Vol.2010-SE-170 No.3
2010/11/11

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

手 塚 裕 輔[†1]　新 田 直 也[†1]

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. しかしながら現実のアプリケーション開発においては, プロジェクトに適合しないアプリケーションフレームワークを選択して, 実装段階になるまでその不適合性に気づかず, 最終的にプロジェクト全体に大きな損失を与えてしまうといった場合も少なくない. そこで本研究では, 最小限のドキュメントとソースコードを元にアプリケーションフレームワークが開発対象のリアルタイムアプリケーションの要求仕様に適合しているか否かを評価する効率の良い手法を提案する. 本稿では, 提案手法の適用事例として 2 つの 3D ゲームフレームワークを対象に適合性の評価を行った. その結果, ある種のゲームアプリケーションを実装する上での問題点を実際に実装を行うことなく発見することができた.

A Support Method for Framework Selection in Developments of Real-time Applications

Yusuke Tezuka[†1] and Naoya Nitta[†1]

Recently, application frameworks are widely used to improve the reusability of designs and imprements of application software. However in a real-world application development, it sometimes happens that the developpers select an application framework which is unsuitable for their project, they cannot notice the unsuitableness until the implementation process has started and finally the unsuitableness causes a serious loss to the project. Therefore in this research, we present an efficient method to assess whether a given application framework meets the requirements specification of a real-time application to be developed from the minimal documents and the source code of the framework. In this paper, we have evaluated two real-world 3D game frameworks based on our method for a case study, and have found several problems on the implementation of a certain game application without implementing it.

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[1)](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制御機構を固定化してしまうため, フレームワークの選択が不適切であると, アプリケーションとして実装できる振る舞いが制限されたり, 場合によっては実装できない機能が生じることもある. そのため, 開発工程のできるだけ早い段階で, 候補となっているフレームワークの適合性を判定できることが望ましいが, 実際には, 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ことなどが原因となって, 実装段階になって初めてフレームワークの不適合性に気づくということも少なくない.

そこで, 本研究では最小限のドキュメントとソースコードを元に, フレームワークが開発対象のアプリケーションの要求仕様に適合しているか否かを評価することで, フレームワークの選択を支援する効率の良い手法の構築を目指す. 具体的には, アプリケーション A とフレームワーク F に対して, A の動作シナリオと F のソースコードを元に, F のソースコードを再利用して A を実装した場合に起こりうる問題点を, 実装前に列挙する系統的な手法を提案する.

同種の手法はすでにいくつか提案されているが, それらと比較して本手法は以下のような特長を持つ.

- 有限個の状態間の遷移で特徴づけることが困難なリアルタイムシステムを対象とする.
- 本手法の入力となる動作シナリオの抽象度が, 現実のソフトウェア開発工程の中で作成される要求仕様の抽象度と近いため, 動作シナリオの構成によって発生する余剰工数は, 比較的小さいと予想される.

†1 甲南大学大学院 自然科学研究科
Graduate School of Natural Science, Konan University

- 現実のソフトウェア開発では, フレームワークの実装上の細かな制限が原因となって, アプリケーション側の実装作業に支障をきたすことも少なくない. 本手法は, そのような制限を実際にアプリケーションの実装を行うことなく見つけることができるよう工夫がなされている.
- 上記項目の反面, 現実のソフトウェア開発では, 多少不適合なフレームワークを選択した場合でも, 実装上の工夫で個々の問題への対処がなされることもある. 本手法では, そのような実装の自由度も考慮に入れつつ問題発生の有無が検討される. これらの理由により, 本手法によって, より現実に即した評価結果を効率よく得ることができると期待される.
- 与えられた動作シナリオが実装可能と判定された場合に, その具体的な実装の指針が得られる.

本稿では, 本手法の適用事例として, リアルタイム 3D ゲームの開発を想定して, 本研究室で開発されたゲームフレームワークである Radish を再利用する場合と, オープンソースのゲームエンジンである jMonkeyEngine を再利用する場合について本手法を用いて比較を行った. その結果, jMonkeyEngine を再利用する場合でいくつかの問題点が発見された. また, 発見された問題点を個別に検討した結果, それらが実装上, 現実に問題となりうることが確認された. 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, 実際にアプリケーションの実装を試みて本手法によって発見された問題点の深刻度を定量的に評価していく予定である.

2. リアルタイムアプリケーション向けフレームワーク

2.1 フレームワーク選択における問題点

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[2)] 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フレームワーク側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には,

(1) 適切に再利用できるようになるための習熟には時間を要する,
(2) アプリケーション内部の主要な制御機構を固定化してしまうため, 選択したフレームワークが開発対象のアプリケーションに適合しなかった場合, アプリケーションとして実装できる振る舞いが制限されたり, 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もある. 本研究ではこれらのうち後者の問題に着目する.

多くの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, ドキュメントに記載されていないフレームワーク側の実装の詳細が, アプリケーション側に複雑な実装上の制約をもたらしている場合も多く, そのため実装段階にならなければフレームワークの不適合性に気づかないといったことも少なくない.

そこで本研究では, 最小限のドキュメントとソースコードを元に, フレームワークが開発対象のアプリケーションの要求仕様に適合しているか否かを評価することで, フレームワークの選択を支援すると同時に, フレームワークを選択することによりプロジェクトに混入する潜在的な問題の見落としを減少させる効率の良い手法の構築を目指す. なお本手法では, 評価にあたってフレームワークのソースコードを一時的に改変することを許容する. ただし, アプリケーションの保守性を維持するため, 実際に再利用する際には, アプリケーションの内部に改変したフレームワークのソースコードを含めることを認めないものとする. これは, 実際のアプリケーション開発においても一般的な仮定である. なお, フレームワークの適合性にはパフォーマンスなどの非機能的要件に関わるものもあるが, 本研究では評価の対象としない.

2.2 リアルタイムアプリケーションに適したイベントモデル

本研究では, 特にリアルタイムアプリケーションに注目し, その開発におけるフレームワークの選択支援手法について考える. ただし, 本研究で提案する手法の適用範囲がリアルタイムアプリケーションに制限されるわけではないことに注意しておく. なお, 本研究では並列および分散アプリケーションを対象としないものとする.

リアルタイムアプリケーションの最大の特徴は, アプリケーションへの入力およびその処理がアプリケーションの実行中にリアルタイムで行われることである. 本研究ではこのような入力をイベントとして捉え, 開発対象となっているアプリケーションの動作シナリオをイ

ベントモデルを用いて記述することを考える．代表的なイベントモデルとしては状態遷移モデルに基づくものが挙げられるが, 多くのリアルタイムアプリケーションの動作は, 有限の内部状態間の遷移で特徴付けることが困難であると考えられるため, 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, 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．これによって, 手法全体の中で自動化できる部分は少なくなるが, より現実に即した評価を導く手法を構築できることが期待される．

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 $\tau : E \times G \to 2^A$ 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 の中で暗に記述される．本イベントモデルを用いて, 3D ゲームアプリケーションの動作シナリオを記述した例を**図 1** に示す．

3. フレームワークの選択支援手法

3.1 手法の概要

本節ではアプリケーションの動作シナリオとフレームワークのソースコードから, フレームワークの動作シナリオに関する適合性を評価する手法を提案する．フレームワークの選択は開発工程全体に大きな影響を及ぼすため, 本手法はアプリケーションの実装を開始する前に適用されることを想定して設計している．ただし, 実装の途中の段階でも適用は可能である．開発工程に関しては, それ以外の点についても特には前提を置かない．

本手法を適用するための条件は以下の通りである．

条件 1　フレームワーク側のソースコードに自由にアクセスでき, 読み書きが可能である．
条件 2　評価を進めるのに必要最小限のドキュメントまたはサンプルコードが存在する．
条件 3　ソースコードに変更を加えたフレームワークは最終成果物に含めることができない．
条件 4　アプリケーションのソースコードは任意に記述できる．

条件 1, 2 は本手法を利用するための前提条件である．条件 3 は本手法の前提条件ではないが 2 節で述べた理由によりアプリケーション開発時の制約として想定する．これらの条件はいずれも現実的に妥当なものであると考えられる．

本手法の手順を以下に示す (**図 2** 参照)．

ステップ 1　開発対象であるアプリケーションの想定される動作シナリオを書く．
ステップ 2　動作シナリオからイベント, ガード, アクティビティを抽出する．
ステップ 3　フレームワークのドキュメントやサンプルコードからフレームワークのイベント制御機構を調べる．
ステップ 4　フレームワークを用いたときのイベントの検出可能性とアクティビティの実装方法を調べ, 各動作シナリオについて評価を行う．
ステップ 5　全ての動作シナリオの評価を総合してフレームワークの適合性の評価を行う．

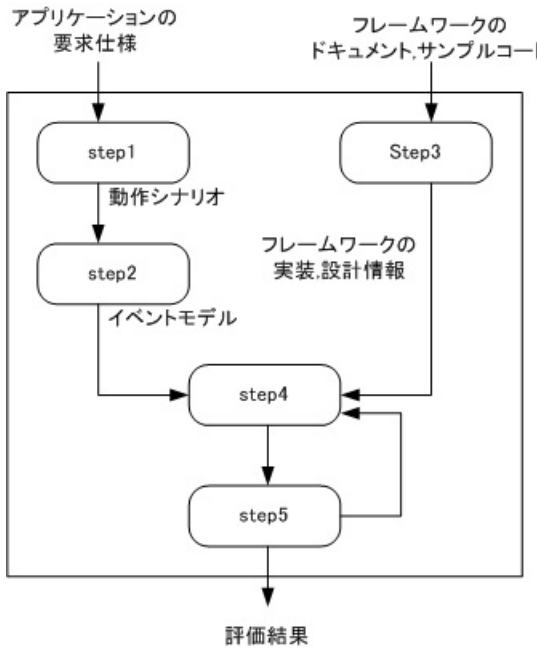

図 2　提案手法の概要
Fig. 2　The outline of our method

3.2 ステップ 4 の詳細

ある動作シナリオ S がイベントモデル $M = (E, G, A, \tau)$ で表現されるとき, S を実装するにはイベント $e \in E$, ガード $g \in G$ の任意の組に対して, アプリケーション実行中で g が

[動作シナリオ 1(S1)] $\underbrace{\text{キャラクターが水平な地面の上にいるとき}}_{\text{ガード (g1)}}$, $\underbrace{\text{右移動 (左移動) ボタンを押すと}}_{\text{イベント (e1)}}$ $\underbrace{\text{キャラクターが地面の上を右に移動する}}_{\text{アクティビティ (a1)}}$. $\underbrace{\text{ボタンを離すと}}_{\text{イベント (e2)}}$, $\underbrace{\text{静止する}}_{\text{アクティビティ (a2)}}$.

[動作シナリオ 2(S2)] $\underbrace{\text{キャラクターが水平な面を右 (左) に移動中}}_{\text{ガード (g2)}}$ $\underbrace{\text{登り坂にさしかかると}}_{\text{イベント (e3)}}$, $\underbrace{\text{斜面の方向に移動を続ける}}_{\text{アクティビティ (a3)}}$.

図 1 動作シナリオの例
Fig. 1 Examples of scenarios

満たされているときに e が発生した場合に, $a \in \tau(e, g)$ を満たすすべてのアクティビティ a が起動されるようにすればよい. フレームワーク F を用いてシナリオ中のある e, g, a の組に対してこのような実装を行おうとした場合, 以下のような選択肢が考えられる.

(1) "g が満たされているときに e が発生した場合に a を実行する" というソースコードが, F の中に既に書かれている場合.

(2) "g が満たされているときに e が発生した場合に a に矛盾するアクティビティを実行する" というソースコードが, F の中に既に書かれている場合.

(3) g が満たされているときに e が発生した場合に実行するソースコードが, F の中に何も書かれていない場合.

上記 (2), (3) の場合は a をアプリケーション側で実装する必要がある. また (2) の場合はさらに a に矛盾するアクティビティを打ち消す処理を a とは別に実装する必要がある. いずれの場合でもその実装部分を e の発生時に呼び出すためには, F からアプリケーション側に e の発生を通知し制御の反転を起こす必要がある.

フレームワーク側からアプリケーション側へ e の発生が通知され, それがアプリケーション側で検出されるようにするためにはいくつかの方法がある. ここでアプリケーションの実行を σ , σ を一定の基準で有限個の実行区間に区切ったものを $T(\sigma)$, そのうち e が内部で発生しているものを $T_e(\sigma)(\subseteq T(\sigma))$, 制御の反転 n が内部で発生しているものを $T_n(\sigma)(\subseteq T(\sigma))$ とする. e の発生の検出の可否について, 以下の場合に e の発生を n で検出可能であるという.

- e が発生した時, かつその時に限り n が起こる ($\forall\sigma.T_e(\sigma) = T_n(\sigma)$)
- e が発生した時に n が起こり, このときアプリケーション側で "実際に e が発生しているか否か" を判別できる. ($\forall\sigma.T_e(\sigma) \subset T_n(\sigma)$)

以下の場合に e の発生を n で検出不可能であるという.

- e が発生した時に n が起こり, このときアプリケーション側で "実際に e が発生しているか否か" を判別できない. ($\forall\sigma.T_e(\sigma) \subset T_n(\sigma)$)
- e が発生しても n が起こらない場合がある ($\exists\sigma.T_e(\sigma) \cap \overline{T_n(\sigma)} \neq \phi$)

以上の議論より, S の実装可能性を評価するため本手法では以下の 2 点について調べる.

(1) イベントの発生をアプリケーション側で検出できるかどうか (イベントの検出可能性).

(2) アクティビティを実装する必要があるか, 必要である場合は実装可能かどうか (アクティビティの実装方法).

これらの評価を総合して S に対する評価を**表 1** のように行う.

表 1 動作シナリオに対する評価基準
Table 1 Evaluation criteria for scenarios

		イベント	
		検出可能	検出不可能
アクティビティ	整合	(a)	(d)
	矛盾	(b)	(e)
	処理無し	(c)	(f)

表 1 に示した (a) から (f) の各項目について, 動作シナリオを実装する上で予想される問題点は以下の通りである.

(a) 特に問題は見当たらない. ただし, アクティビティの仕様変更が発生した場合, 評価が

(b) に変わる可能性が高い.

(b) アプリケーション側でアクティビティに矛盾した処理を打ち消すソースコードが実装でき, かつアクティビティが実装可能ならば問題はない. 実装可能性の判定では, フレームワーク側からアプリケーション側およびアプリケーション側からフレームワーク側に, それぞれ必要な情報の受け渡しができるかどうかに注意する.

(c) アクティビティが実装可能ならば問題はない. 実装可能性の判定では, フレームワーク側からアプリケーション側およびアプリケーション側からフレームワーク側に, それぞれ必要な情報の受け渡しができるかどうかに注意する.

(d) 特に問題は見当たらない. ただし, アクティビティの仕様変更が発生した場合, 評価が (e) に変わる可能性が高い.

(e) イベントが検出不可能であるため実装は困難である.

(f) イベントが検出不可能であるため実装は困難である.

4. 適 用 事 例

リアルタイム 3D ゲームの開発を想定して, 本手法を 2 つのフレームワークに適用した. リアルタイムアプリケーションの例として 3D ゲームを選択した理由は, 一般に制御機構が複雑で実装可能性の判定が難しいこと, およびキャラクターの位置や姿勢など有限個の状態で特徴付けることが明らかに困難であることである. 評価には 2 つの動作シナリオを用いた (図 1 参照). これらは開発対象となっているゲームの動作シナリオの一部であるが, リアルタイム 3D ゲームにおいては典型的なものである.

4.1 Radish への適用

Radish[3] 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 万行である.

4.1.1 Radish の動作シナリオ 1 に対する評価

Radish のメインループは約 1/60 秒毎に起動し制御の反転により IGameState.update() を経由してアプリケーション側を呼び出す. この際, 各キーボードが押されているか否かの情報がフレームワーク側から引数を通じて渡されるので, 1/60 秒以上の間隔で発生するキー入力イベント e1 および e2 は検出可能である (**表 2** 参照). このときフレームワーク側でアクティビティ a1 および a2 に矛盾する処理は実行されない. またキャラクターが地面の上にいるか否かという情報はアプリケーション側で取得可能であり, アプリケーション側でキャラクターの移動速度を設定することも可能である. よって Radish の動作シナリオ 1 に対する評価は表 1 の (c) となる.

4.1.2 Radish の動作シナリオ 2 に対する評価

Radish では動作シナリオ 2 に対して複数の実装方法が存在する. 本検証にはキャラクターの制御に関して推奨されている方式 (方式 A とする) を用いた場合を想定して調査を行う. 方式 A を用いた場合, Actor.motion() に対応するテンプレートメソッドによってキャラクターと物体の衝突時に制御の反転が発生する. この時, 衝突面の法線ベクトルの値からイベント e3 が判別可能である (表 2 参照). 方式 A を用いた場合, アクティビティ a3 の処理はフレームワーク側で全て実装されている. よって Radish の動作シナリオ 2 に対する評価は (a) となる.

4.2 jMonkeyEngine への適用

jMonkeyEngine[4] (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形式を採用している.

4.2.1 jMonkeyEngine の動作シナリオ 1 に対する評価

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 (**表 3** 参照). 駆動するハンドラの登録をアプリケーション側で行うことは可能だが, 一部のキーにはフレームワーク側でハンドラが登録されている. そのためハンドラが登録されたキーに右移動ボタンを割当てるとアクティビティに矛盾する処理が実行される. jME ではキャラクターの移動の実装方法が 2 通りあることが調査の結果判明した (それぞれ方式 B, C とする). まず方式 B では静止状態からの水平面上の移動は問題なく行われる. しかし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. これらの処理はアクティビティ a1 および a2 に矛盾する.

方式 C では方式 B のような問題は発生しない.

よって jME の動作シナリオ 1 に対する評価は方式 B では (b), 方式 C では (c), となる.

4.2.2 jMonkeyEngine の動作シナリオ 2 に対する評価

jME では Radish と同様にキャラクターと物体の衝突時に制御の反転が発生し, 衝突面の法線ベクトルの値からイベント e3 が判別可能である (表 3 参照). 方式 B ではアクティビティに整合する処理がフレームワーク内部で実行される. しかし方式 C では坂にさしかかった所でキャラクターが転倒しアクティビティ a3 に矛盾する. これはフレームワーク内部の

物理演算処理によるものである.

よって jME の動作シナリオ 2 に対する評価は方式 B では (a), 方式 C では (b), となる.

表 2 イベントの検出 (Radish)
Table 2 Event detection (Radish)

イベント	ホットスポット	フレームワーク側の呼び出し	イベント検出の可否
e1,e2	IGameState.update()	AbstractGame.start()	$T_e(\sigma) \subset T_n(\sigma)$(検出可能)
e3	Actor.onIntersect()	Actor.motion()	$T_e(\sigma) = T_n(\sigma)$(検出可能)

表 3 イベントの検出 (jMonkeyEngine)
Table 3 Event detection (jMonkeyEngine)

イベント	ホットスポット	フレームワーク側の呼び出し	イベント検出の可否
e1,e2	InputAction.performAction()	BaseGame.start()	$T_e(\sigma) = T_n(\sigma)$(検出可能)
e3	InputAction.performAction()	BaseGame.start()	$T_e(\sigma) \subset T_n(\sigma)$(検出可能)

表 4 適用事例の結果
Table 4 Results of the case studies

	S1	S2	問題の有無
Radish	(c)	(a)	問題なし
jMonkeyEngine(方式 B)	(b)	(a)	問題あり
jMonkeyEngine(方式 C)	(c)	(b)	問題あり

5. 考　　察

適用事例の結果をまとめたものを**表 4** に示す. 評価結果は, 今回想定した 3D ゲームの開発において jME を再利用した場合に実装上の問題が発生しうることを示している. 実際に, これらの問題点を同時に解消するような実装方法がないかについて, 具体的に実装を行いながら検討を重ねたが, 現在のところ適当な実装方法は見つかっていない. いっぽう, Radish を再利用する場合については本事例では特に問題点は見つからなかったが, これは実際に Radish のアーキテクチャが, 今回適用事例で用いた 3D ゲームの動作シナリオに近いシナリオを想定して設計されたこと, Radish 開発後にこれらの動作シナリオの実装が実際になされたことによく整合する.

アプリケーションの開発において, フレームワークの選択に起因する問題が実装工程で発覚した場合, 大きな手戻りが発生することが予想されるが, 事前に本手法を用いて, フレームワークを再利用した場合の問題点の列挙を行うことで, そのようなリスクを回避できる可能性が高くなると考えられる.

本手法の有効性を議論する上で重要な点として, 本手法の適用により開発工程中に新たに発生する余剰工数の大きさが挙げられる. この余剰工数について本手法の各ステップごとに考察する. まずステップ 1 の動作シナリオの作成については, 現実の開発工程の中で作成される要求仕様と本手法が要求する動作シナリオの抽象度が近いため, 多くの工数を必要としないと考えられる. この点については他のシナリオに基づく評価手法においても同様である. さらにステップ 2 のイベント, ガード, アクティビティ等の抽出によって発生する工数も小さいと考えられる. ステップ 3 および 4 が, 本手法の中で最も多くの工数を要するステップである. しかしながら, ここで行う調査と同様の調査は, 現実の開発工程の中でも, 実際にアプリケーションを実装する場面で行われるものと考えられる. しかもここでの調査結果は, フレームワークを再利用することが決定した場合, アプリケーションの実装作業に対する具体的な指針として直接利用できるものであり, 作業として無駄になることはない. したがって, ステップ 3 および 4 によっても余剰工数はほとんど発生しないと考えられる. 最後のステップ 5 は, それまでの結果より明らかであり, さほどの工数は必要としない. 以上より, 本手法を実際に適用する際に発生する余剰工数は開発全体の工数に対して小さいことが予想される.

6. 関 連 研 究

フレームワークの選択は, アプリケーションの開発初期段階においてアーキテクチャや全体設計の一部を固定化する. そのため, 選択したフレームワークの評価は, 通常の開発におけるアーキテクチャの事前評価と共通する部分が多い. アーキテクチャを開発初期段階で評価できる代表的な手法として, SAAMER[5] が挙げられる. SAAMER は, 幅広い分野における開発工程を考慮に入れた包括的かつ実用的な評価手法であるが, 本研究が対象としているような制御の反転を持つアーキテクチャについては十分な考慮がなされていない. また, 本研究で扱っているような実装工程で発生する抽象度の低い, しかしながら深刻な問題点については触れられていない.

アプリケーションの要求仕様とフレームワークに関する情報からフレームワークの評価を試みる手法としては, 文献 6) の手法がある. この手法は, アプリケーションの要求仕様とフ

レームワークの動作仕様を共にラベル付き遷移システムで表現して, 観測等価性の観点からフレームワークの適合性について評価を行うものである. 本研究とは, アプリケーションの要求仕様およびフレームワークの動作仕様が共に有限の状態間の遷移で表現できることを前提としている点で大きく異なる. 本研究では, イベントモデルの多くの部分を自然言語で記述することを許容しているため, 記述の厳密性が低下しツールなどによる自動化支援が困難になる反面, アプリケーションの要求仕様を有限の状態間の遷移で特徴付ける必要がなくなり, 本研究が対象としているようなリアルタイムシステムにも柔軟に適用することが可能になる. また, ラベル付き遷移システムを構成する作業も不必要になるため, 余剰工数の発生や, 構成作業に伴う誤りの混入を抑制することができる. さらに本手法は, 文献 6) の手法のような観測等価性に基づく評価と比較して, より現実の開発に即した評価を提供する. たとえば, フレームワークとアプリケーションの間で必要な情報の受け渡しを行う仕組みが提供されておらず, アプリケーションの実装ができなくなるような場合でも, 観測等価性に基づく手法では問題として検出することができない. 逆に, アクティビティを打ち消したりイベント検出の方法を工夫するなどして実装レベルで対処できるような問題でも, 観測等価性に基づく手法では問題点として過検出してしまう. 本手法では, これらのような検出漏れおよび過検出にも対応している.

3D ゲーム分野においては, ゲームエンジンアーキテクチャの評価手法の確立の必要性が文献 7) で指摘されている. 同文献では, 主にアーキテクチャのゲームジャンルへの依存性, および CG などの低レベル技術への依存性の観点からの議論が展開されている. 本研究は, 任意の動作シナリオに対してフレームワークの評価ができるため, 特に前者の問題に対して有効な手段を提供できることが期待される.

7. おわりに

アプリケーションの動作シナリオに基づいて, リアルタイムシステム向けのフレームワークをアプリケーションの実装前に評価する手法を提案した. 本手法は, 系が有限個の状態で表現できることを前提としていない, 実装レベルの自由度を考慮に入れて評価を行うことができるなどの特徴を持つ. いくつかの 3D ゲームフレームワークを対象に本手法の適用を行った結果, 本手法によって, 実装上重要な問題点を比較的少ない工数で発見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, 実際にアプリケーションの実装を試みて発見された問題点の深刻度を定量的に評価していきたい.

参 考 文 献

1) Mohamed E. Fayad, Ralph E. Johnson, Douglas C. Schmidt: Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons Inc (1999).
2) Wolfgang Pree, Design Patterns for Object-Oriented Software Development. Addison Wesley (1995).
3) 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).
4) Mark Powell ら, jMonkeyEngine, 販売者無し (2003).
5) Chung-Horng Lung, Sonia Bot, Kalai Kalaichelvan and Rick Kazman: An Approach to Software Architecture Analysis for Evolution and Reusability, Proc. of CASCON (Centre for Advanced Studies Conf.), pp. 144–154 (1997).
6) Teruyoshi Zenmyo, Takashi Kobayashi and Motoshi Saeki: Supporting Application Framework Selection Based on Labeled Transition Systems, IEICE Trans. Inf. & Syst., Vol. E89–D, No. 4 (2006).
7) Eike Falk Anderson, Steffen Engel, Leigh McLoughlin and Peter Comninos: The Case for Research in Game Engine Architecture, ACM FuturePlay 2008, pp. 228–231 (2008).