一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2020-3(2020-07)

ImperSD: Java 言語向け命令型スクリプタブルデバッグ環境

平ノ内奎太† 　小林　隆志†

† 東京工業大学 情報理工学院 情報工学系　〒 152–8550 東京都目黒区大岡山 2–12–1
E-mail: {khiranouchi, tkobaya}@sa.cs.titech.ac.jp

あらまし　デバッグの工程における開発者は何度もプログラムの内部観測作業を繰り返すことがある．これを解決する方法としてデバッグの手順をスクリプトとして記述し作業を自動化できるスクリプタブルデバッガ (SD) が提案されている．既存の SD は，リアクティブな記述を用いているため，初期化処理や過去の観測結果の履歴を利用する処理などが書きづらい，副作用のある式の実行順序が定まらないといった欠点がある．本研究では，デバッガが行う動作をそのままの順序でスクリプト記述する命令型 SD を提案する．本稿では，命令型 SD が満たすべき要件を定め，それらを満たす命令セットを提案する．また，Java 言語向けの命令型 SD 環境として開発した ImperSD について説明する．実際にデバッグ手順を記述し，既存 SD と比較することで提案 SD の得失を議論する．

キーワード　Debugging; Scriptable Debugger; File System; Dynamic Analysis; Software Maintenance

1. はじめに

デバッグの工程において，開発者はしばしば何度も同じ作業を繰り返すことがある．これは，時間を浪費する面倒な作業であるだけでなく，集中力の欠如によりバグを見逃すことにつながる可能性もあり問題である [1].

このような同じ作業の繰り返しを解決する方法の 1 つとして，スクリプタブルデバッガ (以下 SD) がある．SD とはスクリプト言語を用いて記述された操作手順に従って動作するデバッガであり，観測手順の自動化や観測結果に応じた処理を記述することで，繰り返し行うデバッグ作業の効率化が可能となる．

SD に関する既存研究はこれまでに，GDB の文法に条件文とイベントを追加して拡張したもの [2] や，データフロー言語をスクリプト言語として用いたもの [1] など様々なツールが提案されている．これらの既存の SD においては，ユーザはイベントが発生したときのデバッグ処理を記述することで，デバッガの動作を指定することができる．例えば，複数の変数の値がある条件を満たした場合にのみ何らかの動作を行うといったスクリプトを事前に記述しておくと，対象プログラムがデバッグ実行中にその条件を満たしたとき，デバッガは指定した動作を行う．本稿ではこのような既存の SD を**リアクティブ型 SD** と呼ぶ．これらのリアクティブ型 SD は，デバッガの動作を何らかのイベント発生時の処理として記述する必要があるため過去の観測結果の履歴を利用するような処理が直観的に書きづらいなどのデメリットがある．また，デバッグ記述のために独自の言語やユーザが慣れ親しんでいない言語を用いる SD の場合，その言語の学習にかかるコストも無視できない．

本稿では，既存の SD の問題点を解決する Java 言語向けの**命令型 SD** を提案する．提案する SD では，観測時のイベントに対する反応としてデバッグ操作を記述せず，観測結果を任意のタイミングで非同期に取得し，ユーザによってデバッガの動作を制御する命令型の記述を可能とする．例えば，プログラム中のある行の変数に観測ポイントをセットし，しばらく待ち，その変数の観測値の変化を分析して出力するといった手順を記述しデバッガを制御する．SD の命令セットには，観測ポイントのセットやクリアを行う命令，タイムスタンプ付きの観測値を取得する，スリープを挟む命令といったものを用意する．

本研究では，提案する命令型 SD に対する要件を定義し，その要件を満たす命令セットを定義する．また，観測結果に対する非同期アクセスと既存ツールとの相互運用性を確保するために，我々が UNIX の procfs から着想を得て開発した内部状態観測基盤 MewdFS [17] を観測対象とスクリプト実行部との間に導入する．観測結果へのアクセスをファイルシステムへのアクセスで表現することで，利用者の学習コストを低減し，既存ツールを容易にデバッグ作業に組み込むことを可能とする．

また，本稿では，提案手法を実装した Java 言語向け命令型 SD である **ImperSD** について紹介する．ImperSD はスクリプト言語にデバッグ対象言語と同じ Java 言語を採用し，SD の命令セットを Java ライブラリとして提供する．スクリプトの実行環境に JShell を用いることで REPL をサポートする．

本研究では提案手法の有用性を議論するために，ImperSD を用いて実際のアプリケーションにおけるデバッグシナリオの記述を行ない提案 SD の得失を確認した．

以下では，まず 2. で，本研究に関連する研究を紹介する．3. で提案手法，4. でその実装である ImperSD について説明した後，5. で記述実験について述べる．

2. 関連研究

デバッグ作業の自動化に関する初期の研究の多くはコールバック指向であり，ユーザがコールバックを通して，イベントに対応するデバッグ処理を逐一記述する方法をとる．

Johnson らはコールバック指向のデバッグスクリプト言語 Dispel [3] によって記述するデバッガ RAIDE [4] を提案している．また，C 言語などの汎用言語に似た言語をスクリプト言

語として用いる Acid [5] や，Tcl/Tk をベースにしたスクリプト言語を用いる Deet [6] などが提案されてきた．Deet は，Tk を生かした様々なグラフィカルな機能が特徴であり GUI ベースの操作が可能なほか，対象プログラム中のデータ構造をグラフィカルに表示することも可能である．

また，GNU Debugger(GDB) に代表される既存のデバッガの操作を自動化する代表的な手法として，Dalek [2] や Duel [7], Expositor [8] などがある．Delak は GDB の文法に条件文とイベントを追加したスクリプト言語で作業を記述可能であるほか，関数呼び出し等の発生する低レベルイベントから，高レベルイベントを生成し，処理の起動条件を高レベルイベントの条件で指定するデータフロー型の記述も可能である．なお，現在の GDB や，LLDB では操作コマンドをスクリプトとして記述し自動化できるほか，C++や Python の API が提供されており，それらの言語でのスクリプトによる制御が可能である．

近年では，関数型リアクティブプログラミング (FRP) を用いることで，データフローに着目した処理を記述するリアクティブ型の SD が提案されている [1,8,9]．リアクティブ SD では，変数の更新などのイベントが発生したときの処理すべき内容をスクリプト記述することで，観測結果のデータフロー処理を実現する．代表的なものに，イベント指向のデータフロー言語 FrTime をスクリプト言語として用いた SD である MzTake [1] がある．MzTake [1] は，Java Debug Wire Protoco(JDWP) によって JVM 上のプログラムへアクセスする Java 言語を対象にしたデバッガであり，提案手法に最も近い SD である．MzTake では FrTime の持つ豊富なライブラリを用いてデータフロー処理を記述することが可能であることに対し，提案手法では観測結果を保持する仮想的なファイルシステムに対するアクセスの形で処理を記述し，必要に応じて任意のツールとの連携が可能である点が異なる．FPR 形式の SD には他にも，GDB の Python API を拡張したスクリプト言語によって記述可能である Expositor [8] や，Scala を用いて記述する SDN(Software-defined network) 向けのスクリプタブルデバッガ Simon [9] などがある．

本研究が特に着目する SD の利用方法である内部状態の観測結果の加工については，デバッガだけでなくモニタリングツールとも関係が深い．

Java 言語を対象としたモニタリングツールには，ユーザが実行前に SQL ライクなクエリ言語を用いて問い合わせる内部オブジェクトの条件を指定することでができる dynamic query-based debugger [10] がある．また，このツールを改良し，対象プログラムの実行中に対話的にクエリを発行できるツールも提案されている [11,12]．対象は C 言語に限定されるがモニタリングツールのフレームワークとして Alamo [13] や，UFO [14] などの提案がある．

このように，多くの SD に関連する研究がなされているが，Java 言語向けの命令型 SD と言えるものは提案されていない．

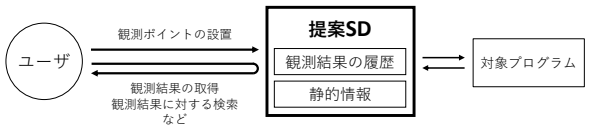

図 1　提案する SD の全体構成

3. 提案手法: 命令型 SD

3.1 命令型 SD の要件

前述のようにリアクティブ型 SD には，コールバック指向形式や時変値に対するデータフロー処理を記述する FRP 形式がある．これらに共通する特徴としてデバッガの行う処理はあるイベントの発生により駆動され，ユーザがそのタイミング・順序を指定することができないということがある．そのため，初期化処理，過去の観測結果の履歴を利用する処理，定期的に実行される処理，複数のイベントを受けて駆動される処理の記述が煩雑となるほか，副作用のある式など処理順序が定まらないと困るケースに対応することが難しい．

そこで，本研究ではデバッガが行う動作を命令型プログラミングのスタイルでスクリプト記述する命令型 SD を提案する．命令型 SD の全体構成を図 1 に示す．命令型 SD はユーザがスクリプトによって命令を与えることで対応した動作を行う．例えば，ユーザが観測ポイントのセットを行う命令を SD に送ると，SD は対象プログラムの観測を開始しその履歴を保持する．

本研究では命令型 SD に対する要件として以下を定める．

- ユーザがスクリプトに書いた順序でデバッガが動作する
- 観測値に対して非同期にアクセスできる

これらの要件が必要となる根拠は以下のとおりである．まず，デバッガが処理を行う順序やタイミングを，ユーザがスクリプト中で任意に指定することができるようにするためには，指定した時間だけ待機するなどタイミングを調整する指定を記述できる必要がある．また，観測対象の観測とその結果に対する処理は別のタイミングで行われる必要がある．観測ポイントを設置してから観測結果を取得するまでの間に，観測対象地点での観測は複数回行われることもあるため，観測された観測値の履歴を観測された時刻であるタイムスタンプを添えてデバッガ自身が保持し，ユーザの必要に応じて提供する必要がある．

次に，デバッガとして最低限必要であると考えられる要件を考える．一般的な対話的デバッガが共通して持っている機能は，指定行へのブレークポイントのセット，実行再開，ステップ実行，プログラム状態（行番号・コールスタック・変数値の組）の取得である [15]．提案する命令型 SD は観測結果へ非同期アクセスを行うため，ブレークポイントで対象の実行を一時停止する形式ではなく，実行を継続しつつ指定したプログラム行の変数の値を観測する形式を考える．実行再開やステップ実行といった対象プログラムに対する実行制御については本手法では対象外とする．

また，本研究ではオブジェクト指向言語を対象とするため，観測結果についてもオブジェクト単位で取り扱う必要があり，非同期アクセスのために観測結果の履歴へアクセスするためそ

```
p = 観測ポイントをセット("…/var1/14")
スリープ(60sec)
result1 = p.観測結果の履歴を取得()
if (result1が条件を満たす) {
  別の観測ポイントをセット
  ……
}
……
filepath = p.観測結果へのファイルパスを取得()
外部コマンド実行(cd filepath)
外部コマンド実行(gnuplot ……)
```

図 2　スクリプトの記述例 1（一括実行）

```
> ClassAにあるメソッド一覧を取得
ClassA methodA methodB
> methodBのコードを表示
……
> p = 観測ポイント"ClassA.java,var1,14"をセット
> p.オブジェクトハッシュ値の一覧を取得
150123456 261234567 372345678
> p.観測結果の履歴を取得(150123456)
[……, 984, 985, 985, 986]
> p.値検索("[1-9][0-9]*")
……
```

図 3　スクリプトの記述例 2（対話的処理）

れらに対する検索機能も必要となる．これらの機能を有するデバッガは著者らの知る限り存在しないが，実行中の Java プログラム内の文字列を検索する手法 [16] が提案されていることからも，その必要性は高いと考える．また，観測履歴に対する処理をスクリプト記述だけでなく，外部プログラムでの処理を可能とすることで，スクリプト記述のコストを削減することも必要である．

3.2　命令型 SD の命令セット

本研究では命令型 SD に必要な命令セットを以下と定めた．以下では，命令セットのそれぞれの命令の詳細について，スクリプトの例（図 2,3) を使って説明する．

- 時間指定の待機命令
- 静的情報の取得
- 観測ポイント（行番号・変数名の組）の設置
- 観測結果の履歴の取得
- オブジェクト毎に観測結果を取得
- 観測結果に対する検索
- 外部プログラムの実行

スクリプトの例では自然言語を用いて命令を記述しているが，**??**で同様の内容を，提案手法を実装した SD「ImperSD」のスクリプト言語を用いて記述する例を紹介する．

命令セットは図のスクリプト例のように時系列順に記述する．例えば，図 2 のように，まず観測ポイントをセットし，しばらく待機したあと，観測結果の履歴を取得するといった具合に記述する．観測ポイントを指定するためには，ファイル名，行番号，変数名を指定する方法だけでなく，静的情報を利用して対話的に選択する命令が用意されている．図 3 の 1 コマンド目は，クラス ClassA に存在するメソッドの一覧を取得している例である．変数に対しては，観測ポイントとしてセット可能な行番号の取得や該当するコードを表示することもできる．

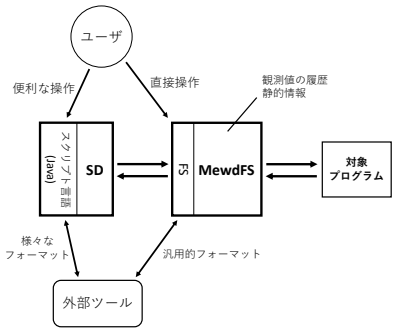

図 4　命令型 SD「ImperSD」の全体構成

観測ポイントをセットすると，SD は指定した行番号を通過するたびに指定した変数の値を記録しタイムスタンプとともに SD 自身が保持する．図 2 の 1 行目は，ファイルパスを指定して観測ポイントをセットしている例であり，var1 という変数に 14 行目にて観測ポイントをセットしている．SD はこれ以降，対象プログラムが 14 行目を通過するたびに，変数 var1 の値を記録して履歴として残すようになる．一方，図 3 の 3 コマンド目では，ファイル名・変数名・行番号の組を用いて観測ポイントをセットしている．図 2 の 3 行目では，1 行目にてセットした観測ポイントにおける観測結果の履歴を取得している．この例ではその観測結果を用いて条件文を記述し，観測結果によって処理を分岐するといったことを行っている．

オブジェクト指向言語を対象とするために，観測結果の履歴は，オブジェクト毎に分けて取得することができる．また，ある観測ポイントの持つオブジェクトのハッシュ値の一覧を取得することもできる．図 3 の 4 コマンド目では，3 コマンド目にてセットした観測ポイントにおけるオブジェクトのハッシュ値の一覧を取得している例である．その後，そのうち 1 つのハッシュ値を用いて，特定のオブジェクトのみから取得した値の履歴を取得している．また，観測結果の履歴に対して，検索を行うことができる．図 3 の最後のコマンドでは，3 コマンド目にてセットした観測ポイントの観測結果の中から，正の整数であるものを検索している．

スクリプト中にて外部コマンドを実行することができる．外部コマンドに対して標準入力を渡すことができ，外部コマンドの実行による標準出力を受け取ることもできる．図 2 の 9 行目では，1 行目にてセットした観測ポイントの観測結果へのファイルパスを取得している．11 行目では，外部コマンドである gnuplot を実行して，観測結果の変数値の時間変化をグラフに描画している．

4.　実　　装

4.1　概　　要

提案手法を実現した Java 言語向け命令型 SD「ImperSD」の全体構成を図 4 に示す．

ImperSD では，SD 部と観測対象は直接接続しないため，JVM との接続についての制約はないが，利用者の学習コストを考慮しスクリプト言語に Java を採用した．前節で定義したデバッグ操作のための SD の命令セットは Java のライブラリ

として提供されている．

SD 部のスクリプト実行環境には，Java 言語を対話的に実行することのできる JShell を採用し，REPL(Read-Eval-Print Loop) を実現している．JShell を用いたことにより，対話中のメソッド名等の入力補完や，表示時の暗黙の toString() 呼び出しなどの利点が生まれた．

内部状態観測基盤として，我々が以前提案し改良を続けているツールである MewdFS [17] を用いた．MewdFS は，インタフェースとしての役割を担う仮想ファイルシステム部と観測対象プログラムの観測を行う内部状態観測部から構成されている．内部状態観測部には，嶋利らが提案した ProbeJ [18] を利用している．ProbeJ は JVM Tool Interface (JVMTI) を利用することで，実行中の Java プログラムを対象とした低侵襲な実行モニタリングツールであり，実行中の Java プログラムに対して観測ポイントの設置と，観測ポイントにて観測された変数値の取得を行うことができる．対象プログラムの実行を継続したまま（短時間の停止だけで）読み取ることができるため，プログラムの実行への影響を最小限にとどめることを実現している．

MewdFS は telnet 通信により ProbeJ と通信し，観測ポイントのセット・クリア，観測情報の受け渡しを行い，観測情報を仮想ファイルシステム上のファイルとして提供する．観測結果の履歴の最大個数は，あらかじめ設定したバッファサイズで決められており最大個数以上の履歴は古いものから上書きされる．観測情報はソースコードの構造と一致するようディレクトリ構造を構成し，ファイルシステムを操作することで観測対象プログラムのデバッグ操作を行うことができる．そのほか MewdFS の使用方法およびファイルシステムインタフェースの詳細については文献 [17] を参照されたい．

SD 部は，isd という C++で記述された単一のプログラムで実現され，仮想ファイルシステム上への観測対象プログラムのマウントや，スクリプト実行環境の起動などを行う．ImperSD の動作環境は仮想ファイルシステムライブラリの制約により Windows のみだが，デバッグ対象については ProbeJ がエージェントとして指定できる任意の JVM を指定できる．

4.2 ImperSD の使用方法

ImperSD でのデバッグは，1) 観測対象プログラムを実行，2) 対象プログラムを仮想ファイルシステムにマウント，3) 記述済みスクリプトを実行・対話環境でのデバッグという手順で行う．

まず，デバッグ対象のプログラムを，ProbeJ を JVM のエージェントに指定して実行する．その後 isd の mount コマンドを用いることで，デバッグ対象プログラムの情報を MewdFS の仮想ファイルシステム上にマウントする．複数プログラムのマウントが可能であり通信ポート番号によって区別される．

マウントした後に script コマンドによってスクリプト実行環境を起動する．指定したスクリプトを実行できるほか，提案命令セットと JShell による REPL 環境が利用できる．

4.3 スクリプト言語の API

SD の命令セットライブラリは，Mewdfs クラス，Probe クラス，ProbeEx クラス，Static クラスの 4 つのクラスからなる．それぞれの役割を図 5 に示す．

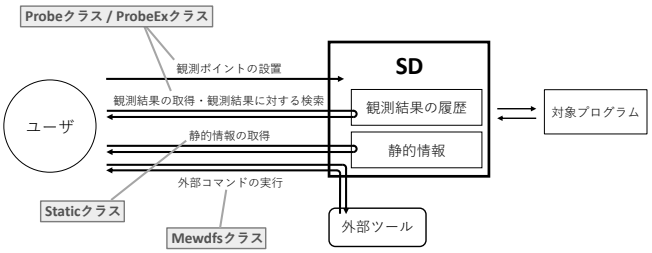

図 5　スクリプト言語の API の全体像

```
var p = new ProbeEx("y:/program1/pkg.ex.com/ClassA/methodB/var1/14")
Thread.sleep(60000)
var result1 = p.getValsList(-1)
if (result1が条件を満たす) {
  var p2 = new ProbeEx(……)
  ……
}
……
var filepath = p.getFullPath()
mfs.exec("cd " + filepath)
mfs.exec("paste <(stat * -c %Y) <(cat *) | gnuplot -e ¥"plot '-'¥"")
```

図 6　ImperSD でのスクリプトの記述例（一括実行）

```
> var s1 = new Static("y:/program1/pkg.ex.com/ClassA")
> s1.list()
[[path="y:/program1/pkg.ex.com/ClassA/ClassA"],
[path="y:/program1/pkg.ex.com/ClassA/methodA"],
[path="y:/program1/pkg.ex.com/ClassA/methodB"]]
> s1.child("methodB").src()
……
> var p = new Probe( "program1", "ClassA.java", "var1", "14")
> var list = p.getHashList()
[150123456, 261234567, 372345678]
> p.getVals(list.get(0))
[……, 984, 985, 985, 986]
> p.searchVal("[1-9][0-9]*")
["y:/program1/pkg.ex.com/ClassA/methodB/var1/14/00",
"y:/program1/pkg.ex.com/ClassA/methodB/var1/14/03",
"y:/program1/pkg.ex.com/ClassA/methodB/var1/14/06"]
……
```

図 7　ImperSD のスクリプト記述例 2（対話的処理）

Probe クラスは観測ポイントを表現している．観測ポイントの位置情報などを指定して生成され，`set()`，`getVal()`，`getTimeValsList()`，`searchValHash()` などの観測開始，観測結果の取得，オブジェクト単位でのアクセスなどを行う API が定義されている．ProbeEx クラスは，Probe クラスのサブクラスであり，観測ポイントをセットしたのち MewdFS に対して一定頻度で自動で問い合わせを行う API を提供する．Static クラスは，観測対象プログラムの静的情報の取得を行うためのクラスであり，Static クラスのインスタンスが，メソッドや変数などの 1 つの静的な要素を表す．

前節で説明した自然言語によるスクリプト記述例と対応する ImperSD の記述を図 6，図 7 に示す．

5. 評　　価

本節では，文献 [1] で用いられている 2 つのデバッグシナリオを使用し，命令型 SD とリアクティブ型 SD との比較を行う．

5.1 Case1:ダイクストラアルゴリズムのデバッグ

文献 [1] では，ダイクストラアルゴリズムを Java 言語で実装したプログラムを MzTake を用いてデバッグする例が挙げ

られている．対象の DijkstraSolver クラスでは PriorityQueue 型の変数 q に，与えられたグラフの全てのノードを格納し，その後，q が空になるまで extractMin() メソッドを用いて取り出し，そのノードの隣接ノードについて距離の計算する．以下では，MzTake におけるシナリオに従い，開発者はまずこの PriorityQueue の実装に問題があるのではないかと仮説を立て，デバッグを行うという状況を想定する．

5.1.1 MzTake によるデバッグ

リスト 1 は文献 [1] における FrTime 言語での記述である．PriorityQueue クラスの特定行を通過するたびに発生するイベントを用いて，add() の 1 行目を通過するたびに'reset という値を，extractMin() の最終行目を通過するたびに要素（Node 型）のフィールド weight の値を記録し，記録されたものをマージする．マージ結果の最新 2 つの値を比較し，結果が不正の場合はプログラムの実行を一時停止している．

リスト 1: FrTime による記述 [1](Case1)

```
1  (define c (start-vm "DijkstraTest"))
2  (define queue (jclass c PriorityQueue))
3  (define inserts
4    (trace ((queue . jdot . add) . jloc . entry)
5      (bind (item) (item . jdot . weight))))
6  (define removes
7    (trace ((queue . jdot . extractMin) . jloc . exit
       )
8      (bind (result) (result . jdot . weight))))
9  (define violations
10   (not-in-order (merge-e removes (inserts . -=> . '
       reset))))
11 (define latest-violation (hold violations false))
12 (define (nv)
13   (set-running-e! (violations . -=> . false)))
14 (define (not-in-order e)
15   (filter-e
16     (match-lambda
17       [('reset _) false]
18       [(_ 'reset) false]
19       [(previous current) (> previous current)])
20     (history-e e 2)))
```

5.1.2 ImperSD によるデバッグ

ImperSD での記述はリスト 2 となる．MzTake と同様の位置である add() メソッドの 1 行目および extractMin() メソッドの最終行目に観測ポイントを設置している．2 つの観測ポイントにおける観測値の履歴が MewdFS に保持されているため，7 行目までで，変数 list1 と list2 にそれぞれ 2 つの観測ポイントにおける観測結果の履歴をリストとして取得する．11〜16 行目にて，2 つのリストを結合してタイムスタンプの順に並び替え，17〜28 行目にて，リスト中の隣り合わせの 2 要素を順番に見て違反を確認している．

5.1.3 議　　論

MzTake では，メソッドの入退出のイベントが発生するたびに，PriorityQueue の正しさに違反していないかを確認する処理までが自動的に行われる．スクリプトを記述する手間がかかるが，違反発生時に即座に停止することができる．

リスト 2: ImperSD での記述の一部 (Case1)

```
1  // 観測ポイントのセット
2  var p1 = new Probe("..."); // add()の最初の行にて
     item.weight
3  var p2 = new Probe("..."); // extractMin()の最終行
     にてitem.weight
4
5  // 観測履歴の取得
6  var list1 = p1.getTimeValsList()
7  var list2 = p2.getTimeValsList()
8  for(var e: list1){
9    e.value = "'reset";
10 }
11 list2.addAll(list1);
12 var list = list2.stream().sorted(new Comparator<
     History>(){
13   public int compare(History h1, History h2){
14     return new Long(h1.time).compareTo(new Long(h2
         .time));
15   }
16 }).collect(Collectors.toList())
17 for(int i = 0 ; i < list.size() - 1 ; i++){
18   if(list.get(i).value.equals("'reset")){
19     continue;
20   }
21   if(list.get(i+1).value.equals("'reset")){
22     i++;
23     continue;
24   }
25   if(Integer.parseInt(list.get(i).value) > Integer
       .parseInt(list.get(i+1).value)){
26     // 違反報告処理
27   }
28 }
```

ImperSD では，add() または extractMin() における要素の値を，観測結果の履歴を非常に簡単なスクリプトの記述のみで取得できている．ユーザの任意のタイミングで結果を加工して，違反していないかを自由に確認することができる．

5.2 Case2: プロファイラとしての利用

MzTake の研究 [1] では，FrTime スクリプトで簡単なプロファイラを実装している例が挙げられている．このプロファイラでは，定期的（50 ミリ秒毎）にプログラムが現在実行しているメソッド名とその呼び出し元のメソッド名を取得している．

5.2.1 MzTake によるデバッグ

FrTime が提供する変数 where を用いることで実現した記述がリスト 3 である．変数 where はコールスタックを保持するものであり，任意の一時停止イベントにて中身が書き換わる．変数 milliseconds を 50 で割った商が変化したら，一時停止イベントを発生させ，その時点でのコールスタック情報を取得することで，プロファイラの機能を実現している．

5.2.2 ImperSD によるデバッグ

ImperSD では，静的情報として対象プログラムのメソッドの一覧やクラスの一覧などを取得することができるため，対象プログラムの全メソッドに観測ポイントを仕掛け，その時点でのメソッド名を取得することが実現できる．

リスト 3: FrTime による記述 [1](Case2)

```
(define pings (make-hash-table 'equal))
((changes where)
  . ==> . (match-lambda
    [(line function context rest ...) (hash-table-
      increment! pings (list function context))]  [
      _(void)]))
(define ticks (changes (quotient milliseconds 50)))
(set-running-e! (merge-e (ticks . -=> . false)
                         (ticks . -=> . true)))
```

リスト 4: ImperSD による記述の一部 (Case2)

```
...
while(true) {
  Thread.sleep(50);
  long time = 0;
  for(var p: plist) {
    var h = p.getTimeVal();
    if(h.time > time){
      methodName = p.getFile().getParentFile().
        getParentFile().getName();
      time = h.time;
    }
  }
}
```

50 ミリ秒毎の処理については，Java 言語の標準機能である Thread.sleep() を利用することができる．

5.2.3 議　　論

定期的に実行されるような処理についてスクリプトを記述する点では，ImperSD による記述の方が簡潔になり，リアクティブ型 SD による記述内容を命令型 SD ではより簡潔に記述できること場合があることがわかった．

また，ImperSD はデバッグ操作毎にプログラムの実行を停止しないため，プログラムの実行に与える影響が小さいというメリットがある．現在の実装では一時停止やコールスタックの取得を実現していないため，MzTake のように一時停止した時点でプログラムが実行している位置の情報を取得することができない．この点は内部観測部の ProbeJ を拡張することで改善が可能である．

5.3 考　　察

2 つのシナリオで比較した結果，過去の観測結果を非同期に利用するような処理や定期的に実行されるような処理，そのほかユーザがタイミングを指定したい処理では，リアクティブ型 SD よりも命令型 SD の方が記述しやすいと考える．

一方，リアクティブ型 SD では観測対象の変化をトリガーとして処理を記述するため，観測対象の変化が即座に分かり，常に最新の値が得られるというメリットがある．ユーザがタイミングを指定しづらく，指定した状態に到達した際に即時に操作が必要なケースではリアクティブ型が有用であると考える．

6. ま と め

本研究では，既存 SD の欠点を補うような新しいスタイルの SD として，デバッガが行う動作をそのままの順序でスクリプト記述する命令型 SD を提案し，Java 言語向けの命令型 SD 環境である「ImperSD」を紹介した．実装したデバッガを用いて実際にデバッグを行い既存 SD と比較することで評価を行い，提案 SD の得失を確認した．

今後は，実際の開発におけるデバッグでの利用シナリオを考慮したうえで，被験者実験を通じて提案手法の有用性を評価する予定である．また，ImperSD の実装を改良し，適用可能な範囲を広げるほか，実用性を高める実装を行う予定である．例えば，現在の実装では，観測ポイントの設置は，原始型とフィールドに原始型を持つ複合型の変数に限定されている．この点は，ProbeJ および MewdFS の実装を拡張することによって解決を目指す予定である．

謝辞　本研究の一部は科研費（#15H02683，#18H03221）の助成を受けた．

文　　献

[1] G. Marceau, G.H. Cooper, J.P. Spiro, S. Krishnamurthi, and S.P. Reiss, “The design and implementation of a dataflow language for scriptable debugging,” Automated Software Engineering, vol.14, no.1, pp.59–86, 2007.
[2] R.A. Olsson, R.H. Crawford, and W.W. Ho, “A dataflow approach to event-based debugging,” Software: Practice and Experience, vol.21, no.2, pp.209–229, 1991.
[3] M.S. Johnson, “Dispel: A run-time debugging language,” Computer languages, vol.6, no.2, pp.79–94, 1981.
[4] M.S. Johnson, “The design and implementation of a run-time analysis and interactive debugging environment,” PhD thesis, The University of British Columbia, 1978.
[5] P. Winterbottom, “Acid: A debugger built from a language,” Proc. USENIX Winter, pp.211–222, 1994.
[6] D.R. Hanson and J.L. Korn, “A simple and extensible graphical debugger,” Proc. USENIX, pp.183–174, 1997.
[7] M. Golan and D.R. Hanson, “Duel: a very high-level debugging language,” Proc. Winter USENIX, pp.107–117, 1993.
[8] Y.P. Khoo, J.S. Foster, and M. Hicks, “Expositor: scriptable time-travel debugging with first-class traces,” Proc. ICSE, pp.352–361, 2013.
[9] T. Nelson, D. Yu, Y. Li, R. Fonseca, and S. Krishnamurthi, “Simon: Scriptable interactive monitoring for SDNs,” Proc. SOSR, p.19, 2015.
[10] R. Lencevicius, U. Hölzle, and A.K. Singh, “Dynamic query-based debugging,” Proc. ECOOP, pp.135–160, 1999.
[11] R. Lencevicius, “On-the-fly query-based debugging with examples,” Proc. AADEBUG, p.14 pages, 2000.
[12] R. Lencevicius, U. Hölzle, and A.K. Singh, “Dynamic query-based debugging of object-oriented programs,” Automated Software Engineering, vol.10, no.1, pp.39–74, 2003.
[13] C. Jeffery, W. Zhou, K. Templer, and M. Brazell, “A lightweight architecture for program execution monitoring,” Proc. PASTE, pp.67–74, 1998.
[14] M. Auguston, C. Jeffery, and S. Underwood, “A framework for automatic debugging,” Proc. ASE, pp.217–222, 2002.
[15] D. Lehmann and M. Pradel, “Feedback-directed differential testing of interactive debuggers,” Proc. FSEACM, pp.610–620 2018.
[16] M. Sulír and J. Porubän, “Runtimesearch: Ctrl+ f for a running program,” Proc. ASE, pp.388–393, 2017.
[17] 平ノ内，野田，小林，“仮想ファイルシステムを用いたプログラム内部状態観測ツールの試作，” 信学技報 SS2018-13，電子情報通信学会，July 2018.
[18] 嶋利，石尾，井上，“ソフトウェアの実行を分析するための低侵襲なモニタリングツールの試作，” ソフトウェアエンジニアリングシンポジウム予稿集，pp.224–227，2017.