Vol.2013-ARC-205 No.11
Vol.2013-OS-125 No.11
2013/4/26

分散システムの網羅的なテストとデバッグを支援するためのフレームワークの提案

三嶽 仁[1,a)]　森田 和孝[1,b)]

概要：クラウドコンピューティングの広まりにより，分散システムの重要性が高まる一方，分散システムを効率的にテスト，デバッグすることは現在でも困難である．その理由として，複数台の計算機に跨ったプロセス群を制御し，決定的な進行を強制することが困難であるため，分散システムの実装が開発者の意図した仕様を満たしていることを示す現実的な手段が無いことが挙げられる．本研究は，この課題を解決することを目的としている．この目的を達成するため，複数の計算機に分散したプロセス全体のメモリ上での状態遷移を監視するメカニズムを提案する．これにより，分散システム全体の状態遷移を監視する実装レベルモデル検査器を実現することを目指している．

キーワード：分散システム，デバッグ支援フレームワーク

Towards debugging framework for exhaustive testing of distributed systems

Hitoshi Mitake[1,a)]　Kazutaka Morita[1,b)]

Abstract: Although distributed systems are becoming more important technologies because of the today's trend of cloud computing, effective testing and debugging targeting on them are still difficult. The main reason is a difficulty of controlling processes spread on multiple computers to make their execution deterministic. This leads to a lack of a method for checking that distributed systems implementations satisfy their specifications which implementers intended. This research is focusing on solving this problem. We propose a mechanism for monitoring global state transitions of distributed processes executed by multiple computers. With this mechanism, we are planning to establish an implementation level model checker which monitors actual global state of distributed systems.

Keywords: Distributed systems, Framework for debugging support

1. 背景

強力な計算資源を備えた計算機と高速なネットワークの低価格化により，クラウドコンピューティングの考え方に基づくサービスが多く普及し，受け入れられるようになった．その結果，それらのサービスの基盤となる分散システムの実装技術はますます重要性を増している．そういった潮流に関わらず，分散システムのテストやデバッグは，単一システムのそれと比較して依然として容易ではない．事実として，広く利用されているウェブサービスが，その基盤である分散システムのバグにより大規模な障害を起こし，ユーザーが不便を強いられるという事態は枚挙にいとまがない [3], [4].

単一システムと比較して分散システムのデバッグが更に困難である理由として，分散システムとは複数の計算機で分散して実行されるプロセスで構成されるという性質上，計算の進行に多くの非決定性を含んでいるというものが挙

[1] 情報処理学会
IPSJ, Chiyoda, Tokyo 101–0062, Japan
[†1] 現在，NTT ソフトウェアイノベーションセンタ
Presently with NTT Software Innovation Center
[a)] mitake.hitoshi@lab.ntt.co.jp
[b)] morita.kazutaka@lab.ntt.co.jp

© 2013 Information Processing Society of Japan

げられる．そのため，バグを再現させるためのコストが大きくなってしまう．

従来，モデル検査のような形式的手法が，このような非決定的な進行を含むシステムの検証に利用されてきた．しかしモデル検査を利用したアプローチは，ソースコードレベルの実装と，それをモデル化した検査器に与える形式的なモデルとの間に乖離が無いことを機械的に示すことが困難であるため，主にプロトコルレベルのデバッグに利用されることが多い．現実世界で利用される分散システムのソースコードは基本的に複雑であり，そのモデルを手動で作成することは，モデル化する対象である実装を熟知しているプログラマにとっても極めて困難な作業である．また，ソフトウェアは機能追加やバグフィックスのため高頻度で更新されるものであるため，実装の更新に合わせてモデルを更新する作業は更に現実的ではない．

本研究では，分散システムの実装を対象に検査を行うことを目的とした，実装レベルモデル検査に分類される技術を提案する．実装レベルモデル検査とは，実際に実行されているプロセスの動作をトレースしてそれらの状態を監視し，非決定性を制御することで網羅的な検査を可能にする技術である．そのため，上述の伝統的なモデル検査と比較して，モデルと実装の間に乖離の発生を許すという問題を解決することが出来るという利点を持つ．本研究が提案する技術は，従来の実装レベルモデル検査と比較して，プログラマが検査したい性質を比較的容易に記述する仕組みを備える．

2. 関連研究

2.1 単一システムを対象とした検証技術

分散システムに限らず，単一システムのデバッグも依然として容易ではない．ここで挙げる関連研究は，単一システムを対象とした検証の技術の例である．

Klein らは，定理証明支援器を利用して OS の形式的検証を行った [16]．しかし検証された OS は機能的に非常に多くの制約を持ったものであり，定理証明支援器を利用した現実的なソフトウェアの検証は未だ困難であることを示している．

単一システムのマルチスレッドのプログラムは，分散システムと同じく非決定性を持つためデバッグが困難である．そのようなマルチスレッドのプログラムのデバッグを支援するための，決定的マルチスレッディングの技術の研究も進んでおり，Kendo と Aikido がその代表例として挙げられる [23], [24]．これらの技術は，マルチスレッドのプログラムに決定的な進行を強制し，デバッグの効率を高めることに応用可能であると考えられている．

単一システムを対象とした網羅的な検査を行う技術の例として，FiSC が挙げられる [19]．この技術は，ファイルシステムの実装を対象としてモデル検査を行うことを可能にするものである．ext3，JFS，ReiserFS といった広く利用されているファイルシステムを検査し，合計で 34 のバグを検出したという実績を持つ．

2.2 分散システムを対象とした検証技術

2.2.1 実際に運用されているシステムをテストするための技術

近年では実際にサービスを提供するためにデプロイされている分散システムをテストするための技術も開発されている．Netflix 社は，Amazon Web Service の IaaS である EC2 上で実行されている VM をランダムに終了させることで，運用されているシステムの問題を早期に発見，修正するための技術として Chaos Monkey を開発した [11]．Gunawi らは，実際のサービスのために運用されている環境での，網羅的なフォールトインジェクションを行うためのフレームワークを提案している [14]．このフレームワークは，Chaos Monkey とは違い，VM の終了に限らないネットワークやディスクのエラーを再現することが可能とする．これらの技術は実装レベルモデル検査とは違い，網羅的なテストを行うことは出来ないが，小規模な問題を人為的に発生させることで大規模な障害を未然に防ぐという効果を得られると考えられている．

Liu らは，デプロイされた分散システムの大域的な状態の上に定義された，ユーザーが与えた述語を評価するための技術を提案している [9]．この技術は，実行中の分散システムがユーザの意図した仕様を満たしているか否かを検査することが可能にし，分散システムのデバッグを支援する．

2.2.2 モデル検査技術

モデル検査器を利用した形式的検証のアプローチでは，並列に動作する複数のプロセスをオートマトンとしてモデル化し，それらの直積のオートマトンとしてシステム全体を表現する．そのオートマトンの各状態についてユーザが与えた性質が成り立つかどうかを調べることで，網羅的な検査を実現する．この手法はプロセッサメーカーなどのハードウェアを製造する企業では採用が進んでいるが [17]，ソフトウェアに適用された事例は多くない．理由としては，伝統的なモデル検査の手法は，上述の定理証明支援器によるアプローチと同じく，作成したモデルが実装の性質を正しく反映していることを示すための手段を欠いているというものが挙げられる．そのため，現実的な規模のソフトウェアに適用することは未だに困難である．実際のシステムのソースコードからモデルを抽出する技術も提案されているが [18]，広く利用されるには至っていない．

そのため，検査対象のプログラムを実際に動作させ，その状態遷移を追跡することで，網羅的な検査を実現する技術も提案されている．それらは実装レベルモデル検査と呼ばれている．上述の FiSC もその一つである [19]．FiSC が単一システムであるファイルシステムを対象としているの

© 2013 Information Processing Society of Japan

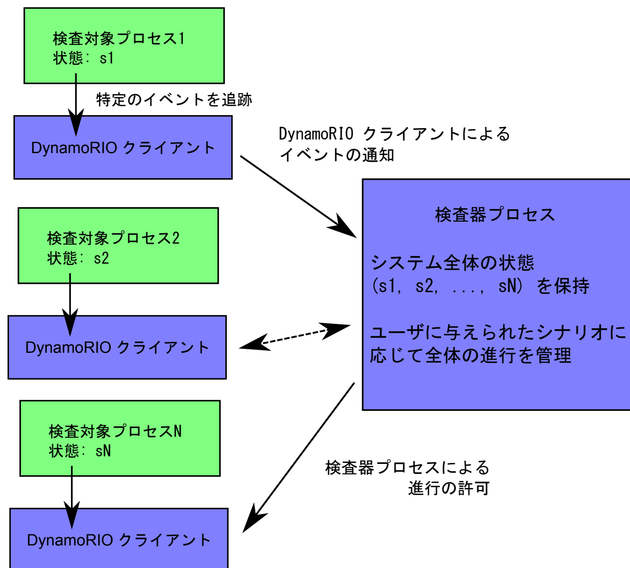

図 1　システム設計

に対し，分散システムを対象とした実装レベルモデル検査の例としては，MODIST という技術が挙げられる [1], [10]. これは，Windows 上で動作するプロセスの Win32 API の呼び出しを契機として状態遷移を監視し，網羅的な検査を実現する．MODIST はプロセスの状態を実行時に取得しそれを元に状態遷移を把握するため，モデルと実装の乖離という問題を解決している．また，著者の Yang らは，分散システムの最もテストが難しい箇所としてタイムアウトの処理部分に特に注目し，それらを効果的にテストするために記号的実行の手法を考案している．この手法は gettimeofday() などの時刻を取得する関数の呼び出し方についての経験則を利用した手法であるため，網羅性という観点からは限定された効果しか持たないが，タイムアウト処理部分の試験を効果的に行うことを可能にした．

MODIST は本研究と最も近いアプローチを採用している．しかし，MODIST は Win32 API の呼び出しのみを状態遷移の単位としているため，実際のプロセス内部の状態を直接記述することが出来ず，ネットワークの I/O の履歴などから推定される必要がある．それに対して，本研究の採用する手法は，メモリ上の状態遷移自体を検査器が把握することを可能としている．この手法により，検査対象となるシステムの実装を把握したプログラマにとっては，より自然な形で状態を記述することが可能になる．

3.　提案手法

3.1　システム設計

本研究で提案するシステムは，形式的なモデルやソースコードではなく，実際に OS 上で動作するプロセスを検査対象とする．本システムは，動作中のプロセスの状態遷移を取得するために，pointer barrierization [15] に類似する

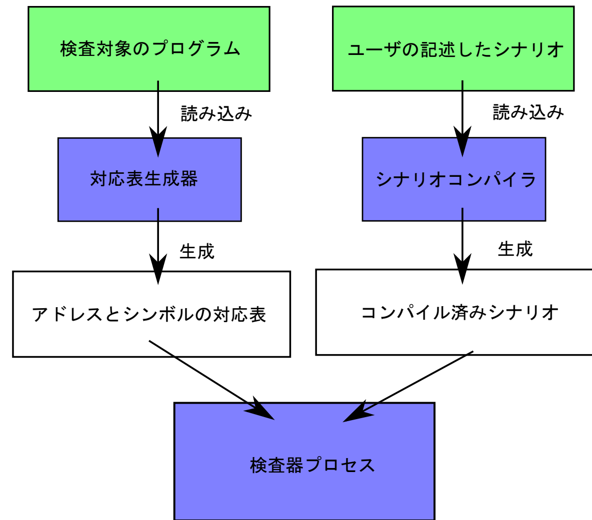

図 2　シナリオコンパイラと記号表生成器の利用方法

技術と，動的追跡ツールの DynamoRIO [5], [6][*1] を利用する．

図 1 に，システム設計を示す．メモリ上の状態の更新の検出に必要な情報を DynamoRIO クライアントが追跡し，検査器プロセスに通知する．メモリの状態の更新自体は，後述の手法で検出し，同じように検査器プロセスに通知する．DynamoRIO クライアントは，通知した更新に対する検査器からの返信を待ち，返信の内容に基いて処理を進行する．検査器プロセスは，送られてきた状態の更新と，ユーザから与えられたシナリオを元に，分散システム全体の進行を管理する．

個々のシナリオは，考えられる状態遷移全ての組み合わせの中の 1 つの要素を表現する．ユーザは検査対象にしたい状態遷移を個別のシナリオとして作成し，それぞれを検査器に与えて個別にテストを行うことで，状態空間全体の探索を行う．

3.2　シナリオの設定方法

図 2 に，シナリオの設定方法の概要を示す．記号表生成器が，検査対象のプログラムのグローバル変数のアドレスと，その型とシンボル名の情報を持ったバイナリファイルを生成する [*2]．これは実行可能ファイルに付随する DWARF の情報を元に行われる．つまり，検査対象となる

*1 DynamoRIO とは，プロセスの動作を実行時に変更するためのフレームワークである．関数やシステムコールの呼び出しの前後に処理を加えたり，実行時のコード書き換えを行う機能を提供する．スレッドの開始，終了や，シグナルの配送などをフックすることも可能である．メモリバグの検出ツールである Dr.Memory [7] の基盤として利用されている．DynamoRIO を利用した，ユーザが作成した追跡用プログラムを，DynamoRIO クライアントと呼ぶ．

*2 現実的な C で記述されたプログラムは，void 型ポインタ，共用体，構造体の末尾の長さ 0 の配列など，別プロセスに単純にコピーしただけでは意味を解釈することが難しいデータを多く含む．これらについては，そのデータの解釈に必要な周辺の情報が得られる場合にのみ，述語からアクセスされるものとする

© 2013 Information Processing Society of Japan

プログラムは，DWARF 情報を持つようビルドされる必要がある．

ユーザは，後述の言語で記述されたシナリオを用意し，シナリオコンパイラでバイナリファイルに変換する必要がある．

上の 2 つのバイナリファイルを，検査器プロセス起動時に与えることで，検査器プロセスはユーザが指定した状態遷移を検査対象の分散システムに強制する．

4. 実装

4.1 メモリ上の状態の取得

本システムでは，検査対象のプロセスのメモリ上の状態の取得を行うために，pointer barrierization [15] に類似する手法を採用する．mprotect(2) を利用して，状態を取得する必要があるメモリの書き込み権限を削除する．結果，状態の取得を行うメモリへの書き込みが発生した場合，OS により SIGSEGV が発行される．SIGSEGV ハンドラの中で，以下のような処理を行う．

(1) `if` 状態を取得する対象のメモリアドレスではない `then`
(2) 検査対象のプロセスのバグ．終了．
(3) `endif`
(4) 退避されたプログラムカウンタのアドレスにある命令をデコード
(5) デコードされた情報を元に，書き込まれた値とバイト数を取得
(6) それらの情報を検査器プロセスに対して送信し，返信を待つ．返信結果に応じた処理を行う．
(7) シグナルを利用して全てのスレッドの実行を停止し，`mprotect(2)` で該当するアドレスを含むページの書き込みを一時的に許可．書き込みを行う．
(8) 退避されたプログラムカウンタの値を，デコードされた命令の長さだけ加算する．

4.2 メモリ上の状態を取得するために必要な他のイベントの追跡

検査器プロセスは，上述のようなメモリの状態の更新の通知を受け取るにあたり，検査対象のプロセス毎に以下のような情報を取得しておく必要がある．これらのイベントは，DynamoRIO クライアントが取得し，その都度検査器プロセスに送信する．

- プロセス初期化時のデータセクションと BSS セクションの初期化
- スレッドの作成と終了
- malloc(), free() 等の標準 C ライブラリによるメモリの確保と開放

4.3 シナリオ記述言語

ユーザは以下の様な言語 [*3] でシナリオを記述する．下記の言語により，基本的なプロセスのメモリ上の状態と，典型的なエラーを表現することが可能になる．

⟨ 入力 ⟩ →
  ⟨ ステップ ⟩+

⟨ ステップ ⟩ →
  ‘(’ ‘step’ ⟨ 目的の状態 ⟩ ⟨ アクション ⟩* **‘)’**

⟨ 目的の状態 ⟩ →
  ⟨ プロセスの状態 ⟩
  ‘(’ ‘and’ ⟨ 目的の状態 ⟩ ⟨ 目的の状態 ⟩ **‘)’**
  ‘(’ ‘or’ ⟨ 目的の状態 ⟩ ⟨ 目的の状態 ⟩ **‘)’**
  ‘(’ ‘not’ ⟨ 目的の状態 ⟩ **‘)’**

⟨ プロセスの状態 ⟩ →
  ‘(’ ‘proc’ ⟨ プロセス *ID*⟩ ⟨ スレッドの状態 ⟩+ **‘)’**

⟨ プロセス *ID*⟩ →
  ⟨ 整数 ⟩

⟨ スレッドの状態 ⟩ →
  ‘(’ ‘thread’ ⟨ スレッド *ID*⟩ ⟨ 変数の状態 ⟩ **‘)’**

⟨ スレッド *ID*⟩ →
  ⟨ 整数 ⟩

⟨ 変数の状態 ⟩ →
  ‘(’ ‘and’ ⟨ 変数の状態 ⟩ ⟨ 変数の状態 ⟩ **‘)’**
  ‘(’ ‘or’ ⟨ 変数の状態 ⟩ ⟨ 変数の状態 ⟩ **‘)’**
  ‘(’ ‘eq’ ⟨ 変数の状態 ⟩ ⟨ 変数の状態 ⟩ **‘)’**
  ‘(’ ‘not’ ⟨ 変数の状態 ⟩ **‘)’**
  ‘(’ ‘add’ ⟨ 原子 ⟩ ⟨ 原子 ⟩ **‘)’**
  ‘(’ ‘deref’ ⟨ 変数の状態 ⟩ **‘)’**
  ⟨ 原子 ⟩

⟨ 原子 ⟩ →
  ⟨ リテラル ⟩
  ⟨ 変数名 ⟩

⟨ アクション ⟩ →
  ‘(’ ⟨ ディスクエラー ⟩ **‘)’**
  ‘(’ ⟨ ネットワークエラー ⟩ **‘)’**
  ‘(’ ⟨ プロセスの終了 ⟩ **‘)’**

4.4 検査器プロセスの動作

検査器プロセスは以下のようなループを実行するイベント駆動プログラムである．

[*3] この構文における”プロセス ID”とは，検査器が割り振った検査対象のプロセスに付けられる ID である．現状，本システムでは監視対象とするプロセスの個数は，検査器の起動時にパラメータとして与えるという方針を採用している．実行時に動的に増減する状況に対処することは出来ない．一方，スレッド ID は OS が割り振るカーネルスレッドの ID である．

表 2　トラップを行った場合の時間的コストの内訳

	転送無し	転送有り
物理的時間	3.925 秒	104.946 秒
ユーザ空間での消費時間	1.027 秒	2.936 秒
カーネル空間での消費時間	2.912 秒	20.889 秒

(1) 変数 `i` を 0 に設定する．設定された数の検査対象のプロセスが揃うまで待つ．
(2) 検査対象のプロセスからイベントの通知を待つ．
(3) 受信したイベントに応じて保存してある状態を更新．
(4) `if` 更新は `i` 番目の目的の状態にマッチする `then`
(5) 　目的の動作を実行．　`i <- i + 1`
(6) `endif`
(7) `if` `i` 番目のステップは存在しない `then` 終了．
(8) `goto (2)`

5. 状態のコピーによるオーバーヘッドの評価

5.1 評価の概要

現時点では本システムは実際のバグを取り除くには至っていないため，トラップした命令のエミュレーションをソフトウェアで行うことのオーバーヘッドと，メモリの状態を別のプロセスにネットワーク経由でコピーすることのオーバーヘッドを，マイクロベンチマークの結果から示す．

このベンチマークでは，検査対象のプロセス 1 つと検査器プロセス 1 つを，別々の計算機上で実行し，検査対象プロセスで連続したストア命令を実行し，そのイベントを検査器プロセスにネットワーク経由で通知する．使用した計算機のプロセッサは Intel Core2 Quad Q6600 (動作周波数 2.4 GHz)，物理メモリの容量は 2 GB である．計算期間のネットワークは 1 Gbps のイーサネットで構成されている．

評価は以下の 3 つの設定で行う．

- トラップもネットワーク経由での状態の転送も行わない．
- トラップは行うが，ネットワーク経由での状態の転送は行わない．
- トラップもネットワーク経由での状態の転送も行う．

1 番目の設定では，ストア命令を 10 億回行う．2 番目，3 番目の設定では，ストア命令は 100 万回行う．1 番目はオーバーヘッドが無く，1 回のストア命令のコストを測定するために 100 万回のストアを行うだけでは十分な時間が経過しないため，10 億回のストアを行う必要がある．

5.2 評価結果

表 1 は，ストア命令 1 回のコストにかかる，物理的な時間のコストを示している．また，表 2 は，トラップを行った場合の 2 つのベンチマークにおいて，ベンチマークに要した物理的な時間のうち，ユーザ空間で消費された時間とカーネル空間で消費された時間の内訳を示している．物理的な時間からユーザ空間とカーネル空間の時間の消費を引いた時間は，ネットワークの入力待ちに使われた時間となる．

5.3 考察

表 1 から，単純にトラップを行うだけでも 1 回のストアのコストが 1000 倍以上に増加することが読み取れる．また，トラップにネットワーク経由での転送のコストが加わると，25 倍以上のコストがかかる．つまり，一回のメモリの状態遷移をネットワーク経由で別のプロセスに転送すると，約 25000 倍のコストがかかってしまう．

一方，表 2 からは，状態遷移の転送のコストの大部分がシステムコールの実行とネットワークの入出力の待ち時間に占められていることが読み取れる．このことから，状態遷移の転送の回数を削減することが全体のコストを削減することに貢献する可能性が高い．詳細は将来課題として後述する．

6. 将来課題

6.1 検査対象とする状態の網羅性を向上させるための技術

現状，本システムではシナリオの網羅性を向上させるための支援機能を持っていない．そのため，網羅的な試験を行うためにはユーザがシナリオ全てを個別に記述する必要がある．網羅的な試験をより効果的に行うために，PROMELA [2] のような状態遷移機械を記述するための言語を定め，それを翻訳し上述のシナリオを生成するためのコンパイラを用意する必要がある．

6.2 検査対象とする状態の拡大とオーバーヘッドの削減

現状，本システムはデータセクション，BSS セクション，ヒープ領域として確保されたメモリの状態のみを監視対象としている．しかし，検査対象とする分散システムの採用するアルゴリズムにとっての重要な情報が，スタック上に保存されているケースは少なくない．そのような分散システムを修正無しで検査対象とするためには，スタック上の情報を効率的に取得する必要がある．

また，関数呼び出しのコールスタックの状態を，目的の状態として記述可能にすることはデバッグの支援に役立つと考えられる．特に，コールスタックの状態を正規表現でマッチすることが可能になれば，より粒度の細かい状態の指定が可能になる．

また， 5.3 章で述べた通り，状態遷移を転送するコストの大部分はネットワークの入出力に占められている．そのため，目的の状態への到達の判定を DynamoRIO クライアント内部で行い，到達時にのみ検査器に通知を行えば，検査に要する時間を短縮出来る可能性が高い．

© 2013 Information Processing Society of Japan

表 1　ストア命令 1 回にかかる物理的な時間のコスト

	トラップ，転送無し	トラップ有り，転送無し	トラップ，転送有り
ストア命令 1 回のコスト	3.35 ナノ秒	3.925 マイクロ秒	104.95 マイクロ秒

6.3　システム VMM を利用したフォールトインジェクション

本研究の将来課題の 1 つに，Xen [20] や KVM [21] などのシステム VMM を利用したフォールトインジェクションのメカニズムの実装が挙げられる．これらのシステム VMM に下記のフォールトインジェクションや時間の加速を実現するメカニズムと，検査器から操作するインターフェイスを実装することで，より広範囲の試験を実現出来ると考えられる．

6.3.1　ディスク，ネットワークのエラーの忠実な再現

前述の通り，従来の実装レベルモデル検査は，Win32 API や入出力を伴う Java 標準 API の呼び出しをフォールトインジェクションを行う単位としていた [1]．しかし，このような方法では，ディスクやネットワークのエラーの忠実な再現が困難である．例えば，ディスクの故障を再現するために特定のプロセスのファイルディスクリプタのみをクローズすることは十分ではない．ログの書き出しが別のプロセスによって行われていた場合，エラーの発生を示すログさえディスクに書き出すことが出来ない，という状況を再現させることが出来ないからである．

また，分散システムが対応すべき重要な問題に，ネットワークの分断が挙げられる．スプリットブレインを招くようなノード間のスイッチやルータの故障は，個々のプロセスのソケットのファイルディスクリプタに対する操作を失敗させるだけでは表現することが難しい．

システム VMM を用いて，仮想デバイスドライバや仮想スイッチのレベルでフォールトインジェクションを行えば，より自然なハードウェアの故障を再現することが可能となり，分散システムの耐障害性の評価としてより適した試験を実現出来る．

6.3.2　タイムアウト処理のテスト

分散システムにおいては，異なるノードとの通信を行なっている際，タイムアウトが発生した状況を処理するためのコードをテストすることが特に困難である [1]．Yang らは，gettimeofday() などの時刻を取得する API の呼び出しをベースとした経験則に基づいた記号的実行により，そのようなコードのテストのカバレッジを向上させる手法を提案している．

しかし，このような手法では経験則に頼っているという性質上，網羅出来る範囲が限定されてしまう．加えて，setsockopt() でタイムアウトを指定し，同期的な読み書きを行なっているソケットのファイルディスクリプタについては，タイムアウトを再現させることは出来ない．poll() や epoll() のタイムアウトの機能を利用している入出力についても同様に再現が不可能である．

システム VMM を利用すれば，タイマ割り込みの間隔を縮めることで，VM 上のゲスト OS の時間を加速させることが可能になる [22]．これにより，より網羅的かつ忠実なタイムアウト処理の試験が実現出来る．

7.　結論

本論文では，実装レベルモデル検査に分類される新しい分散システムを対象とした検証技術の基盤となるメカニズムを提案した．このメカニズムを利用すれば，従来の実装レベルモデル検査技術とは異なる，検査対象となるプロセスの変数の状態の上に定義される述語を記述することが出来るモデル検査器を実現出来ると考えられる．そのようなモデル検査器は，検査のコストを減らし，その正確性を向上させることを可能とする．

クラウドコンピューティングと称されるサービスを提供する企業にとって，基盤となる分散システムのバグはその企業活動にとって致命的な問題と成り得る．それらのバグを効果的に検出，除去するために，実装レベルモデル検査は有効な技術となると期待されている．

参考文献

[1] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In proceedings of the 6th USENIX symposium on Networked Systems Design and Implementation (NSDI), 2009.
[2] Gerard J. Holzmann. The Model Checker SPIN. In journal of IEEE transaction on Software Engineering, Vol. 23, No. 5, May 1997.
[3] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/jp/message/65648/
[4] Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region. https://aws.amazon.com/jp/message/680587/
[5] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE), 2012.
[6] DynamoRIO: Dynamic Instrumentation Tool Platform. http://www.dynamorio.org/
[7] Dr.Memory. http://www.drmemory.org/
[8] Ranjit Jhala, and Rupak Majumdar. Software model checking. In journal of ACM Computing Survey, 2009 October.
[9] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^3S: debugging deployed distributed systems. In proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation

© 2013 Information Processing Society of Japan

(NSDI), 2008.
[10] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), 2011.
[11] The Netflix Tech Blog: Chaos Monkey Released Into The Wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html
[12] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: a programmable tool for multiple-failure injection. In proceedings of the 2011 ACM international conference on Object Oriented Programming Systems Languages and Applications (OOPSLA).
[13] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: a framework for cloud recovery testing. In proceedings of the 8th USENIX conference on Networked Systems Design and Implementation (NSDI), 2011.
[14] Haryadi S. Gunawi, Thanh Do, Joseph M. Hellerstein, Ion Stoica, Dhruba Borthakur, and Jesse Robbins. Failure as a Service (FaaS): A Cloud Service for Large-Scale, Online Failure Drills. The University of Chicago, Technical Report, 2011.
[15] Rei Odaira and Toshio Nakatani. Continuous object access profiling and optimizations to overcome the memory wall and bloat. In proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.
[16] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP), 2009.
[17] Geoff Barrett. Model checking in practice: the T9000 virtual channel processor. In journal of IEEE Transactions on Software Engineering, Vol. 21, No. 2, February 1995.
[18] Gerard J. Holzmann. From Code to Models. In proceedings of the 2nd international conference on Applications of Concurrency to System Design (ACSD), 2001.
[19] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In proceedings of Operating System Design and Implementation (OSDI) 2004.
[20] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), 2003.
[21] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In proceedings of the Ottawa Linux Symposium (OLS), 2007.
[22] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey Voelker. M. DieCast: Testing Distributed Systems with an Accurate Scale Model. In journal of ACM Transaction on Computer Systems (TOCS), Volume 29, Number 2, May, 2011.
[23] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In proceedings of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2009.
[24] Marek Olszewski, Qin Zhao, David Koh, Jason Ansel, and Saman Amarasinghe. Aikido: accelerating shared data dynamic analyses. In proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

© 2013 Information Processing Society of Japan