RO-002

CPS のためのモデル検査・実行テスト統合試験環境の構築

Testing Environment for Cyber Physical Systems by cooperating Model Checking with Execution Testing

青山 裕介† 黒岩 丈瑠†‡ 久代 紀之†
Yusuke Aoyama Takeru Kuroiwa Noriyuki Kushiro

1. はじめに

Cyber Physical System (CPS) と呼ばれるような、大規模、広域の制御システムにおけるマルチベンダーでのシステム開発及びオープンプラットフォームを用いた機器開発が一般化している。 CPS は、社会・人・機械が共存するシステムであるため、高い信頼性が要求される。これらシステムでは、新機種の導入や新機能の追加などの度に膨大な試験ケースにより評価を行う必要があり、開発の工数を増大させている。一方、このようなシステムでは、複数機器・機能が独立して動作するため、次に示すような設計時にはどのように動作するのかを決定できない要素が存在してしまう。

1. 機器間の通信において、メッセージの交換順序が非決定的
2. 共通機器への同時アクセス時、資源を獲得する機器が非決定的
3. 機器単体での動作としても、同時に動作しうる機能間の実行順序が非決定的

このことから、膨大な評価を行ったとしても、必ずしも十分な信頼性を得ることができないという課題があった。

上記のような非決定的な要素を含むシステムを評価する技術としてモデル検査というものがある。モデル検査では、システム中の機器・機能といった並列動作するプロセスの実行順序組み合わせを網羅的にシミュレーションしながら、機器・機能が満たすべき性質を満足しなくなることがないかを確認することができる。しかし、モデル検査による評価には、次の２つの課題があった。

1. モデル検査で評価するための、実行テストとは別の試験ケースが必要
2. モデル検査が出力する試験結果からの不具合箇所の解析が困難

1 について、モデル検査では、モデル検査器の入力となる試験対象のシステムを、モデル検査用の専用言語により記述する必要がある。しかし、あくまでもモデル検査での試験はシミュレーションによるものであることから、実行テスト用の試験ケースとモデル検査用の試験ケースの 2 つの試験ケースを作成する必要が生じ、試験ケース作成、実施の工数を増加させていた。

２について、モデル検査による試験では、予め時相論理という形式でシステムが満たすべき条件を与えておくことで、網羅的なシミュレーション中に、与えた条件を違反することはないかを確認することができる。条件が満たされなかった時、モデル検査器は反例という形で条件に対する矛盾を提示する。反例により示される時相論理内の条件変数の値やシミュレーション時の実行順序が含まれており、これらの情報から、不具合箇所の解析が行われることが意図されていた。

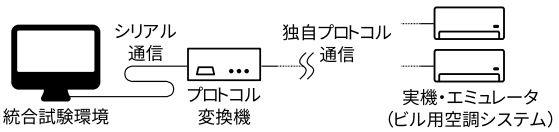

図 1 プロトタイプ版構成

しかし、実装済みシステムを評価するためにモデル検査を用いる場合、次の相反する課題から、モデル検査による不具合解析は困難なものであった。

1. 実装済みのシステムを改めてモデル検査用に記述することはコストが高い
   このため、モデル検査時には実装されたシステムの一部を記述し、試験を行う。
2. モデル検査専用の言語内に記述されていない機器・機能に起因する不具合は検出できない
   このため、限定された規模で試験した場合、不具合が検出されない可能性がある。

これまでも、実装済みのシステムを評価するためにモデル検査を用いる手法[1][6][7][8][9]が提案されている。しかし、これら手法は、モデル検査器の試験失敗時に出力する反例を用いて特定の状態に至る試験手順を得るものであり、どのような手順で操作しても機器・機能が期待する動作（例えば特定の要求に対する応答を返す）が行われるといった、機器・機能の動作順序の非決定性に基づく網羅的な試験を行うものではなかった。また、[4]のように、ソースコードからモデル検査用の専用言語による試験ケースを抽出し、モデル検査を行う手法では、1 の課題は解決できるが、関数呼び出しなど試験が行えない部分が存在した。

本研究では、モデル検査の非決定性の枠組みを実行テストに導入し、非決定的な手順を含んだ試験ケースによる評価の困難さを解決する。本研究においてモデル検査器は、実機の機能を網羅的な順序で呼び出すためのテストドライバとして動作させ、その動作を直接確認できるようにし、また、実機の動作ログからの解析が行えるようにすることで、上記の実装済みシステムに対するモデル検査適用の課題 1、2 を解決する。これに加えて、次の 3 つの機能を合わせた統合試験環境を開発し、CPS における大規模・広域の制御システムにおける評価の課題を解決する。

- 試験ケース時の機器・機能が並列動作する条件発見の支援機能
- 機器・機能の追加の度に必要な膨大な回帰テストの自動実行・自動評価機能

†九州工業大学大学院情報工学研究院情報創成工学専攻
Department of Creative Informatics Kyushu Institute of Technology
‡三菱電機（株）住環境研究開発センター Mitsubishi Electric Corporation Living Environment Systems Laboratory

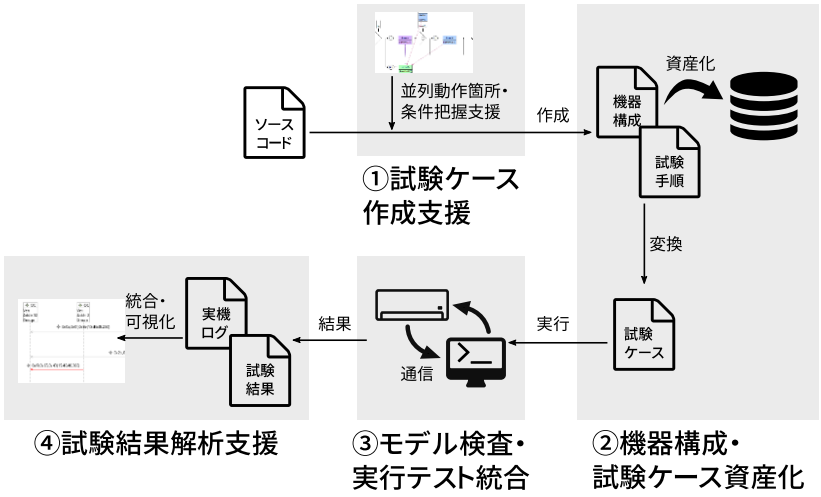

図 2 統合試験環境概要

- 回帰テスト用の試験ケース、及び市場で発見された不具合事例の再現可能な試験ケースの資産化機能

ただし、 CPS は前述の通り大規模システムとなるため、開発した統合試験環境を評価することが難しい。したがって、まず本統合試験環境のプロトタイプとして、ビル用空調管理システムを対象に、図 1 のような構成で開発を行った。ビル用空調管理は、次の通り、 CPS における制御システムの特徴を限定された規模であるが有しているため、CPS を対象とする統合試験環境のプロトタイプの対象として妥当である。

- 室内機・室外機などの複数機器から構成される
- 各機器のアップグレードにより機能追加が行われる
- 各機器が連携して動作するための通信機能を有する

まず、2 節にて統合試験環境について述べた後、3 節にて開発した統合試験環境のプロトタイプの一部機能に対する評価試験内容を述べる。4 節に評価試験の結果を述べた後、5 節にて試験結果を考察する。

2. 実現方法

本研究で開発する統合試験環境の構成は図 2 である。本研究では、CPS で要求される膨大な試験の作成・実行・評価を支援する。試験ケースを増加させているものは、1 節で述べたとおり、並列動作する機器・機能がどのように動作するのかが実行時に初めて決まる、非決定性である。本研究では、この非決定性を含む試験ケースを評価するために、従来、シミュレーションによる非決定性の試験を行っていたモデル検査を、実機に対して試験可能とすることで解決する（③）。

CPS のような大規模システムにおいてモデル検査を行う場合、状態爆発によりモデル検査が扱える状態数を上回ることがあるため、システムの一部分を切り出して試験する必要がある。この結果、試験結果から切りだされた範囲外でのシステムの状態、動作が不具合の原因であった場合、モデル検査による結果出力では不具合の解析が困難になる。これを ④ で解決する。

モデル検査による非決定性の評価が可能となったとしても、 CPS のような広域のシステムでは多数の機器組み合わせがあり、また、継続的な新機種の導入により、その組み合わせは増加する。新機種が導入されるたびに、その機器組み合わせを網羅する試験ケースを作成することはコストが高い。これに対し、試験対象の機器に依存しない試験ケースの記述方法を開発し、試験ケースを資産化することで、試験ケースの再利用性を向上させ、解決する（ ② ）。

モデル検査による試験ケースの作成においても課題がある。モデル検査用の専用言語により試験ケースを記述すれば、非決定性を含む試験ケースを網羅的に試験することができる。しかし、非決定性を含む箇所を特定することは、膨大な機器・機能から困難である。この課題を解決するために、[10]のソースコード中の制御構造可視化ツールを本統合試験環境に統合した（①）。

本論文で新たに実装を行ったのは図 2 中の ② 、③ 、④ である。それぞれ以降に述べる。

2.1 試験ケースの資産化

本節では、図 2 中の ② に相当する、試験ケースの資産化機能について述べる。

試験ケースは、どのような試験を行うかという試験手順と、どの機器・機能を対象に試験を行うのかという機器構成により構成される。しかし、従来では、機器構成・試験手順を分けることなく作成されていたため、試験手順・機器構成どちらか一方のみしか変わらない場合でも、再利用ができず、別の試験ケースとして作成しなければならなかった。本研究の対象とする CPS では特に、1 節で述べたように、非決定的な要素を網羅する膨大な試験ケースを作成する必要が有るため、新機種・機能の追加の度に、新たに試験ケースを作成・実行することはコストが高い。この課題に対し、本研究では図 4 のように、試験手順・機器構成を個別に資産化可能にし、資産化した試験手順・機器構成の組み合わせて新しい試験ケースインスタンスを自動生成する試験ケースの資産化機能を開発する。この機能により、試験ケースの再利用が可能となり、試験ケースの作成・実行にかかるコストを削減することができる。

このために本研究では次を行う

1. 機器構成と試験手順の独立した記述と資産化
2. 資産化した試験手順からの試験ケースインスタンスの自動生成・自動評価

以降それぞれ述べる。

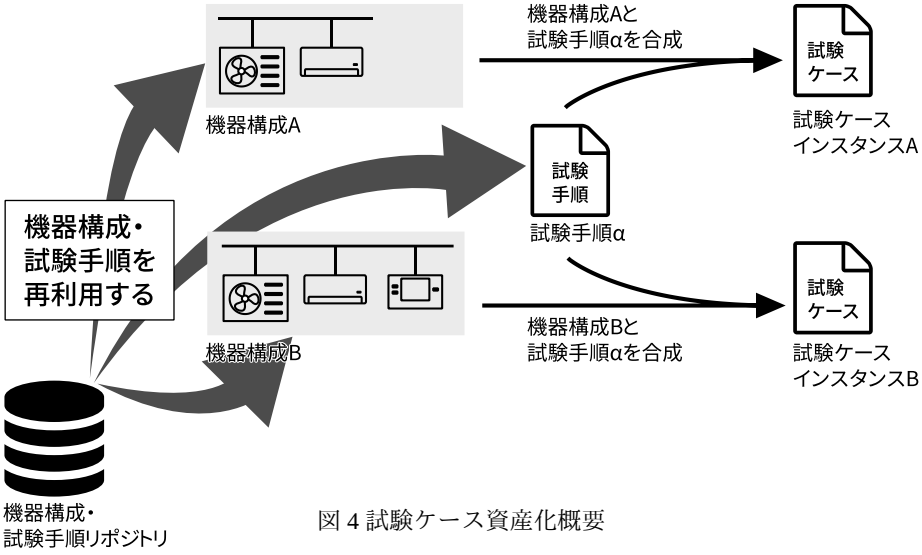

図 4 試験ケース資産化概要

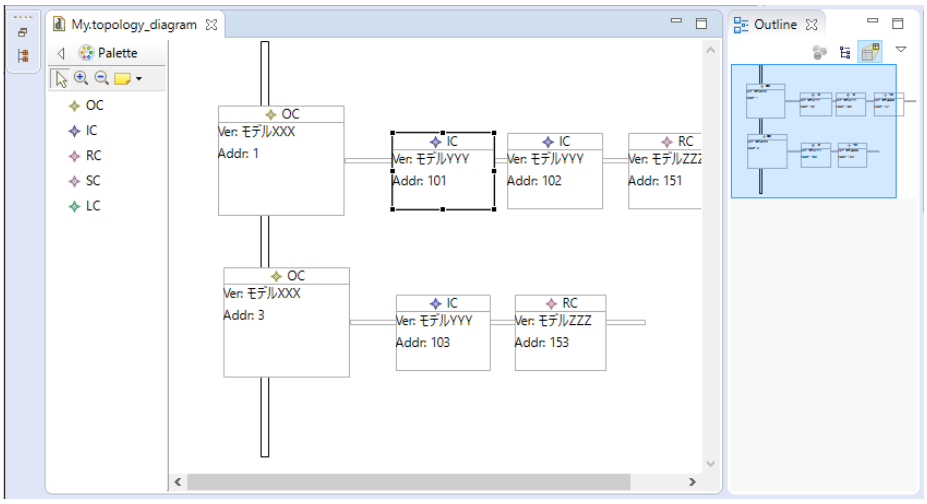

図 3 機器構成編集画面

2.1.1 試験機器構成の記述

試験機器構成については、試験ケースから試験手順を取り除くと、次の情報が必要である。

- どのバージョンの機器に対して試験を行うのか
- （機器間で通信を行うので）機器のアドレスは何か
- 試験対象機器はどの機器が何台あるのか

これを元に、試験機器構成編集可能な、図 3 のような機器構成編集画面を開発し、階層構造を表現可能な XML 形式で保存する。図 3 中、OC、IC、RC と表された資格の図形が各機器を表しており、各機器の図形中の Ver が機器のバージョン、Addr が機器のアドレスを表している。

2.1.2 試験手順の記述

本研究では、前述の機器構成と、機器・機能ごとの試験手順という、静的な情報を資産化し、試験対象機器に応じて試験手順を動的に書き換えることで個別の試験ケースインスタンスを作成する。書き換えを行う項目は次の 2 つである。

- 機器・機能ごとの試験手順を機器・機能の台数分複製
- 実機かモデル検査器によるシミュレーション機かに応じた実行方法の変更

上記書き換えを実現するためには、試験ケース中に記載した試験手順が、どの機器・機能の手順であるのか、というメタな情報が必要となる。本研究では、このメタな情報の記述及び、メタな情報と機器構成に基づく試験手順の書き換えを実現するために、テンプレートエンジンを採用した。テンプレートエンジンを用いると、例えばリスト 1 の 1 行目、26 行目にある“{% for ic in Units.IC %}”及び“{% endfor %}”の記述により、機器構成にある機器 IC の数だけプロセス procIC が複製される。また、8 行目の“{{rc.Addr}}”の記述により、機器構成にある機器 RC のアドレスの値によって、送信先アドレスを変更することができる。

2.1.3 試験ケースインスタンス生成

2.1.1、2.1.2 節で述べた試験機器構成と試験手順を用いてモデル検査器によって実行可能な試験ケースインスタンスを生成する。本研究では試験手順の試験機器構成による動的書き換えを、テンプレートエンジンによって実現した。2.1.2 節で述べたように、試験手順中に動的書き換えが必要な箇所を記述しておき、試験手順と機器構成をテンプレートエンジンに入力することで、試験ケース中のプロセス

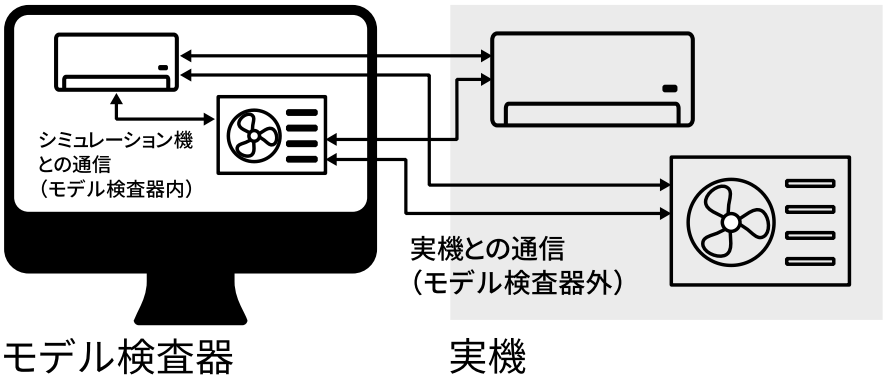

図 5 モデル検査器・実機間通信概要

リスト 1 テンプレートエンジン記述例

```
1. {% for ic in Units.IC %}
2. {% for rc in Units.RC %}
3. active proctype procIC{{suffix(ic)}}() {
4.  // コマンド A, B の送信順序によらず、
5.  // コマンド C は期待通りの応答になる。
6.  if
7.  :: skip ->
8.     // コマンド A を{{rc.Addr}}へ送信
9.     // コマンド A 応答を{{rc.Addr}}から受信
10. :: skip ->
11.    // コマンド B を{{rc.Addr}}へ送信
12.    // コマンド B 応答を{{rc.Addr}}から受信
13. fi;
14. if
15. :: skip ->
16.    // コマンド A を{{rc.Addr}}へ送信
17.    // コマンド A 応答を{{rc.Addr}}から受信
18. :: skip ->
19.    // コマンド B を{{rc.Addr}}へ送信
20.    // コマンド B 応答を{{rc.Addr}}から受信
21. fi;
22. // コマンド C を{{rc.Addr}}へ送信
23. // コマンド C 応答を{{rc.Addr}}から受信
24. }
25. {% endfor %}
26. {% endfor %}
```

の数やアドレスが書き換え、実行可能な試験ケースインスタンスの生成を行う。

2.2 モデル検査と実行テストの統合

本節では、図 2 中の ③ に相当する、モデル検査・実行テスト統合機能について述べる。

実装済みのシステムの非決定性を含む試験ケースをモデル検査で評価しようとした場合、1 節で述べたように、モデル検査に与える試験ケースは、実装済みのシステムすべてを記述するのではなく、モデル検査時には実装されたシステムの一部を抽象化して記述される。こうして記述されたシステムは、実装済みのシステムの一部を取り出したものであるため、

- 実装済みのシステムでは起こる不具合が、モデル検査用の専用言語で記述した範囲では再現できないことがある
- モデル検査器でのエラー出力から、実装済みのシステムのエラー状態を対応付けることが困難である

という課題があった。

上記に対し、本研究では、モデル検査器から実機・エミュレータを直接操作することによって、動作中の実機・エミュレータの動作を直接確認可能とする。さらに、実機での動作ログとモデル検査器によるエラー出力を統合し、どのような通信が行われた結果エラーが発生したのかの解析を支援するとともに、この統合されたエラー出力により従来モデル検査器のエラー出力を改善する。モデル検査器のエラー出力の改善については、 2.3 節で述べる。ここでは、モデル検査から実行テストを行うために必要な機能として、次の項目についてそれぞれ説明する。

1. 実機との通信機能の追加
2. モデル検査の試験最適化処理の削除

2.2.1 実機との通信機能の追加

CPS では多数の機器が相互に通信を行っている。モデル検査器と実機をつなぐためには、図 5 のように、モデル検査器上でもこの通信を可能にする必要がある。本研究ではプロトタイプとして、ビル用空調制御システムとの通信を行うため、下記を行った。

1. 実機との通信を仲介するプロトコル変換機との通信を行うドライバの実装
2. モデル検査器への通信ドライバを用いた通信機能の追加

1 について、モデル検査には表 1 モデル検査器の例があり、採用するモデル検査器によってどのように実装するかが異なってくる。本研究では次のような理由からモデル検査器として、Spin[1]を採用した。

表 1 モデル検査器の例

モデル検査器	特徴
Spin [1]	モデル検査器用の専用言語に C 言語を埋め込める
UPPAAL [2]	時間制約の検証が可能
NuSMV [3]	シンボリックモデル検査器に分類され、探索する状態を抑えた検査が可能

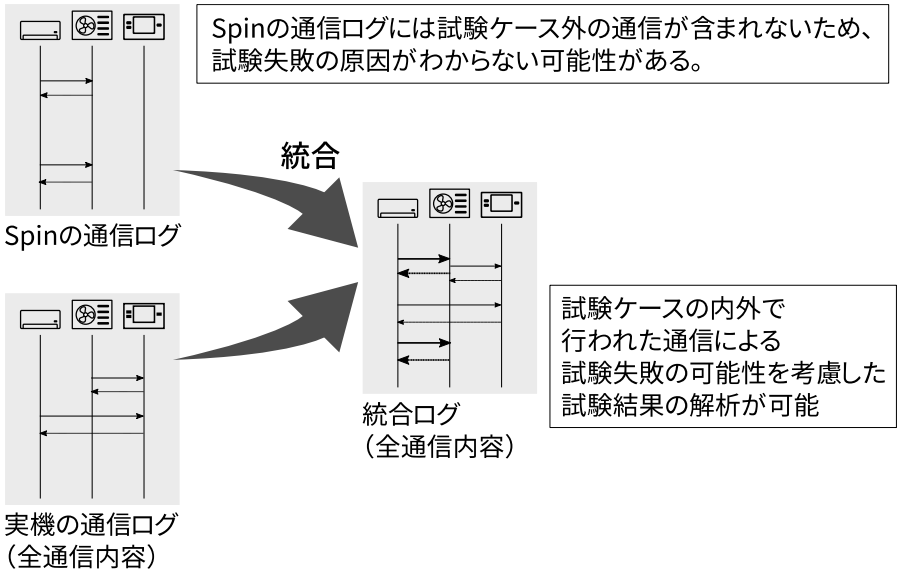

図 6 試験結果可視化概要

- Spin 用の専用言語 Promela には通信を表現する構文があり、CPS で行われるような機器・機能間の通信を表現しやすい
- Promela に C 言語プログラムを埋め込む構文があり、ドライバの機能呼び出しを行う際など、機能拡張が行いやすい

本研究では、このために実機通信用のドライバの開発を行い、モデル検査器 Spin に対する通信機能の追加については、次のように行った。

1. 実機との通信用ドライバの開発及び、C 言語からの呼び出しインターフェースの整備
2. Promela 上の C 言語埋め込み機能を用い、1 のドライバを介した通信機能（受信・送信）呼び出し
3. Promela から変換されたモデル検査プログラム（C 言語）に対する初期化処理の追加

(ア) 1 のドライバ初期化処理
(イ) 試験ケースごとの実機状態の初期化処理

2.2.2 •モデル検査の試験最適化処理の削除

モデル検査ツール Spin ではシミュレーションの回数を減らす最適化処理がなされている。この最適化処理のうち、実行順序組み合わせにおいて分岐が生じた部分（次に実行可能な処理が複数存在する部分）を試験する際、分岐点に到達するまでの経路を最初から試験を実行するのではなく、分岐が生じた時点の状態から試験を始めるという処理がある。 Spin は起こりうる実行順序組み合わせの各処理のステップごとに状態を保持しているため、このような途中実行が可能となっている。しかし、実行テストにおいて、モデル検査用の専用言語には実際のシステムの状態すべてが記述されないため、処理ごとにシステムがどのような状態にあるのかを保持することは困難である。したがって、この試験の実行を削減するための途中実行処理を削除し、試験ケースごとにシステムの初期化を行い、試験ケースの開始ごとにシステムを同一の状態へ初期化した後、試験ケースの最初から試験を実行するように変更を行った。

2.3 試験結果解析支援

本節では、図 2 中の④に相当する、試験結果解析支援機能について述べる。従来のモデル検査器は、試験が失敗した際には反例という形で、モデル検査器に与えた時相論理のロジックの矛盾、シミュレーション環境での各機能の状態遷移系列を出力する。このモデル検査器が出力する試験失敗時の出力はモデル検査用の専用言語で記述した範囲内のものである。しかし、モデル検査用の専用言語で記述された限られた機器・機能のシミュレーションを行うモデル検査とは異なり、実機を直接動作させる実行テストでは、試験する機能以外の複数の機能が動作しうる。したがって、モデル検査用の専用言語で記述した範囲外の情報、

1. モデル検査用の専用言語で記述した範囲外で機器がどのような状態になっているのか
2. モデル検査用の専用言語で記述した範囲外でどのような通信が行われたか

を得ることができなければ試験失敗の原因となった通信の解析ができない可能性がある。

1 については、モデル検査器からエミュレータを操作することによって、試験中に実機がどのような状態になるのかを確認可能にする。2 については、図 6 のように、モデル検査器上でシミュレーションとして行われた通信と、実機上で行われた試験で対象としていなかった通信を統合し、シーケンス図風の形式で可視化することで、試験失敗時の原因の発見を支援する。実際に本統合試験環境にて可視化を行った例は図 9 である。図中赤線のメッセージは試験ケース内に書かれていた通信であり、黒線のメッセージは試験ケース外で行われた通信である。メッセージ名にはコマンドのバイト列とタイムスタンプが表示される。例えば、図 9 中最初のメッセージは、X1 コマンドが時刻 11:56:37.250 に送られたことを示す。

この試験中の全通信ログとモデル検査で行われた通信を統合した通信シーケンス可視化により、試験対象以外の通信が原因で試験が失敗した場合でも試験失敗の原因調査が可能となる。

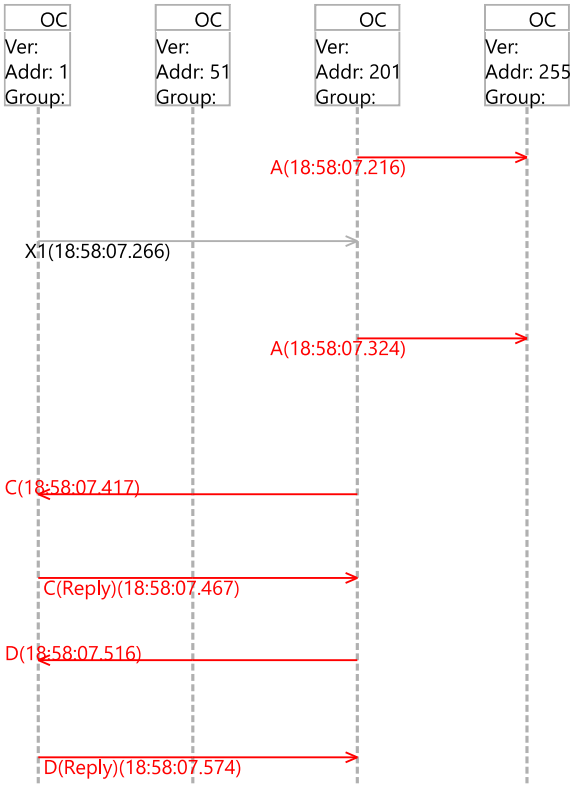

図 7 事例 1 通信ログ(ケース 1)

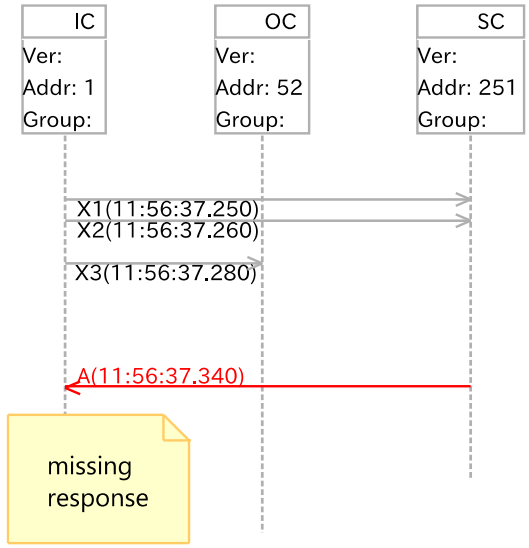

図 9 解析結果の例

3. 評価

非決定的な要素を含み、試験困難であったため工期内に試験を実施することができず、後工程に流出してしまった事例 3 件について、本統合試験環境にて試験ケースを作成・実行が可能であることを確認した。

3.1 事例

以降、各事例について説明する。

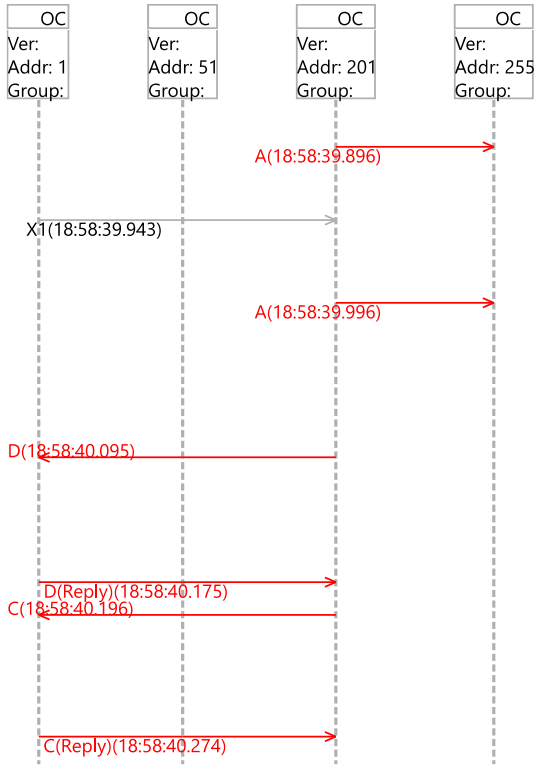

図 8 事例 1 通信ログ(ケース 2)

3.1.1 機器の通信順序の非決定性

システム上のある機器を特定のモードに変更するために、異なる数種類のコマンドを決められた順序で受信しなければならない、という仕様があった。この仕様に基づき、決められた順序でコマンドを送信しても、受信時にはその順序が入れ替わっていることが有り、特定のモードに変更できない、という不具合が後工程に流出した。この事例では試験対象の機器にコマンドを非決定的な順序で送信する試験ケースが必要となる。しかし、実装済みの製品では機能ごとにどのような順序でコマンドを送受信するのかが決まっており、試験時に送信順序を変更することが難しかった。したがって、従来では、網羅的な順序でコマンドを送信する試験を行うことが困難であった。

3.1.2 機器の機能組み合わせの非決定性

システム上のある機能は、他の機能と機器の制御方法が競合するものがあった。制御方法が競合する場合、機能ごとに優先順位があり、優先順位の高い機能の制御を優先する仕様になっていたが、競合する機能がどれであるのか、を特定し、試験することが難しかった。

3.1.3 特定の機器を制御する複数機器間のモード組み合わせの非決定性

ある機器 A を制御する機器が機器 B、機器 C と複数あり、この機器 B、C 間のモードの違いによって、制御される機器 A の特定の機能の制御方法が異なることが有り、このために機器 A の特定の機能が不安定になることがあった。しかし、競合するモードがどれであるのかを特定し、試験することが難しかった。

4. 評価結果

本統合試験環境により従来試験困難であった事例を試験したところ、次のような結果を得た。

4.1 機器の通信順序の非決定性

3.1.1 の事例は図 7 のように、3 つのコマンド A、B、C と、A を除く応答コマンド B(Reply)、C(Reply)から構成され、最初のコマンド A に続いて 3 つのコマンド A、B、C を非決定的な順序で送信しても、コマンド B、C に対してそれぞれ応答 B(Reply)、C(Reply)が返ってくることを確認するものである。試験の結果、最初のコマンド A の後の 3 つのコマンドの送信順序が図 7、図 8 のように入れ替わり、試験全体では起こりうる3! = 6通り全ての送信順序で試験が行われていることを確認した。

4.2 機器の機能組み合わせの非決定性

本試験では、温度の制御方法が競合している複数の機能を動作させる試験を行った。 試験時には、優先順位が低いはずの機能が優先順位の高い機能より優先されて温度制御を行ったため、実機の動作温度の表示が優先順位の低い機能による温度となっていた。

4.3 特定の機器を制御する複数機器間のモード組み合わせの非決定性

現場の事例を引き起こす通信シーケンスを試験ケースに記述し、実行できることを確認した。実行した結果を本統合試験環境における試験結果解析支援機能により可視化すると、図 10 のようになった。

5. 考察

評価実験で行った事例に対する試験の結果を考察する。

5.1.1 機器の通信順序の非決定性

試験ケースには、コマンド A、B、C の通信の順序を非決定的に入れ替える記述を行っていた。評価試験の結果からも、この送受信順序を網羅的に入れ替えた試験が行われていることが確認できた。例えば図 7、図 8 のケースでは、後半の通信 4 つにおいて、コマンド C と D の送信順序が入れ替わっている。

これから、3.1.1 の事例を再現する通信試験を、本統合試験環境にて行うことができたことが確認できる。特に、3.1.1 の事例において、現場での試験を困難にしていた、任意の順でコマンドの送信順序の変更を本統合試験環境が可能にしたこと、不具合の起こる順序を特定しなくてもモデル検査によって、網羅的な順序での試験することで不具合が起こるかを確認する試験ができたことがわかる。

5.1.2 機器の機能組み合わせの非決定性

4.2 節より、機器の画面表示と優先されるべき温度制御による動作温度の矛盾が起きていることがわかる。これは、機器の制御方法が競合した機能が同時に稼働した時、優先されるべき機能が優先されていないという 3.1.2 の事例を再現する試験が行われたことを示す。

5.1.3 特定の機器を制御する複数機器間のモード組み合わせの非決定性

本事例用の試験ケースには、本事例を再現する通信シーケンス（コマンド A、B、C、D を、A→B→C→D の順に送信する）を行う記述をしていた。試験結果を本統合試験環境における試験結果解析支援機能により可視化した図 10 の通信シーケンスを見ると、試験ケースに記述した通り、3.1.3 の事例を再現できていることが確認できる。

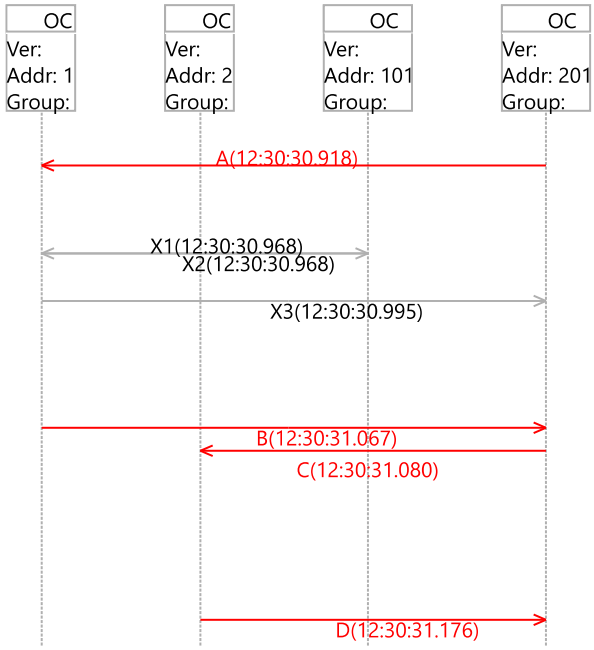

図 10 事例 3 通信ログ

上記 3 件の事例に対する評価実験により、本統合試験環境によって、網羅的な通信順序組み合わせの試験を実機に対して行うことができることを確認した。

6. 今後の展望

5 節より、本統合試験環境によって、現場での事例を再現するための試験が可能であることを確認した。

3.1.1 の事例では、不具合の起こるコマンドの送信順序を特定していなくても、モデル検査によって網羅的な順序で試験することで、不具合が起こるかを確認する試験が可能であった。

しかし、3.1.2 の事例、3.1.3 の事例の再現試験では、競合する機能・モードを特定し、試験ケースに組み込んだ上で試験を行っていた。モデル検査による非決定性を用い、任意の組み合わせで機能・モードを動作させるコマンドを送信すれば、競合する機能・モードがどれであるのかを特定せずに不具合が起こるかを確認する試験が可能である。しかし、その際、試験ケースごとに競合しうる機能・モードに変更するための通信シーケンスをすべて記述しておく必要があるため、再利用が難しくなる。

この課題を解決するために、今後は、機能・モードごとの通信シーケンスの試験ケースを記述・資産化し、資産化した任意の機器・モードの試験ケースを自動で組み合わせた試験ケースを生成する機能を実装していく。

さらに、本研究では、モデル検査を用いて、非決定的な要素を含む試験ケースを網羅的な順序で試験することで評価を行っている。このため、試験ケース中のプロセスの数や、任意の順で送信・受信されるコマンドなど、非決定的な要素が増えるに従い、モデル検査実行時に生成される試験ケースが爆発的に増加する。この膨大な試験ケースの実行により現実的な試験が不可能となることを避けるため、試験ケース中の非決定的な記述を決定的な記述へ変換する

機能を開発する。非決定的な記述がなされた試験ケースに対し、試験実施者の指定によって、優先的に試験すべき箇所の非決定性のみ網羅的に試験し、その他を決められた順序で試験されるようにすることで現実的な試験を可能とする予定である。

7. むすび

非決定的な要素を含むため困難となっている、CPS における評価の課題を解決するための統合試験環境のプロトタイプとしてビル用空調管理システムを対象に開発した。開発した統合試験環境を用いて、現場での事例 3 件を再現する試験ケースの作成・実行が可能なことを確認した。

謝辞

本研究は，JSPS 科研費 16K00100，JST CREST の支援を受けたものである。

参考文献

[1] Ammann, Paul E., Paul E. Black, and William Majurski. “Using model checking to generate tests from specifications.” Formal Engineering Methods, 1998. Proceedings. Second International Conference on. IEEE, (1998).
[2] Behrmann, Gerd, Alexandre David, and Kim G. Larsen. “A tutorial on uppaal.” Formal methods for the design of real-time systems. Springer Berlin Heidelberg, (2004): 200-236.
[3] Cimatti, Alessandro, et al. “Nusmv 2: An opensource tool for symbolic model checking.” Computer Aided Verification. Springer Berlin Heidelberg, (2002).
[4] Gerard J. Holzmann, and Margaret H. Smith. “Automating software feature verification.” Bell Labs Technical Journal 5.2 (2000): 72-87.
[5] Gerard. J. Holzmann, “The SPIN model checker: Primer and reference manual”, Addison-Wesley Reading, (2004).
[6] García-Fanjul, José, Claudio de la Riva, and Javier Tuya. “Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking.” (2006).
[7] Homma, Kei, et al. “Modeling, verification and testing of web applications using model checker.” IEICE transactions on information and systems 94.5 (2011): 989-999.
[8] Rayadurgam, Sanjai, and Mats PE Heimdahl. “Coverage based test-case generation using model checkers.” Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the. IEEE, (2001).
[9] Zheng, Yongyan, Jiong Zhou, and Paul Krause. “A model checking based test case generation framework for web services.” Information Technology, 2007. ITNG'07. Fourth International Conference on. IEEE, (2007).
[10] 青山裕介、黒岩丈瑠、久代紀之, “既存ソフトウェア部品を用いたソフトウェア開発におけるソースコード理解支援ツール,” 情報科学技術フォーラム講演論文集, FIT(電子情報通信学会・情報処理学会)運営委員会, (2015).