状態遷移モデルの一部が明示されていない場合における自動テスト生成手法の提案

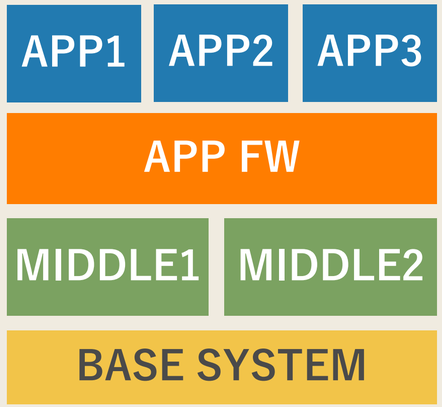
2022/02/25
研究コース5 人工知能とソフトウェア品質
テスト自動生成グループ

研究員：松尾　正裕（パナソニックITS株式会社）
主　査：石川　冬樹（国立情報学研究所）
副主査：栗田　太郎（ソニー株式会社）
副主査：徳本　晋　（富士通株式会社）

AGENDA

1. 背景・動機
2. 課題・提案手法
3. 実験とその結果
4. 考察
5. 今後の展望

背景・動機



(組込系の)分業開発において
影響の大きい不具合を
自動テストにより検出する

状態遷移テスト

APP

開発中の不具合流出を防ぐ
他機能への影響を踏まえて
不具合を**早期発見**する

APP単体のテスト

※開発者が実施するテストとは別に実施

APP — 頻繁に仕様変更、修正
状態遷移モデルの記載なし

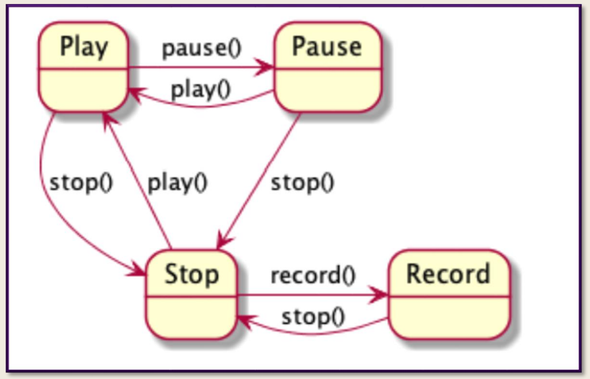
APP FW — 共通機能、ドキュメント充実
状態遷移モデルの記載あり

機能の開発特性によって
設計ドキュメントの
記載内容・充実度が異なる

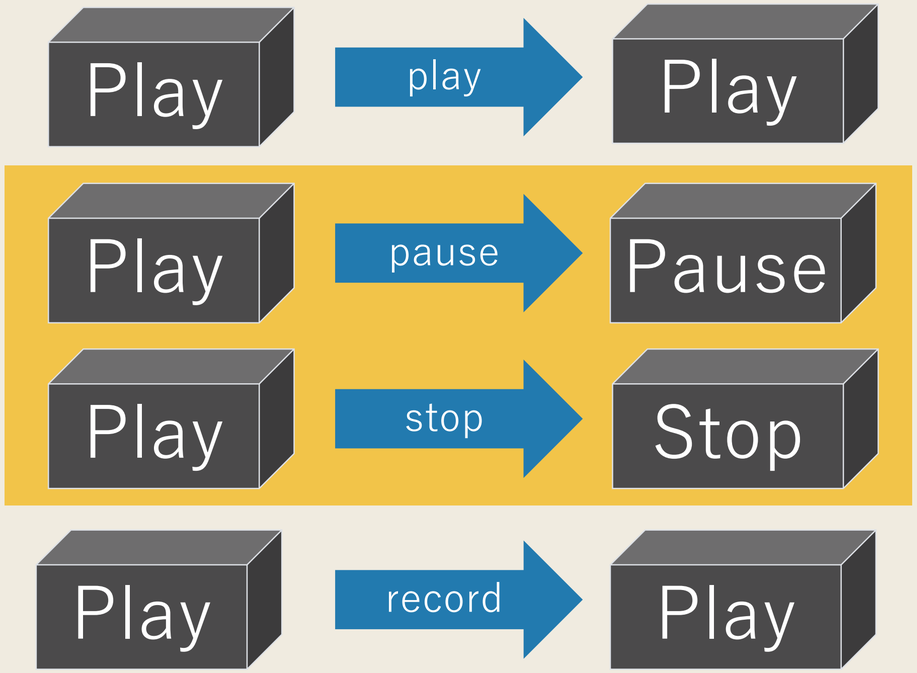
状態遷移モデルを部分的に活用した自動テスト生成

状態遷移テストの自動生成

状態遷移モデルが**明示されていない場合に**、**自動テスト生成**を行うことを考える
総当たりで関数呼び出しやイベント入力を行う



録音アプリ



総当たりでコマンド入力すれば
状態遷移する

状態遷移するテストケースは
全体の一部のみ

入力コマンドだけでは
テスト結果の検証はできない

課題

本取組において、**解決すべき課題は下記の2点**

1. テストオラクル問題

アプリの状態遷移モデルが
明示されていないため、
テストケースの期待値が不明

開発者がテストケース毎に
期待値を記載すればよいが
量的に現実的ではない

2. テストケースの有効性

コマンドの総当たりにより
状態遷移テストは**生成可能**

状態遷移しないテストを
大量実施することになるので
テスト効率が悪い

解決策(提案手法)

課題に対するそれぞれの解決策．**既知の状態遷移モデルを活用する**

1. テストオラクル問題 → → テストケース非依存の検査機構

既知の状態遷移モデルを用いて作成

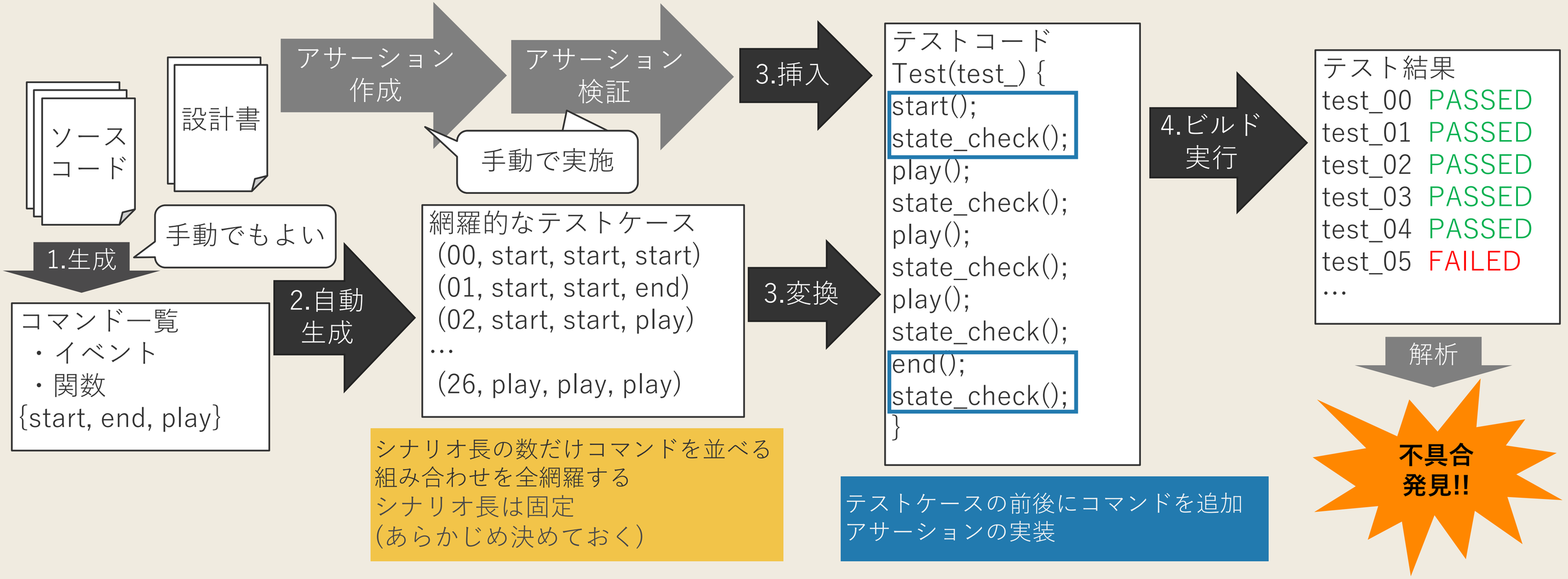
全てのテストケースで使用できる**共通アサーション**を作成
常に満たすべき性質から**外れたら不具合**と判定
ex）「起動状態になったら、いつかは必ず終了状態になる」というアサーションを作成
→**終了状態に遷移しなかったら不具合**と判定

2. テストケースの有効性 → → 既知の状態遷移モデルを活用

状態遷移する可能性の高いコマンドを生成した**テストケースの前後に追加**
ex）各テストケースの前に「**アプリ起動**」コマンドを追加
→アプリ起動前にアプリのコマンドを入力するケースを排除

提案手法（全体像）

先ほど説明した解決策を取り込んだ提案手法を提示する



実験

提案手法に基づいて実際にテストを実施することで、**本手法の有効性を確認**した
実験は２つ実施した

実験内容：

- **状態遷移モデル**を定義する
- **不具合**を**1件混入**させる
- **自動生成したテストコード**を実行して、**テスト結果**(**PASSED**/**FAILED**)を確認する

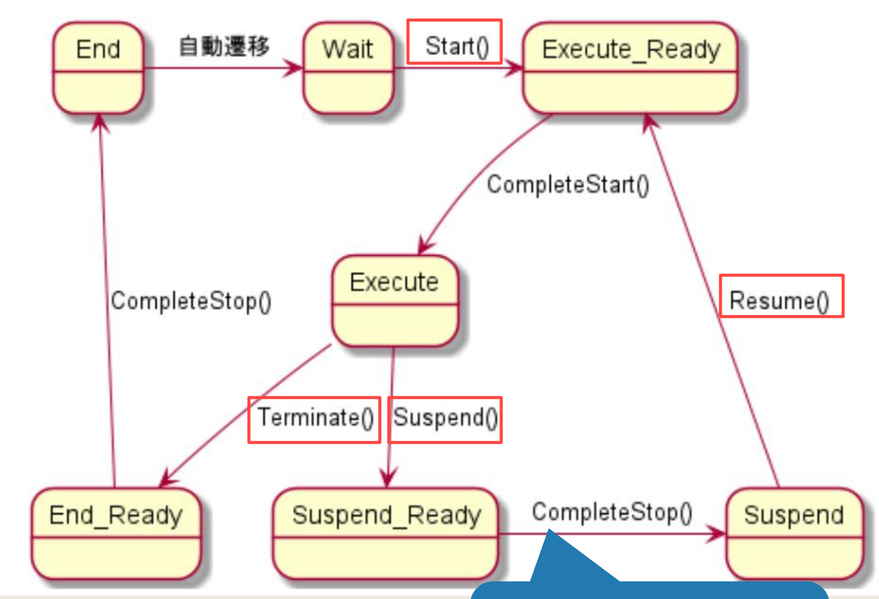
検証内容：

RQ. 1　作成した**アサーション**により**不具合を検出できる**こと
RQ. 2-1 網羅的に生成した**テストケース**により、**不具合が顕在化**すること
RQ. 2-2 テストケースの**絞り込み**により、不具合が**顕在化しやすくなる**こと

実験1:実験内容

アプリフレームワークの状態遷移モデルに対してコマンド網羅による自動テスト生成
Startから始まり**Terminate**で終了する**テストケースを抽出**し、**全体と比較**する

状態遷移モデルと混入不具合



テストケースの生成：

コマンド4つ{Start, Terminate, Suspend, Resume}を5回網羅的に並べる．全部で4^5=**1,024通り**

(Start, Start, Start, Start, Start)
(Start, Start, Start, Start, Terminate)
(Start, Start, Start, Start, Suspend)
(Start, Start, Start, Start, Resume)
(Start, Start, Start, Terminate, Start)
…
(Resume, Resume, Resume, Resume, Resume,)

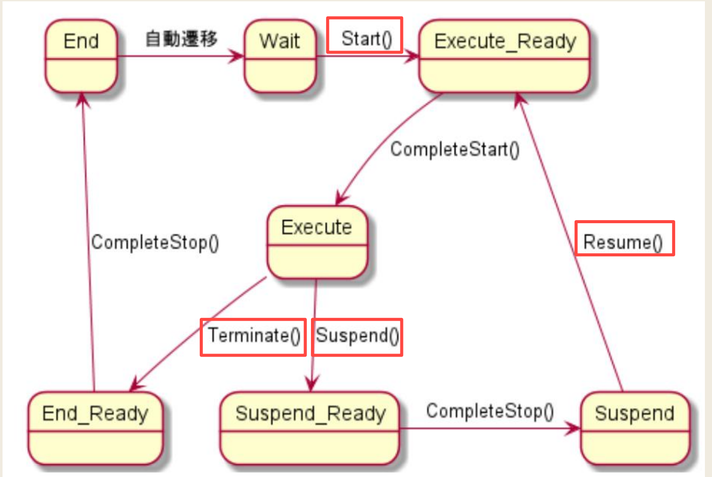
1,024通り

実験1：共通アサーションの作成(解決策1)

共通アサーションを作成する．実験2でも同じアサーションを使用する

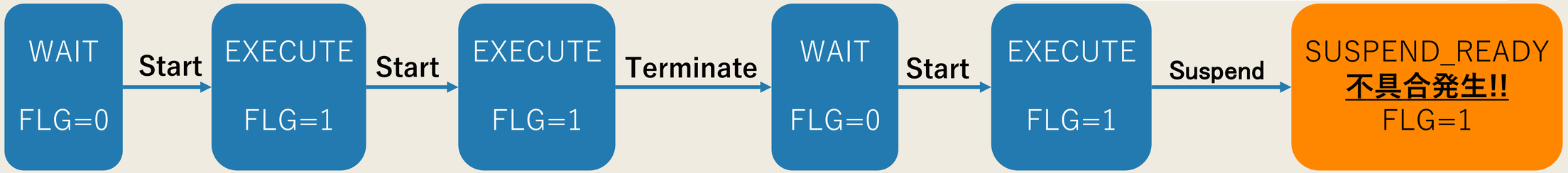
Assert.1: Execute 状態になったら，いつかは Suspend 状態になる　（**公平性を仮定**）
Assert.2: Execute 状態になったら，いつかは Suspend 状態または Wait 状態になる

※モデル検査でLTLの有効性を検証したものを使用する



テストコードにおけるアサーションの実装:
コマンド入力毎に状態を確認して**フラグ**を利用することで判定

テストケース(Start, Start, Terminate, Start, Suspend)
Assert.2を実装



実験1:テストケースの絞り込み(解決策2)

APP FWの状態遷移モデルを活用したテストケースの生成例：

テストケースの**前にStart, 後にTerminate**を挿入する場合

(Start, Start, Start, Start , Terminate)
(Start, Start, Start, Terminate, Terminate)
(Start, Start, Start, Suspend, Terminate)
(Start, Start, Start, Resume, Terminate)
(Start, Start, Terminate, Start, Terminate)
…
(Start, Resume, Resume, Resume, Terminate)

64通り

固定　網羅的に生成　固定

単純
網羅
1024件

絞込
64件

**アプリのコマンドも含めると
絞込みの削減効果大**

実験1の結果

共通アサーションにより、不具合を検出することができた（RQ.1）

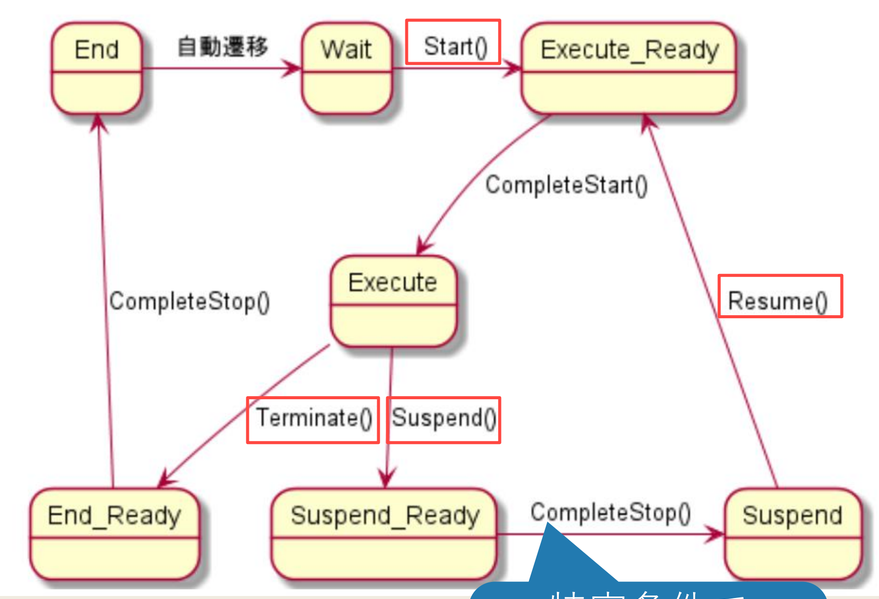
テストケース絞り込みにより、不具合が顕在化しやすくなった（RQ.2）

メトリクス	全体	抽出ケース
テストケース数	1,024	64
不具合顕在化数	299	29
不具合密度	0.292	0.453
Assert.1精度	0.383	0.453
Assert.2精度	0.522	1.000

実験2：実験内容

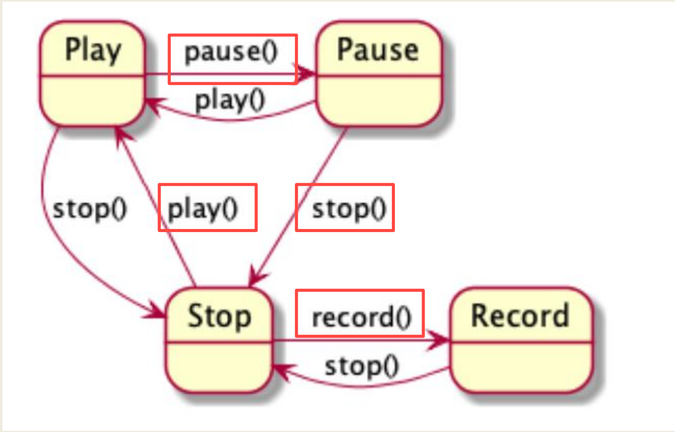
アプリの状態遷移モデルも含めてテストする．アサーションは実験1と同じ．
アプリの状態遷移モデルを定義するが、ブラックボックスとして扱う

状態遷移モデルと混入不具合



特定条件で
応答を返さない
不具合を混入

アプリの状態遷移モデル



実験2では、
コマンド8個を5回呼び出す
テストケースを生成する

8^5=**32,768通り**

実験2で追加

コマンド8個：

{Start, Terminate, Suspend, Resume, **play, stop, pause, record**}

実験2：テスト絞り込み方法（テスト5種類）

実験2では、アプリを含めたテスト自動生成手法の検証とともに、
テスト絞り込み方法の違いについても確認するため、**5種類のテスト**を実施した

テスト 1: 関数 8 個を 5 回呼び出す全てのテストケース
テスト 2: テスト 1 の各テストケース＋**最後に Terminate**
テスト 3: テスト 1 の各テストケース＋**最後に Suspend**
テスト 4: **最初に Start**+テスト 1 の各テストケース＋**最後に Terminate**
テスト 5: **最初に Start**+テスト 1 の各テストケース＋**最後に Suspend**

実験2の結果

コマンド追加により、不具合が顕在化しやすくなったり、Assert.精度の向上が見込める

メトリクス	テスト1	テスト2	テスト3	テスト4	テスト5
テストケース数	32,768	32,768	32,768	32,768	32,768
不具合顕在化数	684	684	3582	2133	5851
不具合密度	0.021	0.021	0.109	0.065	0.179
Assert.1精度	0.050	0.050	0.541	0.086	0.373
Assert.2精度	0.065	1.000	1.000	1.000	1.000

Startの追加により不具合が顕在化しやすくなった
Terminateの追加は影響なし

コマンドの追加により、**Assert. 精度が向上**している
(最後に追加している影響)
公平性を仮定せずに成り立つAssert.のほうが**精度が向上する**

実験結果のまとめ

2つの実験により、各RQ.の内容を検証することができた

検証結果：

RQ. 1　作成した**アサーション**により**不具合を検出できた**

RQ. 2-1 網羅的に生成した**テストケース**により、**不具合が顕在化した**

RQ. 2-2 テストケースの**絞り込み**により、不具合が**顕在化しやすくなった**

既知の状態遷移モデルを用いて**テストケースを絞り込むことによる効果**

- **不具合が顕在化しやすくなる**
- **共通アサーションの精度向上**

考察

今回の実験により、次のことが考えられる

- **実験が成功した要因**
  混入不具合を事前に知っていて、狙ってアサーションを作成した
  実際には不具合の場所はわからないので工夫してアサーションを作成する

- **Assert.2 精度が1になっている要因**
  混入した不具合が1件であったことが要因と考える
  本実験の場合、suspend周りの不具合なら他の不具合でも検出できると考える

- **テスト絞り込み方法**
  実験2のテスト2では、最後にTerminateを追加したが不具合の顕在化に影響なし
  顕在化しやすさは、不具合の性質に依存すると考えられる
  アサーションの精度向上の方が見込みやすい(アサーションの特性はわかるので)

提案手法の課題

今回の実験は簡単なモデルを仮定したため、導入に向けた課題が多数ある
引き続き、導入に向けて検討していく予定

提案手法を導入する課題

- コマンド一覧の生成方法
- シナリオ長の決め方
- テストケース絞り込むためのコマンドの決め方
- 実際に導入した場合における効果の大きさ

今後の発展

提案手法をさらに発展させるための取組として、下記が考えられる

- **最適化によるテストケース絞り込み**
  進化計算アルゴリズムや量子アニーリングを用いたテストケースの絞り込みを想定
  コマンド列の制約により絞り込むことを考えると、「最適化手法」を用いることができる

- **クラスタリング分析**
  テストを大量に実施するので、テスト結果をクラスタリング分析することにより、
  品質を分析することが可能であると考えられる。

- **状態遷移テストにおける不具合の局所化**
  大量の同件不具合が検出されるので、自動で不具合箇所が特定できると解析効率が向上する

謝辞

本研究を行うにあたり，多くの方々にお世話になりました.
主査の**石川冬樹氏**，副主査の**徳本晋氏**，**栗田太郎氏**には，
本研究を遂行する上で様々なご指導や助言をいただきました.
深く感謝申し上げます.
また，本研究会への参加の機会をいただき，
ソフトウェア開発における課題についての情報・助言をいただいた
パナソニックITS(株)の皆様にも感謝致します.
最後に，お世話になりました全ての皆様に，感謝と御礼申し上げます.

ご清聴ありがとうございました

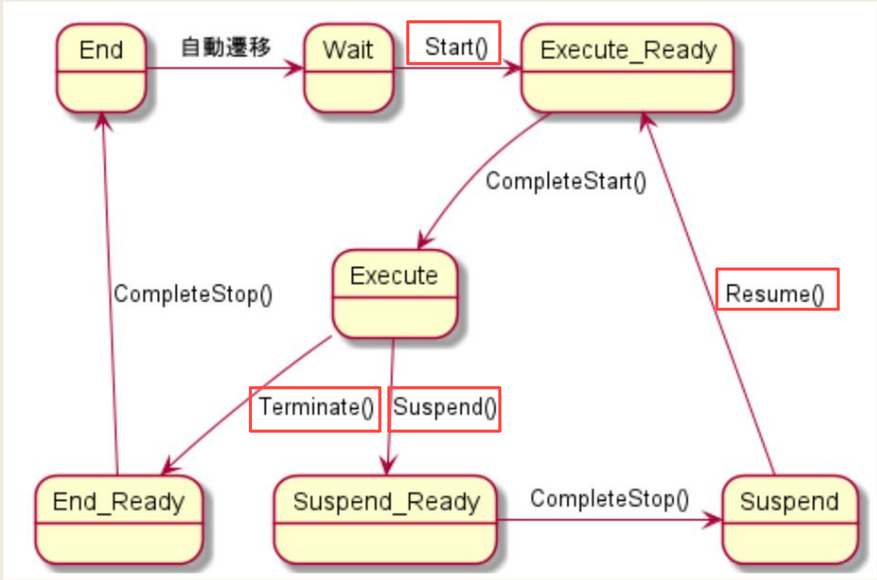
予備スライド

アサーションの生成方法

1. 既知の状態遷移モデルを用いて常に満たすべき性質を記述する
   ex) Execute 状態になったら，いつかは Suspend 状態または Wait 状態になる

2. モデル検査により、状態遷移モデルが1.の状態を常に満たしているかを検査する
   本研究ではモデル検査ツールspinを使用した
   必要に応じて公平性などの条件を設定する
   (今回はモデル検査の具体的な方法は省略する)

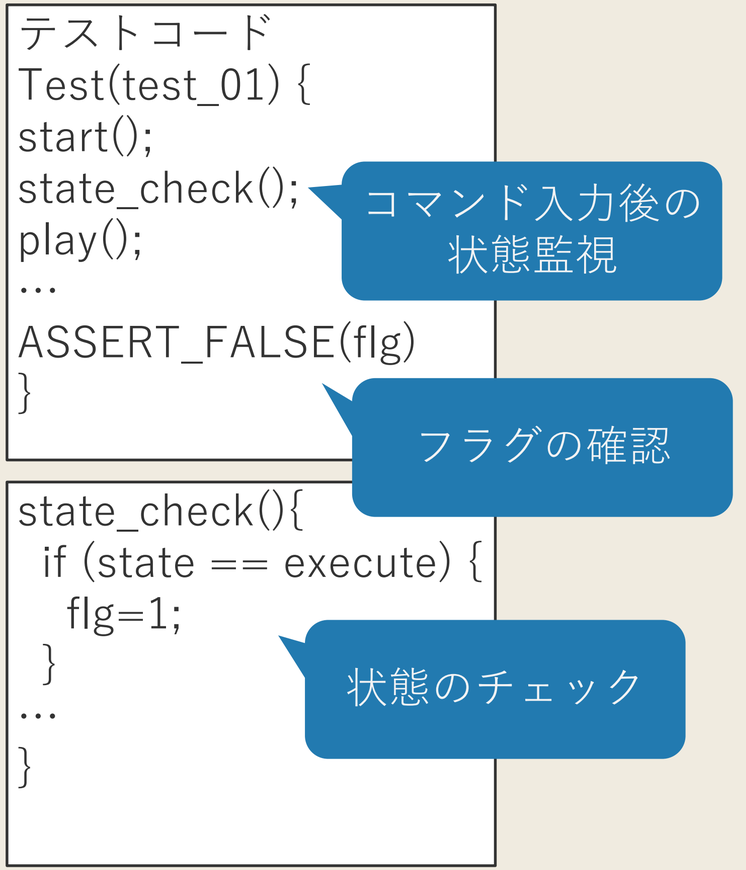
3. モデル検査による検証ができたら、
   テストコードとして実装する



アサーションの実装

アサーションを実装するには、**4つの仕掛け**が必要である

- **コマンド入力後の状態監視**
  コマンド入力後に「状態のチェック」を実施する
- **状態のチェック**
  現在の状態に応じてフラグの状態を変更する
  「常に満たすべき性質」に合わせて実装する
- **フラグ管理**
  テストコードの一部としてフラグを使用する
  スコープが広いので、気をつけてフラグを実装する
- **フラグの確認(アサーション)**
  フラグの中身を確認し、不具合を検出する
  テストの最後で確認する



テスト実施件数の見積もり

今回実施した環境において、テスト実施時間と実用化に耐える件数についての見積もり

- テスト実施環境
  Ryzen 9 5900, メモリ64GBのPCにて実施
  テストフレームワークはGoogleTest

- 実験2の実行速度
  テスト全件は32,768件
  テストケース生成からテスト実行まで2分程度
  160,000件/10分
  1,000,000件/1時間
  24,000,000件/1日

70個のコマンドを4回:70^4=24,010,000件
30個のコマンドを5回:30^5=24,300,000件
17個のコマンドを6回:17^6=24,137,569件