プログラムを停止させないデバッグを可能とする手法の提案

丹野 治門　岩崎 英哉

GUI プログラム，ゲームプログラム，ネットワークプログラム，センサ情報処理プログラムのように，双方向性や実時間性が高いプログラムにおいては，ブレークポイントを用いてプログラムを一時停止させるような従来型のデバッグ手法は適していない．その理由の第一は，これらのプログラムにおいては，ユーザ操作やセンサ入力のタイミングや順番が重要であり，デバッグのためにプログラムの実行を途中で中断させると，プログラムが期待通りの動きをしなくなってしまうためである．第二の理由は，プログラムの内部状態を確認するたびにプログラムを停止させていては，デバッグの効率が著しく悪くなるためである．以上の問題点を解決するため，本研究ではプログラムの実行位置の情報と変数情報を実時間に出力して表示するデバッグの手法を提案し，C# プログラムのデバッガとして実装した．本デバッグ手法を用いることにより，開発者はプログラムを停止させずに効率よくデバッグを行えるようになる．

Programs such as GUI programs, game programs, network programs, and sensor information processing programs are highly interactive and real-time. For such programs, traditional debugging methods that suspend a program using breakpoints are not appropriate for the following two reasons. First, the timing and order of user operations or sensor inputs are very important in these programs. If an execution of a program is suspended by a debugger, it might not work as expected. The second reason is that the debugging is awfully inefficient if the program is suspended each time the developer wants to observe the states of internal data structures. To solve these problems, we propose a debugging method for displaying information on executed paths and variables of the program in real time. We implemented the proposed method as a debugger for C #. By using our debugging method, the developer can debug highly interactive and real-time programs efficiently without suspending their executions.

1　はじめに

GUI プログラム，ゲームプログラム，ネットワークプログラム，センサ情報処理プログラムのように，双方向性や実時間性が高いプログラムは以下の特質を持っている．

- ユーザ操作やセンサ入力等のイベントの発生タイミングや順番が重要である．
- プログラムへの入力と，入力を受け取ったときのプログラムの状態の様々な組み合わせにより，プログラムの挙動が複雑に変化する．

これらの特質によりブレークポイントを用いてプログラムを一時停止させるような従来型のデバッグ手法は適していない．その理由の第一は，これらのアプリケーションにおいては，ユーザ操作やセンサ入力のタイミングや順番が重要であり，これを途中で中断させると，プログラムが期待通りの動きをしなくなってしまうためである．第二の理由は，プログラムの内部状態を確認しようとするたびにプログラムを停止させていては，デバッグの効率が悪くなるためである．

例えば，アクションゲームのプログラムにおいて，プレイヤが敵キャラクタへ攻撃を行うロジックをデバッグすることを考える．C# で記述されたデバッグ対象のコードをソースコード 1 に示す．このコードは，プレイヤの入力を元に，プレイヤが攻撃ボタンを押したときに，プレイヤの近くに存在する敵キャラクタの体力を減らし，敵キャラクタの体力が 0 になるとその敵キャラクタを消滅させるという処理を行う．こ

---

* Debugging Method without Suspending Program
This is an unrefereed paper. Copyrights belong to the Authors.

Haruto Tanno, Hideya Iwasaki, 電気通信大学大学院, The University of Electro-Communications.

ソースコード 1　デバッグ対象のコード例

```
1  public void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList)
2  {
3      int damagePoint = player.OffensivePower;
4      if (input.AttackButton)
5      {
6          if (input.DashButton) damagePoint *= 5; //プレイヤがダッシュしていたらダメージ 5倍
7          foreach (var enemy in enemyList)
8          {
9              if (Vector3.Distance(player.Position, enemy.Position) <= 5.0)
10             {//敵がプレイヤの近距離に存在する
11                 if (!enemy.IsInvincible)
12                 {//敵が無敵状態ではない
13                     enemy.HitPoint -= damagePoint;//敵のHP を減らす
14                     if (enemy.HitPoint <= 0) enemy.Dead();//敵のHP が 0 以下なら消滅
15                 }
16             }
17             enemy.EndPlayerCollision();
18         }
19     }
20 }
```

の処理は一定間隔 (例えば 1 秒間に 30 回) で実時間に行われ，ユーザの入力によりその挙動が変わるため双方向性も高い．このコードにおいて，プレイヤがダッシュボタンを押し，5 倍の攻撃力で近距離の敵キャラクタを攻撃すると 2-3 回で倒せるはずなのに，何らかのバグで倒すことができず，そのバグの原因を特定したいとする．バグの原因は「プレイヤはボタンを押したが `input.AttackButton` が真になっていない」，「 `input.DashButton` が真になっておらずダメージが 5 倍になっていない」,「`EnemyList` に攻撃対象の敵キャラクタが含まれていない」,「`Vector3` の `Distance` の計算結果が正しくない」,「`enemy.IsInvincible` が (想定と異なり) 真になっている」など複数考えられる．プレイヤの様々な操作に対し，プログラムがどの位置まで実行されたか，またそのときの各変数の値がどのようになっているかを確認するため，様々な箇所へブレークポイントを設定すると，何らかのボタンを押す，敵キャラクタの近距離に移動する，といった動作のたびにプログラムが中断されてしまいデバッグの効率が悪くなる．また，例えば，`AttackButton` をまず押し，次に `DashButton` を押すといった操作が，操作の途中でプログラムが中断されることにより正常に行えなくなる．これは，例えば GUI プログラムにおける画面上の要素の操作に対して発生するイベントの処理，ネットワークプログラムやセンサ情報処理プログラムにおいて，実時間に送受信されるデータを扱う処理といった双方向性や実時間性の高いプログラムをデバッグする際に共通する問題である．

このような問題を解決するため，本研究ではプログラムを停止させることなくプログラムの実行経路情報と変数情報を，実時間で出力し可視化するデバッグの手法を提案し，C# プログラムのデバッガとして実装した．本デバッグ手法を用いることにより，開発者は実時間性や双方向性の高いソースコード 1 に示したようなプログラムを，効率よくデバッグすることが可能となる．

本論文の貢献は以下である．

- 手続き型言語やオブジェクト指向言語で記述された双方向性，実時間性の高いプログラム一般に対して広く適用できるデバッグ手法を提案している．
- C# プログラムを対象として，提案するデバッグ手法を実現する方法を具体的に示している．この方法は C# 以外の多くの言語に対しても適応できる．

著者は過去にゲームプログラムに特化した，プログラムを停止させずにデバッグを行える手法 [20] を提案しており，本論文の内容はこれを発展させたものであ

る．著者の過去の手法はゲームシステム記述用のドメイン特化言語を対象としている上，実用上足りない機能が多くあった．本研究ではオブジェクト指向言語で記述された双方向性や実時間性の高いプログラム一般に対して広く適用できるデバッグ手法を提案している．過去の手法に対し，提案手法では実用上有用な特徴として主に，(1) ソースコードファイル，メソッド，ステートメントという 3 段階の粒度で実行された部位を実時間に可視化することで確認したい箇所を徐々に絞り込める機能，(2) ループ処理 (for 文，foreach 文) 等によりプログラム中で複数実行された実行経路について，条件を指定することで，開発者が着目したい実行経路や変数の値の情報のみを実時間に可視化できる機能，(3) マルチスレッドを用いたプログラムへの対応，の 3 つをもつ．

また，C# における本実装方式では，デバッグ対象の C# コードを自動変換し，対象コードのステートメントごとにデバッグ情報を取得するためのコードを埋め込むことで実時間にデバッグ用の情報を取得する．提案手法を適用するにあたっては，ユーザはデバッグ対象のソースコードに対して一切変更を加える必要はなく，変換後のコードを動作させるためのデバッグコード実行用のライブラリをあらかじめ対象コードへリンクしておくだけでよい．

以降，本稿では 2 節で，本研究が対象とするスコープを明確にした上で，既存のデバッグ手法における問題点を述べる．次に 3 節で提案するデバッグ手法のコンセプトとその特徴について述べ，4 節で提案手法の C# における本実装方式について説明する．5 節で提案手法のケーススタディを紹介し，6 節でオーバヘッド計測の結果について述べる． 最後に 7 節で今後の課題と本論文のまとめを述べる．

2 双方向性や実時間性が高いプログラムのデバッグ

2.1 本研究の対象領域

本研究では，図 1 のように動作するプログラムにおける，図 1(2) の部分をデバッグ対象とする．この部分では，入力としてユーザ入力，センサ情報，通信データ等を受け取り，それらに対して何らかの処理を行い，出力として計算結果の表示や，データの蓄積，更新う．GUI プログラム，ゲームプログラム，ネットワークプログラム，センサ情報処理プログラムなどでは，このような部分がプログラムの実行時間の大半を占める上，ユーザ操作や，センサ，通信装置からの様々な入力パターンによりプログラムの内部状態が刻々と変化するため，その挙動は予測しにくくデバッグにも労力がかかる．

バグにはいくつかの種類がある．性能に関するバグの原因特定にはプロファイラが活用できる．機能に関するバグには，実行時エラーと論理エラーの 2 種類がある．実行時エラーはプログラムが動作し続けられなくなるエラーであり，NULL 値の予期しない参照等により発生する．一方，論理エラーは意図した仕様と異なる動きをするものであり，計算ロジックに誤りがある場合などに発生する．実時間性や双方向性の高い処理部分で発生する論理エラーは，既存のデバッガのブレークポイントやステップ実行を用いることでは，取り除くことが難しかったり，大きな労力がかかったりする．本研究では，このような論理エラーをデバッグの対象とする．

以上をまとめると，表 1 のようになる．表 1(a) が本研究の対象領域である．表 1(b) のような非実時間，非双方向であるバッチ処理の部分のデバッグは既存のデバッガでステップ実行を用いてプログラム実行時の情報を調べる方法が有効である．また，表 1(c) については，既存のデバッガを用いて，実行時エラーが発生した際に実行位置や変数位置などを取得してバグ原因を調べることが可能である．次節では，表 1(a) へ既存のデバッグ手法を適用したときの問題点について説明する．

2.2 既存手法の問題点

本研究の対象である表 1(a) へ既存のデバッグ手法を適用したときの問題点について述べる．

古典的なデバッグの方法としては，デバッグ情報取得・表示用のコードをデバッグ対象のコードへ埋め込む方法 (以降，「printf デバッグ」と呼ぶ)，ブレークポイントを用いる方法 (Microsoft Visual Studio De-

表 1　本研究の対象領域

		双方向性，実時間性が高い処理 (図 1(2))	バッチ処理 (図 1(1),(3))
機能	論理エラー	(a) 本研究の対象領域	(b) 既存デバッガが有用
	実行時エラー	(c) 既存デバッガが有用	
性能	(d) プロファイラが有用		

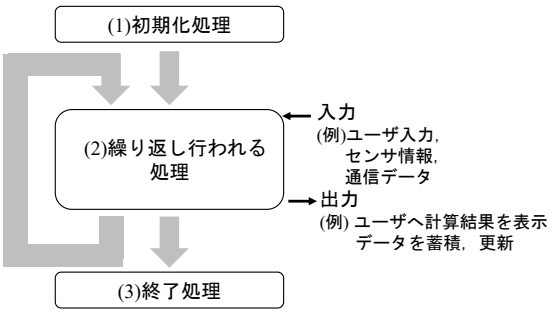


図 1　双方向性や実時間性が高いプログラムの動き

bugger[†1]，GDB[†2] 等)，Assertion をコードへ記述する方法 [14]，プロファイラ (Microsoft Visual Studio Profiler[†3] 等) を用いる方法などがある [16]．printf デバッグや，ブレークポイントを用いる方法は，実際に多くの開発者に使われている [5]．しかしながら，ブレークポイント用いた方法はプログラムを停止させる必要がある．ブレークポイント指定をプログラマブルに行える手法 [18]，自動で行う手法 [19]，行やステートメントではなくオブジェクトの状態に着目して指定できる手法 [13] など，デバッガ利用者の利便性をより向上させた手法もあるが，プログラムを停止させる方法では 1 節で説明したような問題点がある．printf デバッグや Assertion を記述する方法を用いると，プログラムを停止させずにプログラムの内部状態を確認，監視することが可能ではあるが，プログラムの随所にデバッグ用のコードを記述しなければならず，プログラムの保守性や可読性が下がる．アスペクト指向の考え方に基づき，デバッグ用コードを分離して記述する手法 [11]，分離して記述したコードを動的に埋め込むことが可能な手法 [17] も存在するが，いずれにせよデバッグ出力用のコードや Assertion を記述するのは開発者の負担が大きい．プロファイラはソースコードの各メソッドや各行に費やされた CPU 時間を，プログラムを実行し続けながら計測し，その結果をソースコードに重畳表示するツールであり，パフォーマンス改善に主に用いられる．プロファイラを用いると，コードのどの位置が実行されたかはわかるが，どのような入力のタイミングで，どの箇所が実行されたか，またそのときの変数の値は何であったかといった情報は得られない．

より発展したデバッグ手法として，プログラムのログを自動的に取得し，そのログに基づきデバッグを支援するアプローチがある．ETV [15] では，プログラムの実行経路情報を記録し，プログラムを実行させたあとにその情報を視覚的に表示している．JIVE [6] はプログラム実行ログからオブジェクト図，シーケンス図を生成し，更にクエリによって実行ログの中から確認した箇所を絞り込むことも可能である．Herman らの手法 [1] では，表 1(a) の領域を記述する際に用いられるリアクティブプログラム (RP) を対象として，デバッグ対象のプログラムが用いているライブラリへパッチをあてることで可視化のためのログを取得し，そのログからデータフローと処理のタイミングを可視化するデバッガを提案している．また，自動収集したプログラムのログを元に，開発者と対話しながらバグの箇所を絞り込む手法 [12] もある．これらの手法は表 1(a) の領域に対して，プログラムを停止させず，自動でログを集め，後からそのログを確認できるため有用である．しかしながら，これらのアプローチでは，ログの確認はプログラムの実行後になるため，入力に対するプログラムの実行経路や変数の情報を即時に確認し，その情報に基づき次の入力を考えて与える，といったような試行錯誤を効率よく行うこと

†1 https://msdn.microsoft.com/ja-jp/library/windows/desktop/sc65sadd
†2 https://www.gnu.org/software/gdb/
†3 https://docs.microsoft.com/ja-jp/visualstudio/profiling/

ができない．加えて，ログを可視化するアプローチでは，プログラム実行時の全ての情報を記録することが現実的には難しいため，後からログを見ても必要な情報が確認できない場合もある．

プログラムに対するあらゆる入力を記録しておき，プログラムの実行を再現させるアプローチもあり，C 言語，Java，.NET，JavaScript /Node.js など様々な言語，実行環境において実装が行われている [2–4,10]．このアプローチでは，プログラム実行中に I/O を伴うメソッドや乱数生成など，プログラムへの全ての入力を記録したり，プログラムの内部状態のスナップショットを定期的に取得したりしておく．それらの情報を用いてプログラムの実行を再現するので，バグを再現させることができる．このアプローチでは，プログラム実行の再現を繰り返すことによって，特定のバグを何度も再現させることができ，過去に遡って原因を追っていくことも可能である．しかし，ログを可視化するアプローチと同様，入力に対するプログラムの実行経路や変数の情報を即時確認し，その結果を見ながら様々な入力を与え試行錯誤を繰り返し，効率よくバグ原因を探すことができない．また，完全な実行の再現は技術的な障壁が高く，例えばマルチスレッドプログラムでは再現を保証できない [2] などの制約がつき，適用範囲が限られるケースもある．

本研究では，このような既存手法の問題点を解決し，表 1(a) のような双方向性，実時間の高い様々なプログラムのデバッグに際に，入力に対するプログラムの実行経路や変数の情報を即時に可視化し，効率よいデバッグの支援を目指す．

3 提案するデバッグ手法

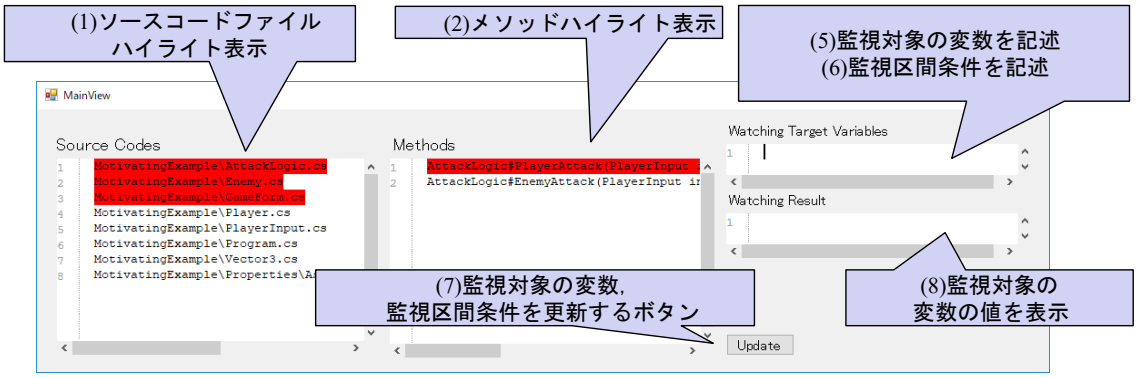
本節では，提案するデバッグ手法について説明する．提案するデバッグ手法に基づいたデバッガを図 2 に示す．デバッガはメインビュー (図 2(a)) とソースコードビュー (図 2(b)) から成る，提案手法ではデバッグ対象のプログラムの実行経路情報と開発者が指定した変数の値を，一定の時間間隔 (例えば 0.1 秒ごと) でこれらのビューに反映し続ける．これにより，開発者は双方向性，実時間の高いプログラムのデバッグを効率よく行うことが可能となる．以降，1 節で紹介したソースコード 1 のデバッグを例にして，提案するデバッグ手法の特徴について図 2 に基づいて説明する．

(1) ソースコードファイルハイライト表示：
デバッグ対象のプログラムのソースコードファイル一覧を表示しており，プログラムが実行したメソッドを含むファイルを実時間にハイライト表示する (ハイライト表示はしばらくすると消える)．これにより，プログラムに何らかの入力を与えたときに，プログラムのどの部分が実行されたかを，開発者は俯瞰的に把握することができる．この例では，プレイヤの攻撃と敵キャラクタの当たり判定の処理として `ActorLogic.cs` ファイルなどのコードが実行されていることがわかる．ここで，開発者がソースコードを選択してクリックすると，そのソースコードで定義されているクラスにおいて，定義されているメソッドの一覧を図 2(2) に表示し，図 2(b) に選択されたソースコードを表示する．

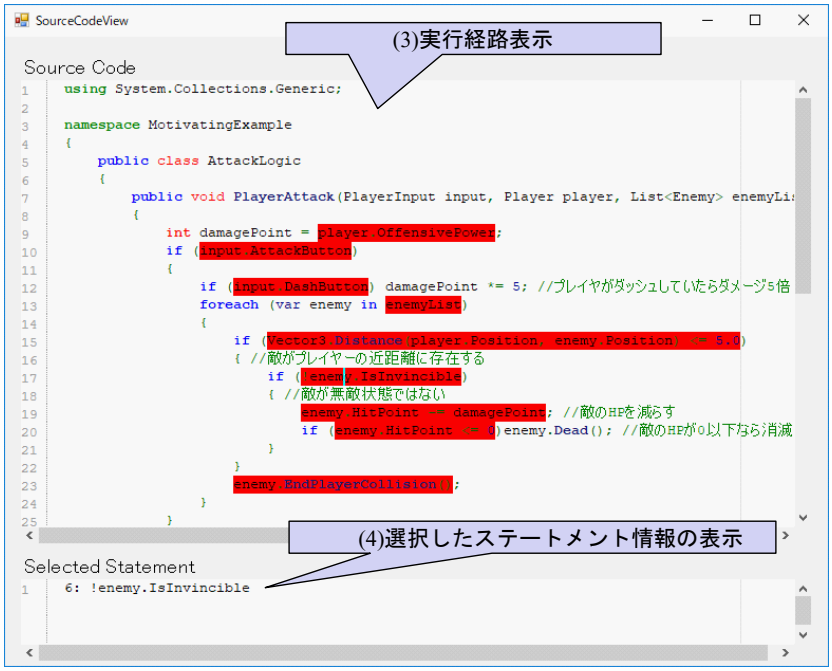
(2) メソッドハイライト表示：
メソッド一覧を表示し，プログラムが実行したメソッドを実時間にハイライト表示する．これにより，図 2(1) と同様に，開発者はどのメソッドが実行されたかを俯瞰的することが可能となる．この例では `ActorLogic` クラスの `PlayerAttack` メソッドが実行されていることがわかる．

(3) 実行経路表示：
ソースコード上でプログラムが一定間隔で実行したステートメントをハイライト表示する．これにより，入力に対してプログラムがどの箇所を実行したかを詳細に把握することができる．この例では，プレイヤが近距離の敵キャラクタに対し (ダッシュボタンは押さず) 攻撃ボタンのみを押して攻撃を行っており，実行経路情報を確認することで，敵キャラクタのいずれかに対し，`enemy.HitPoint -= damagePoint` が実行されていることが即座にわかる．ここで，例えば `DashBurron` をユーザが押すと，図 3 に示すように `damagePoint *= 5` が実行されるようになることも即座に確認できる．

(4) 選択したステートメント情報の表示：
図 2(3) でステートメントをクリックすると，ステー



(a)メインビュー



(b)ソースコードビュー

図 2　提案するデバッグ手法

図 3　ユーザ入力に対する実時間な実行経路の可視化

トメントに対してソースコード中で一意になるよう割り当てられた ID を表示する．この ID とソースコードファイル名を用いて，開発者は関心があるステートメントに対して図 2(5),(6) で後述する監視対象の変数，監視区間とその条件の記述を行うことができる．

(5) 監視対象の変数を記述：
監視したい変数について，表 2 に従って記述することで指定することができる．例えば「`enemy.HitPoint -= damagePoint`」ステートメントの実行前に `enemy.HitPoint` の値を監視したい場合には，図 2(5) の欄へ，監視対象のステートメントが

あるファイルパス「GameForm.cs」を記述し (表 2 の項番 1)，次の行へ watch:8:bf:enemy.HitPoint と記述する (表 2 の項番 2)．watch は変数監視を指示する「指令」であり，8 は図 2(4) で表示されるステートメントの ID である．bf はステートメントが実行する前で値を取得すること意味し (ステートメントを実行した後にしたい場合は af とする)， enemy.HitPoint は，このステートメントを実行するときに評価する式となる．式の値は図 2(8) の表示欄へ，一定間隔ごとの最新の値を表示する．

(6) 監視区間を記述：

開発者が確認したい監視区間とその条件を指定することができる．例えば，foreach(var enemy ...) のループ の内側のステートメントは，複数の Enemy インスタンスそれぞれについて実行される．そのため，Enemy インスタンスがひとつでもステートメントを実行すると，そのステートメントがハイライトされるため，インスタンスごとの区別がつかない．そこで，提案するデバッグ手法では，プログラムの実行経路における特定の区間について，開発者が指定した条件を満たす場合にその区間の実行経路情報をハイライトし，監視対象の変数の値を取得する機能を用意している．例えば，プレイヤの近くにいる敵キャラクタのうち，タイプがスライムである敵キャラクタに関してのみ foreach(var enemy ...) の内側の情報を取得したい場合を考える．そのためには，図 2(6) の欄へ，特定の条件についてのみ監視を行いたい区間の開始ステートメント (例えば，図 2 の 15 行目) へ condstart:5:bf:enemy.TypeName=="Slime" (表 2 の項番 3)，区間の終了ステートメント (例えば，図 2 の 23 行目) へ condend:10:af (表 2 の項番 4) という記述を行う．condstart，condend はそれぞれ区間の開始，終了を表す指令であり，condstart における enemy.TypeName=="Slime" は，この条件が真となるときのみに，区間内のステートメントをハイライトし，監視対象の変数の値を取得することを表す．これにより，開発者はループ処理 (for 文，foreach 文) 等によりプログラム中で複数回実行される経路について，条件を指定することで，ユーザが着目したい経路や変数の値の情報のみを実時間に可視化できる．この条件式ではスレッドの ID なども参照できるため，マルチスレッドプログラムにおいて，特定のスレッドの実行経路情報等を実時間に可視化することも可能である．

(7) 監視対象の変数，監視区間条件を更新するボタン：

図 2(5),(6) への記述は更新ボタンを押すと反映される．

(8) 監視対象の変数の値を表示：

図 2(5) で設定した監視対象の変数の値を表示する．

以上のように，提案するデバッグ手法を用いることで，開発者はプログラムを停止させることなく，確認したい箇所を適宜絞り絞り込み，プログラムへ様々な入力を与えたときに，プログラムの実行経路情報や，変数の値がどのように変化するかを確認しながらデバッグを効率よく行うことが可能となる．

4 C#におけるデバッグ手法の実装

3 節で提案したでデバッグ手法を C# 上に実装した．C# における本実装方式は，デバッグ対象のコードをデバッグ用のコードへ変換するステップ (図 4) と変換したデバッグ用コードを実行し，プログラムを動作させながらデバッグ情報を取得して可視化するステップ (図 5) から成る．本節ではそれぞれのステップについて説明する．

4.1 デバッグ用コードへの変換

図 4 に示すように，本実装方式ではデバッグ対象の C# のコードを変換し (図 4(1))，ステートメントごとにデバッグ情報を取得するためのコードを埋め込む．そして，ユーザは実時間にデバッグ用の情報を取得するライブラリ (図 4(2)) を対象コードにリンクして実行するだけで，3 節に示したようなデバッガを起動することができる．そのため，デバッガを使用するにあたり，ユーザはデバッグ対象のソースコードに対して一切変更を加える必要はないため，負担なく用いることができる．以下，ソースコード 1 を変換した後のコード (ソースコード 2) を例にとり，本実装方式の要点を簡潔に述べる．ソースコード 2 における _Logger クラスは図 4(2) のデバッグ用ライブラリにより提供する．

表 2　指令の一覧

項番	指令種別	記法
1	対象ソースコード	file:ファイルパス
2	監視対象変数	watch:ステートメント ID:ステートメント前後の指定:値取得の式
3	監視区間開始	condstart:ステートメント ID:ステートメント前後の指定:条件式
4	監視区間終了	condend:ステートメント ID:ステートメント前後の指定

ソースコード 2　デバッグ用に変換されたコード

```
1  public void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList)
2  {
3     if (_Logger.IsLogging(0))
4        _debug_PlayerAttack(input, player, enemyList);
5     else
6        _original_PlayerAttack(input, player, enemyList);
7  }
8
9  private void _debug_PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList)
10 {
11    var _l = _Logger.GetLogger(0, 0);
12    int damagePoint = _l.LogFunc(() => player.OffensivePower, 0);
13    if (_l.LogFunc(() => input.AttackButton, 1))
14    {
15       if (_l.LogFunc(() => input.DashButton, 2))
16          _l.LogFunc(() => damagePoint *= 5, 3); //プレイヤがダッシュしていたらダメージ 5倍
17       foreach (var enemy in _l.LogFunc(() => enemyList, 4))
18       {
19       //……中略……
20          _l.LogAction(() => enemy.EndPlayerCollision(), 10);
21       }
22    }
23 }
```

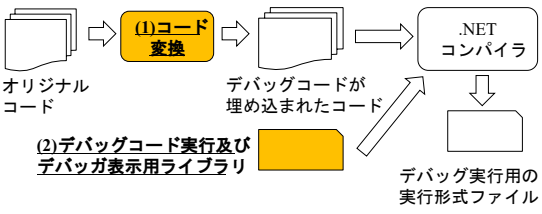


図 4　本実装方式におけるデバッグ用コードへの変換

本実装方式では，大規模なプログラムをデバッグする際に，関心がない部位のデバッグ情報を取得して余計なオーバヘッドが発生することを避けるため，デバッグ情報を取得するか否かをメソッド単位で動的に切り替える機構をもつ．ソースコード 2 の 1 行目の `PlayerAttack` メソッドは，デバッグ対象のコード中のメソッドに一意に割り振られた ID を引数としてとる`_Logger.IsLogging` メソッドの戻り値を確認することで，オリジナルのコードとデバッグ用のコードを，プログラムを停止させずに動的に切り替える．`_original_PlayerAttack` メソッドの定義はソースコード 1 における `PlayerAttack` メソッドの定義と同じである．

ソースコード 2 の 9 行目の`_debug_PlayerAttack` メソッドは，オリジナルの `PlayerAttack` メソッドのコードへデバッグ用コードを埋め込んで変換したコードである．`_Logger.GetLogger` メソッド (ソースコード 2 の 11 行目) は実行中のスレッドに対応するロギング用`_Logger` クラスのインスタンスを返すメソッドである．取得したロギング用インスタンスは`_l` へ格納される．そして，`_l.LogAction`，`_l.LogFunc` という，クロージャにしたステートメントとステートメント ID を引数とするロギング用のメソッドで，全てのステートメントを置き換える．例えば，`player.OffensivePower` という式は `_l.LogFunc(() => player.OffensivePower, 0)` へ

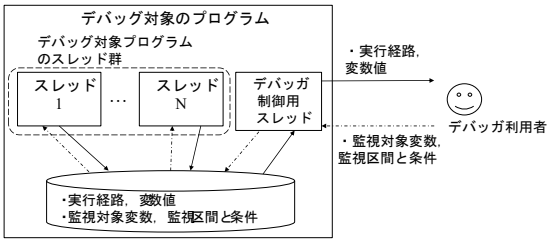


図 5　本実装方式におけるデバッグ実行の仕組み

と置き換える (ソースコード 2 の 13 行目)．LogFunc は戻り値をもつステートメントの置換に使用し，LogAction は戻り値をもたないステートメントの置換に使用する．例えば，LogFunc メソッドではソースコード 3 に示す擬似コードのように，ステートメントの実行前後で，実行経路情報の取得や監視対象のローカル変数の取得，監視区間条件の切り替えなどを行う．Func<T> func にはクロージャとなったステートメントが格納されている．これにより，ステートメントの実行 (ソースコード 3 の 15 行目) の前後で，実行経路情報の取得 (ソースコード 3 の 4 行目) や監視対象のローカル変数の取得 (ソースコード 3 の 10-12 行目)，監視区間の開始点や終了点におけるロギング有効・無効の切り替え (ソースコード 3 の 5-8 行目) をスレッドごとに行う．

4.2　デバッグ実行の仕組み

変換したデバッグ用コードは，図 5 のように実行する．本実装方式では，デバッガ制御用のスレッドはデバッグ対象のプログラムのスレッドの 1 つとして動作する．デバッグ制御用以外の各スレッドは 1 つのステートメントを実行するたびに，デバッグ用コードにより記録する．このようにして集めた実行経路の情報と変数値の情報を取得して可視化する (図 5 の実践矢印)．またデバッガ制御用スレッドは，監視対象変数や監視区間と条件の情報をデバッガ利用者から取得し，その情報を各スレッドにおけるデバッグ用コードが参照することにより，実行経路情報や変数値の記録を行うかどうかを制御する (図 5 の点線矢印)．

実行経路，変数値，監視対象変数や監視区間と条件の情報については，デバッグ対象のプログラムの各スレッドと，デバッガ制御用スレッドが同時にアクセスを行うため，排他制御を行う必要がある．特に実行経路情報についてはこれらのスレッドが頻繁にアクセスを行うため，排他制御を確実に行いつつ，オーバヘッドを軽減する必要がある．そこで，各ステートメントにおける実行回数を記録するための変数に対し，C# の軽量排他制御オペレーションである Interlocked を用いてカウンタのインクリメントやリセットを行い，スレッド間の排他制御を行うよう工夫した．

また，ソースコード 2 の_Logger.GetLogger は，デバッグ対象プログラムのメソッド呼び出しのたびに呼び出され，実行中のスレッドに対応する_Logger クラスのインスタンスが格納されているテーブルへアクセスし，対応するインスタンスを取得する．また，新たに生成されたスレッドが最初にアクセスした場合は，対応するインスタンスを新規生成する必要もある．そのため，このテーブルへのアクセスも排他制御が必要となる．ここでも排他制御のオーバヘッドを軽減するため，スレッドの生成に伴うテーブルへの書き込みは頻繁に行われないと想定し，読み込みについては複数のスレッドで同時に行うことができる排他制御機構である ReadWriterLock を用いる工夫をした．

5　ケーススタディ

本節では， CalculatorCSharp[†4](約 2.5KL) という関数電卓のアプリケーションを題材として，提案手法によるデバッグの例を示す．対象となるバグは，ケーススタディのために人為的に埋め込んだものである．CalculatorCSharp の実行画面を図 6 に示す．今回埋め込んだバグは，「電卓の 1 や 2 など他のボタンを押した場合は正しくその数字が表示されるが，3 ボタンを押したときにのみ画面上部の数字が更新されず，表示が初期状態である 0 のままとなっている」というものであり，期待する動作はボタン 3 を押したときに 3 と表示されることである．

提案手法を用いると図 7 に示すように，プログラムを停止させることなく，効率よくデバッグを行っていくことができる．以下に手順を述べる．

†4　https://github.com/Cleod9/CalculatorCSharp

ソースコード 3　ロギング用メソッドの擬似コード

```
1  public T LogFunc<T>(Func<T> func, int statementId)
2  {
3      //対象ステートメント実行前のデバッグ用処理
4      if(現スレッドでロギングは有効か？) 実行経路の記録
5      if(対象ステートメント実行前の監視区間開始 (condstart)指令があるか？){
6          if(condstart の条件式が真か？) 現スレッドでロギングを有効化
7          else 現スレッドでロギングを無効化
8      }
9      if(対象ステートメント実行前の監視区間終了 (condend)指令があるか？) 現スレッドでロギングを有効化
10     if(現スレッドでロギングは有効か？){
11       if(対象ステートメント実行前のwatch 指令があるか？) 変数値の記録
12     }
13
14     //対象ステートメントの実行
15     T result = func();
16
17     //対象ステートメント実行後のデバッグ用処理
18     //……中略……
19
20     return result;
21 }
```

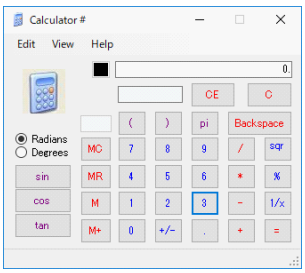


図 6　CalculatorCSharp の実行画面

1. ボタン 3 を押すと，図 7(1) に示すように，CalcEngine.cs と Form1.cs がハイライトされ，これらのソースコードが実行されており，どちらかのソースコードにバグの原因がありそうだということがわかる．今回，表示部分にバグがありそうなため，まず Form1.cs について詳しくみていく．
2. 図 7(2) では，Form1.cs のメソッド一覧を表示し，またボタン 3 を押すことで，どのメソッドが実行されたかをハイライト表示している．ここでは，`updateScreen`，`number_btn`，`formatDisplay` が実行されていることがわかる．それぞれのメソッドの定義を確認してみると，`number_btn` の定義にボタンの種別で分岐する switch 文が含まれているため，`number_btn` についてステートメント単位で実行経路を確認してみる．
3. 図 7(3) で，ボタン 3 を押したときの `number_btn` の実行経路を確認すると，`numValue=3` が実行されていないことがわかる．また，このときボタン 1 などを押すと `numValue=1` などはきちんと実行されていることもすぐにわかる．ボタン 3 を押したときの `num.Name` の値に原因がありそうなので，`num.Name` に変数監視式を設定する．
4. 図 7(4) では，ボタン 3 が押されたときの `num.Name` の値を表示している．値は `thre` となっており，ボタン 3 の `Name` フィールド値を `three` とすべきところ，変数名を誤って設定していたのがバグの原因であるとわかった．

このように，提案手法を用いることで，様々な入力を与えながら，プログラムを停止させることなくプログラムの内部状態を即座に確認することで，効率よくデバッグを行うことができる．

6　オーバヘッドの計測

本節では，本提案手法のオーバヘッドを計測した結果について述べる．以下のように GUI アプリ，ゲー

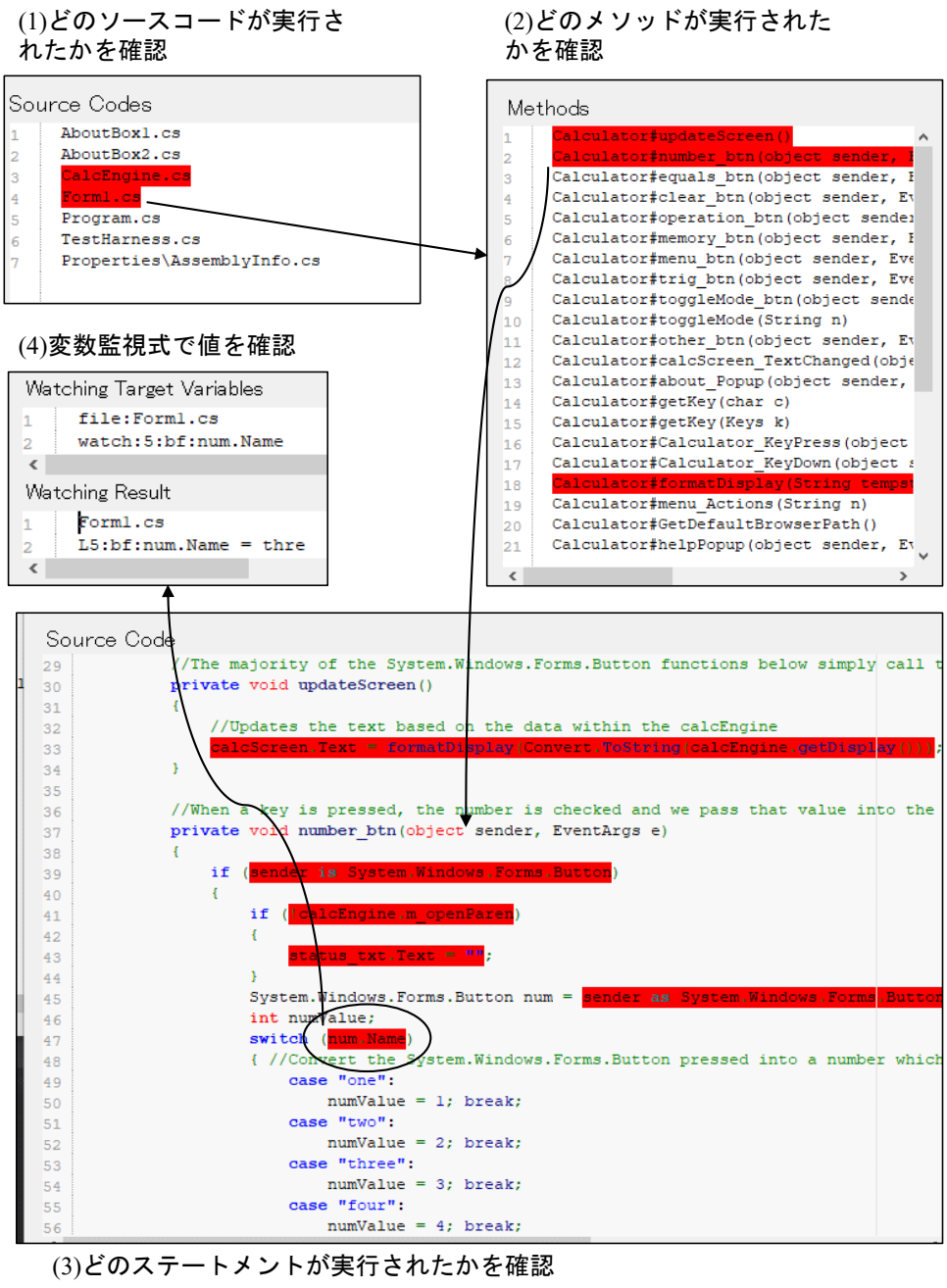


図 7　本実装方式におけるデバッグ実行の仕組み

ムアプリ，そして数値計算という 3 つプログラムを題材にデバッグ時のオーバヘッドの計測を行った．

1. 前節で用いた CalculatorCSharp で数字の 0 のボタンを押す操作を行う．
2. PongCSharp[†5](約 0.38KL) というピンポンゲームでゲームプレイする．一定間隔でユーザ入力受付，対戦相手の移動，ボールの軌道計算，描画な

†5　https://github.com/Cleod9/PongCSharp

表 3　オーバヘッドの計測結果

	実行回数	オリジナルプログラム (単位 ms)	デバッグ用プログラム (単位 ms)
1.CalculatorCSharp	100,000	7,360(100%)	10,344(141%)
2.PongCSharp	1,000	1,632(100%)	1,848(113%)
3.Fib(20)	5,000	3,002(100%)	112,533(3,749%)

どが行われる．

3. Fib：フィボナッチ数列を計算する．

3 つ目は本研究の対象領域ではないが，参考値として計測した．また，今回，実行経路情報の取得及び可視化のみを行い変数監視は行っていない．

評価結果を表 3 に示す．Fib のオーバヘッドが他の 2 つよりも非常に大きくなっているのは，オリジナルのプログラムにおいて，1 つのステートメントで行われている処理の計算量が小さいため，ステートメントごとにデバッグ情報を取得する処理のオーバヘッドが相対的に大きくなるためである．それに対して CalculatorCSharp，PongCSharp は 1 つのステートメントで行われている処理の計算量が多いため，相対的なオーバヘッドは Fib よりも小さくなった．特に，PongCSharp については，.NET のライブラリ側で行われるゲーム画面の描画処理の計算量が大きいため，オーバヘッドがより小さくなっている．

このように，実用的なプログラムではオーバヘッドが 50%未満におさまっており，デバッグという目的を考慮すると十分許容範囲であると判断できる．

7　まとめと今後の課題

本稿では，プログラムを停止させることなくプログラムの実行経路情報と変数情報を，実時間に出力し表示するデバッグの手法を提案し，C# プログラムのデバッガとして実装した．本デバッグ手法を用いることにより，開発者は実時間性や双方向性の高いプログラムを，停止させることなく，効率よくデバッグできるようになる．

今後の課題は 4 つある．第一に，本実装方式ではデバッグ実行のオーバヘッドをなるべく軽減したい．そのため，メソッドごとにデバッグ用実行するか，オリジナルのプログラムを実行するかを切り替えられる機構を実装している．しかし，デバッガから開発者が切り替える機能については現状備わっていない．これを解決するため，デバッガの UI 上で開発者が切り替えを行えるよう改良したい．第二に，デバッグ対象のプログラムによっては，通常の実行速度だと実行経路情報や変数情報の更新が早すぎて視認性が悪くなる可能性もある．これを解決するため，実時間性，双方向性を損なわない程度に実行速度を遅くする，スロー再生機能をデバッガへ追加したい．第三に，原因となるファイルやメソッドの数が多い場合，よりデバッグを効率化するためには，原因を絞り込む手段も必要となる．例えば，デバッグ支援技術 [16] のひとつである Spectrum-based Fault Localization [7] の考え方を応用するアプローチが考えられる．正常に動作したときの実行経路情報と期待通りではない動作をしたときの実行経路情報と比較し，その差異の部分のみを表示することで，バグの原因の箇所を絞り込める可能性がある．ケーススタディのデバッグの例であれば，ボタン 1 を押したときの実行経路情報を記録しておき，ボタン 3 を押したときに，ボタン 1 では通ってない実行経路情報のみを実時間にハイライトすれば，実行経路に差異のある `number_btn` メソッドを早期に絞り込める可能性がある．第四に，複数回実行されたステートメントや，複数回評価された監視対象の変数の式について，現状では詳細に確認するには監視区間を用いる必要があるが，監視区間を用いる前にもう少し俯瞰的に確認できるとよい．例えば，着目したい統計情報を色分けして表現する考え方 [8,9] を活用し複数回実行された実行経路の色を変えて表示する，複数回評価された監視対象の式の結果については，値を全て (もしくはサンプリングしていくつか) 保持しておき，どの値が何回現れたかを表示する，などの工夫が考えられる．

また，これらの機能を実装した上で，実用的に用いられているような様々なプログラムを題材として，提案手法の有効性を確認していきたい．

参 考 文 献

[1] Banken, H., Meijer, E., and Gousios, G.: Debugging Data Flows in Reactive Programs, *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA, ACM, 2018, pp. 752–763.

[2] Barr, E. T. and Marron, M.: Tardis: Affordable Time-travel Debugging in Managed Runtimes, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, New York, NY, USA, ACM, 2014, pp. 67–82.

[3] Barr, E. T., Marron, M., Maurer, E., Moseley, D., and Seth, G.: Time-travel Debugging for JavaScript/Node.Js, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, New York, NY, USA, ACM, 2016, pp. 1003–1007.

[4] Bell, J., Sarda, N., and Kaiser, G.: Chronicler: Lightweight recording to reproduce field failures, *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 362–371.

[5] Beller, M., Spruit, N., Spinellis, D., and Zaidman, A.: On the Dichotomy of Debugging Behavior Among Programmers, *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA, ACM, 2018, pp. 572–583.

[6] Czyz, J. K. and Jayaraman, B.: Declarative and Visual Debugging in Eclipse, *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, New York, NY, USA, ACM, 2007, pp. 31–35.

[7] de Souza, H. A., Chaim, M. L., and Kon, F.: Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges, *CoRR*, Vol. abs/1607.04347(2016).

[8] Eick, S. C., Steffen, J. L., and Sumner, E. E.: Seesoft-a tool for visualizing line oriented software statistics, *IEEE Transactions on Software Engineering*, Vol. 18, No. 11(1992), pp. 957–968.

[9] Gill, A. and Runciman, C.: Haskell Program Coverage, *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, New York, NY, USA, ACM, 2007, pp. 1–12.

[10] Koju, T., Takada, S., and Doi, N.: An Efficient and Generic Reversible Debugger Using the Virtual Machine Based Approach, *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, New York, NY, USA, ACM, 2005, pp. 79–88.

[11] Li, X. and Flatt, M.: Medic: Metaprogramming and Trace-oriented Debugging, *Proceedings of the Workshop on Future Programming*, FPW 2015, New York, NY, USA, ACM, 2015, pp. 7–14.

[12] Lin, Y., Sun, J., Xue, Y., Liu, Y., and Dong, J.: Feedback-Based Debugging, *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 393–403.

[13] Ressia, J., Bergel, A., and Nierstrasz, O.: Object-centric Debugging, *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, Piscataway, NJ, USA, IEEE Press, 2012, pp. 485–495.

[14] Rosenblum, D. S.: A Practical Approach to Programming With Assertions, *IEEE Trans. Softw. Eng.*, Vol. 21, No. 1(1995), pp. 19–31.

[15] Terada, M.: ETV: A Program Trace Player for Students, *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, New York, NY, USA, ACM, 2005, pp. 118–122.

[16] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F.: A Survey on Software Fault Localization, *IEEE Transactions on Software Engineering*, Vol. 42, No. 8(2016), pp. 707–740.

[17] Yanagisawa, Y., Kourai, K., and Chiba, S.: A Dynamic Aspect-oriented System for OS Kernels, *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, New York, NY, USA, ACM, 2006, pp. 69–78.

[18] Yin, H., Bockisch, C., and Aksit, M.: A Pointcut Language for Setting Advanced Breakpoints, *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, New York, NY, USA, ACM, 2013, pp. 145–156.

[19] Zhang, C., Yan, D., Zhao, J., Chen, Y., and Yang, S.: BPGen: An Automated Breakpoint Generator for Debugging, *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, New York, NY, USA, ACM, 2010, pp. 271–274.

[20] 丹野治門: ゲームプログラムに適したリアルタイム性の高いデバッガの提案と実装, 情報処理学会論文誌プログラミング（*PRO*）, Vol. 1, No. 2(2008), pp. 42–56.