Vol.2013-SE-182 No.16
2013/10/24

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ツールの開発

縄江 保宏[1,a)] 新田 直也[1,b)]

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際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している. 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し, 改変方針を提示することによって支援を行う. 本稿では, 手法の一部を自動化する動的解析ツールを開発し, その有効性について評価を行った.

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 [1](以下, 本稿ではフレームワークと呼ぶ) 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, 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改変方針を提示することによって実装の支援を行う.

本稿では本手法の一部を自動化する動的解析ツールの実装を行った. ただし, 現時点では実行効率の確保のため, 時間を要する厳密な解析を避け近似解のみを算出するようアルゴリズムの設計を行っている. そのため, 真の解にどの程度近い解を算出するかについて確認する評価実験を行った. その結果, 先行研究において行った評価事例中の 4 つのケースのすべてにおいて真の解と同一の解を出力することがわかった. 今後は, 実行効率を著しく損なわない範囲で, より厳密な解を算出できるようアルゴリズムの改善を図っていくと同時に, 手法の中の自動化する部分を増やしていく予定である.

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 [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実行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**と呼ぶ. 現

[1] 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

© 2013 Information Processing Society of Japan

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], [3], [4], [5] 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改変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. 実行シナリオと競合

本研究で提案している実装支援手法は, 実装したい実行シナリオと既存のサンプルアプリケーションの動作の間の競合の解消に基づいて構築されている. 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. **実行シナリオ**は開発対象のアプリケーションに求められる動的振る舞いを記述したもので, **イベント**, **ガード**, **アクティビティ**の 3 種類の部分に分けることができる. これらの間には, ガードが満たされている状態でイベントが発生すると, ただちにアクティビティが実行されるという関係が成り立っている.

一般に既存のサンプルアプリケーションの動的振る舞いは, 実行シナリオに記述されている動作とは一致しない. 本研究では実行シナリオとサンプルアプリケーション間の以下の 2 種類の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の動作を実行シナリオに近づけていくことを考える.

- 単独の実行シナリオが既存のサンプルアプリケーションの動的振る舞いと競合をしている場合 (**単純競合**).
- 単純競合している実行シナリオと単純競合していない実行シナリオの 2 つが存在し, かつ前者のシナリオと競合しないようサンプルアプリケーションの改変を行った結果, サンプルアプリケーションの動作が後者のシナリオと競合するようになってしまうような場合 (**複合競合**).

単純競合と複合競合の具体的な定義を与える前にいくつかの諸定義を行う. サンプルアプリケーション A の実行において, 状態の変化を外部から観測できるオブジェクトを A の**リソース**という. フレームワーク F の任意のアプリケーションにおいて存在し, アプリケーション側のコードからフレームワークを呼び出すか, あるいはフレームワーク側からの呼び出し (制御の反転) に対してアプリケーション側が戻り値を返さなければ状態を変更できないようなリソースを F の**占有リソース**と呼ぶ. サンプルアプリケーション A の実行において, ガード g が満たされている状態でイベント e が発生したときに, 制御の反転が発生してアプリケーション側の特定のメソッドが呼び出されるとき, そのメソッドを g と e の**イベントハンドラ**と呼び, $H_A(e, g)$ と書く. 実行シナリオ S_1 にイベント e_1 とガード g_1 が, 実行シナリオ S_2 にイベント e_2 とガード g_2 がそれぞれ含まれているとする. また, S_2 のアクティビティが占有リソース r の状態変化を要求しているとする. このとき, サンプルアプリケーション A において S_2 が S_1 に r **を介して依存**しているとは, 以下のような A の実行 σ が存在することをいう (**図 2** 参照).

- σ において, g_1 が満たされている状態で e_1 が発生した後, g_2 が満たされている状態で e_2 が発生する. (このとき, r の状態は変化してもしなくてもよい.)
- σ 中で先に出現した $H_A(e_1, g_1)$ の実行が後の r の状態変化または状態維持に影響を与えている.

ここで, 実行シナリオ S にイベント e, ガード g, アクティビティ a が含まれているとする. 与えられたサンプルアプリケーション A と S が占有リソース r で単純競合するとは, ある A の実行において, g が満たされている状態で e が発生し, その直後に生じる r の状態変化または状態維持が a が要求する r の状態変化または状態維持と矛盾する場合が存在することをいう. サンプルアプリケーション A と単純競合するリソースが存在するシナリオを**競合シナリオ**, 存

[実行シナリオ 1(S1)] キャラクターが水平な地面の上にいるとき(ガード (g1)), 右移動 (左移動) ボタンを押すと(イベント (e1)) キャラクターが地面の上を右 (左) に移動する(アクティビティ (a1)).

ボタンを離すと(イベント (e2)), 静止する(アクティビティ (a2)).

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

[実行シナリオ 3(S3)] キャラクターが左または右に移動した時(ガード (g3)) 足元に地面がなくなると(イベント (e4)), キャラクターが自由落下を開始する(アクティビティ (a4)).

[実行シナリオ 4(S4)] キャラクターが自由落下中に(ガード (g4)) 地面に到着すると(イベント (e5)), キャラクターが地面の上で停止する(アクティビティ (a5)).

[実行シナリオ 5(S5)] ブロックが自由落下中に(ガード (g5)) 地面に衝突すると(イベント (e6)), ブロックが地面の法線方向に弾性移動する(アクティビティ (a6)).

図 1 実行シナリオの例

シナリオ1の実行の流れ
シナリオ2の実行の流れ
イベントの発生
イベントの発生
状態変化
リソース
フレームワーク側
呼び出し
復帰
影響
呼び出し
復帰
アプリケーション側
イベントハンドラ
イベントハンドラ

図 2 依存関係のイメージ

在しないシナリオを**整合シナリオ**と呼ぶ. サンプルアプリケーション A と実行シナリオ S_1 および S_2 が占有リソース r で複合競合するとは, A と S_1 が r で単純競合し A と S_2 が r で単純競合しない場合で, かつ S_2 が r を介して S_1 に依存している場合をいう. このような場合, A と S_1 の間の単純競合を解消しようとして A のコードの修正を行うと, その影響で元々 A と単純競合していなかった S_2 が単純競合するようになってしまう可能性がある.

4. 競合解消戦略に基づくサンプルアプリケーション改変手法

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 3** 参照).

ステージ 1 与えられた実行シナリオを, 整合シナリオと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. これは単純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って整合シナリオが整合しなくなる可能性があるので, 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. これは複合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.

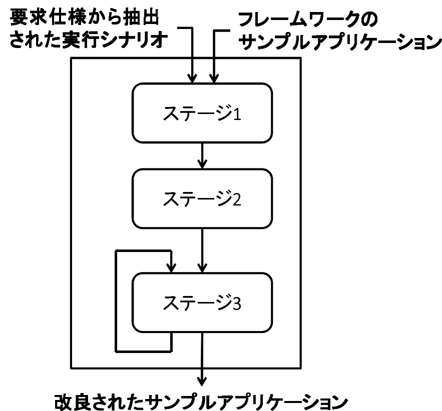

図 3 手法全体の流れ

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 A と競合している競合シナリオ S 中のイベント e とガード g に着目する. このとき e 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 [6] 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. 検出できる場合ハンドラ $H_A(e, g)$ が存在するのでこのコードの内部を改変して, S 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. まず競合している占有リソース r に着目する. A のある実行において, ハンドラ $H_A(e, g)$ の実行が直後の r の状態変化に影響を与えていないとき, または $H_A(e, g)$ が存在しないとき, r の更新は $H_A(e, g)$ に対して**閉じている**という. r が $H_A(e, g)$ に対して閉じていない場合, $H_A(e, g)$ のコードの r の状態変化に影響を与えている部分について改変を促して競合の解消を図る. 一方 r が $H_A(e, g)$ に対して閉じている場合, r の状態変化を打ち消す処理を $H_A(e, g)$ 内部に記述する. 具体的には r に対する更新を $H_A(e, g)$ 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 4**, ステージ2および3の手順を構成すると**図 5** のようになる. 図4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5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では単純競合の解消が図られるが, 解消の方法は競合しているリソースが該当するハンドラに対して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方針を示した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' の間の複合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4, 図5中の (a)〜(e) のいずれかになる.

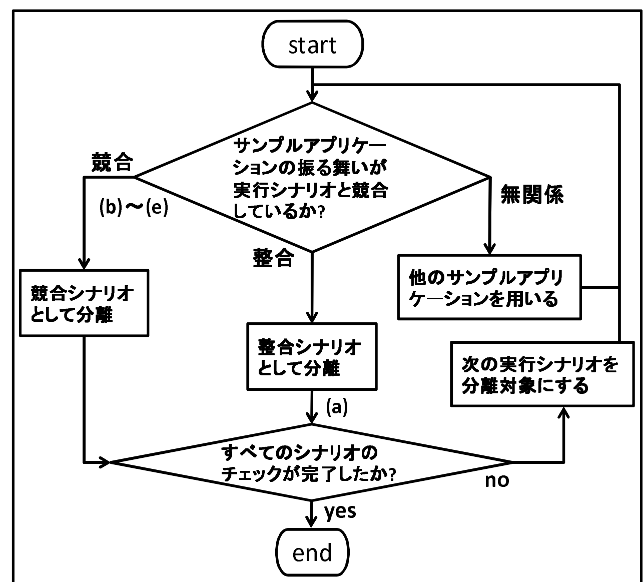

図 4 ステージ1の流れ

5. 実装支援ツールの開発

本研究で提案している実装支援手法の実効性を高めるため, 本手法の一部を自動化したツール (以下, 本ツールと呼ぶ) を Eclipse のプラグインとして開発した. (**図 6** 参照) ツールとして自動化した部分は, 占有リソースの更新がハンドラに対して閉じているかどうかの判定 (以下, 開閉判定と呼ぶ) を行う処理である (図5中の★で示す2箇所).

動的解析を用いて開閉判定を行うためには以下の3つの情報が必要である.

1) 実行シナリオと関連する振る舞いを実行したときのサンプルアプリケーションの実行トレースデータ.
2) 実行シナリオのアクティビティ a によって更新を要求されている占有リソース r.
3) 実行トレース中で e の発生およびガード g の充足に

図 6 本ツールのユーザインタフェース

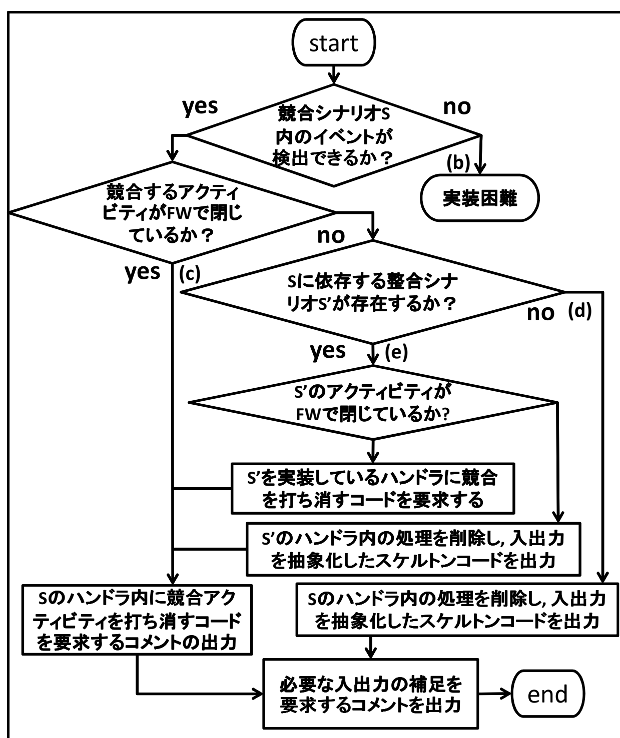

図 5 ステージ 2 およびステージ 3 の流れ

よって引き起こされたイベントハンドラの実行.

ただし, 3) の情報を膨大な実行トレースデータの中で特定することは困難であるため, 代わりに以下の 2 つの情報を与えることにする.

3-a) 実行シナリオのイベント e を検出できるイベントハンドラのメソッドシグニチャ.

3-b) 3-a) だけでは e の発生および g の充足を保証できないことがあるため, 必要に応じてそれを保証するための特有メソッド呼び出し (以下, 特有メソッド呼び出しと呼ぶ).

本ツールの概要を**図 7** に示す.

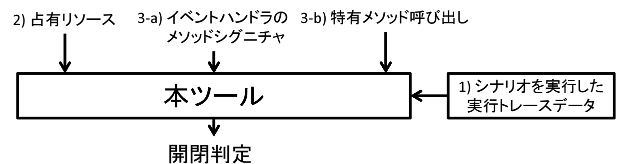

図 7 ツールの概要

実行トレースデータは, 判定対象となる実行シナリオに従って実行したサンプルアプリケーションから, 実行履歴を抽出したテキストファイルであり, 実行プログラムのメソッド呼び出しの履歴が記録されている. 本研究では, この実行トレースデータの抽出には, 本研究室の先行研究 [7] で開発されたプログラム実行履歴抽出ツール (以下抽出ツールと呼ぶ) を用いる. 抽出ツールは AspectJ を用いて実装されており, 解析対象のプログラムに組み込んで実行することで, 占有リソースの更新と呼び出されたメソッドを抽出してテキストとして出力することができる.

本ツールのアルゴリズムは以下のようになる.

(1) 4.2 節で述べた e が検出できない場合には, 入力のイベントハンドラは該当なしとなり, その場合は閉じていると出力して終了する.

(2) 実行トレースデータの末尾から逆方向に r の更新を探索する.

(3) r の更新が見つからなければ, r の更新はイベントハンドラに対して閉じていると出力して終了する.

(4) r の更新が見つかった場合, 更新がイベントハンドラ

の呼び出し先かつ, 特有メソッドの呼び出し先のコードで実行されていたならば, r の更新はイベントハンドラに対して閉じていないと出力して終了する.

(5) r の更新から引き続き逆方向に r の更新を探索して (3) に戻る.

本アルゴリズムでは簡単のため占有リソースの更新を実行しているメソッド呼び出しをたどることで占有リソースの更新処理の制御依存を追跡しているが, データ依存の追跡は行っていない. よって, 本ツールで出力される開閉判定は近似的なものとなる. 具体的には本ツールが閉じていないと出力する場合, 実際に閉じていないことが保障されるが, その逆は必ずしも成り立たない. これによって, 手法全体としては, 実行シナリオが実装可能であるのに実装困難であると判定される可能性が生じることになる. フレームワークの不整合性を検証するという文脈においては, これは安全サイドの近似であるといえる.

6. 実装支援ツールの評価実験

図 1 で示した 3D 格闘ゲームの各実行シナリオのアクティビティが要求している占有リソースの更新の開閉判定を, 本ツールによってどこまで近似できるかを検証するために, 評価実験を行った. 本ツールの適用対象フレームワークとして先行研究 [6] と同様に Radish を選択した. Radish[8] 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.

Radish にはサンプルアプリケーションとして本研究室で開発された 3D 格闘ゲームの RadishFight が提供されており本実験ではそれを用いる. このサンプルアプリケーションは Radish を利用して実装されており, 実装規模は約 1 万行である. 図 1 中の 5 つの実行シナリオの内, RadishFight で実装しているものは S1, S2, S3, S4 である. よって今回の評価実験では RadishFight におけるこれらの実行シナリオのアクティビティで要求される 4 種類のリソースの更新についての開閉判定を本ツールを用いて行った.

評価実験におけるツールへの入力と判定結果および実行トレースデータの容量と解析時間を**表 1** に示す.

7. 考察と今後の課題

表 1 から分かるように RadishFight を対象とした場合, 本ツールの判定結果は全てのシナリオにおいて厳密解と一致した. またツールによる解析時間も高々 3 分程度であった. このことから本ツールが, 本実装支援手法の効率化に有効であると考えられる. しかし, 5 節で説明したように本アルゴリズムではデータ依存を追跡せず制御依存のみを追跡しているため, 他のサンプルアプリケーションを対象とした場合, 正確な判定ができなくなることも予想される.

今後の課題としては, データ依存も追跡できるように本ツールを拡張することと, 今回自動化した本実装支援手法の作業以外にも, 以下の 2 種類の作業 (いずれも図 5 中の作業) の支援が挙げられる.

- 占有リソースや, イベントハンドラの検出.
- シナリオ間の依存関係の有無の判定.

上記の 2 つの作業を未知のフレームワークに対して行った場合, 一般に多くの時間を要することが考えられる. また, これらの作業は今回自動化した作業と同様, サンプルアプリケーションの実行トレースデータを用いて自動化または半自動化することが可能であると考えられる. よってこれらの作業を支援するツールを開発することにより本実装支援手法をより効率化できると期待される. さらに今回は, 本ツールの適用事例として RadishFight を用いた場合を紹介したが, それ以外のサンプルアプリケーションを用いた場合の適用事例も検討して行きたい.

8. おわりに

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について, これまで手作業で行っていたタスクの一部を自動化し, Eclipse のプラグインとして実装した. ただし実装に当たっては, 解析処理の効率化のため近似解を求めるアルゴリズムを採用した. そこで解の精度を求める評価実験を行った結果, 十分な精度が得られていることを確認することができた.

今後, 実行速度を落とすことなく解の精度を向上させるよう, アルゴリズムの改善を図っていきたい. また, 自動化できていない他のタスクについても, 自動化を検討していきたい.

[*1] OS:Windows 7 Professional, CPU:Intel(R) Xeon(R) 3.20GHz, メモリ:16.0GB, JVM build 1.7.0_25

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] Wolfgang Pree, Design Patterns for Object-Oriented Software Development, Addison Wesley (1995).

[3] 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).

[4] 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).

[5] Douglas Kirk, Marc Roper and Murray Wood. Identifying and addressing problems in object-oriented framework reuse, *Empirical Software Engineering*, Vol. 12, No. 3, pp. 243–274 (2007).

[6] N. Nitta, I. Kume and Y. Takemura, “A method

表 1 ツールの入出力結果と厳密解

	S1	S2	S3	S4
占有リソース イベント 特有メソッド呼び出し	キャラクターの速度 ゲームのフレームが進んだ時 あり	キャラクターの速度 地面に接触した時 なし	キャラクターの速度 不検出 なし	キャラクターの速度 地面に接触した時 なし
判定結果	閉じていない	閉じていない	閉じている	閉じていない
厳密解	閉じていない	閉じていない	閉じている	閉じていない
実行トレースデータの容量	1569 $_{Mbyte}$	768.9 $_{Mbyte}$	1954 $_{Mbyte}$	1569 $_{Mbyte}$
解析時間*1	170 $_{sec}$	11.2 $_{sec}$	0.3 $_{msec}$	10.5 $_{sec}$

for early detection of mismatches between framework architecture and execution scenarios, ” Asia-Pacific Software Engineering Conference (APSEC’ 2013), accepted, (2013).

[7] 山根敬史, 新田直也： デルタ抽出を用いたアプリケーションフレームワークの利用例抽出ツールの開発と評価, 情報処理学会研究報告, 2012-SE-178(27) (2012).

[8] 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).

© 2013 Information Processing Society of Japan