2005－SE－150（2）
2005／11／29

モデル検査によるアーキテクチャ設計検証

岸知二†　　野田夏子††

アーキテクチャ設計にモデル検査技術を適用するに際しては、設計モデルと検証モデルの間に厳密性や詳細度にミスマッチが生じるなど、様々な適用上の課題が存在する。本稿ではソフトウェアアーキテクチャ分野の成果を踏まえて、アーキテクチャ設計検証に対するモデル検査技術適用の手法を提案するとともに、実際の組込みソフトウェアの設計検証に適用した事例を紹介する。

Architectural Design Verification utilizing Model Checking Techiniques

Tomoji KISHI† and Natsuko NODA††

In applying model checking techniques to software architectural design verification, there occur some problems, such as mismatch between design model and verification model in its strictness and preciseness. In this paper, we propose a method for applying model checking techniques to architectural design verification, in which we based on results from software architecture field. We also introduce a case study, in which we apply the method to actual embedded software design verification.

1. はじめに

組込みソフトウェアは急速に大規模化・複雑化する一方で、その開発期間は短くなっている。また社会のあらゆるところに組込みソフトウェアが使われるようになっているため、その信頼性に対する要求は一層厳しくなっている。そうした状況の中、現実に組み込みソフトウェアの不具合による製品のリコールなども起こっており、設計品質の向上は組込みソフトウェア開発における重要な課題となっている。従来組込みソフトウェアは熟練者のスキルに依存した開発形態がとられることも多かったが、上述した背景の中、開発形態の改善が求められており、ソフトウェア工学やソフトウェア科学などの成果をその開発に適用することが検討され始めている。

われわれは設計品質の向上をめざし、科学的手法の適用について研究を進めている。具体的にはUMLで記述された設計に対してモデル検査技術を適用し、設計検証を行うことを検討し、そのための支援ツールや事例研究などを進めてきた[9),10),12)]。今までの研究の中で設計検証の有効性が確認されてきた一方、設計という上流工程に対してモデル検査のような精密な技術を適用することの困難さについても実感してきた[11)]。産業界でモデル検査のような科学的手法を適用するためには、体系だった適用手法の提示が不可欠であり、設計検証に対する適用の考え方を整理することが重要であると考えている。

本稿では、ソフトウェア設計にモデル検査技術を適用する際のこうした困難な問題を指摘し、それに対する適用の考え方を提示する。なお一般にソフトウェア設計といっても、上位のアーキテクチャ設計から、コードに近い詳細設計まで様々なものが含まれる。われわれは作業の手戻りなどの多くが上位のアーキテクチャ設計に起因することを考え、そこに議論をフォーカスする。ソフトウェアアーキテクチャの研究分野では、アーキテクチャ設計において必要な視点や設計手法などが提案されているが、こうしたソフトウェアアーキテクチャ研究の成果を踏まえることにより、モデル検査技術の妥当な適用の方針が明確になると考えている。

2章では、アーキテクチャ設計の特徴を、ソフトウェアアーキテクチャ研究の立場から概観する。3章では、アーキテクチャ設計の検証にモデル検査技術のような技術を適用する際に考えられる問題について指摘する。4章では、アーキテクチャ設計の特徴を踏まえた、モデル検査技術による設計検証のアプローチについて提案する。5章では、本アプローチに基づいたアーキテ

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

クチャ設計検証の事例を示すとともに、その有効性と課題について整理する。6 章では、本稿での提案についていくつかの観点から検討する。

2. アーキテクチャ設計

本章ではアーキテクチャ設計の特徴について、ソフトウェアアーキテクチャ研究の立場から概観する。

2.1 アーキテクチャ設計とは

ソフトウェアアーキテクチャに関しては様々な定義が存在するが、例えば Garlan/Perry の定義[5] や、ANSI/IEEE の定義[8] を踏まえ、ここでは「ソフトウェアおよびその開発を支配するソフトウェア構造や構造化の原則」と定義する。

ソフトウェアアーキテクチャの重要性は、それがソフトウェアの諸特性を決定付ける重要な要因であるという点や、その決定が作業分担や作業の手順など様々な開発に多大な影響を持つ点にある。現実規模のソフトウエアを開発する際には、あらゆるソフトウェア構造上の選択肢を残したまま開発を行うことは不可能である。仮にインクリメンタルな開発を行うにしても、その中で随時構造上の判断を行いながら開発を進めなければならず、その判断が間違っていれば手戻り等の問題に結びつく。ソフトウェアアーキテクチャとは、こうした重要性を持った構造を指す。一方品質特性に対する影響が軽微であったり、局所的にその修正が可能なソフトウェア構造はソフトウェアアーキテクチャとは呼ばれない。

またソフトウェアアーキテクチャは具体的なソフトウェア構造を指すだけでなく、構造決定に対する様々な原則をも含む。すなわちアーキテクチャ設計に基づき、詳細な設計へと進行する際に守らなければならない様々な規則や約束事もソフトウェアアーキテクチャに含めるということである。Garlan は再利用を阻害する理由としてのアーキテクチャ不整合という概念を提案しているが[6]、ここではひとつの構造上の決定の背後には様々なアーキテクチャの想定が存在し、それを踏まえなければ正しいソフトウェア設計へとはつながらないことが指摘されている。

アーキテクチャ設計とは、こうしたソフトウェアアーキテクチャの設計を意味する。アーキテクチャ設計の手法の提案は複数存在するが、基本的には上述したような骨格構造へのフォーカス、ビューに応じた様々な品質特性の検討、各種評価技術を組み合わせたアーキテクチャ評価などが重要なポイントとなる[2]。

2.2 アーキテクチャ設計の特徴

一般にアーキテクチャ設計は、詳細設計などと比較し、以下のような特徴を持つ。

- アーキテクチャ上の重要性に照らした骨格構造の設計：
  前述したようにアーキテクチャの本質は品質特性や開発における重要性という観点である。何がアーキテクチャ上の重要性となるかはドメインや開発形態によって異なるが、例えば採用する方式の妥当性、想定される負荷に対する性能、重大なエラー時の挙動など、いろいろなものが考えられる。
  ここで大切なことは、アーキテクチャ設計は、こうしたアーキテクチャ上の重要性に照らして目的指向でなされるということである。すなわち検討すべき問題を明示的に意識し、その問題を捉えやすい視点から対象を捉え、それをアーキテクチャ設計として定義する点にある。
  そのため結果としてのアーキテクチャ設計は、目的視点から抽象化され、その目的に応じた必要十分な範囲が記述されることになる。こうした点は、開発部分の全情報を記述するコードや、ほぼそれに近い詳細設計とは本質的に異なる。
- シナリオを活用した評価：
  アーキテクチャの本質から考えると、アーキテクチャ検討においては、詳細な仕様に照らした評価などは適切ではない。しかしながら漠然とした機能や品質特性を議論しても適切な評価はできない。そのため、シナリオを活用した評価を行うことが多い[1]。
  シナリオにはいくつかの使われ方があるが、個別の詳細な機能を示すのではなく、ユースケースのようにより一般的な要求を示すために使われることがある。また、品質特性への要求を議論する際に、その要求を具体的に特徴づけるために使われることがある。
  仕様に照らした網羅的な評価・確認ではなく、主要なシナリオに照らした評価を行う点も、アーキテクチャ設計における特徴である。
- アーキテクチャ上の想定に対する考慮：
  前述したようにアーキテクチャ設計においては、その背後に置かれた様々な想定を意識する必要がある。Garlan はそうした想定として例えば制御モデル、データモデル、基盤、構築プロセスなどに対する想定を指摘している[6]。あるいは Trew は、アーキテクチャ検証において、そのソフトウェア構造がおかれる文脈への配慮が重要であるとし、その例として他のタスクとの優先度の関係、起こ

りうる割り込み処理、共有リソースに対するアクセスのポリシーなどを指摘している[15]。

コードや詳細設計の段階においては、システムの詳細や稼動するプラットフォームなどが具体的に決まっているため、そうした具体的な環境を前提に検討を進めるが、アーキテクチャ設計やその評価においてはこうしたアーキテクチャ上の想定に対する扱いに注意が必要となる。

なお、ここで重要なことは、コードや詳細設計において確認できる事は、そうした特定のシステム詳細やプラットフォームに依存したシステムの機能や品質特性である点である。それに対してアーキテクチャ設計では、そのアーキテクチャが受け止めることのできる広がり、すなわちアーキテクチャ上で実現されうる様々な機能群や、そのアーキテクチャが機能すべき様々なコンテキストを考え、それらすべてに対して妥当なアーキテクチャであるかどうかを検討・評価することが必要となる。そういう意味でコードや詳細設計においては、アーキテクチャの妥当性を正しく評価することは困難であり、そこにアーキテクチャ設計・評価の重要な意義がある。

3. モデル検査とアーキテクチャ設計検証

本章では、アーキテクチャ設計の検証にモデル検査技術のような精密で厳密な技術を適用する際に考えられる問題について指摘する。

3.1 モデル検査技術

モデル検査技術は、有限状態モデルと論理的な性質が与えられると、そのモデル上でその性質が成立するかどうかを自動的に検査する技術である[3]。

ソフトウェア上のバグを発見するための技術としては、レビューやテストなどいくつかの方法がある。こうした既存の検証技術はそれぞれ利点を持っているが、様々な状況をもれなく考慮した網羅的な確認を行うためには十分ではない。

例えばテストは限られたテストケースに沿ってなされるため、それによってバグの不在を証明することは不可能である。一方モデル検査技術は、対象の網羅検査を行う技術であるため、うまく適用できればバグの不在を確認することが可能であるし、それができなくてもテスト等に比べてはるかに信頼度高く問題の存在を検査することができる。また、並行に動作する複数のタスクの動作順序を網羅的に考慮して、どのような動作順序でも間違いなく動作することを確認することはテスト等の既存技術では困難であったが、モデル検査技術ではそれらのあらゆる動作順序の組み合わせを網羅検査することも可能となる。

モデル検査ツールの発展もあり、近年モデル検査技術のソフトウェア検証への適用がいろいろと検討されているが、様々な外界からのイベントに対して長期間間違いなく動作しなければならない組込みソフトウェアにおいては、上述したような網羅的な検証が適していると考えられ、有効な活用が期待されている。

3.2 アーキテクチャ設計の検証に適用する際の課題

前節で指摘したように、ソフトウェア検証に対してモデル検査技術を適用することは有益であると考えられ、特に組込みソフトウェアの検証には親和性が良いと考えられる。しかしながら、品質や生産性に対して大きな影響を及ぼすアーキテクチャ設計の検証に対してモデル検査技術を適用することには、以下のような課題が考えられる。

- 設計モデルの厳密性：
  モデル検査技術を適用するためには、対象を厳密にモデル化する必要がある。特にその動作意味については注意深い検討を行わないと、モデル検査の結果にナイーブな影響を及ぼす。一方、UML等で書かれる設計モデルは多くの場合人間が読むためのドキュメントとして記述されるため、厳密には記述されないことが多い。特にこうした動的意味に関しては典型的な状況を概略理解するために必要な程度の暗黙の想定を置いて読んだり、特定のプラットフォームを想定して理解するなどされることが多い。
  しかしながらモデル検査技術を適用しようとすると、こうした側面を含め、より厳密で詳細なモデルを記述する必要が出てくる。システム全体をそうした厳密性と詳細度を持って記述することは現実的には困難である。通常用いられている設計記述と、モデル検査技術を適用するために必要とされる記述との厳密性や詳細度の違いをどう埋めるかが課題のひとつである。
- 検証項目の選定：
  モデル検査技術はソフトウェア検証に必要なすべてのカテゴリの検証に適しているわけではなく、例えば実時間の扱いや、データの連続値の扱いなど不得手な分野が存在するが、モデル検査によって検証可能な項目だけを考えても、様々な機能や状況の確認、様々なインスタンス構造に対する確認など、検証できる事柄はいくらでも列挙することができる。これらの検証可能な項目の中から、設計検証の段階でいったい何を検証することが有

効なのか、検証項目の選定も設計検証における課題のひとつである。

- 網羅検査適用の妥当性：
  モデル検査技術に期待される特徴のひとつは、様々な状況に対する網羅検査が可能な点である。しかしながらアーキテクチャレビューと詳細設計レビューの違いを考えれば明らかなように、一般にアーキテクチャ設計では詳細設計に比べて、より発見的な検証がなされることが通常である。こうした特性を持つアーキテクチャ検証に網羅検査ができるというモデル検査技術の特徴がどのように活かされるのかを考える必要がある。モデル検査技術などの科学的手法の適用はそれなりのコストがかかるため、その技術の特徴を有効に活かせないならば、適用することそのものの妥当性を考える必要がある。

4. アプローチ

本章では、前章で指摘した問題を踏まえながら、アーキテクチャ設計検証に対してモデル検査技術を適用するための基本的な考え方を整理し、それを踏まえたアーキテクチャ検証手法を提案する。

4.1 基本的な考え方

前章で指摘したようにアーキテクチャ設計の検証にモデル検査技術を適用することにはいくつかの課題があるが、2.2 に述べたアーキテクチャ設計の特徴に立ち戻ると、以下のように考えることで適用の方向性が見えてくると考えられる。

- アーキテクチャ上の重要性に関わる観点の抽象化：
  アーキテクチャ設計では、アーキテクチャ上の重要性に照らして、その重要な側面の確認に必要な視点から対象を捉える。またアーキテクチャの記述範囲も、その目的にとって必要な部分に限定して設定される。例えば処理方式の確認であれば、その処理方式に関わる側面のみに注目し、それに関わらない部分や細部は捨象される。すなわち、アーキテクチャ上の重要性という観点からソフトウェア構造を抽象化することが本質となる。
  したがって現実規模の問題であっても、個々の検証を行う際には、その検証に関わるアーキテクチャ上の重要性を吟味し、それを捉える視点を明確にし、必要十分な範囲のみ厳密なモデル化をすればよいことになる。これにより、無目的にシステム全体を詳細かつ厳密に記述するなどといった問題を回避することが期待される。
- 重要なシナリオに即した検証項目の選定：
  検証に際しては、やみくもに検証を行うのではなく、シナリオに照らした検証項目を選ぶことで、アーキテクチャ検証として意味のある検証が可能となる。例えば数多くの機能があったとしても、方式確認上は類似した機能の確認をすることは大きな意味はない。むしろ起動、終了、エラー発生時など、重要なシナリオを吟味し、それらに照らした検証を選別的に行うことが妥当であると考えられる。
  なおこの検証項目の選定は、前項の抽象化と関連している。一般に検証項目は複数存在し、また検証項目に応じた抽象化が必要となり得る。
- 考慮すべきアーキテクチャ上の想定に関わる網羅的な検証：
  アーキテクチャ設計では、最終的に実現すべき個々の機能をひとつひとつ捉えることはせず、アーキテクチャ上の重要性を考えたときに本質的でない部分は捨象される。こうした意味で、アーキテクチャ設計は一般的には実際の設計よりも抽象度が高くなるが、一方アーキテクチャ上の想定や文脈に関しては逆により広範囲かつ慎重な検討が必要となる。なぜならばアーキテクチャはその上で開発されるソフトウェア群の多様性を受け止めるための骨格構造であるから、その多様性に関しては十分な考慮を行っておかなければならないからである。前述したように、詳細設計やコード段階では想定に含まれる特定のインスタンスに関する確認を行うことになるが、アーキテクチャ設計においては多様な想定に関して確認を行う必要がある。
  直感的な説明をすれば、アーキテクチャをフレームワークに例えると、そのフレームワークがたまたま特定のアプリケーション開発に使えたというだけではそのフレームワークの妥当性を主張することはできない。それが想定している様々なアプリケーション群に対して利用できることを確認しなければならない。
  モデル検査技術の持つ、網羅検査という特性は、こうしたアーキテクチャ上の想定や文脈の多様性に対して、妥当な性質を持っているかどうかを確認する際に活用することが有用であると考える。

上記の考え方は、モデル検査技術という科学的手法を適用する際にも、ソフトウェアアーキテクチャの研究分野で指摘されてきたソフトウェア工学的な視点を踏まえることによって、妥当な適用方法が見えてくるという立場にたつものである。

4.2 アーキテクチャ設計検証への適用手法の提案

前節の基本的な考え方を踏まえ、アーキテクチャ設計検証にモデル検査技術を適用する際の手法を概略的に説明する。

(1) 通常のアーキテクチャ設計モデルを作成する。従来のアーキテクチャ設計の流れの中で、UML等を活用したアーキテクチャ設計モデルを作成する。これは一般的には人間が読んで理解するためのモデルであるため、モデル検査技術などを適用するためにはその厳密性・詳細度ともに、不十分であるが、全体を把握するためには有効かつ必須である。

(2) アーキテクチャ上の重要性の中から、モデル検査での確認が有効である検証事項にフォーカスして、検証項目を選定する。典型的には、重要な処理の方式がアーキテクチャが想定する様々な状況の中でも機能することを確認することなどに有効であると考えられる。

(3) 上述した検証項目に関わる重要なアーキテクチャ上の想定や文脈を整理する。一般的には外界の多様性、制御モデル（スケジューリング、優先度、割り込み）、同時に動作する他のタスク等との関わり、共有リソースへのアクセスポリシーなどが考えられる。しかしながら潜在的に考えられる想定や文脈をすべて列挙することは無意味であり、現実的でもない。考慮すべき想定や文脈はアーキテクチャ上の重要性同様、経験に基づいて吟味されるべきものである。例えばTrewらは過去の障害レポートを分析することでこれらを洗い出すなどしている[15]。

(4) 選定された検証項目ごとに、その検証を行うために必要な視点、抽象度、記述範囲を明確にし、検証目的のアーキテクチャモデルを記述する。この際、記述の範囲はその検証に関わる部分にフォーカスし不要な部分を捨象することが必要である。一方で、関わる想定や文脈は、どの範囲の想定や文脈を扱うかを明確にし、その範囲での妥当性確認ができるようにモデルに反映させる必要がある。例えばより優先度の高いタスクが同時に動作していても良いという想定であれば、優先度の高いタスクが様々な状況で実行権をとるようなモデルを作成し、それでも確認すべき方式が機能するかどうかを確認する必要がある。

なおモデル検査では無限の概念を扱えないため、例えばUML上で多重度がNといった場合、具体的にどのような値に対してそれを確認すべきか等、特有の配慮が必要となる。

この記述はあくまで設計者の視点で行われるべきであることから、ソフトウェア技術者にとって理解しやすい記法を用いるべきと考える。我々はモデル検査に必要な情報を補足できるように拡張したUML記述を用いることが妥当であると考えている。

(5) 記述された検証目的のアーキテクチャモデルをモデル検査の世界にマッピングし、モデル検査技術によりその妥当性を確認する。その結果はアーキテクチャ設計に反映しながら、妥当なアーキテクチャを検討する。

図1に以上のステップを図示する。

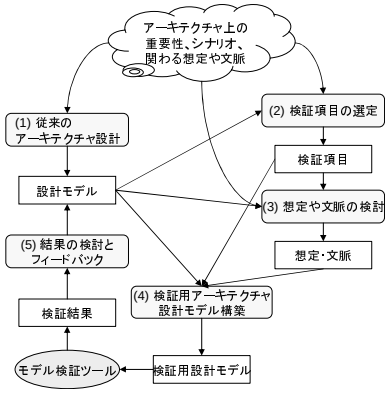

図 1 モデル検査技術適用のステップ

4.3 UML 検証ツール

前章のステップ(4)で構築する検証用設計モデルは、アーキテクチャ設計を行うソフトウェア設計者の視点で記述されることが望ましい。そこでも触れたように、われわれはUML記述を拡張し、ソフトウェア技術者が検証のための設計モデルを構築しやすいように支援することを検討し、支援ツールを開発してきた[9),10)]。本ツールは、Eclipseプラットフォーム[4)]の上で開発され、UMLの記述にはUMLプラグイン[13)]を、検証エンジンにはSPIN[7),14)]を用いている。

このツールは拡張された記法に基づくUMLモデルと、UML中で定義されている概念を参照したLTL式を与えると、それをPromela言語と、Promela言語中の概念を参照したLTL式に変換した上でSPINによりモデル検査を実行し、その結果を再度UMLの概念に再変換して表示する機能を持つ。

なお本ツールの紹介をすることは本稿の主旨ではないため、その詳細はここでは触れない。

5. 事　　例

本章では、前章で紹介した手法を踏まえて、アーキテクチャ設計検証を行った事例について紹介する。

5.1 対象システム

本事例は、企業より提供いただいたカーオーディオシステムの事例に基づき実際にアーキテクチャ設計検証を行ったものである。本システムはリファレンスセットとして開発されており、実際の分析・設計にはUML を、実装には C 言語を用い、32bitCPU の上でμ ITRON を用いて開発されている。

本事例では、本システムの仕様をコンパクトなものに限定し、実際の設計をベースにその仕様の範囲で再設計を行いながら、アーキテクチャ設計の妥当性をモデル検査技術を用いて確認したものである。なお、以下の検証に先立ち、前述のステップ (1) に相当する作業として、設計者が全体を俯瞰するために全体のUML モデルを作成した。このモデルは全体のクラス図と、主要なシナリオに相当するシーケンス図が含まれている。

なお検証には前述した UML 検証ツールを用いた。

5.2 方式設計の妥当性確認

本システムでは、入力されたボタン操作がどういう処理に対応するかは、その時点でどのソース (CD、チューナ) が設定されているかに依存して決定される。

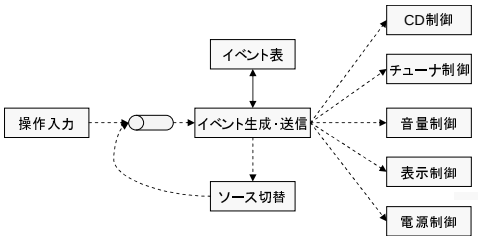

図 2　イベント処理の方式案 1

図 2 は、この処理を行うための方式案のひとつである。ここでは「イベント生成・送信」が「操作入力」からの操作指示を受け取るとソース状態を管理する「ソース切替」に現在ソースを確認した後、「イベント表」を参照して対応する処理（各種制御あるいは「ソース切替」に対する処理の指示群）を判断し、必要な制御や「ソース切替」に処理指示を送信するという方式になっている。この際、「ソース切替」は処理指示を受け取ると、その処理のために、さらに「イベント生成・送信」に対して指示を送ることがあり得る点がポイントである。

この方式の妥当性を検討するにあたり、検証手法に従い、以下の手順で検証を行った。

(1)　上述したように全体の UML モデルを作成。

(2)　検証項目として、本方式がどのような操作入力に対しても、正しくそれを処理できるかどうかを確認することを設定した。具体的には、どのような操作入力に対しても処理が停止しないこと、「ソース切替」から「イベント生成・送信」への処理の依頼が行われる合間に「操作入力」から次の操作指示が来ても、処理の前後関係がくずれることがないこと、を調べる。

(3)　考慮する想定としては、「操作入力」「イベント生成・送信」「ソース切替」各種制御が、並行して動作しどのようなスケジューリングに対しても対応できることを考える。

(4)　上記を踏まえ、以下の方針で検証用のモデルを作成する。

- 方式に関わるイベントの伝達という観点だけに注目し、他の処理は省略する。
- 重要なシナリオとして起動、終了、ソース切替などを取り上げる。CD 再生、停止、チューナ再生等の処理は、本検証目的からするといずれも類似性の高い処理なので、独立したシナリオとしては扱わない。
- ソースの制御に関わらない「音量制御」「表示制御」「電源制御」も省略する。
- それぞれのモジュールは並行に動作し、スケジューリングは特段に行わない (スイッチできる箇所にくればどのモジュールにもスケジュールされる可能性を考慮する)。
- イベントの送受信は非同期に行う。

図 3 に検証用の UML モデルのクラス図、図 4 に検証用の UML モデルのステート図をそれぞれ示す。

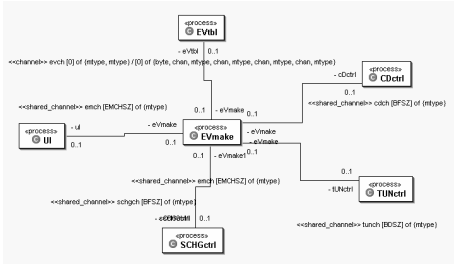

図 3　検証用 UML モデルの例 (クラス図)

(5)　実際に検証を行うと、ある状況で本方式は停

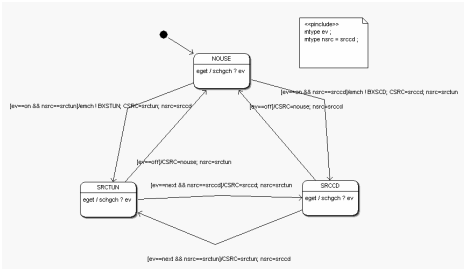

図 4　検証用 UML モデルの例 (ステート図)

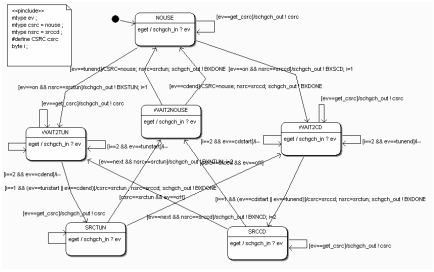

図 6　「イベント生成・送信」のステート図

止してしまうことが確認された。具体的には、「操作入力」からの操作指示を処理する途中で「ソース切替」が「イベント生成・送信」に指示を送ろうとする前に、「操作入力」から次の操作指示が複数到着してバッファがフルとなり、「ソース切替」がブロックしてしまう状況があることが確認できた。

上記の検証のサイクルで最初の方式案に問題があることが確認された。そこで方式を見直し、「操作入力」からの操作入力のためのバッファと、「イベント生成・送信」からの指示のためのバッファを別に設ける方式を検討し、ほぼ上記と同様のステップで検証用モデルを作成し検証を行った。

しかしながらこの方式も、ある状況で処理が停止してしまうことが検証により確認された。図 5 は、この状況を UML 検証ツールの機能でシーケンス図として表示したものである。この問題は、各種制御の処理の終了等を「ソース切替」が確認をしていなかったため、管理している現在ソースが、実態とずれてしまうタイミングがあることに起因している。

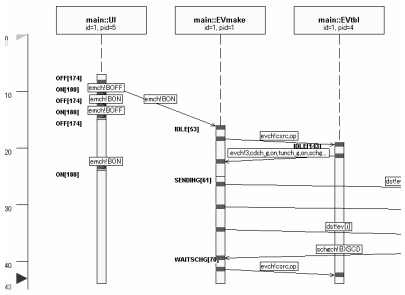

図 5　反例のシーケンス図による表示

この検証サイクルの結果を踏まえ、さらに方式を見直し、各種制御の終了等をソース切替が確認をするように見直した。図 6 は見直した結果の「イベント生成・送信」のステート図である。

この方式は検証によって、様々な操作入力シーケンスを与えても停止することがないことが確認された。さらに操作入力のイベントにタイムスタンプをつけ、各種制御が結果として受け取ったタイムスタンプの順序を確認することにより、イベントの前後関係がくずれることがないことも検証され、アーキテクチャ設計の段階で本方式の妥当性について一定の確認を行うことができた。

5.3　複数の処理スレッド間の競合の確認

「操作入力」からの操作指示は、イベントテーブルの内容や、「ソース切替」からの指示に従い、複数の様々な処理へと展開される。またこれらの処理が複数のリソースを扱うため、複数の操作が輻輳するとそれらが共有リソースへの競合問題などを起こす危険性がある。

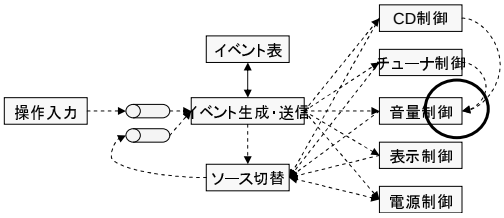

図 7　音量制御への処理の競合

図 7 は、本システムにおける競合の可能性を示したものである。「CD 制御」「チューナ制御」」いずれもイベントを受けると「音量制御」へイベントを送る処理が含まれている。「CD 制御」と「チューナ制御」に輻輳した処理の指示が送られると、場合によっては双方から「音量制御」に対して異なった処理依頼が届き、リソースに対して相反する処理を不適切な手順で行ってしまうなどの危険性がある。

ここでは SPIN の assertion の機能を利用してこうした誤ったアクセスがないかどうかを監視するようにし、いくつかの操作指示を与えながら確認を行った。その結果、今回のモデル化の範囲ではそうした危険性

がないことが確認された。

一方、表示の更新等は割り込みによってなされる場合があり、そうした割り込み処理が他の処理スレッドとリソース競合を起こす危険性がないかを確認するために、独立した処理スレッド (SPIN の process) をつくり、それがリソースをアクセスするようにし、上記同様 assertion を使って競合アクセスを監視した。これに関しては競合が起こることが確認され、排他制御のメカニズムをリソースに含める必要性が分かった。

6. 考　　察

以上、アーキテクチャ設計検証に対してモデル検査技術を適用する際のアプローチを提案するとともに、事例に基づいた設計検証の適用例の一部を示した。

本事例はあくまで適用手法のデモンストレーションを行ってみたにすぎないが、今回の範囲においては、現実規模のソフトウェアであっても、検証目的にフォーカスして検証用の設計モデルを構築することで、比較的少ないコストで検証確認を行うことができアーキテクチャ上の問題点を見つけることができた。方式検討において見つかった最初の問題はひとつの操作指示に起因するひとつの制御スレッドだけを追いかけていては見つからない問題であり、また次の問題は並行に動作している複数の process が、特定の順序でスケジューリングされた場合にのみ発生する問題であり、いずれもモデル検査の特徴を活かした検証結果を得ることができたといえる。

もちろん今回の事例は試行的なものであり、アーキテクチャ設計の段階でどれだけの検証項目を検証することが妥当なのか、また検証項目の内容に応じてどのような視点から検証用のモデルを構築すればよいのか、あるいは考慮すべき想定や文脈としてどのようなものがあるか、など今後さらに検討を深める必要がある。

アーキテクチャ設計においては、品質特性面への要求を踏まえ、それらの品質特性が実行時構造、開発時構造等、様々な構造のどれに関わるかを明らかにし、評価をしつつ適切な構造検討を進める。モデル検査はこうした様々な品質特性や、構造の検討にひとしく使えるものではなく、特に実行時構造の検討に有効であると考えられる。

7. おわりに

本稿では、モデル検査技術をアーキテクチャ設計検証に適用するための手法について提案した。モデル検査技術のような科学的手法を適用するにしても、検証する対象はあくまでソフトウェアであり、ソフトウェアあるいはソフトウェア開発にとって重要な問題を設計段階で検出するという、ソフトウェア工学的な考え方を踏まえなければ有効な検証はできないと考えられる。今後こうしたソフトウェア工学的な視点を踏まえながら、モデル検査技術のより有効な適用手法の検討を進めていきたい。

参 考 文 献

1) Bass, L., Clements, P. and Kazman, R.: Software Architecture in Practice *Second Edition*, Addison-Wesley, 2003.
2) Bosch, J.: Design and use of software architectures, Addison-Wesley, 2000.
3) E. Clarke, O. Grumberg, and D. Peled : Model Checking, MIT 1999.
4) http://www.eclipse.org/
5) Garlan, D. and Peryy, D.E.,: Introduction to the Special Issue on Software Architecture, IEEE Transaction on Software Engineering, Vol.21, No.4. April 1995.
6) Garlan, D., Allen, R., and Ockerbloom, J.: Architectural Mismatch: Why Reuse Is So Hard, IEEE Software, Nov. 1995, p17-26.
7) G.J., Holzmann, :The model checker SPIN, IEEE Trans. on Software Engineering, Vol. 23, No. 5, 1997, pp. 279-295.
8) IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std 1471-2000, 2000.
9) Kishi, T., et. al.: Project Report: High Reliable Object-Oriented Embedded Software Design, The 2nd IEEE Workshop on Software Technology for Embedded and Ubiquitous Computing Systems (WSTFEUS'04), 2004.
10) 岸知二, 他: 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, 2004.
11) 岸知二，野田夏子，片山卓也：アスペクト指向に基づく設計検証フレームワーク，日本ソフトウェア科学会 第 2 回ディペンダブルソフトウェア研究会，2005.
12) Kishi, T., Noda, N. and Katayama, T.: Design Verification for Product Line Development, Software Product Line Conference 2005 (SPLC Europe 2005)
13) http://www.eclipseuml.com/
14) http://spinroot.com/spin/whatispin.html
15) Trew, T.: Enabling the Smooth Integration of Core Assets: Defining and Packaging Architectural Rules for a Family of Embedded Products, Software Product Line Conference 2005 (SPLC Europe 2005)