

不確かさを包容するJavaプログラミング環境

深町 拓也^{1,a)} 鶴林 尚靖^{1,b)} 細合 晋太郎^{1,c)} 亀井 靖高^{1,d)}

概要：ソフトウェア開発の様々な段階において「不確かさ」への対応は避けられない。従来、これはリスク管理の一つとして処理してきた問題であるが、人手に大きく依存するため、バグやコードの難読化などの原因となりがちであった。一般的に「不確かさ」には Known Unknown（既知の未知）と Unknown Unknown（未知の未知）の2つのタイプがある。本論文では、Known Unknown タイプの不確かさをモジュールにプログラミング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* を提案する。*iArch-U* を用いることで不確かさを抱えたままでもアーキテクチャ設計と実装の整合性の維持や、制約条件の検査などを行いながら実装を行うことができる。

キーワード：不確かさ, Partial Model, 統合開発環境

1. はじめに

ソフトウェア開発の様々な段階、例えばモデルの設計段階やコードの実装段階など、において「不確かさ」は避けられないものである。

一般に「不確かさ」は、Known Unknown（既知の未知）と Unknown Unknown（未知の未知）に分けることができる。ソフトウェア開発においては、前者は「実装や設計の方法は分かっているが要求が不明のため確定ができない」といったような何が不確かなのかは分かっているが、結論を出すことが出来ない状態のことと言える。後者は「プロダクトリリース後の想定外の操作によるエラー」など、開発者が想定できないような不確かさを指す。

この内、Known Unknown, すなわち、既知の不確かさはメーリングリストによる共有、仕様書へのマーキング、不確かな部分のコメントアウトなど、それぞれの開発チームでリスク管理の一つとして人手による処理を行ってきた。しかし、このような人手による管理法は人為的ミスが発生しがちであった。

本研究では、ソフトウェア開発における既知の不確かさをモジュール化することによって、何が不確かな状態で開発を行っているかを開発者が管理できるようにする。こうすることで、既知の不確かさを抱えながらも安全に開発を

行うことができるようにする環境を構築することを目的とする。以降の「不確かさ」については全て既知のものとして考える。

ソフトウェア開発における不確かさをうまく包容しつつ開発を進めるために、本研究では *Archface-U* (*Archface-Uncertain*) と呼ばれるインターフェースを用いて実装上の不確かさを表現し、モジュール化することを提案する。*Archface-U* に実装上不確かなメソッドやフローなどを記述し、確定をしたらそのインターフェースを削除することで開発中に何が不確かか、何が確定しているのかを開発者が把握できるようにする。*Archface-U* を記述することによって、コードに記述した不確かなメソッドやフローが実際には成り立っているのかどうかを検査し、開発者へ知らせる。このように不確かさを *Archface-U* へ記述することで開発者は実装中に不確かさが発生した場合でもその不確かさを把握しつつ、安全に開発を行うことができる。

本論文では、2章で関連研究をあげながら、ソフトウェア開発における不確かさについて説明する。3章では、不確かさを表現するインターフェースとして *Archface-U* を提案する。4章では、*Archface-U* を用いた不確かなプログラムの検査について説明する。5章では、*Archface-U* と不確かさを含んだ設計の既存研究である Partial Model 間の双方向変換のアルゴリズムについて述べる。6章においては、3, 4章で述べたアプローチをどのようにツールでサポートしているかについて述べる。最後に、7章では本稿のまとめと今後の課題について述べる。

¹ 九州大学

Kyushu University

a) fukamachi@posl.ait.kyushu-u.ac.jp

b) ubayashi@ait.kyushu-u.ac.jp

c) hosoai@qito.kyushu-u.ac.jp

d) kamei@ait.kyushu-u.ac.jp

2. ソフトウェア開発における不確かさ

本章では、ソフトウェア開発における不確かさに関連する研究を紹介する。

2.1 不確かさと Partial Model

ソフトウェア開発における不確かさは開発のあらゆる段階において発生しうるものである。ソフトウェア工学のコミュニティにおいても、要求分析 [6]、モデリング [4]、テスト [3] など様々な開発段階の不確かさについての研究が行われている。この研究の多くは、各開発段階においてどのように不確かさを許容するかに焦点を当てている。また、不確かさのタイプとしては Known Unknown を対象としている。ここでは、Known Unknown 型の不確かさに関する代表的研究として、Famelis らの「Partial Models: Towards Modeling and Reasoning with Uncertainty」[4] を説明する。

この研究では、設計初期段階において起こりうる、「いくつかの設計候補がありそれらの中からどれを適用するかが定まっていない」という不確かさに関して扱っている。Famelis らは、このような不確かな設計を状態遷移図として表し、その状態遷移図群を Partial Model という 1 つの不確かさを表現している状態遷移図にまとめている。なお、Partial Model は LTS (Labelled Transition System) と呼ばれる状態遷移図を対象とする。

図 1 では、ある P2P ファイル共有システムにおける 6 つの設計候補 (a-f) をまとめた Partial Model (g) を表している。Partial Model では、複数のモデル全てで共通しているエッジ、ノードを実線で表し、それ以外を破線で表現する。メソッド名が同じでもノードの始点と終点が異なる場合は別のエッジとして扱う。

更に、Partial Model を構成している元々の設計候補の状態遷移図を命題論理式によって表現する。表現された命題論理式は破線で表されたエッジやノードについて各候補においてどのエッジやノードが使われているかを表し、それらを OR 演算子で結ぶことで表現が可能である。

また、同じように論理式で表現されたプロパティと呼ばれる制約条件との充足可能性を解析することができる。Partial Model Φ_M 上で、プロパティ Φ_P を検査するためには $\Phi_M \wedge \Phi_P$ 及び、 $\Phi_M \wedge \neg\Phi_P$ について SAT (充足可能性問題) ソルバ*1 で解析を行い、充足可能性問題を解く。SAT ソルバによる結果に応じて表 1 のようなプロパティの検査の結果を出すことができる。ここでの True, False はプロパティが充足, 非充足に対応している。また、

*1 ある一つの命題論理式が与えられたとき、それに含まれる変数の値を True, あるいは False にうまく定めることによって全体の値を True にできるか、という問題 (SAT) を解くことができるプログラムのこと。

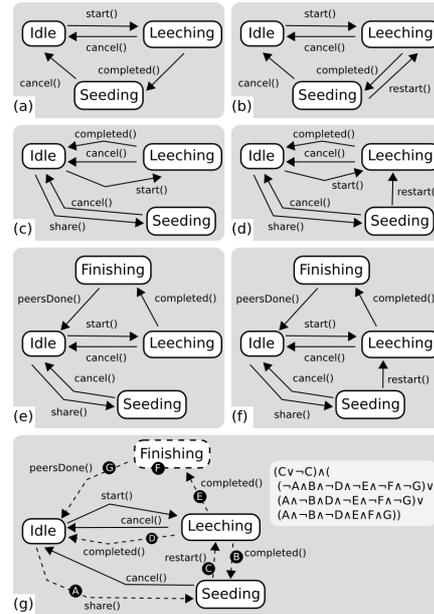


図 1 P2P ファイル共有システムにおける 6 つの設計候補 (a-f) とその候補を統合した Partial Model (g) (元論文 [4] Figure 1. より引用)

表 1 Partial Model M 上でのプロパティ p の検査表 (元論文 [4] Table 1. より引用)

$\Phi_M \wedge \Phi_P$	$\Phi_M \wedge \neg\Phi_P$	Property p
SAT	SAT	Maybe
SAT	UNSAT	True
UNSAT	SAT	False
UNSAT	UNSAT	(error)

Maybe については $\Phi_M \wedge \Phi_P$ 及び、 $\Phi_M \wedge \neg\Phi_P$ がいずれも SAT のときに出力される。これはある状態遷移図群では $\Phi_M \wedge \Phi_P$ で充足可能であるが、前者以外の状態遷移図群において $\Phi_M \wedge \neg\Phi_P$ が充足可能であるという場合に起こりうる。つまり、プロパティが成り立つかどうかはどのモデルを最終的に選択するかによって異なるため、結果的にプロパティが成り立つかどうかは「不確か」であるというのが妥当である。そのため、Maybe と表現されている。また、いずれも UNSAT である場合はそのものの Partial Model を構成している状態遷移図群が充足不能であると考えられるため、設計自体を見直す必要がある。そのため、この場合は error として表現をされている。

2.2 本研究における「不確かさ」

本論文では、Known Unknown タイプの不確かさを扱う。そのためのベースとして Partial Model の考え方をプログラミング支援に導入する。Partial Model では、設計における不確かさに対してのみを扱っていたが、実装においても扱えるように拡張する。この拡張によって設計のみに適用できていた Partial Model によるプロパティの検査

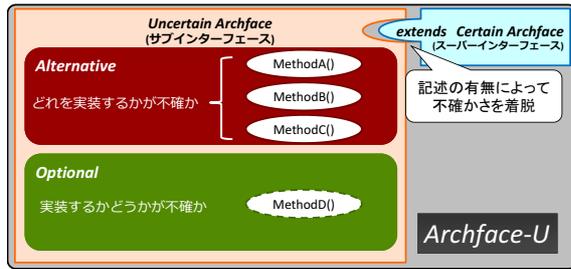


図 2 Archface-U 概要

が実装においても適用できるようになる。また、設計と実装について同じモデルを扱うことができるため設計と実装のトレーサビリティを取ることもつながる。

ただし、Partial Model は状態遷移図の拡張によって設計に関する不確かさが表現されている。そのため、プログラミングについて Partial Model に基づいた不確かさを適用するためには実装についての不確かさを改めて表現することが望ましい。本研究では、このような不確かさを表現するために実装について不確かさを表すインターフェースを記述することで Partial Model に基づいた不確かさの検査を行うことができるようにする。

本研究では Partial Model に基づいたソフトウェア開発における不確かさを以下の 2 つに分類した。

- (1) ある目的に対していくつかメソッドの候補があり、その中でどれを実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) あるメソッドについて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

この 2 つの不確かさのうち、(1) を *Alternative*、(2) を *Optional* と以下呼称する。

3. 不確かさを包容したインターフェース Archface-U

第 2 章で述べた不確かさを表現するために本研究では、Archface[7] インターフェース機構を拡張した Archface-U を用いる。

3.1 Archface-U の概要

Archface は、アーキテクチャ設計と Java による実装の間のギャップを埋めるためのインターフェース機構である。Archface はプログラミングにおけるインターフェースであり、プログラムを実装する際、Archface 中に記述されたアーキテクチャの制約に従うことが強制される。Archface の開発環境である *iArch* はソースコードを解析し、アーキテクチャ制約に違反していないかを検査する。この検査を以後、型検査と呼称する。もし、この制約に違反した場合はコンパイルエラーとして制約違反を開発者に知らせることが確実にアーキテクチャ設計に従った開発を行うことが

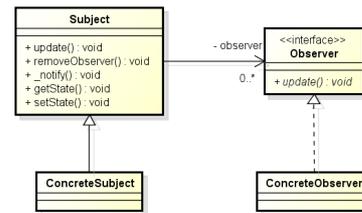


図 3 Observer パターンのクラス図

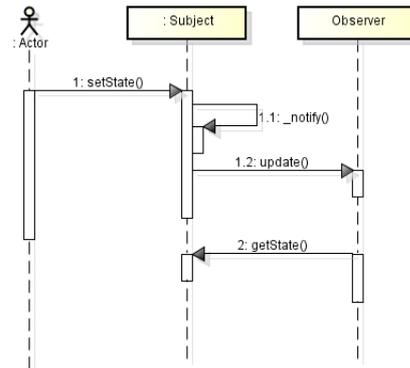


図 4 Observer パターンのシーケンス図

できる。

本研究では上で述べた Archface を拡張し、不確かさを包容したインターフェースとして Archface-U を定義した。Archface-U は従来の Archface である Certain Archface と、そのサブインターフェースである不確かさを含んだ Uncertain Archface の 2 種類のインターフェースによって構成される (図 2)。

また、Uncertain Archface に Certain Archface を記述するか否かによって不確かさを着脱する。

3.2 Archface-U のインターフェース記述

Archface-U は Component-and-Connector アーキテクチャ [1] を記述支援対象としている。このアーキテクチャは計算処理を行うコンポーネント群とそれらの接続関係によってアーキテクチャを記述する。そのため、Archface-U はコンポーネントとそのコンポーネント間の接続関係や協調関係であるコネクタをインターフェースに表現する。

Archface-U では、Java コードのメソッドの呼び出しや実行、クラスやメソッドの宣言などをプログラム点として定義する。

以降、Archface-U を適用する例として Observer パターンを用いて説明する。Observer パターンはデザインパターンの 1 つである。Observer パターンは Observer と Subject の 2 種類のコンポーネントからなり、Subject の状態が変化した際に Observer へ通知を行うためのデザインパターンである。

図 3、図 4 に Observer パターンの UML のクラス図と

```

1  interface component Subject{
2      void setState(State state);
3      State getState();
4      void addObserver(Observer observer);
5  }
6  interface component Observer{
7      void update();
8  }
9  uncertain component uSubject extends Subject{
10     [void _notify();]
11     {
12         void removeObserver(),
13         void deleteObserver()
14     };
15 }
16
17 interface connector cObserverPattern{
18     Subject=(Subject.setState->Observer.update
19     ->Subject.getState->Subject);
20     Observer=(Observer.update->
21     Subject.getState->Observer);
22 }
23 uncertain connector ucObserverPattern
24 extends cObserverPattern{
25     Subject=(Subject.setState->
26     [Subject._notify]->Observer.update
27     ->Subject.getState->Subject);
28 }

```

図 5 Observer パターンの *Archface-U* によるインターフェース記述

シーケンス図を示す。Observer パターンは図 4 のシーケンス図が示す通り、Subject のコンポーネントを `setState` によって変更させると、Subject コンポーネントの `_notify` を呼び出すことで、Observer への通知を `update` の呼び出しによって行っている。Observer コンポーネント内の `update` では、Subject コンポーネント内の `getState` が呼ばれ、Subject コンポーネントの変化した状態を Observer が受け取っている。

また、図 5 に、Observer パターンを *Archface-U* で表記したものを示す。

3.2.1 *Certain Archface* のインターフェース記述

Certain Archface (従来の *Archface*) はコンポーネントとコネクタの 2 つのインターフェースがある。

コンポーネントインターフェース (*interface component*) は、実装において宣言するメソッド名を Java インターフェースに準拠した構文によって記述する。コンポーネント名が Java の実装におけるクラス名に対応し、その中にメソッド名を記述することでプログラム点を定義する。図 5 の例では、`setState` が宣言されていない場合は制約違反となる。

コネクタインターフェース (*interface connector*) は、定義されたコンポーネント同士の接続方法を規定し、公開されたプログラム点群をどのように実行し協調させるかを記述する。具体的には、前述のコンポーネントインターフェースによって定義されたコンポーネント同士がどのように接続しているかを FSP (Finite State Process) [5] に準拠した構文によって記述する。図 5 の例では、`Observer.update` から `Subject.getState` への接続がない場合コネクタインターフェースの `Observer` における制約違反となる。

3.2.2 *Uncertain Archface* のインターフェース記述

Uncertain Archface は、*Alternative*、*Optional* の不確かさを記述することができる。この 2 種類の不確かさを表現するために 2 つの言語要素を導入する。*Alternative* には {}, *Optional* には [] を使い、*Archface-U* のメソッドを囲うことで不確かさを表現する。*Uncertain Archface* ではコンポーネント、コネクタ内に上記の不確かさを表す言語要素によって囲まれたメソッドを記述する。また、*Uncertain Archface* は *Certain Archface* のサブインターフェースとして記述する。この継承関係により、不確かさの着脱もスーパーインターフェースの記述の有無によって可能となる。*Uncertain Archface* のスーパーインターフェースは Java の継承ルールに則り、1 つのみとする。

4. *Archface-U* によるプログラムのプロパティ検査

本章では、*Archface-U* を記述することによって行うことができるプロパティ検査について説明する。

4.1 *Archface-U* を用いた検査の手順

図 6 に示すように、*Archface-U* を変換したことによる Partial Model によってプロパティ検査を行うことを考える。このとき、その *Archface-U* によって型検査を受けている Java コードは *Archface-U* の制約に従った実装を行う。Java コードは *Archface-U* を通じて変換された Partial Model に従っているといえる。この Partial Model のプロパティ検査を 2.1 節における方法で行うことで、Java コードのプロパティ検査を間接的に行うことが可能となる。

4.2 *Archface-U* を用いた実装の型検査

Archface-U は Java コードを型検査することにより、記述した制約に沿って実装を行っているかを調べることができる。本節では、第 3.2 節で示した例を使い、各インターフェースについてどのように型検査を行うかを示す。

4.2.1 *Certain Archface* の型検査

図 5 の例の制約を考える。コンポーネントインターフェースでは、記述されたメソッドが宣言される必要があるため、`setState` が宣言されていない場合は制約違反とな

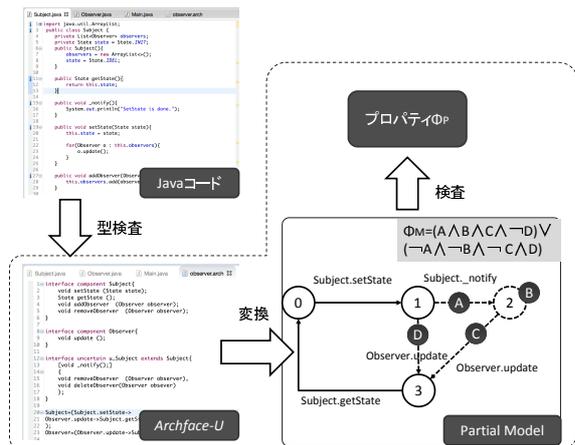


図 6 Archface-U を用いたプログラムのプロパティ検査の流れ

る。コネクタインターフェースでは、記述された接続関係が接続される必要があるため、Observer.update から Subject.getState への接続がない場合コネクタインターフェースの Observer における制約違反となる。

4.2.2 Uncertain Archface の型検査

以下では、各インターフェースでどのようなルールにおいて型検査を行っているのかを示す。

4.2.2.1 コンポーネントインターフェースの場合

図5において、_notify は Optional として定義されている。実装において _notify が宣言されていない場合、これは制約違反ではない。なぜなら、Optional の定義は「システムに組み込まれるかどうか分からない不確かさ」であるため、そのようなメソッドが定義されていないことがアーキテクチャ設計に違反しているとは言えないためである。ただし、そもそも宣言しないメソッドにおいて Optional メソッドが記述されるのは、無意味な Optional メソッドが増える原因となる。ゆえに、Optional メソッドが実装されていない場合、該当メソッドについて警告を返すことで Archface-U のリファクタリングを開発者に促す。

次に、removeObserver, deleteObserver においては、Alternative メソッドとして定義されているので、removeObserver, deleteObserver のいずれかが実装されていれば制約違反とならない。ただし、いずれのメソッドも実装されていない場合は制約違反とみなす。

4.2.2.2 コネクタインターフェースの場合

図5において、ucObserverPattern 内の _notify は Optional として定義されている。このような Archface-U を記述した場合、Subject のコントロールフロー記述がオーバーライドされる。Optional メソッドがコネクタ中に存在する場合接続関係においてそのメソッドは存在してもしなくてもよい。ゆえに、図5における実装では、Subject.setState → Subject.notify → Observer.update → Subject.getState といった接続関

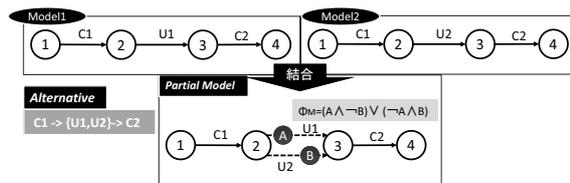


図 7 Alternative メソッドの Partial Model への変換

係か、途中の _notify を無視した Subject.setState → Observer.update → Subject.getState といった接続関係が成り立つ必要がある。

同じように、Alternative もコネクタインターフェースに記述することが出来る。この場合は {} 内に記述されているメソッドのいずれかがフローとして用いられればアーキテクチャ設計が成り立っているとする。

以上のように、Uncertain Archface を記述することで不確かさを残したまま型検査を行うことが可能である。

5. Archface-U と Partial Model 間の双方向変換

本章では第 4.1 節に示したプロパティ検査を実現するため、Archface-U から Partial Model を変換するアルゴリズムを提案する。また、将来的に Partial Model が設計として与えられた場合、設計を実装に反映するために Partial Model から Archface-U を変換するアルゴリズムも提案する。

5.1 Archface-U から Partial Model への変換

この節では、Archface-U から Partial Model への変換について述べる。Archface-U から Partial Model へ変換する際には、Archface-U のコネクタインターフェースに以下の 2 つの手順を適用する。

手順 1 Alternative, Optional の定義に従い複数の Archface およびそれを表す LTS によって Archface-U のコネクタインターフェースを表す。

手順 2 Partial Model の生成アルゴリズムに従い、共通のエッジ、ノードを実線、そうでないエッジ、ノードを破線で表現し、手順 1 で表現したモデルを表す命題論理式を記述する。

以下では、Alternative メソッド、Optional メソッドを用いた場合の例を紹介する。

5.1.1 Alternative メソッドの変換

ここでは、C1 → {U1, U2} → C2 といった Archface-U のコネクタインターフェース記述を変換する場合を考える (図 7)。まず、これに手順 1 を適用すると、Model1 (C1 → U1 → C2), Model2 (C1 → U2 → C2) の 2 つのモデルが作られる。後に手順 2 を適用することで図 7 のような Partial Model を作成することができる。

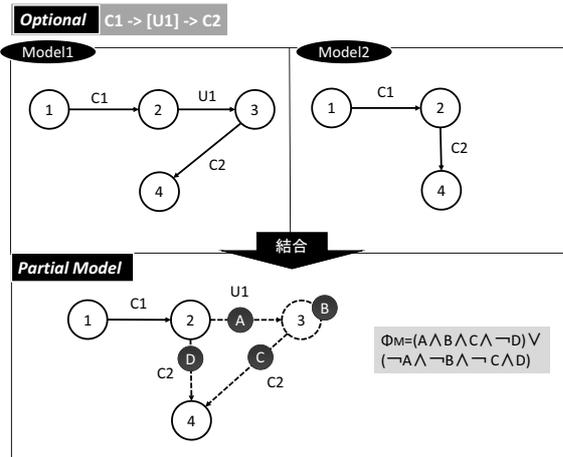


図 8 Optional メソッドの Partial Model への変換

5.1.2 Optional メソッドの変換

ここでは、 $C1 \rightarrow [U1] \rightarrow C2$ といった Archface-U のコネクタインターフェース記述を変換する場合を考える (図 8)。これに手順 1 を適用すると、Model1 ($C1 \rightarrow C2$)、Model2 ($C1 \rightarrow U1 \rightarrow C2$) の 2 つのモデルが作られる。後に手順 2 を適用することで図 8 のような Partial Model を作成することができる。ただし、Optional として指定したメソッドの直後のメソッドである C2 も破線で表現される。これは、Model1 では状態 3 から状態 4 へメッセージを送り、Model2 では状態 2 から状態 4 にメッセージを送っているが、Partial Model はソースとターゲットが異なるメソッドは同じメソッドであっても異なるものとして扱うためである。

5.2 Partial Model から Archface-U への変換

以下の手順に従い、Partial Model によって表された LTS を Archface-U のコネクタインターフェースに変換する。
 手順 1 まずアクション全てを 1 つのアクションのみが含まれる 1 つ 1 つのプロセスにする。また、この細分化したプロセスを Certain Archface のコネクタによって表す。この際、実線か破線は考慮せずに表す。
 手順 2 分解したプロセスの中で破線のアクションが含まれるプロセスのみ、同プロセスを Alternative 言語要素によって合成する。この際、開始する状態と終了する状態を一致するようにする。
 手順 3 次に、全てのプロセスを同一にし、1 つのプロセスへ連結する。これは、分解前のプロセスはそもそも同プロセスであるため可能である。
 手順 4 最後に、プロセスの表示などを削除し Archface-U の構文を整えれば、Partial Model から Archface-U への変換が完了する。

なお、手順 1 ではプロセスを細分化することで、確定しているアクションと不確かなアクションを分類することがで

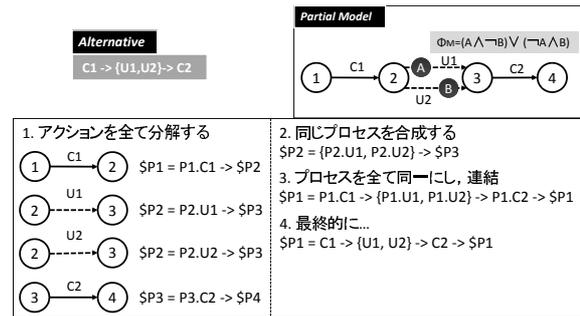


図 9 Alternative メソッドの例における変換

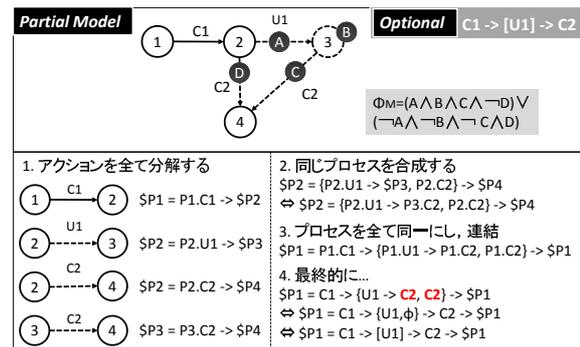


図 10 Optional メソッドの例における変換

きる。また、手順 2 において、この合成を行うことによって 1 つのプロセスに不確かなアクションをまとめることができ、後に全てのプロセスを 1 つに連結させる際に実線のプロセスと同様に扱うことが可能となる。

以下では、簡潔な例として第 5.1 節にて扱った Alternative メソッドの例と、Optional メソッドの例を逆変換することを試みる。

5.2.1 Alternative メソッドの例における変換

Alternative メソッドにおける例は、この変換において最も単純な例の一つである。変換の手順を図 9 において表す。まず、Partial Model に対して手順 1 を実行する。この時、Partial Model で表されているプロセスを \$P1 とする。また、どのアクションがどのプロセスに属しているものかを分かるようにするためにアクション名を「プロセス名. アクション名」という表現によって表している。

続いて、手順 2 について。この例では \$P2 が 2 つあるため、この 2 つを Alternative 言語要素によって合成する。

更に、プロセス同士を接続し、プロセス名を全て \$P1 にすることで Archface-U を 1 つにまとめ、手順 3 が完了する。

最後に、もともと Archface-U にはプロセス名を付与させるようにはしていないためそれを除去する手順 4 を行い、変換が完了する。

この変換後の Archface-U は確かに、第 5.1 節においてあげた例と同様のものである。

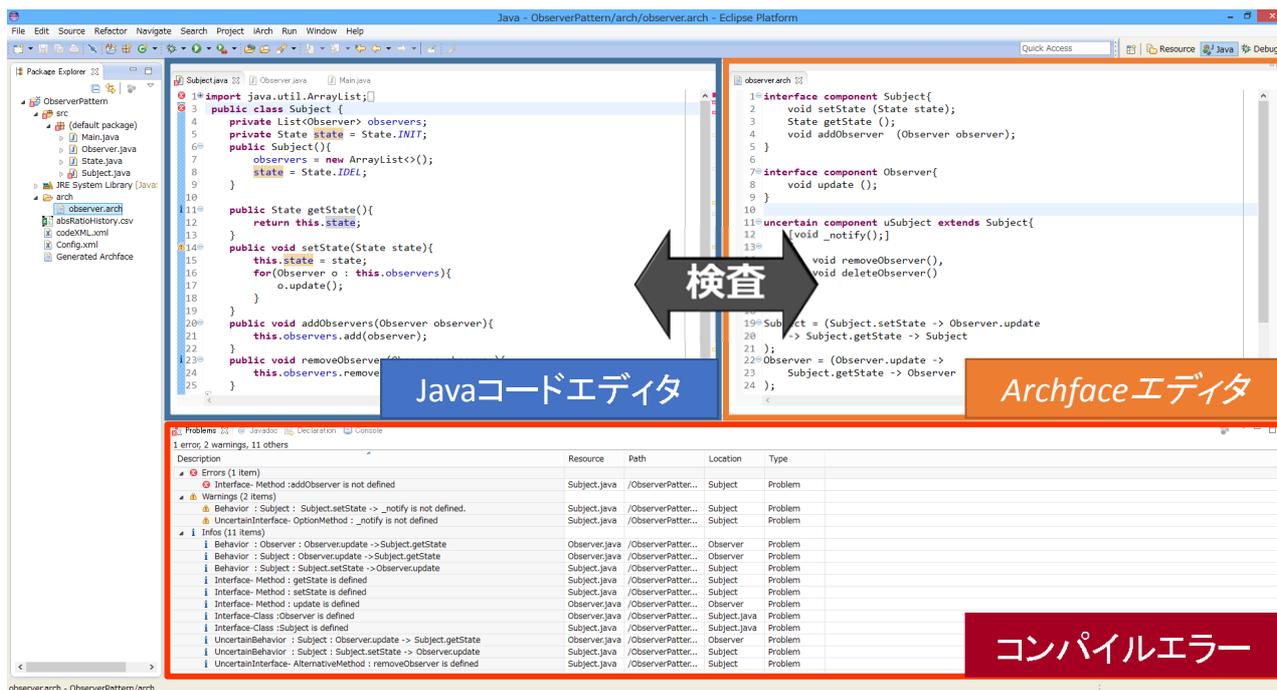


図 11 Archface-U 統合開発環境 iArch-U

5.2.2 Optional メソッドの例における変換

Optional メソッドにおける例においても、手順 1 までは、先の Alternative メソッドにおける例と同様である。

ただし、手順 2 において、 $\$P2 = P2.U1 \rightarrow \$P3$ と $\$P2 = P2.C2 \rightarrow \$P4$ は終端が異なるプロセスで終わっているため、前者のプロセスの $\$P3$ を展開し、終端を $\$P4$ に合わせる。続いて、手順 3 を行い、連結する。

次に、手順 4 を行う。もともと Archface-U は Alternative 言語要素内にアクション接頭辞 (\rightarrow) を含めることは出来ず、コンマ区切りで 1 つのメソッドしか記述ができない。ゆえに、手順 3 までで導出した $\$P1$ は Archface-U の文法から考えると正しくない。しかし、 $\{$ 中の C2 というアクションに注目すると、 $\{$ 中のいずれのアクションが選択されても必ず C2 は実行される。ゆえに、C2 は Archface-U から考えると不確かなメソッドではなく、確立したメソッドであると捉えることができるため、これを $\{$ 中から除外し、確立化する。すると、あとに残った Alternative メソッドは $\{U1, \phi\}$ であり (ϕ は空集合)、これは U1 が選ばれるかあるいは無視されるかのいずれかであるため、Optional メソッドと同値である。よって、これを変換することで、最終的には第 5.1 節における例と同様の Archface-U へ変換することが出来た。

6. ツールの実装

本研究では、iArch を第 3.2 節で述べた Archface-U による不確かさの記述、型検査を行えるように iArch-U へと拡

張し、ツールサポートを試みた (図 11)。

iArch-U は、Eclipse[2] のプラグインとして実装されている。

iArch-U では、Archface-U を XText[8] を用いた DSL (Domain Specific Languages) として定義し、Java のソースコードを JDT (Java Development Tools) によって提供されている AST (Abstract Syntax Tree) 解析 API によって解析することによって第 3.2 節にて述べた型検査を行う。型検査による違反は、コンパイル時に通常の Java ソースコードのコンパイルエラーとともに出力する。

7. まとめ

本論文では、Archface-U に不確かさを含んだインターフェースを記述し、iArch-U を用いてソースコードを型検査することによって、不確かさが実装に含まれていても安全に開発を行うことができる不確かさを包容した Java プログラミング環境を提案した。今回提案した環境を用いることで設計だけに留まっていた Partial Model で表現されていた不確かさを実装のレベルまで適用することが可能になった。

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] Allen, R. and Garlan, D.: Formalizing Architectural Connection, *Proceedings of the 16th International Conference on Software Engineering*, pp. 71–80 (1994).

- [2] Eclipse - The Eclipse Foundation open source community website. "<http://eclipse.org/home/index.php>"(2015/01/11).
- [3] Elbaum, S. and Rosenblum, D. S.: Known Unknowns: Testing in the Presence of Uncertainty, *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, pp. 833–836 (2014).
- [4] Famelis, M., Salay, R. and Chechik, M.: Partial Models: Towards Modeling and Reasoning with Uncertainty, *Proceedings of the 34th International Conference on Software Engineering*, pp. 573–583 (2012).
- [5] Magee, J. and Kramer, J.: *State Models and Java Programs* (1999).
- [6] Salay, R., Gorzny, J. and Chechik, M.: Change Propagation Due to Uncertainty Change, *Fundamental Approaches to Software Engineering*, pp. 21–36 (2013).
- [7] Ubayashi, N., Nomura, J. and Tamai, T.: Archface: A Contract Place Where Architectural Design and Code Meet Together, *Proceedings of the 32nd International Conference on Software Engineering*, pp. 75–84 (2010).
- [8] Xtext - Language Development Made Easy! "<http://eclipse.org/Xtext/>"(2015/01/10).