ソフトウェア設計に対するモデル駆動型検証プロセス

長野伸一[†,††] 吉岡信和[††]
田原康之[††] 本位田真一[††,†††]

本研究では，新しいソフトウェアプロセスの 1 つとしてモデル駆動型検証プロセスを提案する．一般に，設計検証とは，実践的ノウハウや経験を必要とする，習得のハードルが非常に高い作業である．しかし，これまで検証プロセスは，それらのノウハウや経験に沿って体系化されていないため，経験の浅いソフトウェア開発者にとって，検証は依然として困難な作業となっている．提案プロセスは，この課題に対する解決を目的とするものである．本研究では，検証プロセスを，中間成果物である検証モデルに対する変換手続きととらえ，そのうえで検証に必要な具体的作業の順序と内容を体系的に定義する．提案プロセスは次の 2 つの特長を持つ．第 1 に，検証作業に有効な実用的ノウハウの利用方法を明確化している．第 2 に，検証モデル間の変換手続きの逆像が，設計誤り発見のためのデバッグプロセスの定義になっている．さらに，3 つのモデル検査ツール，SPIN，Cadence SMV，LTSA のそれぞれを利用した 3 つの検証プロセスのインスタンスを示し，各プロセスをネットワーク家電の制御ソフトウェア設計に適用した事例を示す．その適用結果に基づいて，提案プロセスの特長と有効性に対する評価について述べる．

Model Driven Verification Process for Software Design

Shinichi Nagano,[†,††] Nobukazu Yoshioka,[††] Yasuyuki Tahara[††]
and Shinichi Honiden[††,†††]

This paper proposes a model-driven verification process that is yet another software process. Generally, verification is a very difficult task that calls for practical tips and experiences. However, no verification processes have been established systematically along with the tips and experiences. As a result, it is still difficult for software developers without enough experiences to begin on the verification task. The proposed process is a solution to this issue. It is defined as transformation procedures of intermediate products yielded during verification, systematizing the order and the contents of the activities needed for verification. The process has the following two advantages. First, it makes clear how to use the practical tips of verification. Second, the inverse images of the model transformation procedures result in the definition of debugging process for detecting design faults. We also illustrate a case study of application of three verification process instances using three model checkers, SPIN, Cadence SMV, and LTSA, respectively, to digital appliance control software. With the results of the applications, we evaluate the advantages and the effects of our proposed process.

1. はじめに

本研究では，新しいソフトウェアプロセスの 1 つとしてモデル駆動型検証プロセスを提案する．一般に，ソフトウェアの設計検証は，ソフトウェア開発と同様に，様々な角度から設計問題をとらえるという実践的なスキルと経験を必要とする，習得のハードルが非常に高い活動である．具体的には 4 種類の活動：問題の分析，検証モデルの作成，モデル検査言語による記述，および検証実施と設計誤り発見，を行う．これらの活動は，ソフトウェア開発における 4 種類の活動：分析，設計，実装，試験，に正に対応付けることができる．また，開発プロセスの入力は曖昧さが排除されていない要求であるのに対して，検証プロセスの入力は，設計仕様，要求仕様，検査仕様である．すなわち，開発プロセスでは要求自体の分析が必要があるのと同様に，検証プロセスでも，設計仕様のどの部分を検証の対象とし，どの前提条件の下で検証を実施するかを明確にする分析活動が必要である．設計仕様の適切な部分が適切な前提条件の下で検証されなければ，その

† 株式会社東芝研究開発センター
Corporate R&D Center, Toshiba Corporation
†† 国立情報学研究所
National Institute of Informatics
††† 東京大学
The University of Tokyo

活動に膨大な時間をかけたとしても，検証結果は意味のないものになってしまう．つまり，この分析活動を，開発プロセスにおける要求分析活動よりも慎重に行う必要がある．なぜならば，開発プロセスの最終成果物は具体的な振舞いを実現するプログラムであるのに対して，検証プロセスの最終成果物は単なる “true” であり，その情報だけから設計仕様が本当に意図どおりに検証されたか否かの判断が一般には困難だからである．

近年，ネットワーク家電に代表されるような組み込み機器の高性能化，複雑化にともない，その制御ソフトウェアの高信頼化に関する設計検証が重要視されている．しかし，現在の検証活動は，熟練者の経験や勘に頼ってアドホックに行われている状況が依然として多く，高い技術と十分な経験とを備えたソフトウェア技術者の育成が大きな課題となっている．一方，ソフトウェアの開発規模は増加の一途をたどっており，このままでは熟練者の経験に依存した検証活動はいずれ破綻し，ソフトウェアクライシスの再来となりかねない．そこで，かつてソフトウェア開発プロセス技術がソフトウェアの大規模化，技術者不足に対する解決に貢献したように，本研究は検証活動をプロセスとしてとらえ，その作業内容を整理して体系化することにより，職人的活動から工学的活動への転換を図る．

本研究では，検証技術の中でも実用化が進んでいる，モデル検査を利用したソフトウェア設計検証について論じる．近年，モデル検査はソフトウェア設計およびソースコードを対象とした試験手法として非常に注目されており，産業界の実問題へ適用した数多くの事例が報告されている．また，実用レベルの高品質な検証ツールが開発され利用されている．特に，UML 図からモデル検査の入力モデルへ変換して検証を行う UML 検証手法に関する研究が多数報告されている[5),7),19)]．これらの手法は，完全に仕様記述された UML 図から，モデル検査の入力言語による記述へ，直接的にあるいは中間言語を介して変換を行うものである．たとえば，HUGO プロジェクト[19)] は，モデル検査ツール SPIN[13)] を利用しており，UML の状態遷移機械から SPIN の入力言語 Promela による記述へ，コラボレーション図を LTL 式へ変換する手法を提案している．これらの手法は，モデル検査の理論に馴染みがない開発者でも，設計検証することが可能となるために実用性は高い．しかし，その利用の際には以下に述べる 3 つの難しさが存在する．

- 開発者は，検証問題ごとに異なる前提条件やシステム環境を考慮した検証モデルを作成する必要がある．一般に，前提条件やシステム環境に関するすべての情報が，設計仕様の UML 図で表されるわけではない．
- 各変換手法は，必ずしも UML 仕様に完全準拠しているわけではなく，UML セマンティクスの解釈に差異がある．開発者は，各変換手法がサポートしている UML 記法やセマンティクスに準じたモデルを作成しなければならない．変換手法と設計仕様のセマンティクスが一致しない場合は変換手法は利用できないので，開発者自身がセマンティクスのモデル化を行う必要がある．
- 大規模システムの検証の際には，UML 図や検証モデルの抽象化を行う必要がある．モデル検査の入力言語は一般の開発者にとって馴染みがないので，変換後に行う抽象化は一般の開発者にとって敷居が高い．

これらの難しさは，それぞれ検証プロセスにおける分析，設計，実装における難しさを表している．したがって，実際の検証では，単純に UML 検証手法を利用するのではなく，検証をプロセスと見なした作業を行うことが必要不可欠である．

また，実際，検証作業や実践的ノウハウには，モデル検査ツールに固有のものと，モデル検査ツールに依存しない汎用的なものとが存在する．検証作業自体はツールに依存しない共通のものである一方，その検証作業の具体的内容はツールごとに独自の手法である場合が少なくない．たとえば，分散システムの設計検証の場合，一般に通信メカニズム（通信ポート，通信チャネルなど）の検証モデルを設計するが，そのモデリング方法はツールごとに異なり，ツール独自のノウハウが存在する．一般の開発者にとって，多数のモデル検査ツール，および整理されていない多種多様な実践的ノウハウの中から，検証問題に応じて適切なものを選択することは容易ではない．このことが，モデル検査技術導入の敷居の 1 つになっている．

本研究では，4 種類の活動：分析，設計，実装，および検証と設計誤り発見，から構成された，モデル駆動型検証プロセスを提案する．提案プロセスを，中間成果物である検証モデルの間の変換手続きとして定義し，そのうえで検証に必要な具体的作業の順序と内容を体系化する．提案プロセスは以下の 2 つの特長を持つ．第 1 に，検証プロセスの各活動において実用的ノウハウを顕在化し，その利用法を明確化している．第 2 に，検証モデル間の変換手続きの逆像が設計誤り発見のためのデバッグプロセスの定義になっている．また，理論的背景の異なる 3 つのモデル検査ツール，SPIN[13)]，Cadence SMV[15)]（以降，SMV と表す），

LTSA [14] のそれぞれを利用した 3 つの検証プロセスのインスタンスを示す．3 つのモデル検査ツールは，広く利用されている代表的なツールであり，いずれも時相論理の体系として線形時相論理 LTL を扱うことができる．複数のモデル検査ツールの検証プロセスを定義した結果，モデル検査ツールに依存しない汎用的な作業や利用方法と，モデル検査ツールに固有のものとが区別され明確になっている．最後に，各検証プロセスを，ネットワーク家電の制御ソフトウェアに対する設計検証に適用した事例を示す．その適用結果に基づいて，提案プロセスの特長と効果に対する評価について述べる．

本稿の構成は以下のとおりである．まず，2 章でモデル駆動型検証プロセスを提案する．次に，3 章で検証プロセスのインスタンスと適用事例を示す．4 章で，適用事例に基づいて提案プロセスの特長と有効性を評価する．最後に，まとめと今後の課題について述べる．

2. モデル駆動型検証プロセス

2.1 提案プロセスの概要

本研究では，ソフトウェア開発プロセスの設計工程において，開発者はすでに要求分析と対象システムの設計を完了していると仮定し，設計仕様が誤りを含まず要求仕様を満足していることの検証について議論する．

検証プロセスの入力は次の 3 つである．(1) UML で記述された設計モデル，(2) 対象システムが実現すべき振舞い要件を含む要求仕様，(3) 対象システムが満足すべき振舞い制約を含む検査仕様．一方，検証プロセスの出力は次の 2 つである．(1) 設計誤り，(2) 検証成否を含む検証結果と，検証エラー時の反例．

モデル駆動型検証プロセスは，4 種類の検証モデルと 6 つの実行ステップから構成される．4 つの検証モデルとは，**図 1** に示すように，設計モデル（Design Model），モデル検査独立モデル（Model Checker Independent Model），モデル検査依存モデル（Model Checker Specific Model），モデル検査記述（Model Checker Description）である．以降では，それぞれ DM，MCIM，MCSM，MCD と表す．各検証モデルは，設計情報や検証情報を表す複数の要素から構成される．たとえば，システムの振舞いを表す状態遷移や，検証すべき品質特性を表す形式的記述などが含まれる．DM は検証プロセスの入力 (1) であり，一般に検証プロセスを意識せずに作成される．残りの 3 つの検証モデル，MCIM，MCSM，MCD は検証プロセスの中間成果物である．一方，1 章では 4 つの検証ステップについて述べたが，以降では，4 番目の検証と設計誤り発見のステップを 3 つのステップに分割し，図 1 に示すように検証プロセスを 6 つのステップで構成する．検証モデルと検証ステップの詳細は 2.2 節で述べる．

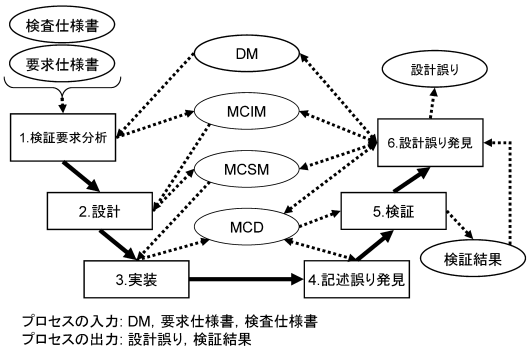

図 1 検証プロセスの概要
Fig. 1 Overview of the verification process.

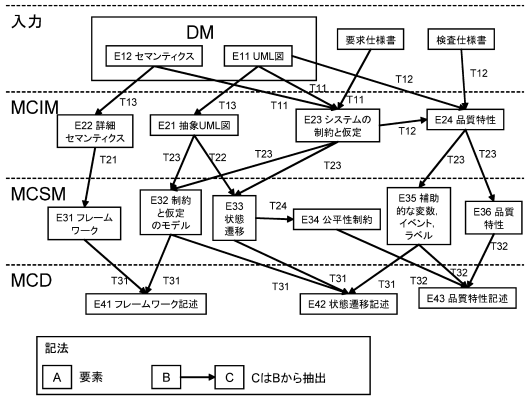

図 2 検証モデル間の依存関係
Fig. 2 Dependency between models.

本研究では，検証プロセスを検証モデル間の変換手続きと見なす．すなわち，検証プロセスの各ステップで参照される要素と新たに生成される要素との関係に強調をおいて，検証プロセスについて述べる．検証ステップと要素の関係を**図 2** に示す．矩形と矢印はそれぞれ，要素およびモデル間の依存関係を表す．図中の第 1 層は検証プロセスの入力を，第 2，3，4 層はそれぞれ，MCIM，MCSM，MCD の要素を表している．矢印にラベル付けされた各数値は，検証プロセスにおけるタスク番号を表している．たとえば，T13 は，DM の要素 E11 から，MCIM の要素 E21 を生成する．

2.2 検証プロセスの詳細

2.2.1 検証要求分析ステップ

検証要求分析ステップの入力は，DM，要求仕様，検査仕様の 3 つで，出力は MCIM である．

DM は 2 つの要素 E11，E12 から構成される．

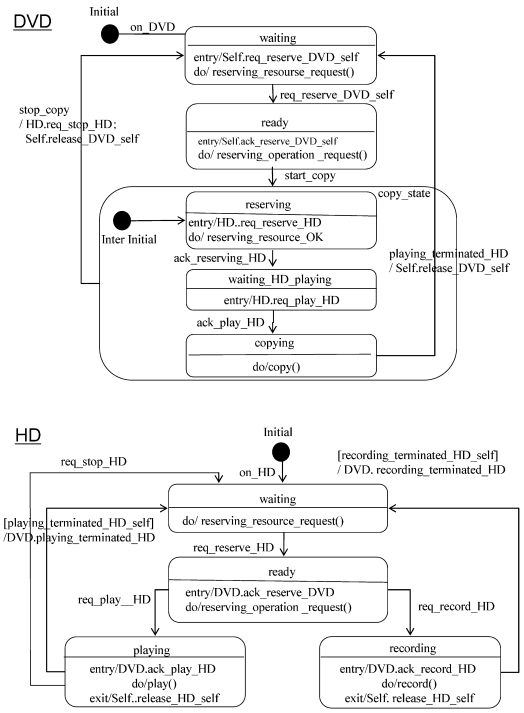

図 3 DM の要素 E11
Fig. 3 Element E11 of the DM.

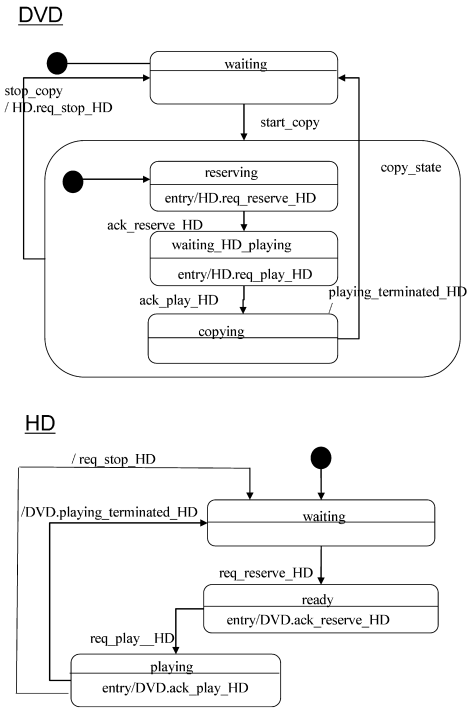

図 4 MCIM の要素 E21 の例
Fig. 4 Element E21 of the MCIM.

E11：対象システムの設計情報である UML 図．**E12**：UML 図のセマンティクス．UML 仕様ではセマンティクスの曖昧さが完全には排除されていないために，開発者は対象システムの環境に応じて，適切で曖昧さのないセマンティクスを定義する必要がある．**図 3** の 2 つの状態機械図は E11 の例を表しており，2 台のネットワーク家電機器，DVD レコーダと HD レコーダとが連携動作する振舞いを定義している．

MCIM は，特定のモデル検査手法のモデリング法や表記法を利用せず，システムの振舞いに関して検証すべき側面をモデル化したものである．MCIM は，4 つの要素から構成される：**E21**：抽象 UML 図，**E22**：形式的で詳細に定義された UML セマンティクス，**E23**：システムの振舞いや構造に関する制約，およびシステム環境に関する仮定，**E24**：LTL 式で記述された品質特性．MCIM の E21 の例を**図 4** に示す．

本ステップは 3 つのタスクから構成される．

T11 検証範囲の決定：まず，対象システムを検証する理由と目的を決定する．次に，対象システムのどの機能や性質を検証すべきかを決定する．UML 図のユースケースや検査仕様のテストケースを参照し，検証すべき品質特性の優先度付けを行う．一般に，多様なネットワーク家電機器の間で排他的なアクセス制御を行うなど，安全性や信頼性に関わるシステムの機能や性質が選択されることが多い．最後に，システムの振舞いに関する制約とシステム環境に関する仮定とを決定する．たとえば，ユーザの利用環境や使用するハードウェアの制約などが含まれる．

T12 品質特性の定義：検証すべき品質特性を LTL 式で定義する（E24）．一般に，品質特性は次の 2 つの観点から導くことができる．一方はシステム利用者の観点で，システムの外部から振舞いを観察する．DM におけるユースケース図や検査仕様を利用する．他方はシステム開発者の観点で，システム内部の振舞いやデータ構造を観察する．DM におけるシーケンス図やコミュニケーション図を利用する．

T13 抽象 UML 図：UML 図（E11）に対して，検証で関心のある部分をスライシングすることによって，UML 図の抽象化（E21）を行う．検証の関心とは関連性の低い変数，イベント，状態遷移を，状態機械図から除外する．また，UML セマンティクス（E22）を曖昧さのないように詳細に定義する．

2.2.2 設計ステップ

設計ステップの入力と出力はそれぞれ，MCIM と MCSM である．

MCSM は，特定のモデル検査手法が扱うことがで

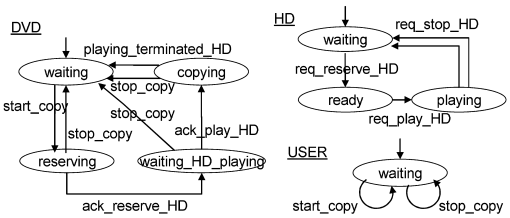

図 5 SMV 版 MCSM の要素 E33 の例
Fig. 5 Element E33 of the MCSM for SMV.

きる表記を利用して，システムの振舞いに関して検証すべき側面をモデル化したものである．SMV，SPIN，LTSA の表記法はそれぞれ，有限状態機械（FSM），抽象 FSM と通信チャネル，有限状態プロセス（FSP）とラベル遷移システム（LTS）である．MCSM は，5 つの要素から構成される：**E31**：MCIM のセマンティクスを記述するためのフレームワーク，**E32**：制約と仮定のモデル，**E33**：状態遷移モデル，**E34**：公平性制約，**E35**：検証用の補助的な変数，イベント，ラベル，**E36**：品質特性（E24 と同一）．一般に，E31〜E33 はモデル検査手法に特有の表記法で記述され，E34 と E35 はコメント形式，E36 は LTL 式で記述される．SMV による MCSM の記述の例を**図 5** に示す．

設計ステップは 4 つのタスクから構成される．

T21 フレームワークの作成：UML 図（E22）のセマンティクスを，モデル検査手法に特有の表記法を利用してフレームワークとして記述する．複数の状態遷移モデル間の実行スケジューリングや通信ポートなど，記述を共通化できるものは部品化する．

T22 MCIM の抽象化：モデル検査手法の表記法の記述力に応じて，抽象 UML 図（E21）をさらに抽象化する．たとえば，LTSA では状態機械の階層の表現が容易ではないので，状態機械の平坦化を行う．ただし，スーパステートとサブステートとの間でイベント受信の優先度が変更になるため，セマンティクスや，制約と仮定への影響を考慮する必要がある．

T23 MCIM から MCSM への変換：まず，適切な変換手続きを決定し，T22 で作成した抽象 UML 図から状態遷移モデル（E33）を作成する．基本的に，MCIM の各状態機械に対応して 1 つの状態遷移モデルを記述し，各イベントを 1 つの変数または値として表現する．次に，状態遷移モデル間の接続関係を記述する．また，システムの振舞いに関する制約とシステム環境に関する仮定（E23）も，モデル検査手法の表記法で記述する．さらに，開発者が検証結果を効率良く分析できるように，補助的な変数，イベント，ラベル（E35）を導入する．最後に，検証すべき品質特性（E36）を MCIM の E24 と同一の LTL 式である．

```
1.  module hd_mod(dvd){

2.    /* define state transtions */
3.    state : {waiting, ready, playing};
4.    init(state) := waiting;
5.    next(state) := case {
6.      t1 : ready;
7.      t2 : playing;
8.      t3 & (enabled = 3) : waiting;
9.      t4 & (enabled = 4) : waiting;
10.     default : state;
11.   };

12.   DEFINE t1 := ~delay & (state = waiting)  & req_reserve_HD;
13.   DEFINE t2 := ~delay & (state = ready)    & req_play_HD;
14.   DEFINE t3 := ~delay & (state = playing);
15.   DEFINE t4 := ~delay & (state = playing)  & req_stop_HD;
16.   DEFINE delay := set_ack_reserve_HD | set_ack_play_HD |
      set_playing_terminated_HD;

17.   /* define nondetermination between t3 and t4 */
18.   enabled : {0,3,5};
19.   init(enabled) := 0;
20.   next(enabled) := case {
21.     next(t3) & next(t4) : {3,4};
22.     next(t3) : 3;
23.     next(t4) : 4;
24.     default  : 0;
25.   };

26.   /* System properties */
27.   reachability : assert F(hd.state=waiting);
28.   liveness     : assert G(hd.state=waiting -> F(hd.state=ready));
29.   liveness     : assert G(hd.state=ready   -> F(hd.state=waiting));
30.
31. }
```

図 6 SMV 版 MCD の例
Fig. 6 Example of a MCD for SMV.

T24 公平性制約の導入：状態遷移モデル間や状態遷移間に非決定的な実行が存在すれば，公平性制約（E34）の導入を検討する．

2.2.3 実装ステップ

実装ステップの入力は MCSM，出力は MCD である．

MCD は，特定のモデル検査ツールの入力言語を用いて MCSM の要素を記述したものである．MCD は，3 つの要素から構成される：**E41**：フレームワークの記述，**E42**：状態遷移の記述，**E43**：品質特性の記述．SMV の入力言語で記述した MCD の例を**図 6** に示す．

実装ステップは 2 つのタスクから構成される．

T31 システム全体の記述：まず，モデル検査ツールの入力言語を用いて，状態遷移モデル（E33）の記述（E42）を作成する．次に，フレームワーク（E31），および制約と仮定（E32）を，フレームワークの記述（E41）として作成する．また，状態遷移モデルとのインスタンスと，それらの間の接続関係を定義して，システム全体の記述を完成させる．必要に応じて，ツールの実行時間と使用メモリ量を削減するために記述の最適化を行う．

T32 品質特性の変換：品質特性（E36）を，LTL 式からツールの入力言語による記述（E43）へ変換する．

2.2.4 記述誤り発見ステップ

本ステップの入力は MCD，出力は単純な記述誤りのない MCD である．タスクは T41 である．

T41 MCD の記述誤り除去：MCD の状態遷移（E42）と MCSM の状態遷移（E33）との対応関係を確認し，単純な記述誤りを除去する．

2.2.5 検証ステップ

本ステップの入力は MCD，出力は検証結果である．検証ステップのタスクは T51 である．

T51 モデル検査ツールを利用した検証の実行：モデル検査ツールの適切なオプションを指定して，品質特性（E43）を検証する．もし満足しない LTL 式が見つかれば，設計誤り発見ステップへ進む．

2.2.6 設計誤り発見ステップ

本ステップの入力は，MCD，MCSM，MCIM，DM，検証結果である．出力は，MCD，MCSM，MCIM，設計誤りである．本ステップのタスクは T61 である．

T61 検証エラーの原因分析：検証結果に含まれる反例を解析し，MCD の中に次の 4 種類の誤りのいずれかの存在可能性を調べる：(F1) 状態遷移の論理的誤り，(F2) 品質特性の設計誤り，(F3) フレームワークの設計誤り，(F4) 検証モデルの変換誤り．次に，MCSM から MCD への変換情報を逆向きに利用して，MCSM の中に誤り（F1）〜(F4) の存在可能性を調べる．同様に，MCIM，DM についても調査する．もしいずれかの検証モデルで設計誤りが見つかれば，上位の検証モデルに対してその対応カ所を調査する．

3. 適 用 事 例

本章では，2 章で提案した検証プロセスを，ネットワーク家電の制御ソフトウェア設計に適用した事例について述べる．まず検証問題の概要を 3.1 節で述べる．次に，3 つのモデル検査ツール，SPIN，SMV，LTSA のそれぞれを適用した検証事例を 3.2 節で述べる．

3.1 検 証 問 題

3.1.1 概　　要

本事例では，**図 7** に示すように，ネットワーク接続されたテレビ，DVD レコーダ，HD レコーダなどのネットワーク家電機器を対象とし，複数機器の連携動作を実現する制御用ソフトウェアの設計検証を問題として扱う．その検証問題では，複数機器の連携動作パターンを網羅的に分析し，ソフトウェア設計から曖昧さや設計誤りを完全に排除しなければならない．しかし，ネットワーク家電に特有の，以下にあげた複雑さが存在する中で，ソフトウェア設計を高品質化する必要があるために，設計検証はコストの高い作業となっている：(1) 機器の振舞いの複雑さ，(2) 機器の不安定さ，(3) ネットワークの不安定さ，(4) ユーザによる想定外の操作，(5) ネットワーク接続される機器の数と種類の多さ．それゆえに，形式的手法の導入による設計検証の効率化に対する期待が非常に大きい．

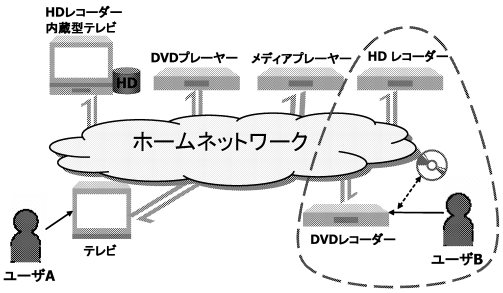

図 7 ホームネットワークシステム
Fig. 7 Home network system.

紙面の都合上，本節では，図 7 に示したシステムのサブセットを考える．具体的には，1 台の HD レコーダと 1 台の DVD レコーダの間のデータコピー操作（図 7 の点線で囲まれたエリア）を対象とし，その制御用ソフトウェアの設計検証を扱う．

3.1.2 入力データ

検証プロセスの入力は，要求仕様，検査仕様，DM の 3 つである．要求仕様の一部を以下に示す．

- HD レコーダに保存されている録画データを，ネットワークを介して DVD レコーダへコピーする．この一連の処理をコピー操作と呼ぶ．
- ユーザの操作によってコピー操作を開始する．DVD レコーダへのコピー操作が完了する前に，ユーザはコピー操作を停止できる．
- 各機器は，コピー操作開始前に必要な資源を予約する．コピー操作終了後，他の機器がその資源を利用できるように資源を解放する．資源の予約と解放はネットワークを介して行う．
- 典型的なシナリオは以下のとおり：(1) DVD レコーダと HD レコーダの電源を入れる．(2) ユーザは DVD レコーダの “copy” ボタンを押す．(3) DVD レコーダは HD レコーダへ，HD レコーダに保存されている録画データの再生開始を要求する．DVD レコーダは HD レコーダから受信したデータのコピーを DVD メディアに格納する．(4) HD レコーダがデータの再生を終了したとき，コピー操作は完了する．(5) 上記 (3) のコピー処理中の任意のタイミングで，ユーザは “stop” ボタンを押し，コピー操作を中断できる．

検査仕様の一部を以下に示す．

- コピー操作の事前条件は，HD レコーダと DVD レコーダの両機器の資源が解放されていること．
- 両機器がコピー操作の実行を完了した後であれば，ユーザから別のコピー操作要求を受理し，コピー操作を開始できる．

DM は 2 つのモデル要素 E11 と E12 から構成される（2.2.1 項を参照）．本事例では，図 3 に示した DVD レコーダと HD レコーダの状態機械図を E11 とし，UML セマンティクスに関する以下の仮定を E12 とする

- 状態機械のイベントキューの長さは 1 とする．キューに格納されているイベントが消費されると，到着したが消費されていない他のイベントは破棄される．イベントが消費されなくても，ガード条件が満たされれば状態遷移が引き起こされる．

3.2 検証プロセスの実施例

3 つのモデル検査ツール，SPIN，SMV，LTSA のそれぞれを利用した検証プロセスを**表 1** に示す．表 1 は，2 章で示した検証プロセスの定義に沿って，各ツールを利用した場合のタスクの具体的な作業内容や，ツール固有の実用的ノウハウを示したものである．

本節では，ツールに固有のタスク作業やノウハウに着目して，これら 3 つの検証プロセスを適用した検証事例について述べる．

3.2.1 検証要求分析ステップ

まず T11 で，ネットワーク家電のユーザモデルをシステム環境（E23）の一部として作成した．ユーザモデルは，2 つの異なるイベント `start_copy` と `stop_copy` を非決定的に生成する状態機械として作成した．

次に T12 では，3.1.2 項に示した検査仕様に基づいて，HD レコーダの状態機械の進行性に関する 2 つの品質特性を考える．

到達可能性：HD レコーダは初期状態から，任意のイベントを待つ状態 `waiting` へ到達できること．

活性：HD レコーダの状態が `waiting` である場合にコピー操作開始イベントが到着すれば，いつか必ず状態 `ready` へ遷移すること．また，HD レコーダの状態が `ready` であれば，いつか必ず状態 `waiting` 状態へ戻ること．

これらの品質特性は以下の LTL 式（E25）で表される．

到達可能性：`F (hd.state=waiting)`
活性：
```
G (hd.state=waiting -> F (hd.state=ready))
  && G (hd.state=ready -> F (hd.state=waiting))
```

T13 では，UML 図（E11）に対して次の抽象化を行うことにより，抽象 UML 図（E21）を作成した．まず，HD レコーダと DVD レコーダはすでに電源が投入された状態にあり，その後の振舞いのみを考えると仮定し，2 つの電源投入イベント `on_DVD` と `on_HD` を削除した．次に，内部イベントおよび “do” アクションは，機器間の相互作用に直接関わるものではないので削除した．HD レコーダの状態機械図は録画機能に関する 1 つの状態と 2 つの状態遷移を含んでいるが，録画機能は検証の関心事ではないので，これらの状態および状態遷移を削除した．図 3 を抽象化した結果が，図 4 の状態機械である．

3.2.2 設計ステップ

以降の 3.2.2 項，3.2.3 項，3.2.4 項では，モデル検査ツールごとに検証ステップの作業内容を示す．紙面の都合上，SMV と LTSA についてのみ述べる．SPIN を利用した検証作業は，表 1 を参照されたい．

SMV まず T21 で，各イベントをブール型の変数としてモデル化し，複数の家電機器は非同期実行モデルとして表す（E31）．また T22 で，MCIM の状態機械からスーパステートを削除し状態機械を平坦化する．次に T23 で，DVD レコーダ，HD レコーダ，ユーザモデルの状態機械をそれぞれ FSM（E33）へ変換する．最後に T24 で，FSM 間の実行スケジューリング，および状態遷移の非決定的遷移に対して公平性制約（E34）を定義し，LTL 式で記述する．SMV 版 MCSM の一部を図 5 に示す．

LTSA まず T21 で，通信フレームワークを作成する．次に LTSA ではスーパステートを直接表現することが困難なので，T22 で MCIM の状態機械からスーパ状態を削除し状態機械を平坦化する．次に T23 で，DVD レコーダ，HD レコーダ，ユーザモデルの状態機械をそれぞ FSP 記述へ変換する．最後に，LTSA は公平性制約をサポートしていないので，T24 では定義しない．

3.2.3 実装ステップ

MCSM から MCD を作成する．

SMV FSM を SMV ツールの入力言語で記述する．図 6 は SMV 版 MCD の一部を表しており，図 5 の MCSM を変換して得られた．2–25 行目が HD レコーダの状態遷移定義を表し，27–29 行目が 2 つの品質特性，到達可能性と活性の記述を表している．

LTSA FSP を LTSA ツールの入力言語で記述する．**図 9** は LTSA 版 MCD の一部を表しており，**図 8** の MCSM を変換して得られた．2–6 行目は状態遷移定義を，7，8 行目は 2 つの品質特性，到達可能性と活性の記述を表している．

表 1 モデル検査ツール SPIN，SMV，LTSA を利用した検証プロセス
Table 1 Verification process instances using SPIN, SMV and LTSA.

タスク	タスクの一般的な例	SPIN	SMV	LTSA
T11	関心事を選定する ・関連するユースケースや機能を選択する 何を検証するかを決定する ・シナリオベースやゴール指向の分析手法を利用して検証項目を洗い出す			
	環境や仮定を考察し，検証の範囲を定める ・起こりうる状況を洗い出す ・仮定のもとでの特殊な振舞いを洗い出す	・タイムアウトの起こりうる状況をテーブルで網羅的に分析する ・UML図の曖昧な部分の洗い出し，振舞いを確定する		
T12	ユーザの視点で品質特性を定義する ・ユースケースからLTL式の導出する	ユースケースにおいて例外ケースやミスユースケースを考え，その記述からLTL式を導く		
	設計者の視点で品質特性を定義する ・シーケンス図からLTL式を導出する	通常システムとして成り立つべき品質性質をインスタンス化する ・応答性：リクエストに対するレスポンス ・デッドロックフリー，ライブロックフリー，進行性，不変性		
T13	関心事と関連のない部分を抽象化，削除 ・Sink/Sourceプロセス削除，カウンタ抽象化を実施する ・状態の合成，同一構造部分の共通化を実施する	・タイムアウトに関連するイベントと状態だけをスライシングする		
T21	並行動作メカニズムを設計する ・スケジューラを設計する	・受信可能なイベントをランダムに1つ選択する非決定動作としてモデル化 ・プロセスの生成条件をprovidedで与える	SMVの非同期実行機能を利用する	ラベルの優先順位を利用したスケジューラを設計する ・timeoutの優先順位を下げる
	通信メカニズムを設計する ・通信ポートや通信バッファを設計する	・通信相手ごとに通信バッファを作成 ・Promelaの通信チャネルを利用	通信バッファを作成する ・通信チャネルをプロセスとして定義する ・通信相手ごとに通信バッファを変数でモデル化する ・通信バッファを共有変数でモデル化する	通信用プロセスを並列プロセスとして定義する
	時間制約やタイムアウトを設計する	・タイムアウトの発生は，明示的なメッセージ送受信でモデル化する	タイムアウトは非決定的遷移としてモデル化する	タイムアウトを通信用プロセス中で発生させる
	その他：システムの振舞いを変更しないようにモデル化を工夫する ・探索空間を調整する ・モデル化に非決定性を利用する	・非決定条件文を利用する	・環境や振舞を制約するものは，探索空間を制約するLTL式として記述する ・非決定的遷移やプロセス間非同期実行に対して，公平性の導入を検討する	・同じ振舞いをするプロセスを添え字で表現する ・ラベルの出現順序を制限するプロセスを並行動作させることで探索空間を制御する ・非決定遷移を加える
T22	等価変換： ・スーパステートをモデル化，あるいは抽象化のために削除する	スーパステートをモデル化する ・スーパ状態とサブ状態の両方を変数で表現する	・状態変化を状態変数でモデル化する ・ネストした各状態機械を1つのプロセスで定義する	・各状態をラベルでモデル化する
T23	MCIMからMCSMへ変換する ・各状態遷移，プロセスごとにパラメータをインスタンス化する ・イベント，状態を変数に変換する	・1つのFSMを1つのプロセスに対応付ける ・プロセス間通信にはPromelaのチャネルを利用する	・process宣言を利用して，プロセス間の非同期実行をモデル化する ・各イベントをboolean変数として表す	・FSPとLTSは等価なので，特別な変換規則を定める必要はない ・プロセスにインデックスをつける
	検証のための仮定情報をMCSMに付加する ・状態空間を調整する ・遷移の条件を追加する	FSM中の遷移に条件を追加する	・検証のための仮定をLTL式として記述する ・探索空間を制約するLTL式を記述する	仮定として成り立つシーケンスをプロセスとして表現する
T24	スケジューラを調整して公平なスケジューリングを行う ・LTL式で記述する ・ツールのオプションで設定する	weak fairnessオプションを利用する	非決定的遷移およびプロセス間非同期実行に関する公平性制約をLTL式で定義する	通信用プロセスのスケジューリングを調整する
T31	記述を最適化する ・マクロやインラインを利用する ・探索空間やデータ型を最適化する	・インラインやマクロで置き換える ・変数のに初期値代入しておく	変数のビット列表現で置き換える	・ラベルのhidingにより，通信範囲を限定する ・プロセスが受け入れるアルファベットを限定する
T32	・ツールのオプションや機能を使って検証できることと，それ以外とを切り分けて記述する ・進行性のための記述を追加する ・assert，claimを追加する	・各検証オプションの指定を検討する ・progressラベルを追加する ・progressを各プロセスごとに付加して検証を行うことを検討する ・進行性検証のためのLTL式を追加する	特定の品質特性を検証する機能は用意されていない．LTL式をSMVツール言語で記述し，MCDへ変換する ・検証するLTL式を，assert文で記述する ・using ～ prove ～文で，仮定となるLTL式と，検証するLTL式を指定する	・safety，progressオプションの指定を検討する ・進行性のための記述をprogressで与える、性質をBuchiオートマトンで与える ・公平性検証は，各プロセス中のラベルに関するprogress記述で実施することを検討する
T41	・MCDの中に故意に設計誤りを入れて，ツールに反例を出力させる	assert(false)を挿入して，反例を出力させる	falseとなるLTL式を意図的に与えて，反例を出力させる	モデルの中に，ERROR Stateを挿入する
	検証モデルの実行経過を観察するために，補助的な変数や状態を導入する	基本的には不要．ログ出力だけで十分である	マクロ変数を導入してシステムのスナップショットを表現する（状態空間には影響なし）．その変数の値の変化を反例上で観察する	・モニタプロセスを定義する ・状態やイベントにラベルを付け，その生成を反例上で観察する
	ブレークポイントを挿入する	・Error文による指定 ・実行カウンタによる指定	特定のスナップショットに到達したときにfalseとなるLTL式を与える	Error Stateを利用して指定する
	その他 ・所望のトレースをとるための記述を追加する ・成り立つはずの性質をチェックしてみる	ループの箇所にprogressラベルを意図的に除外して，Non-Progress Cyclesオプションを指定して検証を行い，検証エラーとなることを確認する	・システムのスナップショット変化を追跡するようなLTL式を与える	・モニタプロセスを定義し，その振舞いがシステムの振舞いとが対応することを検証で確認する ・不成立が自明なラベルに対してprogressをオプションを指定して，反例を出力させる
T51	・検証オプションを切り替えて検証を実施する ・LTL式が正しいかどうかを確認する －LTL式に意図的に誤りを入れて検証する －状態遷移に意図的に誤りを入れて検証する ・状態変化を追跡しやすいLTL式の検証から開始して，徐々に複雑な式を検証する	・到達可能性は“Report Unreachable Code”の設定により検証する ・デッドロックは“Invalid Endstates”オプションの指定により検証する ・公平性制約はwaek firnessオプションを指定する ・LTL式を段階に検証する	・単純なLTL式から順に検証する ・複雑なLTL式を部分式に展開して，部分的に検証し，順にLTL式を統合する ・LTL式が表している状態変化の中間状態を確認する式を挿入して検証し，徐々に式を伸ばしていく ・探索空間を制限した検証から開始し，徐々に制限をゆるくする探索空間を広げる	・デッドロックはsafetyオプションで検証する ・進行性はprogressオプションで検証する ・簡単なprogress文から開始して，徐々に複雑なprogress文を検証する
T61	振舞いの誤りかどうかの観点で調べる 動作確認方法(T41)を利用して，下記を確認する ・進行していない変数・遷移の有無 ・意図しないイベント受信，状態遷移の有無 ・意図しないイベントの消失の有無 ・品質性質の成立・不成立 振舞いの誤りを発見したら，それをモデル化（たとえばLTL式を作成）して検証を行い，振舞いを確認する	・特定の行を通過したかどうかを確認する ・反例トレース解析時に，変数値出力ウィンドウで，変数値が正しく変化しているかどうかを確認する ・XSPINによるシーケンス図を利用する ・振舞いの設計誤りに関するLTL式を与えて，その性質が満たされるかどうかをチェックする	・システムのスナップショットをマクロとして記述する ・時系列表から，値が変化していない変数を確認 ・状態遷移にラベルをつけて，その値がtrueになるかどうかを調べることにより，その状態遷移の発生を確認する	・スナップショットを表すラベルを挿入する ・ANIMATORを使って次に実行可能なラベルが意図したものかどうかを確認する ・意図しない状態遷移をprogressで表現する ・イベント消失ラベルが意図していないところで現れていないかを確認する ・振舞いに関する設計誤りを，モニタプロセスで表して検証する
	品質特性とLTL式の与え方が間違っているかどうかの観点で調べる ・単純なLTL式や部分式を与える ・必ず成立すべきLTL式の成立を調べる ・単純な状態遷移に対して検証を行い，与えた品質特性の成立（不成立）を確認する	・naver claimに変換されたLTL式を確認し，その記述を変えてみる ・progressラベルの位置を変えてみる	・品質特性のLTL式を，状態変化途中のスナップショットも確認できるような式と置き換えて検証する ・複雑な式を複数の単純な式に分解し，各式の成否を1つずつ確認する ・意図的に振舞いを変更して反例を出力させる ・using～prove～文を使って探索空間を制御し，部分空間に対して検証する	・progressをさまざまなラベルで確認してみる ・progress if thenをさまざまなラベルで確認してみる ・モニタプロセスを導入して振舞いを制限し，progressを検証してみる
	変換に誤りがあるかどうか，変換法が正当かどうかの観点で調べる ・簡単な検証モデルを変換してみる ・動作確認方法(T41)を利用して，変換前でも成立するはずの品質性質を検証する	・変換前モデルの振舞いをLTL式で記述し，MCDに対して成り立つか調べる．	・変換前モデルで成立すべき品質特性のLTL式を記述し，MCD上でその成立を検証する ・変換前モデルの振舞いをLTL式で記述し，MCD上でその成立を検証する	・変換前モデルの振舞いを表すモニタプロセスを記述し，MCD上で並行合成してその振舞いを検証する ・変換前モデルで成立すべき品質性質をprogress if then構文で記述し，MCD上でその成立を検証する
	振舞いがセマンティクスに従っているかどうかの観点で，動作確認方法(T41)を利用して調べる ・簡単なモデルで振舞いを確認する ・セマンティクスの各定義どおりの振舞いであることを確認する ・セマンティクスと矛盾する状態遷移，イベント受信，イベントの消失の有無を調べる	・セマンティクスの定義どおりの振舞いを表すLTL式を与えて，振舞いを調べる． ・起こりえない性質をnaver climで定義する ・LTL式を与えて，各イベントの送受を調べる．意図しないイベント受信やイベントの消失があるか確認する	・セマンティクスの定義どおりの振舞いを表すLTL式を与えて，振舞いを調べる ・LTL式を与えて，各イベントの送受を調べる ・セマンティクス上起こりえないスナップショットにラベルを付けて，その成立を調べる	・セマンティクスの定義どおりの振舞いをモニタプロセスとして作成する ・意図しないシーケンスをプロセスとして記述する ・意図しない振舞いが起こっていそうな部分にERROR状態を挿入する

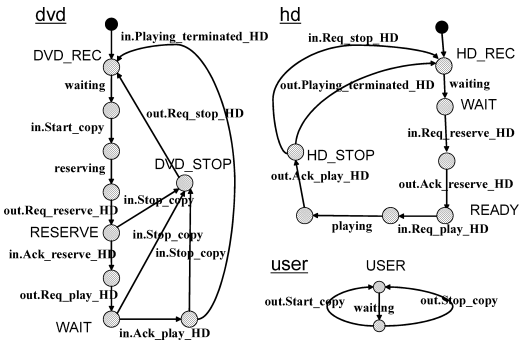

図 8 LTSA 版 MCSM の要素 E33
Fig. 8 Element E33 of the MCSM for LTSA.

```
1.  // definition of HD recorder
2.  HD_REC = (waiting -> WAIT),
3.  WAIT = (in[Req_reserve_HD] -> out[Ack_reserve_HD] -> READY),
4.  READY = (in[Req_play_HD] -> playing -> out[Ack_play_HD] -> HD_STOP),
5.  HD_STOP = (in[Req_stop_HD] -> HD_REC
6.                |out[Playing_terminated_HD] -> HD_REC)+{out[Event]}.

7.  progress HD_WAIT = {hd.waiting}
8.  progress HD_PROG = if {hd.waiting} then {hd.playing}
```

図 9 LTSA 版 MCD の例
Fig. 9 Example of a MCD for LTSA.

3.2.4 設計誤り発見ステップ

SMV による設計誤り発見

活性の検証結果は false となり，HD レコーダの状態遷移が進行していない可能性があることを意味している．SMV が出力した反例の一部を**図 10** に示す．図 10 は，4 つの変数それぞれの値が，クロックの進行とともに変化している様子を表している．変数 dvd.state，hd.state は DVD レコーダと HD レコーダの状態を表し，他の 2 つの変数はともにイベントのアクティブ状態を表している．

- DVD レコーダはイベント stop_copy を消費し，状態 reserving から waiting へクロック 9 で戻る一方，イベント req_stop_HD を HD レコーダへ送信している．
- HD レコーダは，状態 ready でイベント req_play_HD の到着だけを待ち続け，他の状態へは遷移していない．

ここで 2 つの変数 hd.req_stop_HD，hd.state の値の変化を調べる．DVD レコーダは，変数 dvd.set_req_stop_HD に true をセットすることによって，クロック 12 で変数 hd.req_stop_HD の値が true に変化している．その後，クロック 16 で変数 hd.req_stop_HD の値が false に変化しているが，変数 hd.state の値は ready のままである．これは，イベント hd.req_stop_HD が HD レコーダで消費されずに破棄されていることを意味する．以上から，MCD

System clock	8	9	10	11	12	13	14	15	16
dvd_state	reserving	waiting	waiting	waiting	waiting	waiting	waiting	waiting	waiting
dvd.set_req_stop_HD	0	1	1	1	1	0	0	0	0
hd.req_stop_HD	0	0	0	0	1	1	1	1	0
hd_state	ready	ready	ready	ready	ready	ready	ready	ready	ready

図 10 SMV を利用したタイミングエラーの発見
Fig. 10 Detecting a timing error with SMV.

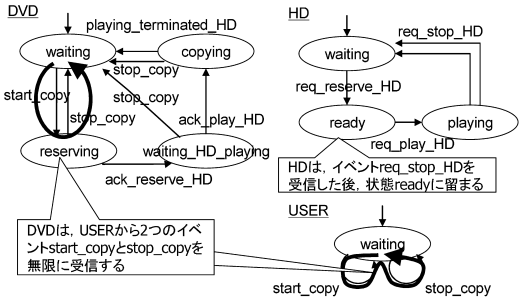

図 11 SMV 版 MCSM に対するエラーの追跡
Fig. 11 Tracing an error in MCSM for SMV.

における設計誤りは，HD レコーダの状態 ready において，イベント hd.req_stop_HD を消費する遷移が存在しないことであると結論づけることができる．これは図 6 の 13 行目に対応する．

次に，MCD で発見した誤りをもとにして，MCD から MCSM へ設計誤りの追跡を行う．図 6 の 13 行目に対応する**図 11** の状態は HD レコーダの状態 ready であるが，状態 ready を起点とし，かつイベント req_stop_HD を消費する状態遷移が定義されていないことが分かる．ここで，HD レコーダの状態が ready であるときにイベント req_stop_HD を受け取った場合について考える．イベント req_stop_HD は消費されずに破棄され，HD レコーダは状態 ready にとどまったままで，他の状態へ遷移しない．このように，検証エラーの原因に関して，MCD と MCSM のそれぞれにに対する分析結果が一致する．したがって，MCSM における設計誤りは，HD レコーダの状態 ready において，イベント hd.req_stop_HD を消費する遷移が存在しないことと結論づけることができる．

同様に，MCSM から MCIM への設計誤りの追跡を行う．その結果として，MCIM における設計誤りは，HD レコーダの状態 ready において，イベント hd.req_stop_HD を消費する遷移が存在しないことであると結論づけることができる．

LTSA による設計誤り発見

LTSA では，活性を調べることにより，状態遷移の全ループが公平に進行していることの検証を実施する．その結果，HD レコーダの状態遷移の中で，ループが進行しないトレースを発見した．**図 12** は，LTSA が出力した実行トレースの一部を示している．トレース

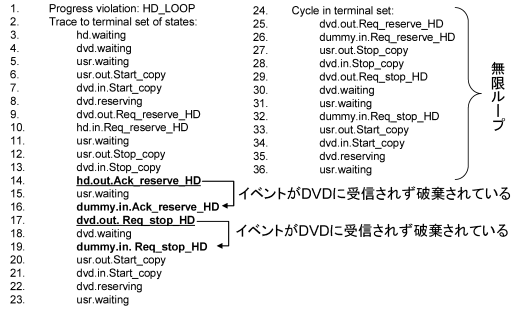

図 12 LTSA を利用したタイミングエラーの発見
Fig. 12 Detecting a timing error with LTSA.

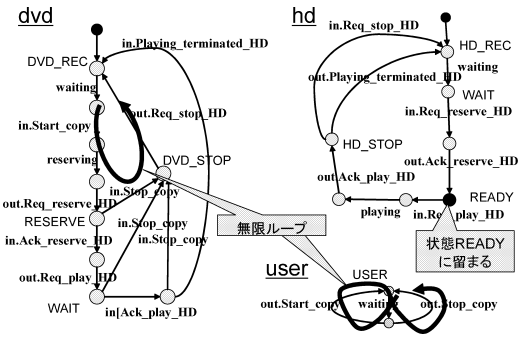

図 13 LTSA 版 MCSM に対するエラーの追跡
Fig. 13 Tracing an error of MCSM for LTSA.

各行は “.” で結合されたラベルを表しており，右から順に，プロセス名，入力または出力，イベント名を表している．このトレース中に文字列 “Cycle in terminal set” が含まれている場合，無限ループが存在することを意味する．トレースと LTS 図の対応を確認することによってその無限ループを発見できる．その結果として，**図 13** に示したように，HD レコーダの状態 `READY` が進行せず，DVD レコーダと USER との間の通信が無限に繰り返されていることが分かった．

次に，イベントを消費するダミープロセスを定義して，イベント破棄をモデル化した．ダミープロセスのラベルに着目しながらトレースを追跡した結果，UML セマンティクスとは異なるイベント消失が生じていることを確認できた．具体的には，イベント `ack_reserve_HD`（LTSA の記述では `hd.out.Ack_reserve_HD`）は，HD レコーダによって送信された（トレースの 14 行目）が，破棄されている（同 16 行目）．破棄の後，イベント `req_stop_HD` が DVD レコーダによって送信された（同 17 行目）が，HD レコーダによって消費されずに破棄されている（同 19 行目）．

LTSA のトレースを分析した結果から，HD レコーダはイベント `ack_reserve_HD` を送信した後にイベント `req_stop_HD` を処理していないことが分かった．少なくとも，MCSM の `READY` 状態で `req_stop_HD` を消費できる状態遷移が必要である．MCSM と MCIM の間の対応関係を見ることによって，MCIM においても同様に，`ready` で `req_stop_HD` を消費できる状態遷移が必要であると結論づけることができた．

4. 考　　察

本章では，モデル検査手法を利用した検証プロセスの研究成果について，適用事例をもとにして考察する．本成果は次の 2 つの観点から述べる．(1) 検証プロセスは，モデル検査に関する実用的ノウハウをプロセスの観点で体系化するものである（4.1 節，4.2 節），(2) 検証プロセスは，検証モデルと検証タスクとの間にある依存関係を明確化するものである（4.3 節，4.4 節）．最後に，提案プロセスの新規性とスケーラビリティについて述べる（4.5 節，4.6 節）．

4.1 実用的ノウハウの明確化

提案プロセスは，モデル検査における活用ノウハウを列挙しているだけではなく，検証プロセスの各タスクで利用可能な実用的ノウハウを明確化している．たとえば，一般に，開発者は検証モデルを作成するときに，抽象化技法を利用して検証の関心事に関連したアスペクトや機能性の側面だけを抽出するが，2.2 節で述べたように，抽象化技法の利用目的や利用観点は，検証プロセスのタスクごとに異なる．T13 では，コピー操作を検証の関心事とし，関心事に関連する設計情報だけを抽出して MCIM を作成した．一方，T22 では，利用するモデル検査手法の表記法の記述力に応じて，MCIM の UML 図を抽象化している．さらに，T31 では，設計ステップで作成したフレームワークを統合した後に，検証コスト（ツールの実行時間や使用メモリ量）を削減するために，MCD の制御フローやデータ構造を抽象化して記述の最適化を行う．

次に，既存の活用ノウハウや関連研究を，提案プロセスのタスクへの対応づけを**表 2** に示す．横軸が提案プロセスの検証タスク，縦軸が関連研究を表す．関連研究の技術やノウハウを活用できるタスクに “x” 印を付けている．関連研究の内容については 5 章で述べる．表 2 を見ると，関連研究は T61 を除いた全タスクを網羅している．つまり，検証エラーの原因分析を行う T61 で有効な実用的ノウハウがいまだ提案されてきていないことを意味する．本研究では，T61 で有効な活用ノウハウを 4.4 節で述べる．以上のように，検証プロセスの各タスクで利用可能な活用ノウハウを示しているので，提案プロセスは実用的な検証指針を提示するものであるいえる．

表 2 検証タスクと既存研究との関連
Table 2 Relations between verification tasks and related works.

	T11	T12	T13	T21	T22	T23	T24	T31	T32	T41	T51	T61
（ミス）ユースケースから検査仕様を作成[21]	x											
仮定の生成[10]	x					x						
UML 記述から LTL 式の生成[20]		x										
抽象化[6]			x		x							
UML 図に対するモデル検査[19]				x		x		x				
半順序手法[12]						x					x	
モデル検査ツールのマニュアル							x	x	x	x	x	
BDD サイズの最適化[4]								x			x	
検証実行時間の削減[18]								x			x	
使用メモリ量の最適化[18]								x			x	
LTL 式から有限状態オートマトンへの変換[9]									x			
限定モデル検査[2]											x	

4.2 検証作業の難しさ

本節は，検証作業における難しさについて議論する．検証タスクごとに，検証問題に依存した多種多様な手法が提案されており，必ずしもすべての検証タスクを系統的でかつ汎用的な方法で実施できるわけではない．したがって，開発者は自身の経験や検証問題に応じて適切な手法を選択しなければならない．つまり，多様な手法を使い分けられるだけの経験的スキルやノウハウの習得が求められる．表 1 の中で灰色で示した部分は，そのような習得が求められるタスクを示している．

一般に，次にあげたタスクは，特定のモデル検査手法には依存しないものの，経験の体得を必要とするものである：(a) T21 で実施するセマンティクスのモデル化とフレームワークの実現，(b) T23 で実施するシステム環境および検証の仮定のモデル化，(c) T31 で実施する MCD の最適化，(d) T51 で実施する LTL 式の妥当性検証，(e) T61 で実施する検証エラー発見および設計誤りの追跡．

次に，各モデル検査ツールの活用ノウハウを必要とするタスクについて論じる．

SPIN SPIN は通信プロトコル検証を目的として開発されたため，ランデブー通信による振舞いのブロックや，非同期通信でのバッファ溢れなどのプロトコル検証問題に固有のモデルが Promela の通信セマンティクスに含まれている．一般の並行システムや分散システムを SPIN で検証する場合に，問題に特化したモデルがネックとなる場合がある．つまり，Promela のセマンティクスが検証問題と合致していれば SPIN を素直に適用できる．しかし，一致しない場合は，タスク T21 で問題に適したフレームワークや通信セマンティクスを Promela で実装する必要があり，その実装は必ずしも容易ではない．SPIN は ω-オートマトン理論に基づいたモデル検査手法であるが，上記の難しさは，ω-オートマトン理論ではなく，通信プロトコルのモデル化に特化して開発された Promera 言語に起因するものである．

SMV SMV は回路検証を目的として開発されたため，通信モデルは用意されていない．そのため，共有変数や通信バッファなど，問題に応じて適切な通信モデルを開発者自身が T21 で作成する必要がある．また，SMV は，インタラクティブ実行やブレークポイント設定などの高度なデバッグ機能を備えていないので，T41 と T61 でシステムの実行スナップショットを表す式や，わざと `false` となる式を与えることによって，振舞いを確認する．次に，SMV は，検証したい振舞いを，基本的にシステムの実行スナップショットの変化を表した LTL 式で与える．つまり，SMV を利用した検証作業の大部分は，LTL 式の利用に依存しており，LTL 式を記述する技術と経験とが開発者にとって必要不可欠である．このような難しさは，SMV を含む記号モデル検査手法にとって共通の課題である．

LTSA LTSA では，否定を表現できないなど LTL 式の記述力が低いので，ツールが提供する機能や雛形を使い分けて品質特性を検証する．たとえば，活性を検証するために `progress` 文を利用する．また，イベント実行順序に焦点をあてて検証する場合には，LTL 式ではなくモニタプロセスを作成して検証することができる．また，検証問題に応じて，通信メカニズムなどのフレームワークを記述する必要があるが，SPIN や SMV と比べて少ない記述量で作成できる．なお，LTL 式の記述に関する制約は，LTSA のベースになっている FSP に起因するものではなく，LTSA ツールに固有のものである☆．また，状態ではなくイベント

☆ LTSA の最新版は，LTL 式の記述力に制約はあるものの，任意の LTL 式から状態遷移定義へ変換する機能を提供している．

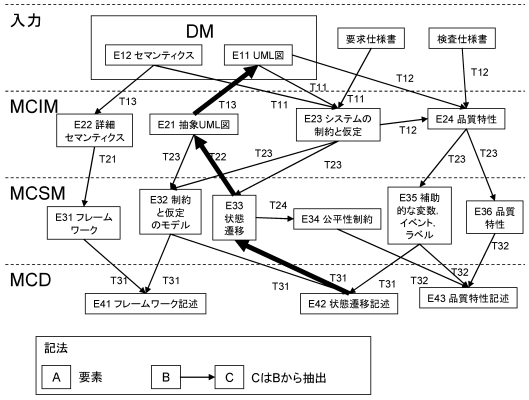

図 14 デバッグプロセスの例
Fig. 14 Example of debugging process.

に着目してシステムの振舞いを記述することは，プロセス代数に基づいた検証ツールに共通するものである．

4.3 デバッグプロセスの定義

2 章で述べたように，提案プロセスの各タスクを，検証モデルの要素間の変換手続きとして定義したことにより，要素とタスクの間の依存関係が明確になった（図 2）．この依存関係は，モデル検査ツールの実行結果が検証エラーであった場合，その検証エラーから設計誤りを追跡するための指標となる．たとえば，3 つの要素，フレームワーク（E41），状態遷移の記述（E42），品質特性（E43）は，図 2 に示したように，MCD の構成要素として作成される．もし検証エラーが見つかった場合，その原因がこれら 3 つの要素のいずれかに含まれている可能性がある．その原因分析はタスク T61 で実施される．検証事例では，HD レコーダの状態遷移記述（E42）の中に設計誤りが発見された．

9 つのタスク T11 から T32 までは，検証モデルの要素間の変換として定義したが，設計誤りのデバッグではこれらの要素間の逆変換を利用できる．デバッグプロセスの例を**図 14** に示す．もし状態遷移の記述（E42）が設計誤りを含んでいれば，MCSM における制約記述（E32）あるいは状態遷移（E33）にも不具合が含まれている可能性がある．E35 は開発者による検証作業を効率化するために導入した補助変数であるので，一般に設計誤り混入の可能性は低い．検証事例では，HD レコーダの状態機械の記述（E42）の中に設計誤りを発見した．

依存グラフにおいて，ノード間のリンクを逆方向に追跡し設計誤りを調べる手続きは，全体でデバッグプロセスを構成している．このデバッグプロセスを利用することによって，検証モデルのどの要素が設計誤りを含まれている可能性があるかの分析を行うことができる．つまり，デバッグプロセスは，不具合発見のチェックポイントを示している．したがって，開発者はこの追跡可能性を利用することによって，系統的に検証モデルの設計誤りを調査することができる．

4.4 実用的なデバッグプロセスの活用法

4.1 節に示したように，T61 は新しい活用ノウハウを必要とするタスクであり，T61 の 4 種類の作業それぞれに対する活用ノウハウを表 1 に示した．各作業は，検証モデル，システムの振舞い，品質特性，セマンティクス，検証モデルの変換過程で混入した設計誤り，それぞれのデバッグの観点を与えている．たとえば，もし意図しないイベント受信と状態遷移を反例で発見したならば，システムの振舞いに関する論理的誤りを検出できるかもしれない．そのような検出を行うためには，SPIN では GUI 付きのデバッグ機能と LTL 式を利用し，SMV では記述力の高い LTL 式を駆使し，LTSA はモニタプロセスを定義してシステムの振舞いを観察する．

また，4.3 節でデバッグプロセスを明確に定義したことにより，開発者はデバッグプロセスを実践的かつ容易に利用することができる．具体的には，以下の 3 つがデバッグプロセスの成果である．(1) 上記で述べたように検証エラー分析調査に対する 4 種類の観点を提示した，(2) 各観点で利用可能な分析方法を示した（表 1 の T61），(3) 図 14 に示した依存関係を利用することにより，上位の検証モデルの中で設計誤りが含まれている可能性が高い要素を絞り込むことができる．

4.5 提案プロセスの新規性

実用ソフトウェアの設計検証作業では，多数の実践すべき検証タスクがあり，検証タスクごとに，検証問題や検証技術に依存した多種多様な手法が提案されている．検証作業に熟練した技術者であれば，自らの経験に基づいて，実施すべき検証タスクや有効な手法の選択を容易に実践することができるが，一般の開発者にとって，多種多様な手法の中から検証問題に応じて適切なものを選択することは容易ではない．本研究では，検証に必要な作業内容と順序を体系化し，各検証タスクで実戦可能なノウハウを整理すること自体が，本質的に重要な課題であると位置づけ，検証プロセスを 2.2 節で詳細に定義し，各検証タスクで実戦可能なノウハウを表 1 に整理している．ところが，このような課題を正面から取り扱い，検証プロセスと実践的ノウハウとを系統的に論じた研究は従来報告されていない．

また，本研究では，理論的背景の異なる代表的な 3

つのモデル検査ツール SPIN，SMV，LTSA を採用し，各ツールを利用した検証で必要となる具体的な作業内容や実践的ノウハウを顕在化している．その結果として，ツールに依存しない汎用的な検証プロセス（プロセスのクラスに相当）を 2.2 節で詳細に定義し，ツール独自の実践的ノウハウを対応させたツールごとの検証プロセス（プロセスのインスタンスに相当）を表 1 に示している．実践的ノウハウをツール依存と非依存とに分けて顕在化すること自体がノウハウであり，従来研究にない着眼点である．

たとえば，分散システムの設計検証の場合，一般に通信メカニズム（通信ポート，通信チャネルなど）の検証モデルを設計する検証タスク T21 を実行する．しかし，通信メカニズムのモデリング方法は，使用するモデル検査ツールの入力言語ごとに異なる（表 1 の T21 を参照）．SPIN の入力言語 Promera は通信チャネル（バッファ）の記述方法を提供しているため，分散システムの通信メカニズムを直接的に表現可能である．SMV では，システムの振舞いを状態変数の変化で表すため，通信チャネルを共有変数でモデル化する．LTSA では，システムの振舞いをイベントの実行順序を規定したプロセスとして表すため，各通信チャネルを 1 つの並行プロセスとしてモデル化する．

また，検証エラーの原因追跡を行うタスク T61 では，与えた品質特性とその LTL 式の誤り可能性を分析し，与えた品質特性とその LTL 式に誤りが存在する可能性を検討する．たとえば，設計の一部に対して LTL 式を評価する，LTL 式の部分式を評価するなどの手法は，ツールに依存せず LTL 式の正しさを検証する典型的な手法である．一方，LTL 式の成否を確認する具体的手法はツールごとに異なる（表 1 の T61 を参照）．SPIN では，progress ラベルの位置を変更することによって，LTL 式の成否の変化を確認する．SMV では，探索の対象とする状態空間を LTL 式で制御して確認を行う．LTSA では，実行イベントのモニタプロセスを定義し，検証したい品質特性の成立を分析する．

4.6 提案プロセスのスケーラビリティ

抽象化や最適化を行う手法が数多く提案されており，またそれらの手法を組み込んだモデル検査ツールも存在している．しかし，それらの手法は開発規模の増加が進んでいる現状のソフトウェア開発において，状態爆発問題に対する十分に本質的な解となるものではないのが実状であるが，スケーラビリティを改善するアルゴリズムを提案することが本稿の目的ではない．

実際のソフトウェア開発では，モデル検査ツールの単純な利用では状態爆発問題に直面してしまうため，開発者がツールを利用する前に検証対象の設計に対して戦略的に抽象化や分割を施し，ツール利用後に検証エラーの原因を系統的に追跡しなければならない．本研究では，これらの不可欠な作業を系統的に整理し，汎用的な検証プロセスとして定義するものである．具体的には，検証要求分析，検証モデル設計，検証モデル実装の各ステップで状態爆発問題に関連するタスクは，T13，T21，T31 を定義している．また，検証プロセスの過程で検証モデルを設計するうえで，着眼すべき観点を検証モデルの構成要素として顕在化し，実施すべき検証タスクと対応付けている（図 2 を参照）．各検証タスク，および検証モデルの各構成要素は，大規模なソフトウェア開発においても不可欠なものであり，また，ネットワーク家電の制御ソフトウェアの設計検証問題への適用実験を通して，提案プロセスの有効性を確認している．

5. 関 連 研 究

検証を，ソフトウェアプロセスと見なしている関連研究について述べる．Gluch ら[11] のプロセスは，ソフトウェアの開発プロセスの中にモデル検査手法を組み込んだもので，要求仕様や設計仕様などの開発プロセスの生成物から検証モデルを作成し，モデル検査手法を適用する．さらに，モデル検査によって発見した不具合を分析し，元の生成物へ反映させる指針を示している．しかし，彼らのプロセスは指針を示すのみで，具体的な手順や適用事例，特定ツールの利用法などはいっさい示していない．Myers ら[16] は，SMV を利用した検証プロセスを提案しているが，その内容はツールのマニュアルから容易に導ける範囲のものであり，適用事例も実用的ノウハウも示していない．Xie ら[22] は，モデル検査をオブジェクト指向開発プロセスへ組み込んだ方法論を提案している．その方法論はモデル検査ツールに依存しないために一般に利用可能なノウハウであるが，実質的にはオブジェクト指向分析モデルからオートマトンへ変換する手順を示しているにすぎない．一方，本研究では，汎用的な検証プロセスを提案するだけでなく，3 つのモデル検査ツールそれぞれを利用した検証プロセスのインスタンスも示している．さらに，各インスタンスを適用した検証事例を示し，その結果から，検証プロセスの観点から実用的ノウハウを顕在化し，検証モデルと検証活動とを具体的に示している．

共通例題へモデル検査手法を適用した結果をもとに，複数手法の比較を論じた研究が報告されている．たと

えば，複数の小規模問題[3]，ユーザインタフェースの設計問題[1),17)]，通信プロトコルの設計問題[8)] などがある．これらの報告は，検証実行時間と使用メモリ量を含むツール性能の比較を主たる目的としている．一方，本研究は，検証プロセスの観点から複数手法を比較するものであり，検証プロセスの各ステップで習得しておくべき実践的ノウハウを，汎用的に利用可能なものとツール固有のものとを体系化し顕在化している．実践的ノウハウは，ツール性能に関連するものに限らず，検証作業の効率化に関するものも含んでいる．

4.1 節で述べた，実践的ノウハウや関連技術に関する既存研究との関連性を説明する．(ミス) ユースケースから検査仕様を作成する手法[21)] は，T11 での検証範囲の決定に活用できる．また，LTSA による検証実行時に検証仮定を自動生成する手法[10)] を利用すれば，すでに実行済みの検証結果を再利用して，T11 と T23 で検証仮定を効率良く作成できる．一般に，LTL 式の作成には数理論理学的な素養が求められるが，UML 図から LTL 式の自動生成する手法[19),20)] が T12 で利用できる．抽象化技術に関しては多数の研究報告[6)] があるが，主に T13 と T22 での利用が有効である．UML 図からモデル検査ツール言語へモデル変換を行う手法[19)] は，その手法がサポートしている UML セマンティクスと，開発者が前提としている UML セマンティクスとが一致すれば，T21，T23，T31 で利用できる．

多くのモデル検査ツールは，検証実行の最適化オプションを備えている．T51 の検証実行時に単にオプションを指定すれば最適化機能を利用できるものがあれば，一方でオプション利用のために MCSM や MCD の作成にノウハウが必要なものもある．Ruys [18)] は SPIN に関する多数のノウハウを収集しており，特に SPIN の検証実行時間と使用メモリ量の削減を目的としたもの（たとえば，変数のビットベクトル化，モデルの再順序）が多い．これらの手法を活用するには，T31 で MCD を作成する際に，使用する最適化手法に応じた記述を行う必要がある．半順序手法[12)] は，アクションの実行順序の半順序関係を利用して，検証実行時に探索する状態空間を削減する．Chan ら[4)] は，記号モデル検査手法で利用されている BDD と呼ばれるデータ構造の最適化手法[4)] を示しており，変数のビット表現への変換や，乗算表現から加算表現への変換などが利用できる．また，4.2 節の脚注で述べた，有限状態オートマトンから LTL 式への変換手法[9)] は，LTSA による検証プロセスの T32 で利用できる．限定モデル検査手法[2)] は，BDD を利用しない記号モデル検査手法で，探索する状態系列の長さをあらかじめ制限し，少なくとも検証処理の早い段階で現れる誤りを検出するものである．Cadence SMV では，T51 でオプションとして限定モデル検査ツールを利用できる．

6. おわりに

本研究では，モデル駆動型のソフトウェア設計検証プロセスを提案し，検証に必要な作業内容と手順とを詳細に定義した．従来の検証は熟練者の経験に依存していたが，検証プロセスを定義したことにより実践的ノウハウを顕在化でき，その効果としてデバッグプロセスを定義することができた．その結果，検証のノウハウを一般の開発者でも利用することが可能になった．さらに，設計誤りの情報を含んだ検証結果をソフトウェア開発プロセスへフィードバックすることができる．3 つのモデル検査ツール，SPIN，SMV，LTSA を適用した検証事例を通して，提案した検証プロセスの有効性を評価した．

今後の研究課題として，検証モデル要素間の変換手続きの形式化に取り組む．検証ノウハウには，モデル検査ツールに共通のものと，ツールに固有のものとが存在するが，共通のノウハウであっても，その具体的手法はツールごとに異なるものも存在している．その意味で，汎用的検証プロセスの上で，検証モデル間の変換ルールを形式化することは容易ではなく，将来課題として取り組んでいく．また，大規模設計へ適用する際に生じる状態爆発問題に対して，抽象化，最適化などの既存技術を具体的に利用するステップをプロセスの中に組み込んでいるが，プロセスを利用した工学的アプローチによる解決については今後掘り下げて検討していく．さらに，これらの課題への取り組みを通して，検証プロセスを洗練化していく．

謝辞 本研究は，文部科学省科学技術振興調整費新興分野人材養成基盤的ソフトウェア「産学融合先端ソフトウェア技術者養成拠点の形成」の一環として実施したものである．

参考文献

1) Avrunin, G.S., Corbett, J.C. and Dwyer, M.B.: Benchmarking Finite-State Verifiers, *Software Tools for Technology Transfer*, Vol.2, No.4, pp.317–320 (2000).
2) Biere, A., Cimatti, A., Clarke, E. and Zhu, Y.: Symbolic Model Checking without BDDs, *Proc. TACAS '99*, pp.193–207 (1999).
3) Chamillard, A.T., Clarke, L.A. and Avrunin, G.S.: An Empirical Comparison of Static Con-

currency Analysis Techniques, Technical report, Dept. of Computer Science, U. of Massachusetts (1999).
4) Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D. and Reese, J.: Model Checking Large Software Specifications, *IEEE TSE*, Vol.24, No.7, pp.498–520 (1998).
5) Chen, J. and Cui, H.: Translation from Adapted UML to Promela for CORBA-Based Applications, *Proc. SPIN 2004*, pp.234–251 (2004).
6) Clarke, E.M., Grumberg, O. and Long, D.E.: Model Checking and Abstraction, *ACM TOPLAS*, Vol.16, No.5, pp.1512–1542 (1994).
7) Clarke, E.M. and Heinle, W.: Modular translation of Statecharts to SMV, Technical report, CMU (2000).
8) Dong, Y., Du, X., Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Sokolsky, O., Stark, E.W. and Warren, D.S.: Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools, *TACAS '99*, pp.74–88 (1999).
9) Giannakopoulou, D. and Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs, *Proc. ASE 2001*, pp.412–416 (2001).
10) Giannakopoulou, D., Păsăreanu, C.S. and Barringer, H.: Assumption Generation for Software Component Verification, *Proc. ASE 2002*, pp.3–12 (2002).
11) Gluch, D.P. and Brockway, J.: An Introduction to Software Engineering Practices Using Model-Based Verification, Technical report, CMU (1999).
12) Godefroid, P. and Wolper, P.: A Partial Approach to Model Checking, *LICS '91*, pp.406–415 (1991).
13) Holzmann, G.J.: *The SPIN model checker: Primer and reference manual*, Addison Wesley (2004).
14) Magee, J. and Kramer, J.: *Concurrency: State Models & Java Programs*, John Wiley & Sons (1999).
15) McMillan, K.L.: *Symbolic Model Checking*, Kluwer Academic Publishers (1993).
16) Myers, K., Dionne, K., Cruz, J., Vijay, V., Dunlap, S. and Gluch, D.P.: The Practical Use of Model Checking in Software Development, *Proc. IEEE SoutheastCon 2002*, pp.21–27 (2002).
17) Păsăreanu, C.S., Dwyer, M.B. and Huth, M.: Assume-Guarantee Model Checking of Software: A Comparative Case Study, *Proc. SPIN '99*, pp.168–183 (1999).
18) Ruys, T.C.: Towards Effective Model Checking, Ph.D. Thesis, U. Twente (2001).
19) Schäfer, T., Knapp, A. and Merz, S.: Model Checking UML State Machines and Collaborations, *Electronic Notes in Theoretical Computer Science*, Vol.55, No.3 (2001).
20) van Lamsweerde, A. and Willemet, L.: Inferring Declarative Requirements Specifications from Operational Scenarios, *IEEE TSE*, Vol.24, No.12, pp.1089–1114 (1998).
21) Wood, D. and Reis, J.: Use Case Derived Test Cases, StickyMinds.com (2004). http://www.stickyminds.com/getfile.asp?ot=XML&id=2021&fn=XDD2021filelistfilename1%2Edoc
22) Xie, F., Levin, V. and Browne, J.C.: Integrating Model Checking into Object-oriented Software Development Processes, Technical report, UTCS (2001).

(平成 16 年 12 月 27 日受付)
(平成 17 年 10 月 11 日採録)

長野 伸一（正会員）

1971 年生．1996 年大阪大学大学院基礎工学研究科物理系専攻博士前期課程修了．1999 年同大学大学院基礎工学研究科情報数理系専攻博士後期課程修了．博士（工学）．同年（株）東芝入社．現在，同社研究開発センター知識メディアラボラトリー所属．2004 年より国立情報学研究所特任講師を兼任．主に，ソフトウェア工学，エージェント技術の研究に従事．電子情報通信学会，IEEE CS 各会員．

吉岡 信和（正会員）

1993 年富山大学工学部電子情報工学科卒業．1995 年北陸先端科学技術大学院大学情報科学研究科博士前期課程修了．1998 年同大学院大学情報科学研究科博士後期課程修了．博士（情報科学）．同年（株）東芝入社．2002 年より国立情報学研究所に勤務，2004 年より同研究所特任助教授，主にエージェント技術，およびソフトウェア工学の研究に従事．現在に至る．日本ソフトウェア科学会会員．

田原　康之（正会員）

1966 年生．1991 年東京大学大学院理学系研究科数学専攻修士課程修了．同年（株）東芝入社．1993〜1996 年情報処理振興事業協会に出向．1996〜1997 年英国 City 大学客員研究員．1997〜1998 年英国 Imperial College 客員研究員．2003 年より国立情報学研究所に勤務．2004 年より同研究所特任助教授．博士（情報科学）（早大）．エージェント技術，およびソフトウェア工学等の研究に従事．日本ソフトウェア科学会会員．

本位田真一（正会員）

1953 年生．1976 年早稲田大学理工学部電気工学科卒業．1978 年同大学大学院理工学研究科電気工学専攻修士課程修了．（株）東芝を経て 2000 年より文部科学省国立情報学研究所．現在，研究主幹・教授，2001 年より東京大学大学院情報理工学系研究科教授を併任，現在に至る．2002 年 5 月〜2003 年 1 月英国 UCL ならびに Imperial College 客員研究員（文部科学省在外研究員）．工学博士（早大）．1986 年度情報処理学会論文賞受賞．エージェント技術，オブジェクト指向技術，ソフトウェア工学の研究に従事．IEEE，ACM，日本ソフトウェア科学会等各会員．本学会理事．