MPIアプリケーションの振る舞い駆動開発

繁谷 悠[1]　桑原 寛明[2]　國枝 義敏[2]

概要：テスト駆動開発では本体コードの前にテストコードを記述するが，最近はソフトウェアの振る舞いを表現する自然言語と階層構造を含めてテストコードを記述する振る舞い駆動開発 (Behavior Driven Development) が注目されている．並列処理を含むソフトウェアでは，並列に動作する各プロセスが通信や同期によって協調して動作するため，ソフトウェアモジュール同士の関係性が重要となる．そのようなソフトウェアの振る舞い駆動開発には，各プロセスの振る舞いを記述できるテストフレームワークが必要である．本稿では，MPI(Message Passing Interface) を使用した並列ソフトウェアの振る舞いが記述可能なテストフレームワークを提案する．さらに，開発者がテストを行う環境を容易に構築できるように，仮想化技術を用いたテスト環境の構築とテストの実施，および本番環境の構築とアプリケーションのデプロイの自動化手法を提案する．

1. はじめに

ソフトウェアの開発においてプログラムの正常な動作を保証するためにテストコードを用いた開発が日々行われている．

テストコードを用いた開発のひとつとしてテスト駆動開発 (TDD : Test Driven Development) [1] が挙げられる．これはソフトウェアのモジュールが正しく動作していることを保証するテストコードを先に記述し，テストコードに書かれたテストが成功するように本体のコードを記述していく開発手法である．本体のコードには必ずテストコードが付属するため保守性の高いソフトウェアをつくることができる．しかし TDD で用いられるテストフレームワークはソフトウェアの各モジュールに対してテストが書かれるため，テストコードの可読性が低いことや要求仕様と関係のないテストコードが記述され開発コストが高くなることが多い．このような TDD 向けのテストフレームワークは逐次処理を対象とするものだけでなく，並行処理を対象とするテストフレームワーク [2] [3] や，並列処理の際に広く用いられる MPI(Message Passing Interface) [4] 向けの MPIUnit [5] といったテストフレームワークも存在する．

近年では TDD から派生した振る舞い駆動開発 (BDD : Behavior Driven Development) [6] と呼ばれる手法が取られることが多い．BDD はテストコードを要求仕様に近い形で記述し，ソフトウェアの振る舞いに注目しテスト記述の抽象度を高めることで要求仕様と関係のないテストを削減する．テストを説明する自然言語と共にテストコードを記述することができ，それがテスト結果と共に出力されるためテスト内容を把握しやすくなる．

並列処理を行うアプリケーションの TDD や BDD において詳細なテスト行うには従来のテストフレームワークでは不十分である．並列処理では複数のプロセスが同時に動作しているため，それぞれのプロセスがどのように動作しているかを記述できなければならない．通信や同期処理によってプロセス同士の関係が複雑になるためそれらのモジュールをテストするテストコードやテスト結果の可読性が低くなる．よって並列処理の複雑なプロセス間の関係やテスト内容を自然言語と階層構造を用いて表現できる BDD 向けのテストフレームワークが必要である．

本稿では，MPI アプリケーションにおいて BDD を行うためのフレームワークを提案する．MPI アプリケーションではそれぞれのプロセスを独立してテストするだけでは不十分である．複数のプロセスを協調動作させたときに相対的な実行順序が非決定的であるために起こるバグが存在する可能性があるため，提案手法のテストフレームワークはこれらのバグを発見しデバッグしやすい機構を提供する．これによってユーザはよりプロセス単体の振る舞いテストに注力することができる．さらにユーザがこれらのテストを行う環境を容易に手に入れられるように軽量な仮想化技術であるコンテナを用いたテスト環境の自動構築とアプリケーションの本番環境へのデプロイの自動化を提案する．

[1] 立命館大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Ritsumeikan University
[2] 立命館大学情報理工学部
College of Information Science and Engineering, Ritsumeikan University

2. 振る舞い駆動開発

振る舞い駆動開発はテスト駆動開発から派生したソフトウェア開発手法のひとつである．テスト駆動開発はプログラマがソフトウェアの保守性を上げるためにテストコードを記述していくが，振る舞い駆動開発は顧客が要求する自然言語の仕様を交えて階層構造を持つテストコードを記述することでプログラムと仕様の乖離を小さくし顧客に対する品質保証をする．テスト駆動開発開発の流れについて説明し，その後振る舞い駆動開発を例を用いて説明する．

2.1 テスト駆動開発

テスト駆動開発はまずソフトウェアのテストを先に記述し，そのテストが成功するように本体のプログラムを実装する開発手法のことである．C 言語でテスト駆動開発を行うための有名なテストフレームワークとして CUnit [7], CppUnit [8] などが挙げられる．

2.1.1 Red, Green, Refactor

テスト駆動開発ではプログラミングの工程が Red, Green, Refactor と呼ばれる 3 つのフェイズに分けられる．3 つの工程を機能ごとに繰り返し行うことで信頼性，保守性の高いソフトウェアをつくることができる．

2.1.1.1 Red

Red フェイズではユニットテストのテストコードを記述する．その後テストを実行しテストが失敗することを確認する．

2.1.1.2 Green

Green フェイズでは Red フェイズで記述したテストコードが通るように本体のコードを記述した後テストを実行し，テストが通るまで本体コードを改変していく．

2.1.1.3 Refactor

Refactor フェイズでは本体コードの保守性を上げる作業を行う．重複するコードの削除やプログラムの抽象化をテストが成功した状態を維持しながら行う．

2.1.2 テストフレームワーク

テスト駆動開発では主に 3 種類のファイルを使用する．本体コードを記述したファイル，テストコードを記述したファイル，テストコードを呼び出すモジュールを記述したメインとなるファイルである．テストフレームワークにはプログラムの挙動をテストするためのモジュールが定義されており，ユーザはそれらをテストコードを記述するファイルで呼び出し本体コードをテストする．メインファイルではテストコードに記述されたどのテストを実行するかを定義しメインファイルを実行することで任意のテストを実行することができる構造である．メイン関数を定義する必要のないプログラミング言語ではこのメインファイルが存在しないテストフレームワークが多い．

```
1 def fizzbuzz(num)
2   return 'FizzBuzz' if (num % 15).zero?
3   return 'Fizz' if (num % 3).zero?
4   return 'Buzz' if (num % 5).zero?
5   num.to_s
6 end
```

図 1 fizzbuzz.rb
Fig. 1 fizzbuzz.rb

```
1  describe 'fizzbuzz' do
2    context '入力された数字が' do
3      it '3で割り切れる場合はFizzを返す' do
4        expect(fizzbuzz(3)).to eq 'Fizz'
5      end
6
7      it '5で割り切れる場合はBuzzを返す' do
8        expect(fizzbuzz(5)).to eq 'Buzz'
9      end
10
11     it '15で割り切れる場合はFizzBuzzを返す' do
12       expect(fizzbuzz(15)).to eq 'FizzBuzz'
13     end
14
15     it '3,5で割り切れない場合は数字を返す' do
16       expect(fizzbuzz(1)).to eq '1'
17     end
18   end
19 end
```

図 2 fizzbuzz_rspec.rb
Fig. 2 fizzbuzz_rspec.rb

```
1 FizzBuzz
2   入力された数字が
3     3で割り切れる場合はFizzを返す
4     5で割り切れる場合はBuzzを返す
5     15で割り切れる場合はFizzBuzzを返す
6     3でも5でも割り切れない場合は入力の数字を返す
```

図 3 RSpec によるテスト結果
Fig. 3 Test results with RSpec

2.2 振る舞い駆動開発の例

振る舞い駆動開発の例を BDD フレームワークの RSpec を用いて説明する．Ruby で記述した FizzBuzz プログラムを図 1 に示し，RSpec によるテストコードの例を図 2 に示す．図 2 の expect は fizzbuzz メソッドの返り値をテストしている．describe，context が階層構造をつくるための記述であり 1 行目の describe が fizzbuzz メソッドのテスト階層であることを示し，2 行目に context によって入力が数値だった場合のテストを記述する階層を定義している．it はスコープ内で行っているテストの説明を自然言語で与える．図 3 に RSpec によるテスト結果を示す．

図 3 では図 2 で書いた自然言語によるテストの説明がそのまま出力されている．テストコードの階層構造もその

まま出力に反映されるため，プログラマはテストごとの関係を読み取ることができる．テストの成否は出力の文字色によって表現され緑なら成功，赤なら失敗を表す．これらの特徴からより可読性の高いテストコードを記述することができ，プログラムの挙動を示したドキュメントをテスト結果から得られることがわかる．テスト駆動開発と比べてユーザは自然言語のソフトウェアの仕様からテストコードの記述を直感的に行うことができるためソフトウェアの実装と仕様の乖離を小さくすることができる．

3. MPI

MPI [4] はクラスタによる並列プログラムを書くための標準ライブラリインターフェースである．C 言語または FORTRAN77 から呼び出せ移植性の良さから，広範な支持を受けている．MPI の実装として MPICH2 [9] や OpenMPI [10] がある．

MPI プログラムが実行されるとユーザから指定された数のプロセスが生成されノードに配置される．配置されたプロセスを識別するための番号をランクと呼ぶ．各プロセスは SPMD (Single Program, Multiple Data streams) 型並列処理の考え方に則って同じプログラムを実行するが，処理フローをランクによって制御することで，各プロセスは異なる処理を実行できる．

3.1 MPI アプリケーション開発における問題

MPI アプリケーションの開発では並列処理に起因する問題が生じるためそれらの説明を行う．

3.1.1 潜在的なバグ

並列計算ではそれぞれの計算プロセス同士の相対的な実行順序が非決定的であることに起因するバグが埋め込まれることがある．これを潜在的なバグと呼ぶ．MPI アプリケーションでもこのようなバグが発生する可能性があり，その原因のひとつに非同期通信後の同期の欠如が挙げられる [11]．MPI アプリケーションの開発を行う際はこのような潜在的なバグに注意しプログラミングをする必要がある．

MPI では非同期通信によって受信側のバッファは参照の前に同期を行うことで決定的な値が保証される．同期処理が欠如した場合，受信側のバッファが送信されたデータで更新されている場合と更新されていない場合が生まれ受信側のバッファが非決定的になる．図 4 に MPI アプリケーションの例を示す．17 行目で non_deterministic 関数を呼び出しランク 0 からランク 1 へ値 2 を送信しているが同期処理である MPI_Wait が欠如しているため 17 行目で関数 non_deterministic が実行された後の buf の値は非決定的となる．10000 回実行した時の変数 buf の値のばらつきを表 1 に示す．同期の欠如による潜在的バグを防止するためには非同期通信に対応する同期処理が欠如していることを警告する仕組みが必要である．

```
1  int non_deterministic() {
2      ...
3      if (rank == 0) {
4          buf = 2;
5          MPI_Isend(&buf, 1, MPI_INT, 1, 0,
6                  MPI_COMM_WORLD, &request);
7      }
8      if (rank == 1) {
9          MPI_Irecv(&buf, 1, MPI_INT, 0, 0,
10                 MPI_COMM_WORLD, &request);
11     }
12     /* MPI_Wait(&request, &status); */
13     return buf;
14 }
15 int main(int argc, char *argv[]) {
16     MPI_Init(&argc, &argv);
17     int buf = non_deterministic();
18     ...
19 }
```

図 4　非同期通信後の同期の欠如で潜在的なバグが起きる例
Fig. 4　Bugs with lack of the synchronization

表 1　非同期通信後の同期の欠如で潜在的なバグが起きる例の実行結果
Table 1　The result of bugs with lack of the synchronization

buf の値	0	2
出現回数	882	9118

3.1.2 ドキュメント

MPI アプリケーションは複数のプロセスが並列に動作しそれぞれのプロセスが通信や同期によって協調する．ドキュメントを作成する場合はアプリケーションで動作させるプロセス分のドキュメントが必要になるため並列処理を行わないソフトウェアよりコストが高くなる．近年はドキュメント作成のコストを下げるためにソフトウェアのテストコードに自然言語のドキュメントを併記し出力する振る舞い駆動開発が行われている．しかし既存の振る舞い駆動開発向けのテストフレームワークを MPI アプリケーションのテストで用いるとそれぞれのプロセスが並列して結果を出力するためドキュメントが崩れて表示されてしまう問題がある．よって標準出力を用いた既存のテストフレームワークで MPI アプリケーションの振る舞い駆動開発を行うことは困難である．MPI アプリケーションの開発で振る舞い駆動開発を行うには並列性を考慮したテストフレームワークが必要である．

3.1.3 実行環境の仮想化

仮想化が主流になる前のソフトウェア開発ではテスト環境として本番環境と同じ構成の実機環境を用意し各開発者はローカル環境で作成したプログラムをその環境でテストしていた．これにより開発者間のテスト環境を統一し開発者の個々のテスト環境に依存するバグが埋め込まれることを防ぐことができた．この手法は 1 台のマシンで動作する

ようなプログラムではコストも低く抑えることができるため有用であった．しかしクラスタのような複数台で動作するプログラムのテスト環境を実機で用意することはコストが増大し容易ではない．近年では仮想化技術が発達しテスト環境には仮想環境が用いられることが多くなり企業はテスト環境を構築するためのコストを大幅に削減することができるようになった．しかし仮想環境を用いてクラスタを構築することで実機を用意するコストを削減することはできるが，その仮想クラスタ上で MPI アプリケーションを動作させる環境をつくる手間は以前と変わらず複数台の仮想マシンの起動，ソフトウェアのインストール，ネットワーク構築を手動で行う必要がある．このような手間がかかることで容易に仮想環境を作り直すことができないため時間が経つにつれ開発者間の仮想環境のソフトウェア構成に乖離が生じる．よって仮想環境を容易に構築，破棄，再構築することができ，常に新しいテスト環境でテストすることができるシステムは開発者のテスト環境に依存するバグを防ぐことができるため有用性が高く MPI アプリケーション開発を行うには必要である．

4. MPISpec

MPI アプリケーションで振る舞い駆動開発を行うためのテストフレームワークである MPISpec を提案する．MPI アプリケーションでは 3.1.1 節で示した潜在的なバグが生じる可能性があるため，MPISpec ではこれらのバグを発見，デバッグしやすい機構を提供する．

4.1 仕様

MPI アプリケーション向けの振る舞い駆動開発テストフレームワークとして以下の仕様を満たす必要がある．

(1) ランクごとに分かれたテストが記述できる．
(2) ランクごとに分かれたテスト結果が得られる．
(3) 自然言語と階層構造を用いてテストを記述できる．
(4) 潜在的的なバグを発見しやすい機能を持つ．
(5) 標準出力，XML，JUnit [12] 形式のので出力できる．

MPI アプリケーションは並列処理であるためテストコードも並列で動作する必要があり，詳細にテストコードを記述できるように特定のランクでしか動作しないテストを定義できる必要がある．MPI では実行毎にランク同士の相対的な実行順序がかわるため，ランクが各々にテスト結果を出力していると毎回テスト結果の順序が変わり見辛くなってしまうので出力結果がランクの実行順序に依存しない仕組みが必要となる．自然言語と階層構造によるテストは RSpec における describe，context，it と同等の機能を実装することで可能となるが，MPI アプリケーションは複数のランクで動作するため describe で定義した関数のテスト階層内でランク毎のテストが記述できなければならない．潜在的なバグである非同期通信の同期処理の欠如によるバグ

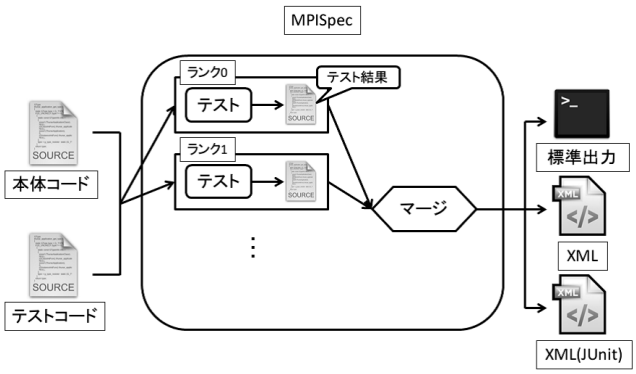

図 5 MPISpec の処理の流れ
Fig. 5 The flow in MPISpec

```
1 char *rank_to_s() {
2     static char rank_s[5];
3     if (rank_s[0] != '\0') return rank_s;
4     int rank;
5     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6     sprintf(rank_s, "%d", rank);
7     return rank_s;
8 }
```

図 6 rank_to_s.c
Fig. 6 rank_to_s.c

を検知するために，非同期通信を同期処理時に開始するようにテストフレームワークで書き換える．これによって同期処理が欠如している場合は通信が発生しないことでデータを受信できないためその部分のテストが必ず失敗する．このように同期処理が欠如している場合にテストが失敗する仕組みで潜在的なバグが埋め込まれることを防止する．

4.2 MPISpec の処理

MPISpec の処理の流れを図 5 に示す．MPISpec は本体プログラムとテストコードを入力としてもち，テストコードに記述されたテストをランクごとに実行しテスト結果を一度ファイルに出力する．出力されたテスト結果をランク順にソートし最後にマージすることでどのような順序でテスト結果が出力されたとしても必ず同じ順序のテスト結果が得られる．

4.3 入力

入力形式を MPI アプリケーションの自分のランク番号を文字列にして返す関数のテストを例に示す．図 6 に本体コードを示す．

MPI アプリケーションを MPISpec を用いてテストするには，本体コードをテストするためのテストコードを記述したファイルとそれらを実行するための main 関数が定義されたメインファイルが必要である．

4.3.1 テストコード

テストコードを記述するファイルでは以下の記述ができ

```
1  MPISPEC_DEF(rank_to_spec)
2      DESCRIBE("function rank_to_s")
3          RANK(0)
4              CONTEXT("rank 0")
5                  IT("return '0'")
6                      SHOULD_MATCH(rank_to_s(), "0")
7                  END
8              END
9          END
10         RANK(1)
11             CONTEXT("rank 1")
12                 IT("return '1'")
13                     SHOULD_MATCH(rank_to_s(), "1")
14                 END
15             END
16         END
17     END
18 END
```

図 7 rank_to_s_spec.c
Fig. 7 rank_to_s_spec.c

```
1  MPISPEC_INIT
2
3      MPISpec_VerboseRun(rank_to_spec)
4
5  MPISPEC_FINALIZE
```

図 8 main_spec.c
Fig. 8 main_spec.c

る必要がある．
(1) メインファイルから呼び出すための階層の定義と自然言語での説明．
(2) 関数のテストを記述する階層の定義と自然言語での説明．
(3) 関数内のランクごとのテストを記述する階層の定義と自然言語での説明．
(4) ランク以外で分ける階層の記述．
(5) モジュールのテストを自然言語で説明するための記述．
(6) モジュールをテストするための関数．

振る舞い駆動開発を行うために自然言語と階層構造を用いてテストを記述することができなければならない．MPI アプリケーションで各モジュールはランク毎に動作が異なることが多いため各ランク毎のテストを記述できる階層を定義できる必要がある．図 7 に図 6 で示した関数 rank_to_s について，ランク 0 では返り値が文字列の 0，ランク 1 では返り値が文字列の 1 であることをテストするテストコードを記述したファイルを示す．

図 7 の 1 行目でメインファイルで呼び出すテストコードの階層を定義する．2 行目から 17 行目は関数 rank_to_s のテストの階層を定義している．3 行目の RANK は 3 行目から 9 行目までを引数のランク 0 でのみ実行することができランクごとにテストを分けて記述できる．4 行目の CONTEXT は DESCRTBE の別名だが可読性を上げるため，関数の階層を定義する場合のみ DESCRIBE を用い，それ以外で階層を分けたい場合は CONTEXT を用いる．5 行目の IT は 6 行目に定義されているテストの説明を自然言語で記述している．6 行目で実際にテストを実行しており SHOULD_MATCH は第 1 引数と第 2 引数の文字列が等価であるかをテストする．このような関数をマッチャと呼ぶ．MPISpec では 10 種類のマッチャを定義している．

4.3.2 メインファイル

テストコードを記述するファイルでは以下の記述ができる必要がある．
(1) MPI の初期化処理を行う記述．
(2) MPI の終了処理を行う記述．
(3) テストコードを実行する記述．
(4) テストコードの出力形式を定義した記述．

MPI アプリケーションでは初期化処理と終了処理が必要なためメインファイルで定義する．テストコードで定義されたテストを呼び出すための記述と出力形式が複数あるならどの出力形式で出力するかを定義できなければならない．図 8 にテストコードを呼び出すメインファイルを示す．

図 8 の 1 行目では MPISpec の初期化処理を，5 行目では終了処理を行っている．3 行目で図 7 の 1 行目で定義したテストを呼び出している．この関数をランナと呼ぶ．ランナを変えることで出力の形式を変えることができる．

4.4 出力

出力は 3 種類から選ぶことができ，ターミナルへの標準出力，XML 形式での出力，Java 用のテストフレームワークである JUnit が出力するものと同じフォーマットの XML 形式がある．

4.4.1 標準出力

MPISpec の標準出力によるテスト結果を 4.3 節で示したプログラムを用いて図 9 に示す．1 行目から 9 行目がランク 0 のテスト結果，11 から 19 行目がランク 1 のテスト結果，21 から 24 行目が全てのランクでテストが成功している割合とテストの実行速度である．図 9 の 2 行目から 6 行目と図 7 の 2 行目から 9 行目を比較すると，テストコードの DESCRIBE，CONTEXT，IT に記述した自然言語のテストの説明がそのまま出力に現れていることがわかる．またテストコードでネストすると出力も同じようにネストして出力されている．テストの説明を自然言語で書き，テスト間の関係を階層構造によって表現することで可読性の高いテスト結果を得ることができる．

4.4.2 XML 出力

XML はプログラムによる解析が容易であるためテスト結果を加工して表示したい場合などに有用である．

```
rank  0:
  function rank_to_s

    rank 0
      - return '0'
        OK: strcmp(rank_to_s(), "0") == 0

--Run Summary: Type      Total  Passed  Failed
               tests         1       1       0

rank  1:
  function rank_to_s

    rank 1
      - return '1'
        OK: strcmp(rank_to_s(), "1") == 0

--Run Summary: Type      Total  Passed  Failed
               tests         1       1       0

[2 Process Results]
============================================[100%]

Run Time: 0.007091 sec
```

図 9　MPISpec の標準出力
Fig. 9　The standerd output with MPISpec

4.4.3　JUnit 形式の XML 出力

JUnit 形式に対応することで JUnit 形式の XML に対応したツールを用いた開発が可能になる．

4.5　潜在的なバグを検知するための機構

MPISpec では MPI アプリケーションで潜在的なバグが埋め込まれるのを防止するために通信をスタブすることで問題を解決する．スタブはあるプログラムの呼び出しの際に別のプログラムを間に差し込み任意の挙動を再現するプログラミング手法である．MPISpec は MPI アプリケーションの非同期通信処理をスタブする．スタブされた非同期通信処理では，テストコードに記述された任意のランク間の非同期通信について送受信するデータをテストコードで宣言した値で上書きし，通信を実際に行うのではなく予約する．予約された通信を非同期通信の同期処理中で行うようにすることで，非同期通信の同期処理が欠如している場合は通信が発生しないためスタブした値のテストが必ず失敗する状態をつくり，同期処理の欠如によって潜在的なバグが埋め込まれることを防止する．

故意に非同期通信後の同期処理を抜きエンバグしたプログラムで MPISpec と既存手法の MPIUnit で潜在的なバグの検知の評価を行った．表 2 に結果を示す．MPISpec は同期処理がないプログラムのテストでは必ず失敗し，既存手法より潜在的なバグの検知率が高いことがわかる．

表 2　非同期通信後の同期の欠如による潜在的なバグの検知率
Table 2　The ratio of detecting bugs with lack of the synchronization

通常実行	MPIUnit	MPISpec
6.2%	0.12%	100%

5.　cluster-packer

この章では MPI アプリケーションを動かすための仮想環境，本番環境をコンテナや Chef [13] を用いて自動構築することができる cluster-packer を提案する．

5.1　仕様

3.1.3 節で述べたように MPI アプリケーションを実行可能なクラスタ環境を作ることは容易ではない．よって cluster-packer では以下のような仕様で問題を解決する．

(1) VM(Virtual Machine) よりもオーバヘッドの小さい仮想環境．
(2) ソフトウェアインストールとネットワーク構築の自動化．
(3) クラスタを 1 つのコンピュータとみなし一元的に管理できるコマンド．

開発者個人のソフトウェア環境に依存するバグを埋め込まないためにもテスト環境は開発者間で共通であることが望ましい．しかしクラスタ環境を VM で構築する場合複数台のノードが必要なことや起動や停止のオーバヘッドが大きいこと，手作業による設定の必要であることによってカジュアルに環境を破棄することができない．容易に再構築できないソフトウェア環境を開発者間で共通に保つことは困難である．よって VM よりもオーバヘッドの小さい仮想化技術を用いてクラスタを構築し，各ノードのソフトウェア環境とネットワークの構築を自動化して容易に再構築することができる必要がある．容易に再構築できるテスト環境が得られれば常に新しい環境でテストができるため，各開発者のテスト環境に依存するバグが発生することを防ぐことができる．クラスタのそれぞれのノードに対してコマンドを送ることは非効率であるため，クラスタを 1 つのコンピュータとみなし複数のノードの構築，停止，ソフトウェアのデプロイと実行をそれぞれの 1 つのコマンドで行えることが必要である．

5.2　cluster-packer の構成

クラスタの台数などを設定ファイルに記述しコマンドを実行するとクラスタの自動構築，ログイン，破棄の操作を行う．

5.2.1　設定ファイル

クラスタを操作するために設定ファイルでは以下の記述が必要である．

(1) 仮想環境と本番環境の区別.
(2) 仮想クラスタのノードの台数.
(3) 本番環境のセットアップを行うユーザ.
(4) 本番環境のファイルサーバの IP アドレス.
(5) 本番環境の MPI ノードの IP アドレス.

仮想環境と本番環境の設定は衝突する可能性があるため分けて記述できる必要がある．仮想クラスタは cluster-packer がノードを立ち上げるため IP アドレスの情報は必要なくユーザが必要とする仮想クラスタのノード数のみを定義する必要がある．本番環境での環境構築はパスワード入力が必要でないユーザ上で実行する必要があるためセットアップ用のユーザを指定する必要がある．MPI アプリケーションをクラスタ上で実行するにはプログラムのバイナリを共有するファイルサーバと MPI アプリケーションを実行するノードが必要で，それぞれにログインするためにユーザが用意した実機の IP アドレスかホスト名が必要である．

5.2.2 コマンド

cluster-packer で実行可能なコマンドと説明を表 3 に示す．ユーザは MPI アプリケーションを実行するためのインフラの構築，停止，アプリケーションのデプロイ，実行をすべてコマンドのみで完結させることができ，アプリケーションの構築に注力することができる．クラスタは cluster-packer が管理しているため複数のノードへの操作を一括して行うことができ各ノードへそれぞれ操作を行う必要がない．

表 3 cluster-packer で実行可能なコマンド

コマンド	説明
cluster init	クラスタ用のイメージの作成などの初期化処理.
cluster up	クラスタの起動，ネットワーク構築.
cluster halt	クラスタの停止.
cluster ssh	クラスタへのログイン.
cluster deploy	クラスタへアプリケーションのデプロイと実行.

5.3 cluster-packer による MPI アプリケーション開発の流れ

以下に cluster-packer を用いたクラスタの自動構築からアプリケーション実行までの流れを示す.

(1) クラスタ用のイメージの作成.
(2) コンテナによる仮想クラスタの構築.
(3) 仮想クラスタへのデプロイ，テスト.
(4) 本番環境の構築，デプロイ.

5.3.1 クラスタ用のイメージの作成

MPI アプリケーションの実行に必要なソフトウェアをあらかじめインストールしたコンテナ用のイメージを作成しておきコンテナの起動を高速化する．作成するイメージは

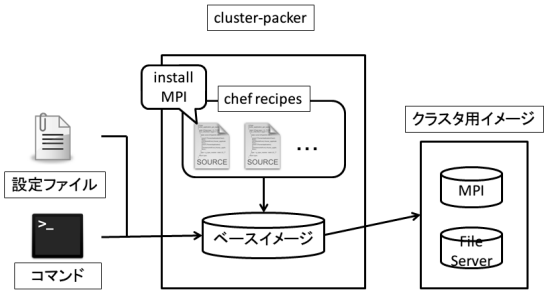

図 10 cluster-packer のベースイメージの作成の流れ
Fig. 10 Making base images with cluster-packer

2 種類でひとつはクラスタのそれぞれのノードが MPI アプリケーションのバイナリを共有するためのファイルサーバのイメージで，もうひとつは MPI アプリケーションを実行するためのイメージである．ファイルサーバ用のイメージには NFS(Network File System) が含まれ，アプリケーション用のイメージには MPI アプリケーションをコンパイル，実行するためのソフトウェアが含まれる．イメージの作成の流れを図 10 に示す．ベースイメージは OS がインストールされただけの初期状態のイメージで，そこに MPI アプリケーションが実行可能なソフトウェア環境を自動構築する．できたイメージを保存しコンテナ起動時にはこのイメージを指定することでソフトウェアのインストールにかかる時間を省略することができる.

5.3.2 コンテナによる仮想クラスタの構築

5.3.1 節で作ったイメージと設定ファイルを用いて任意の台数の仮想クラスタを構築する流れを図 11 に示す．設定ファイルに記述された台数のコンテナとファイルサーバ用のコンテナを立ち上げ，それらをネットワークでつなげる．仮想クラスタの構築にはコンテナを起動するための時間とネットワークを構築するための時間のみが必要でありカジュアルに構築や破棄を行うことができる．カジュアルに再構築することができるためユーザは常に新しい環境でソフトウェアをテストすることができ，開発者個人のテスト環境に依存するバグを防ぐことができる.

5.3.3 仮想クラスタへのデプロイ，テスト

5.3.2 節で作成した仮想クラスタへのソースコードのデプロイとテストの流れを図 12 に示す．設定ファイルで指定されたディレクトリのソースコードとテストコードを仮想クラスタのファイルサーバへとデプロイする．その後仮想クラスタ上でテストを実行し XML のテスト結果を cluster-packer へ送信してすべてのテストが成功しているかをチェックする．もしすべてのテストが成功していれば同じソフトウェア構成の本番環境ではソフトウェア構成に依存するバグは発生しないことが保証できる.

5.3.4 本番環境の構築，デプロイ

5.3.3 節でテストがすべて成功していた場合同じソフトウェア構成の環境でアプリケーションが動作可能であるの

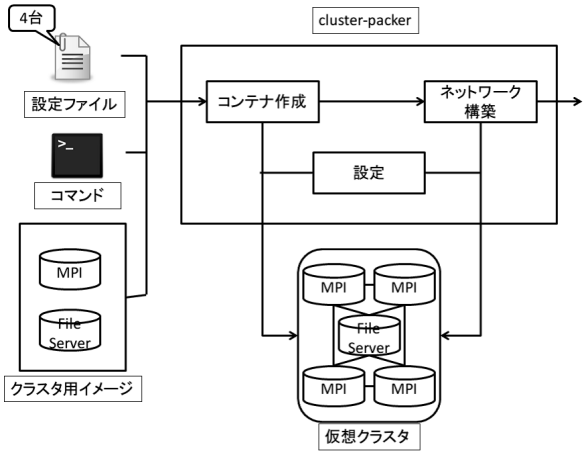

図 11　cluster-packer による仮想クラスタを構築する流れ
Fig. 11　Making a virtual cluster with cluster-packer

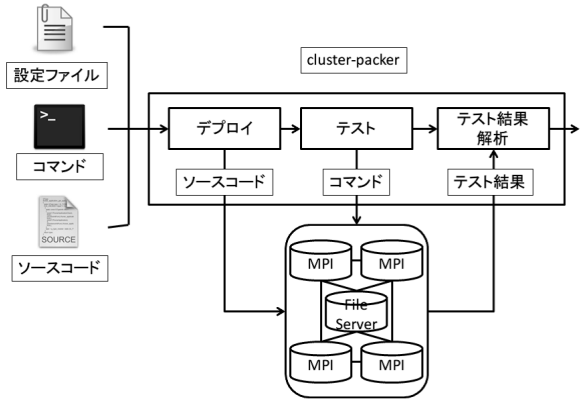

図 12　cluster-packer で仮想クラスタへデプロイ，テストする流れ
Fig. 12　Deploying to the virtual cluster with cluster-packer

で，仮想クラスタを構築した時と同じコードを用いて本番環境を構築しプログラムをデプロイする．本番環境の構築とデプロイの流れを図 13 に示す．設定ファイルに記述された IP アドレスのノードのソフトウェア環境を構築し，ネットワークを構築する．この環境は仮想クラスタと同じコードを用いているため同じソフトウェア構成が保証されているのでテストも成功する．最後にプログラムを本番環境へとデプロイしいつでも実行可能な状況にする．

6. おわりに

本稿では MPISpec，cluster-packer の 2 つのシステムを用いて MPI アプリケーションにおいて振る舞い駆動開発を可能にした．従来手法の MPIUnit でできなかった自然言語と階層構造によるテスト記述と出力を可能にした．自然言語の仕様からテストコードへの落とし込みが容易であり，仕様とテストコードの乖離を防ぐ．テスト結果はテストコードで表現された自然言語によるテストの説明と階層構造によるテスト間の関係を出力し可読性が高い．本体プログラムの非同期通信を同期処理時に行うことで同期処理の欠如による潜在的なバグの検知を可能にし従来手法で数

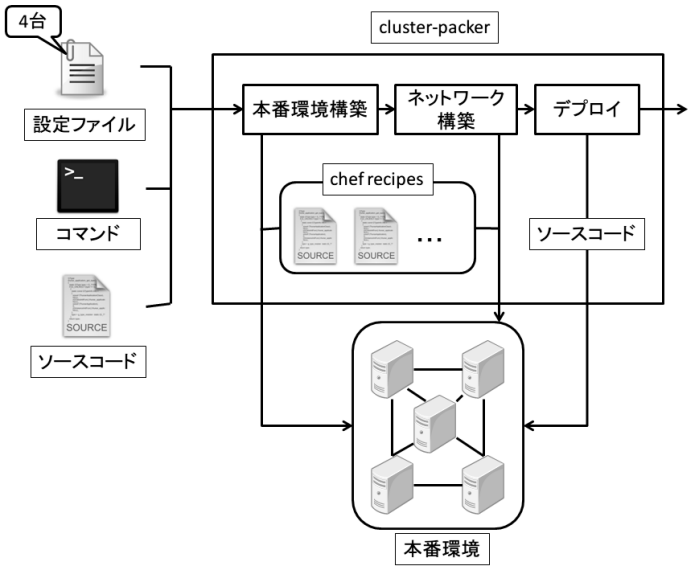

図 13　cluster-packer で本番環境の構築，ソースコードのデプロイの流れ
Fig. 13　Deploying to the cluster with cluster-packer

%程度の検知率である潜在的なバグを必ず検知できる．

cluster-packer は VM よりオーバヘッドの少ないコンテナを用いて容易に仮想クラスタ環境の自動構築することを可能にし使い捨てにできるテスト環境を提供する．

今後の課題として実開発に適用して評価する必要がある．潜在的なバグの検知率の向上が必要であり，現在は非同期通信後の同期処理の欠如を検知することができるが，ユーザが意図しない送信元から通信を受信することで起こる潜在的なバグも存在するためそれらを検知する必要がある．

参考文献

[1] Beck, K.: *Test-Driven Development: By Example. Addison-Wesley Professional* (2002).
[2] Ricken, M. and Cartwright, R.: ConcJUnit: unit testing for concurrent programs, *PPPJ '09*, pp. 129–132 (2009).
[3] Ricken, M. and Cartwright, R.: Test-First Java Concurrency for the Classroom, *SIGCSE '10*, pp. 219–223 (2010).
[4] The MPI Forum: MPI, Message Passing Interface Forum (online), available from ⟨http://wwww.mpi-forum.org⟩ (accessed 2015-02-06).
[5] 阿部真也，西谷泰昭：MPI プログラムのためのランダム遅延による Unit Testing Framework，情報処理学会第 70 回全国大会，pp. 1–2 (2004).
[6] : BDD, `http://dannorth.net/introducing-bdd/`.
[7] : CUnit, `http://cunit.sourceforge.net/`.
[8] : CppUnit, `http://freedesktop.org/wiki/Software/cppunit/`.
[9] : MPICH2, `http://www.mcs.anl.gov/research/projects/mpich2/`.
[10] : OpenMPI, `http://www.open-mpi.org`.
[11] Lefray, A.: Replication for Send-Deterministic MPI HPC Applications, pp. 2–3 (2013).
[12] : JUnit, `http://junit.org`.
[13] : Chef, `https://www.chef.io`.