

不確かさを包容する統合開発環境 *iArch-U*

中村 隼也^{1,a)} 渡辺 啓介^{1,b)} 深町 拓也^{1,c)} 鷗林 尚靖^{1,d)} 細合 晋太郎^{1,e)} 亀井 靖高^{1,f)}

概要：ソフトウェア開発において、顧客の要求が曖昧であったり、設計や実装の方法が確定していないといった理由で発生する「不確かさ」は、避けることのできない問題である。そこで我々は、不確かさを抱えながらも円滑かつ高い品質でソフトウェア開発を進めるために、不確かさを包容する統合開発環境 *iArch-U* を提案してきた。この *iArch-U* では、インターフェース機構 *Archface-U* を中心として、種々の機能を活用しながら、発生した不確かさを適切にマネジメントしつつ開発を進めることができる。*iArch-U* は不確かさを含む設計と実装への対処を容易にする機能として、それぞれモデリング機能とコーディング機能を有し、また不確かさを伴ったソフトウェアの検証に貢献する機能として、単体テスト支援機能とモデル検査機能を有する。本稿では、これら *iArch-U* の種々の機能を具体例に基づいて俯瞰するとともに、その成果を踏まえた今後の課題についても述べる。

1. はじめに

ソフトウェア開発において、「不確かさ」は、絶えず変動し続けるビジネス環境や、ユーザの要求、システム運用などによって発生する。こうした不確かさは要求分析、設計、実装、テストといった様々なソフトウェア開発工程で発生しうる。従来、開発者はこのようなソフトウェア開発における不確かさをリスク管理の一つとして扱い、リスク管理表や、仕様書などにメモとして記載するなどの対処をしてきた(図1)。しかし、このような対処による記述は自然言語に依存するため、記述している不確かさが実装やテストのソースコードや設計モデルのどの箇所に存在しているのかが分からなくなることが多い。

そこで我々は、不確かさを適切に記述し、また設計や実装がその記述に従っていることを検査することで、不確かさがコードに存在しても、実装を進めることができるインターフェース機構 *Archface-U* を提案してきた[10]。さらに、我々はこの *Archface-U* を用いた不確かさのマネジメント手法[11]、*Archface-U* に基づく動的な単体テストの支援[13]、並びに静的なモデル検査手法[12]を提案してきた。

本稿では、これらの手法を実現するツールとして、*Archface-U* の開発環境である *iArch-U* の全体像を、過

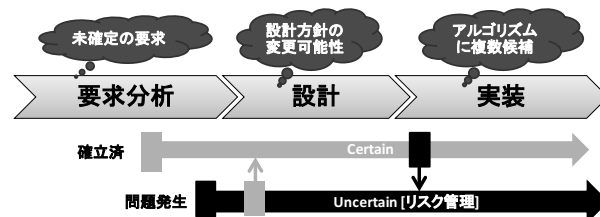


図1 ソフトウェア開発における不確かさとその従来の対処

去の提案からの拡張や発展を含め具体例に基づいて提示し、その成果を踏まえた今後の課題についても述べる。

本稿において、2章では、統合開発環境 *iArch-U* の概要を述べる。3章では、*iArch-U* を用いたソフトウェア開発の基軸となる、インターフェース機構 *Archface-U* の概要を述べる。4章ではバージョン管理システムと連携した、*Archface-U* による不確かさのマネジメントについて述べ、さらに設計工程・実装工程において不確かさに対処するための機能として、5章ではモデリング機能について、6章ではコーディング機能についてそれぞれ述べる。加えて、不確かさを含むソフトウェアの種々の性質の検証を容易にする手法として、7章では単体テスト支援機能について、8章ではモデル検査機能について述べ、9章で関連研究を示し、10章で本研究のまとめと今後の課題を述べる。

2. 統合開発環境 *iArch-U*

本章では、統合開発環境 *iArch-U* を概観し、また以降の章の説明で用いる、ソフトウェア開発において発生する不確かさの例を示す。

¹ 九州大学大学院システム情報科学府情報知能工学専攻
744 Motooka Nishi-ku Fukuoka JAPAN

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

b) watanabe@posl.ait.kyushu-u.ac.jp

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

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

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

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

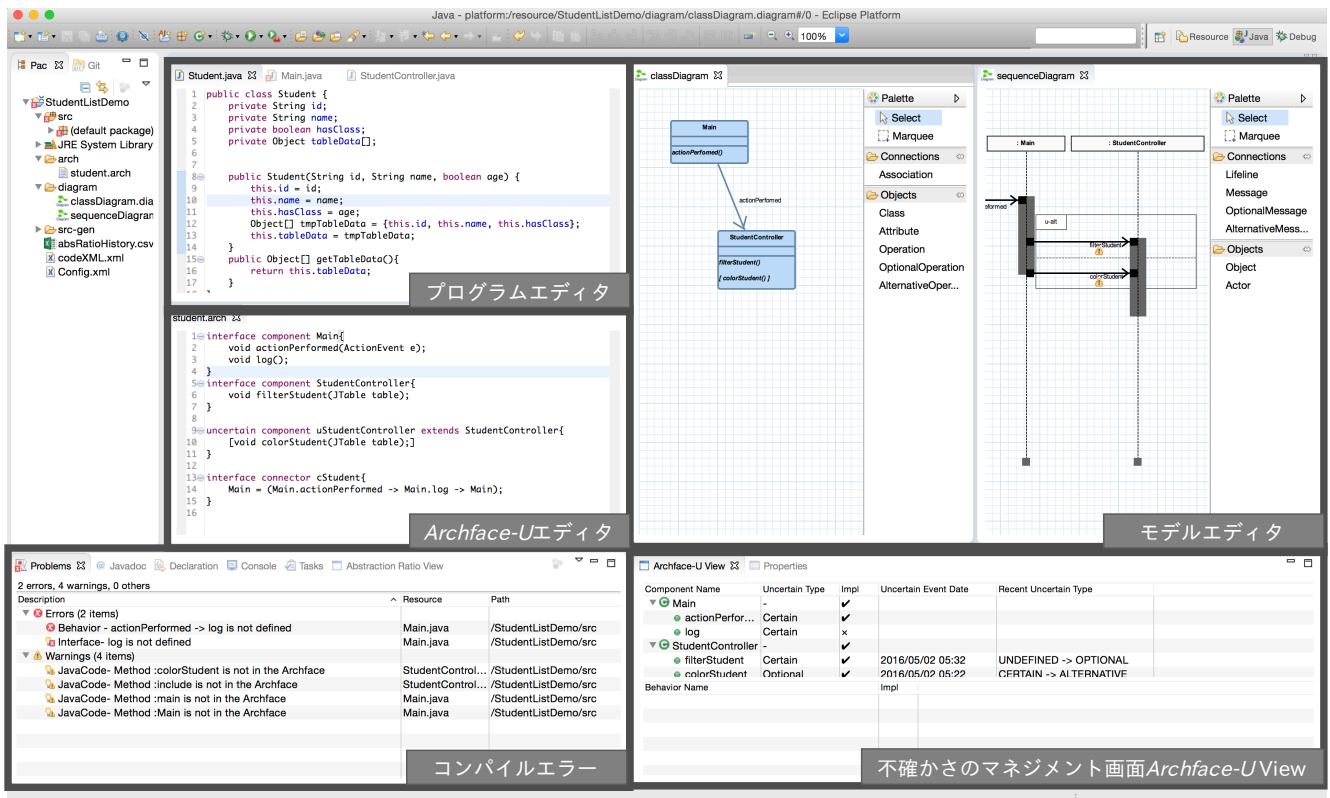


図 2 iArch-U のスナップショット

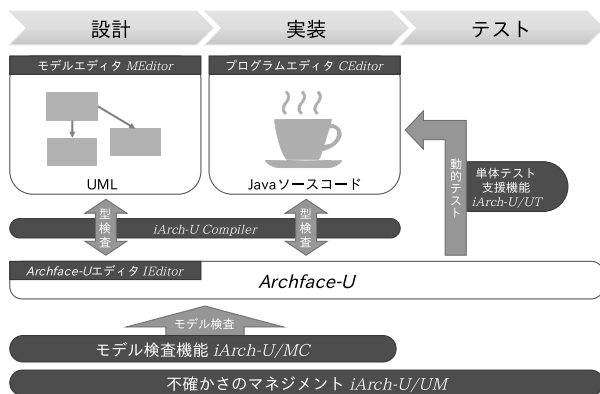


図 3 iArch-U の提供する機能

2.1 概要

1章で述べたように、iArch-U(図2)は不確かさを包容するインターフェース機構 Archface-U の統合開発環境であり、Java プロジェクトを支援の対象としている。iArch-U は、Archface-U の記述に基づいて、不確かさを包容するソフトウェア開発を支援する様々な機能を提供する(図3)。

iArch-U を使用することで、設計段階においては、UML エディタ上で不確かさを含むモデルを定義することができ、Archface-U コンパイラによって、そのモデルが Archface-U に従っているかどうかを検査される。実装段階においては、プログラムエディタ上で Java によるコーディングを行うが、コード上に不確かさが発生した場合は、その不確かさを Archface-U に記述することで、適切に記録するこ

とができ、不確かさを残したまま実装を進めることが可能になる。ここでもコードが Archface-U に従っているかどうかは Archface-U コンパイラによって検査される。

以上のように、Archface-U が設計と実装の仲立ちになっており、設計段階と実装段階で一貫した不確かさへの対処が可能になっている^{*1}。本稿では、設計や実装が Archface-U に従っているかどうかを検査することを型検査と総称する。

また、Archface-U に記述された不確かさに応じた動的な単体テスト機能や、不確かさを含むメソッドの振る舞い関係についての LTSA(Labelled Transition State Analyser) [7] に基づくモデル検査機能は、不確かさを伴うプロジェクトにおいてもソフトウェアの特定の性質を検証する上で有用である。

さらに、iArch-U はバージョン管理システムである Git [5] と連携した不確かさのマネジメント機能も備えており、以上に述べたような、不確かさの発生や解消を伴う開発の過程について、その記録と追跡を可能にしている。以降の章では、これら iArch-U の持つ機能をそれぞれ詳述する。

2.2 実装

iArch-U は Eclipse [3] のプラグインとして実装され、Archface-U は XText [9] を用いた DSL(Domain Specific Languages) として定義されている。Java ソースコードを

^{*1} iArch-U は、Archface [8] の開発環境である iArch [14] を拡張して提案されたものであり、アーキテクチャ記述を中心に、設計と実装の一貫した開発を実現する iArch の特徴を継承している。

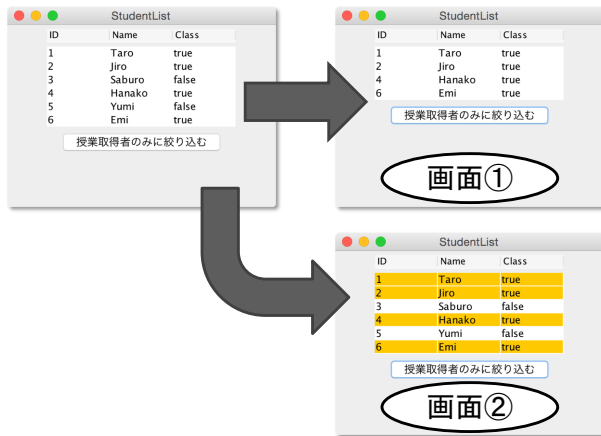


図 4 GUI アプリケーションにおける不確かさの例

JDT(Java Development Tools) によって提供されている AST(Abstract Syntax Tree) 解析 API で解析することによって型検査を行う。また, Graphiti [6] を用いて不確かさを包容する設計モデルを定義し, モデルについても型検査を可能にしている。

2.3 ソフトウェア開発における不確かさの例

本稿で *iArch-U* の諸機能の説明のために用いる, 架空の開発プロジェクトにおける不確かさの例を次に挙げる。

この例では, ある教師が担当しているクラスの生徒の内, 教師が担当している教科を受講している生徒を分かるようにする GUI アプリケーションを考える。この GUI アプリケーションは開発当初はボタンをクリックすると, 受講者ののみを表示するように生徒のリストが切り替わる (画面①) 仕様であった。

ここで, 依頼主である教師が受講をしていない生徒をリスト上から消さずに, 受講をしている生徒に色をつけて表示する (画面②) とどのようになるかを見たいと希望した。このとき, 最終的にこのアプリケーションの GUI が画面①, ②のいずれになるかは教師が要求を確定させるまで分からない (図 4)。これは教師の曖昧な要求によって生じた, 開発者が認識している「不確かさ」であるといえる。

その後, プロトタイプ実装を用いたユーザビリティテストを経て, 教師は画面②を要求した。これによって, 前述した不確かさは解消した。

3. インターフェース機構 *Archface-U*

iArch-U の各機能を述べる準備として, 本章で *iArch-U* による不確かさを包容するソフトウェア開発の基軸となる *Archface-U* の概略を述べる。

3.1 概要

Archface-U は Component-and-Connector アーキテクチャ [1] に基づいたインターフェース機構である。*Archface-*

```

1 interface component Main{
2     void actionPerformed(ActionEvent e);
3     void log(Student s);
4 }
5
6 interface component StudentController{
7     void filterStudent(JTable table);
8 }
9 uncertain component uStudentController
10 extends StudentController{
11     [void colorStudent(JTable table);]
12 }
13
14 interface connector cStudent{
15     Main = (Main.actionPerformed
16         -> Main.log -> Main);
17 }
18 uncertain connector ucStudent extends cStudent{
19     Listener = (Main.actionPerformed ->
20         {uStudentController.colorStudent,
21         StudentController.filterStudent} -> Main);
22 }

```

リスト 1: GUI アプリケーションにおける *Archface-U* によるインターフェース記述の例

U において Component は, クラスおよびメソッドを宣言し, Connector は, メソッドの呼び出し関係を記述する。開発者はモデルとコードがインターフェースである *Archface-U* に従うようにすることで, 2.1 節で述べた型検査により, 不確かさを含む設計と実装の一貫性を確保することができる。

3.2 文法

Archface-U は確立した規約を記述する *Certain Archface* と, 不確かな規約を記述する *Uncertain Archface* の 2 つのインターフェース記述によって構成される。*Uncertain Archface* 内には以下 2 種類の既知の不確かさを記述することができる。

- (1) いくつか Component の候補があるが, その中でどれを実際にシステムに組み込むかがわからないという不確かさ
- (2) ある Component について, 実際にシステムに組み込むかがわからないという不確かさ

この 2 種類の不確かさのうち, (1) を *Alternative*, (2) を *Optional* と以下呼称する。*Archface-U* 内では *Alternative* を { }, *Optional* を [] でメソッドを囲うことによって表現できるようにしている。リスト 1 に, 2.3 節の例における *Archface-U* のインターフェース記述を示す。

ここでは, コードとの対応関係に着目して, リスト 1 を例に *Archface-U* の文法を説明する。

Component インターフェースでは, 実装上の Main クラスや StudentController クラスに対して各メソッドの宣言について記述している。例えば, StudentController クラスにおいて, filterStudent メソッドはコード上で宣言される必要があるが, colorStudent メソッドは *Optional* であるため宣言しても, しなくてもよい。

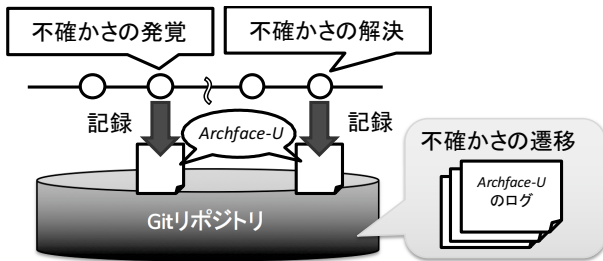


図 5 本研究における不確かさのマネジメントの概要

また, Connector インターフェースは, FSP(Finite State Process) [7] をベースとした記法によりメソッドの呼び出し関係を記述している. 例えば, `actionPerformed -> log` という表記は `actionPerformed` メソッドにおいて `log` メソッドが呼び出されていることを表す. ここで, uncertain connector である `ucStudent` に着目する. `colorStudent`, `filterStudent` が *Alternative* として定義されているため, 実質 `actionPerformed -> colorStudent` あるいは `actionPerformed -> filterStudent` のいずれかのメソッド呼び出し関係が成り立てばよい.

一方, モデルについては, クラス図とシーケンス図を拡張することで不確かさを記述できるようになっており, それによって *Archface-U* との対応が取られている. 詳細は 5 章に述べる.

4. 不確かさのマネジメント

本章では, 3 章で述べた *Archface-U* と, バージョン管理システムである Git を用いて不確かさをどのようにマネジメントするかを述べる.

4.1 概要

本章で述べる不確かさのマネジメント法では, *Archface-U* と Git を用い, 不確かさを管理する. この方法では, 不確かさを記録, 追跡することによって不確かさをマネジメントする. 具体的には, 不確かさをマネジメントしたいプロジェクトを Git で管理し, 不確かさが発見されたり, 解消されたりした場合にその旨を *Archface-U* に記述しコミットする (図 5). 以降では, 本マネジメント法を用いた不確かさの記録, 追跡方法について具体例を用いて説明する.

4.2 不確かさの記録

本マネジメントにおいては, 不確かさが要求, 設計モデル, ソースコード等に見つかった場合, あるいはすでに把握していた不確かさが解消された場合, その旨を *Archface-U* に記述することによって記録する. 具体的には, 不確かさの該当箇所である *Archface-U* の Component や Connector を *Optional* や *Alternative* に変更することによって表現する.

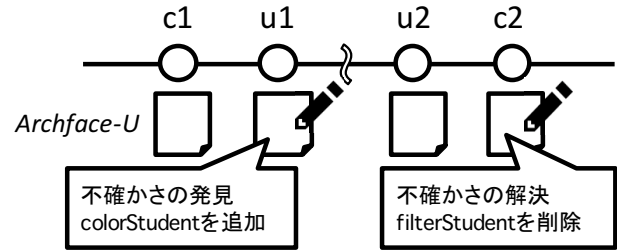


図 6 GUI アプリケーションにおけるマネジメントの例

```
1 interface component Main{
2     void actionPerformed(ActionEvent e);
3     void log(Student s);
4 }
5
6 interface component StudentController{
7     void filterStudent(JTable table);
8 }
9
10 interface connector cStudent{
11     Main = (Main.actionPerformed
12         -> Main.log -> Main);
13     Listener = (Main.actionPerformed ->
14         StudentController.filterStudent -> Main);
15 }
```

リスト 2: 不確かさが発生する前の GUI アプリケーションにおける *Archface-U* によるインターフェース記述の例

以下では, 2.3 節の例を用いて本マネジメント法を用いて不確かさを記録するシナリオを示す. この例において, 不確かさは画面①と画面②のいずれにするかを決めることが出来ないというものである. このとき, *StudentController* クラスにおいて, 生徒のリストを講義の受講者のみにするためのメソッドを `filterStudent`, 受講している生徒に色をつけるためのメソッドを `colorStudent` と定めたとする. このとき, 不確かさが発生する前の *Archface-U* はリスト 2 のように記述することができる. (図 6 コミット c1)

ここで, 教師の要求によって先述の GUI に関する不確かさが発生した. 開発者は *Archface-U* に `colorStudent` によって機能を実装するかもしれないという不確かさを記述し, コミットを行う (図 6 コミット u1). コミット u1 における *Archface-U* はリスト 1 のように記述できる. その後, `colorStudent` の実装を行いながら通常の Git のようにコミットしながら開発を進める.

最終的に, プロトタイプが完成し, 教師に確認してもらい, 画面②の GUI に確定した. このとき, 開発者は画面②用の実装を行い, *Archface-U* の `colorStudent` を *Certain* にし, `filterStudent` の記述を *Archface-U* から削除し, コミットを行うことによって不確かさの解消を表現する. (図 6 コミット c2)

4.3 不確かさの追跡

本サポート機構では, 不確かさの追跡を Git のコミットログとそれに付随するコミットメッセージ, および

U を観察することによって追跡できる．観察する各々において，以下の様な情報を取得できる．

- コミットログ：不確かさの発生・解決日時，発見・解決者，不確かさが存在している・存在していた成果物
- コミットメッセージ：不確かさの発生・解決理由，発生・解決した開発工程
- *Archface-U*：不確かさが存在するメソッドやクラス，不確かさの種別

なお，*Archface-U* の不確かさの種別とはその不確かさが Component か Connector のものか，また *Alternative* か *Optional* かを示す．

2.3 節における例において不確かさの追跡を行うことを考える．図 6 コミット u2 後，教師が再び GUI を画面①に戻してほしいという要求が発生したとする．もし，開発者が画面①を実装した開発者でない場合，Git のコミットメッセージや差分をマニュアルで追跡する必要がある．しかし，4.2 節のように不確かさについてのログを *Archface-U* において残しておけば，ボタンのリスナである `actionPerformed` を含む Connector の *Archface-U* の差分のみを追跡すればよい．そうすることにより，画面①を実装していた `filterStudent` というメソッドを見つけることができる．また，不確かさはコミット単位で記録されているため，該当コミットにおいてブランチを作成し，機能を変更したいブランチへマージを行うことによって `filterStudent` を適用することができる．

5. モデリング機能

本章では，2.3 節で示した具体例を用いて，モデリング機能をどのように利用できるかを述べる．

5.1 概要

我々が提案する不確かさを包容するモデリングでは，クラス図とシーケンス図を拡張し，*Archface-U* における *Optional* と *Alternative* の不確かさを表現ができる．また，*Archface-U* コンパイラが 2.1 節で述べたモデルに対する型検査を行い，*Archface-U* に従ったクラス図，およびシーケンス図を描画できるようにする．

5.2 不確かさを包容したモデルの表現方法

我々が提案するクラス図，シーケンス図における *Alternative* と *Optional* の表現方法を示す．

クラス図では，*Archface-U* の記述と同様，クラス中のメソッドを `[]`，あるいは `{ }` で囲うことで *Optional* および，*Alternative* の不確かさを表現する（図 7(a)）．ただし，*Archface-U* はフィールドの表現をサポートしていないため，クラス図中の属性については無視をする．

シーケンス図では，不確かさを複合フラグメントとして表現をする．ただし，通常の「alt」や「opt」と区別する

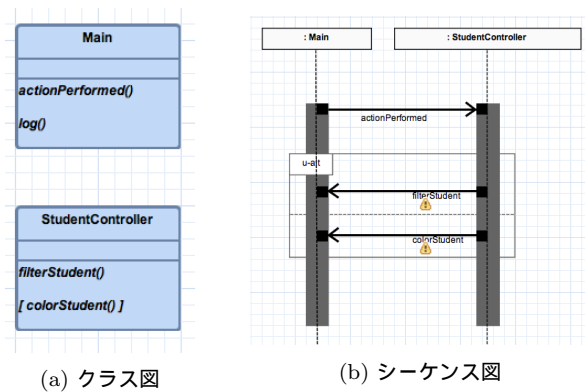


図 7 不確かさを包容したモデリングの例

ために「u-alt」や「u-opt」を用いて表現する（図 7(b)）．なお，ガードは記述しない．

5.3 不確かさを包容したモデリングの流れ

2.3 節における例におけるモデリングの例を示す．まず，リスト 1 のように図 4 の不確かさを含んだ設計を *Archface-U* によって行う．後に，我々が提供するモデルエディタによって図 7 のように不確かさを含んだ設計モデルを記述する．この際，モデルの型検査により各モデルは必ず *Archface-U* の内容を含むモデルの記述を行うことができる．

次に，アプリケーションの GUI が画面②に決定したとする．このとき，画面①を生成する `filterStudent` の定義と呼び出しは不要であるため，モデルから削除する．*iArch-U* には，このような不確かさが確定したことをモデル，*Archface-U* ともに通知する機能がある．この機能を `filterStudent` に用いると，モデルから `filterStudent` から削除され，同時に，*Archface-U* 内の `filterStudent` の Component，あるいは Connector が削除される．また，この逆の機能として，今まで通常の *Archface* として定義している Component や Connector を不確かにする際に同時に *Archface-U* にもその変更を反映する機能も存在する．これらの機能を用いることによってモデルと *Archface-U* のトレーサビリティを保ちつつ不確かさの管理を行うことができる．

6. コーディング機能

本章では，*Archface-U* コンパイラによる実装の型検査を中心とする，コーディング機能について述べる．

Archface-U コンパイラはコードのコンパイルが行われると同時に *Archface-U* をコンパイルし，型検査をコードに対して行う．具体的には，*Archface-U* の Component や Connector のインターフェース記述が，コードのメソッドの定義や呼び出しとして実装されているかどうかを検査され，インターフェース記述に従わない実装を行っている場合，型検査違反として *Archface-U* コンパイラはコンパイ

Name	Selection
▼ StudentController	
▼ Optional Uncertainty	
colorStudent(JTable table) : void	☒

図 8 単体テスト支援機能の GUI

ルエラーを返す．さらに，*Alternative* や *Optional* といった不確かな制約において，現在の実装がどのような状態になっているかの情報を示す．例えば，リスト 1 の例において `colorStudent` は定義してもしなくても良い *Optional* なメソッドであるが，これが現在定義されているかどうかを示すことができる．

Archface-U とコードの間の詳細な対応関係は，3 章に示した．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 Eclipse 上に出る．

7. 単体テスト支援機能

本章では，*iArch-U* のもつ単体テスト支援機能について 2.3 節の例に即して述べる．

7.1 概要

2.3 節の例において，開発者が最終的にいずれの画面を採用するか決定する（不確かさの解消）ためには，画面①，②の実装が完了した後に，依頼主である教師にそれぞれの画面を見せ，あるいは試用させることで，いずれの画面が依頼主のニーズに適しているかを検証する必要がある（ユーザビリティテスト）．それぞれの画面を柔軟に切り替えて表示するためには，例えば実装中に条件分岐を追加するといった方法が考えられるが，不確かさの解消後にはその条件分岐は不要になるため，むだな労力が必要になること，ソースコードの意図が分かりにくくなり，保守性が低下することなどの問題がある．

そこで，*iArch-U* の単体テスト支援機能を使用すると，*Archface-U* に定義されたそれぞれの不確かさに対して，開発者が GUI 上で仮の選択をすることができる（図 8）．このとき *iArch-U* は，その仮の選択を再現するようにソフトウェアの挙動に変更を加える．この挙動の変更は，少数のファイルの追加によってのみ行われ，直接的なソースコードの変更を伴わない．

例えばこの例では，現行の実装が画面①であったとしても，仮の選択としてリスト 1 における `uStudentController` の *Optional* を有効とし，`ucStudent` の *Alternative* で `colorStudent` を選ぶことで，実装に変更を加えることなく，画面②を表示することができる．

ここではユーザビリティテストの例を挙げたが，ユースケースによっては複雑なアルゴリズムの挙動の観察や性能の測定，ハードウェアの動作検証といった動的なテストを必要とする課題の解決にこの機能が役立つ可能性がある．

```

1 @Aspect
2 public class U {
3     @Around("call(void StudentController
4         .filterStudent(JTable)) && args(table) &&
5         withincode(void Main.actionPerformed(ActionEvent))")
6     public void filterStudent(JTable table) {
7         StudentController.colorStudent(table);
8     }
9 }

```

リスト 3: 自動生成されるアスペクトの例

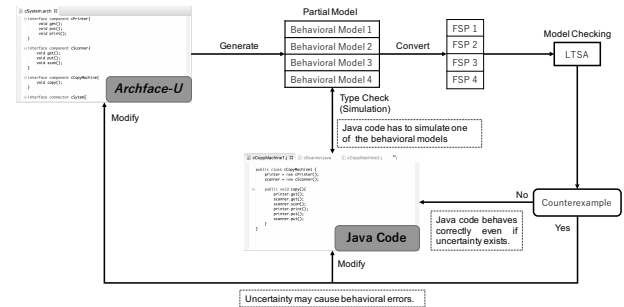


図 9 不確かさを包容したモデル検査

7.2 実装

ソフトウェアの挙動の変更は，AspectJ [2] の文法によるアスペクトのコードを自動的に生成することで実現している．例えば，7.1 節の例において生成されるコードはリスト 3 のようになる．まず，`uStudentController` の *Optional* は有効としているので，処理を置き換える必要はない．一方，`ucStudent` の *Alternative* に対する仮の選択を反映する必要があるので，`Main` クラスの `actionPerformed` メソッドにおいて呼び出される `filterStudent` メソッドの処理を `colorStudent` メソッドの処理に置き換えている．

現行の *iArch-U* の実装では，Uncertain Component のみサポートしており，Uncertain Connector のサポートは今後の課題である *2 ．

8. モデル検査機能

本章では，*iArch-U* のモデル検査機能について述べる．この機能では，Connector インターフェースに記述されたメソッドの振る舞い関係を自動でモデル検査する．Connector インターフェースをモデル検査して問題が無ければ，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安全なものであることが保障される（図 9）．

8.1 コードの安全性の向上

Archface-U の記述をモデル検査することによりコードの安全性が保障される．

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，

*2 現行の実装ではここで挙げた例は動作しない．

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、設計通りの安全な開発が進められているといえる。

しかし、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コードも同じ危険性を持つことになる。*Archface-U*には不確かさを記述することができるが、この不確かさによりコードの実装は設計とは異なるものになる可能性があり、設計段階では存在しなかったバグが発生することにつながる。

本機能はこのような問題を解決するものである。本機能は、*Archface-U*に記述されたメソッドの呼び出し関係をモデルとして、それが有る性質を満たしている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*を用いた開発で不確かさが存在してもコードの安全性を保障することが可能となる。

8.2 モデル検査器 LTSA の概要

Connector インターフェースは FSP に準拠した文法で記述されるため、本機能では FSP をモデル記述言語として用いるモデル検査器 LTSA を利用する。LTSA による検査では、Compose, Safety, Progress と呼ばれる三つのプロパティを検査することができる。Compose 検査は、複数のプロセスが並列に動作したときの挙動を検査する。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、あるアクションがやがては実行されるかを検査する。

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、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。

8.3 状態爆発問題への対処

コードをモデル検査する場合はコードの量が多くなるほど状態爆発が起こりやすい。本機能では、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。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

```

1 interface component cPrinter {
2     public void get();
3     public void put();
4     public void print();
5     [public void utility();]
6 }
7
8 interface component cScanner {
9     public void get();
10    public void put();
11    public void scan();
12    [public void utility();]
13 }
14
15 interface component cCopyMachine {
16     public void copy();
17 }
18 interface connector cSystem (
19     cCopyMachine P, cCopyMachine Q,
20     cPrinter printer, cScanner scanner) {
21
22     GET = (printer.get -> scanner.get);
23     PUT = (printer.put -> scanner.put);
24     COPY = (scanner.scan -> printer.print);
25
26     P.copy = (GET -> COPY -> PUT -> P.copy);
27     Q.copy = (GET -> COPY -> PUT -> Q.copy);
28 }
29
30 interface connector uSystem
31     extends cSystem (
32         cPrinter printer, cScanner scanner) {
33
34     GET = ({printer.get -> scanner.get,
35           scanner.get -> printer.get});
36 }

```

リスト 4: printer-scanner システムの *Archface-U* 記述

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なりすぎると状態爆発を起こす可能性がある。

8.4 ユースケース

printer-scanner システムは、二つのプロセス P と Q が共有資源であるスキャナとプリンターのロックを獲得してコピーを行い、スキャナとプリンターのロックを解放するシステムである。ここで、コードの実装中にプログラマが二つのプロセスのロックの獲得順序に対して疑問を持ったとする。すなわち、二つのプロセスのロックの獲得順序の組み合わせは計四通り考えられるが、どの組み合わせがいいのか、またはどの組み合わせでも構わないのかわからないという状況になったとする。

このときプログラマはコードを変更してどの組み合わせがいいのかを確かめていくのではなく、どのロックの獲得順序の組み合わせを用いたらいいのかわからないという不確かさをリスト 4 のように *Archface-U* に記述する。ただし、リスト 4 のようなメソッドの呼び出し関係のマクロ的な定義や不確かさは現時点の *Archface-U* ではサポートされておらず今後の課題である。*Archface-U* に不確かさが記

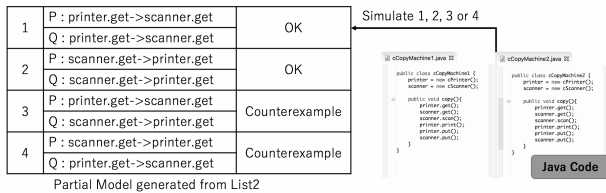


図 10 Partial Model とそれに従うコード

述されると自動で FSP が生成され、三つのプロパティに関してモデル検査が行われる。この場合はモデル検査の結果はデッドロックを引き起こす可能性があることを示す。図 10 の反例よりプロセス P と Q のロックの獲得順序が違えばデッドロックを起すことがわかる。

以上より不確かさが解消されたためプログラマは Archface-U の記述を P と Q のロックの獲得順序が同じになるように変更して、uncertain connector の不確かさを消して開発を続けることができる。もし、モデル検査の結果が問題なしだった場合は、どの組み合わせでコードが実装されても大丈夫であることがわかる。つまり、後々ロックの獲得順序が指定されてコードを変更することがあっても、コードの問題が発生しないことが保障される。このように、不確かさが発生することにより考えられるコードの分岐を自動でモデル検査することにより、その不確かさが及ぼす影響を考慮しながら開発を進めることができる。

9. 関連研究

本研究の関連研究として、Famelis らの研究が挙げられる。Famelis らが提案したのは、不確かさが存在する場合の設計を支援する統合開発環境 MU-MMINT である [4]。MU-MMINT は Archface-U と同等の表現が可能である、不確かな設計モデル Partial Model を基盤とし、モデルに対する性質の検証や変形、不確かさの解決といった一連の操作をサポートすることで、不確かさの解決を延期しつつ開発を継続することを可能にしている。

我々の研究との相違点としては、Famelis らが特にステートマシンによって表現される設計の不確かさに着目し、開発者が Partial Model を容易に扱えるよう、不確かさをグラフィカルに表現するなどの有用な機能を提供しているのに対し、我々は設計・実装・テストといったソフトウェア開発の複数の工程にまたがった統合的な不確かさへの対処を試みている点が挙げられる。

10. おわりに

本稿では、我々が提案してきた iArch-U の持つ様々な機能を俯瞰し、不確かさの統合的な管理と対処を実現していることを示した。

今後の課題としては、第 1 に、iArch-U のリリースが挙げられる。Eclipse のプラグインとして、オンライン上で

の提供を予定しており、使用方法に関するドキュメントの整備などを予定している。

第 2 に、OSS プロジェクトを用いた iArch-U の評価が挙げられる。OSS プロジェクトにおいて発生した不確かさを抽出し、iArch-U のもつ機能がその不確かさの対処にどのように貢献しているのかを検証することによって、実用のプロジェクトにおける有用性や性能の観点から iArch-U の評価を行い、改良すべき点などを明らかにしたいと考えている。

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] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pp. 71–80, 1994.
- [2] The aspectj project. <https://eclipse.org/aspectj/>.
- [3] Eclipse - the eclipse foundation open source community website. <https://eclipse.org/>.
- [4] Michalis Famelis, Naama Ben-David, Alessio Di Sandro, Rick Salay, and Marsha Chechik. Mu-mmint: an ide for model uncertainty. In *Proceedings of the 37th International Conference on Software Engineering*, pp. 697–700, 2015.
- [5] Git. <http://git-scm.com/>.
- [6] Graphiti home. <https://eclipse.org/graphiti/>.
- [7] Jeff Magee and Jeff Kramer. *State Models and Java Programs*. Wiley, 1999.
- [8] Naoyasu Ubayashi, Jun Nomura, and Tetsuo Tamai. Archface: A contract place where architectural design and code meet together. In *Proceedings of the 32nd International Conference on Software Engineering*, pp. 75–84, 2010.
- [9] Xtext - language engineering made easy! <https://eclipse.org/Xtext/>.
- [10] 深町拓也, 鶴林尚靖, 細合晋太郎, 亀井靖高. 不確かさを包容する Java プログラミング環境. 情報処理学会研究報告ソフトウェア工学研究会報告, Vol. 2015-SE-189, No. 21, pp. 1–8, 2015.
- [11] 深町拓也, 鶴林尚靖, 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. ソフトウェア工学の基礎, Vol. 41, pp. 47–52, 2015.
- [12] 中村隼也, 深町拓也, 鶴林尚靖, 細合晋太郎, 亀井靖高. Ltsa 連携による不確かさを包容した自動モデル検査. 情報処理学会研究報告 2016-SE-191, No. 21, 2016.
- [13] 渡辺啓介, 深町拓也, 鶴林尚靖, 細合晋太郎, 亀井靖高. Aspectj による不確かさを包容した単体テスト環境. 電子情報通信学会 信学技報 KBSE2015-58, pp. 57–62, 2016.
- [14] 艾迪, 鶴林尚靖, 李沛源, 李宇寧, 細合晋太郎, 亀井靖高. 設計抽象化のためのリファクタリングパターン. 情報処理学会研究報告ソフトウェア工学研究会報告, Vol. 2014-SE-185, pp. 1–8, 2014.