社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－SE－152 (3)
2006／5／17

UML設計モデル検査技術のための検証パターンの提案

金井勇人† 岸 知二†

本稿ではモデル検査技術を用いたUML設計検証のための検証パターンの提案をする．UML設計検証においては，仕様記述と論理的な記述が必要だが，論理的記述は仕様記述に依存して記述される．また，ソフトウェアの構造によって確認したい典型的な性質があり、またその確認方法にはいくつかの定石がある．そこで本稿では，ソフトウェア構造と性質とを合わせて体系づけたパターンを提案する．また，提案されたパターンを用いて設計検証を行った事例をあわせて紹介する．

Verification pattern for UML design utilizing model checking techniques

HAYATO KANAI† and TOMOJI KISHI†

In this paper, we propose verification pattern for UML design verification utilizing model checking techniques. In verifying UML model, we have to develop target model and define properties depending on the target model. Furthermore, typical software structures have their own listing of important properties. Hence, it is useful to define verification patterns as a set of software structures, their important properties, and verification techniques. We introduce verification pattern based on the idea, and demonstrate its usefulness based on a case study.

1. はじめに

近年，組み込みソフトウェアの開発・検証手法は従来のものでは十分に対応しきれなくなっている．その背景として，色々なところで組み込みソフトウェアが使用されるようになり，その信頼性が社会的な問題となっていることが挙げられる．また組み込みソフトウェアは大規模，かつ複雑になってきており，従来の開発手法の限界が指摘されている．よって，経験測による手法だけでなく，科学的手法の導入が検討されている．この試みは高信頼性組み込み用オブジェクト指向設計技術プロジェクト[4]においても，研究が行われている．

形式的検証手法の1つにモデル検査技術[3]がある．この検証技術は，ソフトウェアの有限状態モデルが，論理で表現された性質を満たすかどうかを状態の網羅的な探索によって検証を行う技術のことである．モデル検査技術をUML等で記述される設計モデルに適用することにより，ソフトウェアの信頼性を高めることが期待できる[9]．しかし，UML等で記述されたソフトウェアのモデルをモデル検査ツールで検証するためには，そのツールに依存した言語(以下，仕様記述言語と呼ぶ)で記述しなければならない．また，モデル検査技術ではソフトウェアが満たしてほしい性質を確認するために，性質を時間的な概念を持たせた論理式(以下，時相論理式と呼ぶ)で記述することも必要になり、ソフトウェア技術者にとって大変な作業になる。検証を支援する手法として，設計モデルを分離，抽象化することで設計モデル上で検査をしやすくする手法[5]などがある．本研究では，設計モデルから仕様記述言語，時相論理式への変換手法を問題とする．

ソフトウェアのそれぞれの構造によって確認したい典型的な性質があり、またその確認方法にはいくつかの定石がある。したがって、我々はそれを体系だって提示することが有効であると考える。そのためには，仕様記述言語の記述と検証したい性質を記述する時相論理式の両方を支援することが必要である。そこで，本研究ではソフトウェアの設計モデルに対しての形式的検証を行う際の形式的記述を支援することを目的とする．具体的には，ソフトウェア検証の定石をソフトウェア技術者が利用しやすい形でパターンとして提示する．

2. モデル検査技術によるUML検証

本研究はモデル検査ツールとしてSPIN[1)2)]を対象とする．以下，SPINにおけるモデルと性質の記述について説明する．

† 北陸先端科学技術大学院大学
Japan Advanced Inttitute of Science and Technology

2.1 モデルと性質の記述

2.1.1 仕様記述言語 PROMELA

SPINでは，対象システムを仕様記述言語PROMELAで記述する．プロセスはproctype宣言文を用いて定義する．複数のプロセスを記述することが可能である．以下に局所型変数宣言と1つの代入文から成り立つプロセスExampleの例を示す．

```
proctype Example(){
    int n;
    n=1;
}
```

2.1.2 時相論理式 LTL

時相論理とは，通常の論理式の構成要素である原始命題，論理積，論理和，論理否定の組み合わせに時間的な概念を持った時相演算子を加えたものであり，時間に関する概念を記述するのに適している．LTLは時相論理式の1つである．以下に例「pが真ならばいつかqは必ず真である」を意味するLTLを示す．

□(p -> <>q)

2.1.3 一般的なUML検証手順

以下に一般的なUML検証手順を示す．

(1) モデル化
UML設計モデルをモデル検査技術で使用できる形にモデル化する[7)8)11)]．このモデルをモデル検査ツールSPINが使用できる仕様記述言語PROMELAに変換して検査を行う．本研究ではUML設計モデルとしてクラス図と状態遷移図を用いる．

(2) 性質の記述
システムが満たすべき性質、もしくは満たしてはならない性質を時相論理式LTLを用いて記述する．このLTLはPROMELAで記述されたシステムの性質を記述するものであるから、その記述はPROMEL Aの記述に依存する．

(3) 検証
記述した性質が成立するか、もしくは成立しないかをモデル検査ツールを用いて検証する．

上記のようにこうした作業は仕様記述言語，時相論理式の文法をよく理解する必要がある．その上UMLで記述した対象システム，性質をどのように仕様記述言語，時相論理式に変換すれば良いかを考える必要があり，ソフトウェア技術者にとって，大変な作業となる．特に，検証したい性質が複雑になると，時相論理式も複雑な記述が必要になり，毎回アドホックに記述することは困難である．

3. 構造付き検証パターンの提案

3.1 従来の代表的な検証パターン

こうした検証を支援するひとつの方法として，ソフトウェア設計にデザインパターン[10)]があるように，ソフトウェア検証にもパターンを提示することが考えられる．現在までに提唱されている代表的な検証パターンとして、Dwyerの検証パターン[6)]がある．この検証パターンはよく使われる時相論理式の記述を性質ごとに分類しパターン化している．分類を図1に示す．

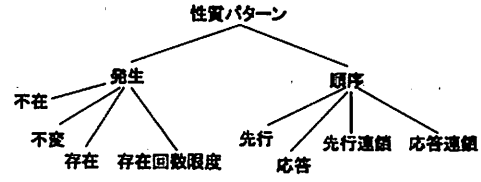


図1 Dwyerが提唱している検証パターン分類

この検証パターンではSPINで利用するLTL式以外のCTL等の時相論理式に対しても，パターン化しているが，ここではLTL式のみを示す．

- 不在...Pはずっと偽である．
  [](!P)
- 不変...Pはずっと真である．
  [](P)
- 存在...Pはいつか真になる．
  <>(P)
- 存在回数限度...Pは高々n回真になる．(以下の例はn=2時の式)
  (!P W (P W (!P W (P W []!P))))
- 先行...Pが真になる前にSが真になる．
  !P W S
- 応答...Pが真ならばSがいつか真になる．
  [](P -><>S)
- 先行連鎖
  - Pが真になる前に，Sが真になり，次にいつかTが真になる．
    <>P ->(!P U (S & !P & X(!P U T)))
  - Sが真になり，次にいつかTが真になる前に，Pが真になる．
    (<>(S & X<>T)) ->((!S) U P)
- 応答連鎖
  - Pが真になり，次にいつかTが真になるならば，Pはいつか真になる．

[](S & X<>T ->X(<>(T & <>P)))

– P が真にならば，いつか T が真になり，次にいつか P は真になる．

[](P -><>(S & X<>T))

3.2 従来の検証パターン特徴と問題点

従来の代表的な検証パターンは対象システムを特定していないので，汎用的に利用できるという特徴を持つ．問題点として，汎用的なので特定のソフトウェア構造上の具体的な性質の表現にどう使うかが明確に示されていない点が挙げられる．また，この検証パターンは LTL のみのパターン化なので，対象システムの記述まで考慮していない．

3.3 本検証パターンの特徴と利点

本検証パターンの特徴はＵＭＬの特定の構造に対して，その構造において重要な性質をリストアップし，そのそれぞれの性質を検証するための LTL のパターン化を行っている点である．このように構造とＬＴＬをパターン化することにより，実際のソフトウェア構造に関わる具体的な性質を検証するためのＬＴＬのパターンを提示することができ，ソフトウェア技術者にとって利用しやすいパターンとなることが期待される．まず，本検証パターンの構成は以下のものである．

- UML から PROMELA への変換規則
- 検証パターンカタログ
  - (1) 設計モデルに多出する構造を抽象化した構造
  - (2) 上記の構造でよく検査される性質を LTL で記述したもの

(1) と (2) をセットとしてまとめたものを検証パターンカタログとしてまとめた．検証パターンカタログについては後述する．本検証パターンを使った典型的な検証手順は以下のようになる（図２）．検証手順は以下のようになる．

(1) 設計モデルを作成する．

(2) 検証パターンカタログより対応する構造を見つける．

(3) 対応した構造でパターン化されている性質で対応する性質を見つける．

(4) 設計モデルを検証パターンに対応した変換規則で PROMELA へ変換する．

(5) 検証パターンカタログの LTL を設計モデルに依存した形で特殊化し，検証を行う．

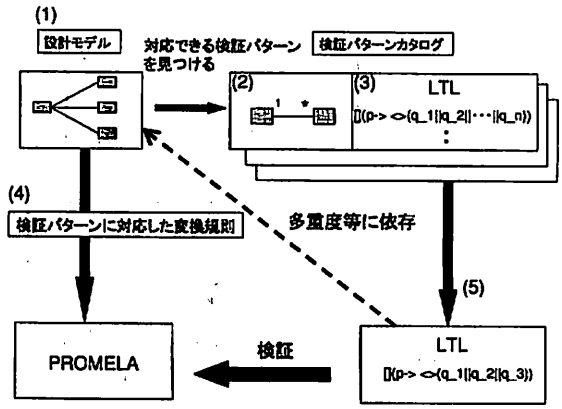


図 2 本検証パターンを使った検証方法

4. UML から PROMELA への変換規則

本研究での変換規則とは，後述する検証パターンカタログに対応した形で，UML から PROMELA への手続き的な変換規則を定義したものである．

まず，クラス図の変換規則，ステートチャート図の変換規則を示す．

4.1 クラス図の変換規則

クラス単体の変換規則

図 3 にクラス単体の変換例を示す．メンバ属性は LTL 式で検証できるように，グローバル変数に変換する．クラスは 1 つの proctype に変換する．操作，定数の返り値は mtype に変換する．

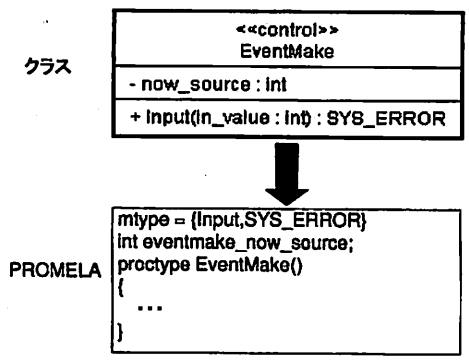


図 3 クラスから PROMELA への変換

クラス関連の変換規則

図 4 にクラス関連の変換例を示す．返り値のあるメソッドは呼び出し用と返り値用の２つのチャネルに変換する．これは呼び出しと返り値とでは，チャネルの

型が変わるので，そのための配慮である．

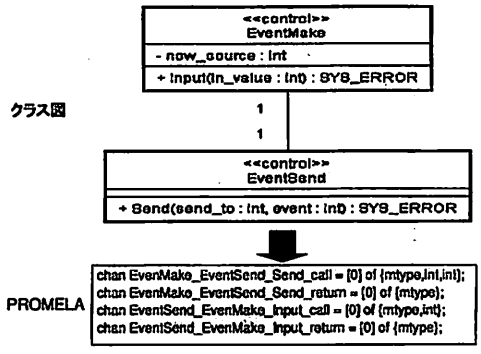


図 4 クラス図から PROMELA への変換

4.2 ステートチャート図の変換

図 5 にステートチャート図の変換例を示す．状態はラベルとグローバル変数に変換する．1 つの状態に対し，ラベル「状態名:」，ラベル「状態名_ENTRY」，ラベル「状態名_EXIT」と，bool 型グローバル変数「クラス名_状態名」に変換する．

以下に，ラベル，変数の意味を示す．

- 状態ラベル，変数の意味
  - 「状態名_ENTRY：」... ある状態に入る
  - 「クラス名_状態名」... ある状態にいるか否か
  - 「状態名_EXIT:」... ある状態から出る

  状態変数はその状態に入ると true になり，出ると false になる．遷移ではどこの状態にもないので，そのクラスの状態変数は全て false になる．上記のような意味を持たせることにより，その状態に入ったのか，いるのか否か，出たのかを検査することができる．

  例えば，ある状態に入ったかを検査したい場合，ラベル「状態名_ENTRY：」に到達したかを検査することで，ある状態に入ったかを検査できる．

  メッセージ送受信はラベルとチャネルを使った送受信文に変換する．メッセージ送信はラベル「SENT_メッセージ名:」とメッセージ送信文「チャネル!メッセージ」に変換する．メッセージ受信はラベル「RECEIVED_メッセージ名:」とメッセージ受信文「チャネル?メッセージ」に変換する．

  以下に，ラベルの意味を示す．
- メッセージ送受信のラベルの意味
  - 「SENT_メッセージ名」... メッセージを送信した
  - 「RECEIVED_メッセージ名」... メッセージを受信した (例 RECEIVED_NOTICEEVENT:)

上記のような意味を持たせることにより，あるメッセージを送信したか，受信したかを検査できる．

例えば，あるメッセージを受信したか検査したい場合，ラベル「SENT_メッセージ名」に到達したかを検査することで，あるメッセージを受信したかを検査できる．

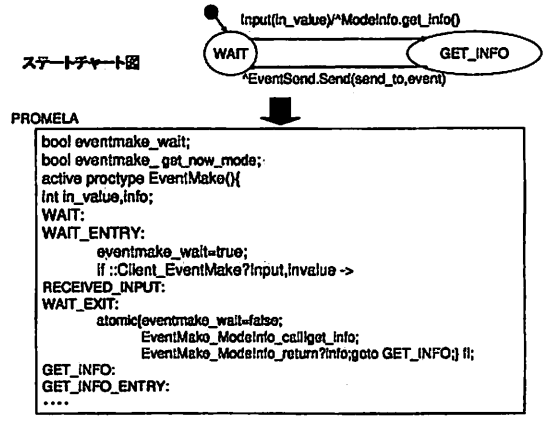


図 5 ステートチャート図から PROMELA への変換

5. 検証パターンカタログ

5.1 検証パターンの記述項目

以下に検証パターンの記述項目を示す．

- 名前
  その検証パターンが対象とするソフトウェアの構造が簡潔に連想できる名前を示す。
- 分類検証対象の性質の分類を示す．現在のカタログでは，メッセージ送受信、状態に分類している。
- 構造概要
  この構造は何をするのか、その原理と意図は何か等を示す。
- 構造
  UML のクラス図で示す。
- 構成要素
  構造に使われているクラスの責任分担を示す。
- 協調関係
  それぞれの構成要素が責任分担を遂行するためにどのように協調するかを示す。ステートチャート図も記述する。
- 検査項目と LTL
  本検証パターンで定義している検証項目とそれを LTL で変換したものを示す。

- 例題
  簡単な例題を使って検証パターンの使用方法の例を示す。
- 注意事項
  間違い易い点や意味が理解しにくい点等の補足を示す。

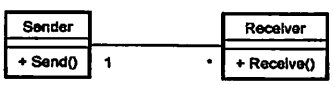
5.2 検証パターンの一覧

今回主に，メッセージの送受信に関する構造と，状態に関する構造に注目して，構造を決定した．以下の構造に対して検証パターンをまとめた．

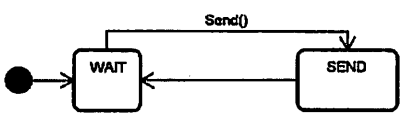
- 1－1メッセージ送受信構造
- 1－Nメッセージ送受信構造
- メッセージ送受信連鎖構造
- クラスの状態管理構造
- リソースの取り合い構造

一例として、1-Nメッセージ送受信構造を示す。

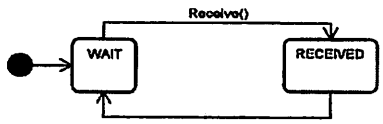
- 名前
  1－Nメッセージ送受信構造
- 分類
  メッセージ送受信
- 構造概要
  1つのメッセージ送信用クラスが複数のクラスにメッセージを送る構造。
- 構造



- 構成要素
  - Senderクラス
    受信したメッセージに対して、適したメッセージを、適した複数のクラスに送信する．
  - Receiverクラス
    メッセージを受信する．
- 協調関係
  Senderクラスが Sendメッセージを受信したら、Rceiverクラスに Receiveメッセージを送信する。
  - Senderクラス



  - Receiverクラス



- 検査項目と LTL
  - (単純応答)
    送信クラスがメッセージを送信したら、受信クラスの1つ以上のインスタンスが受信する。
    LTL
    以下に LTL を示す．

```
#define send Sender@RECEIVED_Send
#define receive
  (Receiver[pid_1]@
          RECEIVED_Receive
 ||Receiver[pid_2]@
          RECEIVED_Receive
 || ··· ||Receiver[pid_n]@
          RECEIVED_Receive)

[](send -> <>receive)
```

    LTLと要素の対応関係
    * send…Senderクラスのインスタンスが Sendメッセージを受信した。
    * receive…Receiverクラスのインスタンスが receiveメッセージを受信した。
  - (特定応答)
    送信クラスがメッセージを送信したら、受信クラスの特定のインスタンスを含む1つ以上が受信する。
    LTL
    以下に LTL を示す．

```
#define send Sender@RECEIVED_Send
#define receive Receiver[pid_n]@
                RECEIVED_Receive

[](send -> <>receive)
```

    LTLと要素の対応関係
    * send…Senderクラスのインスタンスが Sendメッセージを受信した。
    * receive…Receiverクラスのインスタンスが receiveメッセージを受信した。

– (網羅)
送信クラスがメッセージを送信したら、全ての受信クラスのインスタンスが受信する。

LTL

以下に LTL を示す.

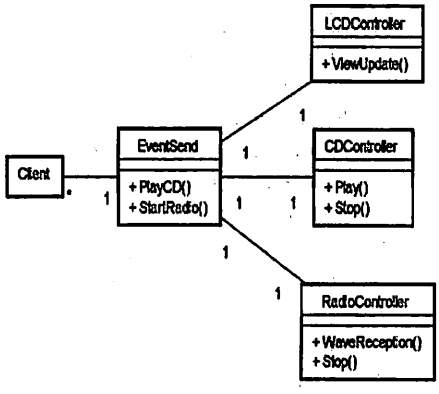
```
#define send Sender@RECEIVED_Send
#define receive_1
  Receiver[pid_1]@RECEIVED_Receive
#define receive_2
  Receiver[pid_2]@RECEIVED_Receive

#define receive_n
  Receiver[pid_n]@RECEIVED_Receive

[](send -> <>receive_1 &&
<>receive_2 && ··· && <>receive_n)
```

* send…Sender クラスのインスタンスが Send メッセージを受信した。
* receive…Receiver クラスのインスタンスが receive メッセージを受信した。

- 例題例として以下のある CD ステレオの一部の構造を考える。



LTL

– (単純応答)
EventSend クラスが PlayCD メッセージを受信したら、RadioController クラスが Stop メッセージを受信するか、CDController クラスが Play メッセージを受信するか、LCDController クラスが ViewUpdate メッセージを受信する。

```
#define send EventSend@
                RECEIVED_PlayCD
#define receive
  (RadioController@RECEIVED_Stop
  ||CDController@RECEIVED_Play
  ||LCDController@
          RECEIVED_ViewUpdate)

[](p -> <>q)
```

- (特定応答)
EventSend クラスが PlayCD メッセージを受信したら、少なくとも CDController クラスが Play メッセージを受信する.

LTL

以下に LTL を示す.

```
#define send EventSend@
                RECEIVED_PlayCD
#define receive CDController@
                RECEIVED_Play

[](send -> <>receive)
```

- (網羅)
EventSend クラスが StartRadio メッセージを受信を行ったら、LCDController クラスが View メッセージを，CDController クラスが Stop メッセージを，RadioController クラスが WaveReception メッセージをそれぞれ受信する.

LTL

以下に LTL を示す.

```
#define send
  EvedSend@RECEIVED_StartRadio
#define receive_1
  LCDController@RECEIVED_View
#define receive_2
  CDController@RECEIVED_Stop
#define receive_3
  RadioiController@
        RECEIVED_WaveReception

[](send -> <>receive_1 &&
    <>receive_2 &&  <>receive_3)
```

- 注意事項
  – メッセージの受信前述の「UML から PROMELA へのマッピング方法」でも記述しているよう

にメッセージの受信は以下のように記述する。

```
...
channel?MSG ->
RECEIVED_MSG: operational sentence;
...
```

この記述は厳密には、メッセージの受信により真になるのではなく、"operational sentence;"が実行されて真になる。本検証パターンでは、メッセージが受信される順番とメッセージが受信され最初の命令文が実行されることは等価であると考え、上記のような表記をしている。純粋に受信のみを検査したい場合は適当ではない。

以下のように記述すると、メッセージを受信する前にラベルが真になってしまうので、適当ではない。

```
...
RECEIVED_MSG: channel?MSG ->
operational sentence;
...
```

6. 事例への適用

企業から提供されたカーオーディオ・システムの設計モデルへの適用を行った．具体的には，このカーオーディオ・システムの重要な構造に対して重要な性質を選定し，それを本検証パターンを適用し検証した．

6.1 対象モデル

外からユーザーによって入力される情報に対してどの制御にどのメッセージを送ればいいかを，制御している構造に対して検証をおこなった．この検証用に抽象化した設計モデルを図 6 に示す．

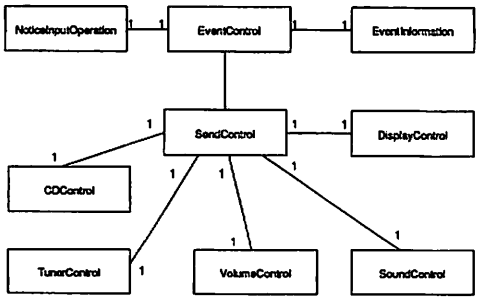


図 6 イベント生成構造

構成要素

- NoticeInputOperation クラス
- EventControl クラス
- EventInformation クラス
- SendControl クラス
- DisplayControl クラス，CDControl クラス，TunerControl クラス，VolumeControl クラス，SoundControl クラス

6.2 イベント生成構造におけるメッセージの網羅性

SendControl クラスがメッセージ送る各制御クラス全てに，メッセージが遅れているか検証する．検証項目は以下のようになる．

検証項目

SendControl クラスが NoticeEvent メッセージを受信したら，DisplayControl クラス，CDControl クラス，TunerControl クラス，VolumeControl クラス，SoundControl クラス，全てがいつか NoticeEvent メッセージを受信する．

適用する検証パターン

検証パターンカタログから「1-N メッセージ送受信構造」を適用する．検証項目として，「網羅」を適用する．

設計モデルと検証パターンのクラスの対応を以下に示す．「設計モデルクラス → 検証パターンクラス」のように表記する．

- SendControl クラス → Sender クラス
- Receiver クラス → CDControl クラス，TunerControl クラス，VolumeControl クラス，SoundControl クラス，DisplayControl クラス

検証パターンの LTL

まず，検証パターンの LTL を以下に示す．

```
#define send
    EvedSend@RECEIVED_StartRadio
#define receive_1
    LCDController@RECEIVED_View
#define receive_2
    CDController@RECEIVED_Stop
#define receive_3
    RadioiController@
        RECEIVED_WaveReception

□(send -> <>receive_1 &&
    <>receive_2 && <>receive_3)
```

設計モデルを検証する LTL

次に，上記の検証パターンの LTL を利用して，記述

した事例の設計モデルを検証する LTL を以下に示す.

```
#define send
        EvedSend@RECEIVED_StartRadio
#define receive_1
        CDControl@RECEIVED_NoticeEvent
#define receive_2
        TunerControl@RECEIVED_NoticeEvent
#define receive_3
        VolumeControl@RECEIVED_NoticeEvent
#define receive_4
        SoundControl@RECEIVED_NoticeEvent
#define receive_5
        DisplayControl@RECEIVED_NoticeEvent

[](send -> <>receive_1 && <>receive_2 &&
<>receive_3 && <>receive_4 && <>receive_5)
```

6.3 考　　察

以上のように、検証パターンを用いることにより、手続きだって検証モデルや LTL を作成することができた。

特に、本検証パターンはソフトウェア構造のパターンに基づいて体系化しているため、検証対象となる UML モデルの構造に照らし合わせて適用することができ、ソフトウェア技術者ににとって利用しやすいものだと考えている。

7. ま と め

本研究では，2つの成果が得たれた．1つ目は，本研究の目的である，モデル検査技術で検査を行う際の記述の支援である．本検証パターンを利用することによって、PROMELA 記述や LTL 記述を，パターンに基づいてより容易に記述することができると期待される．2 つ目は，新たな検証パターンの提案である．従来の LTL だけの検証パターンは抽象度が高かったが，本検証パターンは具体的なソフトウェア構造上の性質に基づいて検証パターンを定義しているため，ソフトウェア技術者にとってはより利用しやすい．なお，今回作成した 5 つのソフトウェア構造以外にも，この枠組みに基づいて検証パターンを整理することができると考えている．

今回は，今までに実際に行った検証事例で多出した5つのパターンを整理した．今後，今回のパターン化手法を用いてもっと多くの構造，性質をパターン化し，さらに検証パターンカタログを充実させていくとともに，さらに適用事例を増やして，その有効性について確認をしていきたい．

参 考 文 献

1) http://spinroot.com/spin
2) G.J,Holzmann. : The SPIN Model Checker. Addison-Wsley 2004.
3) E.Clarke,O.Grumberg,and D.Peled : Model Checking, MIT 1999.
4) 岸知二，青木利晃，中島震，野田夏子，片山卓也：プロジェクト紹介：高信頼組込み用オブジェクト指向設計技術，情報処理学会ソフトウェア工学研究会，SE146-7, pp41-46, 2004.
5) 丸山陽太郎, 岸 知二, 片山卓也: 組込みソフトウェア設計へのモデル検査適用手法の提案と実験・評価, 情報処理学会 組込みソフトウェアシンポジウム, pp64-71, 2005.
6) Patterns in Property Specifications for Finite-state Verification, Matthew B. Dwyer, George S. Avrunin and James C. Corbett to appear in Proceedings of the 21st International Conference on Software Engineering, May, 1999
7) Timm Schafer, et. al: Model Checking UML State Machines and Collaborations,Workshop on Software Model Checking,2001.
8) Lilius,J. and Paltor,I. P.: vUML: a Tool for Verifying UML Models,TUCS Technical Report No.272, 1999.
9) Kishi, T., et. al.: Project Report: High Reliable Object-Oriented Embedded Software Design, The 2nd IEEE Workshop on Software Technology for Embeddedand Ubiquitous Computing Systems (WSTFEUS'04),2004.
10) E.Gamma,R.Helm,R.Johnson and J.Vlissides : Design Patterns Elements of Reusable Object-Oriented Software, ADDISON-WESLEY 1995.
11) Schafer T., Knapp A. and Merz, S., Model Checking UML State Machines and Collaborations, Electronic Notes in Theoretical Computer Science 55(3),2001.