Suspend-less Debugging for Interactive and/or Realtime Programs

Apr. 25, 2019

Haruto Tanno

Hideya Iwasaki

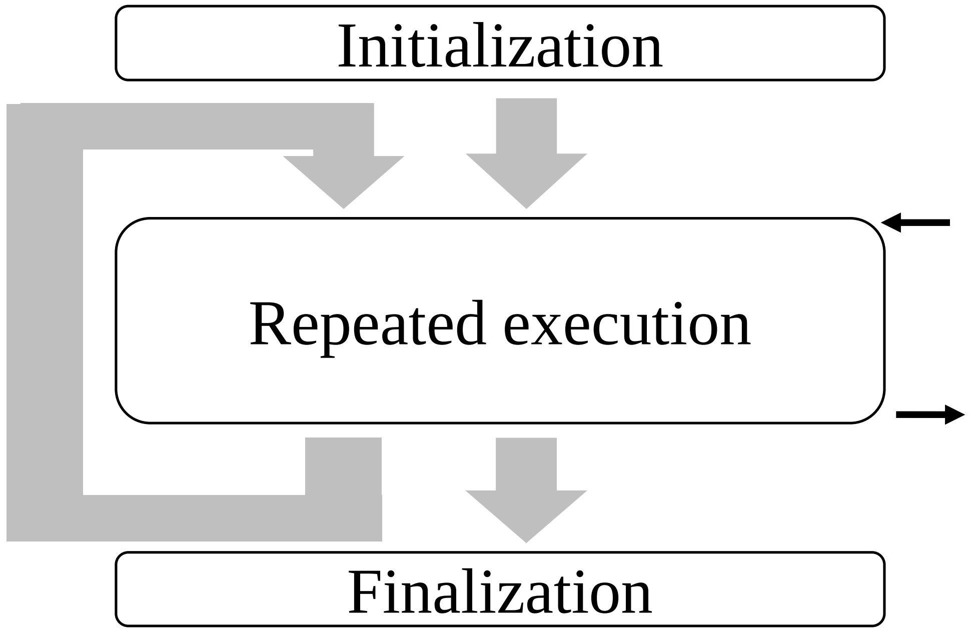
The University of Electro-Communications (Japan)

Outline

- Debugging interactive and/or realtime programs
  - GUI programs, action-game programs, network-based programs, sensor information processing programs
  - Traditional breakpoint-based debugging is NOT suitable.
- Suspend-less debugging
  - Features
  - SLDSharp (Suspend-less debugger for C#)
  - Case study (Demo)
- Implementation
- Related work
- Conclusion

Interactive and/or realtime programs

◆ Typical execution flow of interactive and/or realtime programs



Input
input events from user, from sensor devices, via the network, etc.

Output
output results onto a display, into memory as internal data, or to somewhere else via a network

- The timings and order of input event occurrences such as user operations are quite important.
- The behaviors of a program intricately change with the input events and the program’s internal states.

Debugging interactive and/or realtime programs

- The main target of debugging is “Repeated execution” part that occupies most of the execution time.

- Traditional breakpoint-based debugging
  – We have to suspend execution of target program to observe its internal state.

  ➔It is NOT suitable for debugging interactive and/or realtime programs.
    - Program will not behave as expected if its execution is suspended at a breakpoint.
    - Suspending a program to observe its internal states will degrade the efficiency of debugging.

Example: action game program

Game logic for the player character's attacks to enemies.

```
public void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList){
    int damagePoint = player.OffensivePower;
    if (input.AttackButton){
        // player attack strength increases five times
        if (input.DashButton) damagePoint *= 5;
        foreach (var enemy in enemyList){
            if (Vector3.Distance(player.Position, enemy.Position) <= 5.0) {// enemy is nearby
                if (!enemy.IsInvincible){   // enemy is not invincible
                    enemy.HealthPoint -= damagePoint;
                    if (enemy.HealthPoint <= 0) enemy.Dead(); // enemy is defeated
                }
            }
            enemy.EndPlayerCollision();
        }
    }
}
```

- Its behaviors change with the input events determined. ➔ Interactive
- It is executed at a certain interval. ➔ Realtime

Example: action game program

Game logic for the player character's attacks to enemies.

```
public void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList){
    int damagePoint = player.OffensivePower;
    if (input.AttackButton){
        // player attack strength increases five times
        if (input.DashButton) damagePoint *= 5;
        foreach (var enemy in enemyList){
            if (Vector3.Distance(player.Position, enemy.Position) <= 5.0) {// enemy is nearby
                if (!enemy.IsInvincible){   // enemy is not invincible
                    enemy.HealthPoint -= damagePoint;
                    if (enemy.HealthPoint <= 0) enemy.Dead(); // enemy is defeated
                }
            }
            enemy.EndPlayerCollision();
```

Pushing DushButton makes attack power five times stronger.
➔ The programmer expects that pushing AttackButton two or three times defeats an enemy, but the enemy is not defeated due to a bug.

e

- It is executed at a certain interval. ➔ Realtime

Example: action game program

There are several possibilities for the cause of this bug.

The value of input.AttackButton is false in spite of the player pushing it.

The power of the player character’s attack remains normal because the value of input.DashButton is not true.

EnemyList does not include the enemy character that is attacked.

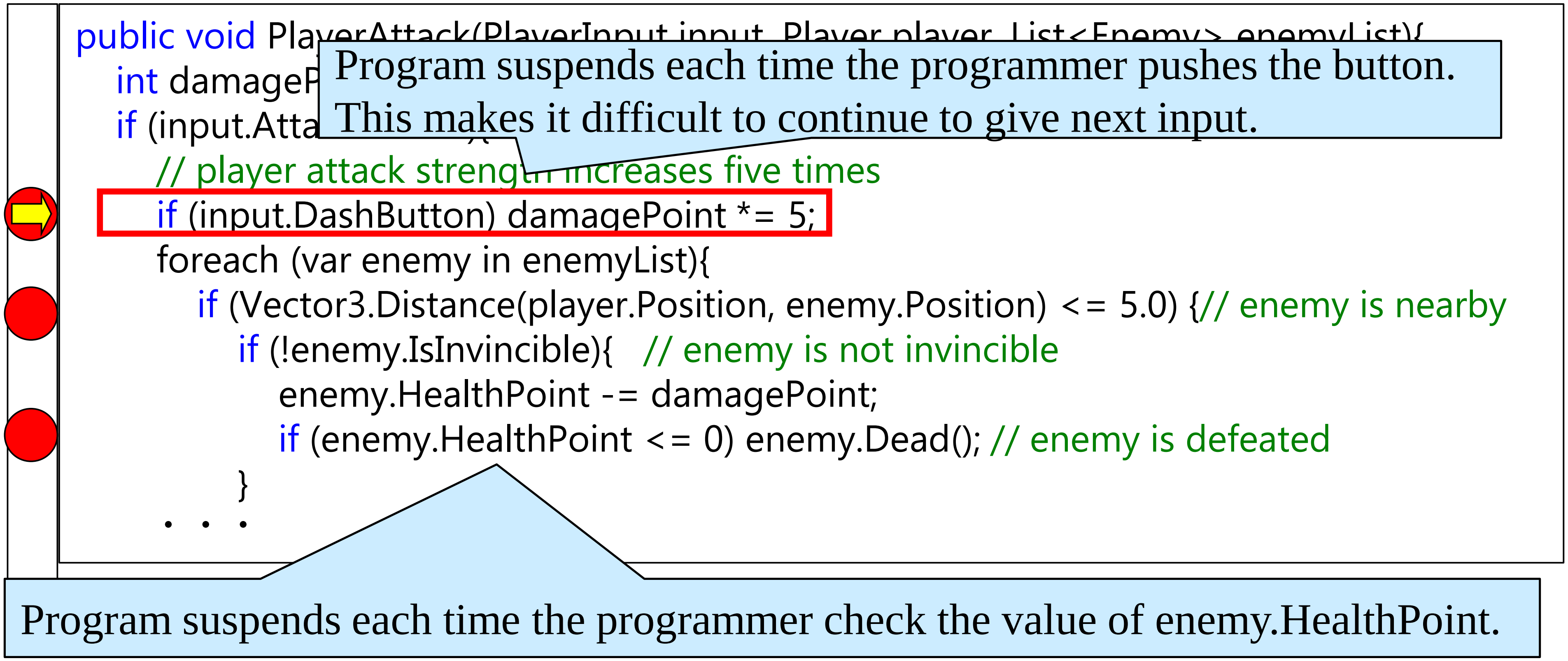
The result of Vector3.Distance is incorrect.

The value of enemy.isInvisible is unexpectedly true.

```
    int damag      nt = player.OffensivePo
    if (input.AttackButton){
        // player attack strength increases five times
        if (input.DashButton) damagePoint *= 5;
        foreach (var enemy in enemyList){
            if (Vector3.Distance(player.Position, e                                  by
                if (!enemy.IsInvincible){  // enemy is not invincible
                    enemy.Healt  Point -= damagePo
                    if (enemy.Healt    int <= 0) e                          d
                }
            }
            enemy.EndPlayer
        }
    }
}
```

Problems of breakpoint-based debugging

- It suspends execution of target program to observe its internal state.
  - Program will not behave as expected.
  - It will degrade the efficiency of debugging.



```
public void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList){
   int damageP
   if (input.Atta          ){
      // player attack streng          increases five times
      if (input.DashButton) damagePoint *= 5;
      foreach (var enemy in enemyList){
         if (Vector3.Distance(player.Position, enemy.Position) <= 5.0) {// enemy is nearby
            if (!enemy.IsInvincible){   // enemy is not invincible
               enemy.HealthPoint -= damagePoint;
               if (enemy.HealthPoint <= 0) enemy.Dead(); // enemy is defeated
            }
      . . .
```

Program suspends each time the programmer pushes the button.
This makes it difficult to continue to give next input.

Program suspends each time the programmer check the value of enemy.HealthPoint.

These problems are common to many interactive and/or realtime programs.

Our proposal: suspend-less debugging

- **Approach**
  - It visualizes both the information on the execution path and the values of the expressions of interest in realtime.
  - ➔It enables the programmer to interactively explore possible causes of a bug WITHOUT having to suspend the program.

- **SLDSharp: Debugger for C#**

Features of suspend-less debugging

(1) Currently executing place (execution path) and the values of expressions at a certain interval are presented in realtime.

➔The programmer can recognize the internal states immediately.

(2) Information on the execution paths is presented on three levels.

- File level, function/method level, statement level

➔The programmer can narrow down parts to be investigated step by step.

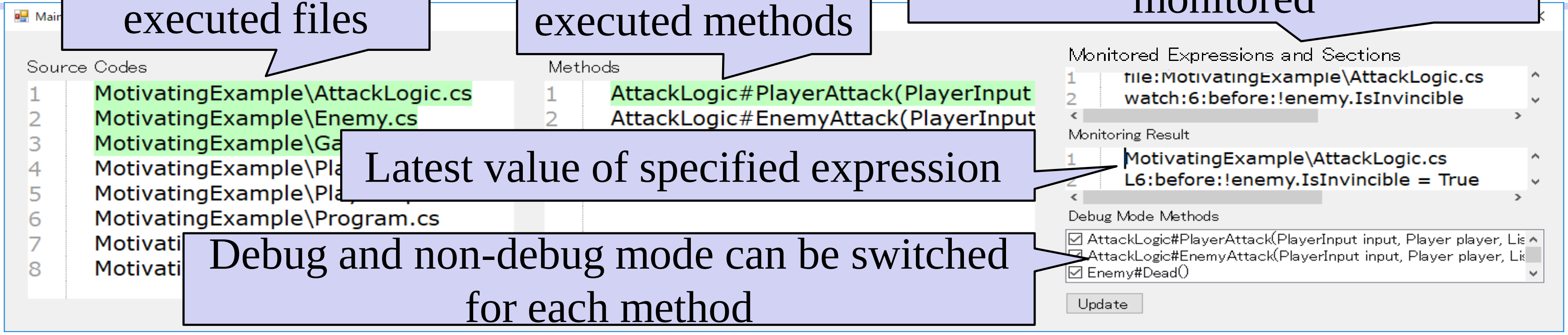
(3) Sections and conditions for visualization can be specified.

- E.g. focus on a specific element in for/foreach, a specific thread

➔The programmer can focus on the information of interest without any noisy information.

(4) Debug mode and non-debug mode can be switched dynamically.
➔The programmer can debug the program without extra overhead.

SLDSharp: debugger for C#



Highlights of executed files

Highlights of executed methods

- Expressions to be monitored
- Sections and conditions to be monitored

Latest value of specified expression

Debug and non-debug mode can be switched for each method

Source Codes

```
MotivatingExample\AttackLogic.cs
MotivatingExample\Enemy.cs
MotivatingExample\Ga
MotivatingExample\Pla
MotivatingExample\Pla
MotivatingExample\Program.cs
Motivati
Motivati
```

Methods

```
AttackLogic#PlayerAttack(PlayerInput
AttackLogic#EnemyAttack(PlayerInput
```

Monitored Expressions and Sections

```
file:MotivatingExample\AttackLogic.cs
watch:6:before:!enemy.IsInvincible
```

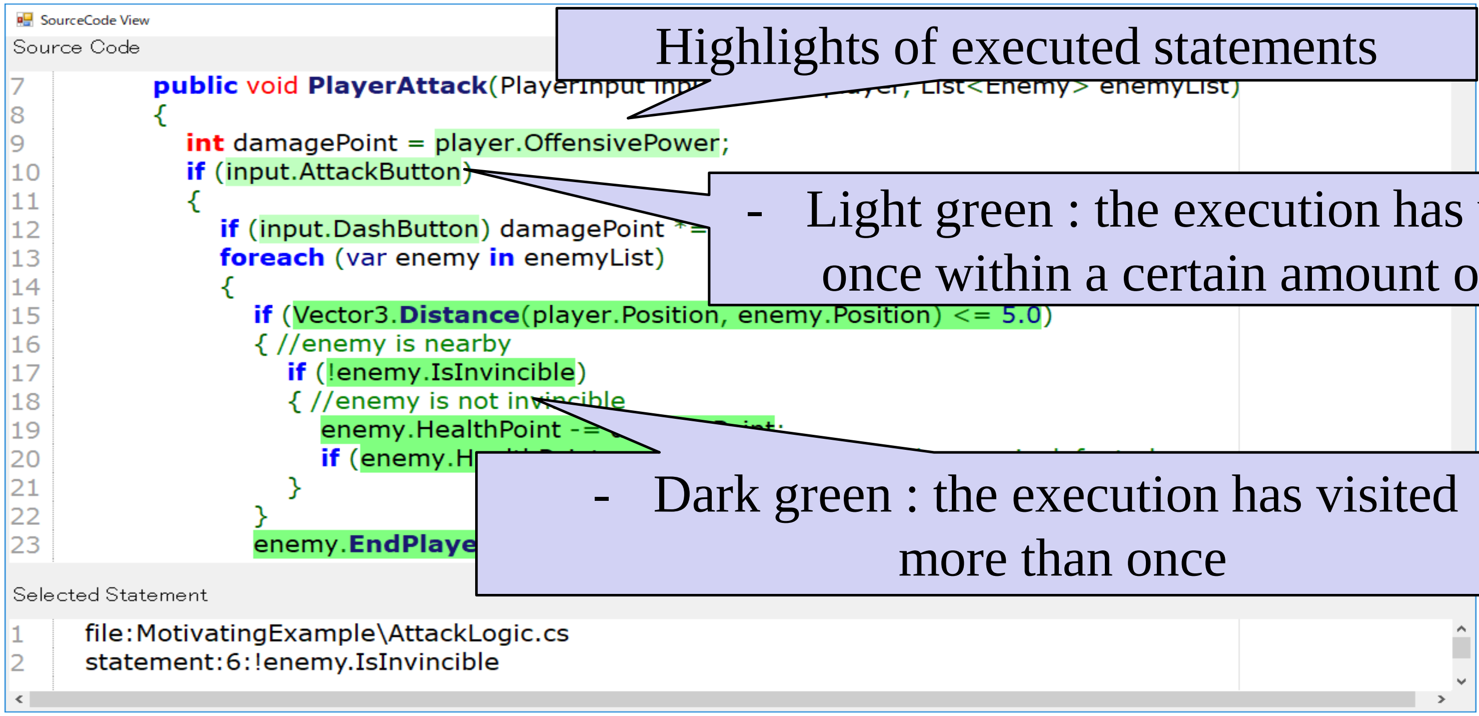
Monitoring Result

```
MotivatingExample\AttackLogic.cs
L6:before:!enemy.IsInvincible = True
```

Debug Mode Methods

- ☑ AttackLogic#PlayerAttack(PlayerInput input, Player player, Lis
- ☑ AttackLogic#EnemyAttack(PlayerInput input, Player player, Lis
- ☑ Enemy#Dead()

Source file list and method list view



Highlights of executed statements

- Light green : the execution has visited once within a certain amount of time

- Dark green : the execution has visited more than once

Source Code

```
        public void PlayerAttack(PlayerInput inp              ayer, List<Enemy> enemyList)
        {
          int damagePoint = player.OffensivePower;
          if (input.AttackButton)
          {
            if (input.DashButton) damagePoint *=
            foreach (var enemy in enemyList)
            {
              if (Vector3.Distance(player.Position, enemy.Position) <= 5.0)
              { //enemy is nearby
                if (!enemy.IsInvincible)
                { //enemy is not invincible
                  enemy.HealthPoint -=
                  if (enemy.H
                }
              }
              enemy.EndPlaye
```

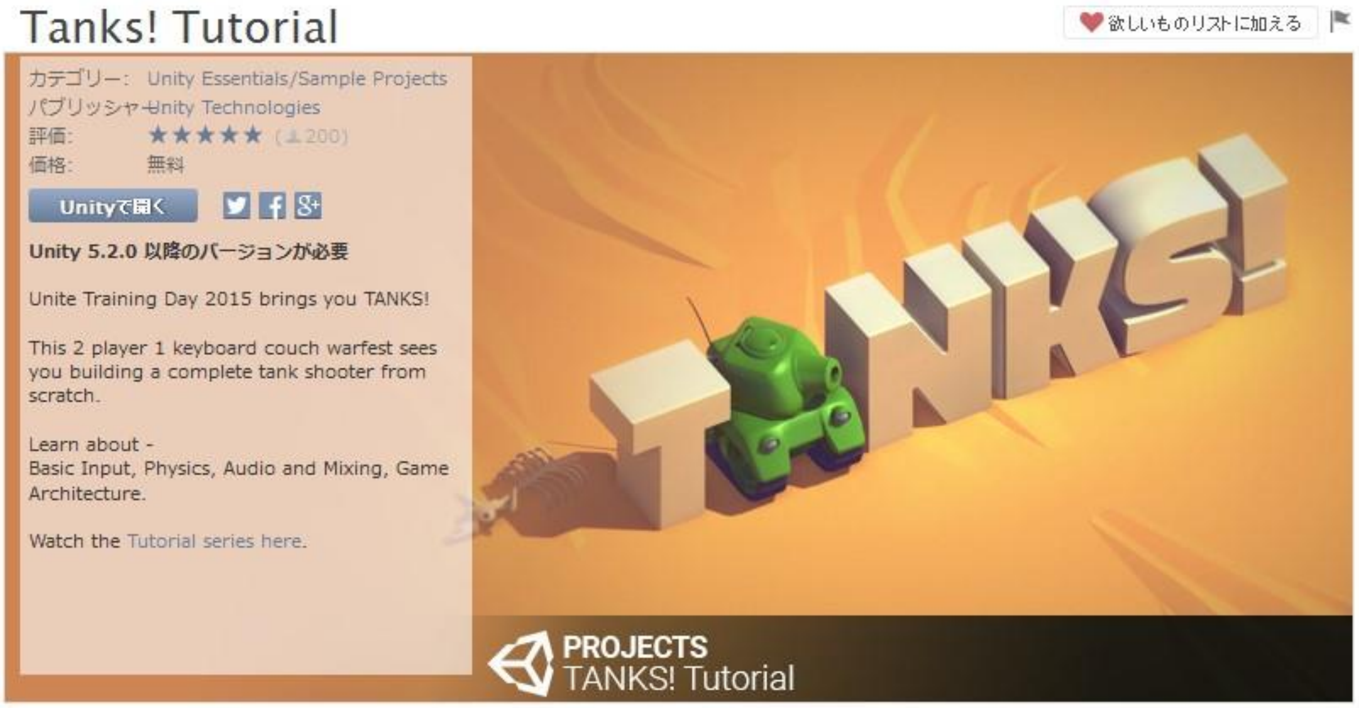
Selected Statement

```
file:MotivatingExample\AttackLogic.cs
statement:6:!enemy.IsInvincible
```

Source code view

SLDSharp: demo

- **Features**
- **Case study**



Tanks! Tutorial

欲しいものリストに加える

カテゴリー: Unity Essentials/Sample Projects
パブリッシャー: Unity Technologies
評価: ★★★★★ (200)
価格: 無料

Unityで開く

Unity 5.2.0 以降のバージョンが必要

Unite Training Day 2015 brings you TANKS!

This 2 player 1 keyboard couch warfest sees you building a complete tank shooter from scratch.

Learn about -
Basic Input, Physics, Audio and Mixing, Game Architecture.

Watch the Tutorial series here.

Unity Tanks! (1.5KL C# program)
・Each of the two players controls a tank and shoot shells at the opponent tank to destroy.

https://www.youtube.com/watch?v=iI-WG13qx8c

Features of suspend-less debugging

Reshown

(1) Currently executing place (execution path) and the values of expressions at a certain interval are presented in realtime.

➔The programmer can recognize the internal states immediately.

(2) Information on the execution paths is presented on three levels.

- File level, function/method level, statement level

➔The programmer can narrow down parts to be investigated step by step.

(3) Sections and conditions for visualization can be specified.

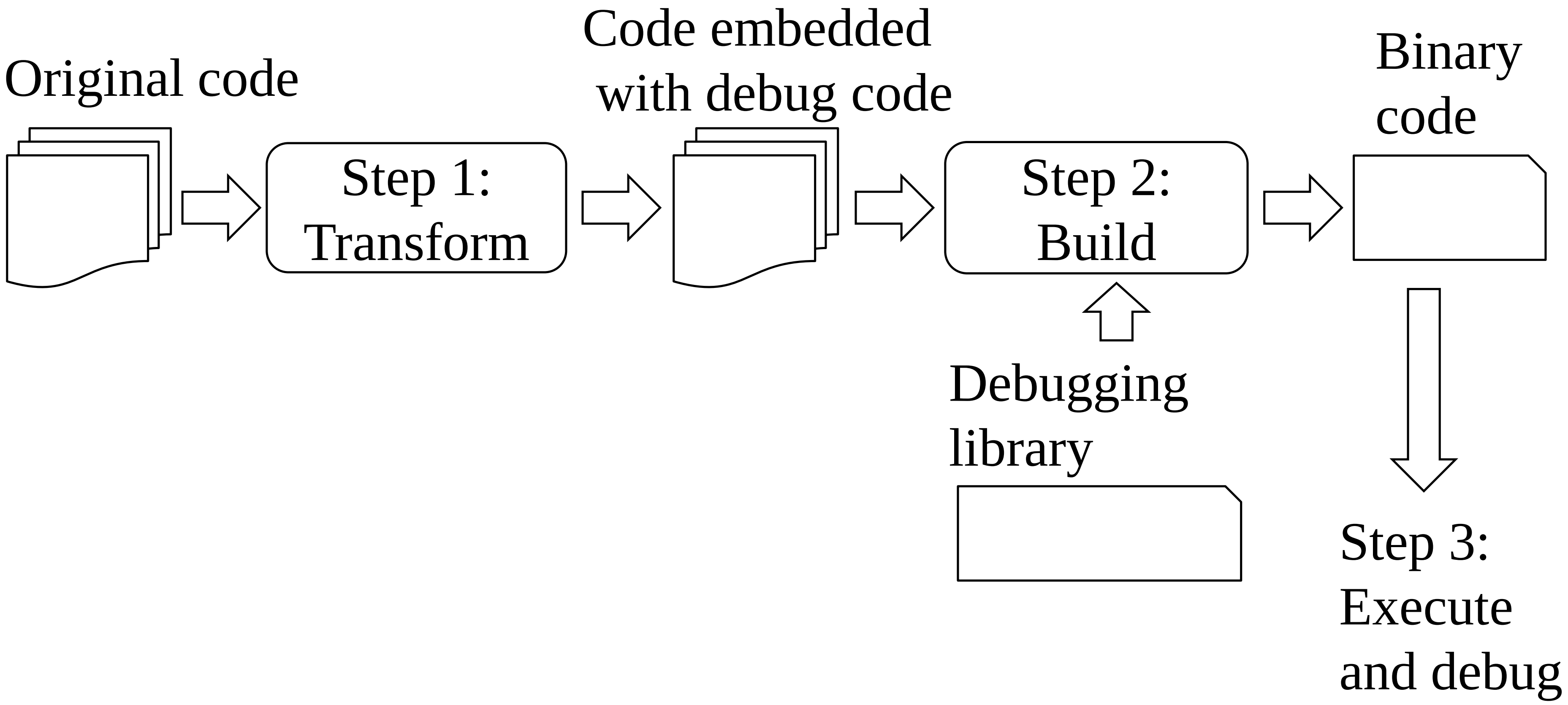
- E.g. focus on a specific element in for/foreach, a specific thread

➔The programmer can focus on the information of interest without any noisy information.

(4) Debug mode and normal mode can be switched dynamically.
➔The programmer can debug the program without extra overhead.

These features do not restrict the domain of debuggee programs and are applicable to general programming languages.

Implementation



Embedding debug code is automatically done by code transformation.
➔The programmer need not to make any changes to the debuggee code.

Mechanism of code transformation

Original code

```
public void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList){
   int damagePoint = player.OffensivePower;
   if (input.AttackButton){
…
```

Code transformation

Code embedded with debug code

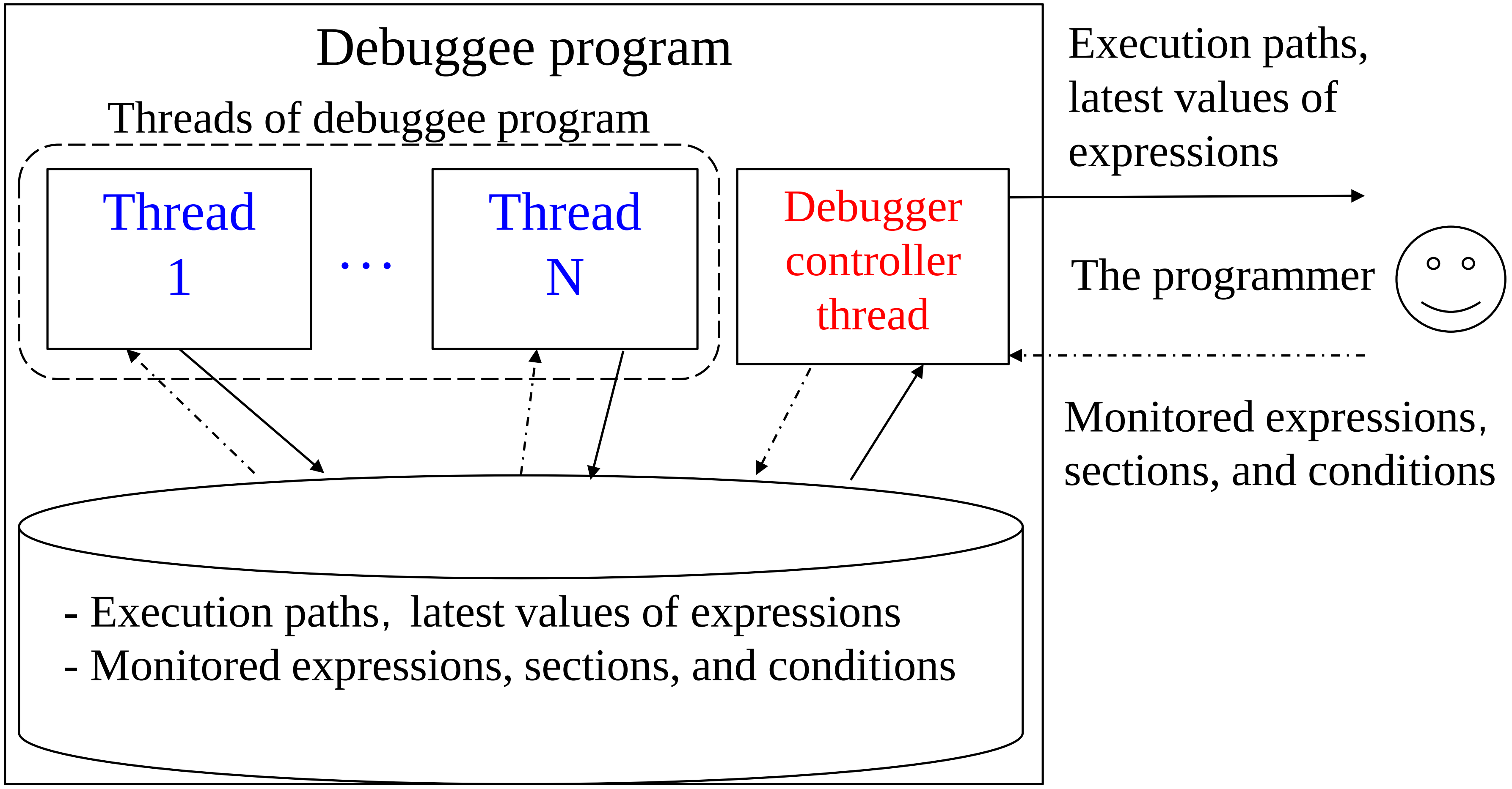
```
public void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList){
   if (_Logger.IsLogging(0)){ _debug_PlayerAttack(input, player, enemyList);}//debug
   else                      {_original_PlayerAttack(input, player, enemyList);}//non-debug
}

public void _debug_PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList){
   var _logger = _Logger.GetLogger(0, 0);
   int damagePoint = _logger.LogFunc(() => player.OffensivePower, 0);
   if (_logger.LogFunc(() => input.AttackButton, 1))
…
```

Can be switched dynamically

```
public T <T>(LogFunc<T> statement,  int stateId){
   DebugProc(…);
   return statement();  }
```

Mechanism of debugging execution

- **Debugger controller thread runs as an extra thread in a debuggee process.**
- **Each thread except the debugger controller thread records logs generated by the embedded code.**



Debuggee program
Threads of debuggee program
Thread 1
…
Thread N
Debugger controller thread
Execution paths, latest values of expressions
The programmer
Monitored expressions, sections, and conditions
- Execution paths, latest values of expressions
- Monitored expressions, sections, and conditions

Related work

- **Log based debugging**
  - It collects logs automatically without suspending the debuggee program and exploits the obtained logs **AFTER** its execution.
  - Programmer cannot see debug information in realtime.
    ➔ Trial and error style debugging is difficult.
  - It is impractical to collect all information due to the overhead.
    ➔ The programmer may be unable to find desired information in the logs.
- **Time travel debugging**
  - It records all inputs to the program and reproduce the program execution.
    - E.g. Java[Barr et al, 2014], JavaScript/Node.js[Barr et al, 2016]
  - This approach is a log-based one. (Trial and error style debugging is difficult.)
  - There is a technical hurdle for implementing perfectly replaying execution.
    - E.g. multithreaded programs do not always replays perfectly.

  ➔ The programs to which this method can be applied are restricted.

Conclusion

- **We propose suspend-less debugging.**
  - Displays information on execution paths and the values of expressions in a program in realtime without suspending program execution
- **We implemented it in SLDSharp, a debugger for C# programs.**
- **We demonstrated its effectiveness through a case study using a game program.**

- **Future work**
  - Examine the effectiveness of the suspend-less debugging on more various subjects
    - E.g. network-based programs, and sensor information processing programs
  - Implement the proposed debugger for languages other than C#
    - E.g. JavaScript

Thank you for your attention!