

# フレームワークサンプルアプリケーションを利用した 実行シナリオの実装支援手法

縄江 保宏<sup>1,a)</sup> 新田 直也<sup>1,b)</sup>

**概要:** 近年、アプリケーションの設計と実装の再利用性を高める仕組みとしてアプリケーションフレームワークが広く用いられ成果をあげている。しかしながら、フレームワークを利用するには様々な取り決めや制約が存在し、所望の振る舞いを矛盾なく実装するのが困難となる場合がある。そこで本研究ではフレームワークのサンプルアプリケーションを利用して、与えられた実行シナリオの実装を支援する手法を提案する。具体的には、実装したい実行シナリオと近い振る舞いをするサンプルアプリケーションを選び、その中の実行シナリオと競合する振る舞いを実装している箇所を特定、変更することによって支援を行う。本稿では、複数のフレームワークと複数の実行シナリオに対して本手法を適用し、その有効性を確認した。

## 1. はじめに

近年、さまざまな分野のソフトウェア開発においてアプリケーションフレームワークの利用が一般化しつつある。アプリケーションフレームワーク<sup>[1]</sup>(以下、本稿ではフレームワークとよぶ)は、主にオブジェクト指向技術に基づいて設計及び実装を再利用する仕組みであり、アプリケーションコードにおける被呼び出し側だけでなく、メインループなどの呼び出し側の制御機構も柔軟に再利用できるという特徴を持つ。そのため、アプリケーションの開発において適切なフレームワークを選択すれば、設計や実装に要する工数を効果的に削減することが可能である。

しかしながら、フレームワークはアプリケーション内部の主要なリソースや制御機構を占有してしまうため、利用する上で固有のさまざまな取り決めや制約が発生して、所望の振る舞いを実装するのが困難になったり、場合によっては実装できない機能が生じることもある。特に、フレームワークについての開発者の知識が不足していたり、フレームワークに関するドキュメントの整備が行届いていない場合は、実装に要する工数の正確な見積もりが困難になったり、実装行程まで実装できない機能が存在することに気づかないといった状況も起こり得る。

そこで、本研究ではフレームワークのサンプルアプリケーションを利用して、要求仕様として与えられた実行シナリオが実装可能かどうかを判定し、実装可能ならその実装を

支援する手法を提案する。本手法は、サンプルアプリケーションのコードの中で実装したい実行シナリオと競合している部分を特定し、競合を解消するようなコードの変更方針を提示することによって実装の支援を行う。

本稿では本手法の適用事例として、2つのゲームフレームワークを対象に3D格闘ゲームの実行シナリオが本手法の支援によって実装できるか否かの検証を行った。その結果、いずれのフレームワークを再利用した場合でも適切な実装方針を得られることがわかった。特に、我々の先行研究[2]で実装が困難と判定された事例でも本手法の方針に従うと実装できることが判明した。ただし今回の事例研究では、ほとんどすべての作業を人手で行った。今後、手法の一部を自動化しツールによって支援できるように手法の改善を行っていく予定である。また、より多くのフレームワークに対して本手法を適用していくと同時に、実用規模のアプリケーションの開発にも適用して、本手法の有効性の検証を行っていく予定である。

## 2. フレームワークを用いた開発の特徴

フレームワークは、特定のドメイン内で繰り返し出現するオブジェクト間の協調やアプリケーション全体の制御構造などを抽象化した再利用可能なクラス群である。これらのクラスにはアプリケーション側で変更可能なホットスポット<sup>[3]</sup>が抽象メソッドとして用意されており、アプリケーション開発者は、このような抽象メソッドをアプリケーション側で作成した子クラスでオーバーライドすることによってアプリケーション固有の振る舞いを実装する。このとき、アプリケーション側で実装したメソッドは実行

<sup>1</sup> 甲南大学大学院 自然科学研究科  
Graduate School of Natural Science, Konan University  
a) m1224003@center.konan-u.ac.jp  
b) n-nitta@konan-u.ac.jp

中にフレームワーク側のコードから呼び出されることになる。このような制御の仕組みを一般に**制御の反転**とよぶ。現在フレームワークは、Webアプリケーションなどさまざまなドメインにおいて開発されており、既存の適切なフレームワークを再利用することによって、アプリケーション開発における設計および実装工程を効率化することができる。

しかしながら一般にフレームワークの利用においては、

- (1) 適切に再利用できるようになるための習熟に時間を要する、
- (2) アプリケーション内部の主要なリソースや制御機構が占有されてしまうため、利用する上で固有のさまざまな取り決めや制約が発生して、所望の振る舞いを実装するのが困難になったり、場合によっては実装できない機能が生じる、

などといった問題点が指摘されている(文献[1],[4],[5],[6]参照)。本研究ではこれらのうち後者の問題に着目する。

フレームワークではアーキテクチャや設計に関するドキュメントが用意されているのが通常であるが、その整備が不十分であったり保守が行届いていないといった場合も多く見受けられる。また、たとえドキュメントが整備されていたとしても、実装する上での細かな取り決めや制約などは記載されていない場合がほとんどで、そのため実装に要する工数の正確な見積もりが困難になったり、場合によっては実装途中で実装できない機能の存在に気づくといったこともある。

一方、フレームワークには複数のサンプルアプリケーションが提供されていることがほとんどであり、アプリケーションの実装においてこれらサンプルアプリケーションのコードが参考にされる場合が多い。そこで本研究では、実装したいアプリケーションの要求仕様が実行シナリオの形式で与えられることを前提に、与えられた実行シナリオと近い振る舞いをするサンプルアプリケーションを選んで、アプリケーションの実装の支援を行う手法の構築を目指す。具体的には、フレームワークのリソース占有に起因する実行シナリオとサンプルアプリケーションの間の競合に着目し、競合を解消するようにサンプルアプリケーションのコードの変更を繰り返すことによって、目的とするアプリケーションの実装を得るアプローチをとる。このとき、同時に実装不可能性についての判定も行う。

### 3. 諸定義

本研究で提案する手法は競合の解消に基づいて構築されている。そこで本節では、以下の3つのレベルの競合について議論する。

- (1) 実装したい複数の実行シナリオの間の競合。
- (2) 実装したい実行シナリオとサンプルアプリケーションの外的振る舞いの間の競合。
- (3) 実装したい実行シナリオとフレームワークのリソース

占有機構の間の競合。

これらの競合をモデル化するため、実行シナリオおよびサンプルアプリケーションの外的振る舞いをイベントモデルを用いて表現する。また、フレームワークのリソース占有機構をサンプルアプリケーションの実行モデルを用いてモデル化する。

#### 3.1 イベントモデルおよび実行モデル

実行シナリオの例として、3D格闘ゲームにおいて典型的な5つの実行シナリオの例を図1に示す。これら実行シナリオは以下で説明するイベントモデルによってモデル化される。代表的なイベントモデルとしては有限状態遷移モデルに基づくものが挙げられるが、本研究では状態を明示するモデルを採用しない。そのかわりに、内部状態やその間の遷移に相当する部分を自然言語を用いて表現する。本研究における**イベントモデル** $M = (E, G, A, \tau)$ は、入力および内部イベントの集合 $E$ 、ガードの集合 $G$ 、アクティビティの集合 $A$ 、イベントおよびガードからアクティビティの部分集合への写像 $\tau: E \times G \rightarrow 2^A$ によって構成される。 $\tau(e, g)$ は、イベント $e$ が発生したとき条件 $g$ が満たされていた場合に実行されるすべてのアクティビティからなる集合を表す。 $E, G, A$ の各要素はそれぞれ自然言語によって記述され、内部状態およびその間の遷移は、 $G$ および $A$ の中で間接的に規定される。以下では、 $a \in \tau(e, g)$ を満たす組 $(e, g, a)$ を $M$ の**イベント応答**とよぶ。本イベントモデルを用いて、3D格闘ゲームの実行シナリオをモデル化した例を図1に示す。これらのシナリオはキャラクターの座標や速度などが連続的な値をとり、またその値がリアルタイムで変化するため、通常の有限状態遷移モデルでは適切にモデル化できない例となっている。本研究では、実装したい実行シナリオおよびサンプルアプリケーションの外的振る舞いとともにこのイベントモデルで表現することによって、それらの間の競合のモデル化を行う。

次にフレームワークのイベント処理機構およびリソース占有機構をモデル化するためのサンプルアプリケーションの実行モデルについて説明する。以下ではサンプルアプリケーション $A$ を固定して考える。このサンプルアプリケーションの振る舞いを表すイベントモデル $M_A = (E_A, G_A, A_A, \tau_A)$ の各イベント応答 $\langle e, g, a \rangle$ に対して、 $e$ が発生したとき $g$ が満たされていた場合に必ず呼び出されるアプリケーション側のメソッドが存在するときそのメソッドを**イベントハンドラ**といい $H(e, g)$ で表す。ただし、フレームワーク側に必ず呼び出されるメソッドが定義されていた場合であっても、そのメソッドをアプリケーション側でオーバーライド可能な場合は、そのメソッドが便宜上アプリケーション側に存在するものとして扱い、同様に $H(e, g)$ で表す。なお、いずれの場合に相当するメソッドも存在しないときは $H(e, g) = \emptyset$ とする。また、 $a$ の実行によって更新されるフ

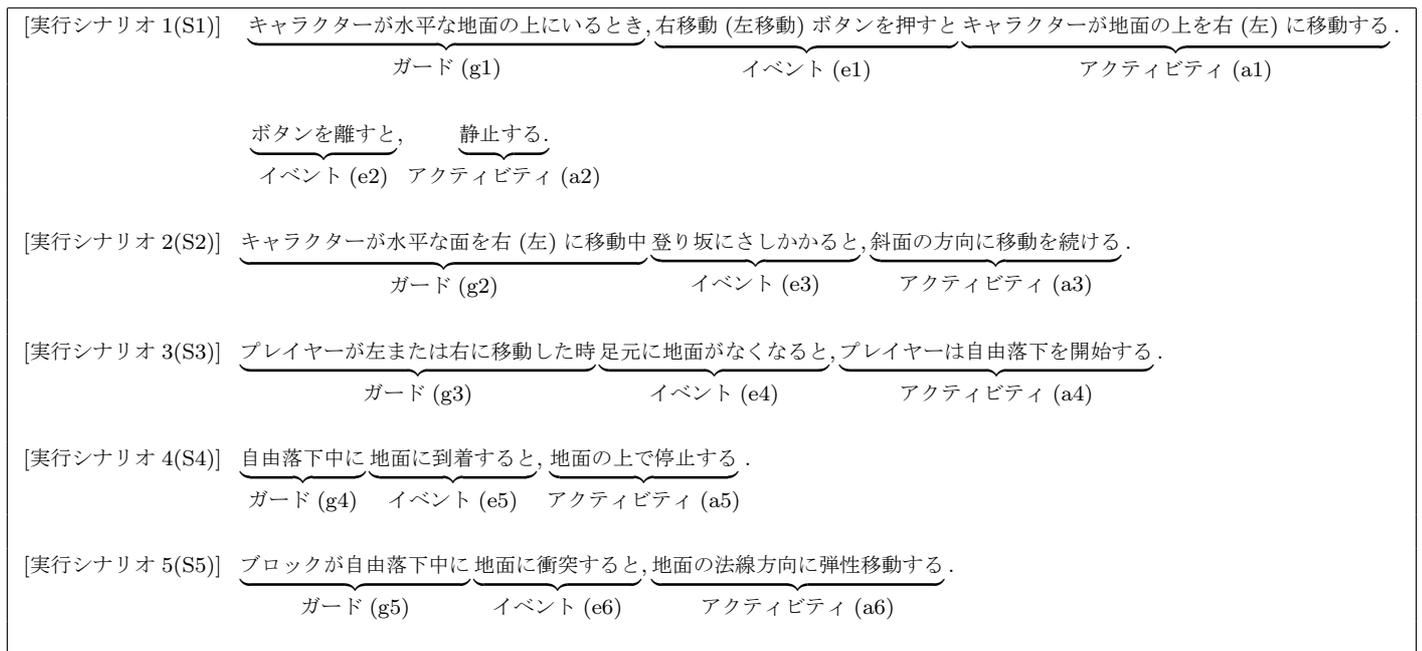


図 1 実行シナリオの例

フレームワークの占有リソースの集合を  $R(a)$  とおく。占有リソースとは実行時にフレームワーク側クラス内のフィールドのみで保持されているオブジェクトのことをいう。ここでサンプルアプリケーションのある実行  $p$  を考える。このとき、一般に  $p$  にはアプリケーション側コードからのフレームワーク側コードの呼び出しと、フレームワーク側コードからのアプリケーション側コードの呼び出しの双方向の呼び出しが含まれる。特に後者のタイプの呼び出しは、制御の反転と呼ばれる。ここで、 $p$  におけるアプリケーション側とフレームワーク側間の情報の流れとして以下の 6 通りのものを考える。

- アプリケーション側からフレームワーク側への入力:
  - アプリケーション側のコードからフレームワーク側のコードの呼び出しとその引数。
  - フレームワーク側のコードからアプリケーション側のコードの呼び出し (制御の反転) 時の戻り値。
  - フレームワーク側のコード内で宣言された変数へのアプリケーション側のコード内での代入。
- フレームワーク側からアプリケーション側への出力:
  - フレームワーク側のコードからアプリケーション側のコードの呼び出し (制御の反転) とその引数。
  - アプリケーション側のコードからフレームワーク側のコードの呼び出しにおける戻り値。
  - フレームワーク側のコード内で宣言された変数のアプリケーション側のコード内での参照。

今、 $p$  におけるアクティビティ  $a$  のある出現の中で更新される占有リソース  $r \in R(a)$  を考える。このとき、 $p$  における  $a$  の各出現の中で発生する  $r$  の更新全体からなる集合を  $\text{Upd}(p, a, r)$  とおく。また各  $r$  の更新  $u \in \text{Upd}(p, a, r)$  につ

いて、 $u$  に影響を及ぼすアプリケーション側からフレームワーク側への入力列が存在するとき、その入力列を生成するアプリケーション側のコード片を  $I_{A \rightarrow F}(p, u)$  とおく。また簡単のため、任意のアクティビティ  $a$  および占有リソース  $r \in R(a)$  に対して  $I_{A \rightarrow F}$  を以下のように再定義する。

$$I_{A \rightarrow F}(a, r) = \bigcup_p \left\{ \bigcup_{u \in \text{Upd}(p, a, r)} I_{A \rightarrow F}(p, u) \right\}.$$

これは直観的に、 $a$  による  $r$  の更新に影響を与え得るアプリケーション側からフレームワーク側への入力コード全体からなる集合を表している。ここで、 $M_A$  のあるイベント応答  $\langle e, g, a \rangle$  および  $r \in R(a)$  について、

$$I_{A \rightarrow F}(a, r) \cap H(e, g) = \emptyset$$

が成り立つとき、 $a$  はフレームワーク内部で閉じているという。これは、 $a$  がサンプルアプリケーションのコードに依ることなくフレームワーク内部で実行されていることを示す。また、 $M_A$  の異なる 2 つのイベント応答  $\langle e_1, g_1, a_1 \rangle$ ,  $\langle e_2, g_2, a_2 \rangle$  について、 $r_2 \in R(a_2)$  が存在し、

$$I_{A \rightarrow F}(a_2, r_2) \cap H(e_1, g_1) \neq \emptyset$$

が成り立つとき、 $\langle e_2, g_2, a_2 \rangle$  は  $\langle e_1, g_1, a_1 \rangle$  に  $r_2$  に関して依存しているという。

### 3.2 競合

各リソースは同時に異なる 2 つの状態を取り得ないことから、アクティビティ  $a_1$  および  $a_2$  について、 $r \in R(a_1) \cap R(a_2)$  を満たすリソース  $r$  が存在し、 $a_1$  実行後の  $r$  と  $a_2$  実行後の  $r$  で状態が異なる場合があるな

らば,  $a_1$  と  $a_2$  は競合するといひ,  $a_1 \otimes a_2$  で表す. また,  $r \in R(a_1) \cap R(a_2)$  を満たすすべてのリソース  $r$  について,  $a_1$  実行後と  $a_2$  実行後で常に  $r$  の状態が一致するとき,  $a_1$  と  $a_2$  は整合するといひ,  $a_1 \simeq a_2$  で表す. ここで, 実行シナリオの集合  $S$  を表すイベントモデル  $M_S$  において相異なる2つのイベント応答  $\langle e_1, g_1, a_1 \rangle, \langle e_2, g_2, a_2 \rangle$  を考える. もし,  $e_1 = e_2$  または  $e_1$  と  $e_2$  が同時に発生し, その際に  $g_1$  と  $g_2$  が同時に満たされる可能性があり,  $a_1 \otimes a_2$  となる場合があるとき,  $\langle e_1, g_1, a_1 \rangle$  と  $\langle e_2, g_2, a_2 \rangle$  は競合するといひ, これをシナリオ間競合と呼ぶ. シナリオ間競合はアプリケーションの実装方法や使用するフレームワークに依存することなく議論することができる. したがって, シナリオ間競合は本研究の対象外とする.

次に,  $S$  を表すイベントモデル  $M_S$  とサンプルアプリケーション  $A$  の振る舞いを表すイベントモデル  $M_A$  を考え,  $M_S$  のイベント応答  $\langle e_S, g_S, a_S \rangle$  と  $M_A$  のイベント応答  $\langle e_A, g_A, a_A \rangle$  を考える. このとき,  $e_S = e_A$  または  $e_S$  と  $e_A$  が同時に発生し, その際に  $g_S$  と  $g_A$  が同時に満たされる可能性があつて,  $a_S \otimes a_A$  となる場合があるとき,  $\langle e_S, g_S, a_S \rangle$  と  $\langle e_A, g_A, a_A \rangle$  は競合するといひ, これをシナリオ-サンプルアプリ間競合と呼ぶ. 提案手法では基本的に, サンプルアプリケーションと実装したいシナリオの間のシナリオ-サンプルアプリ間競合を解消していくことによってサンプルアプリケーションがシナリオを満たすように段階的に改変していくアプローチをとる.

最後に, フレームワークのリソース占有によつてもたらされる競合を定義する.  $S$  を表すイベントモデル  $M_S$  の相異なる2つのイベント応答  $\langle e_{S_1}, g_{S_1}, a_{S_1} \rangle, \langle e_{S_2}, g_{S_2}, a_{S_2} \rangle$  と,  $A$  の振る舞いを表すイベントモデル  $M_A$  の相異なる2つのイベント応答  $\langle e_{A_1}, g_{A_1}, a_{A_1} \rangle, \langle e_{A_2}, g_{A_2}, a_{A_2} \rangle$  を考える. ただし,  $e_{S_1} = e_{A_1}$ ,  $e_{S_2} = e_{A_2}$ ,  $g_{S_1} = g_{A_1}$ ,  $g_{S_2} = g_{A_2}$  が成り立つとする. このとき,  $a_{S_1} \otimes a_{A_1}$  かつ  $a_{S_2} \simeq a_{A_2}$  が成り立ち,  $a_{A_2}$  がフレームワーク内部で閉じており, さらに  $\langle e_{A_2}, g_{A_2}, a_{A_2} \rangle$  が  $\langle e_{A_1}, g_{A_1}, a_{A_1} \rangle$  にリソース  $r$  に関して依存しているならば,  $\langle e_{S_1}, g_{S_1}, a_{S_1} \rangle$  と  $\langle e_{S_2}, g_{S_2}, a_{S_2} \rangle$  は,  $r$  について競合するといひ, これをシナリオ-シナリオ-リソース間競合と呼ぶ. これが競合している理由は以下の通りである. まずこのサンプルアプリケーションの与えられた実装においては,  $a_{S_1} \otimes a_{A_1}$  が成り立っているので, この時点でシナリオ-サンプルアプリ間競合が発生している. そこで, この競合を解消するようにサンプルアプリケーション側のコード  $H(e_{A_1}, g_{A_1})$  の内部を修正すると, このコードに依存したアクティビティ  $a_{A_2}$  の挙動に変化が生じる. その結果, 今度は  $a_{S_2} \otimes a_{A_2}$  が成り立つことになり, 再びシナリオ-サンプルアプリ間競合が発生する. しかも,  $a_{A_2}$  はフレームワーク内部で閉じているため, アプリケーション側のコードの修正によつてシナリオと整合させるのが一般に困難となる. そのため, これも競合と考えることができる.

## 4. 競合解消戦略に基づくサンプルアプリ改変手法

### 4.1 手法の全体の流れ

本節では提案手法の流れについて説明する. 本手法はフレームワークのサンプルアプリケーションを改変していくことで与えられた実行シナリオの実装を支援する手法である. 本手法で利用するサンプルアプリケーションは実装したい実行シナリオと可能な限り近い振る舞いをするものを選択する必要がある. サンプルアプリケーションの改変は実装予定の実行シナリオと競合しているサンプルアプリケーションの振る舞いを特定し, その振る舞いを実装しているサンプルアプリケーション中のコードを改変することで競合を解消していく競合解消戦略に基づいて進められる.

本手法の競合解消戦略の流れを以下に示す (図 2 参照).

**ステージ 1** 与えられた実行シナリオを, サンプルアプリケーションの振る舞いと整合するシナリオ (以下, 整合シナリオ) と競合 (シナリオ-サンプルアプリ間競合) するシナリオ (以下, 競合シナリオ) に分離する.

**ステージ 2** どの整合シナリオも依存していない競合シナリオについて, その競合を解消するようにサンプルアプリケーションのコードを改変する. これはシナリオ-サンプルアプリ間競合の解消に相当する.

**ステージ 3** 一つでも依存している整合シナリオがある競合シナリオについて以下の作業を行う. そのような競合シナリオが存在しなければ終了する. 競合シナリオと競合しているサンプルアプリケーションのコードを改変し, 競合を解消する. さらにその改変によつて整合シナリオが整合しなくなる可能性があるため, その場合はサンプルアプリケーションを改変し, 新たな競合を解消する. これはシナリオ-シナリオ-リソース間競合の解消に相当する. ステージ 3 はすべての競合が解消されるまで繰り返す.

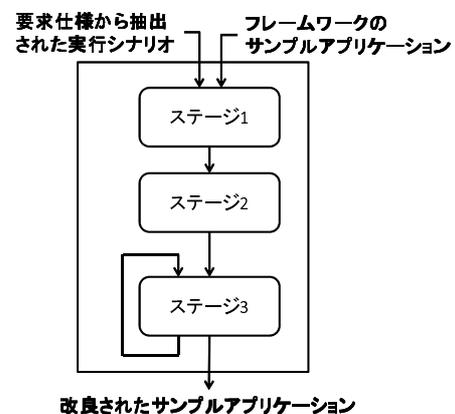


図 2 手法全体の流れ

## 4.2 手法の詳細

本手法では競合シナリオと競合しているしているサンプルアプリケーションのコードを次のように特定してその改変を促すことで競合の解消を支援する。

まず、サンプルアプリケーションと競合している競合シナリオ中のイベント応答  $\langle e_S, g_S, a_S \rangle$  に着目する。ここで  $\langle e_S, g_S, a_S \rangle$  と競合しているサンプルアプリケーション側のイベント応答を  $\langle e_A, g_A, a_A \rangle$  とおく。このとき  $e_S$  がサンプルアプリケーションによって検出できるか否かを文献 [2] の方法で判定する。検出できない場合は競合の解消は困難となる。検出できる場合  $H(e_A, g_A) \neq \emptyset$  となるのでこのコードの内部を改変して、実行シナリオとの間の競合の解消を図る。

競合の解消の方法は2通りある。まず競合しているアクティビティ  $a_A$  がフレームワーク内で閉じていない場合  $a_A$  の処理の主要部分が  $H(e_A, g_A)$  内に記述されているためこの部分の改変を促す。具体的には  $H(e_A, g_A)$  内部のフレームワーク側からの入力とフレームワーク側への出力を受け持っているコード特定し、さらに出力コードの中で  $R(a_A)$  に影響を与えているものを特定する。ここまでの一連の作業はサンプルアプリケーションを実行しながら基本的にユーザが手動で行う。次に  $R(a_A)$  に影響を与えている出力にさらに影響を与えている入力コードを特定し、それらの入力と出力のコード間のデータの流れを独立したメソッドとして抽出する。この部分はツールによる支援が可能であると考えられる。抽出されたメソッド内部の処理を書き換えることによって競合の解消を図る。一方  $a_A$  がフレームワーク内で閉じている場合  $a_A$  の処理を打ち消す処理を  $H(e_A, g_A)$  内部に記述する。具体的には  $a_A$  によって  $R(a_A)$  に行われた更新を  $H(e_A, g_A)$  内で上書きして打ち消す処理を加える。そのような処理をアプリケーション側に記述できない場合もあるが、そのときは競合の解消は不可能となる。

以上を踏まえてステージ1の手順を構成すると図3、ステージ2および3の手順を構成すると図4のようになる。図3における無関係とはサンプルアプリケーションと実行シナリオの間で等価なイベントおよびガードを共有していない場合をいう。図4の手順はステージ1で分離した各競合シナリオに対して行われる。ステージ2ではシナリオ-サンプルアプリ間競合の解消が図られるが解消の方法は競合しているサンプルアプリケーションのアクティビティがフレームワーク内で閉じているか否かによって変わる。いずれの場合でも、サンプルアプリケーション内の競合部分に競合解消のためのコメントが記載されたスケルトンコードが出力される。このとき、実際の競合解消のための実装はユーザに委ねられる。ステージ3では競合シナリオ  $S$  と  $S$  に依存している整合シナリオ  $S'$  の間のシナリオ-シナリオリソース間競合の解消が図られるがいずれのシナリオにつ

いても影響範囲を局所化するために、アクティビティの打消しによって競合の解消が図られる。本手法を適用した場合の実行シナリオの実装方針は図3、図4中の(a)~(e)のいずれかになる。

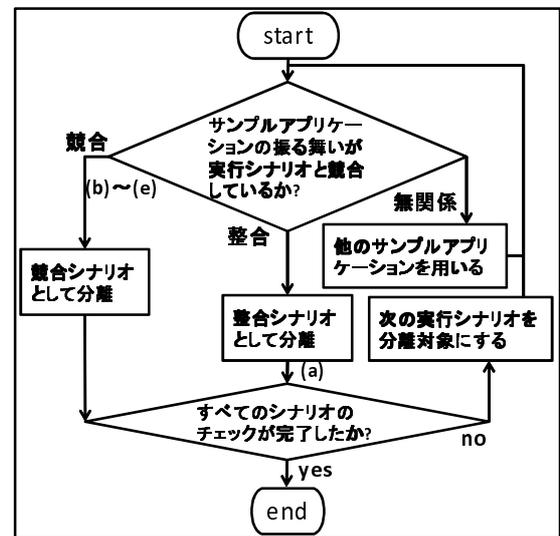


図3 ステージ1の流れ

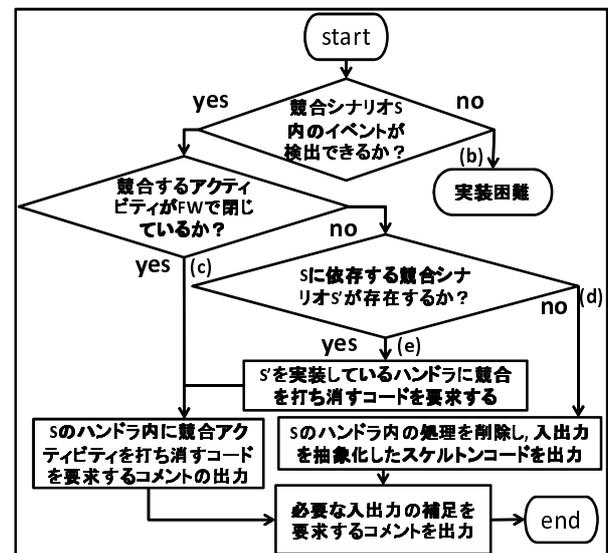


図4 ステージ2およびステージ3の流れ

## 5. 事例研究

2つのゲームフレームワークを対象に図1で示した3D格闘ゲームの実行シナリオが本手法の支援によって実装できるか否かを検証した。サンプルアプリケーションは各フレームワークで提供されているものから適当なものを選択した。

## 5.1 Radish

Radish<sup>[7]</sup>は我々の研究室でJava3Dを用いて開発された3Dゲームフレームワークである。実装言語はJavaで、規模は約1.4万行である。サンプルアプリケーションとしてRadishFightを用いる。RadishFightは本研究室でRadishを再利用して開発された3D格闘ゲームである。

RadishFightの振る舞いを確認すると、既にシナリオ5以外のすべての実行シナリオに整合する振る舞いが見られた。RadishFightではブロックが地面に衝突した時に地面の上で停止する振る舞いをする。この振る舞いはシナリオ5のアクティビティa6と競合するので、RadishFightの振る舞いと実行シナリオ5は競合していると判定する。よって以下ではシナリオ5の競合の解消を行う。

RadishFightにおいてシナリオ5と競合するアクティビティを実装しているハンドラを手動で調べたところ、フレームワーク側のクラス内のOvergroundActor.onIntersect()メソッドであることが分かった。このメソッドはアプリケーション側でオーバーライド可能であるため、イベントは検出可能と判定する。競合するアクティビティの処理は検出したハンドラ内で実装されており、衝突面の情報、キャラクターの位置、速度を入力してキャラクターの位置を衝突面の法線方向に押し戻す処理と、速度の面の法線方向の成分を0にする処理が実装されている。これによってキャラクターが着地するようになっている。以上の処理でフレームワークの占有リソースであるキャラクターの位置と速度が更新されている。このハンドラはフレームワーク側のクラス内で定義されているが、オーバーライドすることにより打ち消すことができるので、アクティビティa6に競合するアクティビティはフレームワーク内部で閉じていないと判定する。オーバーライドしたハンドラ内のコードを空にして実装したところ、他の整合している4つのシナリオのアクティビティが影響を受けている様子も特に見られなかったため、依存する整合シナリオは存在しないと判定する。

以上の判定から本手法におけるRadishFightを用いたシナリオ5の実装方針は第4節で説明した(d)になるため、アクティビティ実装のための入出力を抽象化したスケルトンコードを出力する。このスケルトンコードの内容を手動で改変することによりシナリオ5の実装を完了することができた。シナリオ5の競合を解消する前のソースコードを図5(a)、シナリオ5の競合を解消した後のソースコードを図5(b)に示す。

## 5.2 jMonkeyEngine

jMonkeyEngine<sup>[8]</sup>(以下、jMEと略す。)はオープンソースのゲームエンジンである。実装言語はJavaで、規模は約31.5万行である。全体のアーキテクチャはフレームワークの形式を採用している。サンプルアプリケーションとして

チュートリアル中のLesson8を用いる。Lesson8は物理演算シミュレーションを行うサンプルアプリケーションで、キーボード入力により画面上に表示されているブロックをリアルタイムで操作できる仕様になっている。Lesson8を選択した理由は、jMEに付録されているサンプルアプリケーションの中で今回開発を想定する3Dゲームの挙動に最も近い挙動をするためである。

Lesson8の振る舞いを確認すると、既にシナリオ2, 3, 5に整合する振る舞いが見られた。Lesson8では、シナリオ1においてボタンが押されたときキャラクターの静止状態からの水平面上の移動は問題なく行われるが、ボタンを離すとフレームワーク内部の物理演算処理によりキャラクターは即座に停止できない。また、例えば右移動ボタンを離した瞬間左移動ボタンを押すとキャラクターは即座に切り返すことができない。これらの振る舞いはキャラクターの速度に関してシナリオ1のアクティビティa1, a2に競合しているため、Lesson8の振る舞いと実行シナリオ1は競合していると判定する。さらにLesson8では、キャラクターが自由落下中に地面に到着すると、地面の法線方向に跳ね返る振る舞いをする。この振る舞いはキャラクターの速度に関してシナリオ4のアクティビティa5に競合するので、Lesson8の振る舞いと実行シナリオ4は競合していると判定する。よって以下ではシナリオ1, 4の競合解消および実装を行う。

jMEはキー入力に対してあらかじめ登録してあるハンドラを駆動するイベント駆動の仕組みが採用されている。Lesson8においてシナリオ1に競合するアクティビティを実装しているハンドラを手動で調べたところ、サンプルアプリケーション中でボタン押下イベントで駆動するハンドラを見つけ出せたため、シナリオ1のイベントe1, e2は検出可能と判定する。サンプルアプリケーションのイベントe1, e2のハンドラのコードを空にして実装すると、右(左)ボタンを押してもキャラクターは動かなくなり、それによって登り坂の上でも斜面の方向に移動することができなくなる。よってサンプルアプリケーションのアクティビティa1, a2に競合するアクティビティはフレームワーク内部で閉じていないと判定し、既に整合しているシナリオ2が競合シナリオ1に依存していると判定する。

以上の判定から本手法におけるLesson8を用いたシナリオ1の実装方針は第4節で説明した(e)になり、シナリオ2が依存していることが分かるためシナリオ競合の解消のためシナリオ2のイベントe3のハンドラ内に競合を打消すコードを要求するコメントを出力し、シナリオ1のイベントe1, e2のハンドラ内に競合を打消すコードを要求するコメントを出力する。整合シナリオ2に競合を打消すコードを要求するコメントを出力する理由は、シナリオ1の競合を解消した場合にシナリオ-シナリオリソース間競合により、依存しているシナリオ2が競合シナリオに変わる可能

性があるからである。コメントが出力されたハンドラに競合するアクティビティを打ち消す処理を手動で実装することでシナリオ 1 の実装を完了することができる。シナリオ 1 のイベント e1, e2 のハンドラのソースコードを図 6 に示す。なお整合シナリオ 2 に関して競合を打ち消すコードを要求するコメントの出力は後ほど図 7 のコードで示す。

Lesson8 においてシナリオ 4 に競合するアクティビティを実装しているハンドラを手動で調べたところ、サンプルアプリケーション中でキャラクターが地面に到着するときには駆動するハンドラを見つけ出せたので、シナリオ 4 のイベント e5 は検出可能と判定する。地面の法線方向に弾性移動する処理はアプリケーション側では実装しておらず、サンプルアプリケーションのイベント e5 のハンドラ内のコードを空にして実装しても、本来の Lesson8 の挙動と全く変わらない振る舞いをする。このことから実行シナリオのアクティビティ a5 と競合するアクティビティはフレームワークに閉じていると判定する。

以上の判定から本手法における Lesson8 を用いたシナリオ 4 の実装方針は第 4 節で説明した (c) になり、シナリオ 4 のイベント e5 のハンドラ内に競合を打ち消すコードを要求するコメントを出力する。コメントが出力されたハンドラに a5 と競合するアクティビティを打ち消す処理を手動実装することでシナリオ 4 の実装を完了することができる。シナリオ 4 のイベント e5 のハンドラのソースコードを図 7 に示す。

表 1 事例研究によって得られた実装方針

	S1	S2	S3	S4	S5
Radish	(a)	(a)	(a)	(a)	(d)
jMonkeyEngine	(e)	(a)	(a)	(c)	(a)

## 6. 考察と今後の課題

事例研究の結果をまとめたものを表 1 に示す。この結果から今回想定した 3D 格闘ゲームの実行シナリオの実装において、本手法によって得られる実装方針が有効であると考えられる。特に jME の例で、我々の先行研究 [2] で実装が困難と判定された実行シナリオが本手法の方法に従うと実装できることが判明した。ただし、実際に実装可能な例で、先行研究では実装可能と判定されるにもかかわらず、本手法では実装不能と判定されるものがあることにも注意が必要である。今回の事例研究では Radish では RadishFight, jME では Lesson8 を利用することでアプリケーションの開発に成功したが、与えられた実行シナリオと振る舞いが十分に近くないサンプルアプリケーションを利用した場合、すべての実行シナリオの実装は成功しない可能性があると考えられる。よって本手法において、改変するサンプルアプリケーションの選択が重要になると考えられる。jME を

```

public void onIntersect
    (CollisionResult cr, long interval) {
    // めり込んだ面の法線方向に押し戻す
    Vector3d back = (Vector3d) cr.normal.clone();
    back.scale(cr.length * 2.0);
    body.apply( new Position3D(
        body.getPosition3D().add(back)), false);

    // 速度の面の法線方向の成分を 0 にする
    Vector3d v = (Vector3d) ((Solid3D)body)
        .getVelocity().getVector3D().clone();
    double d = v.dot(cr.normal);
    v.scaleAdd(-d, cr.normal, v);
    body.apply(new Velocity3D(v), false);
}
(a) アクティビティ a6 の競合を解消する前のソースコード

public void onIntersect
    (CollisionResult cr, long interval) {
    /* 自動生成されたコメント (シナリオ 5 実装) */
    // 入出力を抽象化したスケルトンコード呼び出し
    body.apply(someMethod1(body, cr), false);

    /* 自動生成されたコメント (シナリオ 5 実装) */
    // 入出力を抽象化したスケルトンコード呼び出し
    body.apply( someMethod2(
        (Solid3D)body, cr , interval), false);

    /* 自動生成されたコメント (シナリオ 5 実装) */
    // 目的のアクティビティを実装する上で
    // 不足しているリソースの更新がある場合は、
    // そのリソースを更新するコードを追加してください。
}
/**
 * 自動生成されたスケルトンコード (シナリオ 5)
 * 目的のアクティビティを実装する上で引数が
 * 不足している場合は追加してください。
 */
private Position3D someMethod1
    (Object3D arg0, CollisionResult arg1) {
    Vector3d back = (Vector3d) arg1.normal.clone();
    back.scale(arg1.length * 2.0);
    return new Position3D(arg0.getPosition3D().add(back));
}
/**
 * 自動生成されたスケルトンコード (シナリオ 5)
 * 目的のアクティビティを実装する上で引数が
 * 不足している場合は追加してください。
 */
private Velocity3D someMethod2(Solid3D arg0
    , CollisionResult arg1 , long interval) {
    Force3D f = PhysicsUtility
        .calcForce(interval, arg1.normal, arg0);
    arg0.move(interval, f, body.getPosition3D());
    return arg0.getVelocity();
}
(b) アクティビティ a6 の競合を解消した後のソースコード
(下線はユーザが手動で実装した部分)
    
```

図 5 RadishFight においてシナリオ 5 と競合するソースコード (OvergroundActor クラス内)

```

public void performAction( InputActionEvent evt ) {
    /* 自動生成されたコメント (シナリオ 1 実装) */
    // 整合シナリオ 2 が競合シナリオ 1 に依存しています。
    // 競合しているアクティビティを
    // 打ち消す処理を実装してください。

    if ( evt.getTriggerPressed() ) {
        player.getMaterial()
            .setSurfaceMotion( direction );

        Vector3f vel = normal.cross(
            new Vector3f(0.0f, 0.0f, 1.0f));
        float scale = direction.length();
        if(direction.getX() > 0) {
            scale = -scale;
        }
        vel.scaleAdd(scale, ZERO);
        player.setLinearVelocity(vel);
    } else {
        player.getMaterial().setSurfaceMotion( ZERO );
        player.setLinearVelocity(ZERO);
    }

    // 目的のアクティビティを実装する上で
    // 不足しているリソースの更新がある場合は、
    // そのリソースを更新するコードを追加してください。
}
    
```

図 6 Lesson8 におけるシナリオ 1 との競合を解消したソースコード (下線はユーザが手動で実装した部分)

```

public void performAction( InputActionEvent evt ) {
    /* 自動生成されたコメント (シナリオ 1 実装) */
    // 整合シナリオ 2 が競合シナリオ 1 に依存しています。
    // 整合シナリオ 2 が競合した場合は
    // その競合を打ち消す処理を実装してください。

    /* 自動生成されたコメント (シナリオ 4 実装) */
    // 競合しているシナリオ 4 のアクティビティが
    // フレームワーク内で閉じています。
    // 競合しているアクティビティを
    // 打ち消す処理を実装してください。
    ContactInfo contactInfo
        = (ContactInfo) evt.getTriggerData();
    if ( contactInfo.getNode1() == floor
        || contactInfo.getNode2() == floor ) {
        playerOnFloor = true;
    }
    normal = contactInfo.getContactNormal(null);
    Vector3f velocity
        = player.getLinearVelocity(null).clone();
    float d = velocity.dot(normal);
    velocity.scaleAdd(-d, normal, velocity);
    player.setLinearVelocity(velocity);
    /* 自動生成されたコメント (シナリオ 4 実装) */
    // 目的のアクティビティを実装する上で
    // 不足しているリソースの更新がある場合は、
    // そのリソースを更新するコードを追加してください。
}
    
```

図 7 Lesson8 におけるシナリオ 4 との競合を解消したソースコード (下線はユーザが手動で実装した部分)

再利用した場合の実行シナリオ 1 の実装において、整合シナリオである実行シナリオ 2 が競合シナリオである実行シナリオ 1 に依存しており、シナリオ-シナリオ-リソース間競合が発生した。本研究では競合を解消できたが、フレームワークの内部構造によってはシナリオ 1, 2 を同時に整合させることができなくなる可能性も存在すると考えられる。

今後の課題としては本手法の一部を自動化したツールの実装が挙げられる。ツールの支援により、アプリケーション開発において本手法をより効果的に使うことができると期待される。また本手法において、シナリオ-シナリオ-リソース間競合が発生した場合のより包括的な競合の解消方法についての議論なども課題として挙げられる。

## 7. おわりに

サンプルアプリケーションを利用して実行シナリオの実装を支援する手法の提案を行った。また、手法の有効性を検証するために 2 つのゲームフレームワークを対象に 3D 格闘ゲームの実行シナリオが実装できるか否かの検証を行った。その結果、本手法によって得られた実装方針にしたがって、実装を完成させることができることがわかった。今後、より多くのフレームワークに対して本手法を適用していくと同時に、本手法が実用規模のアプリケーションの開発にどの程度有効であるかについての検証を行っていく予定である。また、現時点で人手でおこなっている作業の一部を自動化しツールによって支援できるように手法の改善を行っていく予定である。

## 参考文献

- [1] Mohamed E. Fayad, Ralph E. Johnson and Douglas C. Schmidt. Building application frameworks: object-oriented foundations of framework design, John Wiley & Sons (1999).
- [2] 手塚 裕輔, 新田 直也: リアルタイムアプリケーションの開発におけるフレームワークの選択支援手法, 情報処理学会研究報告, 2010-SE-170(3) (2010).
- [3] Wolfgang Pree, Design Patterns for Object-Oriented Software Development, Addison Wesley (1995).
- [4] Sheikh I. Ahamed, Alex Pezewski and Al Pezewski. Towards framework selection criteria and suitability for an application framework, In *Proceedings of the IEEE Int. Conf. on Information Technology: Coding and Computing (ITCC)*, pp. 424-428 (2004).
- [5] Garry Froehlich, H. James Hoover, Ling Liu and Paul G. Sorenson. Choosing an object-oriented domain framework, *ACM Comput. Surv.*, Vol. 32, No. 1 (2000).
- [6] Douglas Kirk, Mare Roper and Murray Wood. Identifying and addressing problems in object-oriented framework reuse, *Empirical Software Engineering*, Vol. 12, No. 3, pp. 243-274 (2007).
- [7] 新田 直也, 久野 剛司, 久米 出, 武村 泰宏: 3D ゲームエンジン Radish の開発とそのアーキテクチャ比較への応用, 日本デジタルゲーム学会, デジタルゲーム学研究, Vol. 4, No. 1, pp.1-12 (2010).
- [8] Mark Powell ら: jMonkeyEngine, 販売者なし (2003).