アクターシステムを対象とした リバースデバッギングのためのフレームワーク

柴内 一宏　渡部 卓雄

アクターモデルは並行計算モデルの一つであり，アクターと呼ばれる計算主体間の非同期メッセージ通信により計算が行われる．アクターモデルでは原則として状態を共有しないため，データ競合やデッドロックといった単純な並行バグは回避できる．しかしながら，メッセージ到着順序の非決定性に起因するバグは依然として発生しうる．我々はこのようなバグの発見を容易にすることを目的として，アクターモデルにもとづく各種言語および実行系を対象としたデバッギングフレームワーク Actoverse を設計・実装した．Actoverse はリバースデバッギングおよびメッセージ到着順の制御機構を備えており，メッセージ到着順序の非決定性に伴うバグの発見を支援する．本稿では Actoverse の概要について述べ，Akka による例題を通してその有効性について議論する．

The Actor model is a concurrent computation model based on asynchronous communication among computational entities called actors. The inherent shared-nothing principle and the absence of locks guarantee that actor-based programs can avoid simple concurrency bugs such as data races and deadlocks. However, they are not completely free from application level concurrency flaws that occur, for example, due to the indeterminate arrival order of messages. To assist discovering such bugs in actor-based systems, we designed and implemented Actoverse, a debugger that adopts reverse debugging and provides an interactive aid for controlling the arrival order of messages upon re-execution. In this paper, we overview its architecture and discuss its effectiveness by debugging Akka-based example applications.

1　はじめに

並行システムにおけるバグは，従来の決定的なプログラムとは異なった性質を持つ．デッドロックやデータ競合といったスレッドベースのプログラムにおいて一般的な並行バグは，同一のリソースに対して複数のプロセスが関与することによって発生する．

しかし，そのようなバグは（純粋な）アクターモデルに基づくシステムでは発生しない．アクターモデルでは，アクター間で状態は共有されず，また待ち状態を保持するようなロックも持たないためである．

それでもなお，アクターモデルのプログラムにおいては，アクターモデルに特有の並行バグが発現する可能性がある．そのほとんどはメッセージの到着順序に関連している．アクターモデルにおいては，送信されたメッセージがその順序通りに受信されることは保証されていない．

現在までに，スレッドベースの並行バグに関するデバッグ手法については多くの研究が行われてきた．しかしながら，昨今のアクターモデルに基づく言語やフレームワークの発展と実用化に比較して，アクターモデルを対象としたデバッグに関する研究は相対的にごく少ないのが現状である．アクターモデルに基づく言語やフレームワークのデバッガはいくつか存在するが，上で述べたアクター特有の並行バグに焦点を当てたデバッガーは稀であることが Lopez らによって指摘されている [7]．

本研究で我々は，リバースデバッグの手法を採用し

---

* A Reverse-Debugging Framework for Actor-based Systems

Kazuhiro Shibanai, Takuo Watanabe, 東京工業大学 情報理工学院 情報工学系, Dept. of Computer Science, Tokyo Institute of Technology.

たデバッガ Actoverse[†1] を提案する．リバースデバッグとは，実行履歴を保存しておき，のちに実行中のある時点へと状態を遡るデバッグ機構である．加えて Actoverse は，アクター間のメッセージについて，そのアドレスや内容に基づいたブレークポイントを定義する機構を提供することで，非決定的なバグの再現を可能にする．

本論文では，まず研究の背景を説明したのち，提案するデバッガ Actoverse の主機能について概説する．その後，機能を実現する具体的な構成や手法についての説明を行う．最後に，デバッガのオーバーヘッドを計測するメモリ消費に関する実験について述べる．

2 背景

2.1 アクターモデル

アクターモデルは，Hewitt により提唱されその後 Agha らによって発展されてきた並行計算モデルの一つである [1]．アクターモデルでは，アクターと呼ばれる並行かつ独立に動作する計算主体が相互にメッセージを送受信することによって計算を進行させる．また，アクターは他のアクターの内部状態を直接アクセスすることはできない．またメモリや時計などを共有せず，ロックによる同期も行われない．

一般にアクターがメッセージを受信すると，以下のような振る舞いを行う．

- メッセージを他のアクターに送信する．
- 新たなアクターを生成する．
- 自分自身の振る舞い（メッセージの処理方法）を変更する．

受信したメッセージの処理は任意の順番で行われる．すなわち，メッセージの処理順に関する保証や制約は存在しない．

2.2 アクター特有の並行バグ

アクターモデルのシステムでは，単純な競合状態やデッドロックとは異なった種類の並行バグが生じうる．Lopez らはアクターモデルの特有のバグの研究において，以下のことを主張している [7]．

まず，複数のアクターが相互に待ち状態になり，暗黙的なデッドロックが発生する可能性がある．例えば，2 つのアクター A と B との間でメッセージの送受信を行うプロトコルを考える．ここでは A が B からのメッセージ x を待ち，B に応答を行うとする．しかし B が誤ったメッセージ y を送ると A は応答できず，結果としてプロトコルは進行しない．この種のバグは，アクターが実際にはブロックされずに，かつ他のアクターからのメッセージは通常通り処理することができるため，検出が困難である．

また，アクターモデルでは，アクターが独立して動作しデータを共有しないため，データ競合あるいは不可分性違反といった「競合状態」は発生しない．しかし，メッセージ配信の順序は保証されないことから，不正なメッセージやクラッシュ等の予想外のイベントの発生や，想定したプロトコルとは異なった動作をすることによって予期しない結果やシステムのクラッシュを引き起こす可能性がある．

2.3 並行プログラムにおけるデバッグ

シングルスレッドで動作するような決定的なプログラムにおけるデバッグでは，バグが発生した場合，デバッグ対象を停止して再実行し，その挙動を再現することが行われる．また，そこではプログラムを特定の時点で一時停止するためにブレークポイントが用いられる場合がある．

それに対し，並列計算や非同期 I/O 等を含む非決定的なプログラムでは，バグが常に再現されるとは限らない．アクターモデルでも同様に，システムが非同期的に動作するため，結果が常に同じであるという保証は無い．

これまでに並行計算等の非決定的なプログラムに対するデバッグ手法は多く提唱されているが，ここでは，**リバースデバッグ**と**レコードアンドリプレイ**に焦点を当てる．

リバースデバッグ（タイムトラベルデバッグ/双方向デバッグ）は，プログラムが実行される方向とは逆方向に実行ログを遡行し，過去の状態を復元するデバッグ手法である．リバースデバッグでは，プログラ

†1 Actoverse は Github で公開されている．
`https://github.com/45deg/Actoverse`

ムの実行中にバグに遭遇した際，プログラムを最初から再実行することはせずに，発生直前の状態に戻してから，バグの詳細を精査する．

リバースデバッグはいくらかのデバッガで実装されている．例えば，GDB[†2]，Microsoft IntelliTrace[†3]，TotalView ReplayEngine[†4] 等がある．リバースデバッグを実装する最も一般的な方法は，実行中の各ステップで十分な状態を保存し，過去に戻るときに復元するものである．リバースデバッグに関する詳しい実装と用途についての研究は，[2] で行われている．

他の非決定的な並行バグに対する手法としては，**レコードアンドリプレイ**が挙げられる．レコードアンドリプレイでは，命令の実行パスやプロセス間のメッセージの流れが逐次記録され，デバッグの際には保存された通りにプログラムを再実行する．リバースデバッグとは違い，プログラムを最初から再実行する必要がある．一般的に，レコードアンドリプレイはリバースデバッグに比べ実装が簡単である．

3 Actoverse の概要

Actoverse は，本研究で提案するアクターシステムを対象としたライブデバッガである．Actoverse の主な特徴として，アクターに特化したリバースデバッグ機構がある．これは，メッセージ到達時の各時点でのアクターの内部状態のスナップショットを取得することにより実現されている．

さらに，メッセージ指向のブレークポイントにより，アクターに対するメッセージの到達順序をユーザが制御することが可能となる．その「ブレークポイント」はユーザが指定したメッセージの内容に関する条件で設定され，該当するメッセージはアクターに到達する以前に保留される．リバースデバッグとブレークポイントの機能を利用することにより，並行的なアクターシステムのプログラムに対して決定的なデバッグを行うことが可能となった．

†2 https://sourceware.org/gdb/news/reversible.html

†3 https://msdn.microsoft.com/library/mt228143.aspx

†4 http://www.roguewave.com/products-services/totalview/features/reverse-debugging

その他 Actoverse には，イベントの因果性を再現し再実行を行うレコード・アンド・リプレイ機能や，メッセージログの可視化，内部状態のインスペクタ等が実装されている．以下，各機能の概要を示す．

3.1 インスペクタ

インスペクタには，デバッグ対象のシステムに存在する各アクターの内部状態が表示される．表示された値はアクターの状態が変化した場合に更新が反映される．これにより，予期しないイベントによる状態の変化に対し容易に気づくことが可能となる．

3.2 メッセージタイムライン

アクター間のメッセージの流れは，時間軸に沿ったシーケンス図として可視化される．

各アクターは垂直の線として表現され，アクター間の各メッセージは送信アクターから受信アクターへの矢印として表現される．また，内容に応じた色つけによりメッセージの種類の識別を容易にしている．

線上にある各点は，アクターがメッセージを受信した時点を表している．点にマウスカーソルを重ねると，その時点でのアクターの状態のスナップショットがポップアップで表示され，アクターの状態の変化を通時的に確認できる．

タイムラインの点と矢印は，実際の時間の尺度ではなく，因果の半順序関係を整数値で表現する Lamport タイムスタンプ [6] を基準に配置され，イベントまたはメッセージの順序を定めている．

この抽象化により，システムの開発者が明示的にグローバルな同期クロックを導入することなく，因果関係の時系列に沿ってメッセージをトレースすることが可能となる．

3.3 状態の復元機構

タイムライン上のイベントを表す点をクリックすることで，アクターがその時点での状態に復元される．復元の処理は，先述の Lamport のタイムスタンプに基づいて実行されるため，復元対象のアクターの他に，そのアクターがメッセージ送信により影響を及ぼした別のアクターに関しても状態がロールバックさ

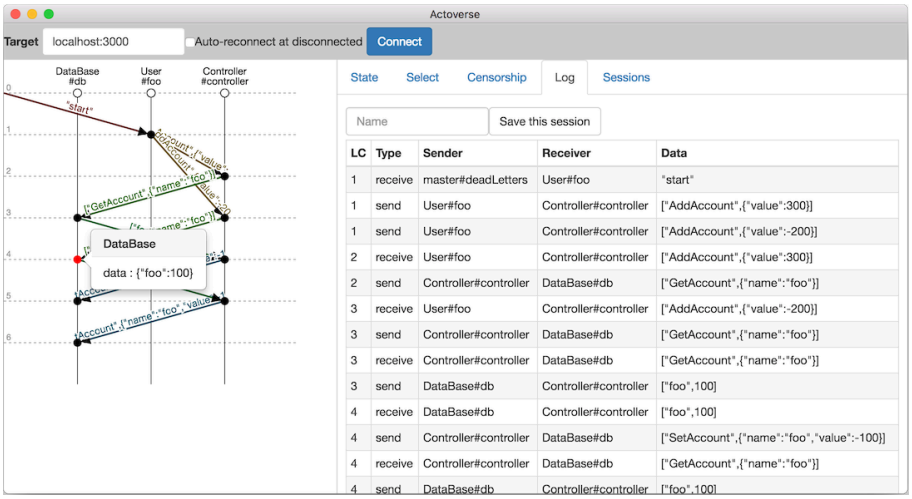

図 1　デバッガの UI 画面

れる．

復元処理が実行された後では，復元ポイント以降の記録（メッセージ，アクターの状態等）は，メモリリークを避けるために消去される．しかし，Actoverse では，メッセージの履歴の消去されない保存を可能とするセッションリプレイ機能が提供されている．これについては後の章で述べる．

3.4　メッセージ指向のブレークポイント

メッセージ指向ブレークポイントは，ユーザによって指定された条件に合致するメッセージを受信時に退避させ，処理を保留する機構である．

以下の様なメッセージ条件が指定できる．

- 送信者のアクターの名前
- 受信者のアクターの名前
- メッセージデータの部分一致及び完全一致

処理の保留はメッセージ単位で行われ，条件に合致しないメッセージは通常の方法でアクターに到着し，処理が進行される．

3.5　セッションリプレイ

ある特定のメッセージ順序でバグが発生した場合，プログラムの修正後の検証の際に，同じメッセージ順序でプログラムを再実行する必要がある．この場合，レコードアンドリプレイが有効となる．Actoverse では「セッションリプレイ」という名前で実装され，各セッションはメッセージの一連の履歴を保持している．

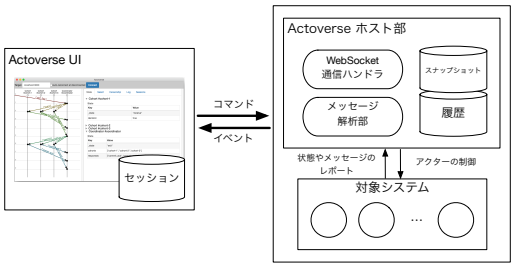

図 2　デバッガの構成

Actoverse が起動している間，セッションは永続的であり消失することは無い．そのため，プログラムが修正された後に対象システムに対し再実行・再接続された場合でも問題の状況の復元が可能となる．さらに，複数のセッションを保持することができるため，ユーザは複数の状況を再現しテストすることが可能となる．

4　実装

4.1　アーキテクチャ

Actoverse は UI 部とホスト部の 2 部で構成されている．

UI 部では，メッセージの履歴やそれぞれのアクターの現在および過去の状態の表示を行う．また，対象シ

ステムの制御パネルとしても用いられ，過去の状態の遡行や，メッセージブレークポイントの設定をこのインターフェースで行う．

ホスト部では，対象システムの内部においてアクターの監視や制御を行う．ホスト部は，アクターの状態の更新を UI に通知し，また UI から制御コマンドを受信した際に，アクターの状態の操作やメッセージの再送等を行う．

UI とホスト部の通信は，TCP 上の双方向通信プロトコルである WebSocket によって行われ，アクターやメッセージの情報や UI からのコマンドがやり取りされる．

4.2 セッションの実装

セッションは，一連の受信したメッセージのリストとして UI 部に保存される．したがって，対象システムが終了して接続が失われた場合でも，セッションは破棄されないため，対象システムの修正後もセッションの再現が可能となる．

セッションの復元の処理の概要を以下に示す．

1. UI は,「初期状態に戻す」コマンドを対象のホスト部に送信する．続けて，UI は全てのメッセージを保留するブレークポイントの設定コマンドを送信する．
2. 対象システムが初期状態に戻った後，UI はセッションに保存されたメッセージ履歴の先頭からメッセージを再送する．ホスト部は，UI から受信したメッセージに一致するものを保留されたメッセージ群から取り出し，対象のアクターに転送する．もしメッセージが保留リストにない場合，このアルゴリズムは中断される．
3. セッション内のすべてのメッセージが配信された場合，終了．

4.3 ホスト部の実装

Actoverse は，特定のアクターモデル実装に依存しない設計となっている．アクターベースの言語やフレームワークは，要求されるプロトコルを実装すれば Actoverse で利用可能となる．

本研究では，アクターシステムのフレームワークである Akka[†5] 上にリファレンス実装を行った．なお，Akka は Java と Scala で利用できるが，以降では Scala の場合における説明を行う．

4.3.1 Akka フレームワークの拡張

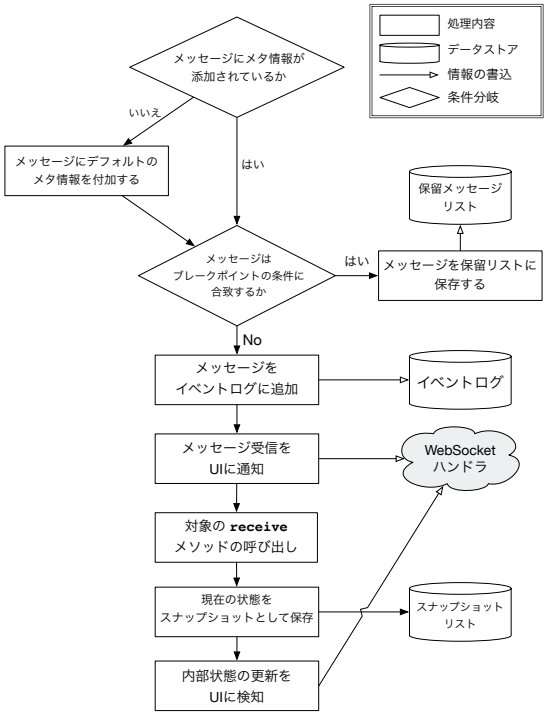

図 3　Akka 実装でのメッセージの流れ

Actoverse の機能を実装するため，Akka フレームワークに対する拡張を行った．[†6]

- メッセージの流れやイベントの因果関係を追跡するために，Lamport タイムスタンプやアクターの名前等が送信時にメッセージにメタ情報として付加される．
- アクターシステム上の送受信等のイベントは Actoverse のホスト部に通知される．また，ホスト部は UI からコマンドを受信し，対象のアクターへの操作を行う．
- 各アクターは，自身の生成時とメッセージの受

†5 http://akka.io/

†6 https://github.com/45deg/Actoverse-Scala

信時に自らの状態のスナップショットを記録する．

- アクターの内部状態は連想配列の形で保存され，過去の状態を復元する際に利用される．Akka の実装においては，明示的に可変である変数にアノテーション `@State` を付加することで保存すべき対象を指定する必要がある．
- 到着する全てのメッセージはユーザが指定したブレークポイントの条件に合致するか検査される．合致したメッセージは，アクターの処理メソッドへの受け渡しを行わず，ホスト部にある一時リストに保留される．そして，UI から「進行」のコマンドを受け取った時に，一時からメッセージが取り出され，対象のアクターのメソッドへ受け渡しが行われる．

受信されたメッセージの流れを図 3 に示す．

4.3.2 実装上の制約

現在，Akka フレームワークや Scala の実装上の制約から，以下の様なものに対してはデバッグが困難あるいは不可能である．

1. ファイル及びネットワークといった外部入出力を取り扱うプログラム．現在，そういった変更を検知および状態の保存・復元する機構は実装されていない．
2. アクターの停止及びクラッシュの取り扱い．Akka Actor のアクターのライフサイクル管理では，同一の参照 (ActorRef) を保ったままアクターを再生成することはできない．したがって，内部変数でアクターの参照を保持していた場合，状態復元の際に元のアクターの参照が失われている可能性がある．

5 関連研究

5.1 アクターモデルのためのデバッガ

アクターモデルのプログラムに特化した発展的な機能を持つデバッガがこれまで複数提唱されている [7]．Causeway [8] は E 言語のための 事後検知デバッガである．Causeway は，メッセージの因果関係を示し，バグの原因の特定が可能となっている．

REME-D [4] は AmbientTalk 言語のデバッガであり，Causeway のようなメッセージ追跡の他，メッセージ指向のブレークポイントが実装されている．あるいは，アクターモデルをベースにした Erlang や Elixir などの言語に対するデバッガにおいても，並行バグのデバッグをサポートしているものが存在する．

しかし，リバースデバッギングを実装しているアクターベースのデバッガーはこれまで存在しなかった．さらに，前述したデバッガでは実装されていない，ログのグラフィカルな可視化やセッションの保存などの機能も Actoverse では実装している．

5.2 モデル検査

モデル検査は，並行バグを検出する手法の一つであり，プログラムが仕様（論理式で書かれていることが多い）を満たしているかを論理計算により検証する形式手法を指す．

検査器は，到達可能な状態を探索し，与えられた仕様を満たさない反例があるか検査する．モデル検査は，古くから多くの研究がなされており，また産業的な応用も広く知られている．

代表的なモデル検査器として SPIN が挙げられる．SPIN [5] は，分散アプリケーション用の汎用的な検査機であり，メッセージパッシングを含むさまざまな計算モデルに対応している．McErlang [3] は，Erlang を対象としたモデル検査器である．McErlang では，状態のスナップショットを取り出せるように Erlang 言語を拡張し，到達可能な状態の探索を可能としている．

しかし，モデル検査器は「デバッガ」とは性質が異なる．モデル検査では，プログラム実行以前の段階で，ソースコード等を解析しバグを検出するような設計となっている．一方デバッガは，実際に実行されているプログラムにおいて発生するバグを検出する．

そのほか，デバッガでは起こりうる可能性のある全てのバグを検出できないが，仕様外の想定されていないバグに関しても対処が可能となる性質もある．一方モデル検査では，モデリング言語自身の知識のみならず，検証されるプログラムの仕様を把握し記述する必要がある．

6 結論と今後の課題

本研究では，アクターモデルのプログラムをデバッグするための手法を提案し，その実装を行った．アクターの状態の独立性を利用することで，リバースデバッギングをアクターベースのデバッグに適用することができ，並行バグに対し効果的であることが示された．また，メッセージ指向のブレークポイントにより，非決定性のある動作に関しても再現性を保ちながらデバッグが可能となることが示された．

さらに，Actoverse のユーザーインターフェイスにより，アクターの状態の検査や操作といったシステムとの直感的なインタラクションが可能となり，並行バグの特定と修正が可能となった．

Actoverse デバッガの対象であるサンプルプログラムは Github に公開されている †7．サンプルには，不正なメッセージの割り込みやプロトコル違反などの並行バグが含まれている．

現在このデバッガには取り組むべき課題が残されている．まずスケーラビリティの問題である．アクターモデルのシステムでは，多くのアクターが膨大な数のメッセージを送受信するような物が珍しくない．このようなシステムの場合，ユーザーインターフェースに大量のメッセージが表示されるためデバッグは困難となる．この問題に対処するためには，多くのプロセスとメッセージを可視化する手法に関するさらなる研究が必要となる．

また，Actoverse の API を実装する言語やフレームワークとそのケーススタディについてもさらなる研究が必要である．Actoverse の API を実装する際に，解決すべき実装上の問題は少なくない．例えば，アクターの内部状態を取り出し保存する手法や，過去の状態に遡る機構の実装などが挙げられる．

謝辞

本研究は JSPS 科研費 15K00089 の助成を受けている．

参 考 文 献

[1] Agha, G.: *Actors: A model of concurrent computation in distributed systems*, MIT Press, 1986.

[2] Engblom, J.: A review of reverse debugging, *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference (S4D)*, (2012), pp. 28–33.

[3] Fredlund, L.-c. and Svensson, H.: McErlang: A Model Checker for a Distributed Functional Programming Language, *SIGPLAN Not.*, Vol. 42, No. 9(2007), pp. 125–136.

[4] Gonzalez Boix, E., Noguera, C., and De Meuter, W.: Distributed Debugging for Mobile Networks, *J. Syst. Softw.*, Vol. 90(2014), pp. 76–90.

[5] Holzmann, G. J.: The Model Checker SPIN, *IEEE Trans. Softw. Eng.*, Vol. 23, No. 5(1997), pp. 279–295.

[6] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Commun. ACM*, Vol. 21, No. 7(1978), pp. 558–565.

[7] Lopez, C. T., Marr, S., and Gonzalez Boix, E.: Towards Advanced Debugging Support for Actor Languages Studying Concurrency Bugs in Actor-based Programs, *AGERE! 2016*, 2016.

[8] Stanley, T., Close, T., and Miller, M. S.: Causeway: a message-oriented distributed debugger, Technical report, HP Labs, 2009. HP Labs tech report HPL-2009-78.

†7 `https://github.com/45deg/Actoverse-Scala-Demos`