Scalaの並行プログラム検査における状態空間探索手法

島田 工[1,a)]　前澤 悠太[1,b)]　鄭 顕志[2,c)]　田辺 良則[2,d)]　本位田 真一[1,2,e)]

概要：アクターモデルは，広く利用されている並行プログラミングモデルの 1 つであり，各 Actor の独立性に着目することで効率的に検証を行う手法が提案されてきている．しかし，Scala 言語による現実世界のプログラムでは，しばしば純粋なアクターモデルは用いられず，他の並行モデルが併用される．この場合には既存手法は適用できない．本研究ではアクターモデルと併用されることの多い Future 等が存在する場合でも，検証を可能にする手法を提案する．実験結果は，既存手法では検出が難しかった不具合が提案手法で発見できることを示しており，より多くの並行プログラムに対して不具合発見を支援できるようになったと考えられる．

キーワード：Scala，アクターモデル，並行プログラミング，ソフトウェア検証

A state space exploration technique for verifying concurrent programs in Scala

Abstract: Actor model, one of widely used concurrent programming models, has efficient verification methods focusing on the independence of each actor. However, Scala programs in the real world often use actor model together with other concurrency models instead of the pure actor model. Such applications cannot leverage existent methods. In this research, we propose a verification method that can apply to programs which use actor model and Future, which is often used together with actor model. From our experiment results, our method could detect bugs in the subject programs which are difficult for the existent methods to verify and we consider this method can help bug detection in wider range of concurrent programs.

Keywords: Scala, actor model, concurrent programming, software verification

1. はじめに

より高い性能を達成するために，並行プログラムはますます重要になっている．しかし，並行プログラムでは，デッドロックやデータレースのような特有のバグがあり，これらはスケジューリングの非決定性により再現や発見が困難なこともある．

Scala[1] はオブジェクト指向と関数型言語を組み合わせた Java virtual machine（JVM）上で動作する言語として，広く用いられている [2]．Scala では並行プログラムは標準ライブラリから Actor を使い，アクターモデルでの実装が可能である．このアクターモデル [3] は並行プログラミングモデルの 1 つで，独立し並行動作する Actor と呼ばれるオブジェクトがメッセージを非同期で送り合うことで協調し処理を進めていくモデルである．各 Actor が独立して動作し，共有メモリや同期機構を持たないため，デッドロックやデータレースを防ぐことができる．しかし，Scala の並行プログラムでは純粋なアクターモデルで実装されていない場合が多くある [4]．純粋なアクターモデルよりも他の並行モデルを使ったほうが簡単に実装できることがあるためである．

そこで本論文は，Actor と Future を使った並行プログラムに着目した．これは，Tasharofi ら [4] が Actor と Future の併用が多いことを明らかにしたからである．さらに，synchronized メソッドによるブロッキングや一定の制限下

[1] 東京大学
The University of Tokyo
[2] 国立情報学研究所
National Institute of Informatics
[a)] shimada@sakamura-lab.org
[b)] maezawa@nii.ac.jp
[c)] tei@nii.ac.jp
[d)] y-tanabe@nii.ac.jp
[e)] honiden@nii.ac.jp

での共有メモリの存在も考慮する．これらの状態遷移をActor のメッセージ通信による状態遷移に変換した．変換したプログラム上で共有変数が条件下でどのように影響するかを考慮した結果，既存の検証手法を用いることができることがわかった．また，既存ツールを用いた実験も行い，今まで扱えなかったバグを発見できることを確認した．これにより，既存の状態探索によるバグ検証手法がより広い範囲での適用が可能になる．

2. 背景

2.1 アクターモデル

アクターモデルとは，並行プログラミングモデルのひとつである．このモデルでは，Actor と呼ばれる独立し並行動作するプロセッサが互いにメッセージを送り合い，そのメッセージを処理することで動作を進めていく．それぞれの Actor の実行は非同期で行なわれるが，Actor 内での処理はシーケンシャルに行なわれる．また，Actor 間の通信はメッセージのやり取りのみによってのみ行なわれ，共有メモリは持たない．Actor に対して送られてきたメッセージは Actor の持つメールボックスにためられ，Actor はメッセージをランダムに 1 つ選びこれを処理することができる．このアクターモデルは Scala のみならず，様々な言語やライブラリによって実装するための機能が提供されている [5], [6], [7].

このアクターモデルは共有メモリが存在せず，ブロッキング命令もないため，デッドロックやデータレースのようなバグを防ぐことが出来る．しかし，メッセージがメールボックスに溜まっているにも関わらず，すべての Actor がメッセージを処理できない状況に陥ることがある．これをアクターモデルでのデッドロックと呼ぶことにする．また，メッセージを処理する順番によってプログラムの実行結果が期待したものと異なることが起こりうる．これをアクターモデルでのデータレースと呼ぶことにする．そのため，アクターモデルでもバグ検証のための手段が必要になる．すでにアクターモデルのプログラムの状態を効率的に探索し，バグを発見するツールはいくつか存在する [8], [9].

2.2 半順序簡約（POR）

並行プログラムは実行順序に関し不確定性を持つため，実行ごとにその結果が異なることがある．そのため，並行プログラムを検証するために，考えうる実行順序をすべて試す手法が用いられる．しかし，実行順序のパターン数は指数関数的に増加するため，すべて試すことは難しい．探索する実行順序のパターン数を減らす方法として半順序簡約（Partial Order Reduction, POR）[10] が知られている．

POR とはプログラムの実行による状態遷移間に成り立つ関係を利用し，不要な探索を避ける手法である．この手法は Java Path Finder（JPF）[11] や SPIN[12] のような様々な検証ツールで用いられている．POR は通常，遷移の関係を静的に解析するが，動的に依存関係を解析する DPOR[13] という手法も存在する．アクターモデルでは状態遷移の競合関係が推移律を満たすことを利用して，DPOR の改良版である TransDPOR[14] を利用することができる．

3. 基本アイデア

本論文では，Actor と Future が用いられるプログラムを対象とする．Future による状態遷移を Actor によるメッセージ通信による状態遷移に変換することで，アクターモデルと同様の状態空間探索を実現する．また，探索空間削減手法として，TransDPOR の適用も行う．TransDPOR では状態遷移の依存関係を判定する方法が必要となる．また，状態遷移の競合関係が推移律を満たす必要もある．これらに関して，対象とするプログラムでどうなるかを検討する．

3.1 対象とするプログラム

本論文で検証の対象とする Scala プログラムは以下のような条件を満たすものとする．

(1) Actor・Future を用いた並行処理が存在するが，java.lang.Thread のような他の並行処理は存在しない
(2) Actor のメッセージ待ち，Future によるブロッキング，synchronized メソッドによるブロッキングは存在するが，wait/notify のような他の同期処理は存在しない
(3) 複数の Actor・Future からアクセスされる変数は，存在してもよいが，その変数を上書きする処理が存在する時，同時にその変数に対して書き込み・読み込みは実行不可能である．
(4) ある Actor があるメッセージを処理できるかどうかは Actor のローカルな状態のみにより，共有変数には依存しない
(5) プログラムには無限ループを含まず，必ず終了するかデッドロックする

３つ目の条件について補足をする．ある変数に対して同時に動いている複数のスレッドから読み込みが行なわれることは許される．スレッドのうちひとつでも書き込みをしているのであれば，同時に存在するスレッドは読み込みも書き込みもすることはできない．共有変数に書き込みを行うときはかならず synchronized メソッドのような同期機構を用いたりすることで，他のスレッドが読み込みや書き込みを行うことがないようにしなければならない．

5 つ目の条件は TransDPOR を用いたプログラム検証を行うために必要な条件である [14].

4. 変換の手法

Baker ら [15] によると，Future は *process*，*cell*，*queue*

からなるタプルと定義される．process は Future が評価すべき式で，その結果は cell に書き込まれる．queue は Future の結果を待っているプロセスの集合である．Future の process は非同期で開始される．評価が終わるとその結果が cell に書きこまれ，process は終了する．Future の結果を使うプロセスは，もしまだ cell に結果がないのであれば，queue に登録され，そのプロセスは休止する．queue に登録されているプロセスは Future の処理が終わるといっせいに再開される．cell に結果が書きこまれているのであれば，それを読み込み処理を続ける．

我々は Future を以下のように変換することで，各種状態と状態遷移をアクターモデルと同様のものにする．

- Future を 1 つの Actor とする
- Future を変換した Actor は生成と同時にスタートする
- Future の結果を使うという処理は Future を変換した Actor に対してメッセージを送り，その返信を待つという処理に変換する
- Future を変換した Actor はまず，評価すべき式の評価を行い，その評価が終了すると，メッセージを待つ状態になる．メッセージを受け取ると，結果を返信する．

この変換は process を Actor のスタート時に行う処理に変え，cell を Actor 内のローカル変数として，queue を Actor のもつ mailbox にそれぞれ対応させている．cell へのアクセスや queue にたまっているプロセスの再起動はメッセージのやりとりによって実現される．これにより，変換後も実行結果が保たれているとわかる．

この変換において Future を変換した Actor の終了するタイミングは特に考えない．このようにしてしまうと，実際のプログラムでは Actor がすべて終了しないと全体の実行が終了したことにならないので，実際のプログラムと挙動が異なってしまう．しかし，デッドロック判定やデータレース判定において，Actor が終了状態になったかどうかは問題にはならない．よって，この問題は検証において無視する．

4.1 Future モデルの変換

```
val f = future{
  p() // () => T
}

Await.result(f,Duration.Inf)
```

図 1 変換前のコード

変換の手法についてソースコードを用いて具体的に説明する．図 1 のコードを考える．これを今回の手順で変換すると，図 2 のようにできる．この変換を応用することで

```
case object Wait
trait FutureActor[A] extends Actor {
  start()
  def process(): A
  def act() = {
    val result = process()
    loop { react {
      case Wait =>
        sender ! result
    }}
  }
}

val f = new FutureActor[T]{
  def process() = { p() }
}

(f !? Wait).asInstaceOf[T]
```

図 2 変換後のコード

scala.concurrent.Future に含まれるメソッドで，onComplete や andThen などのコールバックを登録するメソッドや ready や result といったブロッキングをつくるメソッドも変換できることを確認した．

4.2 synchronized メソッドの変換

synchronized メソッドは排他処理を実現するメソッドであるが，本来アクターモデルにはないブロッキングを生むことになり，デッドロックの原因になる可能性がある．

そこで，オブジェクトをつくる時，それに対応するロック用の Actor をつくることでアクターモデルに組み込む．ロック用の Actor はつくられると同時に開始され，以下のフリー状態とロック状態の 2 つの状態をとる．

フリー状態 メッセージが来たら，実行許可のためのメッセージを返信し，自身はロック状態となる状態．

ロック状態 実行完了のメッセージを受信したら，フリー状態に戻る状態．他の種類のメッセージは処理しない．

synchronized メソッドに渡された関数の処理に入る際，まずそのオブジェクトに対応するロックアクターにロック用のメッセージを送信し，実行許可のメッセージが来るまで停止する．実行許可のメッセージが来たら，synchronized メソッドに渡された関数の処理を実行し，処理が完了したらロック用のアクターに実行完了のメッセージを送信する．これにより，synchronized メソッドと同じ性質を保つことが出来る．

具体的にロック用の Actor を実装すると図 3 のようになる．

```
case object Lock
case object Free
case object LockOK
class SyncActor extends Actor {
  start()
  def act() = { loop { react {
    case Lock =>
      reply(LockOK)
        react {
          case Free =>
        }
  }}}
}
```

図 3　ロック用のアクター

5. 探索手法

5.1 定義の変更

アクターモデルと同じ探索手法を用いるが，共有メモリが存在するため，用語の定義に変更が必要である．ここでは，Tasharofi ら [14] と同じ手順で，変更が必要な箇所を修正していく．まず，*configuration* は各アクターとそのローカル状態の対応 $\alpha : \mathbb{A} \to \mathbb{L}$（$\mathbb{A}$ はアクターの集合，$\mathbb{L}$ はアクターのローカル状態の集合），グローバル状態 $\eta \in \mathbb{G}$（$\mathbb{G}$ はグローバル状態の集合）と溜まっているメッセージの集合 $\mu \subseteq \mathbb{M}$（$\mathbb{M}$ はプログラム上存在する全てのメッセージの集合）から成り立つ．つまり $\kappa = \langle \alpha, \mu, \eta \rangle$ となる．プログラム上存在する全ての configuration の集合を $\mathbb{K}$ とする．また，アクターは自身のローカル状態によって一部のメッセージを処理しないでおくことができる．これを制約とよび，アクター a に対し，その制約 $c_a \subseteq \mathbb{L} \times \mathbb{M}$ をアクターのローカル状態とメッセージ上での述語とし，処理できるのであれば真となる．この上で状態遷移を定義し直す．

定義 5.1　メッセージ m に対して，状態遷移 t_m は部分関数 $t_m : \mathbb{K} \rightharpoonup \mathbb{K}$ である．与えられた $\langle \alpha, \mu, \eta \rangle \in \mathbb{K}$ に対し，m のレシーバーを a，その状態を s，制約を c_a とする．状態遷移 t_m は $t_m(\langle \alpha, \mu, \eta \rangle)$ が定義されていて $\langle s, m \rangle \in c_a$ ならば実行可能である．もし，t_m が実行可能であれば，実行することで，新たな *configuration* をつくる．このとき，ローカル状態が s から s' になり，グローバル状態が η から η' になり，新たに送られるメッセージの集合を $out_{s,\eta}(t_m)$ とし，新たにつくられるアクターの集合を $new_{s,\eta}(t_m)$ とする．すなわち：

$$\langle \alpha, \mu, \eta \rangle \xrightarrow{t_m} \langle \alpha[a \mapsto s'] \cup new_{s,\eta}(t_m), \mu \backslash \{m\} \cup out_{s,\eta}(t_m), \eta' \rangle \quad (1)$$

対象とするプログラムはマクロステップスケジューラ [16] による実行とみなすことが出来る．これは，あるアクターがメッセージを処理し，次のメッセージを待つ状態になるまでの 1 つの状態遷移がアトミックに実行され，スケジューラは次の処理すべきメッセージをランダムに決定し，またその状態遷移を実行するというものである．実際の Scala のスケジューラであれば，1 つの状態遷移を行なっている間に他のアクターの状態遷移を実行することが可能ではあるが，対象プログラムでは節 3.1 での 3 番目の条件により，同時実行可能な状態遷移のうち，相互に作用するものがないため，アトミックな実行とみなしても実行結果は同じである．証明は Agha ら [16] と同じようにできるが，省略する．

5.2 依存関係を用いた探索空間の削減

並行プログラムの状態探索では，状態遷移の依存関係と競合関係を判定することで半順序簡約を行うことで，探索すべき領域を減らすことが出来る．アクターモデルの場合，状態遷移間の依存関係が推移律を満たすことを利用した TransDPOR[14] を利用することが出来る．

本論文の対象プログラムでも，アクターモデルと同じように TransDPOR を用いることができる．TransDPOR を用いるためには状態遷移が依存していないための十分条件と，競合していないための十分条件が必要である．これは TransDPOR において，依存関係・競合関係の擬陽性は問題にならないためである．また，競合関係については推移律を満たす必要がある．まず，状態遷移の依存関係の定義を確認する．以下，依存していない状態のことを独立と呼ぶ．

定義 5.2　状態遷移 t_1，t_2 が独立であるとは以下の条件がプログラムが到達しうる任意の *configuration* κ において成り立つことをいう．

(1) t_1 が κ において実行可能で，$\kappa \xrightarrow{t_1} \kappa'$ ならば，t_2 は κ でも κ' でも実行可能性は変化しない．

(2) t_1，t_2 ともに κ で実行可能であれば，唯一の κ' が存在し，$\kappa \xrightarrow{t_1,t_2} \kappa'$ と $\kappa \xrightarrow{t_2,t_1} \kappa'$ が成り立つ．

アクターモデルのプログラムであれば，以下が成り立ち [14] 本論文の対象プログラムでも同じことが成り立つ．

定理 5.1　*2* つの状態遷移 t_1, t_2 が独立であるためには以下の全ての条件を満たせば十分である*:*

(1) $actor(t_1) \neq actor(t_2)$*;*

(2) $msg(t_1) \notin out(t_2)$ *and* $msg(t_2) \notin out(t_1)$*;*

(3) $actor(t_1) \notin new(t_2)$ *and* $actor(t_2) \notin new(t_1)$*.*

このことが，本論文の対象プログラムでも成り立つことの証明の簡易版を与える．

証明（スケッチ）1　アクターモデルと異なる点は共有変

数の存在のみなので，共有変数がどのような影響を与えるかを考える．すなわち，*Actor* モデルのプログラムにおいて定理 *5.1* の独立であるために十分条件を満たしているにも関わらず，*2* つの状態遷移が依存しているという場合を考える．状態遷移が依存するとは定義 *5.2* での条件のうち少なくとも一方を満たさないということである．

まず，定義 *5.2* の（*1*）の条件を満たさないとすると，状態遷移 t_2 の実行可能性が t_1 の実行により変化するということである．しかし，実行可能性は今回の対象プログラムの場合，グローバルな状態には依らない．つまり，共有変数の存在によりこの条件が満たされないということはない．

定義 *5.2* の（*2*）の条件を満たさない場合を考える．つまり，*2* つの状態遷移の実行順番を変えてしまうと実行結果が変わるということである．実行結果はメッセージとローカル状態とグローバル状態によってのみ決定されるが，メッセージはすでに発行されているため変わることはなく，ローカル状態も *2* つの状態遷移でのメッセージのレシーバーが違うという仮定より，これも変化することはない．よって，グローバル状態がどちらか片方の遷移で変化し，その影響でもう一方の遷移の結果が変わるということになる．しかし，これは片方の遷移である共有変数に対し書き込みを行い，もう一方の遷移で同じ共有変数に読み込みか書き込みを行うということを意味する．この *2* つの状態遷移は κ において同時に実行可能であるので，今回の検証対象のプログラムの節 *3.1* での仮定の *3* 番目の条件に反する．よって，*2* つ目の条件を満たさないという場合は存在しない．以上より，今回のプログラムでもアクターモデルのプログラムと同じように状態遷移の依存関係を判定できる．（証明終わり）

これにより競合関係の判定もアクターモデルと同じように出来る．2 つの状態遷移 t_1 と t_2 が競合しているとは，この 2 つの遷移が依存関係にありかつ 2 つとも実行可能な状態にあることを意味する．今回の対象プログラムについて，ある状態において実行可能な状態遷移 $\langle t_1, t_2 \rangle$ が競合関係にあるとき，かならず $actor(t_1) = actor(t_2)$ を満たす．依存関係にあるためには定理 5.1 の 1 つ目の条件を満たすが，2 つ目・3 つ目の条件を満たさない場合も考えられるが，t_1 も t_2 もこの時点で実行可能であるという仮定より，この場合，2 つ目・3 つ目の条件は常に満たすことがわかる．また，競合関係が推移律を満たすこともわかる．これにより，TransDPOR を使うことができるとわかる．

TransDPOR の健全性，つまり TransDPOR を用いた探索で用いなかった場合に探索しないシーケンスを探索することがないこと，はアルゴリズムから自明である．TransDPOR の完全性，つまり TransDPOR を用いても本来探索すべき状態を探索しないということがないこと，については Tasharofi ら [17] の証明と同様にできる．詳細は省略する．

6. 評価

いくつかの節 3.1 の条件を満たす簡単なプログラムを用意し，それらを手動で Actor のみを用いたプログラムに変換した後，Scala で実装されたアクターモデルのプログラムを検証するためのツールである Basset[8] を利用することで検証を試みた．

`dataproc` はバグが含まれていないプログラムで，共有変数である配列に複数の Future によるスレッドから書き込まれるプログラムである．各スレッドは synchronized メソッドを使うことでデータレースを防いでいる．`deadlock` はデッドロックするパターンを含むプログラムである．Actor と Future を使用していて，synchronized メソッドによるロックを取得するタイミングによってデッドロックする場合とそうでない場合が存在する．`datarace` はデータレースを含むプログラムで，synchronized メソッドによるロックを取得する順番によって最終的な状態が変化してしまう．

それぞれのプログラムについて Basset で検証を行い，たどった状態遷移列のパターン数，状態遷移そのものの数，かかった時間について調べた．バグがあるプログラムに関してはバグを発見してもそのまま探索をおこない，すべての状態を探索した．実験は 1199MHz Intel Core(TM)i5 の Ubuntu12.04 で OpenJDK の JVM 1.7.0_51 で行った．結果は表 1 にまとめた．変換後のプログラムの行数（LOC）も記載した．

表 1 探索した状態遷移の数

Subject	LOC	traces	transitions	time[sec]
dataproc	110	459	1852	10
deadlock	103	108	390	3
datarace	120	27	130	2

すべてのプログラムについて，現実的な時間で検証が終了している．また，バグのあるプログラムについてはバグを検出できたことを確認した．これ以外のベンチマークも 1 つ用意したのだが，本来さほど状態数をもたないはずのプログラムであるのにも関わらず，Basset が大量の状態を探索し，検証が終わらなかった．この原因はわかっていないため，この実験については再考する必要がある．

7. 関連研究

Scala は JVM 上で動作するため，JPF[11] を使うことで，今回の対象のプログラムも検証することができる．しかし，JPF は Java のバイトコードレベルでの検証となってしまうため，アクターのライブラリの中身まで探索してしまい，検証に膨大な時間がかかってしまう．Basset[8] は JPF の拡張として実装されていて，アクターモデルの性質とアクターライブラリの情報を用いることで，効率的に

アクターモデルで実装されたプログラムを検証することができる．その他にも，アクターモデルに着目して検証するツールは存在するが [9], [18]，我々の知る限りではどれも純粋なアクターモデルを前提としている．

しかし，Tasharofi ら [4] は純粋なアクターモデルを用いていないプログラムが多く存在することを指摘している．彼らはこのような混ざったモデルを用いたプログラムの作者にインタビューをおこない，その原因を聞いた．プログラマーの経験不足やライブラリの実行効率なども原因であったが，そもそも，純粋なアクターモデルでは実装が難しくなる場合があることも原因であった．

8. 結論と今後の課題

我々は Scala のアクターと Future を用いた並行プログラムでの効率的な検証手法を提案した．すべての状態遷移をアクターモデルのでのメッセージ通信に変換し，共有変数が適切に保護されているという前提のもとでなら，従来手法が用いることができ，それを従来ツール上で確認した．

しかし，本研究のツール上での実験は対象プログラムの規模も数も少なく，また検証が上手くいかない場合があることも確認している．また，Actor への変換に関しては手作業で行っている．節 3.1 での共有変数に関する仮定が適切かどうかも検討しなければならない．つまり，現実のプログラムではこの仮定を意図的に守っていない，あるいは守ろうとして失敗しているというものが多いかもしれない．今回、探索空間を削減する手法として TransDPOR を用いたが，他の手法についての検討も行うべきであろう．

そのため，今後の課題としてすべての作業を自動化した上で，対象プログラムが正確に検証できるツールを実装する必要がある．そのツールを用い，より規模の大きいプログラムで実験を行い，バグを正しく発見できるか，検証は現実的な時間で終わるか，他の手法は有効か，といったことを確かめることも求められる．共有変数に関しては，守ろうとして失敗しているケースについてはこの仮定が守られているかどうかをチェックする機能をつけることで，検出できるのではないかと考えている．

参考文献

[1] Dean Wampler, A. P.: *Programming Scala*, O'Reilly Media (2009).
[2] Odersky, M. and Rompf, T.: Unifying Functional and Object-oriented Programming with Scala, *Commun. ACM*, Vol. 57, No. 4, pp. 76–86 (2014).
[3] Agha, G. A.: ACTORS: A Model of Concurrent Computation in Distributed Systems (1985).
[4] Tasharofi, S., Dinges, P. and Johnson, R. E.: Why Do Scala Developers Mix the Actor Model with other Concurrency Models?, *ECOOP 2013 —Object-Oriented Programming* (Castagna, G., ed.), Lecture Notes in Computer Science, No. 7920, Springer Berlin Heidelberg, pp. 302–326 (2013).
[5] Armstrong, J.: Erlang, *Commun. ACM*, Vol. 53, No. 9, p. 68—75 (2010).
[6] Dekorte, S.: Io Programming Guide, (online), available from ⟨http://iolanguage.org/scm/io/docs/IoGuide.html⟩ (accessed 2014-01-27).
[7] Typesafe Inc: *Akka Scala Documentation* (2013).
[8] Lauterburg, S., Karmani, R. K., Marinov, D. and Agha, G.: Basset: a tool for systematic testing of actor programs, *SIGSOFT FSE* (Roman, G.-C. and Sullivan, K. J., eds.), ACM, pp. 363–364 (2010).
[9] Sen, K. and Agha, G.: Automated Systematic Testing of Open Distributed Programs, *Fundamental Approaches to Software Engineering* (Baresi, L. and Heckel, R., eds.), Lecture Notes in Computer Science, No. 3922, Springer Berlin Heidelberg, pp. 339–356 (2006).
[10] Godefroid, P.: Using partial orders to improve automatic verification methods, *Computer-Aided Verification* (Clarke, E. M. and Kurshan, R. P., eds.), Lecture Notes in Computer Science, No. 531, Springer Berlin Heidelberg (1991).
[11] Visser, W., Havelund, K., Brat, G. and Park, S.: Model checking programs, *The Fifteenth IEEE International Conference on Automated Software Engineering, 2000. Proceedings ASE 2000*, pp. 3–11 (2000).
[12] Holzmann, G.: The model checker SPIN, *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279–295 (1997).
[13] Flanagan, C. and Godefroid, P.: Dynamic Partial-order Reduction for Model Checking Software, *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, New York, NY, USA, ACM, p. 110—121 (2005).
[14] Tasharofi, S., Karmani, R. K., Lauterburg, S., Legay, A., Marinov, D. and Agha, G.: TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs, *Formal Techniques for Distributed Systems* (Giese, H. and Rosu, G., eds.), Lecture Notes in Computer Science, No. 7273, Springer Berlin Heidelberg, pp. 219–234 (2012).
[15] Baker,Jr., H. C. and Hewitt, C.: The Incremental Garbage Collection of Processes, *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, New York, NY, USA, ACM, p. 55—59 (1977).
[16] Agha, G., Mason, I. A., Smith, S. F. and Talcott, C. L.: A Foundation for Actor Computation, *Journal of Functional Programming*, Vol. 7, p. 1—72 (1998).
[17] Tasharofi, S.: Efficient testing of actor programs with non-deterministic behavior, *University of Illinois at Urbana—Champaign, Tech. Rep* (2013).
[18] Sirjani, M. and Jaghoori, M. M.: Ten Years of Analyzing Actors: Rebeca Experience, *Formal Modeling: Actors, Open Systems, Biological Systems* (Agha, G., Danvy, O. and Meseguer, J., eds.), Lecture Notes in Computer Science, No. 7000, Springer Berlin Heidelberg, pp. 20–56 (2011).