わったあと，メインプロセッサの

© 2011 Information Processing Society of Japan

Vol.2011-SLDM-149 No.11
Vol.2011-EMB-20 No.11
2011/3/18

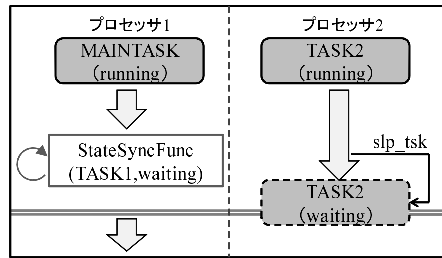

図 6 StateSync
Fig. 6 StateSync

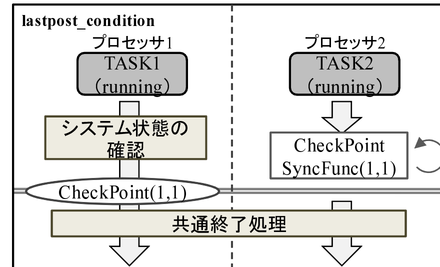

図 7 LastCheckedSync
Fig. 7 LastCheckedSync

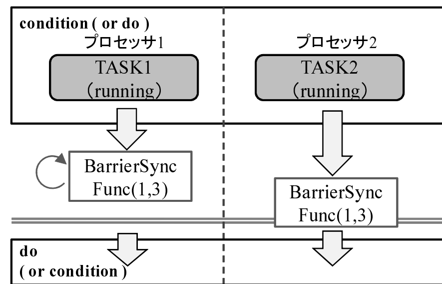

図 8 ConditionSync
Fig. 8 ConditionSync

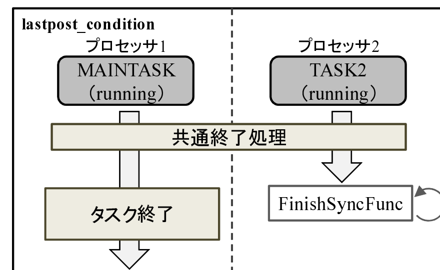

図 9 LoopTerminated
Fig. 9 LoopTerminated

み，全プロセッサのタスク終了処理を行う．タスクの終了処理には ter_tsk を発行することで行うが，この API は，対象タスクが休止状態である場合にはエラーとなってしまう．プログラムに記述されている全ての処理を終えたタスクは休止状態に遷移してしまうため，メインプロセッサからタスク終了処理を行うまで実行状態を維持している必要がある．

3.4 同期メカニズム

本節では，4 つの同期メカニズムについて説明する．

(1) StateSyncFunc

StateSyncFunc は，特定のタスクが特定の状態になるまで待つ同期方法である．CheckPointSyncFunc とは異なり，対象のタスクが処理後に実行状態以外となる場合にも利用できる．制約として，目的の処理後に同期対象のタスクが状態遷移を伴わない場合，すなわち実行状態のままである場合には利用できない．また，カーネルの API では割込みハンドラなど，タスク以外の実行状態を参照できないので，タスクにのみ利用可能である．実行状態以外の状態から実行状態に遷移する場合にも StateSyncFunc を利用することは可能であるが，その場合には CheckPointSyncFunc を用いることとした．

図 4 では，TASK2 が slp_tsk によって待ち状態に遷移することを，管理タスクが待つ (L03) ことによって，StateSync を実現している．**図 7** に同期のイメージを示す．

(2) CheckPointSyncFunc

CheckPointSyncFunc の名前が示す通り，チェックポイントを利用して，特定プロセッサの特定番号のチェックポイントの通過を待つことによって同期処理を実現する．制約として，同期対象の処理単位がチェックポイントを通過しなければならないので，対象が目的の処理後に実行状態である必要がある．

図 4 では，TASK1 のシステム状態確認処理直後に配置したプロセッサ 1 の 1 番目のチェックポイント (L26) を，TASK2 が待つ (L34) ことによって，LastCheckedSync を実現している．今回のケースではサブプロセッサに個別の終了処理がないので，実際には LastCheckedSync を用いなくても構わない．**図 6** に同期のイメージを示す．

(3) BarrierSyncFunc

BarrierSyncFunc は，2 つ以上の指定したプロセッサ間での実行タイミングを合わせる同期方法である．それぞれの指定プロセッサ上にある実行状態の処理単位が，指定した処理を完了するまで互いに待機する．複数プロセッサに実行状態の処理単位が存在する場合に利用できる．同期対象の処理単位が互いに CheckPointSyncFunc を利用することでも BarrierSyncFunc と同様の処理を行うことが可能だが，ConditionSync に限って用いることとする．ConditionSync は,「同期する処理単位」と「同期される処理単位」という 1:1 の関係ではなく，3 つ以上の処理単位が等しく「同期する処理単位」であるという，他の同期パターンとは異なる性質を持つためである．CheckPointSyncFunc は 1:1 の関係に対する同期処理であるので，同期するプロセッサ数が増えるとソースコードが複雑化する恐れがある．

図 4 では，do と post_condition の間で TASK1 と TASK2 が実行タイミングを合わせる (L22,L33) ことで，ConditionSync を実現している．**図 8** に同期のイメージを示す．

(4) FinishSyncFunc

FinishSyncFunc は，ループによって実行状態を維持する同期方法である．FinishSyncFunc によって待ち続けているタスクは，同期処理の終了によって処理を再開することがあり得ない．CheckPointSyncFunc を用いて，存在しない番号のチェックポイントを待つ

ⓒ 2011 Information Processing Society of Japan

Vol.2011-SLDM-149 No.11
Vol.2011-EMB-20 No.11
2011/3/18

ことでもこの同期パターンを実現可能であるが，本来の用途から外れるため，実装アルゴリズムが変わった場合に副作用が生じる恐れがある．また，同期失敗時のエラーメッセージが適切でなくなり，利便性を損なう．これらの理由のために LoopTerminated 用の同期メカニズムとして用意した．

図 4 では，後状態で実行状態である TASK2 が管理タスクの終了処理を待つ (L35) ことで，LoopTerminated を実現している．**図 9** に同期のイメージを示す．

4. 評　価

FMP カーネルの実行には ARM7 ベースのシミュレータである SkyEye[7] を利用した．FMP カーネル用の全テストシナリオ 2,542 件を 1 つのテストプログラムとして生成し，テストプログラムが通過することを確認した．同期処理が十分でなければエラーが発生すること，テストシナリオの正当性を保証するために取得したカバレッジを分析し，想定したパスを通ることを確認していることから，テストプログラムがテストシナリオ通りに動作したことを確認した．

同期パターン別の出現回数を**表 2** に示す．表から，同期処理は平均して 1 つのテストシナリオに 7,8 回登場する．テストの開発者が同期処理を手作業によって実装した場合，適切な同期パターンに相当する同期処理を記述する必要があるとともに，テストシナリオの修正に伴う同期処理の修正が必要となるので，非効率である．

SkyEye では，各プロセッサの実行タイミングを意図的にばらつかせることが可能であり，極端にばらつくような設定をした状態で数十時間のヒートラン (長時間動作試験) を行ってもプログラムがフェイルしないことを確認した．また，4 コアの ARM11MPCore を搭載した NaviEngine[10] を用いて，実機においても生成したテストプログラムが動作することを確認した．異なる 2 つのターゲットで動作したことによって，自動生成された同期処理に問題がないことと，同期メカニズムに汎用性があることを確認した．

表 2　同期パターン別の出現回数
Table 2　Number of appearance of the synchronization pattern

同期パターン	出現回数	同期パターン	出現回数
ActiveSync	1,397	DoStartSync	139
StateSync	2,198	DoFinishSync	187
LockSync	610	ExecSequenceSync	738
TimeSync	232	LastCheckedSync	4,041
ConditionSync	3,673	LoopTerminated	6,269
			合計 19,484

5. お わ り に

本研究では，FMP カーネルに対する API テストにおいて，必要となるプロセッサ間同期を設計した．同期の状況を同期パターンとして 10 種類にまとめ，同期の方法を同期メカニズムとして 4 種類にまとめた．そして，TTG に適切な同期処理を出力する機能を実装した．結果として，必要であると判断された全てのテストシナリオに対応することができた．同期設計は FMP カーネルの仕様に依存していないので，他のマルチプロセッサ対応 RTOS に対しても参考にできる．今後，新たに別の RTOS に対応する際に他の同期処理が必要になった場合は，必要に応じて同期パターンおよび同期メカニズムの追加を検討する．

謝辞　本研究にあたり，多くのご協力を頂いたコンソーシアム型研究の皆様に謹んで感謝の意を表する．

参 考 文 献

1) 名古屋大学大学院情報科学研究科附属組込みシステム研究センター, http://www.nces.is.nagoya-u.ac.jp/
2) TOPPERS プロジェクト, http://www.toppers.jp/
3) 鴨原一人, 眞弓友宏, 本田晋也, 高田広章, “マルチプロセッサ対応 RTOS を対象としたテストシナリオ記述法とテストプログラム生成ツール”, 情報処理学会組込みシステム (EMB), Vol.2010-EMB-18 No.1, pp.1–8, 2010.
4) 鴨原一人，松浦光洋，眞弓友宏，本田晋也，山本雅基，高田広章ほか，“組込みリアルタイム OS に対する API テストの実施”，ソフトウェアテストシンポジウム 2010 予稿集，pp.46–53，2010.
5) OJL による最先端技術適応能力を持つ IT 人材育成拠点の形成, http://www.ocean.is.nagoya-u.ac.jp/
6) 小林隆志, 沢田篤史, 山本晋一郎, 野呂昌満, 阿草清滋: On the Job Learning: 産学連携による新しいソフトウェア工学教育手法, 電子情報通信学会信学技報 SS2009-28, (Vol.109, No.170, pp.95-100), 2009.
7) TOPPERS 新世代カーネル統合仕様書, http://www.toppers.jp/docs/tech/ngki_spec-120.pdf
8) YAML, http://www.yaml.org/
9) 安積卓也, 古川貴士, 相庭裕史, 柴田誠也, 本田晋也, 冨山宏之, 高田広章, “オープンソース組込みシステム向けシミュレータのマルチプロセッサ拡張”, コンピュータソフトウェア, Vol.27, No.4, pp.24-42, Nov. 2010.
10) NaviEngine, http://www2.renesas.com/automotive/ja/assp/naviengine.html

© 2011 Information Processing Society of Japan