Vol.2016-SE-191 No.21
2016/3/15

LTSA 連携による不確かさを包容した自動モデル検査

中村 隼也[1,a)]　深町 拓也[1,b)]　鵜林 尚靖[1,c)]　細合 晋太郎[1,d)]　亀井 靖高[1,e)]

概要：ソフトウェア開発工程のさまざまな段階で発生する「不確かさ」に開発者は対処しなければならない．開発者はソフトウェアの品質を保障するために仕様を検査するが，不確かさが発生することにより検査が進まない，検査したものが変更されてしまい意味がなくなってしまうということが起こり得る．不確かさは一般に Unknown Unknown（未知の未知）と Known Unknown（既知の未知）に分類できる．本研究では，後者の既知の不確かさをモジュラーに管理できるインターフェース機構 Archface-U とその支援機構である iArch-U に自動モデル検査機能を追加して，不確かさが実装段階に存在してもそれを包容して設計を検査できる環境を提案する．

1. はじめに

ソフトウェア開発において，要求分析，仕様記述，実装など様々な段階で現れる「不確かさ」は避けられないものである．不確かさは大きく二つに分類することができる．一つは Unknown Unknown（未知の未知）であり，ソフトウェア開発のあらゆる段階で不確かさが顕在していない状態である．もう一つは Known Unknown（既知の未知）であり，不確かさが存在していることがわかっている状態である．本研究では後者の Known Unknown を対象にしており，以降の「不確かさ」は全て既知の未知とする．

ソフトウェアの開発において，ソフトウェアのリリース後に不具合が発覚することがないよう，開発者は設計の段階で設計に問題がないかを検査するが，実装段階で不確かさが発生し実装が変更されると検査した内容が通用しなくなる．不確かさの発生の度に検査を繰り返すのは現実的には難しく，ソフトウェアの十分な検証が欠けたままリリースされてしまうということが起こり得る．

本研究では，不確かさを抱えたまま実装段階で自動でモデル検査を行い，ソフトウェアの安全な開発を支援する環境の構築を提案する．そのために，不確かさを扱うことができ，設計と実装のトレーサビリティを保ちながら開発を行うことができるインターフェース機構 Archface-U [16] に自動モデル検査機能を追加する．

[1] 九州大学
Kyushu University
[a)] nakamura@posl.ait.kyushu-u.ac.jp
[b)] fukamachi@posl.ait.kyushu-u.ac.jp
[c)] ubayashi@ait.kyushu-u.ac.jp
[d)] hosoai@qit.kyushu-u.ac.jp
[e)] kamei@ait.kyushu-u.ac.jp

Archface-U はクラスが宣言しなければならないメソッドや，メソッドの呼び出し関係をアーキテクチャ設計として記述することができ，Java コードがその制約に従うことを保障する．したがって，Archface-U に記述された不確かなメソッドを含んだメソッドの呼び出し関係が設計が求める制約を満たすかをモデル検査し，制約を満たすことが保障されるならば，そのメソッドの呼び出し関係に従って実装される Java コードは制約を満たして実装されていることが保障される．このようにして，本研究ではモデル検査を用いて不確かさを抱えたまま安全にソフトウェアの開発を進めることができる環境の構築を目指す．

本論文では，2 章で不確かさとモデル検査の関連研究について述べる．3 章では不確かさを包容したインターフェース機構 Archface-U を紹介する．4 章では Archface-U とモデル検査器を用いた，不確かさを包容したモデル検査を提案する．5 章では 3，4 章で述べたアプローチをどのようにツールとして実装したかについて述べ，6 章ではツールに例題を適用して，使い方とどのようなことができるかを示す．7 章ではまとめと今後の研究の課題について述べる．

2. 関連研究

本章では，ソフトウェア開発における不確かさの関連研究と，モデル検査の関連研究について紹介する．

2.1 不確かさの関連研究

不確かさはソフトウェア開発の様々な段階で発生する．本節では，Diego らによるソフトウェア開発における不確かさの分類 [14] について述べ，本研究で扱う不確かさを分類する．その後，Famelis らによる不確かさを包容したモ

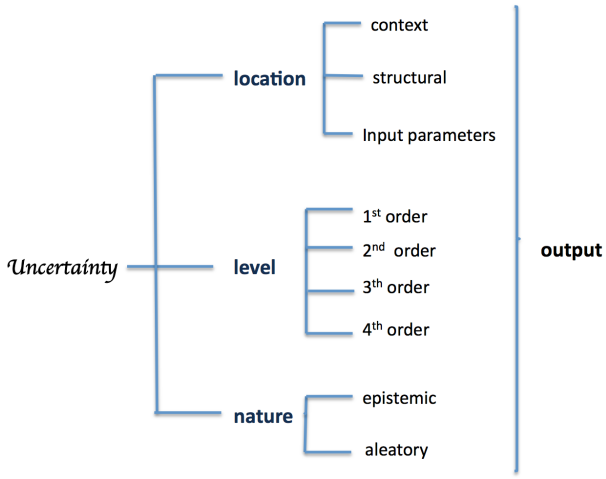

図 1 Diego らによる不確かさの分類
（元論文 [14]Figure 2. より引用）

デル Partial Model [8] を紹介する．

2.1.1 不確かさの分類

Diego らは不確かさを場所 (Location)，レベル (Level)，性質 (Nature) の 3 つの観点から分類した（図 1）．

- 場所 (Location): 不確かさがモデル中のどこに現れるかを表す．
  - コンテキスト: モデル化される情報に関する不確かさ
  - モデル構造: モデル自体の構造に関する不確かさ
  - 入力パラメータ: モデルに入力される変数の値に関する不確かさ．
- レベル (Level): すでにある知識と何もわかっていない知識との間をレベル付けしたものである．
  - レベル 0: 不確かさがない状態．すでにある知識．
  - レベル 1: 知識が欠けているが，それを認知している状態．既知の不確かさ．
  - レベル 2: 知識が欠けていて，それを認知していない状態．未知の不確かさ．
  - レベル 3: 認識が欠けていることを見つけるプロセスが欠けている状態．
  - レベル 4: 不確かさのメタ的な視点による不確かさ．
- 性質 (Nature): 不確かさが知識の不完全性によるものなのか，もしくは現象に固有の変動性によるものなのかを分類したものである．
  - 認識的 (Epistemic): 知識を獲得するためのデータの不足や，データから知識を獲得する手段がないことにより発生する不確かさ．
  - 偶発的 (Aleatory): イベント固有のランダム性により発生する不確かさ．

本研究では，不確かさが存在していることが明らかになっている場合を対象としており，その不確かさは開発が進むにつれて得られる情報により解消される．したがって本研究において扱う不確かさは以下のように分類できる．

- 場所: コンテキスト，モデル構造，入力パラメータ

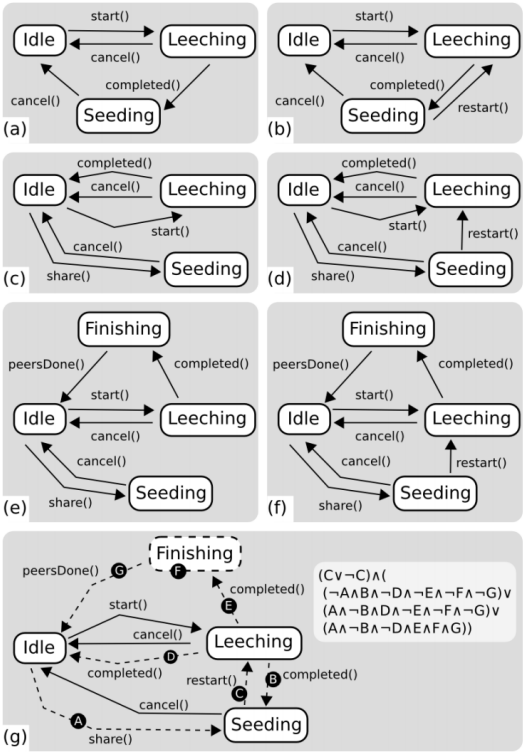

図 2 P2P ファイル共有システムにおける 6 つの設計候補 (a∼f) とそれらを統合した Partial Model(g)
（元論文 [8]Figure 1. より引用）

- レベル: レベル 0，レベル 1
- 性質: 認識的

本研究では，「不確かでない」，すなわち確定状態も扱うことができるため，レベル 0 も対象としてある．

2.1.2 Partial Model

Famelis らは設計の初期段階で起こりうる「設計候補がいくつか存在するが，どれを採用するかが決まっていない」という不確かさを扱っている [8]．いくつかの設計候補を一つのモデルとして表した Partial Model と呼ばれるモデルを用いて不確かさを表現する．Partial Model で扱う不確かさは本研究で扱う不確かさと同じ分類である．図 2 はある P2P ファイル共有システムの 6 つの設計候補 (a∼f) と，それらをまとめた Partial Model(g) の図である。

Partial Model を用いて設計候補がある性質を満たすかどうかを検査することができる．以後，ある性質をプロパティ，プロパティを満たすかどうかの検査をプロパティ検査と呼称する．Partial Model のプロパティ検査を行うには，まず Partial Model を論理式で表す．次に Partial Model を Φ_M，プロパティを Φ_P として，$\Phi_M \wedge \Phi_P$ 及び $\Phi_M \wedge \neg\Phi_P$ について SAT（充足可能性問題）ソルバで解析を行い充足可能性問題を解く．SAT ソルバの結果とプロパティ p の結果は表 1 のようになる．プロパティ p の True と False はそれぞれプロパティが充足，非充足を表す．Maybe は最終的に選択するモデルによってプロパティ

表 1 Partial Model M のプロパティ p の検査結果の表
（元論文 [8]Table 1. より引用）

$\Phi_M \wedge \Phi_P$	$\Phi_M \wedge \neg\Phi_P$	プロパティ p
SAT	SAT	Maybe
SAT	UNSAT	True
UNSAT	SAT	False
UNSAT	UNSAT	(error)

が成立するか否かが変化するということである．error は Partial Model を構成する状態遷移群が充足不能であり，設計自体の見直しが必要であることを表している．

プロパティ p の検査結果が False，または Maybe だった場合はそうなった原因を分析し，また，重要なプロパティが Maybe だった場合はプロパティに違反する候補を除外してプロパティが True となるように Partial Model を洗練し，どのような設計案がよいのかを導くことができる．

2.2 モデル検査の関連研究

モデル検査は検査対象のシステムの振る舞いを有限オートマトンとしてモデル化して，それがある性質を満たすかどうかをモデルの全ての遷移を探索することでシステムの振る舞いを検査する．

2.2.1 様々なモデル検査器

SPIN(Simple Promela Interpreter) [1]，PRISM [10]，LTSA(Labelled Transition System Analyser) [11]，など様々なモデル検査器が現在開発されている．

SPIN(Simple Promela INterpreter model checker) [1] は代表的なモデル検査器であり，並行システムのソフトウェアを対象としている．Promela(PROcess MEta LAnguage) というモデル記述言語でシステムの振る舞いを記述し，検証する性質は線形時相論理 LTL(Liner Temporal Logic) を用いて記述する．SPIN を使って Java のソースコードを検証する JAVA PathFinder [9] や C プログラムを検査する SLAM [6] などのツールの研究が行われている．

PRISM [10] はマルコフ連鎖を状態遷移系として扱うモデル検査器である．マルコフ連鎖は有向グラフの辺に頂点から頂点へと遷移する確率がついてあり，未来の遷移が現在の値によって決定され，過去の挙動に依らないという性質を持つ．どのような遷移をするかがわからないという点で本研究で扱う不確かさと類似している．しかし，本研究で焦点を当てている不確かさは実装候補がいくつかあることにより発生するという認識的不確かさであるため，システム自体に確率的要素があるマルコフ連鎖モデルが持つ偶発的不確かさとは不確かさの性質が異なる．

LTSA(Labelled Transition System Analyser) [11] は並列システムの検証を行うモデル検査器である．LTSA は検査するシステムと，検証したいプロパティを有限オートマトンを用いてモデル化する．その二つのモデルを合成して，プロパティに違反する箇所がないかを網羅的に検査する．モデルは LTS(Labelled Transition System) と呼ばれる状態遷移系で表される．LTSA は FSP(Finite State Process) と呼ばれるモデル記述言語を用いて LTS を表現する．不確かさを包容したインターフェース機構 Archface-U はメソッドの呼び出し関係を FSP に準拠した構文で記述するため，本研究では LTSA をモデル検査器として利用する．

2.2.2 モデル検査における状態爆発問題

実装されたソースコードを全てモデル検査しようとした場合などに状態爆発が起きる可能性がある．この問題に対処するために，システムを単純化して状態数を減らす抽象化 [7] や，状態遷移図を作らず二分決定図によってグラフを表現する手法 [12] など様々な手法が研究されている．本研究では，ソースコード自体ではなく，設計全体のうちソースコードが必ず従わなければならない制約として任意に選んだ設計情報をモデル検査の対象とすることでソースコードを間接的に検査する．したがって検査対象は設計の一部分であり，モデル検査によって生成される状態数は抑えられるため，状態爆発が起こる可能性を減らすことができる．

3. 不確かさを包容したインターフェース機構 Archface-U

本章では，不確かさを包容したインターフェース機構 Archface-U とその支援機構 iArch-U について説明する．Archface-U は，設計と実装のトレーサビリティを保ちながら開発を行うためのインターフェース機構 Archface [15] を拡張して不確かさを扱えるようにしたものである．

3.1 Archface と iArch の概要

Archface とは，アーキテクチャ設計と実装間の食い違いを無くすためのインターフェース機構である．Archface はプログラムにおけるインターフェースのような役割であり，実装する Java コード全体が Archface に記述されたアーキテクチャの制約に従うように強制する．

iArch は Archface を用いた開発を支援する統合開発環境である．iArch 上で Java コードがアーキテクチャ設計に従っているかどうかを検査しながら開発を行う．

3.2 Archface-U と iArch-U の概要

Archface-U とは，Archface を拡張して不確かさをモジュール化して扱えるようにしたインターフェース機構である．従来の Archface をインターフェース Certain Archface とし，不確かさを含むものは Certain Archface のサブインターフェース Uncertain Archface として設計情報を記述する．クラスやメソッドの宣言は Component に記述し，Component に記述されたメソッドの呼び出し関係を Connector に記述する．

Archface-U で扱う不確かさは以下の二つである．

(1) ある Component にいくつかの候補があり，どれを実

```
1   interface component P2P{
2     void start();
3   }
4   interface component P2PClient{
5     void cancel();
6     void completed();
7   }
8   interface component P2PServer{
9     void cancel();
10    void restart();
11  }
12  uncertain component uP2P extends P2P{
13     {void share(),void share(File file)};
14  }
15  uncertain component uP2PServer extends P2PServer{
16    [void restart();]
17  }
18  interface connector P2PSystem{
19    P2P = (P2P.start->P2PClient);
20    P2PClient = (P2PClient.completed->P2PServer);
21    P2PServer = (P2PServer.cancel->P2P);
22  }
23  uncertain connector uP2PSystem extends P2P{
24    u P2P = ({uP2P.share}->P2PServer);
25    u P2PServer = ([uP2PServer.restart]->P2PClient);
26    u P2PClient = (P2PClient.completed->P2P);
27  }
```

図 3 P2P ファイルシステムの Archface-U によるインターフェース記述例

際にシステムに組み込むかわからないという不確かさ

(2) ある Component が実際にシステムに組み込まれるかわからないという不確かさ

以下，(1) の不確かさを Alternative，(2) の不確かさを Optional と呼称する．

3.3 Archface-U のインターフェース記述

Archface-U を図 2 の P2P ファイル共有システムに適用して Archface-U のインターフェース記述の説明をする．

3.3.1 Certain Archface のインターフェース記述

Certain Archface は Component と Connector の二つのインターフェースから成る．Component インターフェースには，実装するクラスとそのクラスが宣言するメソッドを記述する．図 3 の例の 1~3 行目に記述より，Java コードでメソッド start がクラス P2P に実装されていなければ制約違反となる．

Connector インターフェースには，Component インターフェースに記述したメソッド同士の呼び出し関係を記述する．図 3 の例の 18 行目の記述より，Java コードが P2P.start から P2PClient へ接続していなければ制約違反となる．

3.3.2 Uncertain Archface のインターフェース記述

Uncertain Archface は Certain Archface の Component と Connector を継承して，新たに不確かさを扱えるようにしたものである．Optional メソッドはメソッド名を [ ] で

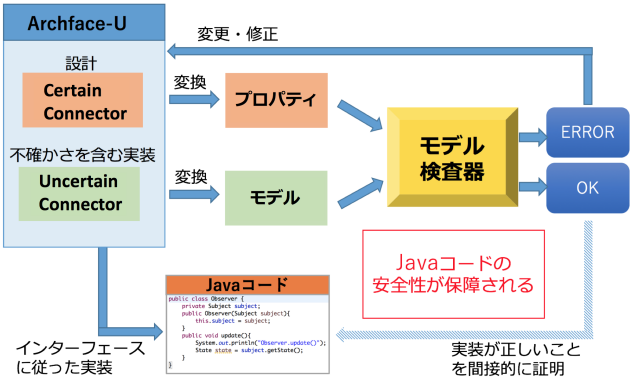

図 4 不確かさを包容したモデル検査の概念図

囲むことで表現し，Alternative メソッドはメソッド候補を { } の中にカンマで区切って記述することで表現する．図 3 の例では，Optional メソッド [void restart()] は実装されるか否かが不確かであることを表しているため，Java コードで実装されていてもされていなくても制約違反にはならない．また，Alternative メソッド {void share(),void share(File file)} は，どちらのメソッドも Java コードで実装されていなければ制約違反となり，いずれかのメソッドが実装されていれば制約違反にはならない．

Uncertain Connector には不確かさのあるメソッドの呼び出し関係を記述できる．

4. 不確かさを包容した自動モデル検査の概要

この章では，本研究で提案するシステムについて述べる．iArch-U に LTSA の機能を追加し，Connector インターフェースの記述を自動でモデル検査する．Java コードの実装は Archface-U のインターフェース記述に従っているため，その記述をモデル検査して問題が無ければ，Java コードの実装は設計通りの安全なものであることが保障される．

4.1 Java コードの安全性の向上

Uncertain Connector の記述をモデル検査することにより Java コードの安全性が保障されることを説明する．

iArch-U を用いて開発を行うためには，設計情報をインターフェースとして記述しなければならない．もし，Java コードの実装がインターフェースの記述と食い違っていた場合は型検査違反としてコンパイルエラーとなる．すなわち，Java コードは必ずインターフェースの記述に従っていることが保障されているため安全な開発が進められているといえる．しかし，それはインターフェースの記述に誤りがないことが前提であり，もしインターフェースの記述にエラーを引き起こす危険性などが存在するならば Java コードも同じ危険性を持つことになる．Archface-U は不確かさをインターフェースに記述して扱うため，不確かさが発生することにより設計段階では存在しなかったバグが発生する可能性がある．そのため，iArch-U を用いた開発

は iArch を用いた開発よりも Java コードの安全性が保障されないという問題があった．

本研究はこのような問題を解決するものである．本研究では，Uncertain Connector に記述されたメソッドの呼び出し関係をモデルとして，それが Connector インターフェースに記述された設計に違反しないかをモデル検査する．違反が見つからなかった場合は，不確かなメソッドに関係なくインターフェース記述は正しいということであるため，不確かなメソッドを普通のメソッドと区別せずに実装を進めて問題ない．違反が見つかった場合は，出力される反例を参考に原因となる遷移を特定して，不確かなメソッドがそのような遷移をとらないような設計にすることで問題は解決する．このようにして，iArch-U を用いた開発により Java コードの安全性を保障することが可能となる．

4.2 モデル検査器 LTSA の概要

LTSA による検査では，Safety と Progress と呼ばれる二つのプロパティを検査することができる．Safety 検査は，プロパティとして記述した仕様から逸脱するような動作をしないかを検査する．Progress 検査は，指定した動作を全てくまなく実行できるかを検査できる．

LTSA のモデル記述言語 FSP はいくつかのプロセスとアクションからなる．一般にモデル検査ではプロパティを時相論理を用いて表現するが，LTSA ではプロパティの記述も FSP で行う．そのため，Safety と Progress 以外のプロパティを検査する場合は，FSP でプロパティを記述して生成した LTS と検査対象のモデルの LTS を合成してプロパティに反する遷移がないかを検査する．

4.3 LTSA を用いた Connecotr の自動モデル検査

本節では，LTSA を用いてどのように Archface-U のインターフェース記述をモデル検査するかを説明する．

設計には問題が無いものとして，不確かさを含む実装が設計の仕様を満たすかどうかを検査する．すなわち，Archface-U に記述された不確かさを含む FSP 群を検査対象のモデルとして，Archface-U に記述された不確かさを含まない FSP 群をプロパティとしてモデル検査をする．

Optional メソッドや Alternative メソッドで記述された不確かさは LTSA では扱えないため，これらを LTSA で検査できるように不確かさを展開する方法を示す．

4.3.1 Optional メソッドの展開

Optional メソッドを用いた，$P1 = (c1 \rightarrow [u1] \rightarrow c2 \rightarrow P1)$ という Connector の記述を LTSA で扱えるように変換する．P1 はプロセスを表し，c1,c2,u1 はアクションを表す．Optional の定義から，P1 は遷移 1:$(c1 \rightarrow u1 \rightarrow c2 \rightarrow P1)$ と遷移 2:$(c1 \rightarrow c2 \rightarrow P1)$ のいずれかの遷移の仕方をとる．したがって，これらの遷移を Choice プロセス “|” で区切って，$P1 = (c1 \rightarrow u1 \rightarrow c2 \rightarrow P1|c1 \rightarrow c2 \rightarrow P1)$ とすることで Optional メソッドの変換が完了する．

4.3.2 Alternative メソッドの展開

Alternative メソッドを用いた，$P2 = (c1 \rightarrow \{u1, u2\} \rightarrow P2)$ という Connector の記述を LTSA で扱えるように変換する．Alternative の定義から，P2 は遷移 1:$(c1 \rightarrow u1 \rightarrow P1)$ と遷移 2:$(c1 \rightarrow u2 \rightarrow P1)$ のいずれかの遷移の仕方をとる．したがって，Optional メソッドの展開と同様にこれらの遷移を Choice プロセス “|” で区切って，$P2 = (c1 \rightarrow u1 \rightarrow P2|c1 \rightarrow u2 \rightarrow P2)$ とすることで Alternative メソッドの変換が完了する．{ } で囲まれたメソッドの候補が 3 つ以上の場合も同じ手順で候補の数だけプロセスを展開することができる．

4.4 Connector のモデル検査の手順

Connector に記述されたメソッドの呼び出し関係は以下の手順で変形され，LTSA を用いてモデル検査される．

手順 1 Certain Connector のプロセスをそれぞれカンマでつなぎ，最後のプロセスの末尾にピリオドをつけて一つのプロセス式とする．このプロセス式をプロパティとする．

手順 2 Uncertain Connector の不確かさを展開する．

手順 3 Uncertain Connector の展開したプロセスをカンマでつなぎ，最後のプロセスの末尾にピリオドをつけて一つのプロセス式とする．このプロセス式を検査対象のモデルとする．以上で Connector が LTSA で検査できる形式に Connector の変換が完了する．

手順 4 LTSA を用いて上記の二つのプロセス式をコンパイルした後に合成を行う．

手順 5 Safety 検査と Progress 検査を行い，結果を表示する．

すなわち，検査したい設計対象を Uncertain Connector に記述し，それが満たしてほしい性質を Certain Connector に記述してモデル検査を行う．

4.5 状態爆発問題への対処

ソースコードをモデル検査する場合はソースコードの量が多くなるほど状態爆発が起こりやすい．本研究では，検査対象を Connector インターフェースの記述のみに絞ることでこの問題に対処する．4.1 節で説明したように，Java コード自体を検査するのではなく Connector インターフェースの記述を検査することで間接的に Java コードを検査したことになる．Connector インターフェースの記述は設計情報の一部であるためソースコードに比べて小さく，これをモデル検査しても状態爆発が起きる可能性は低い．そのため，膨大なソースコードをモデル検査せずにソースコードがプロパティを満たすか判定することが可能となる．ただし，不確かさの展開により不確かさの数が増えるとともに状態数は増えるため，不確かさの数が大きく

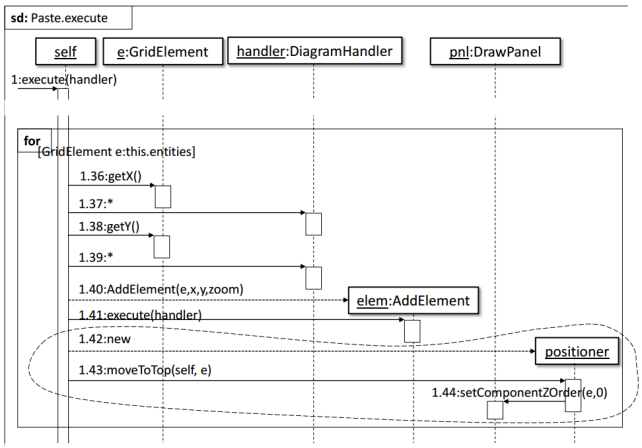

図 5 Femilis らによる Paste の execute オペレーションのバグ修正（元論文 [8]Figure 7. より引用）

なりすぎると状態爆発を起こす可能性がある．

5. ツールの実装

本研究では，iArch-U に自動モデル検査機能を追加し，ツールサポートを試みた．iArch-U は，Java 開発環境である Eclipse [2] のプラグインとして実装されている．自動モデル検査機能は，モデル検査器 LTSA の Eclipse プラグイン (LTSA Eclipse Plug-in (v2.0 beta)) [3] の機能を利用している．このプラグインを用いてモデル検査を行うには，プロジェクトに拡張子が「ltl」のファイルを作成し，そのファイルに FSP を記述してモデル検査を行う必要があるが，本研究ではこの過程を無くして自動でモデル検査を行うようにした．Safety 検査と Progress 検査の結果は iArch-U の View にテーブル形式で出力する．

6. 例題適用

この章では，作成したツールに対して例題を適用して，実際にこのツールを使う流れと，何ができるのかを示す．

6.1 例題シナリオ

Famelis らは MDE（Model-Driven Engineering）ソフトウェアのメンテナンスタスクの例をケーススタディとして用いている [8]．以下にその概要を示す．例題シナリオは，次の通りである．あるエンジニアがソフトウェアに発生したバグを無くすために，ツールを用いて自動でいくつかの解決案となる UML 図を生成した．しかし，それらの解決案がシステムへ与える影響をが不明で，どの解決案を選んでいいかがわからない．この場合に Partial Model を用いることでどうエンジニアの実装を支援できるかを説明する．

ソフトウェアのバグ修正の具体例として UMLet [5] というオープンソースプロジェクトで発生したバグを修正する．UMLet は Java ベースの UML エディタである．メンテナンスタスクは，Github のオンライン Issue ログ上の「copied items should have a higher z-order priority」 [4] で挙げられたバグの修正である．このバグは，ユーザがコピーしたアイテムをペーストしたとき，ペーストしたアイテムが一番上に現れないというものである．したがって，P1:「ペーストされたアイテムの z-order が常に 0 にならなければいけない」というプロパティを満たしたバグ修正を行う必要がある。ペースト機能は UMLet の Paste クラスの execute メソッドにより実装されている．図 5 は Famelis らによるバグ修正を加えたシーケンス図の一部であり，提案されたバグ修正は丸で囲まれた部分である．しかし，この修正方法だと次の 2 つの制約によってシーケンス図からコードを生成することができない．

1)ClasslessInstance クラスが無いオブジェクトの存在がこの制約に違反する．修正案は以下の通りである．

- RC1: positioner を削除する
- RC2: positioner を既存のクラスを持つオブジェクトに変える
- RC3: positioner に既存のクラスを割り当てる
- RC4: positioner に新しいクラスを割り当てる

2)DanglingOperation メッセージを受け取るオブジェクトのクラスがそのメッセージによって表されるオペレーションを宣言していなければこの制約に違反する．修正案は以下の通りである．

- RD1: moveToTop を受信側のオブジェクトのクラスに追加する
- RD2: moveToTop を受信側のオブジェクトの既存のオペレーションと取り替える
- RD3: moveToTop を削除する

この二つの制約は Mens らの論文 [13] に基づいている．

以上が Famelis らがケーススタディとして用いたメンテナンスタスクの例の概要である．どのように Archface-U を使ってこのメンテナンスタスクに対処するかを示す．

6.2 Archface-U の適用

6.2.1 検査の適用範囲

UMLet の開発は iArch-U を用いて行い，Component と Connector にはすでに設計が記述されて，実装がそれに従って行われているものとする．今回は Paste クラスの execute を構成するクラスとオペレーションの記述があれば十分であるため，Paste 以外のクラスなどは考慮しない．

本稿では Archface-U のインターフェース記述の能力を考慮して，ClasslessInstance と DanglingOperation の修正案のうち次の組み合わせのみについて考える．

1) positioner を既存のオブジェクトに変更する (RC2).

2) moveToTop を positioner のクラスに追加する (RD1,RD3)．moveToTop を positioner クラスが持つ他のオペレーションに変更する (RD2).

RC2 は図 5 のオブジェクトのうちどれかを positioner と

```
1   interface connector Paste{
2     Paste=(AddElement.execute->DrawPanel.moveToTop
3        ->DrawPanel);
4     DrawPanel = (DrawPanel.setComponentZOrder->Paste);
5   }
6   uncertain connector UPaste extends Paste{
7     u Paste = (GridElement.getX -> GridElement.getY
8        -> AddElement.AddElement -> AddElement.execute
9        -> [UDrawPanel.moveToTop] -> DrawPanel);
10    u DrawPanel = (DrawPanel.setComponentZOrder
11       -> Paste);
12  }
```

図 6 UMLet の Paste クラスの Archface-U による Connector インターフェース記述例

する．RC3 は RC2 とほぼ同じ結果になるため省く．RD1 は positioner のクラスのインターフェースに moveToTop を Optional メソッドとして記述する．すなわち，moveToTop がある場合とない場合の両方を検査することになるため RD3 も実現される．RD2 は certain component に記述してある全てのメソッドを uncertain component 内で Alternative メソッド扱いにする．

6.2.2 検査する性質

P1 をプロパティとして certain connector に記述することを考える．ペーストが実行されるのは，1:41execute で，z-order が 0 になるのは 1:43moveToTop を実行した後に 1:44setComponentZOrder が実行されるときである．したがって，P1 は「execute が呼び出されたとき，その後に必ず moveToTop が実行された後に setComponentZOrder が呼び出される」と言い換えることができる．これを FSP 形式で interface connector に記述すると図 6 のようになる．

6.2.3 Archface-U を用いた検査の手順

検査の進め方は次の通りである．まず今回対象としているクラスとオペレーションをそれぞれ interface component に記述する．次に前述した通りに各クラスの uncertain component に Optional と Alternative のメソッドを記述する．positioner となるオブジェクトを一つ選び，選んだオブジェクトに従って uncertain connector の記述を変更して RD1 及び RD3 の場合と RD2 の場合を検査して，他のオブジェクトでも同じように検査していく．今回は，setComponentZOrder が実行されるかを検査したいため，Progress 検査の結果を見ればいい．

6.3 シミュレーション結果

RC2 と RD1 及び RD3 の組み合わせで検査を行うと，プロパティ違反が見つかった．Progress 検査の結果，$getX \rightarrow getY \rightarrow addElement \rightarrow execute \rightarrow setComponentZOrder$ という遷移でエラーが報告されている．これは Optional メソッドである moveToTop が採用されなかった場合の遷移を表してあり，P1 を満たすためには moveToTop が必要であることを示している．

RC2 と RD2 の組み合わせで検査を行うと，for ループ中で使われているメソッド getX，getY，AddElement が moveToTop の役割となるときに違反が見つかる．これは P1 を AddElement.execute の後に上記の moveToTop の役割となるメソッドが来るようにしているためである．すなわち，シーケンス図上では for ループ内の getX，getY，AddElement は，AddElement.execute 以前に呼び出されるが，P1 にはそれらが AddElement.execute の次に来ることを要求しているため違反が報告されてしまう．プロパティの記述を変更して，AddElement.execute 以前のメソッドの流れも記述するとこのエラーはなくなり，全てのパターンで違反無しと報告される．

Famelis らによる Partial Model を用いた検証結果と比較する．Famelis らによると，P1 の検査結果は Maybe であった．Archface-U を用いた検査では，RC2 と RD1 の組み合わせで Optional メソッドの moveToTop が遷移に存在する場合はプロパティに違反せず，遷移に存在しない場合はプロパティ違反になるという結果となった．これは P1 が Maybe であるといえるため，Famelis らと同じ検査結果が求められた．また，Famelis らは P1 に違反する反例として moveToTop が存在しないモデルを見つけ，moveToTop は P1 を満たすために必要であることを示した．Archface-U による検査でも moveToTop が無い場合にプロパティ違反を検出したことから，Famelis らと同様に P1 を満たすためには moveToTop が必要であることを求めることができた．

以上の結果より，Archface-U を用いて不確かさを表現し，それをモデル検査することによりどのようなモデルなら設計に反しないかを求めることができた．

6.4 考察

ClasslessInstance の修正案のうち RC1 と RC4 は適用ができなかった．これらは，実際にそのクラスを使うか否かがわからない Optional なクラスであると言える．現在はメソッドの不確かさのみを扱っているが，クラスも不確かなものとして扱うことができれば Archface-U による支援対象が幅広くなり，より柔軟な開発が行えることがわかった．

7. まとめ

ソフトウェア開発の実装段階で発生する不確かさは，設計時には存在しなかったエラーを引き起こす危険性がある．本研究では，iArch-U に自動モデル検査機能を追加してこの問題に対処した．iArch-U によって Java コードはインターフェース記述に従って実装されているため，インターフェースのメソッドの呼び出し関係の正当性が証明されることは Java コードの正当性が証明されることにつながる．従って，Archface-U によって保障される安全な開発をより強固に保障することが可能となった．

今後の課題として任意のプロパティを LTL で記述して検

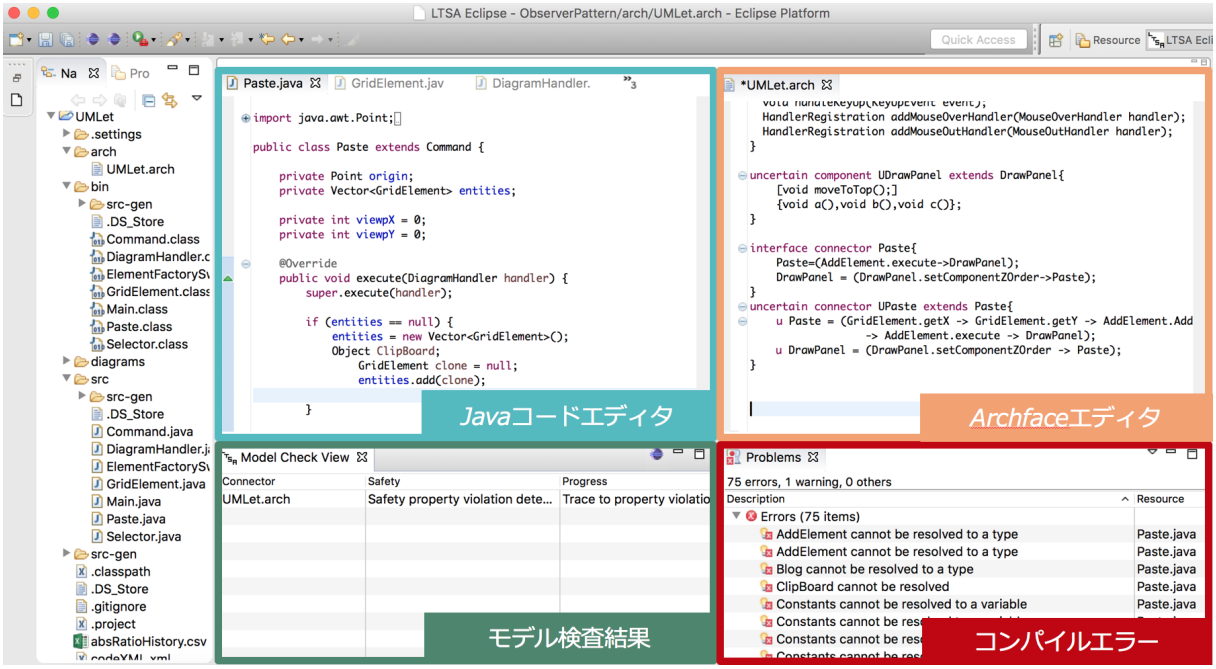

図 7 iArch を用いた開発画面イメージ

査したい性質の表現を拡張することについて考えている．また，2.2 節で触れた PRISM のように，性質が偶発的である不確かさも本研究で扱えるようにすることが課題として挙げられる．これは FSP の遷移矢印に確率を付与してマルコフ連鎖を表現することで対応できると考えられる．

謝辞 本研究は, 文部科学省科学研究補助費基盤研究 (A)(課題番号 26240007) による助成を受けた．

参考文献

[1] *The model checker SPIN*. IEEE Computer Society, 1997.
[2] *Eclipse*. http://www.eclipse.org/, (2016/2/1).
[3] *LTSA - Labelled Transition System Analyser*. http://www.doc.ic.ac.uk/ltsa/, (2016/2/1).
[4] *UMLet Issue List*. https://github.com/umlet/umlet/issues, (2016/2/1).
[5] *UMLet website*. http://www.umlet.com/, (2016/2/1).
[6] Thomas Ball and Sriram K. Rajamani. *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings*, chapter The SLAM Toolkit, pp. 260–264. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.
[7] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 5, pp. 1512–1542, September 1994.
[8] M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 573–583, June 2012.
[9] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, Vol. 2, No. 4, pp. 366–381.
[10] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 2.0: A tool for probabilistic model checking. *Quantitative Evaluation of Systems, International Conference on*, Vol. 0, pp. 322–323, 2004.
[11] J Magee and J Kramer. *State Model and Java Programs*. WILEY, 2 edition, 2006.
[12] Kenneth L. McMillan. *Symbolic Model Checking*, chapter Symbolic Model Checking, pp. 25–60. Springer US, Boston, MA, 1993.
[13] Tom Mens and Ragnhild Straeten. *Recent Trends in Algebraic Development Techniques: 18th International Workshop, WADT 2006, La Roche en Ardenne, Belgium, June 1-3, 2006, Revised Selected Papers*, chapter Incremental Resolution of Model Inconsistencies, pp. 111–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
[14] Diego Perez-Palacin and Raffaela Mirandola. Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pp. 3–14, New York, NY, USA, 2014. ACM.
[15] Naoyasu Ubayashi, Jun Nomura, and Tetsuo Tamai. Archface: A contract place where architectural design and code meet together. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pp. 75–84, New York, NY, USA, 2010. ACM.
[16] 深町拓也, 鵜林尚靖, 細合晋太郎, 亀井靖高. 不確かさを包容するソフトウェア開発プロセス. 第 22 回 ソフトウェア工学の基礎ワークショップ (FOSE2015), 11 2015.

「LTSA 連携による不確かさを包容した自動モデル検査」　正誤表

内容に誤りがありましたので謹んで訂正いたします.

該当頁	誤	正
P.2 上から 7 行目	モデル化される情報に関する不確かさ	環境に関する不確かさ