# シーケンス図を用いたモデル検査支援ツール

後藤 隼弐　吉岡 信和

モデル検査を実際の業務に適用する場合，モデル検査の知識や経験を有した人材の不足やモデル検査未習熟者のための学習コストの発生が問題となる．そこで，著者は従来の設計ツールであるシーケンス図を用いてモデル検査を可能とするモデル検査支援ツールを開発した．開発したツールでは，シーケンス図で記載した内部仕様と外部仕様を，検査モデルと検査仕様にそれぞれ変換することでモデル検査ツールへの入力を可能とする．また，モデル検査ツールが出力する反例情報をシーケンス図に変換することで，設計者はモデル検査を意識することなくモデル検査の恩恵を受けることができる．提案ツールの評価として，業務モデルに提案ツールを適用し，業務で実際に検出した再現性の低い不具合が学習コストを抑えながら効果的に検出できることを確認した．

In case of applying a model checking method to a business, a learning cost and a stable securement of human resources are problems. So author developed a model checking support tool that enables model checking method using a sequence diagram widely used as a design tool. In the tool, internal specifications and external specifications written by sequence diagrams are translated into inspection models and inspection specifications, and a counterexample as a result of verification by a model checking tool is translated into a sequence diagram. As a result, a designer can take benefits of a model checking method without learning it. As a evaluation of the tool, author confirmed that a low reproducibility defect is detected effectively as a result of applying the method to a business model.

## 1　はじめに

著者は，デバイス制御ミドルウェアの開発業務経験より，再現性の低い不具合を早期かつ効果的に検出することが困難であることを問題として認識している．再現性の低い不具合は，負荷耐久試験といったシステム試験工程など，数多く試行することで初めて不具合が顕在化することが多く，開発工程の最下流で検出されることで改修の手戻りが大きくなり，コスト増やスケジュール遅延の原因となることが多い．

再現性の低い不具合を設計工程で検出するために，状態遷移表を作成して考慮に漏れがないか確認することが一般的に行われる．しかし，状態遷移表の妥当性を人手によるレビューで確認するのは容易ではなく，また，状態数に比例してレビューに要する工数が増大するため，設計工程で検出するコストメリットが少なくなってしまうことがある．実際に，手戻りリスクを覚悟して早めに動作検証を実施するほうが全体的なコストを低減できる場合がある．

人手によるレビューで確認するのではなく，システムがとりうるすべての状態に対して期待する性質が満たされるかどうかを自動的に検証する技術として，モデル検査手法がある．しかし，モデル検査を実際の業務に適用する場合，モデル検査の知識や経験を有した人材が不足していることや，モデル検査未習熟者への学習コストが発生することが問題となる．たとえば，モデル検査を実施するためにはモデル検査ツールへの入力となる検査モデルや検査仕様を記述する必要があるが，検査モデルを記述するためにはモデル検査ツール毎に定義されたモデル記述言語を習得する必要があり，検査仕様を記述するためには命題論理や述語

---

A model checking support tool using a sequence diagram

Junji Goto, フェリカネットワークス株式会社, FeliCa Networks, inc..

Nobukazu Yoshioka, 国立情報学研究所, National Institute of Informatics.

論理，時相論理，集合論，オートマトンといった知識を習得する必要がある．また，モデル検査ツールが出力する反例情報は，シーケンス図などの設計ツールとして一般的に利用される形式で出力されるわけではないため，解釈するのが容易ではなく，複雑な反例情報を解釈するためにはある程度の経験が必要となる．

ここで，モデル検査を業務に適用する場合の問題を解決するための課題として，次の 3 つを設定する．

**課題 1** 検査モデル記述能力習得コストの低減

**課題 2** 検査仕様記述能力習得コストの低減

**課題 3** 反例情報解析能力習得コストの低減

著者は，これらの課題の解決により，学習コストを抑えながらモデル検査の適用を可能とすることを目的としたモデル検査支援ツール (以降，“提案ツール”と表現する) を開発した．提案ツールは，設計者がモデル検査ツールの入出力データを直接記述，解析しなくてもモデル検査を実施できるようにすることで学習コストの低減を実現している．提案ツールはオープンソースソフトウェアとして公開している [13].

開発にあたり，著者は，まず要求を分析し提案ツールの実現により達成するべき目標を定め，提案ツールに求められる機能を整理した．要求分析については 2 章にて述べる．つづいて，実際の開発は，機能毎に要件を整理し，要件毎に設計方針を定め実装を行った．要件と設計方針については 3 章にて述べる．実装については 4 章にて述べる．提案ツールの評価としては，業務で実際に開発したソフトウェアに提案ツールを適用することで効果を確認した，また提案ツールを適用しなかった場合との比較を行い，目標の達成度を確認した．評価については 5 章にて述べる．最後に，6 章でまとめと課題，今後の展望について述べる．

## 2 要求分析

本章では，要求分析として提案ツールの実現により達成するべき目標を明確にし，目標を達成するために必要な提案ツールに要求される機能がなにかを明確にする．

### 2.1 目標

著者は，提案ツールの実現により達成するべき目標として，著者が業務で実際に検出した再現性の低い不具合が，学習コストをかけずに検出できることを設定した．当該不具合は発生確率が低く，開発の最終工程として実施する連続運転による負荷耐久試験により検出された．リリース直前の工程で検出されたため，影響が大きい根本的な改修は実施することが出来ず，設計としては筋の良くない改修をするしか選択肢がなかった．もし設計工程にてモデル検査を用いて当該不具合を検出することができていれば，より望ましい根本的な改修が可能であったと考えられる．

当該不具合を検出したソフトウェアはデバイス制御ミドルウェアである．デバイス制御ミドルウェアの構成を図 1 に示す． デバイス制御ミドルウェアは，アプリケーション向けの API とデバイス制御デーモンで構成される．API はライブラリとして提供され，アプリケーションにリンクされる．API とデーモンはプロセス間通信を行い連携して動作する．API はアプリケーションからデバイスの操作要求を受け取り，デーモンに伝達する．デーモンは API からの要求を受信すると，実際にデバイスを制御してその結果を API に通知する．

デバイス制御ミドルウェアで検出した不具合は，デバイス制御の処理中断機能の外部仕様違反である．処理中断機能の期待動作となるシーケンス図を図 2 に，不具合発生時のシーケンス図を図 3 に示す．

処理中断機能は，デーモンがデバイス制御の処理完了を待っている間にアプリケーションからデバイスクローズ要求を受信した際に処理完了待ちを中断する

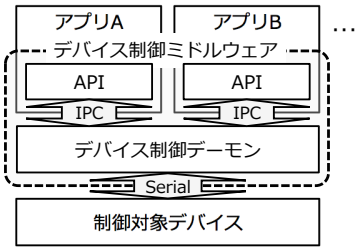



**図 1 デバイス制御ミドルウェアの構成**

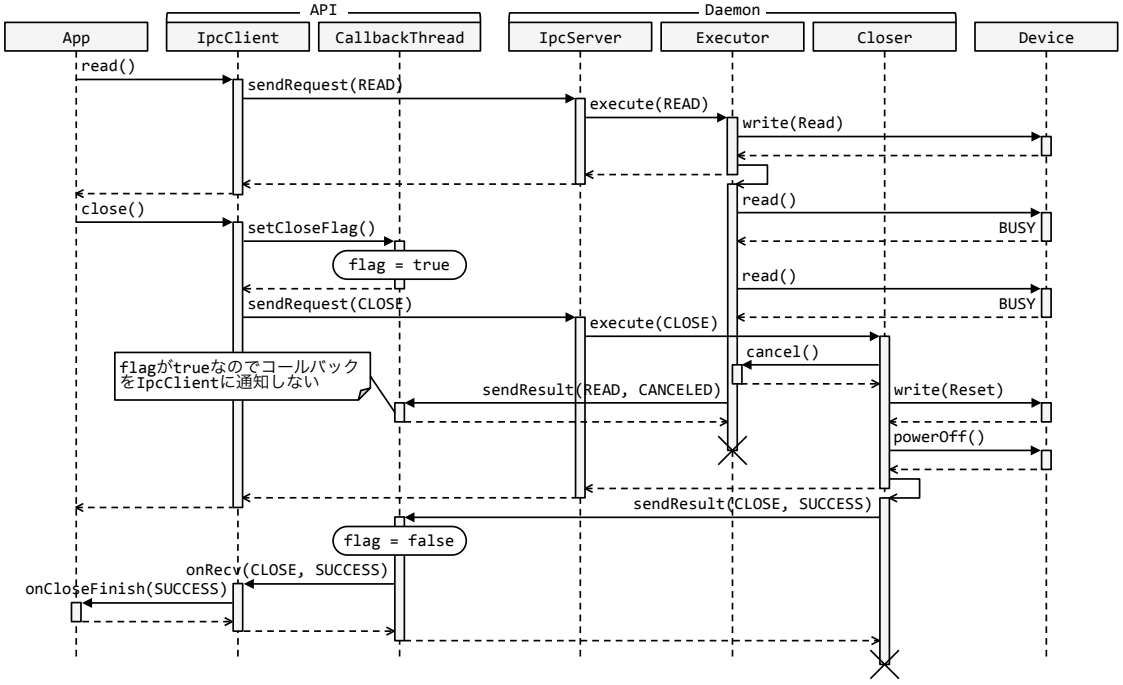



**図 2　処理中断機能の期待動作シーケンス図**

機能である．アプリケーションが API を実行するとデバイス制御をデーモンに要求し，その結果はコールバックにより通知されるが，アプリケーションがコールバックの通知を待つ間にデバイスクローズ API を実行すると，通知待ち対象のコールバックの通知がキャンセルされるというのがデバイス制御ミドルウェアの外部仕様になっている．内部仕様としては，キャンセル対象のコールバックがデーモンから API に通知された後，API 内のコールバック通知抑制フラグによりアプリケーションに通知するか否かを制御する仕組みとしていた．フラグは，デバイスクローズ要求の実行時に真に設定され，デバイスクローズ要求のコールバックが通知された時に偽に設定されるように実装していた．コールバックはフラグが偽のときのみアプリケーションに通知される．

しかし，検出した不具合では，キャンセルされるべきコールバックが通知されてしまうという外部仕様違反を起こしていた．不具合は，キャンセル対象となるコールバックよりも先に，デバイスクローズ要求のコールバックが通知されることにより，フラグが意図しないタイミングで偽に設定されてしまうことでアプリケーションへのコールバック通知の抑制に失敗してしまうことで発生していた．つまり，内部仕様と外部仕様との間に整合性がなかったことが不具合の原因である．

### 2.2　要求される機能

2.1 節にて，提案ツールの実現により達成するべき目標として，業務で検出した不具合が学習コストをかけずに検出できることを設定した．また，業務で検出した不具合が，内部仕様と外部仕様の不整合に起因することを説明した．

内部仕様と外部仕様の不整合は，モデル検査を用いて内部仕様と外部仕様の詳細化関係，つまり内部仕様が外部仕様を満たしているかを検査することで検出することができる．学習コストをかけずにモデル検査を実施するためのアプローチとして，ソフトウェア設計者がモデル検査ツールの入出力を直接扱わないようにする方針を採用した．この方針に従い，提案ツールに要求される機能として，次の 3 つを設定した．

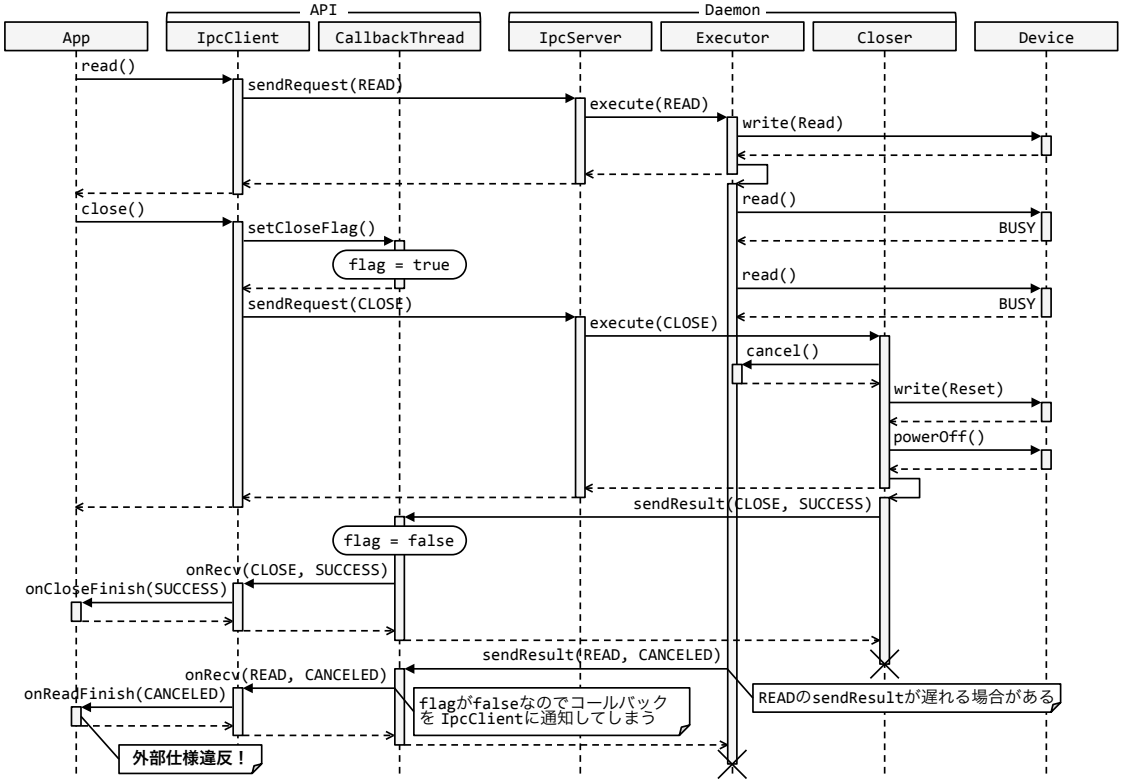



**図 3　処理中断機能の不具合発生時のシーケンス図**

**機能 1**

シーケンス図で表現された内部仕様を検査モデルに変換する機能

**機能 2**

シーケンス図で表現された外部仕様を検査仕様に変換する機能

**機能 3**

反例情報をシーケンス図に変換し期待動作と異常動作の差分を表示する機能

これらの機能は，1 章で設定したモデル検査を業務に適用する場合の 3 つの課題に対応している．

提案ツールは，オブジェクト指向モデリング [8] でモデル化されたシステムを対象とし，UML(Unified Modelling Language) [4] [5] シーケンス図 (以降，単に “シーケンス図” と表現する) を，モデル検査ツールに入力するための検査モデルや検査仕様へ変換する機能 (機能 1，機能 2)，および，モデル検査ツールが出力する反例情報をシーケンス図へ変換する機能 (機能 3) を提供する．つまり，提案ツールは，モデル検査ツールの入出力を，設計者にとって馴染みのあるシーケンス図に置き換える機能を提供する．提案ツールはモデル検査を実施する機能は提供しないため，外部のモデル検査ツールを用いてモデル検査を実施する．図 4 に提案ツールを利用した場合のモデル検査実施の流れを示す．

機能 3 では，単に反例情報をシーケンス図に変換するだけでなく，期待動作と異常動作のシーケンス図を両方表示し差分を明示することで，異常動作に至る経緯を容易に推測できることが要求される．これは，検査対象のシステムが複雑である場合を考慮すると，反例情報として単に異常動作のシーケンス図を表示したとしても，異常の原因および異常に至る内部動作の分析自体が困難であると考えたためである．

提案ツールを利用することで，設計者は，シーケン

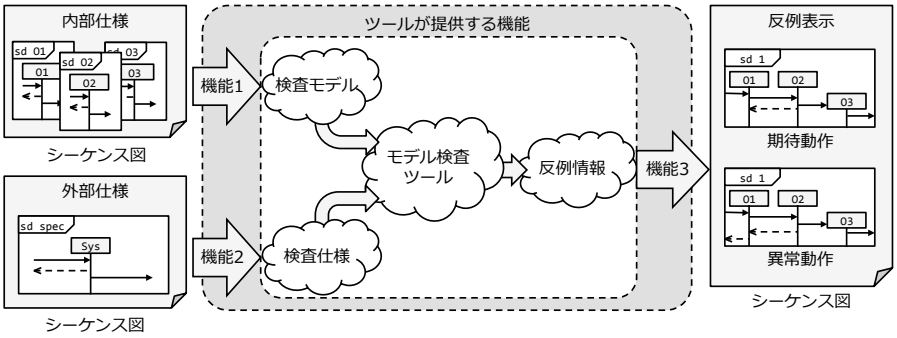



**図 4　提案ツールを利用した場合のモデル検査**

ス図を用いてシステムの仕様を記述するだけでモデル検査ツールへの入力を準備することができ，検査モデルや検査仕様を記述するための学習コストを必要としない．また，モデル検査ツールが出力する反例情報も提案ツールによりシーケンス図へ変換され表示されるため，モデル検査ツール毎の反例情報形式の学習や慣れるための経験を積む必要がなく，さらに，期待動作と異常動作の差分が明示されるため容易に反例情報を解釈することができる．

## 3　設計

本章では，提案ツールの設計について述べる．はじめに，提案ツールが対象とするモデル検査ツールについて述べ，つぎに，2.2 節で述べた提案ツールに要求される機能毎に要件を整理し，要件毎に設計方針を述べる．

### 3.1　モデル検査ツール

提案ツールは，シーケンス図を検査モデルおよび検査仕様へ変換し，反例情報をシーケンス図に変換する機能を提供するが，シーケンス図の変換先である検査モデルおよび検査仕様は，対象とするモデル検査ツール毎に異なる．シーケンス図への変換元となる反例情報についても同様である．本節では，対象とするモデル検査ツールの選定と，選定したモデル検査ツールを用いて内部仕様と外部仕様の整合性を検査する方法について述べる．

#### 3.1.1　モデル検査ツールの選定

バックエンドで動作させるモデル検査ツールとしては，プロセス代数 CSP(Communicating Sequential Processes) [3] [7] に基づくモデル検査ツールである FDR3(Failures/Divergence Refinement 3) [11] [12] を採用した．FDR3 を採用した理由は，次の 3 点である．

- FDR3 は CSP でモデル化された 2 つのプロセス間の等価関係や詳細化関係を検証することに適しており，内部仕様と外部仕様の整合性検査に適しているため
- FDR3 では検査モデルと検査仕様の記述に同じ言語である $\mathrm{CSP_M}$ [10] を用いるため，シーケンス図から検査モデル，検査仕様への変換に同じ仕組みを適用できるため
- FDR3 は反例情報を，JSON(JavaScript Object Notation) 形式 [2] や YAML(YAML Ain't Markup Language) 形式 [1] といった，機械処理が容易な形式で出力することができ，シーケンス図への変換が容易であるため

FDR3 への入力は内部仕様と外部仕様のシーケンス図から変換した $\mathrm{CSP_M}$ 形式のファイルである．また，シーケンス図への変換元となる FDR3 が出力する反例情報の形式は YAML 形式とした．

#### 3.1.2　内部仕様と外部仕様の整合性検査

内部仕様と外部仕様の整合性の検査は FDR3 を用いて次のような手順で実施する．

1. 検査対象のシステムを構成する各オブジェクト

毎に，内部仕様として，オブジェクトが受理可能なメッセージと，他オブジェクトとのメッセージ交換による相互作用の関係を $CSP_M$ を用いてモデル化する．

2. 各オブジェクトの内部仕様を表す複数のモデルを，オブジェクト間のメッセージ交換による相互作用が同期するように並行合成し，合成内部仕様を表すひとつのモデルを生成する．
3. システムの外部仕様として，システム外部のオブジェクトとシステムの相互作用の関係を $CSP_M$ を用いてモデル化する．モデル化された外部仕様の $CSP_M$ が検査仕様となる．ここで，モデル化された検査仕様を Spec とする．
4. 合成内部仕様を表すモデルに対し，外部仕様のモデルと合成することで，システム外部との相互作用が適用されたシステムの検査モデルが得られる．
5. 検査モデルを検査仕様で検査できるようにするために，検査モデルからシステム外部に公開されていない内部メッセージを隠蔽することで，検査モデルが扱うメッセージ集合を検査仕様と同等にする．ここで，内部メッセージが隠蔽された検査モデルを System とする．
6. System が Spec の失敗詳細化 (Spec $\sqsubseteq_F$ System) であることを，FDR3 を用いて検査する．

### 3.2 内部仕様シーケンス図から検査モデルへの変換 (機能 1)

本節では，2.2 節で提案ツールに要求される機能のひとつとして設定した，内部仕様のシーケンス図から検査モデルへ変換する機能 (以降，“機能 1” と表現する) について，要件毎に設計方針を述べる．

機能 1 に対する要件としては次の 4 つが挙げられる．

**要件 1-1**

入力となるシーケンス図が機械処理できる形式になっていること

**要件 1-2**

メッセージの返り値や引数，状態変数による式が評価できる検査モデルに変換できること

**要件 1-3**

シーケンス図をセマンティクスを維持したまま検査モデルに変換できること

**要件 1-4**

複数オブジェクトの内部仕様のシーケンス図を合成し，システムの検査モデルに変換できること

#### 3.2.1 要件 1-1 設計方針

内部仕様を検査モデルの $CSP_M$ 形式に変換するために，入力となる内部仕様のシーケンス図が $CSP_M$ 形式への変換に適した，機械処理できる形式になっていることが求められる．そのためには，シーケンス図から機械処理できる形式へ変換できること，あるいは，機械処理できる形式からシーケンス図へ変換できることのどちらかが求められる．

要件 1-1 に対する設計方針としては，機械処理できる形式からシーケンス図へ変換する方式を採用した．つまり，シーケンス図から直接 $CSP_M$ 形式へ変換するのではなく，シーケンス情報が表現できる DSL(Domain-Specific-Language)(以降，sd-DSL と表現する) から $CSP_M$ 形式に変換することで内部仕様の検査モデルを作成し，sd-DSL からシーケンス図を描画できる提案ツールを用いてシーケンス図を作図する．

この方式を採用した理由は，後述する反例情報からシーケンス図へ変換 (機能 3) する際に，YAML 形式の反例情報からシーケンス図を描画する必要があることを考慮すると，内部仕様と反例情報が同じシーケンス図の作図ツールで描画されたほうが，表現が統一される点で望ましいと考えたためである．

この設計方針に従って，sd-DSL からシーケンス図を作図できる提案ツールを調査し，最終的に PlantUML [6] を採用した．PlantUML を採用した理由は，内部仕様を記述するために必要な，複合フラグメントや注釈といったシーケンス図の構成要素を作図できるためである．内部仕様を記述するために必要なシーケンス図の構成要素については要件 1-2(3.2.2 小節) にて述べる．PlantUML の sd-DSL の例を図 5 に，図 5 の sd-DSL を PlantUML でシーケンス図に変換した例を図 6 に示す．

```
1  @startuml
2  title IpcServer
3  control IpcClient
4  control IpcServer
5  control Executor
6  control Closer
7  hnote over IpcServer : Ready
8  ====
9  hnote over IpcServer : Ready
10 IpcClient -> IpcServer ++ : sendRequest(apiName)
11 alt apiName==CLOSE
12     IpcServer -> Closer ++ : execute(CLOSE)
13     return
14 else
15     IpcServer -> Executor ++ : execute(apiName)
16     return
17 end
18 return
19 hnote over IpcServer : Ready
20 @enduml
```

**図 5　PlantUML の sd-DSL 例**

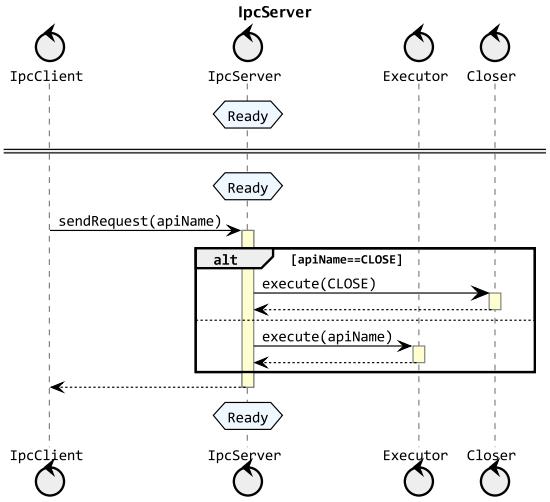



**図 6　図 5 の sd-DSL を PlantUML でシーケンス図に変換した例**

### 3.2.2　要件 1-2 設計方針

内部仕様を厳密に表現するためには，シーケンス図上のメッセージの返り値や引数を用いた式を扱える必要がある．たとえば，メッセージの返り値を元に条件分岐の条件式を指定したり，状態変数を用いた何らかの演算結果をメッセージの引数に指定したりする場合である．その場合，内部仕様のシーケンス図を検査モデルに変換する際に，検査モデルでは整数・論理・集合・列などの演算が扱える必要があり，そのためには検査モデルにてメッセージを定義する際に返り値や引数の型情報が必要となる．

シーケンス図は振る舞いを表現するのに特化した図であり，メッセージの型情報を表現するには適していない．そのため，提案ツールでは，メッセージの引数や返り値の型情報を定義する定義情報ファイルを別途用意し，内部仕様を検査モデルに変換する際に，定義情報も検査モデルに埋め込む方針とした．

定義情報は YAML 形式で記述し提案ツールに入力する実装とした．表 1 に，定義情報と対応する $\mathrm{CSP_M}$ の例を示す．定義情報の 1-3 行目は，ユーザ定義型として `Result` と `Mode` を，代数的データ型として定義している．これらは $\mathrm{CSP_M}$ では `datatype` を用いる定義に変換される．定義情報の 4 行目以降はクラス毎にメッセージの引数および返り値の型情報を定義している．定義情報で定義されたクラスは，$\mathrm{CSP_M}$ では `Class_`型の要素として定義される ($\mathrm{CSP_M}$3 行目)．メッセージは $\mathrm{CSP_M}$ では “クラス名_メッセージ名．引数の型”(応答メッセージの場合は “`return_`クラス名_メッセージ名．返り値の型”) という形式で `Method_`型の要素として定義される ($\mathrm{CSP_M}$4 行目以降)．5-8 行目は `Daemon` クラスのメッセージを定義しており，`start` メッセージは引数が無く，`Result` 型を返却することを表現している．11 行目は引数が複数ある場合の記述例であり，15 行目は返り値が無い場合の記述例である．16 行目の `cancel` メッセージには返り値に関する記述が無いが，これは `cancel` メッセージが非同期メッセージであることを示している．

### 3.2.3　要件 1-3 設計方針

sd-DSL を検査モデルの $\mathrm{CSP_M}$ に変換する際，シーケンス図のセマンティクスを維持したまま $\mathrm{CSP_M}$ に変換する必要がある．本小節では，提案ツールで採用したシーケンス図と $\mathrm{CSP_M}$ の対応関係について述べる．

シーケンス図におけるライフラインとメッセージは，$\mathrm{CSP_M}$ のプロセスとイベントに対応付ける．ライフラインとメッセージの $\mathrm{CSP_M}$ への変換例を表 2 に示す．内部仕様のシーケンス図はオブジェクト毎に用意するが，表 2 はオブジェクト `A` に関する内部仕様を表している．$\mathrm{CSP_M}$ においてメッセージの送信は，プレフィックス演算子により，メッセージ送信前のオブジェクトを表すプロセスが，メッセージ送

**表 1　定義情報の $CSP_M$ への変換例**

定義情報 (YAML)

```
1  Data:
2      Result: [ SUCCESS, FAILED, CANCELED ]
3      Mode: [ RW_MODE.Bool.Bool, CTRL_MODE ]
4  Class:
5      Daemon:
6          start:
7              args: []
8              return: Result
9      Device:
10         open:
11             args: [ Int, Mode ]
12             return: Bool
13         close:
14             args: []
15             return: null
16         cancel:
17             args: [ Bool ]
```

$CSP_M$

```
1  datatype Result = SUCCESS | FAILED | CANCELED
2  datatype Mode = RW_MODE.Bool.Bool | CTRL_MODE
3  datatype Class_ = Daemon | Device
4  datatype Method_ = Daemon_start | return_Daemon_start.Result |
5                     Device_open.Int.Mode | return_Device_open.Bool |
6                     Device_close | return_Device_close |
7                     Device_cancel.Bool
```

信を表すイベントの実行後に，メッセージ送信後のオブジェクトを表すプロセスとして振る舞うように定義される．たとえば，表 2 では，シーケンス図上のオブジェクト A から B への m1 メッセージの送信が，$CSP_M$ の 4 行目に対応しているが，ここでは，$CSP_M$ の A_Init_0() がメッセージ送信前のオブジェクト A を，msg_.A.B.B_m1!0 が m1 メッセージの送信イベントを，A_Init_1() がメッセージ送信後のオブジェクト A を表しており，プレフィックス演算子->により A_Init_0() が，m1 メッセージの送信後に A_Init_1() のように振る舞うように定義されている．メッセージ送信後の復帰やメッセージの受信についても，メッセージ送信と同じようにイベントとプレフィックス演算子を用いて $CSP_M$ に変換する．メッセージ送信を表すイベントは，表 2 $CSP_M$ の 3 行目に示すとおり，送信元のオブジェクト，送信先のオブジェクト，送信するメッセージの三つ組をパラメータに持つチャネル msg_として定義される．

内部仕様を作成する際，オブジェクトの状態毎に処理を記述することで作成が容易になることがある．シーケンス図において，オブジェクトの状態をライフライン上に六角形の注釈 (以降，“状態注釈” と表現する) を記述することで表現することにした．状態遷移と状態変数の $CSP_M$ への変換例を表 3 に示す．シーケンス図上の状態注釈は，使われる箇所によって 2 つの意味合いを持つとした．1 つ目は，二重線の直後に現れる状態注釈であり，これは状態注釈で表される状態の開始を表す．2 つ目は，二重線の直後以外に現れる状態注釈であり，これは状態注釈で表される状態への遷移を表す．シーケンス図の開始直後は，初期状態 (Init) であるとした．

また，状態は状態変数を持つことができ，状態変数をメッセージの引数や複合フラグメントでの条件判定に用いることができるとした．状態変数は，状態の開始を表す状態注釈で状態名の後に “変数名:型名” を括弧内に記述することで定義する．変数は複数定義でき，その際は変数名と型名の定義をカンマ (“,”) で区切り記述する．状態遷移の状態注釈では遷移先の状態名の後に括弧で遷移後の状態変数の値を指定できるとした．たとえば，表 3 のシーケンス図では，オブジェクト A の状態 Ready が状態変数 v0(型：Int) を持ち，シーケンス図の最後で，状態 Ready に状態変数 v0 の値が v0+ret となるように遷移していることがわかる．

状態変数は，ライフライン上で四角形の注釈を用いることで値を更新できるとした．たとえば，表 3 のシーケンス図では，オブジェクト A の状態変数 v0 が v0+1 に更新される例を示している．

$CSP_M$ において，状態は，プロセス名の一部として変換される．たとえば，オブジェクト A が状態 Ready であるプロセスは A_Ready_$n$ として表される．$n$ は当該状態での処理毎に採番される整数である．$CSP_M$ において，状態変数は，状態変数を管理するプロセスと状態変数を操作するためのイベントを用いて表現される．表 3 $CSP_M$ では，3 行目でオブジェクト A の状態変数 v0 をユーザ定義型 StateVar_の要素とし

**表 2　ライフラインとメッセージの $CSP_M$ への変換例**

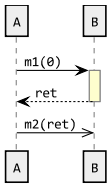
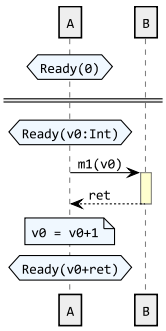
| シーケンス図 | sd-DSL | $CSP_M$ |
|---|---|---|
|  | 1 @startuml<br>2 participant A<br>3 participant B<br>4 A -> B ++ : m1(0)<br>5 return ret<br>6 A ->> B : m2(ret)<br>7 @enduml | 1 datatype Class_ = A \| B<br>2 datatype Method_ = B_m1.Int \| return_B_m1.Int \| B_m2.Int<br>3 channel msg_ : Class_.Class_.Method_<br>4 A_Init_0() = msg_.A.B.B_m1!0 -> A_Init_1()<br>5 A_Init_1() = msg_.B.A.return_B_m1?ret -> A_Init_2(ret)<br>6 A_Init_2(ret) = msg_.A.B.B_m2!ret -> A_Init_3(ret)<br>7 A_Init_3(ret) = SKIP |

**表 3　状態遷移と状態変数の $CSP_M$ への変換例**

| シーケンス図 | sd-DSL | $CSP_M$ |
|---|---|---|
|  | 1 @startuml<br>2 participant A<br>3 participant B<br>4 hnote over A : Ready(0)<br>5 ====<br>6 hnote over A : Ready(v0:Int)<br>7 A -> B ++ : m1(v0)<br>8 return ret<br>9 note over A : v0 = v0+1<br>10 hnote over A : Ready(v0+ret)<br>11 @enduml | 1 datatype Class_ = A \| B<br>2 datatype Method_ = B_m1.Int \| return_B_m1.Int<br>3 datatype StateVar_ = A_Ready_v0.Int<br>4 channel msg_ : Class_.Class_.Method_<br>5 channel init_, sync_, update_ : StateVar_<br>6 SV_A_Ready_v0_Init = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>7 SV_A_Ready_v0(val) = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>8 [] update_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>9 [] sync_.A_Ready_v0!val -> SV_A_Ready_v0(val)<br>10 A_Init_0() = init_.A_Ready_v0!0 -> A_Ready_0()<br>11 A_Ready_0() = sync_.A_Ready_v0?v0 -> msg_.A.B.B_m1!v0<br>12 -> A_Ready_1()<br>13 A_Ready_1() = msg_.B.A.return_B_m1!ret -> A_Ready_2(ret)<br>14 A_Ready_2(ret) = update_.A_Ready_v0!(v0+1) -> A_Ready_3(ret)<br>15 A_Ready_3(ret) = init_.A_Ready_v0!(v0+ret) -> A_Ready_0() |

て定義し，5 行目で状態変数の操作に用いるチャネル `init_`，`sync_`，`update_`を定義している．`init_`は初期化，`sync_`は値の参照，`update_`は値の更新を表している．また，6-7 行目では，オブジェクト `A` の状態変数 `v0` を管理するプロセス `SV_A_Ready_v0` を定義している．

シーケンス図における複合フラグメントの alt と opt は，ガードの有無によって $CSP_M$ での表現が異なる．ガード有りの alt と opt では条件分岐を用いて表現する．表 4 に，複合フラグメント (ガード有り alt，ガード有り opt) の $CSP_M$ への変換例を示す．

一方，ガード無しの alt と opt は $CSP_M$ では外部選択を用いて表現する．ここで，内部選択を用いない理由は，本論文において内部仕様は実装工程への入力となる詳細設計情報となることを想定しているため，非決定的な振る舞いを意図して alt や opt を利用することは無いと考えたためである．内部仕様でガード無しの alt や opt を用いる必要があるのは，外部からのメッセージを受信する際に，メッセージの受信有無や，受信するメッセージの種類などがメッセージ送信オブジェクトにより決定される場合である．表 5 にガード無しの alt と opt の $CSP_M$ への変換例を示す．表 5 のシーケンス図では，alt 複合フラグメントによりオブジェクト `B` のメッセージ `m1` か `m2` がオブジェクト `A` により選択され送信されることを表現している．変換後の $CSP_M$ では 4 行目にて外部選択を用いて表現されている．

シーケンス図における複合フラグメントの loop は，$CSP_M$ では条件分岐と逐次合成を用いて表現する．表 6 に，複合フラグメント (loop) の $CSP_M$ への変換例を示す．ここで，逐次合成を用いる理由は，繰り返し処理の終端で先頭に戻る処理を考慮する必要がなくなり，変換処理を簡略化できると考えたためである．たとえば，表 6 $CSP_M$ の 16 行目は，sd-DSL の 9 行目，つまり繰り返し処理の終端箇所を表しているが，単に成功終了しているだけであり，先頭に戻るための処理を必要としない．

#### 3.2.4　要件 1-4 設計方針

3.1.2 小節で説明したとおり，内部仕様と外部仕様の整合性を検査するためには，オブジェクト毎に定

**表 4　複合フラグメント (ガード有り alt，ガード有り opt) の $CSP_M$ への変換例**

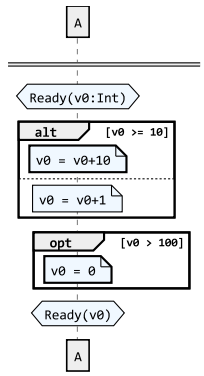
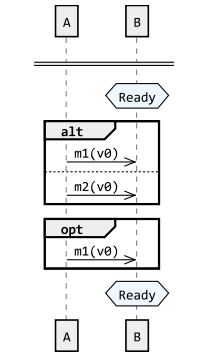
| シーケンス図 | sd-DSL | $CSP_M$ |
|---|---|---|
|  | 1 @startuml<br>2 participant A<br>3 ====<br>4 hnote over A : Ready(v0:Int)<br>5 alt v0 >= 10<br>6 note over A : v0 = v0+10<br>7 else<br>8 note over A : v0 = v0+1<br>9 end<br>10 opt v0 > 100<br>11 note over A : v0 = 0<br>12 end<br>13 hnote over A : Ready(v0)<br>14 @enduml | 1 datatype Class_ = A<br>2 datatype StateVar_ = A_Ready_v0.Int<br>3 channel init_, sync_, update_ : StateVar_<br>4 SV_A_Ready_v0_Init = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>5 SV_A_Ready_v0(val) = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>6 [] update_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>7 [] sync_.A_Ready_v0!val -> SV_A_Ready_v0(val)<br>8 A_Ready_0() = sync_.A_Ready_v0?v0 -> if v0 >= 10<br>9 then A_Ready_1()<br>10 else A_Ready_3()<br>11 A_Ready_1() = update_.A_Ready_v0!(v0+10) -> A_Ready_2()<br>12 A_Ready_2() = A_Ready_5()<br>13 A_Ready_3() = update_.A_Ready_v0!(v0+1) -> A_Ready_4()<br>14 A_Ready_4() = A_Ready_5()<br>15 A_Ready_5() = sync_.A_Ready_v0?v0 -> if v0 > 100<br>16 then A_Ready_6()<br>17 else A_Ready_8()<br>18 A_Ready_6() = update_.A_Ready_v0!(0) -> A_Ready_7()<br>19 A_Ready_7() = A_Ready_8()<br>20 A_Ready_8() = init_.A_Ready_v0!(v0) -> A_Ready_0() |

**表 5　複合フラグメント (ガード無し alt，ガード無し opt) の $CSP_M$ への変換例**

| シーケンス図 | sd-DSL | $CSP_M$ |
|---|---|---|
|  | 1 @startuml<br>2 participant A<br>3 participant B<br>4 ====<br>5 hnote over B : Ready<br>6 alt<br>7 A ->> B : m1(v0)<br>8 else<br>9 A ->> B : m2(v0)<br>10 end<br>11 opt<br>12 A ->> B : m1(v0)<br>13 end<br>14 hnote over B : Ready<br>15 @enduml | 1 datatype Class_ = A \| B<br>2 datatype Method_ = B_m1.Int \| B_m2.Int<br>3 channel msg_ : Class_.Class_.Method_<br>4 B_Ready_0() = B_Ready_1() [] B_Ready_3()<br>5 B_Ready_1() = msg_.A.B.B_m1?v0 -> B_Ready_2()<br>6 B_Ready_2() = B_Ready_5()<br>7 B_Ready_3() = msg_.A.B.B_m2?v0 -> B_Ready_4()<br>8 B_Ready_4() = B_Ready_5()<br>9 B_Ready_5() = B_Ready_6() [] B_Ready_8()<br>10 B_Ready_6() = msg_.A.B.B_m1?v0 -> B_Ready_7()<br>11 B_Ready_7() = B_Ready_8()<br>12 B_Ready_8() = B_Ready_0() |

義した内部仕様を，システム全体の内部仕様として 1 つの検査モデル $CSP_M$ に合成する必要がある．提案ツールでは，内部仕様をオブジェクト毎に $CSP_M$ に変換した後，複製型アルファベット付並行合成演算子を用いて，オブジェクトのプロセス同士をメッセージの送受信が同期するように並行合成することでシステム全体の検査モデルを生成する方針とした．内部仕様合成の $CSP_M$ への変換例を表 8 に示す．表 8 は，システムを構成するオブジェクトであるオブジェクト `A` とオブジェクト `B` の内部仕様シーケンス図を合成する例を示している．シーケンス図中の `U` はシステム外部のアクターであり，内部仕様としては定義しない．内部仕様の sd-DSL は，オブジェクト毎に `@startuml` と `@enduml` で囲み `title` で定義するオブジェクト名を記述する．変換先の $CSP_M$ の 25 行目にて，複製型アルファベット付並行合成演算子を用いてオブジェクト `A` とオブジェクト `B` を合成している．複製型アルファベット付並行合成演算子に与えるプロセスとイベントは，22-27 行目で定義した関数 (`getProcIf_()` および `getProc_()`) により得る．これらの関数は，クラス (`Class_`の要素) を引数により，当該クラスに対応するプロセスやメッセージ送受信イベント集合を返す．

### 3.3　外部仕様シーケンス図から検査仕様への変換 (機能 2)

本節では，機能 2 である，外部仕様のシーケンス図から検査仕様への変換について，要件と要件に対する設計方針について述べる．

機能 2 に対する要件は，シーケンス図から $CSP_M$

**表 6　複合フラグメント (loop) の $CSP_M$ への変換例**

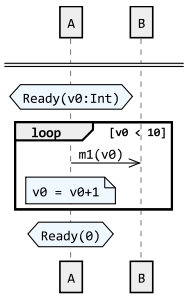
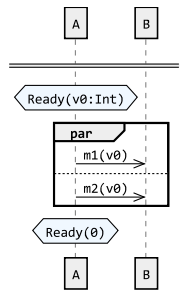
| シーケンス図 | sd-DSL | $CSP_M$ |
|---|---|---|
|  | 1 @startuml<br>2 participant A<br>3 participant B<br>4 ====<br>5 hnote over A : Ready(v0:Int)<br>6 loop v0 < 10<br>7 A ->> B : m1(v0)<br>8 note over A : v0 = v0+1<br>9 end<br>10 hnote over A : Ready(0)<br>11 @enduml | 1 datatype Class_ = A \| B<br>2 datatype StateVar_ = A_Ready_v0.Int<br>3 datatype Method_ = B_m1.Int<br>4 channel msg_ : Class_.Class_.Method_<br>5 channel init_, sync_, update_ : StateVar_<br>6 SV_A_Ready_v0_Init = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>7 SV_A_Ready_v0(val) = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>8 [] update_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>9 [] sync_.A_Ready_v0!val -> SV_A_Ready_v0(val)<br>10 A_Ready_0() = sync_.A_Ready_v0?v0 -> if v0 < 10<br>11 then A_Ready_1(); A_Ready_0()<br>12 else A_Ready_4()<br>13 A_Ready_1() = sync_.A_Ready_v0?v0 -> msg_.A.B.B_m1!v0<br>14 -> A_Ready_2()<br>15 A_Ready_2() = update_.A_Ready_v0!(v0+1) -> A_Ready_3()<br>16 A_Ready_3() = SKIP<br>17 A_Ready_4() = init_.A_Ready_v0!(v0) -> A_Ready_0() |

**表 7　複合フラグメント (par) の $CSP_M$ への変換例**

| シーケンス図 | sd-DSL | $CSP_M$ |
|---|---|---|
|  | 1 @startuml<br>2 participant A<br>3 participant B<br>4 ====<br>5 hnote over A : Ready(v0:Int)<br>6 par<br>7 A ->> B : m1(v0)<br>8 else<br>9 A ->> B : m2(v0)<br>10 end<br>11 hnote over A : Ready(0)<br>12 @enduml | 1 datatype Class_ = A \| B<br>2 datatype StateVar_ = A_Ready_v0.Int<br>3 datatype Method_ = B_m1.Int \| B_m2.Int<br>4 channel msg_ : Class_.Class_.Method_<br>5 channel init_, sync_, update_ : StateVar_<br>6 SV_A_Ready_v0_Init = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>7 SV_A_Ready_v0(val) = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>8 [] update_.A_Ready_v0?v -> SV_A_Ready_v0(v)<br>9 [] sync_.A_Ready_v0!val -> SV_A_Ready_v0(val)<br>10 A_Ready_0() = (A_Ready_1() \|\|\| A_Ready_3()); A_Ready_5()<br>11 A_Ready_1() = sync_.A_Ready_v0?v0 -> msg_.A.B.B_m1!v0<br>12 -> A_Ready_2()<br>13 A_Ready_2() = SKIP<br>14 A_Ready_3() = sync_.A_Ready_v0?v0 -> msg_.A.B.B_m2!v0<br>15 -> A_Ready_4()<br>16 A_Ready_4() = SKIP<br>17 A_Ready_5() = init_.A_Ready_v0!(v0) -> A_Ready_0() |

へ変換する点において，機能 1 と基本的には同等である．具体的には，要件 1-1，1-2，1-3 と同等の要件が機能 2 に対しても存在する．ここでは，機能 2 に対する要件 1-1，1-2，1-3 と同等の要件をそれぞれ要件 2-1，2-2，2-3 とする．

基本的に要件 2-1，2-2，2-3 に対する設計方針は要件 1-1，1-2，1-3 に対する設計方針と同等であるが，要件 2-3 に対する設計方針において次のような違いがある．

- ガード無し alt，ガード無し opt を $CSP_M$ に変換する際に，外部選択 ([]) ではなく内部選択 (|~|) を用いる
- メッセージ受信時のイベントのフィールドタイプを決定的入力 (?) ではなく非決定的入力 ($) を用いる

これは，システム内部のイベントは外部からは観測できないため，システム外部とシステムとの相互作用を外部視点で表現する外部仕様としては，上記のようなケースに対しては非決定的な振る舞いとして定義するのが適当であると考えたためである．

### 3.4　反例情報からシーケンス図への変換 (機能 3)

本節では，機能 3 である，反例情報からシーケンス図への変換について，要件と要件に対する設計方針およびその実装について説明する．

機能 3 に対する要件としては次の 3 つが挙げられる．

**要件 3-1**

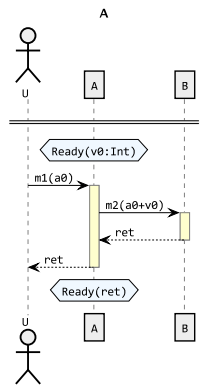
反例情報 (YAML 形式) から sd-DSL 形式へ変換できること

**要件 3-2**

反例情報から異常動作だけでなく，期待動作のシーケンス図も表示できること

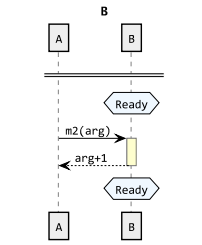
表 8　内部仕様合成の $CSP_M$ への変換例

シーケンス図




sd-DSL

```
1  @startuml
2  title A
3  Actor U
4  participant A
5  participant B
6  ====
7  hnote over A : Ready(v0:Int)
8  U -> A ++ : m1(a0)
9  A -> B ++ : m2(a0+v0)
10 return ret
11 return ret
12 hnote over A : Ready(ret)
13 @enduml
```




```
1  @startuml
2  title B
3  participant A
4  participant B
5  ====
6  hnote over B : Ready
7  A -> B ++ : m2(arg)
8  return arg+1
9  hnote over B : Ready
10 @enduml
```

$CSP_M$

```
1  datatype Class_ = U | A | B
2  datatype Method_ = A_m1.Int | return_A_m1.Int
3                   | B_m2.Int | reutrn_B_m2.Int
4  datatype StateVar_ = A_Ready_v0.Int
5  channel msg_ : Class_.Class_.Method_
6  channel init_, sync_, update_ : StateVar_
7  SV_A_Ready_v0_Init = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)
8  SV_A_Ready_v0(val) = init_.A_Ready_v0?v -> SV_A_Ready_v0(v)
9                     [] update_.A_Ready_v0?v -> SV_A_Ready_v0(v)
10                    [] sync_.A_Ready_v0!val -> SV_A_Ready_v0(val)
11 A_Ready_0() = msg_.U.A.A_m1?a0 -> A_Ready_1()
12 A_Ready_1() = sync_.A_Ready_v0?v0 -> msg_.A.B.B_m2!v0
13              -> A_Ready_2()
14 A_Ready_2() = msg_.B.A.return_B_m2?ret -> A_Ready_3(ret)
15 A_Ready_3(ret) = msg_.A.U.return_A_m1!ret -> A_Ready_4(ret)
16 A_Ready_4(ret) = init_.A_Ready_v0!(ret) -> A_Ready_0()
17 B_Ready_0() = msg_.A.B.B_m2?arg -> B_Ready_1(arg)
18 B_Ready_1(arg) = msg_.B.A.return_B_m2!(arg+1) -> B_Ready_2(arg)
19 B_Ready_2(arg) = B_Ready_0()
20 getSv_(A) = Union({productions(A_Ready_v0)})
21 getSvCh_(x) = {init_.s, sync_.s, update_.s | s<-getSv_(x)}
22 getProc_(A) = A_Ready_0() [|getSvCh_(A)|] SV_A_Ready_v0_Init
23              \ getSvCh_(A)
24 getProc_(B) = B_Ready_0()
25 getProcIf_(x) = Union({
26                   {msg_.x.t.m|t<-diff(Class_, x), m<-Method_},
27                   {msg_.f.x.m|f<-diff(Class_, x), m<-Method_}})
28 InternalSystem_ = || x:Class_ @ [getProcIf_(x)] getProc_(x)
```

**要件 3-3**

異常動作と期待動作の差異を明示できること

### 3.4.1　要件 3-1 設計方針

反例情報をシーケンス図に変換するために，反例情報を sd-DSL 形式に変換できる必要がある．提案ツールでは，FDR3 が出力する YAML 形式の反例情報から違反箇所に至るまでのイベント列を取得し，イベント列のうち，メッセージの送受信と状態変数の更新を表すイベントを sd-DSL 形式に変換する方針とする．

表 9，に反例情報と変換後の sd-DSL およびシーケンス図の例を示す．表 9 の反例情報例は説明に必要な箇所のみを抜粋しており，実際の反例情報はもう少し複雑である．反例情報の 5 行目以降は，イベントの識別子となる整数 (以降，イベント ID と呼ぶ) とイベントの対応を定義している．そして 1 行目は検査モデルが検査仕様に違反しているイベント ID を示しており，2-4 行目が違反に至るイベント ID のトレース列を示している．

反例情報を元に sd-DSL へ変換するには，違反に至るイベント列の内容に従い変換していけば良い．たとえば反例情報の 6 行目は，オブジェクト `A` からオブジェクト `B` へメッセージ `m1` を引数の値を `0` として送信するイベントであることがわかるため，それをそのまま sd-DSL での表現に変換すればよい．違反している箇所については複合フラグメント neg を用いて囲み，メッセージを赤色で表示することとした．

### 3.4.2　要件 3-2 設計方針

反例情報から得た異常動作のシーケンス図だけでは，違反箇所に至る経緯や要因が把握しにくい可能性がある．そのため，課題 3 である反例情報の解析能力習得コストを低減するためには，異常動作だけでなく，期待動作のシーケンス図も表示できることが望ましい．

期待動作のシーケンス図を作成するためには，期待動作となる場合のシステム内部のイベント列を取得する必要がある．一方，FDR3 は検査仕様に違反する場合のイベント列を反例情報として出力する機能はあるが，違反しない場合，つまり期待動作である場合のイベント列の情報を出力する機能は有していない．

提案ツールでは，内部仕様と外部仕様の整合性検査

**表 9　反例情報と変換後の sd-DSL およびシーケンス図の例**

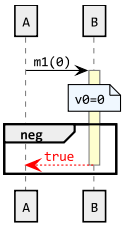
反例情報例 (抜粋)

```
error_event: 2
trace:
    - 0
    - 1
event_map:
    0: msg_.A.B.B_m1.0
    1: update_.B_Ready_v0.0
    2: msg_.B.A.return_B_m1.true
```

sd-DSL

```
@startuml
participant A
participant B
A -> B ++ : m1(0)
note over B : v0=0
group neg
    A <-[#red]- B -- : <color red>true</color>
end
@enduml
```

シーケンス図




の結果として取得した反例情報を，異常動作の振る舞い (外部仕様に違反する振る舞い) を行う外部仕様として位置づけ，その外部仕様と内部仕様との整合性検査を行うことで反例情報として期待動作 (外部仕様に違反しない振る舞い) のシステム内部のイベント列を取得する方針とした．

#### 3.4.3　要件 3-3 設計方針

違反箇所に至る経緯や違反の要因の把握を容易にするためには，期待動作のシーケンス図と異常動作のシーケンス図の差分が明確であり，比較しやすいように表示が調整されている必要がある．

期待動作のシーケンス図と異常動作のシーケンス図の差分を明確に表示するために，期待動作と異常動作の sd-DSL の差分箇所について，メッセージの色を変えハイライト表示する方針とする．また，期待動作のシーケンス図と異常動作のシーケンス図を，左右に並べて差分が確認できるように，メッセージの表示位置が一致するようにメッセージの間隔を調整する方針とする．

## 4　実装

本章では，3 章で述べた設計方針に従い機能 1，2，3 を実装したツールについて述べる．

### 4.1　機能 1 および機能 2 の実装

3.3 小節で説明したとおり，機能 1 と機能 2 は，sd-DSL を $\mathrm{CSP_M}$ へ変換するという点において一部の設計方針を除き基本的には同等の機能の実装を必要とするため，実際には 1 つのツールとして実装されている．機能 1 と機能 2 を実装したツールは，内部仕様を表現する PlantUML の sd-DSL 形式のファイルと外部仕様を表現する PlantUML の sd-DSL 形式のファイル，また定義情報を記述した YAML 形式のファイルを入力にとり，3.2 節および 3.3 節で説明した設計方針に従い，3.1.2 小節で述べた内部仕様と外部仕様の整合性が FDR3 を用いて検査することができる $\mathrm{CSP_M}$ の検査モデルおよび検査仕様に変換したのち，結果となる $\mathrm{CSP_M}$ をファイルに出力する機能を提供する．実装言語は Python [9] であり，空行を除いた有効行数は 600 行となった．

### 4.2　機能 3 の実装

機能 3 は，要件 3-1 に対応するツールと，要件 3-3 に対応するツールの 2 つツールにより実装されている．要件 3-2 についてはツールによる自動化に対応せず，設計者が手動で対処する方針とした．

要件 3-1 に対応するツールは，FDR3 が出力する YAML 形式の反例情報を入力にとり，3.4.1 小節で説明した設計方針に従い PlantUML の sd-DSL に変換したのち，結果となる sd-DSL をファイルに出力する機能を提供する．実装言語は Python で空行を除いた有効行数は 122 行となった．

要件 3-3 に対応するツールは，期待動作の sd-DSL と異常動作の sd-DSL を入力にとり，差分となるメッセージや状態変数の更新箇所を色づけてハイライトし，差分前後のメッセージや状態変数の更新箇所の表示位置が一致するようにメッセージの間隔を調整した sd-DSL を出力する機能を提供する．実装言語は Python で空行を除いた有効行数は 28 行となった．

## 5　評価

本章では，提案ツールの評価として，2.1 節で説明した業務で実際に検出した不具合事例に提案ツールを適用することで目標の達成を確認する．

### 5.1　機能 1 と機能 2 に対応するツールの適用

内部仕様として，業務モデルのシステム内部の各オブジェクトのシーケンスを PlantUML の sd-DSL 形式で記述した．図 6 は，業務モデルの `IpcServer` というオブジェクトの内部仕様の記述例である．業務モデルのすべてのオブジェクトの内部仕様 sd-DSL の空行を除く有効行数の合計は 190 行となった．記述した sd-DSL は PlantUML によりシーケンス図に作図できることを確認している．sd-DSL を記述する際，シーケンス図を確認しながら作図できるため，内部仕様の妥当性を視覚的に確認しながら記述することができることを確認した．

同様に，業務モデルの外部仕様として，図 2 で示した業務モデルの期待動作のシーケンスを PlantUML の sd-DSL 形式で記述した．図 7 に業務モデルの外部仕様を示す． 外部仕様 sd-DSL の空行を除く有効行数は 62 行となった．記述した sd-DSL は PlantUML によりシーケンス図に作図できることを確認している．こちらも内部仕様と同様にシーケンス図を確認しながら作図できるため，外部仕様の妥当性を視覚的に確認しながら記述することができることを確認した．

機能 1，機能 2 では内部仕様と外部仕様で現れるメッセージの型情報を定義情報として別途ツールに入力することで，検査モデルでの式の評価を可能としている．業務モデル用に記述した定義情報の空行を除く有効行数は 70 行となった．

業務モデルの内部仕様と外部仕様および定義情報を 4.1 節で述べた sd-DSL を $\mathrm{CSP_M}$ に変換するツールに入力し，検査モデルと検査仕様の $\mathrm{CSP_M}$ が得られることを確認した．出力された $\mathrm{CSP_M}$ の空行を除く有効行数は 381 行となった．381 行のうち，内部仕様である検査モデルの記述に該当するのは約 320 行，外部仕様である検査仕様に該当するのは約 80 行であった．

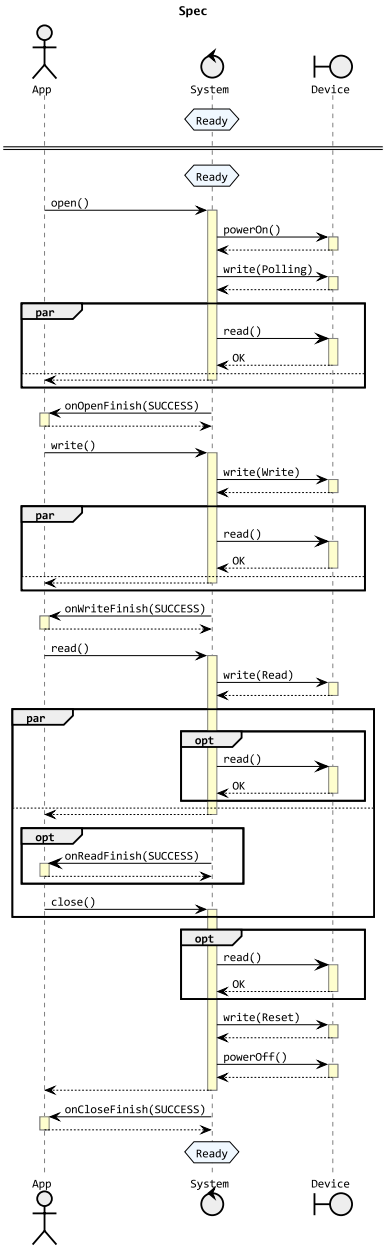



**図 7　業務モデルの外部仕様**

### 5.2　FDR3 による内部仕様と外部仕様の整合性検査

5.1 節で述べた検査モデルと検査仕様の $\mathrm{CSP_M}$ を FDR3 に入力し，内部仕様と外部仕様の整合性検査の結果として YAML 形式の反例情報が得られることを確認した．FDR3 の検証に要した時間は，CPU：

1.8GHz Intel Core i7，メモリ：4GB の環境で 0.745 秒であった．

### 5.3 要件 3-1 に対応するツールの適用結果

5.2 節で得られた YAML 形式の反例情報を，4.2 節で説明した，反例情報を sd-DSL に変換するツールにより入力することで，異常動作の sd-DSL が出力されることを確認した．異常動作の sd-DSL の空行を除く有効行数は 146 行であった．異常動作の sd-DSL は PlantUML を用いてシーケンス図に変換できることを確認している．

異常動作のシーケンス図は，外部仕様の違反箇所がシーケンス図の複合フラグメント neg によりハイライト表示されていることを確認しており，図 3 で示した異常動作のシーケンス図の内容と一致していることを確認した．

### 5.4 要件 3-2 の実施結果

要件 3-2 については，4.2 節で述べたとおり，手動で異常動作に含まれる違反箇所を外部仕様に取り込むことで対応する．具体的には，5.1 節で記述した外部仕様 sd-DSL を元に，5.3 節で取得した異常動作の sd-DSL から違反箇所の sd-DSL の記述を取り込み，反例情報として期待動作のイベント列を取得することができる外部仕様の sd-DSL を作成した．作成した期待動作取得用の外部仕様の sd-DSL と内部仕様の sd-DSL を，4.1 節で述べたツールにより $\mathrm{CSP_M}$ へ変換したのち，FDR3 を用いて検査することで期待動作を表す反例情報が出力され，その反例情報を 4.2 節で述べた要件 3-1 に対応するツールを用いることで期待動作の sd-DSL に変換できることを確認した．作成した期待動作の sd-DSL は，PlantUML を用いてシーケンス図を作図できることを確認している．期待動作のシーケンス図は，図 2 で示した期待動作のシーケンス図の内容と整合していることを確認した．

### 5.5 要件 3-3 に対応するツールの適用結果

5.3 節と 5.4 節により得られた，期待動作の sd-DSL と異常動作の sd-DSL を，4.2 で説明した期待動作と異常動作の sd-DSL の差分をハイライトし，メッセージの間隔を調整するツールに入力することで，差分が明確になるように調整された sd-DSL が出力されることを確認した．差分がハイライトされた sd-DSL は，PlantUML によりシーケンス図を作図できることを確認している．表 10 に差分がハイライトされた期待動作と異常動作のシーケンス図を示す．違反箇所が赤色に，両者の差分が青色にハイライトされ，また両者の同じメッセージの位置が一致するようにメッセージの間隔が調整され，期待動作と異常動作の振る舞いを容易に比較できることがわかる．これにより，期待動作と異常動作の差異が，`Executor` から `CallbackThread` への `sendResult(READ, CANCELED)` メッセージの呼び出しタイミングに起因することが容易に判断できる．

### 5.6 提案ツールを適用しなかった場合との比較

本節では，提案ツールを適用しなかった場合と適用した場合について課題毎に比較し，提案ツールの効果を確認する．

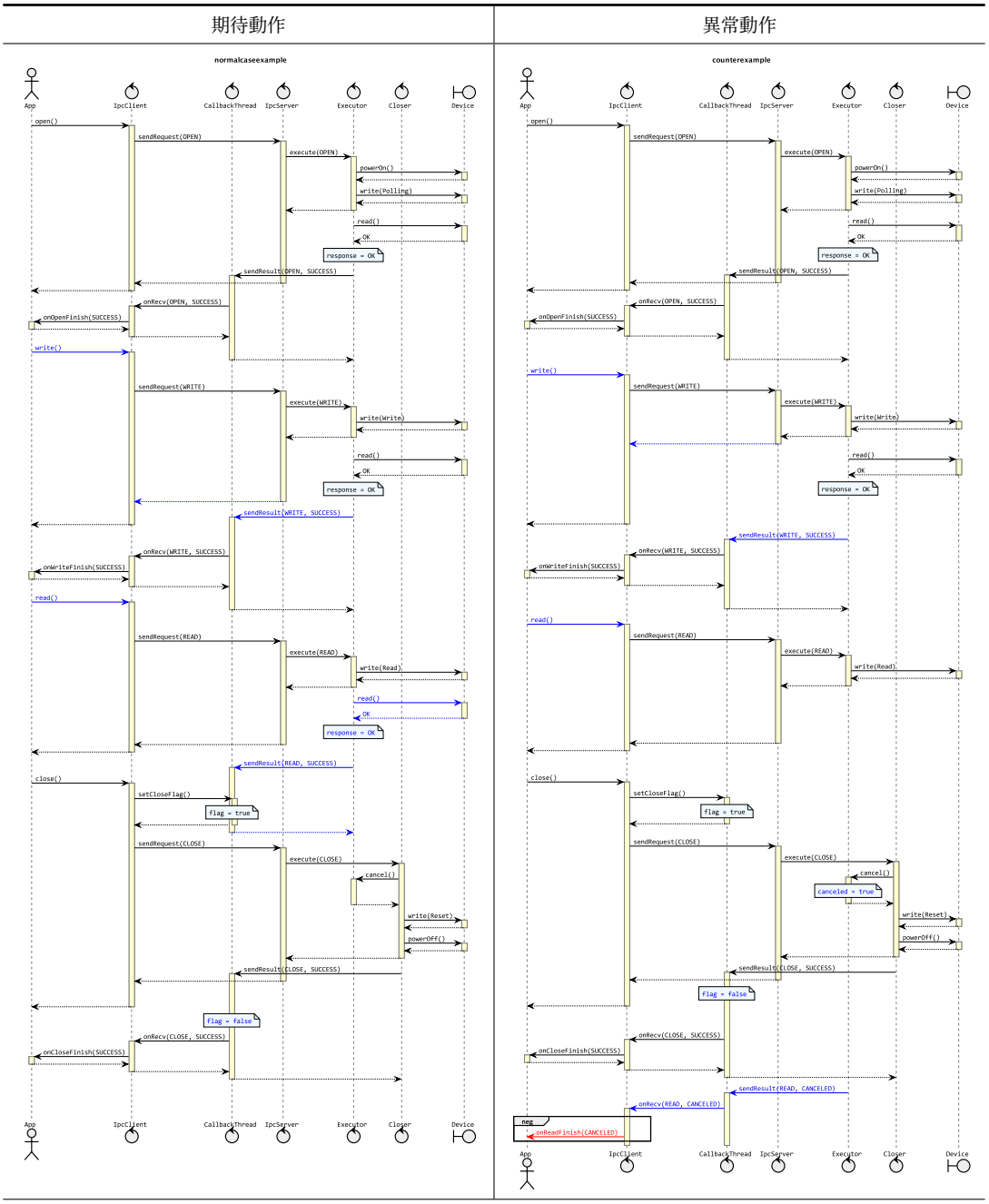
#### 5.6.1 検査モデルの記述 (課題 1)

提案ツールを適用しない場合，内部仕様である $\mathrm{CSP_M}$ を直接記述する必要があり，$\mathrm{CSP_M}$ の学習コストがかかる．また，$\mathrm{CSP_M}$ の記述の妥当性を確認する方法としては $\mathrm{CSP_M}$ の有識者によるレビューしか方法がなく，プロジェクト内に $\mathrm{CSP_M}$ 有識者が複数人いることが必要となる．

一方で，提案ツールを適用した場合は，内部仕様を sd-DSL で記載するため，sd-DSL の記法を学習する必要はあるが，sd-DSL の記法はシーケンス図の構成要素と対応がとれており，作図しながら sd-DSL の記載内容が確認できるため比較的学習が容易である．また，内部仕様がシーケンス図として表現されるため，妥当性を確認するために sd-DSL を習得する必要はなく，一般的な設計者によるレビューが可能である．

内部仕様の記述量を比較すると，5.1 節で述べたとおり，内部仕様を $\mathrm{CSP_M}$ で直接表現する場合は約 320 行の記述が必要であるが，sd-DSL と定義情報で記述する場合は，それぞれ 190 行と 70 行の記述で済むため，提案ツールを適用した場合のほうが検査モデルの記述コストを下げられることがわかる．

表 10　提案ツールが出力する期待動作と異常動作のシーケンス図の差分表示の例




### 5.6.2　検査仕様の記述 (課題 2)

検査仕様を記述する場合においても，5.6.1 小節と同様の理由で，提案ツールを適用せずに $\mathrm{CSP_M}$ を直接記述する場合に比べて，提案ツールを適用して sd-DSL を記述した場合のほうが学習コストや妥当性確認の容易性，記述コストの面で優位性があるといえる．

実際に外部仕様の記述量を比較すると，外部仕様を $\mathrm{CSP_M}$ で直接表現する場合は約 80 行の記述が必要であるが，sd-DSL と定義情報で記述する場合は，定義

情報は内部仕様と共有であるため sd-DSL の 62 行の記述で済み，提案ツールを適用した場合のほうが検査仕様の記述コストを下げられることがわかる．

#### 5.6.3 反例情報の解析 (課題 3)

提案ツールを適用しない場合，FDR3 が出力する反例情報を直接解析する必要がある．図 8 に，業務モデルの検査モデルと検査仕様を FDR3 に適用し出力された反例情報の表示例を示す．

FDR3 が出力する反例情報は，FDR3 独自の形式で表示される．具体的には，縦にプロセスの一覧が並び，横にイベントが，発生したプロセス上に発生した順で並んで表示される．この表示形式の問題点は，すべてのプロセス間のメッセージのやりとりが平坦に表示されるため，メッセージがどのプロセス間でやりとりされているのか一見しただけではわからないことである．そのため，検査対象が複雑なモデルになればなるほど，反例を解析する場合に違反箇所に至る経緯を推測するのが非常に困難になる．実際に，図 8 では業務モデルの反例情報を FDR3 で表示しているが，膨大なイベント列が平坦に表示されているため，違反箇所に至る要因をこの反例情報から解析するのが困難であることがわかる．また，反例情報として仕様に違反する異常動作のみが表示されるため，期待動作となる場合の検査モデルの振る舞いが明確でない状態で解析する必要があることも，解析を困難にする要因となっている．

一方で提案ツールを適用した場合は，表 10 で示したとおり，反例情報が馴染みのあるシーケンス図で表示されるため，直感的，視覚的に違反に至る経緯を把握することができる．さらに，期待動作と異常動作を比較しながら確認することができ，また違反箇所や期待動作と異常動作の差分がハイライトされるため，違反箇所に至る要因を容易に解析することができる．

### 5.7 評価結果のまとめ

5.1 節から 5.5 節にて，業務モデルに対し提案ツールを適用した場合の効果を確認した．また，5.6 節にて，提案ツールを適用しなかった場合と比較することで，提案ツールにより外部仕様と内部仕様の不整合に起因して発生する再現性の低い不具合事例が，学習コストを抑えながら効果的に検出・解析できることを確認した．以上より，2.1 節で掲げた提案ツールの実現により達成するべき目標が達成できたと判断した．

## 6 おわりに

本論文では，はじめに，著者がデバイス制御ミドルウェアの開発業務経験より，再現性の低い不具合を効果的に検出することが困難であることに問題意識をもっていることを示し，モデル検査を業務に適用する場合に 3 つの課題があることを説明した．そして，これらの問題意識と課題より，提案ツールの実現により達成するべき目標と要求される機能について述べた．次に，要求される機能毎に求められる要件を説明し，それぞれの要件毎に設計方針としてどのような方法で実現したのかを述べた．また，設計方針に従った提案ツールの実装について述べた．最後に，提案ツールを業務モデルに適用した場合の効果と，適用しなかった場合との比較を説明し，提案ツールの有効性を示した．結果として，提案ツールを適用することで，外部仕様と内部仕様の不整合に起因して発生する再現性の低い不具合事例が，学習コストを抑えながら効果的に検出・解析できることを確認した．

提案ツールに関する今後の展開としては，4.2 節でツールではなく手動で対応すると述べた要件 3-2 を自動化することが挙げられる．要件 3-2 の作業をツールにより自動化するためには，違反箇所が外部仕様のどの箇所で発生しているかをツールが判断できるようにする必要があるため，外部仕様の各メッセージに識別番号を振るといった修正が必要になると考えている．また，本論文により業務モデルの不具合事例が効果的に検出・解析できることは確認できたが，他の事例に対しても有効であるか，つまり，汎用的に利用できるツールとなっているかについては未確認であるので，他の事例に適用し効果を測定することも有益であると考えている．

## 参 考 文 献

[1] Ben-Kiki, O., Evans, C., and Ingerson, B.: YAML Ain't Markup Language (YAML) $^{TM}$Version 1.2, Technical report, YAML.org, 2009.

[2] Crockford, D.: JSON: Javascript object nota-

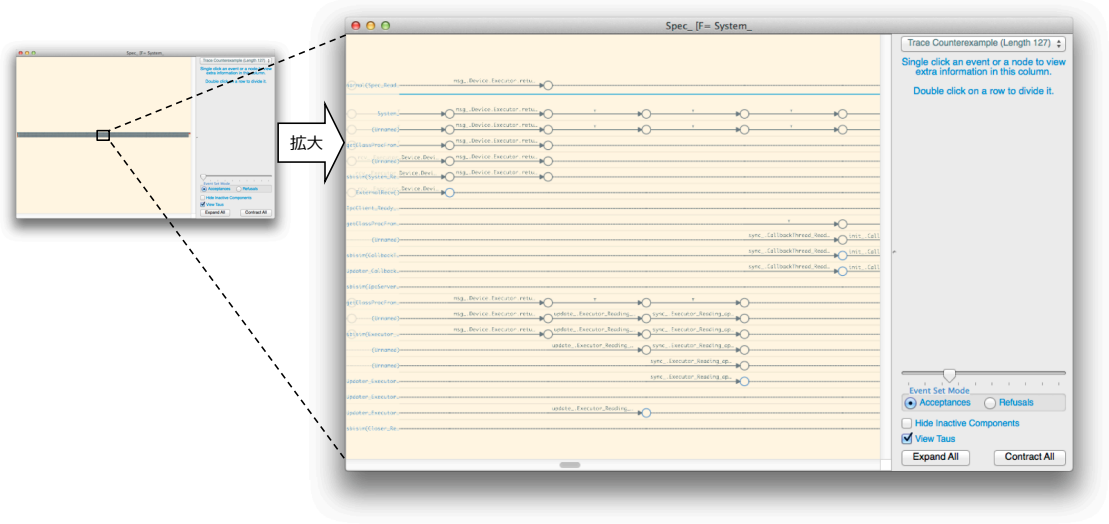



**図 8　FDR3 が出力する業務モデルの反例情報の表示例**

tion, 2006.

[3] Hoare, C. A. R.: *Communicating Sequential Processes*, Prentice Hall, 1985.

[4] Object Management Group, inc.: Unified Modeling Language (UML): Infrastructure specification, version 2.4.1, 2011.

[5] Object Management Group, inc.: Unified Modeling Language (UML): Superstructure specification, version 2.4.1, 2011.

[6] Roques, A.: PlantUML, `http://plantuml.sourceforge.net`, 2013.

[7] Roscoe, A. W.: *The Theory and Practice of Concurrency*, Prentice Hall, 1997.

[8] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. E., et al.: *Object-oriented Modeling and Design*, Vol. 199, No. 1, Prentice-hall Englewood Cliffs, NJ, 1991.

[9] Sanner, M.: Python: A Programming Language for Software Integration and Development, *J. Mol. Graphics Mod*, Vol. 17(1999), pp. 57–61.

[10] Scattergood, B.: *The Semantics and Implementation of Machine-Readable CSP*, PhD Thesis, Oxford University, 1998.

[11] Thomas Gibson-Robinson, Philip Armstrong, A. B. A. R.: FDR3 — A Modern Model Checker for CSP, *Tools and Algorithms for the Construction and Analysis of Systems*, brahm, E. and Havelund, K.(eds.), Lecture Notes in Computer Science, Vol. 8413, 2014, pp. 187–201.

[12] University of Oxford: Failures-Divergence Refinement – FDR 3 User Manual, `https://www.cs.ox.ac.uk/projects/fdr/manual/`, 2013.

[13] 後藤隼弐: csp-seq, `https://github.com/goto-junji/csp-seq`, 2014.