修 士 論 文

モデル検査による
大規模な計算機環境での検証手法

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

戸川 博貴

2010 年 3 月

修 士 論 文

モデル検査による
大規模な計算機環境での検証手法

指導教員　青木利晃 准教授

審査委員主査　青木利晃 准教授
審査委員　小川瑞史 教授
審査委員　緒方和博 准教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

0810042 戸川 博貴

提出年月: 2010 年 2 月

概 要

リアルタイム OS をモデル検査を用いて検証するためには対象システムの振る舞いを記述した検査対象モデルの他に外部環境を表現した検査モデルが必要になる。これまでの研究の成果として外部環境を自動的に生成するツールが開発された。しかし、1 台の計算機で全ての検査モデルにおいて対象システムを検証することは困難である。そこで、本研究では、大規模の計算機環境を適用してモデル検査を行う方法を提案する。

目 次

第1章　はじめに

1.1　背景

近年、社会のあらゆるところにソフトウェアが使われているため、ソフトウェアの信頼性を保証することが重要になっている。ソフトウェアの信頼性を保証する方法の 1 つにモデル検査がある。これまでに OSEK/VDX を対象にモデル検査を用いた検証手法を研究し複数の検査モデルを生成できるようになっている。しかし、検査モデルが多くなると 1 台のマシンでは CPU、メモリなど資源不足の問題があり検証することは難しい。さらに、検証結果が膨大になることで問題点を把握することが困難になる。
そこで、本研究では大規模な計算機環境において SPIN を用いて検証を行い、その結果をどのように解析・表示するか検討する。本研究の特色は、実際に大規模な計算機環境で検証を行うことである。SPIN は誤った結果に至るまでの反例を表示する特徴があるので、大規模な計算機環境に利用すれば複数の検査モデルの検証が可能になるだけではなく、個々の検査モデルを比較・検討することも可能となる。ただし、検証を行うにはいくらか問題がある。1 つ目は、検査モデルにより検証したとき状態爆発問題が起きた場合は検証結果がでない。よって、どの検査モデルが検証可能か事前に判別できるようにしなければならない。2 つ目は、個々のマシンの CPU、メモリが異なる状態で検査モデルをどのマシンに渡せば効率よく検証可能か決める必要がある。これらの問題を踏まえて小規模な計算機環境から実験を行い、問題を整理し、SPIN を用いて大規模な計算機環境でも適用できるような検証方法を考案する。

1.2　目的

本研究の目的は、モデル検査ツール SPIN を用いて大規模な計算機環境で検証を行う方法を提案することである。
現在までに OSEK/VDX 仕様に基づいたリアルタイム OS を対象に SPIN を用いた検証手法に関する研究を行ってきた。その成果として OS により取り扱うタスクと資源間の構造、およびそれらの優先度などから複数の検査モデルを生成できるようになった。そこで本研究では、検査モデルを各マシンに渡して検証を行い、その検証結果を 1 台のマシンに収集し分析を行う。ここで、大規模な計算機環境で検証を行う場合の問題点がいくらかある。1 つ目は、多くの検査モデルが生成されるので検査モデルを選択する方法を決めることである。そのため、検査モデルの選択方法のメリット・デメリットを調べる必要があ

る。2 つ目は、検査モデルが多くなるのでその結果を分析し、どこに問題があるか解析し表示を行う方法を考えなければいけない。3 つ目は、個々のマシンスペックが異なる場合は検査したいモデルをどのように振り分けるか、どの程度の検査モデルを渡すか考慮する必要がある。これらの問題を実際に実験を行いながら解決し、検証結果の解析・表示を行えるようにする。

第2章　リアルタイムOSを対象にしたモデル検査手法

本章では、リアルタイム OS を対象としたモデル検査の適用手法について説明する。

2.1　OSEK/VDX

OSEl/VDX はドイツとフランスの自動車産業が自動車制御を行うエンジンコントロールユニット（ECU）で用いるプログラムの業界標準作成を目標としたプロジェクトである。また、そのプロジェクトが規定したオペレーティングシステム仕様も指す。OSEK/VDX の仕様書は、以下のサイトよりダウンロードが可能である。2010 年 2 月現在バージョン 2.2.3 が発行されており、本研究ではこのバージョンを対象にしている。

http://portal.osek-vdx.org

OSEK/VDX の主な機能としてタスク管理機能、応用状態 (アプリケーションモード)、割り込み処理機能、イベント制御機能、資源 (資源)、警告 (アラーム)、伝言 (メッセージ)、フックルーチンなどがある。OSEK/VDX の仕様書は自然言語で記述されているため意味に曖昧性を含む。これまでの研究で、OSEK/VDX 仕様の機能の内、タスク管理機能と資源に関する機能をモデル検査の対象としている。その理由は、タスクの振る舞いはリアルタイム OS の主機能であり、高い信頼性が要求されるからである。資源機能について説明する。タスクは、GetRecource というサービスコールを発行することにより資源を占有する。この時、プライオリティシーリングが起こる。プライオリティシーリングはタスクが資源を占有している時、タスクの優先度が一時的に資源に設定されている優先度になるという仕様である。これにより資源を占有しているタスクの優先度が高くなるため、他のタスクが RUNNING 状態に遷移することはなくなる。また、タスクが資源を占有しているときタスクは SUSUPEND 状態に遷移することは認められていないので SUSPEND 状態へ遷移するようなサービスコールは禁止されている。タスクが ReleaseRecource を発行するとタスクは資源を解放する。この時、タスクの優先度は元の優先度に戻る。本研究では、タスク、資源が正しく遷移しているか、タスクが資源を占有した時プライオリティシーリングがなされているかに焦点をあてる。

2.2 モデル検査

2.2.1 概要

モデル検査とは形式手法の一つである。形式手法では、数学的に基づいて検査したい性質の正しさを証明する。モデル検査では、検査対象となるソフトウェアやハードウェアの振る舞いを表現した状態遷移モデルを有限オートマトンに対応付け、有効グラフで表現する。特徴は有効グラフで遷移しうるすべての実行系列を網羅的に探索することである。網羅的に探索を行うため、デッドロックや無限の検出に適している。ただし、モデル検査の問題点として状態爆発問題がある。状態爆発問題とは、状態数が増加することでコンピュータのメモリの容量が不足し、検査できない問題である。この問題を解決する方法の一つとして検査したい性質以外の情報を抽象化することが挙げられる。

2.2.2 モデル検査ツール SPIN

モデル検査ツール SPIN は、AT & T ベル研が開発したモデル検査ツールである。SPIN では並行プロセス、個々のプロセスで非決定的な振る舞いを専用記述言語 Promela で記述する。記述したソースコードをもとに可能な動作を網羅的に探索して、検査したい性質が成立するかチェックする。検査したい性質は、ラベルや表明などで指定できる。並行動作や非決定動作は乱数を指定して実行することでシミュレーション実行できる。また、LTL(Linear Temporl Logic) を性質オートマトンに自動変換する機能も組み込まれている。検査したい性質に違反した場合は、反例として違反に至る経路を示すことができる。Promera では状態をラベル、遷移を goto 文で記述する方法がある。遷移のガード条件を非決定的に記述可能とうい特徴がある。

2.3 モデル検査に必要なもの

本研究では、検査対象としてリアルタイム OS を扱っている。リアルタイム OS は、単独で動作せず、外側から機能を呼びだされて動作する。例えばプリンタはライアントが印刷要求を出すことで動作したり、リアルタイム OS は、タスクからのサービスコールにより動作することが挙げられる。そのため対象ステムだけでなく、外側の挙動も記述する必要がある。モデル検査を行う場合、以下の 2 つが必要になる。図 2.1 に必要なモデルを示す。

- 検査対象モデル
  検査したいシステムの振る舞いを記述したモデルである。設計書や仕様書、ソースコードなどから検査する内容に関連した部分を切り出し、検査に必要な最低限の情報に抽象化して作成される。本研究では、OSEK/VDX のモデルを事例として扱っている。OSEK/VDX のモデルには、不可分実行を記述している。不可分実行を記

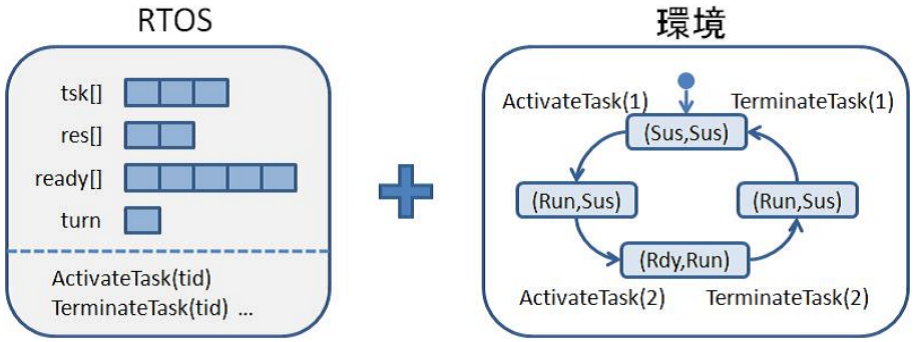


図 2.1: リアルタイム OS の検査に必要なモデル

述している箇所は割り込み処理が起こらないため、モデル検査を行うとき割り込みで発生したときの状態を考えない。

- 環境モデル
  検査対象のシステムがどのような環境で実行されるかを表現したモデルである。環境モデルには、検査対象とタスクと資源の多重度や優先度、遷移する条件などが記述されている。

2.3.1 モデル検査を行うときの問題点とその解決策

検査対象モデルと環境モデルからモデル検査を行うときの問題点として状態爆発問題がある。検査したいシステムの環境は、タスク（アプリケーション）の数やタスクが使用するメモリやドライバなど資源の数、タスクと資源の優先度、タスクと資源の参照関係などから膨大な組合せが存在する。モデル検査ツール SPIN はそれらの組合せを網羅的に探索するため、コンピュータのメモリが不足し全てを検査することはできない。その問題を解決する方法は各環境を構造的に分割して生成することである。図 2.2 に環境モデルを使ったモデル検査手法を示す。図 2.2 のように環境モデルから分割して生成した検査モデルと対象システムでモデル検査を行う。環境の構造を分割することで 1 つの環境で検査する状態数は少なくなるので状態爆発問題を回避することができる。これまでの研究の成果として環境モデルから検査モデルを自動的に生成するツールが開発された。

2.4 環境モデルから検査モデルの生成法

環境モデルから検査モデルを生成する流れを説明する。環境モデルから検査モデルを自動的に生成するためには以下のモデルが必要である。

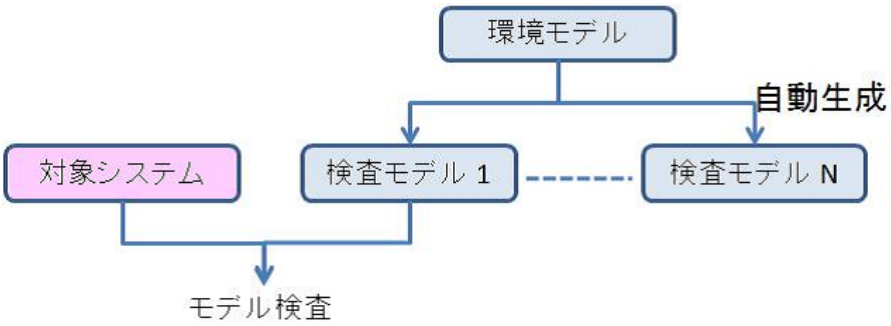


図 2.2: 環境モデルを使ったモデル検査手法

- 環境モデルのクラス図

- 環境モデルのステートチャート図

2.4.1 環境モデルのクラス図

環境モデルのクラス図は、検査したいシステムと構成する環境を記述したモデルである。検査対象に対してのモデル検査用クラス図を図 2.3 に示す。図 2.3 はタスクの数が 2、資源の数が 1 の環境におけるクラス図である。環境モデルのクラス図は、検査対象のリアルタイム OS に対して外部の環境クラスはタスククラスと資源クラスの二つで構成される。検査対象となる RTOS には、入力のバリエーションが記述されており、タスクや資源がどのような操作を行うか記述されている。RTOS クラスとタスククラスには関連があり、多重度は 1 つの RTOS に対して 0..2 である。同様に RTOS クラスと資源クラスにも関連があり、多重度は 1 つの RTOS に対して 1 である。タスクと資源間にも関連があり、タスクに対して資源の多重度は 0..1 であり、資源に対してのタスクの多重度は 1..2 である。タスク間同士にも関連があり、多重度は 0..2 である。タスククラスは属性として優先度とタスクの状態を持つ。また、資源クラスは属性として優先度と資源の状態を持つ。

2.4.2 環境モデルのステートチャート図

タスククラスのモデル検査用ステートマシン図を図 2.4 に示す。ステートチャート図では、タスクオブジェクトの状態とタスクがどのような条件で他の状態に遷移するか記述されている。

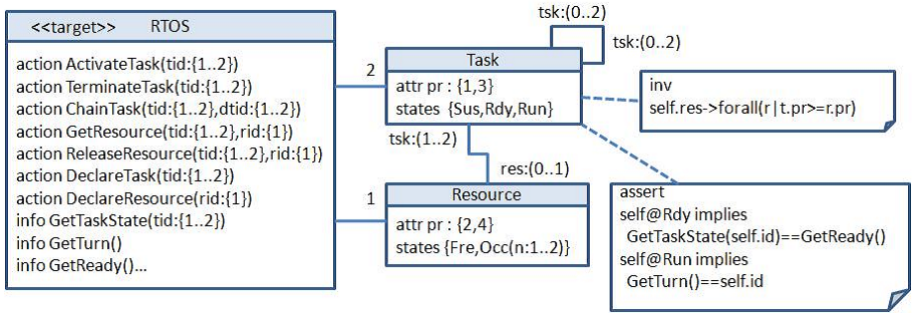


図 2.3: 環境モデルのクラス図の例

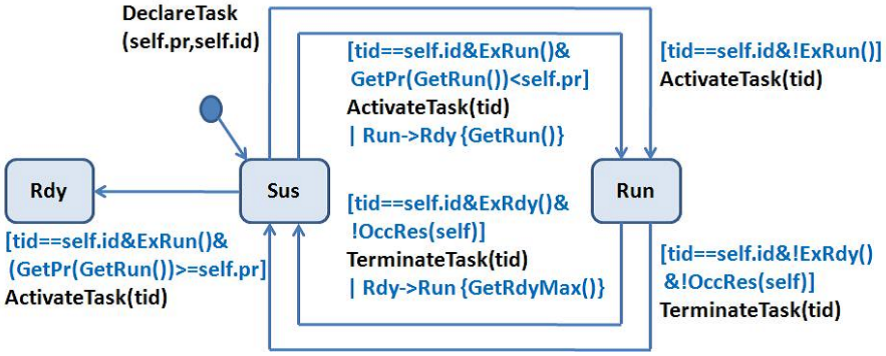


図 2.4: 環境モデルのタスクのステートチャート図の例

2.4.3 環境モデルのクラス図からオブジェクトを生成

環境モデルのクラス図より任意のタスクの数および資源の数におけるオブジェクトグラフを実体化する。図 2.5 は、タスクの数が 2、資源の数が 1 におけるオブジェクトグラフの例である。実体化されたオブジェクトグラフはタスクと資源の優先度やタスクと資源間の参照関係により考えられる組合せを全て実体化する。

2.4.4 生成したオブジェクトグラフとステートチャートの合成

実体化したオブジェクトグラフだけではタスクや資源がどのような振る舞いをするか記述されていないので環境モデルのステートチャート図を合成し、タスクと資源の振る舞いを記述する。図 2.6 にタスクの数が 2、資源の数が 1 の環境で実体化したオブジェクトグラフにステートチャート図を合成して得たステートチャート図を示す。合成して求めたステートチャート図はタスクと資源の状態とどのようなサービスコールが発行されたとき他の状態へ遷移するか記述されている。各状態には、タスク、資源の状態をする。図 2.6 では、各状態の記述は (タスク 1、タスク 2、資源 1) の順に状態を記述する。図??の状態に

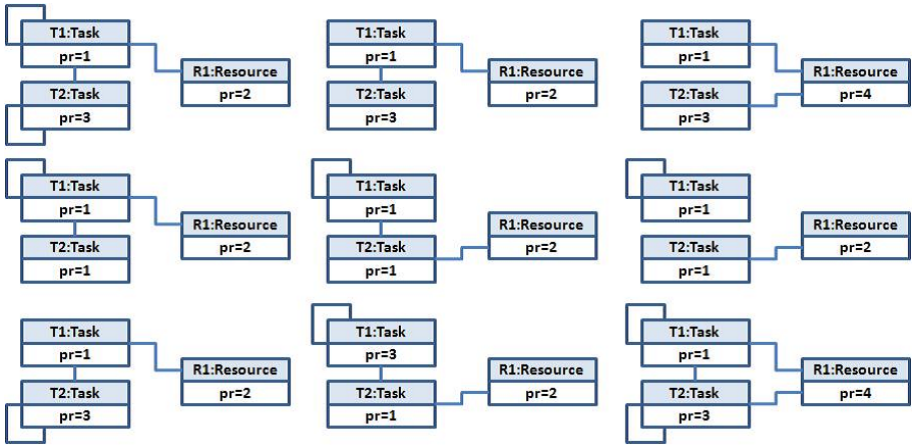


図 2.5: オブジェクトグラフの例

記述されている Sus、Rdy、Run、Fre、Occ の内容は以下である。

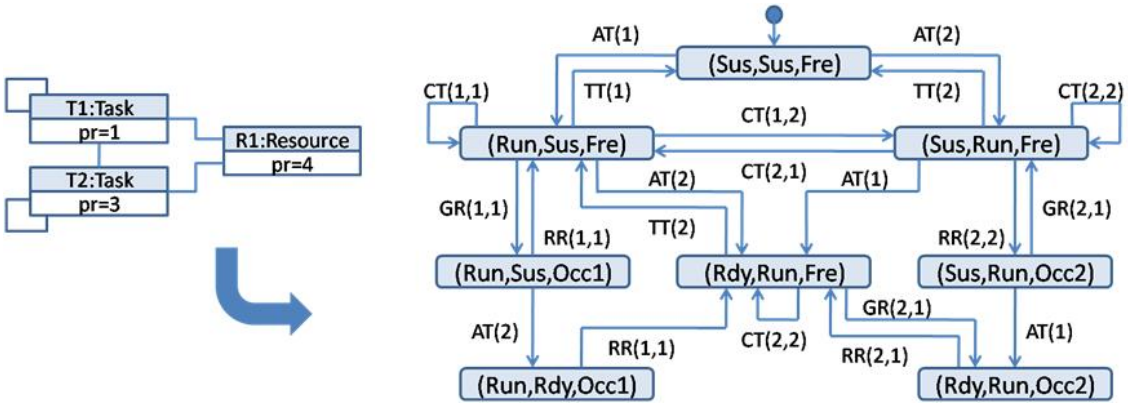


図 2.6: タスクの数が 2、資源の数が 1 のステートチャート図の例

- タスクの状態
  - Sus：SUSPENDED 状態を表す。タスクが起動されてるのを待っている状態
  - Rdy：READY 状態を表す。タスクの処理を待っている状態
  - Run：RUNNING 状態を表す。タスクの処理を行っている状態
- 資源の状態
  - Fre：資源が占有されていない状態

– Occ：資源が占有している状態

また、各サービスコールの概要について以下にまとめた。

- AT(TID)
  サービスコール ActivateTask である。呼ばれたタスク ID(TID) を SUSPENDED 状態から READY 状態へ遷移させる。

- TT(TID)
  サービスコール TerminateTask である。RUNNING 状態のタスク ID（TID) を SUSPENDED 状態へ遷移させる。

- CH(TID1、TID2)
  サービスコール ChainTask である。呼び出し元のタスク (TID1) を RUNNING 状態から SUSPENDED 状態に遷移させ、呼び出し先のタスク (TID2) を READY 状態から RUNNING 状態に遷移させる。

- GR(TID、RID)
  サービスコール GETRECOURDE である。タスク (TID) が資源を占有する。呼ばれた資源 (RID) は FREE 状態から OCCUPIED 状態へ遷移させる。

- RR(TID、RID)
  サービスコール RELEACERECOURCE である。タスク (TID) が資源を解放する。呼ばれた資源 (RID) は OCCUPIED 状態から FREE 状態へ遷移させる。

2.5 検査モデルツールの使い方

検査モデルを生成するツールの使い方について説明する。図 2.7 は環境モデルから検査モデルを生成するスクリプトである。検査モデル生成ツールは、対象となるシステムを検査するとき必要な環境を自動的に生成する。ユーザが任意の環境を指定することで、タスクと資源の優先度、タスクと資源の参照関係を考慮した組合せを全て生成する。検査モデル生成ツールは、タスク、資源の数を指定する箇所、タスクと資源がそれぞれどのような状態で遷移するか条件が記述されている。

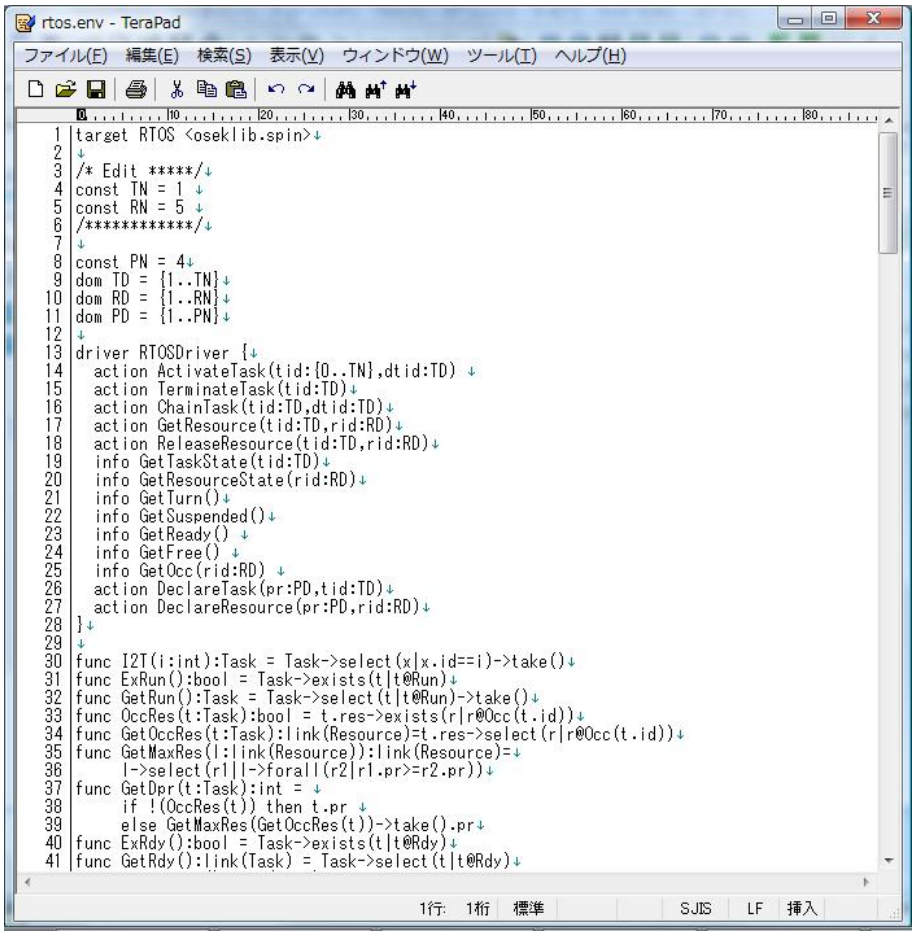


図 2.7: 環境モデルのスクリプト

```
1    target RTOS <oseklib.spin>
2
3    /* Edit *****/
4    const TN = 1
5    const RN = 5
6    /************/
7
8    const PN = 4
9    dom TD = {1..TN}
10   dom RD = {1..RN}
11   dom PD = {1..PN}
..     ......
57   class Task : TN {
58     attr pr : {1,2,3}
59     assoc tsk : Task (0,TN)
60     assoc res : Resource (0,RN)
..     ......
155  class Resource : RN {
156    attr pr : {2,3,4}
157    assoc tsk : Task (1,TN)
```

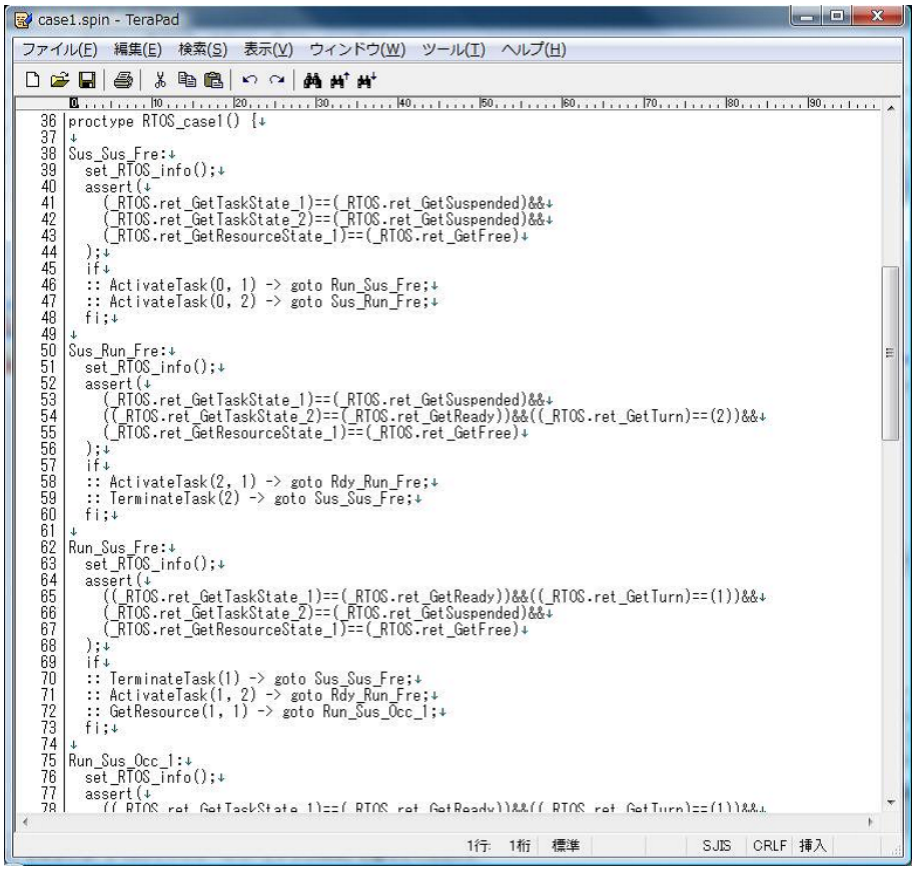
環境モデルの一部について説明する。環境モデルにおいて以下の 3 つを編集することで検査に必要な検査モデルを生成する。

- 検査に必要なタスクと資源の数を指定する。
  タスクの数と資源の数はそれぞれ 4 行目の const TN = 1 、5 行目の const RN = 5 の値を設定する。

- タスクの優先度のバリエーションを設定する。
  タスクの優先度のバリエーションを変更したい場合は 58 行目の {} 内の値を変更する。

- 資源の優先度のバリエーションを設定する。
  資源の優先度のバリエーションを変更したい場合は 156 行目の {} 内の値を変更する。

図 2.8 は環境モデルから生成した検査モデルの例である。図 2.8 は、図 2.6 のステートチャート図をモデル検査 Promela に変換して生成したモデルである。生成される検査モデルスクリプトには、タスクと資源の状態をラベルで記述されている。各状態には、検査したい性質、サービスコールが発行されたときの遷移先の状態が記述されている。この他に、環境の規模が大きくなるとタスクの数、資源の数が多くなるので、区別を図るためタ




```
proctype RTOS_case1() {↓
↓
Sus_Sus_Fre:↓
  set_RTOS_info();↓
  assert(↓
    (_RTOS.ret_GetTaskState_1)==(_RTOS.ret_GetSuspended)&&↓
    (_RTOS.ret_GetTaskState_2)==(_RTOS.ret_GetSuspended)&&↓
    (_RTOS.ret_GetResourceState_1)==(_RTOS.ret_GetFree)↓
  );↓
  if↓
  :: ActivateTask(0, 1) -> goto Run_Sus_Fre;↓
  :: ActivateTask(0, 2) -> goto Sus_Run_Fre;↓
  fi;↓
↓
Sus_Run_Fre:↓
  set_RTOS_info();↓
  assert(↓
    (_RTOS.ret_GetTaskState_1)==(_RTOS.ret_GetSuspended)&&↓
    ((_RTOS.ret_GetTaskState_2)==(_RTOS.ret_GetReady))&&((_RTOS.ret_GetTurn)==(2))&&↓
    (_RTOS.ret_GetResourceState_1)==(_RTOS.ret_GetFree)↓
  );↓
  if↓
  :: ActivateTask(2, 1) -> goto Rdy_Run_Fre;↓
  :: TerminateTask(2) -> goto Sus_Sus_Fre;↓
  fi;↓
↓
Run_Sus_Fre:↓
  set_RTOS_info();↓
  assert(↓
    ((_RTOS.ret_GetTaskState_1)==(_RTOS.ret_GetReady))&&((_RTOS.ret_GetTurn)==(1))&&↓
    (_RTOS.ret_GetTaskState_2)==(_RTOS.ret_GetSuspended)&&↓
    (_RTOS.ret_GetResourceState_1)==(_RTOS.ret_GetFree)↓
  );↓
  if↓
  :: TerminateTask(1) -> goto Sus_Sus_Fre;↓
  :: ActivateTask(1, 2) -> goto Rdy_Run_Fre;↓
  :: GetResource(1, 1) -> goto Run_Sus_Occ_1;↓
  fi;↓
↓
Run_Sus_Occ_1:↓
  set_RTOS_info();↓
  assert(↓
    ((_RTOS.ret_GetTaskState_1)==(_RTOS.ret_GetReady))&&((_RTOS.ret_GetTurn)==(1))&&↓
```

図 2.8: 生成される検査モデルの例

スクと資源のそれぞれ ID を割り振っている。この生成した検査モデルと検査対象モデルでモデル検査を行うことで検査対象モデルが正しく動作するか検証する。表 2.1 と表 2.2 に検査モデル生成ツールを利用して各環境で生成された検査モデルの数を表に示す。

表 2.1、表 2.2 は、タスクの数、資源の数に着目して生成した数である。各環境で生成した検査モデルの数はタスクや資源の優先度、タスクと資源の参照関係などの組合せにより数が異なる。例えば、表 2.1 のタスクの数が 1 、資源の数が 0 から 5 において数が異なるのは資源の優先度の組合せがいくつもあるからである。表 2.1 はタスクの優先度を {1、3}、資源の優先度を {2、4} に設定したときに生成された検査モデルの数である。表 2.2 はタスクの優先度を {1、2、3}、資源の優先度を {2、3、4} に設定したときに生成された検査モデルの数である。表 2.1 と比較すると表 2.2 は優先度のバリエーションが多くなっているので各環境で生成される検査モデルの組合せが多くなるのでこのような結果になる。

表 2.1: 生成した検査モデルの数 1

		タスクの数			
		1	2	3	4
資源の数	0	4	20	140	1540
	1	6	74	1014	19882
	2	8	272	7928	
	3	10	930		
	4	12	3140		
	5	14			

表 2.2: 生成した検査モデルの数

		タスクの数			
		1	2	3	4
資源の数	0	6	42	406	6090
	1	16	296	5944	
	2	30	1606	71250	
	3	48	7360		
	4	70	31142		
	5	96			

2.6 モデル検査を適用するときの問題点と解決策

モデル検査で 1 番の問題点ある状態爆発問題は、環境モデルから検査モデルを構造的に分割して生成することでより回避できる。しかし、検査モデルを使ったモデル検査には 2 つの問題点が挙げられる。

1. １台の計算機だけで検査するとき時間を要する
   検査対象モデルを個々の検査モデルにおいてモデル検査を適用しても現実の問題では、時間は限られるので全ての検査モデルにおいて検査することはできない。そこで、短時間でできるだけ多くの検査モデルにおいて検査対象モデルが正しく動作するか検証する方法を考えなければいけない。

2. 膨大な数の検査結果を得るとき、比較・検討に時間を要する
   表 2.1、表 2.2 のように合計で約 10 万程の検査スクリプトが生成される。これらを全て同時に検査するとその検査結果は膨大な数になるため検査結果を把握するたけで相当な時間を要する。また、検査結果から膨大な数の反例が検出されたとき、全

ての反例を解析することは困難である。この他に、各検査モデルは構造的に分割しているので、それぞれの環境間で比較・検討を行えることが可能になる。そのため、比較・検討を行える方法を考える必要がある。

以上の 2 つの問題点があるが、１つめの問題を解決する方法として大規模な計算機環境を利用することである。大規模の計算機環境を利用する利点は、１台当りに割り当てる検査モデルの数が少なくなるため検査時間の短縮につながるからである。そこで、実際に大規模の計算機環境を利用して実験を行う。2 つ目の問題を解決する方法として検査結果の表示方法を考え、比較を行える方法を提案する。より詳細な解析を行うためにクラスター分析を適用して解析できる方法を考える。解析を

2.7　検査の流れ

図 2.9 に大規模の計算機環境を用いてモデル検査を行う流れを示す。

1. 検査モデル生成ツールを用いて検査モデルを生成する。
   図 2.7 の検査モデル生成ツールを用いて検査するシステムを調べたい環境で検査するため検査モデルを生成する。

2. 生成した検査モデルをネットワークを使用して各計算機に割り振る。
   大規模の計算機環境を用いて検査を行うので検査モデルを分配する。今回の手法では、予め検査モデルを分配するか決めてから各計算機に分配する。理由はいくつかある。１つ目は、検査モデルの分配するアルゴリズムが複雑と考えたからである。動的に検査モデルを計算機に渡すことも可能だが、実装に至るまで時間がようするため実験を行えないと考えたからである。２つ目は、提案手法では、１つの計算機から複数の計算機に検査モデルを渡すためネットワークのトラフィックにより検査モデルを確実に渡せない可能性があると考えたからである。以上の理由から検査モデルを割り振る方法を考えるためには、静的に分配する方法が確実に実験結果を得るには適していると考えた。また、各計算機に分配するときに注意することは、検査を行うとき各計算機で均一な時間で検査を行うことである。検査モデルは、環境の規模が大きくなるにつれてせいせいされる状態数、遷移する条件が増加するため検査スクリプトの大きさが大きくなるのでモデル検査に時間を要するため検査時間がばらばらになる可能性がある。そこで、最適に分配しなければ、一部の計算機で検査が完了しても他の計算機が依然モデル検査を行っているので検査効率を悪くするので、最適に分配する。

3. 各計算機でモデル検査を行う。
   各計算機でモデル検査を行う。モデル検査は、シェルスクリプトを使って分配された検査モデルにおいて検査対象モデルを自動的に検査する。各検査モデルにおいてモデル検査を行った結果と反例は、各計算機に保存する。

4. 検査結果と反例をネットワークを介してユーザ側に収集する。
   各計算機で得た検査結果と反例は、1 つの計算機に保存されているので SSH を介してアクセスして検査結果と反例を入手する。

5. 反例の解析や検討が行えるような表に取り込んで表示を行う。

本研究では、計算機環境に検証施設さつきを用いて実験を行う。

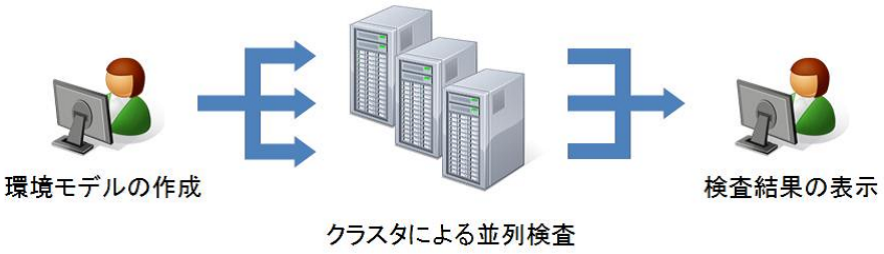


図 2.9: 検査の流れ

2.8 検証施設さつき

検証施設は、組込みシステムの信頼性向上を目的とし、ソフトウェアの信頼性技術の研究、技術移転および技術者育成のため学術機関，産業界に先端的な 検証施設を提供するもので、大容量メモリ高速演算クラスタと大規模演算クラスタの 2 種類のクラスタより構成、次のような検証、研究開発を行うことができる。

1. 大規模モデル検査

2. 大規模充足可能性判定

3. 大規模シミュレーションによる検証

図 2.10 にさつきを利用概要図を示す。検証施設さつきを利用するユーザは、一度現地に赴いて利用に関する研修を受ける。研修を受けたのち、外部から SSH を利用してアクセスする。図では利用者ワークステーションから SSH でアクセスすることに相当する。ユーザがさつきを使って検査を行いたいときは、SSH を使って検査したいモデルを送る。送ったモデルはログインサーバーに格納される。ログインサーバーは、ユーザが検査結果などを整理したり、大容量メモリ高速演算クラスタや大規模演算クラスタにアクセスするために利用する計算機である。大容量メモリ高速演算クラスタと大規模演算クラスタを使ってモデル検査を行う場合は、ログインサーバーから SSH を使ってアクセスする。検査するモデルはログインサーバーのホームディレクトリと同じパスに格納されているので、検査

したいモデルを送る必要はない。大容量メモリ高速演算クラスタと大規模演算クラスタにアクセスすると計算機を利用できるのでモデル検査を行い検査結果を調べる事ができる。ユーザに検査結果を送りたい場合は、ユーザの計算機まで戻り ssh を使うことでログインサーバーにある検査結果を入手できる。

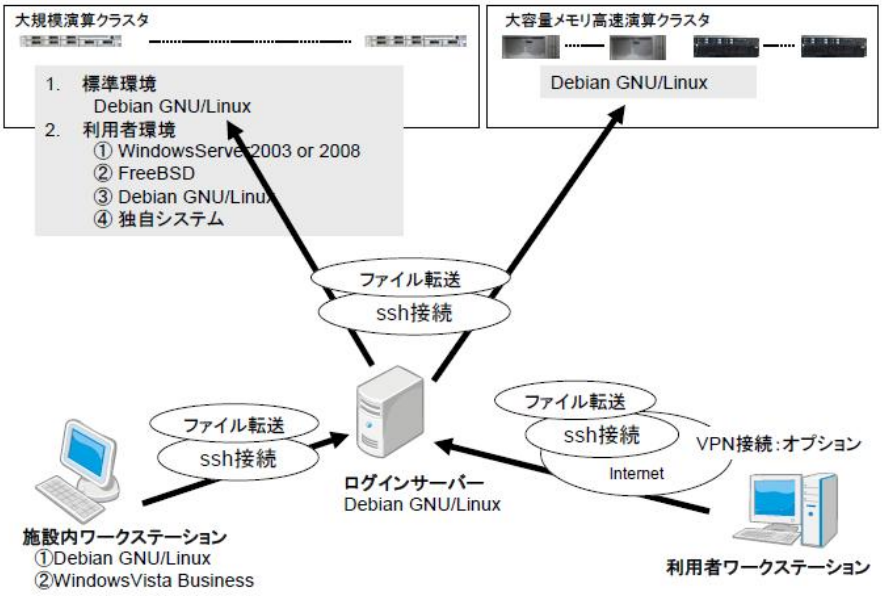


図 2.10: 検証施設さつきの利用概要図

第3章　実験

3.1　予備実験：1台の計算機によるモデル検査

大規模の計算機環境を用いてモデル検査の適用方法を考える前に実際に1台の計算機を用いてモデル検査を行ったときの検査時間、探索した状態数を調べる必要がある。この章では、1台の計算機でモデル検査を行った実験内容と実験結果について説明する。実験は、以下の計算機と検査モデルを使用した。表3.2は今回の実験に使用した検査モデルの表である。タスクの数、資源の数をそれぞれ変えたとき生成された検査モデルの数を表している。

表 3.1: 使用した計算機

計算機	デスクトップ
CPU	Core2DUO 2.13GHz
メモリ	2GByte

表 3.2: 実験に仕様した検査モデル

		タスクの数		
		1	2	3
資源の数	0	4	20	140
	1	6	74	1014
	2	8	272	7928
	3	10	930	
	4	12	3140	
	5	14		

3.1.1 実験手順

1. １台の計算機に生成した全ての検査モデルを格納する。

2. モデル検査を自動的に行う。
   検査の流れは環境の規模が小さい検査モデルから順次モデル検査を行い、1 つの環境でモデル検査が終了すると前の環境から資源の数が 1 つ多い環境に移り、モデル検査を行う。その検査結果を保存していく。

3. 長時間経過した後にモデル検査を終了し、そのとき探索した生成した状態、同じ状態を探索した状態数を調べる。
   各環境ごとのディレクトリに格納されている検査結果のファイルを読み取り、生成された状態数、探索の遷移の数をそれぞれ変数に代入し、検査した検査モデルの数全ての合計値を求める。

3.1.2 実験結果と考察

表 3.3 に長時間モデル検査を行った時の結果を示す。表 3.3 は、モデル検査を行うときに不可分実行を適用した場合と適用していない場合の 2 つで検査を行った。不可分実行をつけた検査では約 6 時間行った。そのとき検査した総モデル数は 2,472 個、検査した総モデル数 2,472 個において探索した状態数は 369,810 状態になった。一方で不可分実行を外した検査では約 13 時間行った。そのとき検査した総モデル数は 2,015 個、検査した総モデル数 2,015 個において探索した状態数は 8,771,795 状態になった。
実験結果より、1 台の計算機で長時間モデル検査を行っても検査モデル全体の一部しか

表 3.3: 実験結果

不可分実行	検査時間	検査したモデル数	状態数の合計
あり	約 6 時間	2472	369810
なし	約 13 時間	2015	8771795

表 3.4: 単位時間当たりの検査モデル数と状態数

不可分実行	検査したモデル数	状態数の合計
あり	400	60,000
なし	155	650,000

表 3.5: 全ての検査モデルを検査したときかかる時間

不可分実行	検査時間
あり	約 34 時間
なし	約 88 時間

検査できないことが分かった。表より、不可分実行をつけた場合でモデル検査を行った場合は単位時間当たり約 400 個の検査モデルにおいて検査対象システムを検査している。また、状態数は単位時間当たりに約 60,000 状態を探索したことが分かる。一方、不可分実行を外してモデル検査を行った場合は単位時間当たり約 155 個の検査モデルにおいて検査対象システムを検査している。また、状態数は単位時間当り約 650,000 状態を探索したことが分かる。この結果から表 3.2 の検査モデルを全て検査したときかかる時間を見積もることができる。不可分実行をつけた場合は、約 34 時間かけて検査を行う。一方で不可分実行を外した場合は、約 88 時間かけて検査を行うことが予想できる。
2 つの結果を比較すると、不可分実行を外してモデル検査を行ったとき検査したモデル数は不可分実行をつけた場合の半分以下しか検査を行っていないが、探索した状態数は不可分実行をつけたとき探索した状態数に比べて約 10 倍以上の状態数を探索していることが読み取れる。システムの不具合を検証するためには、あらいゆる状況を想定して検査を行う必要があるので、より大きくの状態数を探索することは重要である。表より一般的な計算機を使用したところ単位時間で約 650,000 状態を探索したので、計算機を n 台用意すれば短時間当たりに探索する状態数は、650,000 状態＊ n 台になる。このことから、1 台の計算機でモデル検査をおこなうより多くの計算機を用意してモデル検査を行う方が有効であることが言える。

3.2 実験 1：複数台の計算機環境によるモデル検査

1 台の計算機環境でモデル検査を行った場合、設計モデルの検査に長時間要することがわかった。その解決策として複数台の計算機を用いてモデル検査を行う。複数台の計算機を利用する利点は、1 台の計算機に割り当てる検査モデルの数が少なくなるため、検査に要する時間を短縮できることである。しかし、検査モデルを計算機環境に割り振るときに問題がある。環境モデルの規模が大きくなることで生成される検査モデルの規模も大きくなるため、各検査モデルで検査時間にばらつきが生じる可能性がある。検査時のばらつきを抑えるため、検査モデルを最適に割り振る方法を提案し評価する。

各環境の検査モデル 1 個当りにおいて検査に要する予測検査時間を求める。求めた予測時間より計算機で検査に要する時間を求め、その時間に到達するまで検査モデルを割り振ってモデル検査を行い、均一な時間で検査ができるか調べる。実験に使用した計算機の概要と検査モデルを表 3.6、表 3.7 に示す。表 3.6 は、検証施設さつきに設置されている計

算機 1 台の性能である。機種は、Sun Fire X4150 であり、CPU は、Intel 社 Xeon X5260 3.3GHz Dual Core、メモリの容量は 8GByte、ハードディスクの容量は 250GByte である。これまでの実験で StarBED を用いてモデル検査を行ったがメモリの容量の問題でコンパイルすることができなかった。しかし、この計算機を使用することで規模の大きい環境においてモデル検査を行ってもメモリの容量を超えることはなくなるのでコンパイルエラーになることはない。また、ハードディスクの容量も大きいことから膨大な数の検査結果を得たとしても全て保存できるので、データを逐次別のハードディスクに移動させたり、他の計算機にコピーすることはない。この計算機を今回の実験では、70 台使用して行う。表 3.7 は、これまでに生成した検査モデルの一部である。表は、タスクの数と資源の数に焦点を当てたときの検査モデルの数を表している。各環境の検査モデルには、タスク、資源の優先度やタスクと資源間の参照関係の組合せで用意されている。今回の実験でタスクの数が 3 個までの環境で行った理由は、検査モデルの数が多くなると実験にかかる時間がどのくらいになるか予想がつかないからである。そこでタスクの数が 3 個までに制限を加えて実験を行った。タスクの数が 3 個までに制限を加えても検査モデルの数は、約 12,000 程になるのでこの数から何かしらの傾向がつかめるか実験を行う。

表 3.6: 計算機の性能

機種	Sun Fire X4150
CPU	Intel 社 Xeon X5260 3.3GHz Dual Core
メモリ容量	8GByte
ディスク容量	250GByte
台数	70 台

表 3.7: 実験に使用した検査モデル

		タスクの数		
		1	2	3
資源の数	0	4	20	140
	1	6	74	1014
	2	8	272	7928
	3	10	930	
	4	12	3140	
	5	14		

表 3.8: 各環境で抽出した検査モデルの数

		タスクの数		
		1	2	3
資源の数	0	4/4	20/20	140/140
	1	6/6	74/74	200/1014
	2	8/8	200/272	200/7928
	3	10/10	200/930	
	4	12/12	200/3140	
	5	14/14		

3.2.1 実験手順

1. 各環境から検査モデルを抽出して検査モデル 1 個当りの予測検査時間を求める。各環境で抽出した検査モデルの数を表 3.8 に示す。検査モデルの抽出数は、環境の規模が小さいモデルで生成される検査モデルの数は少ないのでその場合は、全ての検査を抽出する。環境の規模が大きいモデルでは、タスクや資源の優先度、タスクと資源を参照関係などにより組合せが膨大な数になる。そこで、環境の規模が大きいモデルからは 200 個の検査モデルを抽出して検査を行う。しかし、タスクや資源の優先度、タスクと資源を参照関係などによりいろいろな組合せが存在するが生成される検査モデルはどれがどのような環境を表現しているか分からない。今回は、ランダムに 200 個の検査モデルを抽出した。

2. 各環境で検査にかかった時間を求める。時間の求め方は ls コマンドを使用した。ls コマンドを使用したり理由は、オプションを加えることでファイル名だけでなく、ファイルが出力された時間を表示するからである。

```
69        ls -l ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$x".txt >>
/home/users/h-togawa/result/result"$j"-"$z"-"$g"/date.txt
70        ls -l ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$y".txt >>
/home/users/h-togawa/result/result"$j"-"$z"-"$g"/date.txt
 ....
80        ls -l ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$x".txt >>
/home/users/h-togawa/result/result"$j"-"$z"-"$g"/date.txt
81        ls -l ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$i".txt >>
/home/users/h-togawa/result/result"$j"-"$z"-"$g"/date.txt
```

69 行と 80 行は各計算機で最初に検査した結果のファイルが出力された時間を各環

境を ID としたディレクトリ内の date.txt に保存する。また、70 行と 81 行は各計算機で最後に検査した結果のファイルが出力された時間を各環境を ID としたディレクトリ内の date.txt に保存する。図 3.1 に datae.txt の内容を示す。1 行目は、最初に検査した時間と検査に使用した検査モデルの ID が出力される。2 行目は、計算機で最後に検査した時間と検査モデルの ID を示している。data.txt に出力された時間を使って検査モデル 1 つ当りの検査時間を求める。

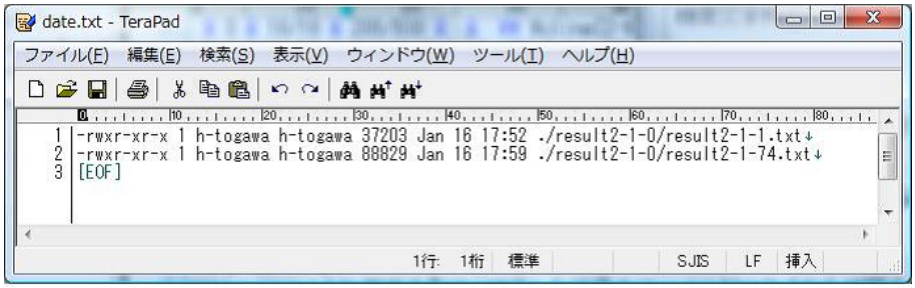


図 3.1: date.txt の例

3. 検査に要した時間を求める。求めた時間を各環境で抽出したモデル数で除算し、検査モデルの平均時間とする。以下に平均時間を求める式を以下のようにまとめた。

   - 検査開始時間：Tstart
   - 検査終了時間：Tend
   - 検査時間時間：T=Tstart-Tend
   - 抽出した検査モデルの数：x
   - 予測検査時間：Taverage = T/x = (Tend-Tstart)/x

   検査開始時間 Tstart は、各計算機で最初に検査した結果が出力される時間である。検査終了時間 Tend は、計算機で最後に検査した結果が出力される時間である。検査時間 T は、検査終了時間から検査開始時間を引いた時間である。抽出したモデルの数 x は、各環境で抽出した検査モデルの数を表している。予測検査時間は、求めた検査時間 T を抽出した検査モデルの数で除算した値である。表 3.9 に各環境での検査モデル 1 つあたりの検査時間を示す。時間の単位は [秒/個] である。表 3.9 より、タスクの数、資源の数をそれぞれ増加させたとき検査にかかる時間ばらつきが生じている。これは、検査モデルを 200 個抽出した環境モデルでは時間を有効に活用するため、1 台の計算機に 100 個の検査モデルを検査モデルの数が多くなっていることでそれぞれの検査もでるにおいてどれだけの時間を要するか分からなかったからである。今回の実験では、検査時間がより長くかかった方をその環境の検査時間とする。

表 3.9: 各環境の検査モデル 1 つあたりの予測検査時間

		タスクの数		
		1	2	3
資源の数	0	1	6	18.9
	1	1	7	18〜19
	2	1	12	27〜33
	3	12	19.8	
	4	30	43〜54	
	5	124		

表 3.10: 各環境の予測検査時間

		タスクの数		
		1	2	3
資源の数	0	4	120	2660
	1	6	518	19266
	2	8	3264	237840
	3	120	18600	
	4	360	150720	
	5	1736		

次に、この結果から各環境の検査時間を求める。表 3.10 に各環境で検査に要する検査時間を示す。例えば、タスクの数が 2、資源の数が 2 の環境において全ての検査モデルにおいて検査にかかる時間は 3264 秒である。時間単位は秒である。表 3.10 は、表 3.9 より得た各環境の検査時間を表 3.7 の各環境の検査モデルの数だけ積した値である。

この結果から各環境の検査モデルにおいて検査対象をモデル検査で検査する場合、435,222 秒要する。日数に換算すると約 5 日要することになる。今回の実験では 70 台の計算機を使用するので、計算機 1 台あたり約 6217 で検査が終了すると考えられる。

4. 計算機環境に検査モデルを割り振り均一な時間で検査が行われるか調べる。各計算機に割り振った内容は付録 A に記す。検査モデルを割り振る方法は、環境がタスク数１、資源数０の検査モデルから検査予測時間を加算する。1 つの環境でまだ 6127 秒に達しなければ資源の数を 1 つ多い環境の検査モデルを加えていく。

表 3.11: 実験結果の一部

割り振った検査モデル	モデル合計数	検査時間
タスク数 1 資源数 0〜5 タスク数 2 資源数 0〜2 タスク数 2 資源数 3(case1〜11)	461	2 時間
タスク数 2 資源数 3(case12〜317)	305	1 時間 36 分
タスク数 2 資源数 4(case787〜917)	130	1 時間 2 分
タスク数 2 資源数 4(case918〜1048)	130	4 時間 48 分
タスク数 3 資源数 2(case7723〜7928)	206	6 時間 27 分

5. 計算機でモデル検査を行う。より多くの状態を探索するため不可分実行を外してモデル検査を行う。

6. 検査に要した時間は ls コマンドを用いて測定する。

3.2.2 実験結果

実験結果の一部を表 3.11 に、全体の検査時間の結果を 3.2 に示す。この他の結果は付録 A に記す。表 3.11 の割り振った検査モデルは、それぞれの計算機にどのような検査モデルを分配したか記述している。タスクの数 1 資源 0〜5 は、タスクの数が 1 のとき資源の数を 0 から 5 の環境モデルを全て 1 つの計算機に分配したことを意味する。また、タスク数 2 資源数 3(case1〜11) はタスク数が 2、資源数が 3 の環境の中で 1 番目から 11 番目の検査モデルを計算機に分配することを意味する。モデルの合計数は、1 つの計算機に分配した検査モデルの合計値である。検査時間は、それぞれの計算機に分配した検査モデルで検査にかかった時間を表す。

また、このときモデル検査により作成した状態数を表 3.12、一度訪れた状態を再度探索した状態数を表 3.13、探索した状態数を表 3.14、使用したメモリ量を表表 3.15 にまとめた。今回の実験ではより多くの状態を探索するため不可分実行を外して行った。

表 3.12 は、各環境の検査モデル全てモデル検査を行ったとき作成した状態数を合計した値である。例えばタスクの数が 2、資源の数が 2 の環境において生成した状態数は 1359540 状態になる。表 3.12 よりタスクの数、資源の数を増加させることで状態数が増加することが分かる。

表 3.13 は、各環境の検査モデルを全てモデル検査を行うとき 1 度作成して訪れた状態に戻ってきた状態数の合計である。例えばタスクの数が 2、資源の数が 2 の環境において一度訪れた状態数は 5732 状態である。表 3.13 よりタスクの数、資源の数を増加させることで 1 度訪れた状態に戻る状態数は多くなっている。

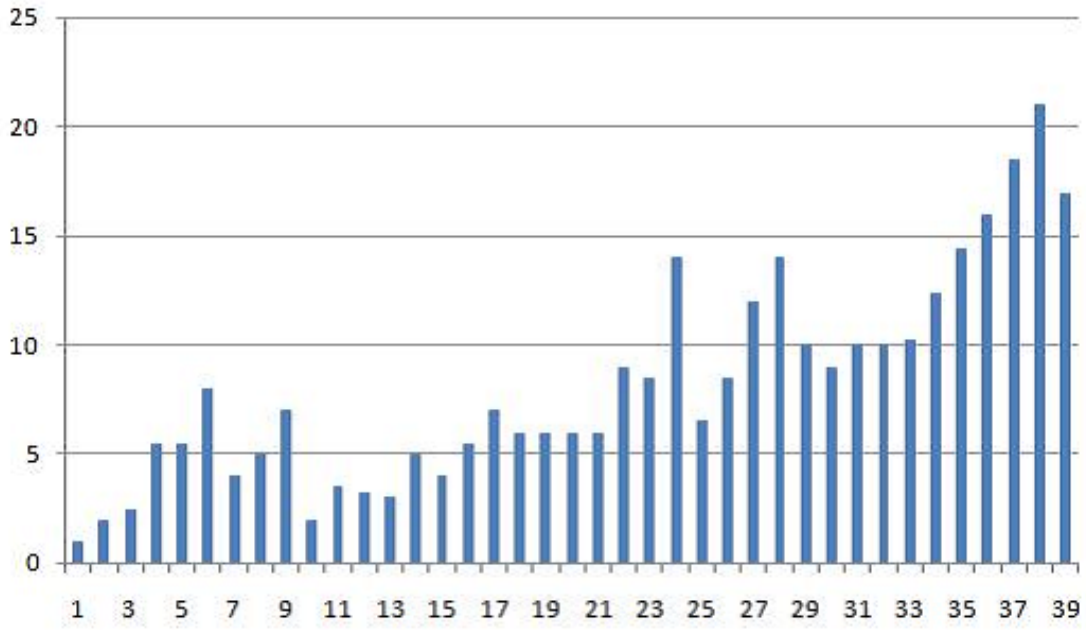


図 3.2: 実験結果の全体

表 3.14 は、各環境の検査モデルで探索した状態数を合計した値である。探索した状態数は、表 3.12 と表 3.13 のそれぞれの各環境を合計した値である。例えば、タスクの数が 2、資源の数が 2 の環境において探索した状態数は、1,365,272 状態である。この値は、表 3.12 と表 3.13 のタスクの数が 2、資源の数が 2 の値を合計したい値と等しくなる。表 3.12 と表 3.13 のタスクの数が 2、資源の数が 2 の値より 1,359,540 + 5,732 = 1,365,272 となることから等しいことが言える。表 3.14 より、タスクの数を 1 の固定し、資源の数を 0 から増加させる方法と資源の数を固定し、タスクの数を増加させる方法を比較すると、資源の数を固定し、タスクの数を増加させる方法がより多くの状態を探索していることが分かる。このことから、より多くの状態を探索したい場合は、タスクの数を増加させる方法がよい。表 3.15 は、各環境の検査モデルを全てモデル検査を行ったとき使用したメモリの容量の合計である。表 3.15 よりタスクの数が 1、資源の数を 0〜5 の環境でモデル検査を行ったときのメモリの使用量とタスクの数が 2、資源の数を 0〜4 の環境でモデル検査を行ったときのメモリの使用量を比較するとタスクの数が多いほどメモリの使用増加率も高くなっている。これは、タスクの数が増えることでサービスコールを発行する振る舞いが多くなるのでモデル検査で探索する状態数が増加するためである。

3.2.3 考察

表 3.11 より環境の規模が小さい検査モデルを重点的に割り振った計算機と規模の大きい環境の検査モデルを少数割り振った計算機を比較すると検査時間をある程度縮めるこ

表 3.12: 各環境で作成した状態数

		タスクの数		
		1	2	3
資源の数	0	3166	41078	495014
	1	6285	214003	4122372
	2	15246	1359540	40771203
	3	41915	8962747	
	4	121134	60075361	
	5	348263		

表 3.13: 一度訪れた状態数

		タスクの数		
		1	2	3
資源の数	0	6	156	1408
	1	15	856	12513
	2	52	5732	126301
	3	185	40496	
	4	606	283931	
	5	1827		

表 3.14: 探索した状態数

		タスクの数		
		1	2	3
資源の数	0	3172	41234	496422
	1	6300	214859	4134885
	2	15298	1365272	40897504
	3	42100	9003243	
	4	121740	60359292	
	5	350090		

表 3.15: モデル検査に使用したメモリ量

		タスクの数		
		1	2	3
資源の数	0	20.566	125.875	1154.187
	1	32.313	517.691	9290.395
	2	49.334	2405.11	88443.568
	3	81.493	12030.609	
	4	162.244	66693.105	
	5	370.28		

表 3.16: 生成した状態数、探索した状態数、使用したメモリ量

生成した状態数	116,577,327 状態
探索した状態数	117,051,411 状態
使用したメモリ量	181,376.77Mbyte

とができた。検査時間は最長で約 6 時間であることから 1 台の計算機で検査を行うよりも約 18 倍の速さで検査を行うことができた。この結果から、大規模の計算機環境でモデル検査を行えば時間を短縮できることが言えた。しかし、環境がタスク数が 2、資源数が 4 の検査モデルを割り振った計算機の結果を比較すると約 3 時間半の時間差が生じた。原因は、規模の大きい環境の検査モデルをランダムに抽出し、検査モデル 1 つの当りの時間を求めたため誤差が生じたと考えられる。今回の方法は、ランダムに抽出した検査モデル以外のモデルにおける検査時間は調べていないので、一部のモデルには検査にかなりの時間を要するモデルが含まれている可能性がある。この結果から全ての検査モデルについてモデル検査を行い、検査時間の傾向を調べる必要がある。

3.3 実験 2 行数と検査時間の関係により検査時間を求める

実験 1 より、検査モデル 1 つあたりの検査時間を求めることである程度時間のばらつきを抑える方向性は見えた。しかし、実験 1 の結果から検査時間のばらつきはまだあるため、より詳細な検査時間を測定する必要がある。そこで詳細な検査時間を測定するため検査モデルのスクリプトの行数とコンパイル時間の関係から近似式を求める。求めた近似式から各検査スクリプトのコンパイル時間を求める。実験に使用した計算機の概要と検査モデルを表 3.17、表 3.18 に示す。表 3.17 は、実験 1 と同様の計算機を用いて実験を行う。今回使用する計算機は 68 台である。実験 1 と比較して 2 台少なくなった理由は、検査モ

デル 1 つ当りの検査にかかる時間を求め、計算機に均一に割り振るとき 68 台までにおさまったからである。表 3.18 は、実験 1 と同様の検査モデルを使用する。実験 1 同じにする理由は、実験 1 との検査結果を比較するためである。

表 3.17: 使用した計算機の概要

機種	Sun Fire X4150
CPU	Intel 社 Xeon X5260 3.3GHz Dual Core
メモリ容量	8GByte
ディスク容量	250GByte
台数	68 台

表 3.18: 実験に仕様した検査モデル

		タスクの数		
		1	2	3
資源の数	0	4	20	140
	1	6	74	1014
	2	8	272	7928
	3	10	930	
	4	12	3140	
	5	14		

3.3.1 実験手順

1. 1 度各環境の検査モデルを用いてモデル検査を行い、検査モデルのコンパイルにかかった時間とモデル検査にかかった時間求める。今回の実験ではより詳細な時間を測定するため time コマンドを使用した。time コマンドを使うことで msec 単位の時間を求めることができる。

```
 1  START=‘date +%s‘
 2  spin -a -DNATOMIC case"$i".spin
 3  spin -a case"$i".spin
 4  gcc -DSAFETY pan.c -o pan
 5  END=‘date +%s‘
 6  compiletime=‘expr $END - $START‘
 7  START=‘date +%s‘
 8  ./pan > ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$i".txt
 9  END=‘date +%s‘
10  pantime=‘expr $END - $START‘
```

時間の測定方法について説明する。1 行目から 6 行目は検査モデル 1 つ当りのコンパイル時間を求めている。今回の実験では、Promela から c コードを生成する処理と c コードをコンパイルする処理の流れを 1 つのコンパイル時間として測定する。コンパイル時間の測定方法は、1 行目でコンパイル処理を行うときの時間を代入する。5 行目でコンパイルが終了したときの時間を代入する。コンパイルにかかった時間を求めるには 5 行目で代入した値から 1 行目に代入した値を引くことで求める事ができる。モデル検査は、8 行目の処理により実行される。そのため、7 行目にモデル検査を開始するときの時間を代入し、10 行目はモデル検査を終了したときの時間を代入する。モデル検査にかかった時間を求めるには 10 行目で代入した値から 8 行目に代入した値を引くことで求める事ができる。

2. 各検査モデルの行数を調べる。検査モデルの行数を調べる方法は shell に含まれている wc コマンドを用いて調べる。

```
 1  while :
 2  do
 3    if [ $stop ]
 4    then
 5      cd ..
 6      break 1
 7    fi
 8
 9    if test -f case"$i".spin
10    then
11      wc case"$i".spin | awk '{print $4" "$1}' >>
 /spin/rtoscases4/gyousuut"$a"r"$b".txt
12    fi
13    if test $i -eq $c
14    then
15      break 1
16    fi
17    i=`expr $i + 1`
18  done &
```

1 行目から 18 行目は検査モデルの行数を調べるソースコードである。9 行目から 12 行目は検査モデルが存在するとき、その検査モデルのファイル名と行数を gyousuut" $a"r" $b".txt に保存する。$a はタスクの数、$b は資源の数を代入する。13 行目から 16 行目は調べる検査モデルがない場合終了する処理である。図**??**に検査モデルスクリプトの行数結果の出力例を示す。

3. 行数と検査モデルのコンパイルに要する時間、およびモデル検査に要する時間の関係をグラフにまとめ、その関係から近似式を求める。図 3.4 に検査モデルの行数とコンパイル時間の関係を示す。横軸は検査モデルのスクリプトの行数、縦軸は各検査モデルでのコンパイル時間である。図 3.4 より検査モデルスクリプトの行数が増加するにつれてコンパイル時間も増加していることが分かる。また、一部の検査モデルは、近似式から逸脱した結果でコンパイルに時間を要しているモデルが存在した。原因を調べるため、検査モデルの状態数、遷移数を調べたが、どれも似た内容であるため原因を特定することができなかった。行数とコンパイル時間の関係より指数関数により近似式を求めた。指数関数を近似式に選んだ理由は、行数とコンパイル時間の関係を目視で見ると行数が 800〜1000 行の当りでコンパイル時間が急激に増加したからである。

4. 各行数での検査モデルのコンパイル時間を求める。

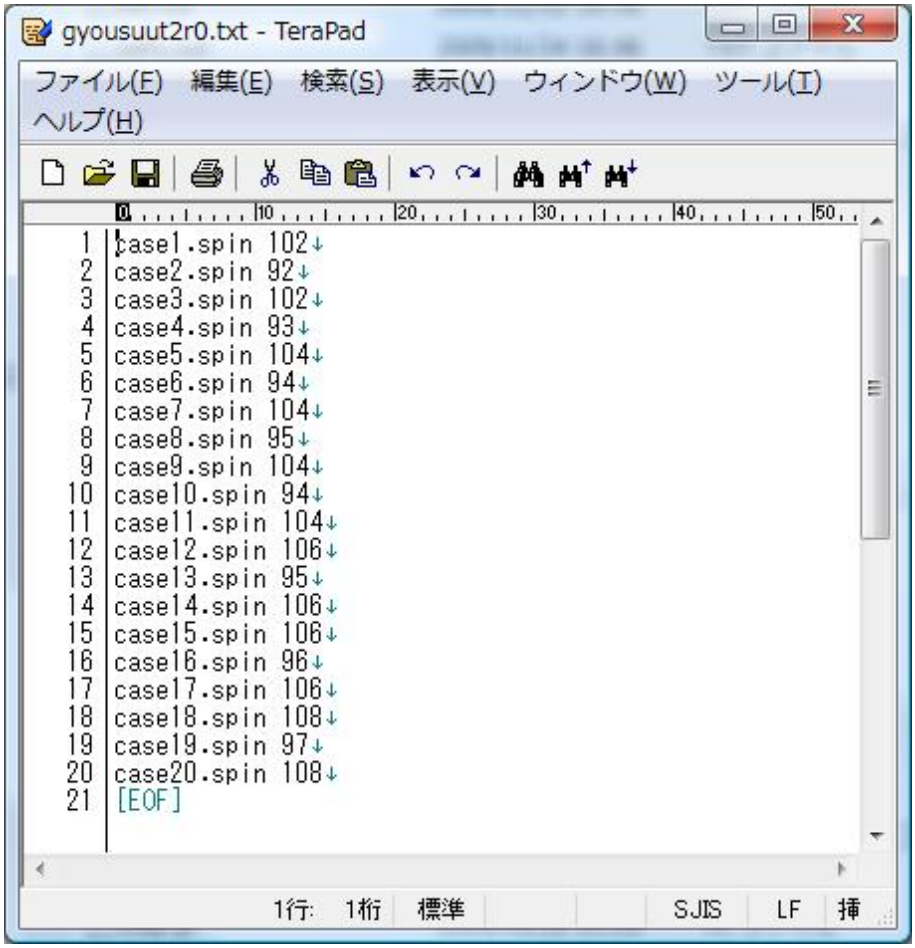


図 3.3: タスク数 2、資源数 0 における検査モデルスクリプトの行数結果

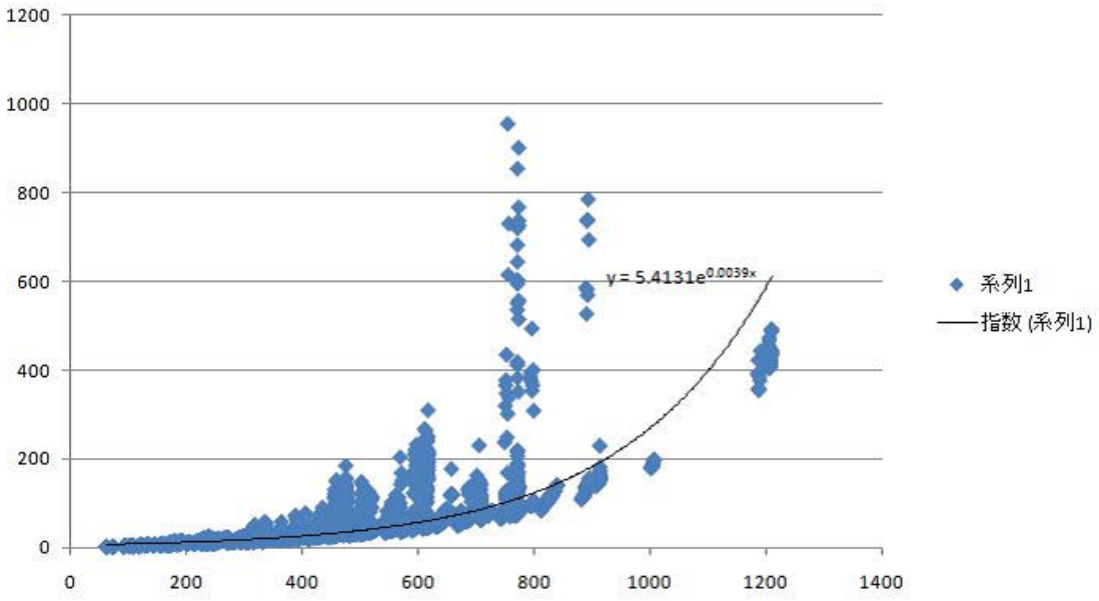


図 3.4: 行数とコンパイル時間の関係図

(a) 行数を定間隔に区切り、階級値を設定する。今回の実験では、行数の範囲と階級値を表 3.19 のように設定した。行数の範囲は 100 行刻みで 1 つの区間としている。このときの階級値は、階級を 100 にしていることからその中間値を階級値とする。階級を使用した理由は、行数の範囲が広いため検査時間を求めるために行数の範囲を設定しなれば計算する時間がかかるからである。

(b) 設定した階級値を近似式に代入しコンパイル時間を求める。表 3.20 は、設定した行数の範囲で求めた階級値を図 3.4 で求めた近似式に代入して得たコンパイル時間である。

(c) 各行数のモデル数を調べ、検査時間の合計値を求める。表 3.21 に行数のコンパイル時間を示す。表 3.18 の検査モデルの行数を設定した階級においてどれだけの数があるか調べる。各階級で求めた個数を 3.17 のコンパイル時間と乗算し、各階級での総コンパイル時間を求める。表より、検査モデルのスクリオウト行数が 300〜499 のとき多く存在していることが分かる。また、検査モデルの数が多いことによりこの区間一番時間がかかることもわかる。

(d) 表 3.21 の合計時間から 70 台の計算機で検査を行うときは平均で約 8750 秒になるように検査モデルを割り振る。しかし、割り振った結果表 3.17 の使用した計算機の台数に示すように 68 台におさまった。再度、検査時間を計算し分配しなおすことも検討されるが 70 台かかるところを 68 台で済むということは検査時間の効率を上げたことを意味するのでこのまま実験を行う。

表 3.19: 行数の範囲と階級値

行数範囲	階級値
1〜99	50
100〜199	150
200〜299	250
300〜399	350
400〜499	450
500〜599	550
600〜699	650
700〜799	750
800〜899	850
900〜999	950
1000〜1099	1050
1100〜1199	1150
1200〜1299	1250

表 3.20: 各行数のコンパイル時間

行数範囲	階級値	コンパイル時間 [秒]
1〜99	50	6.250649914
100〜199	150	9.324871409
200〜299	250	13.91106973
300〜399	350	120.75287181
400〜499	450	30.95963838
500〜599	550	46.18634074
600〜699	650	68.90190527
700〜799	750	102.7895363
800〜899	850	153.3439277
900〜999	950	228.762197
1000〜1099	1050	341.2730036
1100〜1199	1150	509.1193584
1200〜1299	1250	759.5166286

表 3.21: 各行数のコンパイル時間

行数範囲	個数	コンパイル時間 [秒]
1〜99	18	112.5
100〜199	296	2758.72
200〜299	1068	14857.2
300〜399	3143	65217.25
400〜499	3777	116935.9
500〜599	2468	113996.9
600〜699	1517	104521.3
700〜799	862	88613.6
800〜899	179	27440.7
900〜999	192	43929.6
1000〜1099	8	2730.4
1100〜1199	8	4072.8
1200〜1299	36	27342
	合計	612516.9

3.4 実験結果と考察

検査モデルを分配し、モデル検査を行った結果の一部を表 3.22 に示す。他の結果は付録 B に示す。表 3.22 と付録 B より 66 台の計算機では 2 時間〜2 時間半の範囲で検査を行うことができた。実験 1 と比較すると使用した計算機の台数が 2 台少ないが検査時間の観点でみると検査時間のばらつきを抑えたので行数のコンパイル時間に着もして検査モデルを分配する方法は有効である。しかし、残り 2 台の計算機では検査に 6 時間と 9 時間かかった。原因は、行数が 700〜800 行間で近似式では 200 秒ほどで終了すると予測されたが実際は 800〜1000 秒要するモデルが存在したためである。何故、同じ行数で検査時間にこれほどの差が生じたか原因を調べた。しかし、検査モデルスクリプトに記述されている状態数、遷移する数などを調べたが原因を特定することができなかった。この問題を解決する方法はいくつかある。1 つ目は、より多くの検査モデルを用いてモデル検査を行い、コンパイル時間のデータを収集し、より精度を高める方法である。2 つ目は、検査時間の誤差が一番大きい 700〜800 行間で時間の平均を求め、その値をこの区間の検査時間と考える方法がある。さらに考慮すると検査時間においてモデル数が分散している可能性があるので偏差値を求めて重みづけを行い、重みづけに基づいて分配する方法が考えられる。

表 3.22: 検査時間の結果

割り振った検査モデル	モデル合計数	検査時間
タスク数 1 資源数 0〜5 タスク数 2 資源数 0〜2 タスク数 2 資源数 3(case1〜189)	609	2 時間 30 分
タスク数 2 資源数 3(case190〜629)	439	2 時間 23 分
タスク数 2 資源数 4(case668〜860)	192	6 時間 32 分
タスク数 2 資源数 4(case1106〜1369)	263	2 時間 22 分
タスク数 3 資源数 2(case4996〜5269)	273	2 時間 20 分

第4章　検査結果の表示方法

この章では、計算機環境から得た結果を解析するための表示方法を提案する。

4.1　検査結果の表示

数万個の検査モデルを用いて検査を行ったとき、膨大な数のモデルで反例が検出される可能性がある。このとき表 4.1 のような記述や図 4.1 のようなクラス図を用いて各環境モデルのうち何個のモデルにおいてエラーが検出されたか表示する方法が考えられる。しかし、環境モデルはタスクや資源の優先度、タスクと資源間の参照関係などの要素が含むのでどのモデルがエラーを検出したかわかりづらい。また、エラーを検出したモデルの特定だけでなく、それぞれの検査モデル間で差分解析を行えることが望ましい。この問題を解決するために直感で問題の特定できる表示方法を提案する。そして、提案した方法を検査対象にバグを埋め込んだときの検査結果に適用し評価する。

4.2　検査結果の表示方法

検査結果の表示方法を検討するため、対象モデルに以下のバグを埋め込んだ。

- システムコール ActivateTask において Task が Redy 状態の場合でも enque 処理を行う。

表 4.1: 検査結果の例

		タスクの数		
		1	2	3
資源の数	0	2/4	17/20	139/140
	1	3/6	64/74	1011/1014
	2	4/8	237/272	7919/7928
	3	5/10	815/930	
	4	6/12	2754/3140	
	5	7/14		

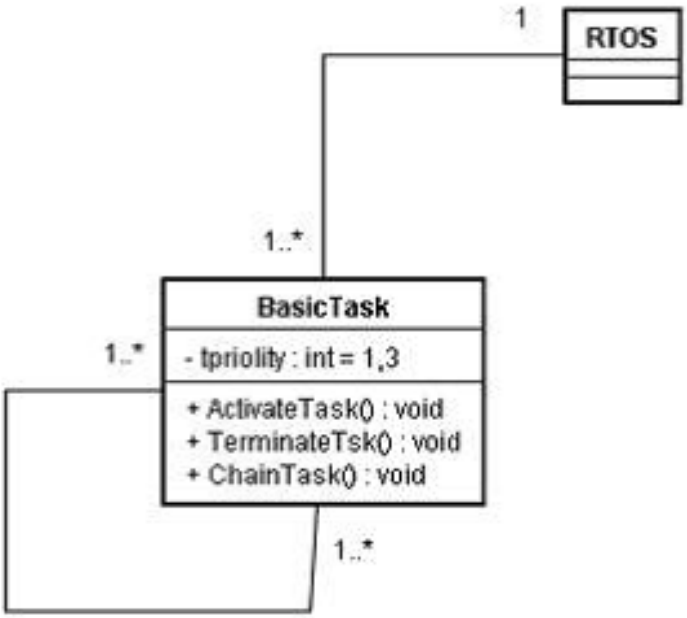


図 4.1: タスクの数が 1、資源の数が 0 の時の反例が得られた検査モデルのクラス図

- ChainTask の ActivateTask 処理を省略した。
- タスクがリソースを獲得するときタスクの実行優先度よりリソースの優先度が低い時、タスクの実行優先度はリソースの優先度に変更されるがタスクの実行優先度をタスクの優先度に変更する。

表 4.2 に上記のバグを埋め込んだ時検出した反例の数を示す。表 4.2 は、タスクの数と資源の数ごとの環境において分母は環境での検査モデルの数、分子は反例を検出した数である。

表 4.2 の結果を用いて検査結果の表示方法について実験し検討する。

4.2.1 実験 1

- 目的
  表 4.2 の検査結果を用いてタスクの数ごとに注目して検査結果を表示する。
- 実験結果
  検査結果をタスクの数ごとに表示した結果を表 4.3 に表示する。検査結果は反例がなければ「○」、反例が 1 つでも存在すれば「×」を表示する。

表 4.2: 埋め込んだバグによる検査結果

		タスクの数	
		1	2
資源の数	0	2/4	17/20
	1	3/6	64/74
	2	4/8	237/272
	3	5/10	815/930
	4	6/12	2754/3140
	5	7/14	

表 4.3: タスクの数のみに注目した検査結果表の例

タスクの数	検査結果
1	×
2	×
3	×

- 考察
  表 4.3 よりタスクの数のみに注目した環境全てに対して反例が検出されていることが分かる。しかし、この表からはエラーに関する情報を得ることができない。このことから詳細な表を追加で表示する。

はじめに表 4.2 の検査結果を用いて表 4.3 のように表示する。

4.2.2 実験 2

- 目的
  表 4.3 の中からタスクの数の箇所を指定した検査結果を表示する。

- 実験結果
  表 4.4 は、タスクの数を 1 に指定した時、どの検査モデルにおいて反例を検出したか表示している。表 4.4 の検査モデル No には、各検査モデルのファイル ID を示している。このとき、検査モデルのファイル ID は、混同しているのでファイルの ID に資源の数を加えている。例えば case0-1 は、資源の数が 0 の検査モデルのファイル ID が 1 であることを示している。検査結果は、各検査モデルで反例が検出されなければ「○」、反例が検出されれば「×」を表示する。

表 4.4: タスクの数が 1 の環境における各検査モデルの検査結果表の例

検査モデル No	検査結果
case0-1	○
case0-2	○
case0-3	×
case0-4	×
case1-1	○
case1-2	○
case1-3	○
case1-4	×
case1-5	×
case1-6	×
case2-1	○
case2-2	○
case2-3	○
case2-4	○
case2-5	×
case2-6	×
case2-7	×
case2-8	×

表 4.5: 資源の数に区別した検査結果表の例

資源の数	検査結果
1	×
2	×
3	×
4	×
5	×

- 考察
  表 4.4 より分かることは 2 つある。1 つは、指定したタスクの数の検査モデル全体においてどの位の反例を検出したか分かることである。2 つ目は、検査モデルのファイル ID を表示しているのでどのモデルが反例を検出したか特定できることである。しかし、この表かではまだ直感で各モデル間においてエラーに至る要因をつかむことはできない。そのため、より詳細な検査結果表を表示する。
  ところで、ここまで進めるとタスクの数を指定したときの検査結果を表示しているので、資源の数ごとに検査結果を表することができる。表 4.5 は、資源の数ごとに検査結果を表示する表の例である。表 4.5 よりタスクの数を 1 に指定し、資源の数ごとに注目した環境全てに対して反例が検出されていることが分かる。ただし、タスクの数、資源の数に注目したときの検査結果は表 4.2 で表示されているので、表 4.4 の表示は参考としてとらえる。

4.2.3 実験 3

より詳細な検査結果を表示するため、検査対象に関連する要素を追加する。本研究で対象としているシステムには、タスクや資源の数の他にタスクや資源の ID、タスクや資源の優先度の組合せ、タスクと資源間の参照関係の組合せ、タスクや資源の多重度などが挙げられる。これらの中から調べたい項目を指定したときの検査結果表を表示する。

実験 3-1

- 目的
  資源の ID が 1 の優先度に注目したときの検査結果を表示し、有効である検討する。

- 実験結果
  表 4.6 に資源の ID が 1、優先度が 2 の検査結果を示す。表 4.6 は、表 4.4 にタスクの数、資源の数、資源 ID1 の優先度をを追加したときの表示結果である。検査モデル

No は、各検査モデルを区別するため検査モデルのファイル ID に資源の数を加えている。例えば case1-1 は、資源の数が 1 の検査モデルで ID が 1 であることを示している。検査結果は、検各検査モデルで反例が検出されなければ「○」、反例が検出されれば「×」を表示する。補足としてタスクの数、資源の数を表示している。
資源 ID1 の優先度は、資源 ID の優先度が 2 の検査モデルにおいて存在する資源の優先度を全て表示している。複数の資源を持つ検査モデルにおいて資源の優先度を全て表示した理由は、注目したい資源の優先度だけを表示すると情報落ちによりエラーの傾向を見落とす可能性があると考えたので資源の優先度を全て表示している。
優先度の値は、左側から資源 ID1 の優先度を表示している。例えば、検査モデル No が case5-4 では、優先度の値は 2,2,4,4,4, で表示されており、これは、資源 1、資源 2、資源 3、資源 4、資源 5 の順番で対応している。

- 考察
表 4.6 より、資源の優先度に注目して反例の傾向について検討を行った。しかし、表より何の傾向を発見することができなかった。この表の有効性を示すためには、今回埋め込んだバグ以外の組合せでバグを埋め込んで検査を行い、その時の結果をこの表に適用しなければいけない。

実験 3-2

- 目的
タスクと資源間の参照関係に注目した検査結果を表示し、この表示方法が有効であるか検討する。

- 実験結果
タスク ID1 との参照関係に注目した検査結果表を表 4.7、表 4.8 に示す。表 4.7、表 4.7 は、表 4.4 にタスクの数、資源の数、タスクと資源の参照関係を追加したときの表示結果である。検査モデル No は、各検査モデルを区別するため検査モデルのファイル ID に資源の数を加えている。例えば case1-1 は、資源の数が 1 の検査モデルで ID が 1 であることを示している。検査結果は、検各検査モデルで反例が検出されなければ「○」、反例が検出されれば「×」を表示する。補足としてタスクの数、資源の数を表示している。
参照関係は、今回の実験ではタスク ID1 と参照関係になっているタスク、資源を表示している。タスク ID1 と参照関係になっていない場合は、参照無を記述する。タスク ID1 と参照関係になっているものはタスクの場合は T、資源の場合は R を記述する。複数のタスクや資源と参照関係になっている場合は、区別するため T や R の右隣にそれぞれのタスク、資源に対応している ID を記述する。例えば、表 4.7 の case2-5 の参照関係は、T1、R1、R2 である。これはタスク ID1 がタスク ID1、資源 ID1、資源 ID2 と参照関係になっていることを表している。

表 4.6: 資源 ID1 の優先度に注目して比較を行う場合の例

検査モデル No	検査結果	タスクの数	資源の数	資源 ID1 の優先度
case1-1	○	1	1	2
case1-4	×	1	1	2
case2-1	○	1	2	2,2
case2-5	×	1	2	2,2
case3-1	○	1	3	2,2,2
case3-6	×	1	3	2,2,2
case4-1	○	1	4	2,2,2,2
case4-7	×	1	4	2,2,2,2
case5-1	○	1	5	2,2,2,2,2
case5-8	×	1	5	2,2,2,2,2
case5-2	○	1	5	2,2,2,2,4
case5-9	×	1	5	2,2,2,2,4
case4-2	○	1	4	2,2,2,4
case4-8	×	1	4	2,2,2,4
case5-3	○	1	5	2,2,2,4,4
case5-10	×	1	5	2,2,2,4,4
case3-2	○	1	3	2,2,4
case3-7	×	1	3	2,2,4
case4-3	○	1	4	2,2,4,4
case4-9	×	1	4	2,2,4,4
case5-4	○	1	5	2,2,4,4,4
case5-11	×	1	5	2,2,4,4,4
case2-2	○	1	2	2,4
case2-6	×	1	2	2,4
case3-3	○	1	3	2,4,4
case3-8	×	1	3	2,4,4
case4-4	○	1	4	2,4,4,4
case4-10	×	1	4	2,4,4,4
case5-5	○	1	5	2,4,4,4,4
case5-12	×	1	5	2,4,4,4,4

表 4.7、表 4.7 をみるとタスク ID1 と参照関係になっているかなっていない場合で検査結果が異なっていることが一目でわかる。さらに詳しく見るとタスクの数が 1、資源の数が 0 の場合でタスク ID1 がタスク ID1 と参照関係になっているかなっていない場合で検査結果が異なっていることがわかる。このことからタスク ID1 がタスク ID1 と参照関係になっている全ての環境モデルにおいて反例を検出することが読み取ることができる。

- 考察
  実験結果より、タスクと資源の参照関係に注目することで環境間でエラーを引き起こす要因を発見することができた。今回は、意図的にバグを埋め込んだことからある程度の傾向はつかむことができたので環境間でエラーを引き越す境界を発見できた。しかし、通常はどのようなエラーが発生しどのようなバグが潜んでいるか分からない。この他の反例にも対応するタスクと資源の参照関係の組合せを検討しなければならない。

4.2.4 実験 4

- 目的
  実験 1 から実験 3 までは比較的簡単に検査結果の表示方法を検討するためにタスクの数を 1 に指定して実験を行った。しかし、環境の規模が大きくなるとそれだけ検査結果も多くなるので検査結果表も大きくなる。このとき、100 個の検査モデルを指定して表示する。ここでは環境の規模が大きくなるときの対応方法について提案する。その提案した方法が有効であるか実験し考察する。

- 実験結果
  検査結果をタスクの数ごとに表示した結果を表 4.9 に表示する。検査結果は反例がなければ「○」、反例が 1 つでも存在すれば「×」を表示する。「 … 」は検査結果の数が多くなるので省略している。実際の表示は、100 個の検査結果を表示する。

- 考察
  表 4.9 より分かることは 2 つある。1 つは、指定したタスクの数の検査モデル全体においてどの位の反例を検出したか分かることである。2 つ目は、検査モデルのファイル ID を表示しているのでどのモデルが反例を検出したか特定できることである。しかし、この表かではまだ直感で各モデル間においてエラーに至る要因をつかむことはできない。そのため、より詳細な検査結果表を表示する。
  ところで、ここまで進めるとタスクの数を指定したときの検査結果を表示しているので、資源の数ごとに検査結果を表することができる。表 4.5 は、資源の数ごとに検査結果を表示する表の例である。表 4.5 よりタスクの数を 1 に指定し、資源の数ごとに注目した環境全てに対して反例が検出されていることが分かる。ただし、タ

表 4.7: タスクと資源間の参照関係に注目して比較を行う場合の例 1

検査モデル No	検査結果	タスクの数	資源の数	参照関係
case0-1	○	1	0	参照無
case0-2	○	1	0	参照無
case0-3	×	1	0	T1
case0-4	×	1	0	T1
case1-1	○	1	1	参照無
case1-2	○	1	1	参照無
case1-3	○	1	1	参照無
case1-4	×	1	1	T1,R1
case1-5	×	1	1	T1,R1
case1-6	×	1	1	T1,R1
case2-1	○	1	2	参照無
case2-2	○	1	2	参照無
case2-3	○	1	2	参照無
case2-4	○	1	2	参照無
case2-5	×	1	2	T1,R1,R2
case2-6	×	1	2	T1,R1,R2
case2-7	×	1	2	T1,R1,R2
case2-8	×	1	2	T1,R1,R2
case3-1	○	1	3	参照無
case3-2	○	1	3	参照無
case3-3	○	1	3	参照無
case3-4	○	1	3	参照無
case3-5	○	1	3	参照無
case3-6	×	1	3	T1,R1,R2,R3
case3-7	×	1	3	T1,R1,R2,R3
case3-8	×	1	3	T1,R1,R2,R3
case3-9	×	1	3	T1,R1,R2,R3
case3-10	×	1	3	T1,R1,R2,R3

表 4.8: タスクと資源間の参照関係に注目した表の例 2

検査モデル No	検査結果	タスクの数	資源の数	参照関係
case4-1	○	1	4	参照無
case4-2	○	1	4	参照無
case4-3	○	1	4	参照無
case4-4	○	1	4	参照無
case4-5	○	1	4	参照無
case4-6	○	1	4	参照無
case4-7	×	1	4	T1,R1,R2,R3,R4
case4-8	×	1	4	T1,R1,R2,R3,R4
case4-9	×	1	4	T1,R1,R2,R3,R4
case4-10	×	1	4	T1,R1,R2,R3,R4
case4-11	×	1	4	T1,R1,R2,R3,R4
case4-12	×	1	4	T1,R1,R2,R3,R4
case5-1	○	1	5	参照無
case5-2	○	1	5	参照無
case5-3	○	1	5	参照無
case5-4	○	1	5	参照無
case5-5	○	1	5	参照無
case5-6	○	1	5	参照無
case5-7	○	1	5	参照無
case5-8	×	1	5	T1,R1,R2,R3,R4,R5
case5-9	×	1	5	T1,R1,R2,R3,R4,R5
case5-10	×	1	5	T1,R1,R2,R3,R4,R5
case5-11	×	1	5	T1,R1,R2,R3,R4,R5
case5-12	×	1	5	T1,R1,R2,R3,R4,R5
case5-13	×	1	5	T1,R1,R2,R3,R4,R5
case5-14	×	1	5	T1,R1,R2,R3,R4,R5

表 4.9: タスクと資源間の参照関係に注目した表の例 2

検査モデル No	検査結果
case2-1	○
case2-2	×
case2-3	○
case2-4	×
case2-5	×
case2-6	×
case2-7	×
case2-8	×
case2-9	×
case2-10	×
•••	•••
case2-60	×
case2-61	×
case2-62	×
case2-63	×
case2-64	×
case2-65	○
case2-66	○
case2-67	○
case2-68	○
case2-69	○
case2-70	○
case2-71	×
case2-72	×
•••	•••
case2-96	×
case2-97	×
case2-98	×
case2-99	○
case2-100	○

表 4.10: 資源数での検査結果表の例

資源の数	検査結果
1	×
2	×
3	×
4	×
5	×

スクの数、資源の数に注目したときの検査結果は表 4.2 で表示されているので、表 4.10 の表示は参考としてとらえる。

4.2.5 実験 5

より詳細な検査結果を表示するため、検査対象に関連する要素を追加する。本研究で対象としているシステムには、タスクや資源の数の他にタスクや資源の ID、タスクや資源の優先度の組合せ、タスクと資源間の参照関係の組合せ、タスクや資源の多重度などが挙げられる。これらの中から調べたい項目を指定したときの検査結果表を表示する。

- 目的
  タスクの ID が 1 の優先度に注目したときの検査結果を表示し、有効である検討する。

- 実験結果
  表 4.11 にタスクの ID が 1、優先度が 2 の検査結果を示す。表 4.11 は、表 4.9 からタスク ID1 の優先度を含む検査モデルの結果を表示した例である。検査モデル No は、各検査モデルを区別するため検査モデルのファイル ID に資源の数を加えている。例えば case1-1 は、資源の数が 1 の検査モデルで ID が 1 であることを示している。検査結果は、検各検査モデルで反例が検出されなければ「○」、反例が検出されれば「×」を表示する。補足としてタスクの数、資源の数を表示している。また、表 4.11 は検査モデルの数が多いので 20 個のモデルを抽出して表示している。
  タスク ID1 の優先度は、タスク ID の優先度の他に検査モデルにおいて存在するタスクの優先度を全て表示している。複数のタスクを持つ検査モデルにおいてタスクの優先度を全て表示した理由は、注目したいタスクの優先度だけを表示すると情報落ちによりエラーの傾向を見落とす可能性があると考えたのでここではタスクの優先度を全て表示している。
  優先度の値は、左側からタスク ID1 の優先度を表示している。例えば、検査モデル No が case0-2 では、優先度の値は 1,3 で表示されており、これは、タスク 1、タスク 2 の順番に対応している。

表 4.11: タスク ID1 の優先度に注目する場合の例

検査モデル No	検査結果	タスクの数	資源の数	タスク ID1 の優先度
case0-1	○	2	0	1,1
case0-2	○	2	0	1,3
case0-3	○	2	0	3,3
case0-4	○	2	0	1,3
case0-5	×	2	0	1,1
case0-6	×	2	0	1,3
case0-7	×	2	0	3,3
case0-8	×	2	0	1,3
case0-9	×	2	0	1,1
case0-10	×	2	0	1,3
case0-11	×	2	0	3,3
case0-12	×	2	0	1,1
case0-13	×	2	0	1,3
case0-14	×	2	0	3,3
case0-15	×	2	0	1,1
case0-16	×	2	0	1,3
case0-17	×	2	0	3,3
case0-18	×	2	0	1,1
case0-19	×	2	0	1,3
case0-20	×	2	0	3,3

- 考察
  表 4.11 より、タスクの優先度に注目して反例の傾向について検討を行った。しかし、表より何の傾向を発見することができなかった。今回埋め込んだバグではこの表の有効性を示すことができなかった。しかし、今回埋め込んだバグ以外のバグを埋め込んだ場合に有効性を示す可能性がある。そのため、今回埋め込んだバグ以外の組合せでバグを埋め込んで検査を行い、その時の結果をこの表に適用しなければいけない。

4.2.6 実験 6

より詳細な検査結果を表示するため、検査対象に関連する要素を追加する。ここでは、タスクの数が 2 つあるのでタスク ID1 との参照関係に注目した検査結果表とタスク ID2 と参照関係に注目した検査結果表を作成し、埋め込んだバグを解析できるか評価する。

実験 6-1

- 目的
  タスク ID1 と資源間の参照関係に注目した検査結果を表示し、この表示方法が埋め込んだバグに対して有効であるか検討する。

- 実験結果
  タスク ID1 との参照関係に注目した検査結果表を表 4.12、表 4.13 に示す。表 4.12、表 4.7 は、表 4.4 にタスクの数、資源の数、タスクと資源の参照関係を追加したときの表示結果である。検査モデル No は、各検査モデルを区別するため検査モデルのファイル ID に資源の数を加えている。例えば case1-1 は、資源の数が 1 の検査モデルで ID が 1 であることを示している。検査結果は、検各検査モデルで反例が検出されなければ「○」、反例が検出されれば「×」を表示する。補足としてタスクの数、資源の数を表示している。
  参照関係は、今回の実験ではタスク ID1 と参照関係になっているタスク、資源を表示している。タスク ID1 と参照関係になっていない場合は、参照無を記述する。タスク ID1 と参照関係になっているものはタスクの場合は T、資源の場合は R を記述する。複数のタスクや資源と参照関係になっている場合は、区別するため T や R の右隣にそれぞれのタスク、資源に対応している ID を記述する。例えば、表 4.12 の case0-8 の参照関係は、T1、T2 である。これはタスク ID1 がタスク ID1、タスク ID2 と参照関係になっていることを表している。
  表 4.12 をみるとタスク ID1 と参照関係になっているかなっていない場合で検査結果が異なっていることが一目でわかる。さらに詳しく見るとタスクの数が 1、資源の数が 0 の場合でタスク ID1 がタスク ID1 と参照関係になっているかなっていない場合で検査結果が異なっていることがわかる。このことからタスク ID1 がタスク ID1

表 4.12: タスク ID1 との参照関係に注目する場合の例

検査モデル No	検査結果	タスクの数	資源の数	T1 との参照関係
case0-1	○	2	0	参照無
case0-2	○	2	0	参照無
case0-3	○	2	0	参照無
case0-4	○	2	0	参照無
case0-5	○	2	0	T1
case0-6	×	2	0	T2
case0-7	×	2	0	T2
case0-8	×	2	0	T1,T2
case0-9	×	2	0	参照無
case0-10	×	2	0	参照無
case0-11	×	2	0	参照無
case0-12	×	2	0	T1
case0-13	×	2	0	T1
case0-14	×	2	0	T1
case0-15	×	2	0	T2
case0-16	×	2	0	T2
case0-17	×	2	0	T2
case0-18	×	2	0	T1,T2
case0-19	×	2	0	T1,T2
case0-20	×	2	0	T1,T2

と参照関係になっている全ての環境モデルにおいて反例を検出することが読み取ることができる。

- 考察
  表 4.13 より、タスク 1 との参照関係に注目して反例の傾向について検討を行った。表より、参照関係がある場合とない場合で反例が検出される場合は発見することができた。しかし、今回埋め込んだバグ以外のバグを埋め込んだ場合に有効性を示す可能性がある。そのため、今回埋め込んだバグ以外の組合せでバグを埋め込んで検査を行い、その時の結果をこの表に適用しなければいけない。

表 4.13: タスク ID2 との参照関係に注目する場合の例

検査モデル No	検査結果	タスクの数	資源の数	T2 との参照関係
case0-1	○	2	0	参照無
case0-2	○	2	0	参照無
case0-3	○	2	0	参照無
case0-4	○	2	0	参照無
case0-5	×	2	0	T1
case0-6	×	2	0	T1
case0-7	×	2	0	T1
case0-8	×	2	0	T1
case0-9	×	2	0	T2
case0-10	×	2	0	T2
case0-11	×	2	0	T2
case0-12	×	2	0	T2
case0-13	×	2	0	T2
case0-14	×	2	0	T2
case0-15	×	2	0	T1,T2
case0-16	×	2	0	T1,T2
case0-17	×	2	0	T1,T2
case0-18	×	2	0	T1,T2
case0-19	×	2	0	T1,T2
case0-20	×	2	0	T1,T2

4.2.7　考察

抽象的な表をまず表示し、段階を経て詳細な内容を表示する方法を行った。段階を経て詳細な内容を順序をたどって解析することで理解できる。また、表 4.7、表 4.8、表 4.12、表 4.13 のように参照関係から一目で違いを見つけることが可能である。ただし、提案した方法は、同一の環境においての検査結果表しか考えていない。差分解析を行うためにはタスクの数や資源の数が異なる検査モデル同士を比較・検討する必要があるので異なる環境間での差分解析を行える表示方法を考える必要がある。本来の目標は、タスクの数や資源の数、タスクや資源の優先度の組合せ、タスクと資源の参照関係など多くの要素から調べたい項目着目して表示を行うことである。ただし、この方法を解決するには 2 つの問題点がある。1 つは、環境の規模が大きくなるにつれてタスクの数や資源の数、タスクや資源の優先度、タスクと資源の参照関係のバリエーションが増大するので、どのような方法を適用して表示を行うかが問題になる。2 つ目は、今回提案した手法のようにエラーを検出した検査モデル同士を比較してエラーの要因となるものを発見しやすくするように表を作成しなければならない。これらの問題を解決して表示する方法を提案することが今後の課題となる。

第5章　反例解析の方法

5章では、エラーを検出したモデルがどのような環境で発生したか理解できる表を提案した。しかし、エラーの傾向を表から解析できない場合は反例の内容を解析する必要がある。この章では、反例の解析方法を提案し評価する。

5.1　反例解析における問題点

反例解析における問題点がいくつかある。1つ目は、各検査モデルで起因するエラーのバリエーションが多くなると解析に手間がかかることである。表5.1、表5.2は反例結果の例を示す。タスクの数、資源の数における全体の検査モデルの数を分母に記述し、分子はそのうちエラーを検出した検査モデルの数を表している。表5.1のように各環境の全ての検査モデルにおいてエラーが検出されたとき反例の解析を行わなくても同一のエラーに起因していることが分かる。しかし、表5.2のように各環境でエラーを検出した検査モデルがばらばらの場合、エラーのバリエーションが複数存在するため表からでは原因特定が困難になる。2つ目は、同一のエラー内容においてサービスコールの実行列が異なる場合区別していなければ修正されず放置される可能性がある。これらの問題の解決方法としてクラスター分析を適用し有効であるか調べる。

表 5.1: 検査結果の例

		タスクの数		
		1	2	3
資源の数	0	4/4	20/20	140/140
	1	6/6	64/74	1014/1014
	2	8/8	272/272	7928/7928
	3	10/10	930/930	
	4	12/12	3140/3140	
	5	14/14		

表 5.2: 検査結果の例 1

		タスクの数		
		1	2	3
資源の数	0	2/4	17/20	139/140
	1	3/6	64/74	1011/1014
	2	4/8	237/272	7919/7928
	3	5/10	815/930	
	4	6/12	2754/3140	
	5	7/14		

5.2 クラスター分析

クラスター分析は、大量のデータが存在するとき類似しているデータ同士を 1 つのグループにまとめる方法である。図 5.1 にクラスター分析の適用例を示す。クラスター分析を行う利点は以下の二つがある。

- 個体のデータ構造からグループ化を行える。
  個体をグループごとに分ける場合、区別するための基準が必要になる。しかし、クラスター分析は個体のもつデータ構造からグループ化を行えるため簡潔に区別することができる。図 5.1 では、「読解力」と「計算力」の 2 つのデータからグループ分けができる。

- グラフィカルに表示するため直感で理解できる
  図 5.1 のようにそれぞれの個体が点在しているとき、類似している個体同士を円により 1 つのグループを表現するため第 3 者が見ても理解できる。

このクラスター分析を使用するため、統計解析ソフト R を使用する。

5.3 統計解析ソフト R

統計解析ソフト R は、データを統計的に解析する R 言語を使用してデータを解析する解析ツールである。統計解析ソフト R は、フリーで入手することができるので、誰でも統計的にデータを解析することができる。また、統計的に解析する方法は、いくつかの関数が用意されておりそらを利用することで解析できる。この他に、エクセルで収集したデータをそのままコピーすることで大量のデータを解析できる。

統計解析ソフト R によるクラスター分析の方法を説明する。

1. エクセルのデータを R に読み込む

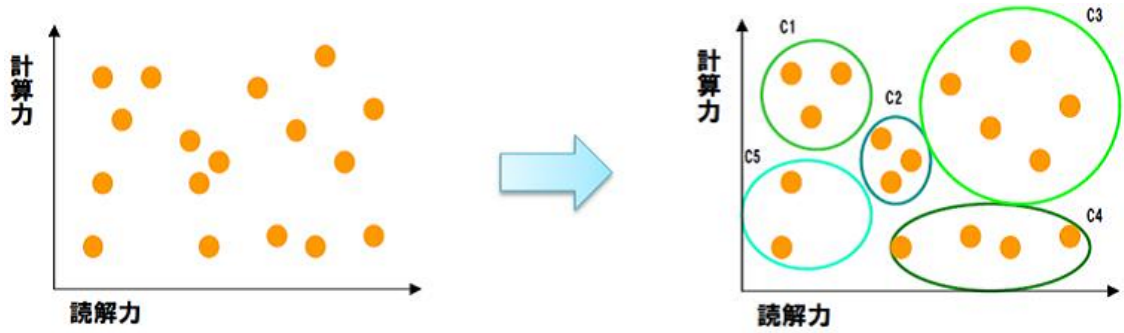


図 5.1: クラスター分析

2. dist(読み込んだデータ列名) により各反例間の距離を求める。この結果を「データ列名.d」に保存する。記述は、「データ列名.d ¡- dist(データ列名)」と記述する。

3. 次に「(データ列名.hc) ¡- hclust(データ列名))」を記述する。

4. クラスター分析の結果を表示するため「par(mfrow=c(2, 2))」と記述する。ここで、c(2,2) の数字は、結果をどれだけの数で表示するための値である。例えば c(2,2) は、2*2=4 個の分析結果を表示する。

5. クラスター分析の結果を表示する。「plot(データ列名.hc,hang =-1,main=”最長距離法”)」と記述することでクラスター分析の結果を表示できる。

5.4 反例解析の実験方法 1

この実験では反例結果に出力されるステップ数を使ってクラスター分析を適用しグループ化できるか検討する。

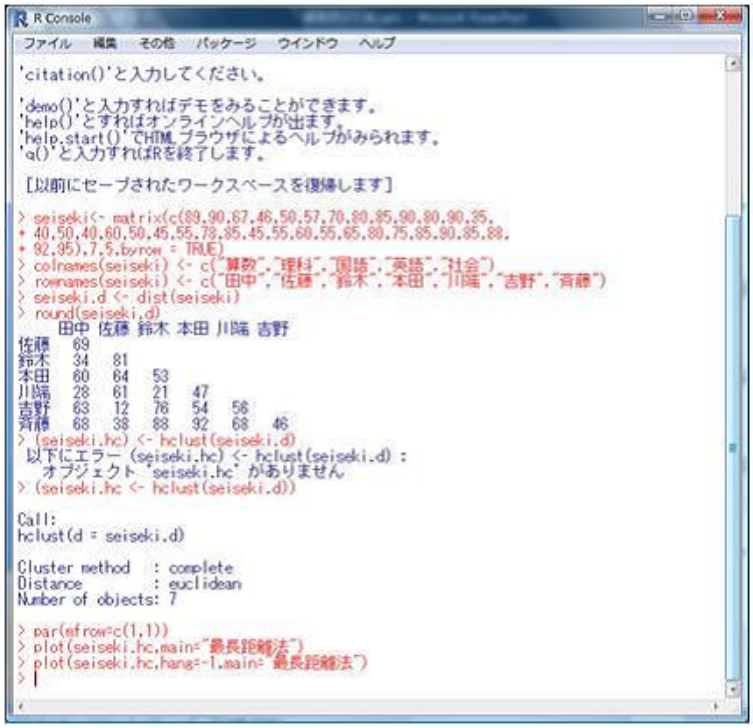


図 5.2: 統計解析ソフト R

```
 1       BEGIN0:initialization of the library
 2        END
 3        BEGIN0:DeclareTask(1, 1)
 4        END
 5            BEGIN1:GetTaskState(1)
 6            END
 7            BEGIN1:GetResouceState(0)
 8  spin: trail ends after 77 steps
 9  #processes: 2
10  ready[0] = 100
11  ready[1] = 100
12  ready[2] = 100
13  ready[3] = 100
14  ready[4] = 100
15  tsk_state[0].tpriority = 1
16  tsk_state[0].dpriority = 1
17  tsk_state[0].tstat = 1
18  tsk_state[0].actcnt = 0
```

ステップ数は、上記の 8 行目に示されている。ステップ数は、エラーが検出されたときど

れだけの行数でエラーに至ったかを表示する。実験の手順は以下のように進める。

1. 検査対象モデルにいくつかのバグを埋め込む。
   今回の実験では以下のバグを埋め込んだ。

   (a) ActivateTask の enque の位置を変更

   (b) TerminateTask の deque、ActivateTask 処理の省略

   (c) リソース獲得時のタスクとリソースの優先度比較を変更

   以上のバグを埋め込んだ理由は、ソースコードを記述するとき誤字脱字が起こりやすそうだと考えたからである。

2. モデル検査で各検査モデルでの設計モデルを検査する。
   モデル検査を行うときに用いた検査モデルは、タスクの数が１、資源の数が０～５の環境の検査モデルを合計 54 個使用した。

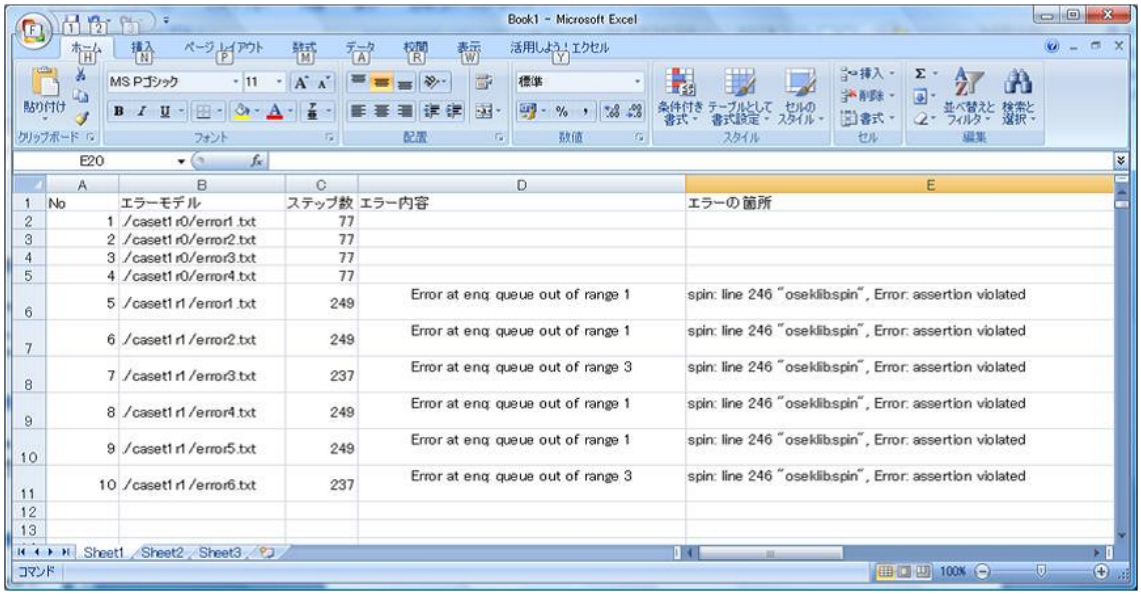
3. エクセルにエラーのモデル、ステップ数、設計モデルでエラーが発生した箇所、表明違反があるか書き込む。他の結果を付録 C に示す。
   図 5.3 に反例内容をエクセルに表示した例を示す。表には、No、ステップ数、エラーの内容、エラーの箇所の 4 つを表示してる。それぞれの項目について説明する。

   - No
     クラスター分析を適用するとき文字列を認識できない。そのかわりに実数を通し番号として No をつけている。
   - エラーモデル
     各環境においてどの検査モデルでエラーを検出したか表示している。
   - ステップ数
     エラーが検出されたときどれだけの行数でエラーに至ったかを表示する
   - エラーの内容
     各検査モデルでエラーを検出したときどのようなエラーを検出したか表示する。
   - エラーの箇所
     反例が出力されるとき検査対象モデルのどの箇所で発生したか表するので解析を補助する役割として記述している。

4. エクセルのステップを使ってクラスター分析を行う。

クラスター分析の適用実験 1

a,b,c のバグを３つとも組み込んだ場合でクラスター分析を適用した。図 5.4 にクラスター分析を適用した結果を示す。横軸は、エクセルの No を示す。縦軸は、各エラーモデ



No	エラーモデル	ステップ数	エラー内容	エラーの箇所
1	/caset1 r0/error1.txt	77		
2	/caset1 r0/error2.txt	77		
3	/caset1 r0/error3.txt	77		
4	/caset1 r0/error4.txt	77		
5	/caset1 r1 /error1.txt	249	Error at enq: queue out of range 1	spin: line 246 "oseklibspin", Error: assertion violated
6	/caset1 r1 /error2.txt	249	Error at enq: queue out of range 1	spin: line 246 "oseklibspin", Error: assertion violated
7	/caset1 r1 /error3.txt	237	Error at enq: queue out of range 3	spin: line 246 "oseklibspin", Error: assertion violated
8	/caset1 r1 /error4.txt	249	Error at enq: queue out of range 1	spin: line 246 "oseklibspin", Error: assertion violated
9	/caset1 r1 /error5.txt	249	Error at enq: queue out of range 1	spin: line 246 "oseklibspin", Error: assertion violated
10	/caset1 r1 /error6.txt	237	Error at enq: queue out of range 3	spin: line 246 "oseklibspin", Error: assertion violated

図 5.3: 反例内容の表示

ルの距離を表している。表より細かな部分で反例を区別していることが分かる。しかし、エラーの内容やエラーの箇所により区別を行っていないので、この結果からエラーの内容を把握することはできない。

クラスター分析の適用実験 2

b の ActivateTask の処理省略,c の 2 つを組み込んだ場合でクラスター分析を適用した。図 5.5 にクラスター分析を適用した結果を示す。横軸は、エクセルの No を示す。縦軸は、各エラーモデルの距離を表している。表より実験 1 と比較するとクラスターの数が減少していることがわかる。また、反例間の距離も実験 1 より短くなっていることが分かる。このことから、情報量を少なくすることで反例間の距離は短くなり、クラスターの数も減少することがわかった。クラスター分析を適用するためにはいかにして情報の抽出量を考えるか必要である。しかし、エラーの内容やエラーの箇所により区別を行っていないので、この結果からエラーの内容を把握することはできない。

クラスター分析の適用実験 3

a,c と b の ActivateTask の処理省略を組み込んだ場合でクラスター分析を適用した。図 5.6 にクラスター分析を適用した結果を示す。横軸は、エクセルの No を示す。縦軸は、各エラーモデルの距離を表している。表より実験 1、実験 2 と比較するとクラスターの数がさらに減少していることがわかる。また、反例間の距離は実験 1 や実験 2 を比べると長く

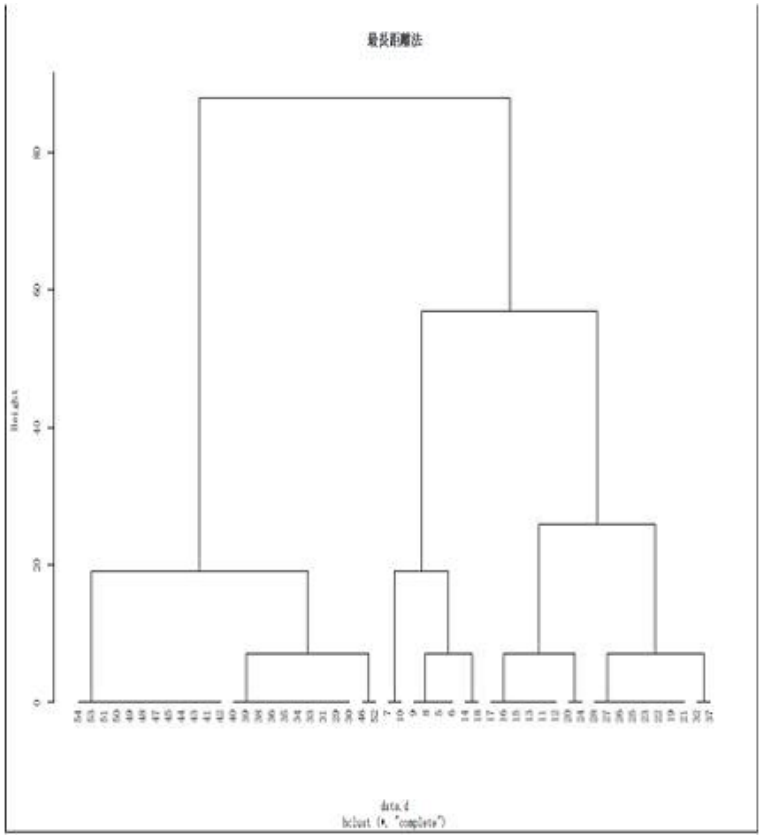


図 5.4: クラスター分析の適用例 1

なっていることが分かる。このことから、情報量を少なくすることで反例間の距離は短くなり、クラスターの数も減少することがわかった。クラスター分析を適用するためにはいかにして情報の抽出量を考えるか必要である。しかし、エラーの内容やエラーの箇所により区別を行っていないので、この結果からエラーの内容を把握することはできない。3 通りの実験を行った結果、エラーに至るまで情報がこの表示結果からは読み取ることができないので、ステップ数に注目しても反例の解析に有効でないことがわかった。しかし、バグを埋め込む組合せにより反例を検出するモデル数が異なること、反例間の距離が異なることからクラスター分析を適用してグラフィカルに表示することで視覚的に改善または悪化したかの判断が容易にできる利点があることがわかった。
また、何百という大量の反例が検出されたとき、それらをクラスター分析に適用すると図 5.7 のようになる。図 5.7 のように横軸の検査モデル No の字がつぶれてわかにくくなってしまっている。このことから、1 回のクラスター分析に適用できる最大利用モデル数は、約 50 個までが限界となる。よって、全ての反例を調べるために、クラスター分析を階層的に行うか、ランダムに反例を 50 個ずつ抽出してクラスター分析を適用する方法が挙げられる。この他にも有効な方法が考えられるがどの手法が一番最適な方法であるか検討を行う必要がある。

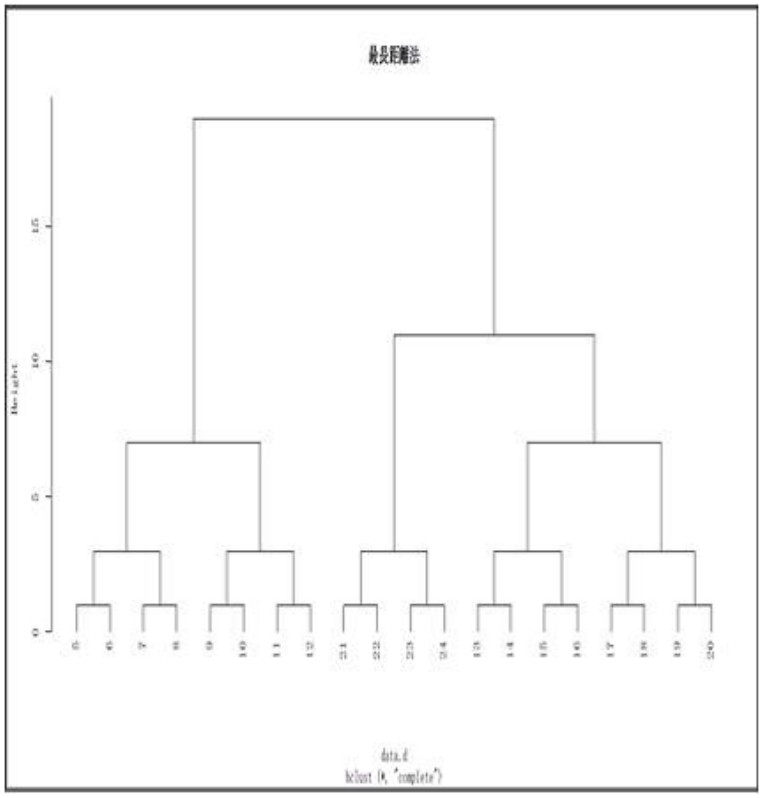


図 5.5: クラスター分析の適用例 2

5.5 反例解析の実験方法 2

実験 1 ではステップ数に注目してクラスター分析を適用した。しかし、エラーの内容ごとにクラスター化を行うためには情報が不十分であるため区別できているか判断することが難しいことがわかった。今回の実験は、より詳細にエラーの内容ごとに区別をおこなうため反例の内容を使ってクラスター分析を適用する。反例を解析する前にいくつかの手順を行ってから解析を行う。反例解析の手順を図 5.8 に示す。

1. 検査対象にいくつかバグを埋め込み、各環境の検査モデルのバリエーションにおいてモデル検査を行う。
   今使用している検査対象となるモデルは正しく動くとされているので意図的にバグを埋め込んでおく。こうすることで各検査モデルにおいてモデル検査を行ったとき反例が検出される。バグの埋め込むとき何通りかの組合せて検査を行い、どのような方法でエラーを区別すればよいか検討を行う。

2. 検出された反例を grep コマンドを用いて必要な情報を抽出する。この他に、検出した反例には、いろいろな情報が格納されている。図 5.8 に反例の例を示す。図 5.8 の 3 行目、5 行目はタスクを生成したときのサービスコール、7 行目以降はサービス

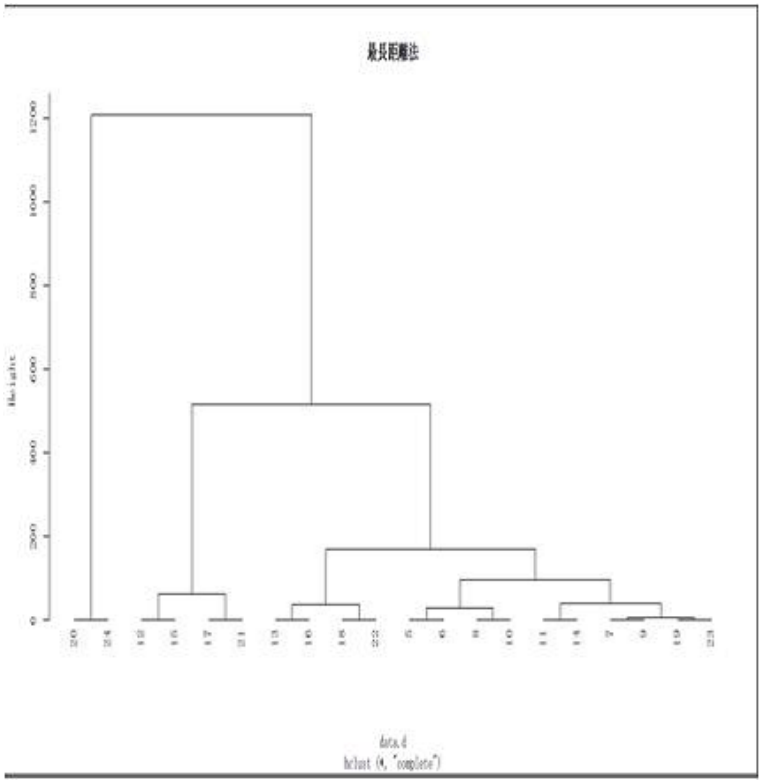


図 5.6: クラスター分析の適用例 3

コールの実行列である。また、各サービスコーの右端にはタスクや資源の ID、タスクや資源の優先が記述されている。反例の最後はエラーの内容が記述されている。検査したいシステムの動作状況などが出力される。これらの情報の中からどのようにして各々の反例の差分を求めるか問題である。

3. grep コマンドで抽出した情報を用いて diff コマンドを用いて反例間の差分を求める。

4. 得られた差分から反例間の距離を求める。

次に、差分結果から反例間の距離を求める方法について説明する。

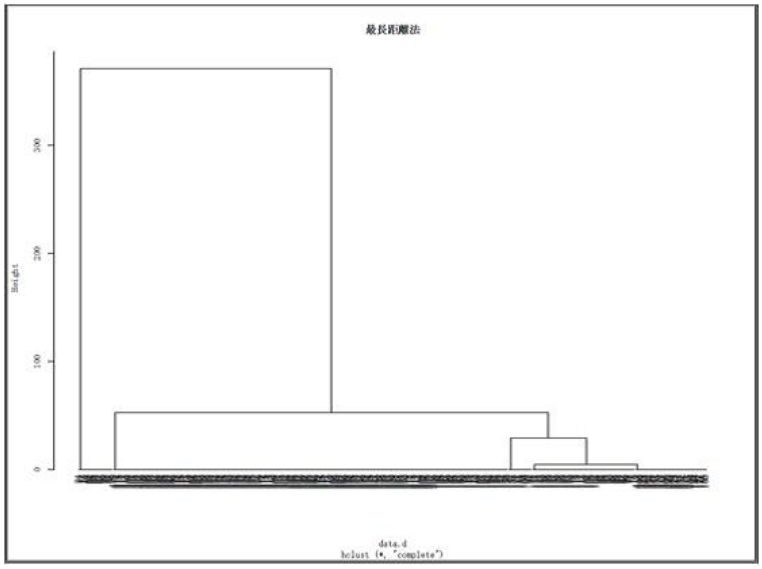


図 5.7: 大量のデータをクラスター分析に適用した例

```
 1  if( $line =~ /^hanrei/ ) {
 2  $name=$line;
 3  }elsif( $line =~ /^</ ) {
 4  $x=$x+1;
 5  }elsif( $line =~ /^>/ ) {
 6  $y=$y+1;
 7  }elsif( $line =~ / / ) {
 8  #<と>の数が等しい場合
 9  if ($x == $y){
10  $kyori=$x;
11  }#<の数が>の数より多い場合
12  elsif ($x > $y){
13  $kyori=$x;
14  }#<の数が>の数より少ない場合
15  elsif ($x < $y){
16          $kyori=$y;
17  }
```

差分結果には「<」と「>」が出力される。

- <：前者のファイルに記述されている文字列が後者のファイルに存在しない
- >：後者のファイルに記述されている文字列が前者のファイルに存在しない

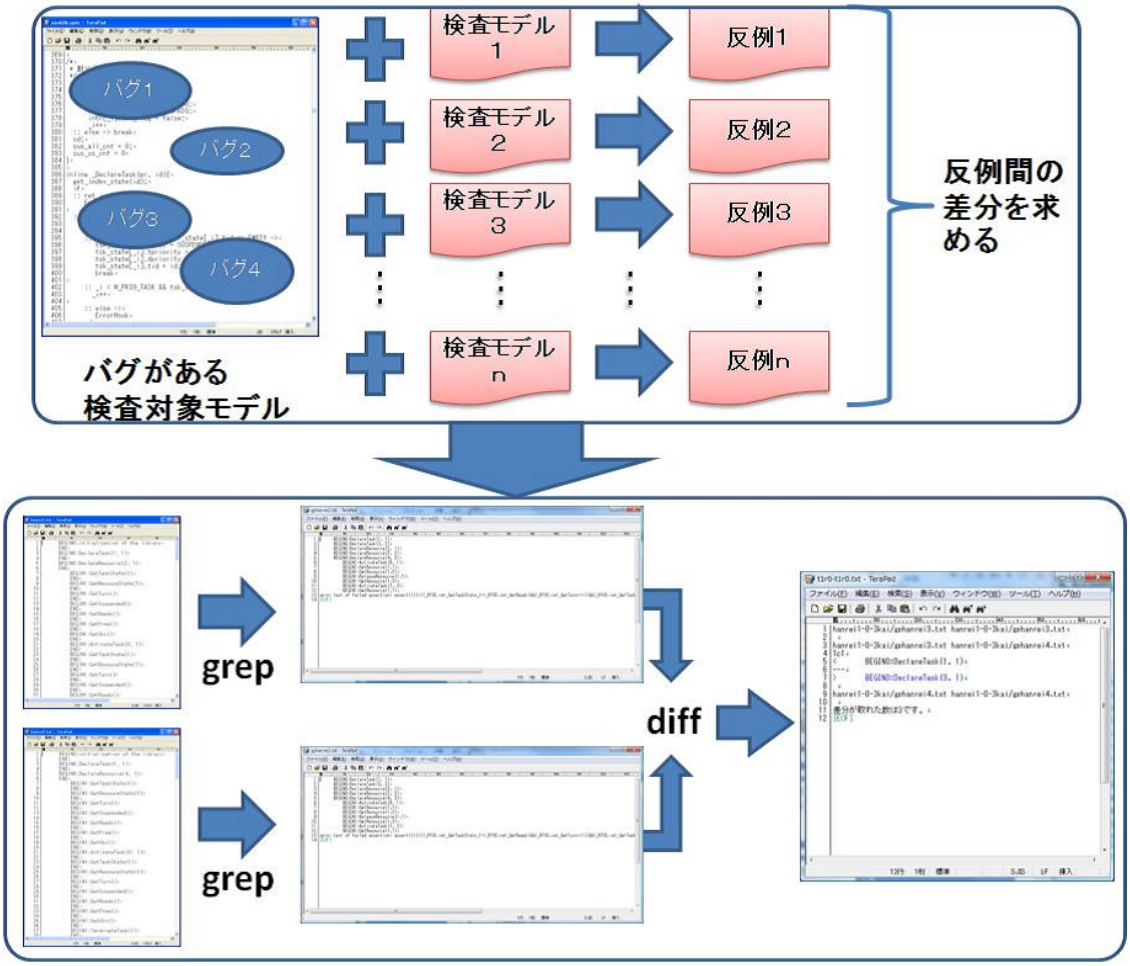


図 5.8: 反例解析の実験手順

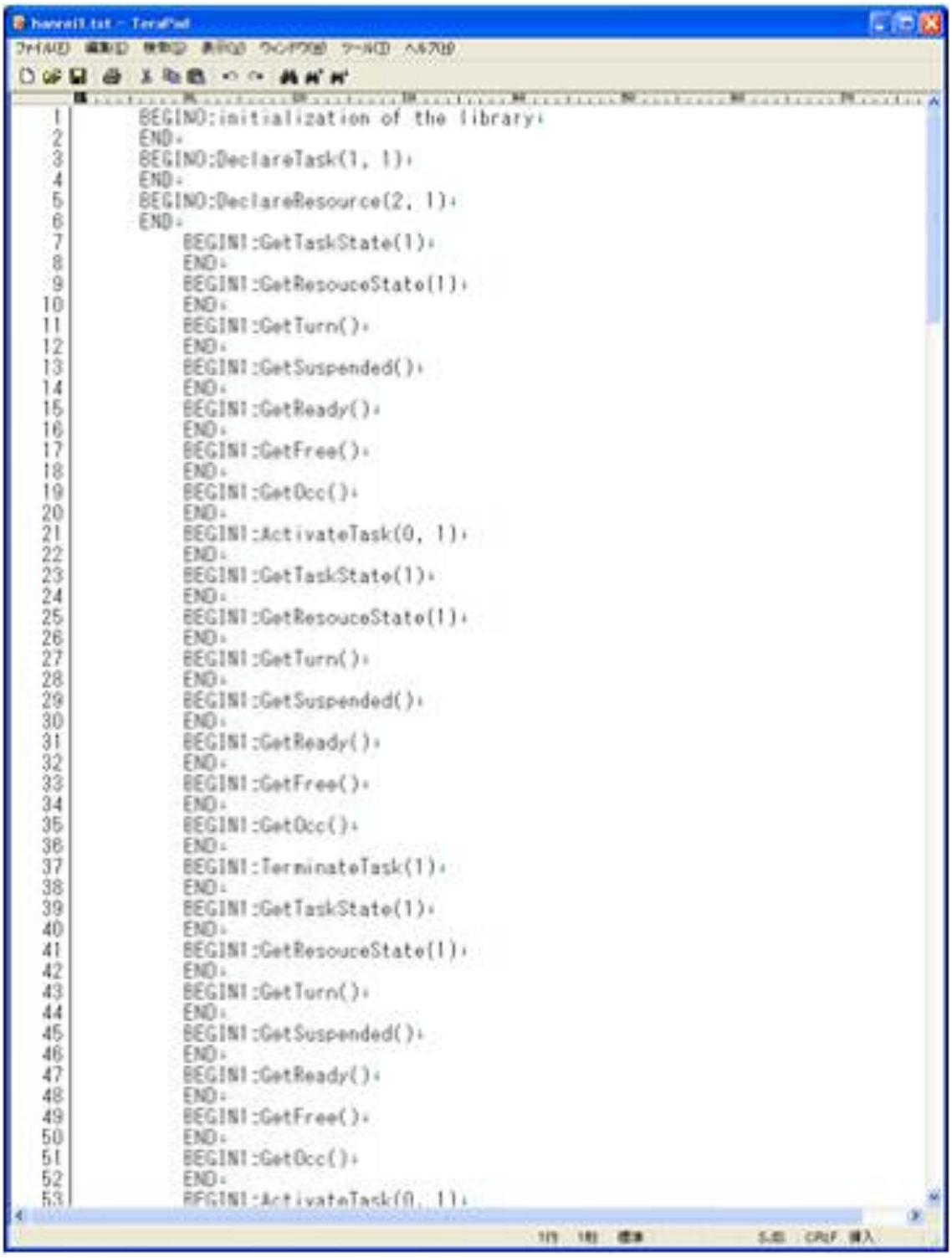

BEGIN:initialization of the library
END
BEGIN:DeclareTask(1, 1)
END
BEGIN:DeclareResource(2, 1)
END
BEGIN:GetTaskState(1)
END
BEGIN:GetResouceState(1)
END
BEGIN:GetTurn()
END
BEGIN:GetSuspended()
END
BEGIN:GetReady()
END
BEGIN:GetFree()
END
BEGIN:GetOcc()
END
BEGIN:ActivateTask(0, 1)
END
BEGIN:GetTaskState(1)
END
BEGIN:GetResouceState(1)
END
BEGIN:GetTurn()
END
BEGIN:GetSuspended()
END
BEGIN:GetReady()
END
BEGIN:GetFree()
END
BEGIN:GetOcc()
END
BEGIN:TerminateTask(1)
END
BEGIN:GetTaskState(1)
END
BEGIN:GetResouceState(1)
END
BEGIN:GetTurn()
END
BEGIN:GetSuspended()
END
BEGIN:GetReady()
END
BEGIN:GetFree()
END
BEGIN:GetOcc()
END

図 5.9: 反例の例

上記の「<」と「>」をそれぞれ数え、個数が多い方を反例間の距離とする。図 5.9 に反例間の距離を出力した例を示す。距離の結果は、差分を求めた 2 の反例のファイル ID と反例間の距離で出力する。反例ファイルの ID は「タスク数」、「資源数」検査モデル ID の順に表示している。図 5.9 の 1 行目を例に挙げるとタスク数が 2、資源数 0 の検査モデル No2 の反例とタスク数が 2、資源数 0 の検査モデル No5 の反例の差分を求めている事を示す。この結果を用いてクラスター分析に適用する。

5.6 反例からの距離の求め方

反例同士の距離を求めるために反例のどの情報から抽出するかという問題がある。そこで、どの情報が反例同士の距離を求めるために有用であるか調べる。

5.6.1 実験 1

モデル検査を行い、反例を検出させるため設計モデルに埋め込むバグは以下である。

1. システムコール ActivateTask において enque 処理の手順を変更
2. TerminateTask の deque 処理の省略
3. TerminateTask で行う ActivateTask の発行を省略
4. 資源獲得の際に行うタスクとリソース間で優先度比較を変更

- パターン 1：1，2，3，4 のバグを埋め込む
- パターン 2：2,4 のバグを埋め込む
- パターン 3：4 のバグを埋め込む

それぞれの組合せで検査した結果を表 5.3、表 5.4、表 5.5 に示す。表は、タスクの数と資源の数ごとの環境において分母は環境での検査モデルの数、分子は反例を検出した数である。パターン 1 の結果より各環境の検査モデル全てにおいて反例を検出したので解析を行わなくても同一のエラーである可能性がある。パターン 2 は、資源の数が 0 の環境において反例は検出されなかった。資源の数が 0 のとき反例を検出した理由は、環境モデルの不備により検出された。今回は、資源の数が 1 以上の場合に焦点を当てると資源の数が 1 のときと 2 のときの境界線でエラーに起因することが隠れていると考えられる。パターン 3 では、資源の数が 0 のとき反例を検出している。これはパターン 2 と同様に環境モデルの不備により検出された反例である。タスクの数が 1、資源の数が 5 の環境は反例が検出されていないことからタスクの数が 1、資源の数が 4 の環境の間で資源の数が 1 つでエラーに関係していることが分かる。以上の結果よりこの実験ではクラスター分析を適用しなくても解析できることが分かった。

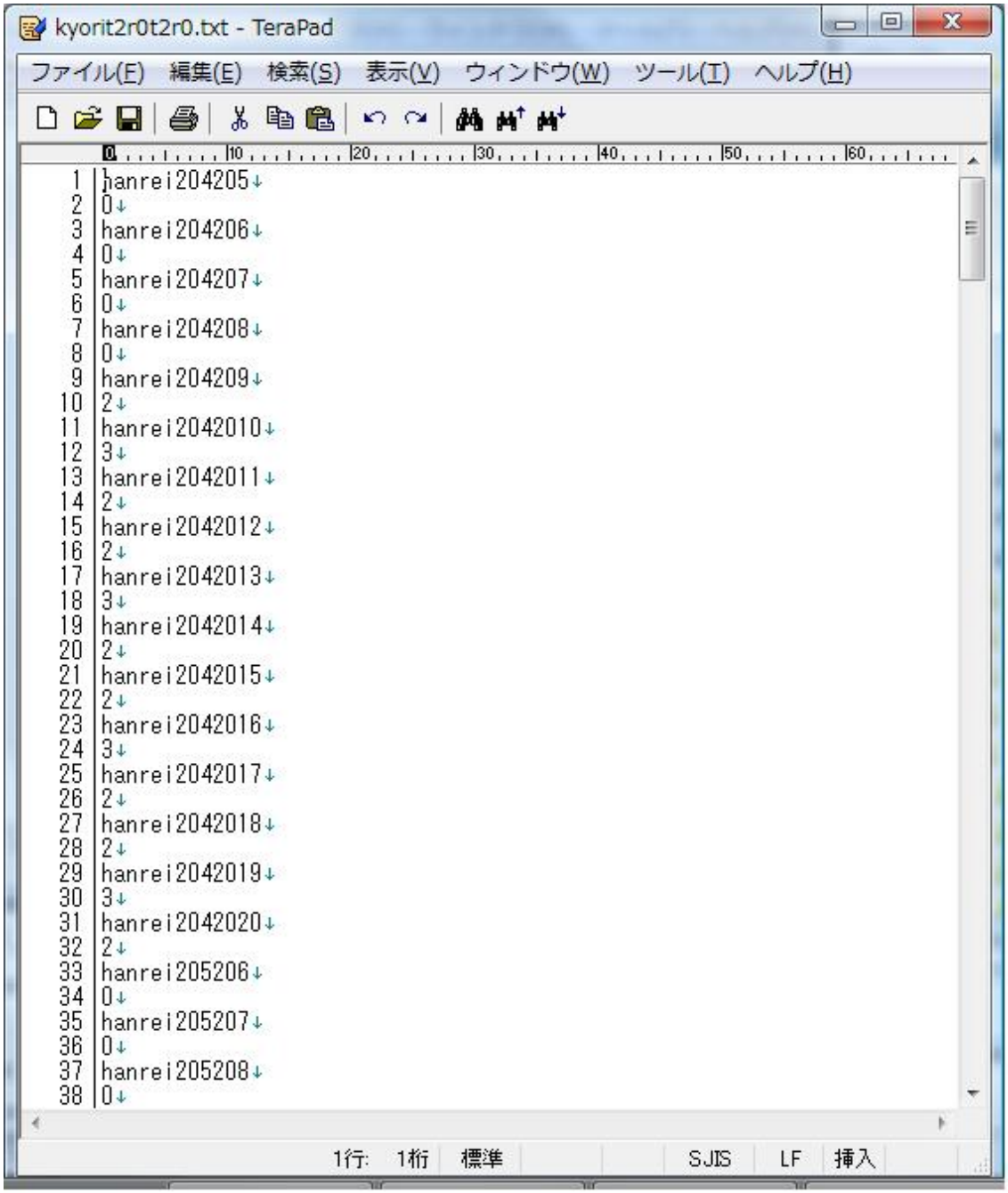


図 5.10: 反例間の距離の結果の例

表 5.3: パターン 1 の検査結果

		タスクの数	
		1	2
資源の数	0	4/4	20/20
	1	6/6	74/74
	2	8/8	272/272
	3	10/10	930/930
	4	12/12	3140/3140
	5	14/14	

表 5.4: パターン 2 の検査結果

		タスクの数	
		1	2
資源の数	0	4/4	20/20
	1	0/6	0/74
	2	2/8	46/272
	3	4/10	308/930
	4	6/12	
	5	14/14	

5.6.2 実験 2

検査対象モデルに埋め込んだバグを以下に示す。

- システムコール ActivateTask において Task が Redy 状態の場合でも enque 処理を行う。
- ChainTask の ActivateTask 処理を省略した。
- タスクがリソースを獲得するときタスクの実行優先度よりリソースの優先度が低い時、タスクの実行優先度はリソースの優先度に変更されるがタスクの実行優先度をタスクの優先度に変更する。

表 5.6 に上記のバグを埋め込んだ時検出した反例の数を示す。表 5.6 は、タスクの数と資源の数ごとの環境において分母は環境での検査モデルの数、分子は反例を検出した数である。5.6 より、各環境の検査モデルのうちいくつか反例を検出したことがわかる。タスクの数が 1 の時の環境をみるとそれぞれ検査モデルの半分でエラーを検出したことが分か

表 5.5: パターン 3 の検査結果

		タスクの数	
		1	2
資源の数	0	4/4	20/20
	1	0/6	0/74
	2	2/8	46/272
	3	4/10	
	4	6/12	
	5	0/14	

る。これより、しらみつぶしに調べれば傾向を見つける事ができる。しかし、反例を解析の問題点があり環境の規模が大きくなると反例を検出したモデルがある程度の数あるため全てを解析することは難しい。また、どのような要因でエラーを検出したかわからないので 5 章のような検査結果を表で見せても直感で理解することは困難である。この結果を用いてクラスター分析の適用方法を考える。

表 5.6: 実験 2 の検査結果

		タスクの数		
		1	2	3
資源の数	0	2/4	17/20	139/140
	1	3/6	64/74	1011/1014
	2	4/8	237/272	7919/7928
	3	5/10	815/930	
	4	6/12	2754/3140	
	5	7/14		

5.6.3 実験 3

検査対象モデルに埋め込んだバグを以下に示す。

- ChainTask の ActivateTask 処理を省略した。
- タスクがリソースを獲得するときタスクの実行優先度よりリソースの優先度が低い時、タスクの実行優先度はリソースの優先度に変更されるがタスクの実行優先度をタスクの優先度に変更する。

表 5.7: 実験 3 の検査結果

		タスクの数		
		1	2	3
資源の数	0	2/4	17/20	139/140
	1	3/6	64/74	1011/1014
	2	4/8	237/272	7019/7928
	3	5/10	815/930	
	4	6/12	2627/3140	
	5	7/14		

表 5.7 に上記のバグを埋め込んだ時検出した反例の数を示す。表 5.7 は、タスクの数と資源の数ごとの環境において分母は環境での検査モデルの数、分子は反例を検出した数である。表 5.7 より、各環境の検査モデルのうちいくつか反例を検出したことがわかる。タスクの数が 1 の時の環境をみるとそれぞれ検査モデルの半分でエラーを検出したことが分かる。これより、しらみつぶしに調べれば傾向を見つける事ができる。しかし、反例を解析の問題点があり環境の規模が大きくなると反例を検出したモデルがある程度の数あるため全てを解析することは難しい。また、どのような要因でエラーを検出したかわからないので 5 章のような検査結果を表で見せても直感で理解することは困難である。

5.6.4 反例結果からエラーのバリエーションごとに区別する。

検査対象モデルにバグを埋め込み、意図的に反例を出力した。この反例を用いてどのようにしてエラーのバリエーションごとに区別すればよいかいろいろな方法を適用して検討する。実験に使用する反例は、実験 2 で得た検査結果を用いる。実験 2 の検査結果を使用する理由は、各環境における全体の検査モデルの数に対して検出された反例が異なるからである。この表からどのようなエラーが隠されているか分からないので実際にクラスター分析に適用して区別可能か調べる。実験は、反例からいろいろな組合せで情報を抽出し、差分を求める。この結果をクラスター分析に適用したときエラーのバリエーションごとに表示できるか調べる。

実験 1 反例の結果をそのまま利用する

検出される反例は、図 5.9 のようにタスクを生成するサービスコール、タスクや資源を操作するサービスコール、タスクや資源の ID、タスクや資源の優先度など多くの情報がある。この実験では、これらを全て載せた状態で反例間の差分を求める。ただし、反例結果の下の行には、エラーになったときのタスクや資源の状態、タスクや資源の優先度の情

報が表示されているが今回の実験では、無視する。理由は、エラーに至るまでの情報から反例間の距離を求めたいので、エラーになったときの情報を考慮する必要はないと考えたからである。以下はタスク、資源を生成するサービスコール、タスク、資源を操作するサービスコール、エラーの内容を抽出したスクリプトの例である。

```
1        BEGIN0:DeclareTask(1, 1)
2        BEGIN0:DeclareResource(2, 1)
3            BEGIN1:ActivateTask(0, 1)
4            BEGIN1:ChainTask(1, 1)
5  spin: text of failed assertion: assert((((_RTOS.ret_GetTaskState_1==_RTOS.ret_Ge
```

1 行目、2 行目はタスク、資源の生成を行っている。括弧内の左側は優先度、右側は ID を表している。3 行目、4 行目はエラーに至るまでタスクや資源を操作するサービスコールの実行列である。5 行目は、エラーの内容を表している。各反例において上記のように抽出し、反例同士の差分を求める。図 5.11 に差分結果の例を示す。図 5.11 よりタスク、資源を生成するサービスコールで優先度が異なることで差分が検出されている。同様にタスクや資源の優先度が同じとき、ID が異なる場合でも差分が検出されることが言える。また、図よりタスク、資源を操作するサービスコールの実行列やエラーの内容においての差分は検出されていない。これは、同じ実行列でタスクや資源を操作し、同じエラーに至っていることがわかる。しかし、この実験の目的はエラーのバリエーションによって反例を分けることなので、タスクや資源を生成するサービスコールにおいて差分が検出されるとより細かなグループを形成するので問題を把握しずらくなる。また、この手法で行ったとき問題が 1 つある。図 5.12 に実際に差分を求めた結果を示す。図 5.12 は、外部環境がタスクの数が 2、資源の数が 2 のときの反例同士で差分を求めた結果の一部である。図よりタスクや資源を生成するサービスコールによる差分、タスクや資源を操作するサービスコールなどによる差分、エラーの内容での差分をを検出している。しかし、エラーの内容は異なる場合があるのでエラーの内容も差分に含めると距離の一つとして数値に変換されるので正確なグループ化を行うことができない。そのため、差分解析を用いるときはエラーの内容を除き、他の情報で差分を求める必要がある。

実験 2 反例からタスク、資源を生成するサービスコールとタスク、資源を操作するサービスコールの実行列を抽出する

実験 1 よりエラーの内容を含んで差分を求める方法は、エラーのバリエーションごとにグループ化を行うことはできない。次に反例の結果からタスク、資源を生成するサービスコールとタスク、資源を操作するサービスコールの実行列を抽出する。以下はタスクや資源を生成するサービスコール、タスクや資源を操作するサービスコールを抽出したスクリプトの例である。

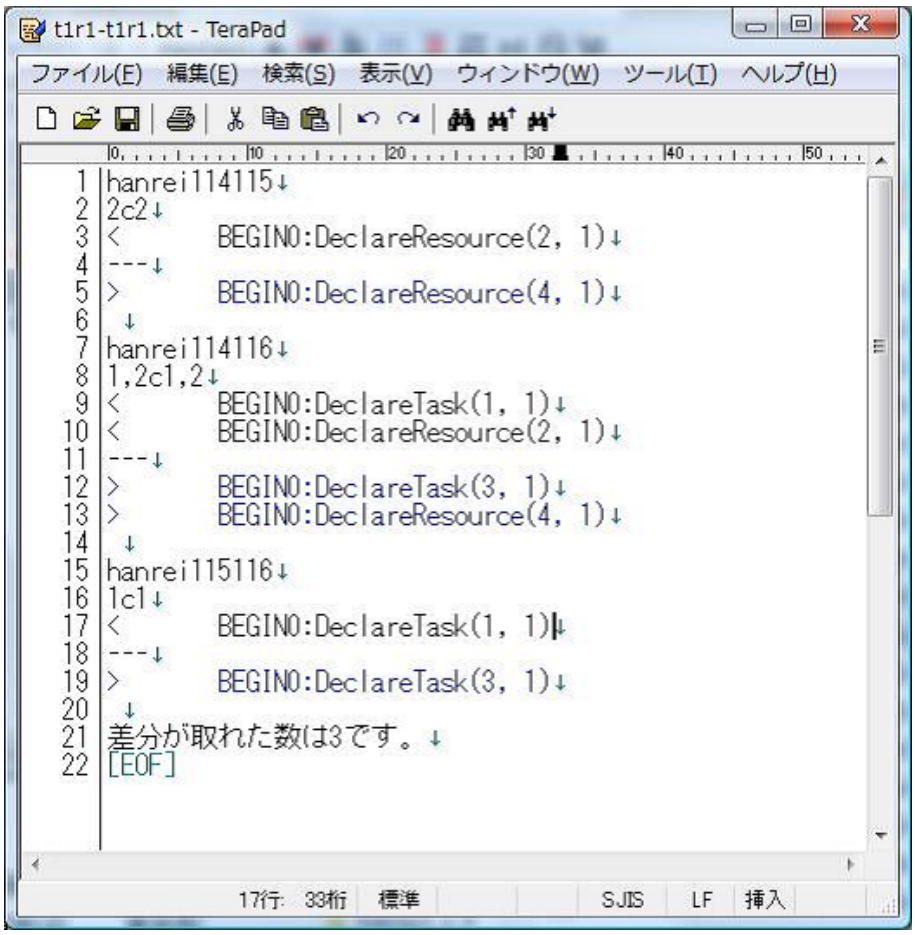


図 5.11: 差分結果の例 1

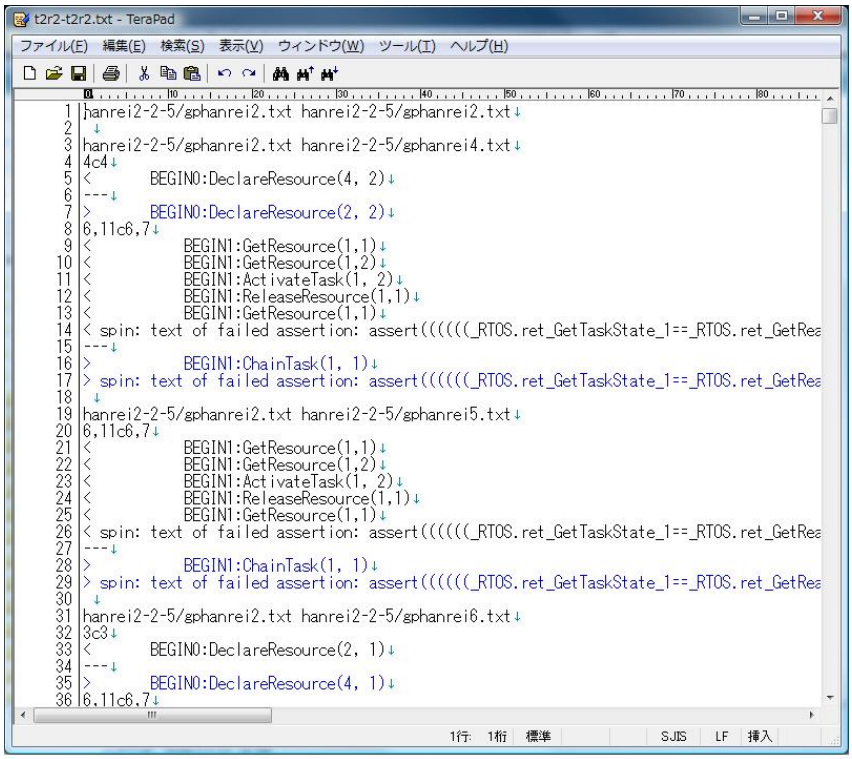


図 5.12: 差分結果の例 2

```
1         BEGIN0:DeclareTask(1, 1)
2         BEGIN0:DeclareResource(2, 1)
3             BEGIN1:ActivateTask(0, 1)
4             BEGIN1:ChainTask(1, 1)
```

1 行目と 2 行目は、タスクを生成する。サービスコールである。4 行目と 5 行目はタスクや資源を操作するとき発行されるサービスコールである。各反例において上記のように抽出し、反例同士の差分を求める。図 5.12 にタスクに数が 1、資源の数が 2 の環境における反例同士をクラスター分析に適用した結果を示す。図 5.12 の横軸は差分を求めた反例ファイル名を記述している。縦軸は、反例間の距離を表している。横軸のファイル名を実数で表現している理由は、使用しているソフトが文字列に対応していないので、タスクの数、資源の数、反例を検出した検査モデルのファイル ID を使って記述した。図 5.13 は、タスク、資源の優先度から距離が求められたので反例は、優先度の組合せでクラスター化された。今回の実験では、エラーの内容を考えず差分を検出し、クラスター化を行うことができた。このことから、同一のエラーを含むモデルを用いてより詳細にクラスター化できるという利点があることがわかった。

実験 3 反例の結果からタスク、資源を操作するサービスコールの実行列を抽出する

実験 2 においてより詳細にクラスター化できることがわかった。今回の実験は、タスクや資源を生成する実行列も省略し、タスクや資源を操作する実行列のみに注目して差分を求め、クラスター分析に適用する。

```
1             BEGIN1:ActivateTask(0, 1)
2             BEGIN1:ChainTask(1, 1)
```

上記のように反例からタスク、資源のサービスコールのみを抽出する。各反例で抽出した実行列を diff コマンドで差分を求め、クラスター分析に適用する。図 5.14 は、タスクの数が 1、資源の数が 1 の環境においてタスク、資源のサービスコールのみ抽出した実行列にクラスター分析を適用した結果である。他の結果は付録 D を参照。図 5.14 より、実行列が全く同じであるため距離が 0 となり 1 つのクラスターを形成した。タスクの数が 1 の環境では全て同じ結果になる。タスクの数が 2 より多くなると実行列により差分が検出され、いくつかのクラスターを形成する。実験 2 と比較するとクラスターの数は、さらに少なくなったことがわかる。理由は、抽出した情報が少なくなったことで反例同士の距離が短くなったのでクラスターになる可能性が大きくなったからである。しかし、生成される検査モデルは構造的に分割されているのでタスクや資源を生成する。実行列を考えなくてもよいと考えられる。また、差分を求めるために抽出する情報が少ないので実験を行いやすいこと、検討を行いやすいことがあげられるのでこの方法を適用することも有効である。

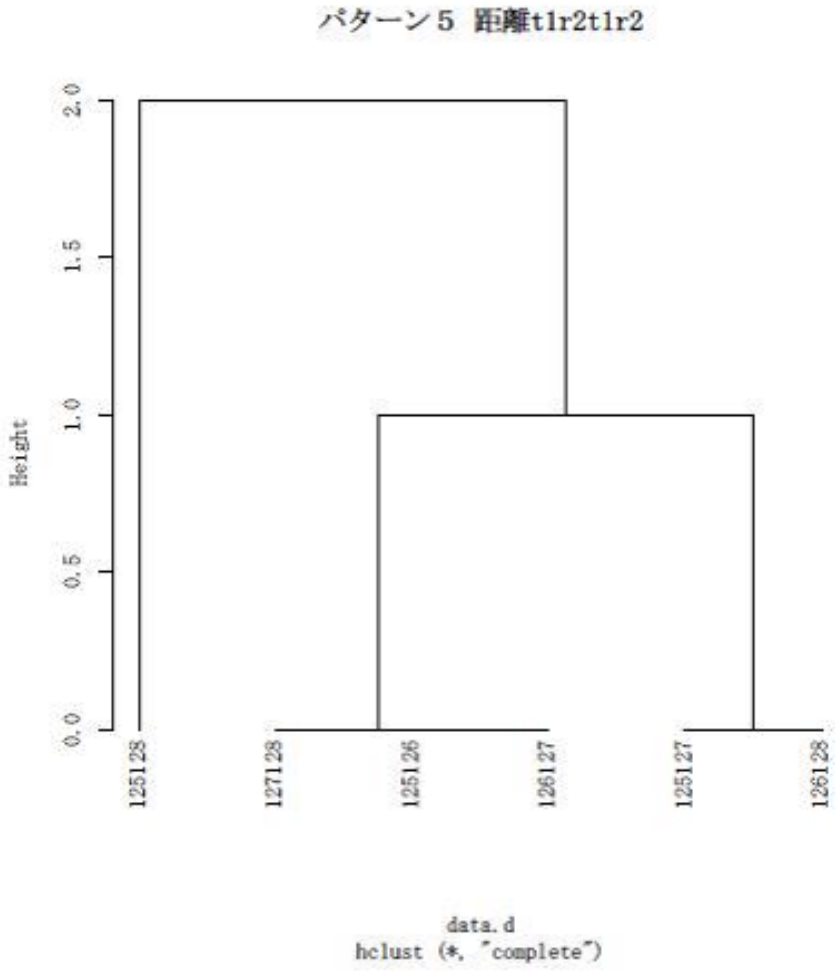


図 5.13: クラスター分析の適用結果

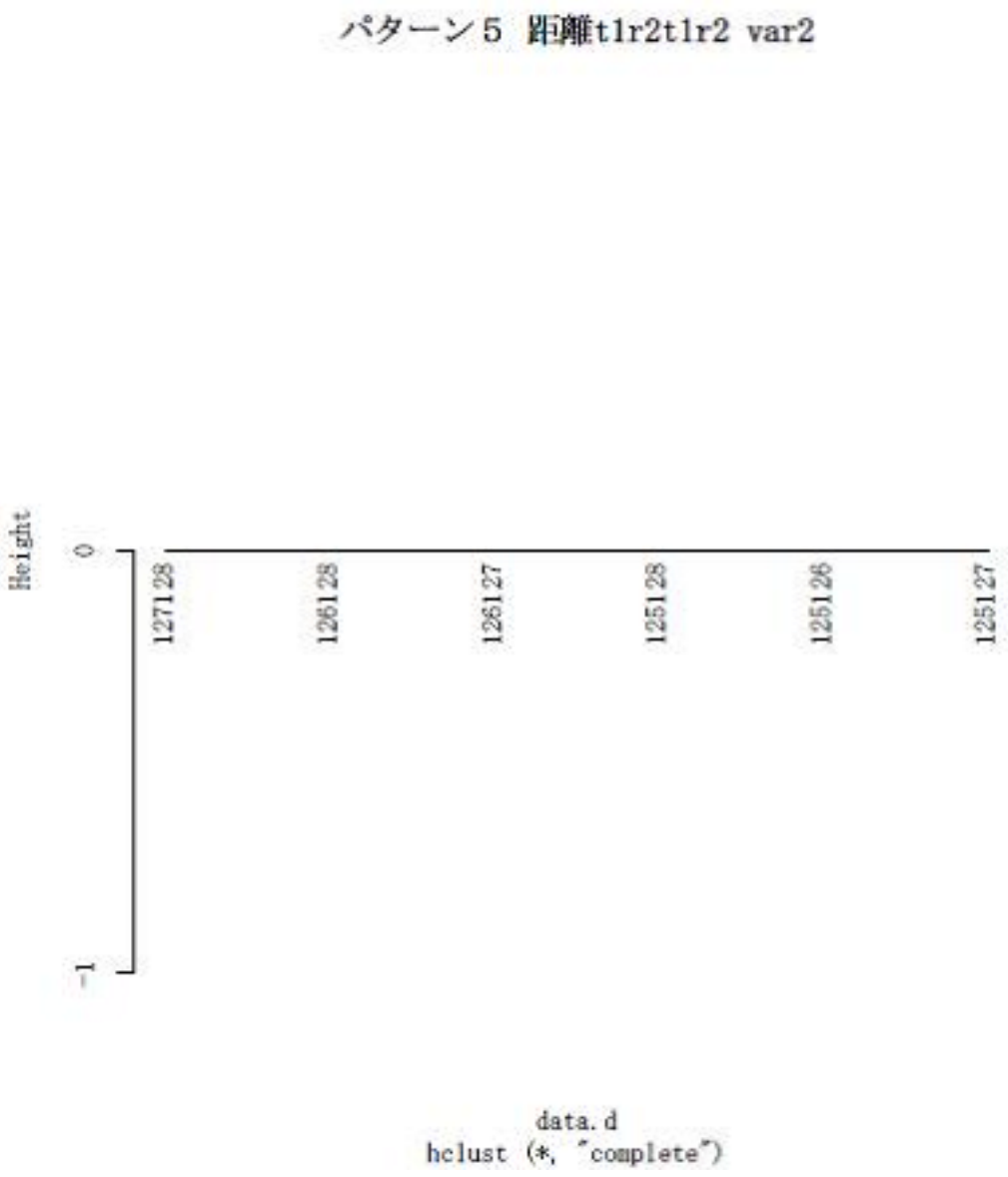


図 5.14: クラスター分析の適用結果

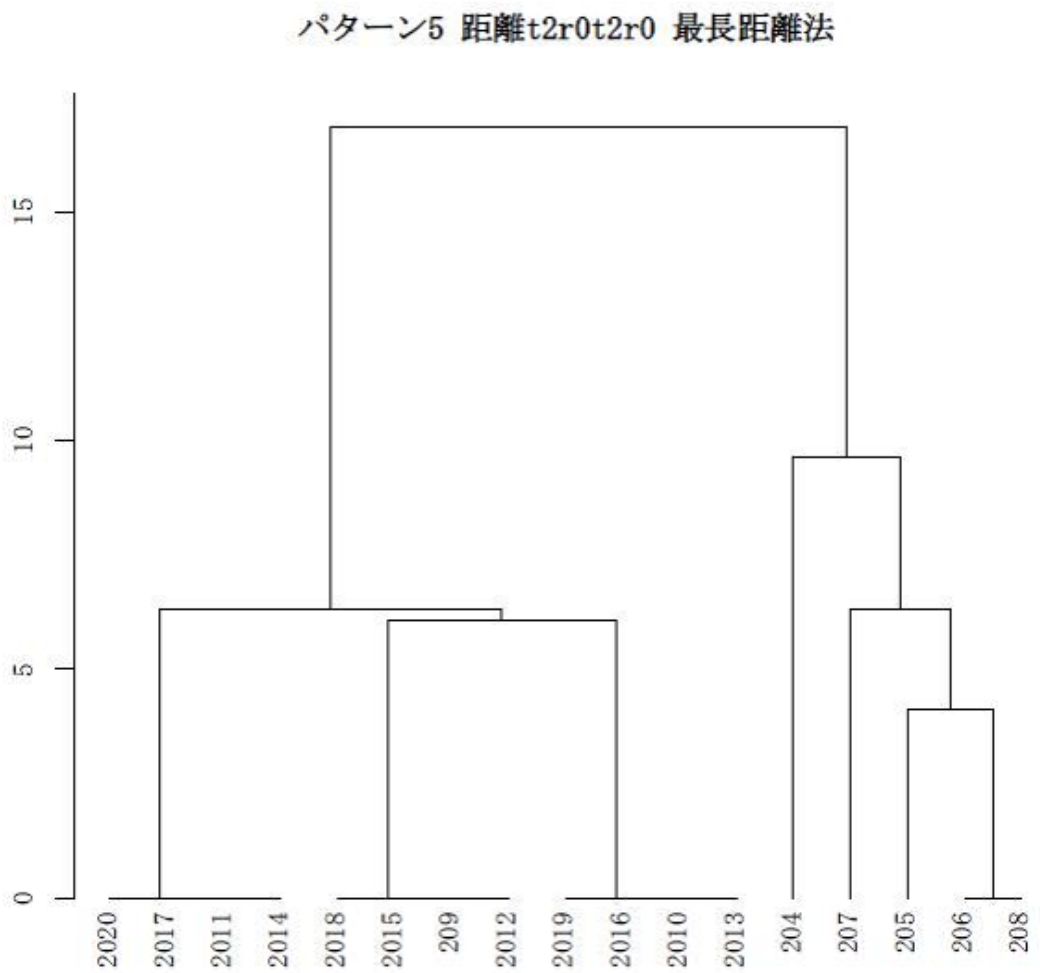


図 5.15: クラスター分析の適用結果

実験 4 クラスター分析の適用方法の変更

これまでの実験においてクラスター化をすることが可能であることがわかったがエラーの分類に適用できるかわからない。そこで、クラスター分析の適用方法を変更した。各反例同士の距離を求め、クラスター分析に適用した。**??**に結果を示す。図 5.15 より、各反例間でクラスターごとにまとめる事が出来た。この図を上位から調べ、埋め込んだバグを発見する。まず、距離が約 16 当りで 2 つのクラスターに分類されている。それぞれの区クラスターから 1 つの反例を抽出し、埋め込んだバグであるか調べる。もし、ない場合は距離が 10 当りまで進む。ここでクラスターが 3 つになる。この 3 つからそれぞれ抽出し、埋め込んだバグがあるか調べる。この結果、埋め込んだバグを発見すること判断できなかった。しかし、似た反例をクラスター化してまとめることで検討にかける時間を短縮できることは分かった。

5.6.5 考察

図よりクラスター分析を適用することでエラーの内容ごとにグループ化を行うことができた。また、クラスター分析の特徴であるグラフィカルな表示方法により反例のグループを一目で理解できるようになった。ただし、課題としてより大きな規模の検査モデルにおいてモデル検査を行う場合、検査モデルの数が多くなることでエラー内容の種類が増大する問題がある。そのため、エラーの種類が追加されるのでその都度対応付けを行う必要がある。今回は、エラーの内容ごとに区別する方法を提案した。しかし、同一のエラーにおいてサービスコールの実行列が異なる場合がある。今回の方法のように反例間の差分を距離とし、クラスター分析に適用すればグループ化ができると考えられる。いくつかの実験より、タスクや資源を操作する実行列のみ抽出して差分を求めることで実行列においてもクラスターを形成できることがわかった。問題は、反例にはタスクや資源の ID、優先度の情報が入っているので、これらが差分にどれだけ関係しているか実験を行い、検討を行う必要がある。また、今回は、小規模の環境を利用してクラスター分析を適用できるか調べた。大きい環境においては検討の余地があるので実験を行ってみてどのような結果になるか調べる必要がある。この他に、エラーのバリエーションごとにクラスター分析を適用する方法と同一のエラーにおいてクラスター分析を適用する方法が見出されたのでこの両者を 1 元で行える方法を考える事が今後の課題となる。

第6章　まとめ

6.1　研究のまとめ

本研究では、大規模の計算機環境を用いてモデル検査を行うときの検証手法について提案し、実験を行って評価した。大規模の計算機環境を用いてモデル検査を行うときの問題点である検査モデルを割り振る方法では、検査モデルからランダムに抽出して検査モデルの検査時間を求め、実際に検査モデルを割り振って実験を行った。また、検査モデルのスクリプトの行数に注目してそのときのコンパイル時間を調べた。行数とコンパイル時間の関係から近似式を求め、近似式に基づいて検査モデルを割り振る。そして実験を行い考察した。エラー内容を直感で理解できるようにするための表の提案をした。エラー内容をさらに詳細に区別するためクラスター分析を適用してグラフィカルに区別できる方法を提案し評価を行った。

6.2　今後の課題

大規模の計算機環境に検査モデルを割り振る方法は、ある程度均一な時間で検査が行えることがわかった。しかし、検査モデルの一部には検査時間にかなりの時間を要するモデルも混在している。したがって近似式をそのまま使用しても若干のばらつきが生じる。この問題を解決するためには検査モデルの割り振り方法の精度を高める必要がある。

表の見せ方では、一部の結果に適用してみて直感でわかることが示されただけである。検査モデルの数が多くなるとそれだけ表示する情報も多くなるのでどのように解決するか検討する必要がある。また、今回はタスクと資源の優先度、タスクと資源の参照関係に着目して表を作成した。他に表現方法が考えられるか検討する必要がある。同一の環境においての検査結果表しか考えていない。差分解析を行うためにはタスクの数や資源の数が異なる検査モデル同士を比較・検討する必要があるので異なる環境間での差分解析を行える表示方法を考える必要がある。表の見せ方は、提案にとどまったため実装を行う必要がある。

クラスター分析を用いたエラーの区別方法は、エラーの内容において区別する方法を提案し評価を行ったが同一のエラーにおいて区別する方法はタスク、資源を操作するサービスコールに着目して行えばクラスター分析は適用可能である。ただし、タスクや資源に

は ID や優先度が含まれてるのでこの点を考慮した時は、また別の方法が考えらる。「タスクと資源の ID と優先度無視して同じ名前のサービスコールを発行したときは同一である。」と決めて行う方法も考えれる。この他にもクラスター分析を適用する方法があるか検討することが課題である。

謝辞

本研究を進めるあたり、ご指導賜りました青木利晃准教授に厚くお礼を申し上げます。また、有益な助言を頂いた同大学院 二木厚吉教授、千葉勇輝助教授、矢竹健朗特任助教授に感謝の意を示します。
また、本研究を進めるために、相談にのっていただいた、青木研究室　土肥雅俊さん、岸研究室　金井勇人さん、細合晋太郎さんに感謝いたします。最後に研究に関する議論にも応じてくださったデファゴ研究室、岸研究室、二木研究室、青木研究室の皆さんにも感謝いたします。

参考文献

[1] 萩原昌己　吉岡信和　青木利晃　田原康之　共著, SPIN による設計モデル検証, 近代学者 2008

[2] [連載] フリーソフトによるデータ解析・マイニング第 28 回R とクラスター分析 (1)

[3] http://portal.osek-vdx.org/.

付録

自動的にモデル検査を行うソースコード

```
#!/bin/sh

#調べるモデルのタスク、リソース数を指定する
echo "検査開始時のタスクの値を入力ください。"
read j
echo "検査終了の検査モデルのタスクの値を入力してください。"
read tasklimit
echo "検査開始時のリソースの値を入力してください。"
read z
echo "検査終了の検査モデルのリソース値を入力してください。"
read resourcelimit
echo "検査開始時の検査モデルの ID を入力してください。"
read x
echo "終了するときの検査モデルの ID を入力してください。"
read y
echo "反例を保存するディレクトリの作成"
read g

#CTRL+C で強制終了させる
stop=
trap 'stop=1; trap 2 3' 1 2 3 15
#trap 'echo "trapped."; exit 1' 1 2 3 15

#各検査モデルをモデル検査する。
while :
do
  if test -d caset"$j"r"$z"
  then
      #echo "Verification of Next Model is caset"$j"r"$z""
      cd caset"$j"r"$z"
```

```
      i=$x
      rm -r cheack
      rm -r result$j-$z-$g
      rm -r /home/users/h-togawa/result/result"$j"-"$z"-"$g"
      mkdir cheack
      mkdir result$j-$z-$g
      while test -f case"$i".spin
      do
          if [ $stop ]
          then
             cp -r ./result"$j"-"$z"-"$g" /home/users/h-togawa/result/
             echo "Now Checking Model is caset"$j"r"$z""
             cd ..
             break 2
          fi

  #START=‘date +%s‘
  /usr/bin/time -p -o ./result"$j"-"$z"-"$g"/panlog"$i".txt spin -a -DNATOMIC
case"$i".spin
          #spin -a case"$i".spin
  /usr/bin/time -p -o ./result"$j"-"$z"-"$g"/compilelog"$i".txt gcc -DSAFETY
pan.c -o pan
  #END=‘date +%s‘
  #compiletime=‘expr $END - $START‘
  #START=‘date +%s‘
          /usr/bin/time -p -o ./result"$j"-"$z"-"$g"/verifylog"$i".txt ./pan
> ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$i".txt
  #END=‘date +%s‘
  #pantime=‘expr $END - $START‘
  #totaltime=‘expr $compiletime + $pantime‘
  line=‘wc case"$i".spin | awk ’{print $1}’‘
          #echo "case"$i".spin $line $compiletime $pantime $totaltime" >> ./result"$j
  chmod +x ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$i".txt
          #echo "case"$i" cheak ok" >> ./cheack/cheack.txt
          if test $j -eq $tasklimit
          then
            if test $z -eq $resourcelimit
            then
```

```
              if test $i -eq $y
              then
                 cp -r ./result"$j"-"$z"-"$g" /home/users/h-togawa/result/
 ls -l ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$x".txt >> /home/users/h-togawa/result
 ls -l ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$y".txt >> /home/users/h-togawa/result
                 break 2
              fi
            fi
          fi
          i=‘expr $i + 1‘
          cheackcount=‘expr $cheackcount + 1‘
      done
      i=‘expr $i - 1‘
      cp -r ./result"$j"-"$z"-"$g" /home/users/h-togawa/result/
      ls -l ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$x".txt >> /home/users/h-togawa/r
      ls -l ./result"$j"-"$z"-"$g"/result"$j"-"$z"-"$i".txt >> /home/users/h-togawa/r
      x=1
      cd ..
      z=‘expr $z + 1‘
   else
     j=‘expr $j + 1‘
     z=0
 fi
done &
```

モデル検査の結果から状態数、探索した状態数の合計を求める

```
#print "検査結果のタスク数を入力する。\n";
chomp($a = <STDIN>);
#print "検査結果のリソース数を入力する。\n";
chomp($b = <STDIN>);
#print "検査結果のパターンを入力する。\n";
chomp($c = <STDIN>);

my $dirname="result" . "$a" . "-" . "$b" . "-" . "$c";

&traverse('.');

sub traverse {
#my $dirname = shift;
```

```
my $delim = '/';
opendir (DIR, $dirname) or die "$dirname: $!";
foreach my $entry (readdir(DIR)) {
next if ($entry eq '.');
next if ($entry eq '..');
if ($dirname =~ /[\/\\]$/) {
$delim = '';
}
my $filename = "$dirname$delim$entry";
if (-d $filename) {
&traverse($filename);
} else {
&do_file($filename);
}
}
closedir(DIR);
}

sub do_file {
my $filename = shift;
#print $filename, "\n";
return unless ( $filename =~ /result/);
open(FILE, $filename) or die "$filename: $!";
while ( my $line = <FILE>) {
if( $line =~ /State-vector \d+ byte/ ) {
@vector = split(/ /, "$&");
$subvector = "$vector[1]";
}
if( $line =~ /\d+ states, stored/ ) {
@state = split(/ /, "$&");
$substate = "$state[0]";
}
if( $line =~ /\d+ states, matched/ ) {
@match = split(/ /, "$&");
$submatch = "$match[0]";
}
if( $line =~ /\d+ transitions/ ) {
@transitions = split(/ /, "$&");
```

```
$subtransitions = "$transitions[0]";
}
if( $line =~ /\d+.\d+ memory usage/ ) {
#print"$line\n";
@memory = split(/ /, "$&");
$submemory = "$memory[0]";
#print "$submemory\n";
}
}
close(FILE);
$totalvector = "$totalvector" + "$subvector";
$totalcount = "$totalcount" + 1;
$totalstate = "$totalstate" + "$substate";
$totalmatch = "$totalmatch" + "$submatch";
$totaltransitions = "$totaltransitions" + "$subtransitions";
$totalmemory = "$totalmemory" + "$submemory";
}
$totalcount = "$totalcount" - 1;
print"$dirname の結果\n";
print"モデルの合計数$totalcount\n";
print"ベクターサイズの合計は$totalvector byte です。\n";
print"生成された状態数の合計は$totalstate 状態です。\n";
print"一度訪れた状態数の合計は$totalmatch 状態です。\n";
print"遷移した合計は$totaltransitions 状態です。\n";
print"使用したメモリの合計は$totalmemory MByte です。\n";
print"\n";
```

反例から必要な情報を抽出する

```
#!/bin/sh

#grep を行うモデルのパターン ID を指定する
echo "grep を行うパターンの ID を入力ください。"
read a

#grep を行うモデルのタスク数、リソース数を指定する
echo "grep 開始時のタスクの値を入力ください。"
read b
echo "grep 開始時のリソースの値を入力してください。"
read c
```

```
echo "grep 終了時の検査モデル ID を指定する。"
read d

#CTRL+C で強制終了させる
stop=
trap 'stop=1; trap 2 3' 1 2 3 15
#trap 'echo "trapped."; exit 1' 1 2 3 15

#反例結果の内容を grep で抽出する
cd pattern"$a"
mkdir  hanrei"$b"-"$c"-"$a"kai
i=1

while :
do
  if [ $stop ]
  then
    cd ..
    break 2
  fi

  if test -f ./hanrei"$b"-"$c"-"$a"/hanrei"$i".txt
  then
    #通常の反例シミュレート
    #sed -e 's/ActivateTask\(.*\)/ActivateTask/g' -e 's/TerminateTask\(.*\)/Terminate'
-e 's/ChainTask\(.*\)/ChainTask/g' -e 's/GetResource\(.*\)/GetResource/g' -e
's/ReleaseResource\(.*\)/ReleaseResource/g' -e 's/"case.*.spin"/"case.spin"/g'
hanrei"$b"-"$c"-"$a"/hanrei"$i".txt > hanrei"$b"-"$c"-"$a"kai/tphanrei.txt
    #egrep 'BEGIN0:DeclareTask|BEGIN0:DeclareResource|ActivateTask|TerminateTask|Chai
ercd =|Error at|spin: text of failed assertion:' hanrei"$b"-"$c"-"$a"/hanrei"$i".txt
> hanrei"$b"-"$c"-"$a"kai/gphanrei"$i".txt
    #egrep 'ActivateTask|TerminateTask|ChainTask|GetResource[^State]|ReleaseResource[
ercd =|Error at|spin: text of failed assertion:' hanrei"$b"-"$c"-"$a"/hanrei"$i".txt
> hanrei"$b"-"$c"-"$a"kai/gphanrei"$i".txt
    #egrep 'Error ercd =|Error at|spin: text of failed assertion:' hanrei"$b"-"$c"-"$
> hanrei"$b"-"$c"-"$a"kai/gphanrei"$i".txt
    egrep 'ActivateTask|TerminateTask|ChainTask|GetResource[^State]|ReleaseResource[^
hanrei"$b"-"$c"-"$a"/hanrei"$i".txt > hanrei"$b"-"$c"-"$a"kai/gphanrei"$i".txt
```

```
    if test $i -eq $d
    then
        break 1
    fi
    i=‘expr $i + 1‘
  else
    if test $i -eq $d
    then
        break 1
    fi
    i=‘expr $i + 1‘
  fi
done
```

反例間の差分を求める

```
#!/bin/sh

#調べるモデルのタスク、リソース数を指定する
echo "比較先である反例のパターン ID を入力する。"
read a
echo "比較元である反例のタスク数を入力する。"
read b
echo "比較元である反例のリソース数を入力する。"
read c
echo "比較元の開始ファイル ID を指定する。"
read d
echo "比較元の終了ファイル ID を指定する。"
read i
echo "比較先である反例のタスク数を入力する。"
read e
echo "比較先である反例のリソース数を入力する。"
read f
echo "比較先の開始ファイル ID を指定する。"
read g
echo "比較先の終了ファイル ID を指定する。"
read h

#g=1
#h=1
```

```
diffcount=0
cd pattern"$a"

#diff ディレクトリの作成。
if test -d ./diff
then
  continue
else
  mkdir diff
fi

#既に比較結果ファイルが存在する場合、消去する。
if test -f ./diff/t"$b"r"$c"-t"$e"r"$f".txt
then
  rm -r ./diff/t"$b"r"$c"-t"$e"r"$f".txt
fi
while :
do
      #初期設定したファイルがない場合はファイルが見つかるまで調べ、発見された場
合はそのファイルを用いて比較する。
      while :
      do
        if test -f hanrei"$b"-"$c"-"$a"/gphanrei"$d".txt
        then
            break 1
        else
            while :
            do
            #通常の反例シミュレートの場 d 合
            if test -f ./hanrei"$b"-"$c"-"$a"/gphanrei"$d".txt
            #詳細な反例シミュレートの場合
            #if test -f ./hanrei"$b"-"$c"-"$a"/gptphanrei"$d".txt
            then
                break 1
            else
                d=‘expr $d + 1‘
            fi
            done
```

```
  fi
done

#diff コマンドを用いて反例の差分を求める
while :
do
#通常の反例シミュレートの場合
if test -f ./hanrei"$e"-"$f"-"$a"/gphanrei"$g".txt
#詳細な反例シミュレートの場合
#if test -f ./hanrei"$e"-"$f"-"$a"/gptphanrei"$g".txt
then
    if test $d -eq $g
    then
        g=‘expr $g + 1‘
    fi

    #比較先の開始ファイル ID が終了ファイル ID より大きいか確認。
    if test $g -gt $h
    then
        #比較元の開始ファイル ID が終了ファイル ID より大きいか確認。
        if test $d -eq $i
        then
            echo "差分が取れた数は"$diffcount です。"" >> ./diff/t"$b"r"$c"-t"$e"r'
            break 2
        else
            d=‘expr $d + 1‘
            g=$d
            break 1
        fi
    fi

    #通常の反例シミュレートの場合
    echo "hanrei"$b""$c""$d""$e""$f""$g"" >> ./diff/t"$b"r"$c"-t"$e"r"$f".txt
    diff ./hanrei"$b"-"$c"-"$a"/gphanrei"$d".txt ./hanrei"$e"-"$f"-"$a"/gphanre
>> ./diff/t"$b"r"$c"-t"$e"r"$f".txt
    echo " " >> ./diff/t"$b"r"$c"-t"$e"r"$f".txt

    #詳細な反例シミュレートの場合
```

```
            #echo "hanrei"$b"-"$c"-"$a"/gptphanrei"$d".txt hanrei"$e"-"$f"-"$a"/gptphan
>> ./diff/t"$b"r"$c"-t"$e"r"$f".txt
            #diff ./hanrei"$b"-"$c"-"$a"/gptphanrei"$d".txt ./hanrei"$e"-"$f"-"$a"/gptp
>> ./diff/t"$b"r"$c"-t"$e"r"$f".txt
            #echo " " >> ./diff/t"$b"r"$c"-t"$e"r"$f".txt

            diffcount=‘expr $diffcount + 1‘
            g=‘expr $g + 1‘
        else

            #通常の反例シミュレートの場合
            #echo "hanrei"$e"-"$f"-"$a"/gphanrei"$g".txtはありません。" >> ./diff/t"$b"r"

            #詳細な反例シミュレートの場合
            #echo "hanrei"$e"-"$f"-"$a"/gphanrei"$g".txt.txtはありません。" >>
./diff/t"$b"r"$c"-t"$e"r"$f".txt

            #比較先の開始ファイルIDが終了ファイルIDより大きいか確認。
            if test $g -gt $h
            then
                #比較元の開始ファイルIDが終了ファイルIDより大きいか確認。
                if test $d -gt $i
                then
                    echo "差分が取れた数は"$diffcountです。"" >> ./diff/t"$b"r"$c"-t"$e"r"
                    break 2
                else
                    d=‘expr $d + 1‘
                    g=$d
                    break 1
                fi
            fi
            g=‘expr $g + 1‘
        fi
        done
done
```

反例間の差分を距離に変換する

```
#!/usr/local/bin/perl
```

```
#調べるモデルのタスク、リソース数を指定する
#print "比較元である反例のタスク数を入力する。\n";
chomp($a = <STDIN>);
#print "比較元である反例のリソース数を入力する。\n";
chomp($b = <STDIN>);
#print "比較先である反例のタスク数を入力する。\n";
chomp($c = <STDIN>);
#print "比較先である反例のリソース数を入力する。\n";
chomp($d = <STDIN>);

$x=0;
$y=0;

my $filename="t" . "$a" . "r" . "$b" . "-t" . "$c" . "r" . "$d" . ".txt";
#print "$filename";
open(FILE, $filename) or die "$!";
while ( my $line = <FILE>) {
if( $line =~ /^hanrei/ ) {
$name=$line;
}elsif( $line =~ /^</ ) {
$x=$x+1;
}elsif( $line =~ /^>/ ) {
$y=$y+1;
}elsif( $line =~ / / ) {
#<と>の数が等しい場合
if ($x == $y){
$kyori=$x;
}#<の数が>の数より多い場合
elsif ($x > $y){
$kyori=$x;
}#<の数が>の数より少ない場合
elsif ($x < $y){
         $kyori=$y;
}
                $difftext = "$name"."$kyori\n";
         print "$difftext";
$x=0;
$y=0;
```

```
}
}
close(FILE);
```

付録 A

3 章の実験 1 の検査結果を示す。

ノード	検査を行うとき用いた検査モデル	所要時間
a013	t1r0~5,t2r0~2,t2r3(1~11)	2:00
a014	t2r3(12~317)	1:36
a015	t2r3(318~624)	1:16
a016	t2r3(625~930)	2:55
a017	t2r4(1~131)	1:53
a018	t2r4(132~262)	1:14
a019	t2r4(263~393)	3:03
a020	t2r4(394~524)	1:59
a021	t2r4(525~655)	2:02
a022	t2r4(656~786)	2:12
a023	t2r4(787~917)	1:02
a024	t2r4(918~1048)	4:43
a033	t2r4(1049~1179)	1:23
a034	t2r4(1180~1310)	1:16
a044	t2r4(1311~1441)	2:54
a045	t2r4(1442~1572)	2:30
a046	t2r4(1573~1703)	2:29
a047	t2r4(1704~1834)	2:04
a048	t2r4(1835~1965)	4:23
a049	t2r4(1966~2096)	6:27
a050	t2r4(2097~2227)	2:02
a051	t2r4(2228~2358)	2:29
a052	t2r4(2359~2489)	1:38
a053	t2r4(2490~2620)	3:27
a054	t2r4(2621~2751)	3:46
a068	t2r4(2752~2882)	3:52
a069	t2r4(2883~3013)	4:24
a070	t2r4(3014~3140)	測定できなかった
a071	t3r0,t3r1(1~187)	1:32
a072	t3r1(188~515)	2:00
a073	t3r1(516~842)	2:25
a074	t3r1(843~1014),t3r2(1~99)	2:32
a075	t3r2(100~306)	1:29
a076	t3r2(307~512)	1:58
a077	t3r2(513~718)	2:02
a078	t3r2(719~924)	1:50
a079	t3r2(925~1130)	1:39
a080	t3r2(1131~1336)	1:59
a081	t3r2(1337~1542)	2:08
a082	t3r2(1543~1748)	2:19
a083	t3r2(1749~1954)	1:28
a084	t3r2(1955~2160)	1:42
a085	t3r2(2161~2366)	2:07
a086	t3r2(2367~2572)	2:15
a087	t3r2(2573~2778)	2:10
a088	t3r2(2779~2984)	2:36
a089	t3r2(2985~3190)	3:03
a090	t3r2(3191~3396)	3:07
a091	t3r2(3397~3602)	2:37
a092	t3r2(3603~3808)	2:06
a093	t3r2(3809~4014)	3:00
a094	t3r2(4015~4220)	4:43

ノード	検査を行うとき用いた検査モデル	所要時間
a095	t3r2(4221~4426)	2:02
a096	t3r2(4427~4632)	2:26
a097	t3r2(4633~4838)	3:10
a098	t3r2(4839~5044)	3:37
a099	t3r2(5045~5250)	1:59
a100	t3r2(5251~5456)	2:20
a101	t3r2(5457~5662)	2:48
a102	t3r2(5663~5868)	2:54
a103	t3r2(5869~6074)	3:00
a104	t3r2(6075~6280)	3:27
a105	t3r2(6281~6486)	4:04
a106	t3r2(6487~6692)	4:27
a107	t3r2(6693~6898)	4:00
a108	t3r2(6899~7104)	2:51
a109	t3r2(7105~7310)	3:21
a110	t3r2(7311~7516)	4:03
a111	t3r2(7517~7722)	4:43
a112	t3r2(7723~7928)	6:27

付録B

3章の実験2の検査結果を示す。

ノード	検査を行うとき用いた検査モデル	所要時間
a013	t1r0~5,t2r0~2,t2r3(1~189)	2:30
a014	t2r3(190~629)	2:23
a015	t2r3(630~930),t2r4(1~6)	2:21
a016	t2r4(7~232)	2:11
a017	t2r4(233~501)	2:24
a018	t2r4(502~667)	1:52
a019	t2r4(668~860)	6:32
a020	t2r4(861~1105)	2:18
a021	t2r4(1106~1369)	2:22
a022	t2r4(1370~1511)	2:16
a023	t2r4(1512~1604)	3:30
a024	t2r4(1605~1777)	9:15
a033	t2r4(1778~1893)	1:49
a034	t2r4(1894~1955)	1:15
a044	t2r4(1956~2057)	1:47
a045	t2r4(2058~2118)	1:53
a046	t2r4(2119~2341)	3:04
a047	t2r4(2342~2530)	2:09
a048	t2r4(2531~2617)	3:05
a049	t2r4(2618~2719)	2:56
a050	t2r4(2720~2769)	1:52
a051	t2r4(2770~2852)	1:53
a052	t2r4(2853~2949)	2:00
a053	t2r4(2950~3003)	1:54
a054	t2r4(3004~3045)	1:46
a068	t2r4(3046~3091)	1:52
a069	t2r4(3092~3109)	1:48
a070	t2r4(3110~3123)	1:28
a071	t2r4(3124~3137)	1:30
a072	t2r4(3138~3140),t3r0,t3r1(1~372	2:22
a073	t3r1(865~1014),t3r2(1~230)	2:36
a074	t3r2(231~577)	2:29
a075	t3r2(578~950)	2:32
a076	t3r2(951~1329)	2:33
a077	t3r2(1330~1640)	2:44
a078	t3r2(1641~1992)	2:30
a079	t3r2(1993~2359)	2:41
a080	t3r2(2360~2676)	2:38
a081	t3r2(2677~2943)	2:30
a082	t3r2(2944~3183)	2:41
a083	t3r2(3184~3378)	2:21
a084	t3r2(3379~3620)	2:30
a085	t3r2(3621~3906)	2:27
a086	t3r2(3907~4106)	2:19
a087	t3r2(4107~4351)	2:15
a088	t3r2(4352~4631)	2:30
a089	t3r2(4632~4818)	2:17
a090	t3r2(4819~4995)	2:24
a091	t3r2(4996~5269)	2:20
a092	t3r2(5270~5537)	2:29
a093	t3r2(5538~5778)	2:39
a094	t3r2(5779~5988)	測定できなかった

ノード	検査を行うとき用いた検査モデ	所要時間
a095	t3r2(5989~6180)	2:23
a096	t3r2(6181~6341)	2:20
a097	t3r2(6342~6486)	2:25
a098	t3r2(6487~6610)	2:15
a099	t3r2(6611~6724)	2:08
a100	t3r2(6725~6862)	2:13
a101	t3r2(6863~7071)	2:21
a102	t3r2(7072~7262)	2:24
a103	t3r2(7263~7401)	2:06
a104	t3r2(7402~7526)	2:03
a105	t3r2(7527~7641)	2:07
a106	t3r2(7642~7748)	2:09
a107	t3r2(7749~7820)	1:45
a108	t3r2(7821~7889)	1:52
a109	t3r2(7890~7928)	1:10
a110	t3r1(373~864)	2:46
a111		
a112		

付録C

5章のクラスター分析の適用結果を示す。

No	エラーモデル	ステップ数	エラー内容
1	./caset1r0/error1.txt	77	
2	./caset1r0/error2.txt	77	
3	./caset1r0/error3.txt	77	
4	./caset1r0/error4.txt	77	
5	./caset1r1/error1.txt	249	Error at enq: queue out of range 1
6	./caset1r1/error2.txt	249	Error at enq: queue out of range 1
7	./caset1r1/error3.txt	237	Error at enq: queue out of range 3
8	./caset1r1/error4.txt	249	Error at enq: queue out of range 1
9	./caset1r1/error5.txt	249	Error at enq: queue out of range 1
10	./caset1r1/error6.txt	237	Error at enq: queue out of range 3
11	./caset1r2/error1.txt	268	Error at enq: queue out of range 1
12	./caset1r2/error2.txt	268	Error at enq: queue out of range 1
13	./caset1r2/error3.txt	268	Error at enq: queue out of range 1
14	./caset1r2/error4.txt	256	Error at enq: queue out of range 3
15	./caset1r2/error5.txt	268	Error at enq: queue out of range 1
16	./caset1r2/error6.txt	268	Error at enq: queue out of range 1
17	./caset1r2/error7.txt	268	Error at enq: queue out of range 1
18	./caset1r2/error8.txt	256	Error at enq: queue out of range 3
19	./caset1r3/error1.txt	287	Error at enq: queue out of range 1
20	./caset1r3/error10.txt	275	Error at enq: queue out of range 3
21	./caset1r3/error2.txt	287	Error at enq: queue out of range 1
22	./caset1r3/error3.txt	287	Error at enq: queue out of range 1
23	./caset1r3/error4.txt	287	Error at enq: queue out of range 1
24	./caset1r3/error5.txt	275	Error at enq: queue out of range 3
25	./caset1r3/error6.txt	287	Error at enq: queue out of range 1
26	./caset1r3/error7.txt	287	Error at enq: queue out of range 1
27	./caset1r3/error8.txt	287	Error at enq: queue out of range 1
28	./caset1r3/error9.txt	287	Error at enq: queue out of range 1
29	./caset1r4/error1.txt	306	Error at enq: queue out of range 1
30	./caset1r4/error10.txt	306	Error at enq: queue out of range 1
31	./caset1r4/error11.txt	306	Error at enq: queue out of range 1

32	./caset1r4/error12.txt	294	Error at enq: queue out of range 3
33	./caset1r4/error2.txt	306	Error at enq: queue out of range 1
34	./caset1r4/error3.txt	306	Error at enq: queue out of range 1
35	./caset1r4/error4.txt	306	Error at enq: queue out of range 1
36	./caset1r4/error5.txt	306	Error at enq: queue out of range 1
37	./caset1r4/error6.txt	294	Error at enq: queue out of range 3
38	./caset1r4/error7.txt	306	Error at enq: queue out of range 1
39	./caset1r4/error8.txt	306	Error at enq: queue out of range 1
40	./caset1r4/error9.txt	306	Error at enq: queue out of range 1
41	./caset1r5/error1.txt	325	Error at enq: queue out of range 1
42	./caset1r5/error10.txt	325	Error at enq: queue out of range 1
43	./caset1r5/error11.txt	325	Error at enq: queue out of range 1
44	./caset1r5/error12.txt	325	Error at enq: queue out of range 1
45	./caset1r5/error13.txt	325	Error at enq: queue out of range 1
46	./caset1r5/error14.txt	313	Error at enq: queue out of range 3
47	./caset1r5/error2.txt	325	Error at enq: queue out of range 1
48	./caset1r5/error3.txt	325	Error at enq: queue out of range 1
49	./caset1r5/error4.txt	325	Error at enq: queue out of range 1
50	./caset1r5/error5.txt	325	Error at enq: queue out of range 1
51	./caset1r5/error6.txt	325	Error at enq: queue out of range 1
52	./caset1r5/error7.txt	313	Error at enq: queue out of range 3
53	./caset1r5/error8.txt	325	Error at enq: queue out of range 1
54	./caset1r5/error9.txt	325	Error at enq: queue out of range 1

No	エラーモデル	ステップ数	エラーの箇所
1	./caset1r0/error1.txt	77	
2	./caset1r0/error2.txt	77	
3	./caset1r0/error3.txt	77	
4	./caset1r0/error4.txt	77	
5	./caset1r1/error1.txt	249	spin: line 246 "oseklib.spin", Error: assertion violated
6	./caset1r1/error2.txt	249	spin: line 246 "oseklib.spin", Error: assertion violated
7	./caset1r1/error3.txt	237	spin: line 246 "oseklib.spin", Error: assertion violated
8	./caset1r1/error4.txt	249	spin: line 246 "oseklib.spin", Error: assertion violated
9	./caset1r1/error5.txt	249	spin: line 246 "oseklib.spin", Error: assertion violated
10	./caset1r1/error6.txt	237	spin: line 246 "oseklib.spin", Error: assertion violated
11	./caset1r2/error1.txt	268	spin: line 246 "oseklib.spin", Error: assertion violated
12	./caset1r2/error2.txt	268	spin: line 246 "oseklib.spin", Error: assertion violated
13	./caset1r2/error3.txt	268	spin: line 246 "oseklib.spin", Error: assertion violated
14	./caset1r2/error4.txt	256	spin: line 246 "oseklib.spin", Error: assertion violated
15	./caset1r2/error5.txt	268	spin: line 246 "oseklib.spin", Error: assertion violated
16	./caset1r2/error6.txt	268	spin: line 246 "oseklib.spin", Error: assertion violated
17	./caset1r2/error7.txt	268	spin: line 246 "oseklib.spin", Error: assertion violated
18	./caset1r2/error8.txt	256	spin: line 246 "oseklib.spin", Error: assertion violated
19	./caset1r3/error1.txt	287	spin: line 246 "oseklib.spin", Error: assertion violated
20	./caset1r3/error10.txt	275	spin: line 246 "oseklib.spin", Error: assertion violated
21	./caset1r3/error2.txt	287	spin: line 246 "oseklib.spin", Error: assertion violated
22	./caset1r3/error3.txt	287	spin: line 246 "oseklib.spin", Error: assertion violated

No	エラーモデル	ステップ数	エラー内容
1	./caset1r0/error1.txt	76	
2	./caset1r0/error2.txt	76	
3	./caset1r0/error3.txt	76	
4	./caset1r0/error4.txt	76	
5	./caset1r2/error2.txt	390	Error ercd = 1
6	./caset1r2/error6.txt	390	Error ercd = 1
7	./caset1r3/error2.txt	488	Error ercd = 1
8	./caset1r3/error3.txt	419	Error ercd = 1
9	./caset1r3/error7.txt	488	Error ercd = 1
10	./caset1r3/error8.txt	419	Error ercd = 1
11	./caset1r4/error10.txt	448	Error ercd = 1
12	./caset1r4/error2.txt	842	Error ercd = 1
13	./caset1r4/error3.txt	522	Error ercd = 1
14	./caset1r4/error4.txt	448	Error ercd = 1
15	./caset1r4/error8.txt	842	Error ercd = 1
16	./caset1r4/error9.txt	522	Error ercd = 1
17	./caset1r5/error10.txt	906	Error ercd = 1
18	./caset1r5/error11.txt	560	Error ercd = 1
19	./caset1r5/error12.txt	481	Error ercd = 1
20	./caset1r5/error2.txt	1598	Error ercd = 1
21	./caset1r5/error3.txt	906	Error ercd = 1
22	./caset1r5/error4.txt	560	Error ercd = 1
23	./caset1r5/error5.txt	481	Error ercd = 1
24	./caset1r5/error9.txt	1598	Error ercd = 1

No	エラーモデル	ステップ数	エラー内容
1	./caset1r0/error1.txt	76	
2	./caset1r0/error2.txt	76	
3	./caset1r0/error3.txt	76	
4	./caset1r0/error4.txt	76	
5	./caset1r2/error2.txt	389	Error ercd = 1
6	./caset1r2/error6.txt	389	Error ercd = 1
7	./caset1r3/error2.txt	487	Error ercd = 1
8	./caset1r3/error3.txt	418	Error ercd = 1
9	./caset1r3/error7.txt	487	Error ercd = 1
10	./caset1r3/error8.txt	418	Error ercd = 1
11	./caset1r4/error10.txt	447	Error ercd = 1
12	./caset1r4/error2.txt	841	Error ercd = 1
13	./caset1r4/error3.txt	521	Error ercd = 1
14	./caset1r4/error4.txt	447	Error ercd = 1
15	./caset1r4/error8.txt	841	Error ercd = 1
16	./caset1r4/error9.txt	521	Error ercd = 1
17	./caset1r5/error10.txt	905	Error ercd = 1
18	./caset1r5/error11.txt	559	Error ercd = 1
19	./caset1r5/error12.txt	480	Error ercd = 1
20	./caset1r5/error2.txt	1597	Error ercd = 1
21	./caset1r5/error3.txt	905	Error ercd = 1
22	./caset1r5/error4.txt	559	Error ercd = 1
23	./caset1r5/error5.txt	480	Error ercd = 1
24	./caset1r5/error9.txt	1597	Error ercd = 1

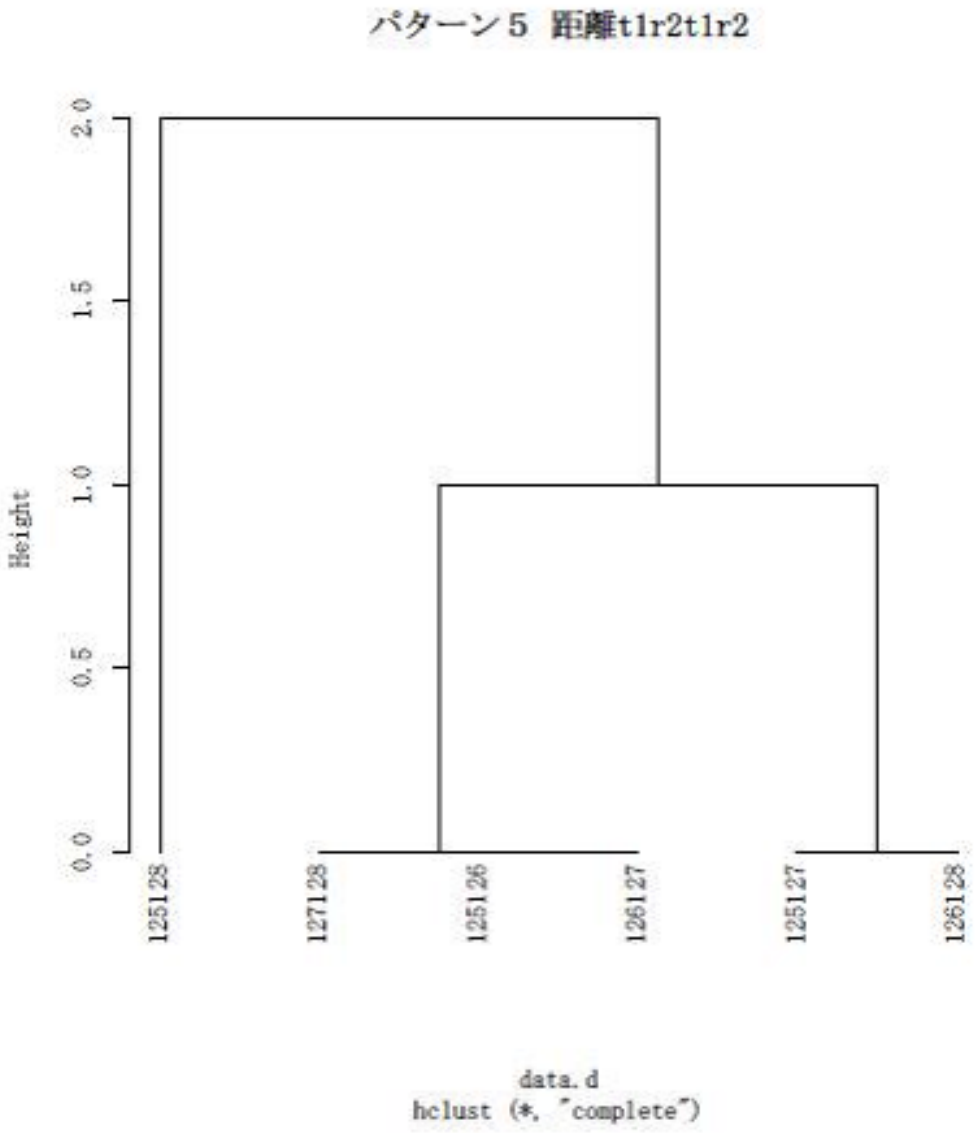
No	エラーモデル	ステップ数	エラーの箇所
1	./caset1r0/error1.txt	76	
2	./caset1r0/error2.txt	76	
3	./caset1r0/error3.txt	76	
4	./caset1r0/error4.txt	76	
5	./caset1r2/error2.txt	389	spin: line 700 "oseklib.spin", Error: assertion violated
6	./caset1r2/error6.txt	389	spin: line 700 "oseklib.spin", Error: assertion violated
7	./caset1r3/error2.txt	487	spin: line 700 "oseklib.spin", Error: assertion violated
8	./caset1r3/error3.txt	418	spin: line 700 "oseklib.spin", Error: assertion violated
9	./caset1r3/error7.txt	487	spin: line 700 "oseklib.spin", Error: assertion violated
10	./caset1r3/error8.txt	418	spin: line 700 "oseklib.spin", Error: assertion violated
11	./caset1r4/error10.txt	447	spin: line 700 "oseklib.spin", Error: assertion violated
12	./caset1r4/error2.txt	841	spin: line 700 "oseklib.spin", Error: assertion violated
13	./caset1r4/error3.txt	521	spin: line 700 "oseklib.spin", Error: assertion violated
14	./caset1r4/error4.txt	447	spin: line 700 "oseklib.spin", Error: assertion violated
15	./caset1r4/error8.txt	841	spin: line 700 "oseklib.spin", Error: assertion violated
16	./caset1r4/error9.txt	521	spin: line 700 "oseklib.spin", Error: assertion violated
17	./caset1r5/error10.txt	905	spin: line 700 "oseklib.spin", Error: assertion violated
18	./caset1r5/error11.txt	559	spin: line 700 "oseklib.spin", Error: assertion violated
19	./caset1r5/error12.txt	480	spin: line 700 "oseklib.spin", Error: assertion violated
20	./caset1r5/error2.txt	1597	spin: line 700 "oseklib.spin", Error: assertion violated
21	./caset1r5/error3.txt	905	spin: line 700 "oseklib.spin", Error: assertion violated
22	./caset1r5/error4.txt	559	spin: line 700 "oseklib.spin", Error: assertion violated

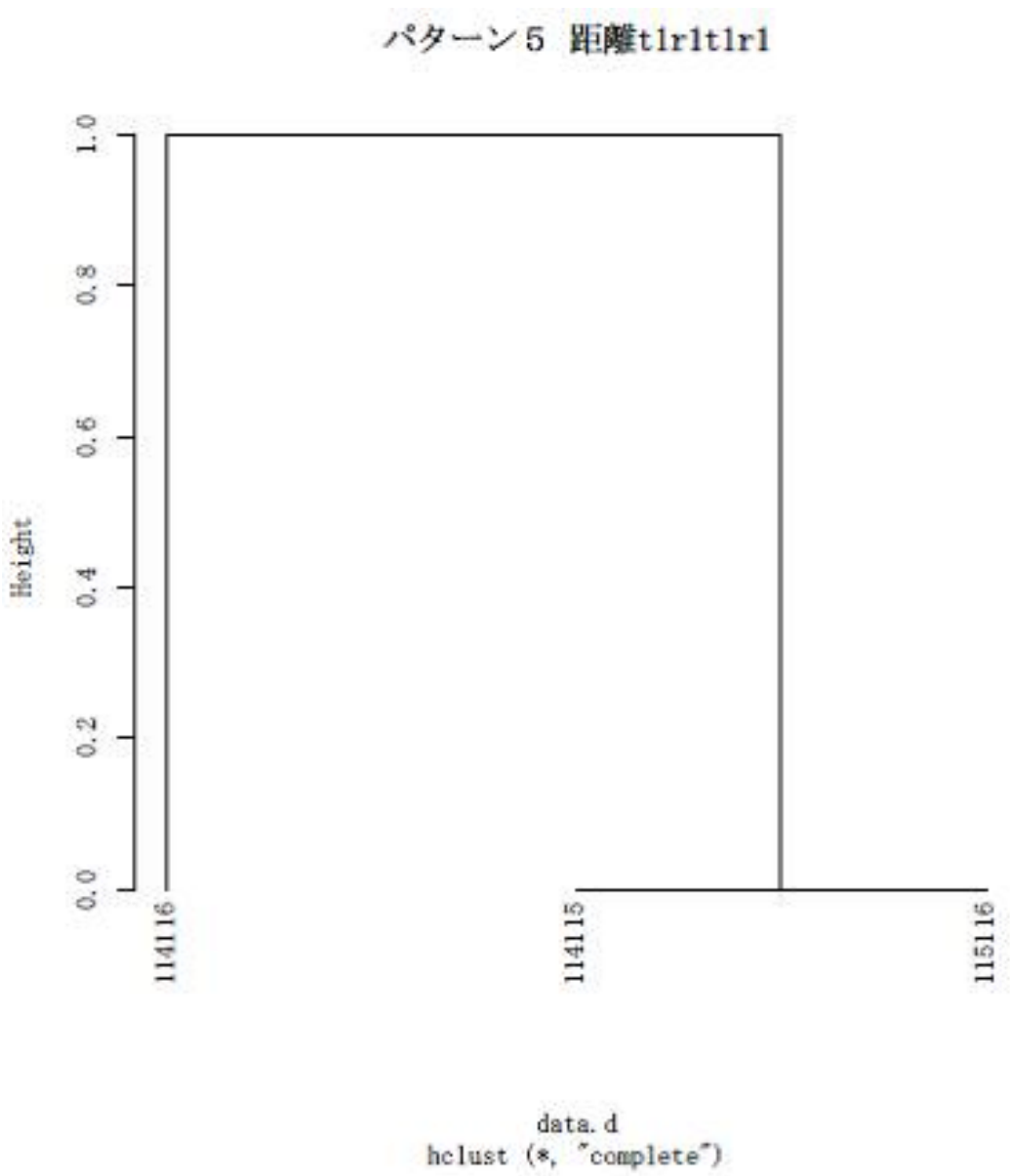
No	エラーモデル	ステップ数	エラーの箇所
23	./caset1r5/error5.txt	480	spin: line 700 "oseklib.spin", Error: assertion violated
24	./caset1r5/error9.txt	1597	spin: line 700 "oseklib.spin", Error: assertion violated

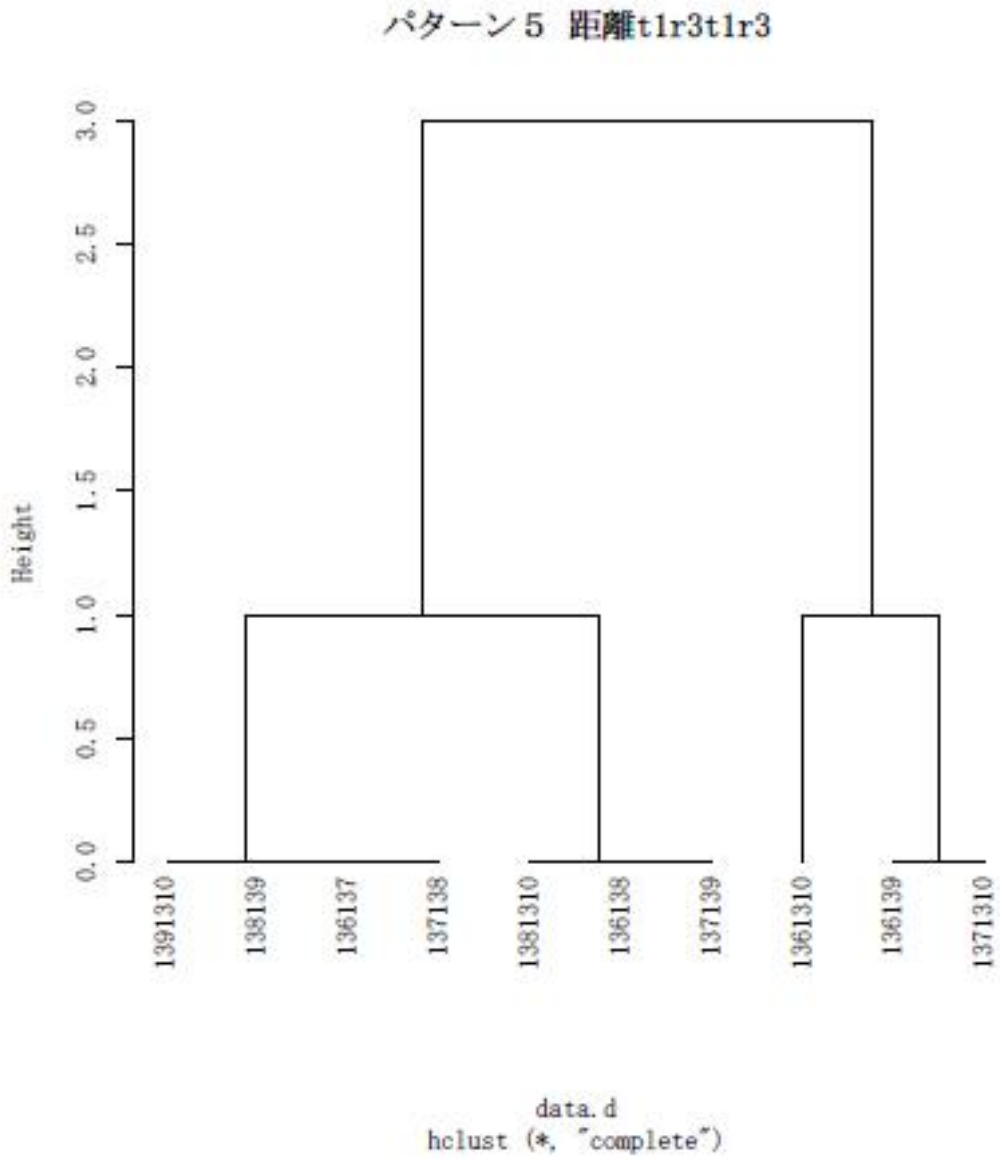
23	./caset1r3/error4.txt	287	spin: line 246 "oseklib.spin", Error: assertion violated
24	./caset1r3/error5.txt	275	spin: line 246 "oseklib.spin", Error: assertion violated
25	./caset1r3/error6.txt	287	spin: line 246 "oseklib.spin", Error: assertion violated
26	./caset1r3/error7.txt	287	spin: line 246 "oseklib.spin", Error: assertion violated
27	./caset1r3/error8.txt	287	spin: line 246 "oseklib.spin", Error: assertion violated
28	./caset1r3/error9.txt	287	spin: line 246 "oseklib.spin", Error: assertion violated
29	./caset1r4/error1.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
30	./caset1r4/error10.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
31	./caset1r4/error11.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
32	./caset1r4/error12.txt	294	spin: line 246 "oseklib.spin", Error: assertion violated
33	./caset1r4/error2.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
34	./caset1r4/error3.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
35	./caset1r4/error4.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
36	./caset1r4/error5.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
37	./caset1r4/error6.txt	294	spin: line 246 "oseklib.spin", Error: assertion violated
38	./caset1r4/error7.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
39	./caset1r4/error8.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
40	./caset1r4/error9.txt	306	spin: line 246 "oseklib.spin", Error: assertion violated
41	./caset1r5/error1.txt	325	spin: line 246 "oseklib.spin", Error: assertion violated

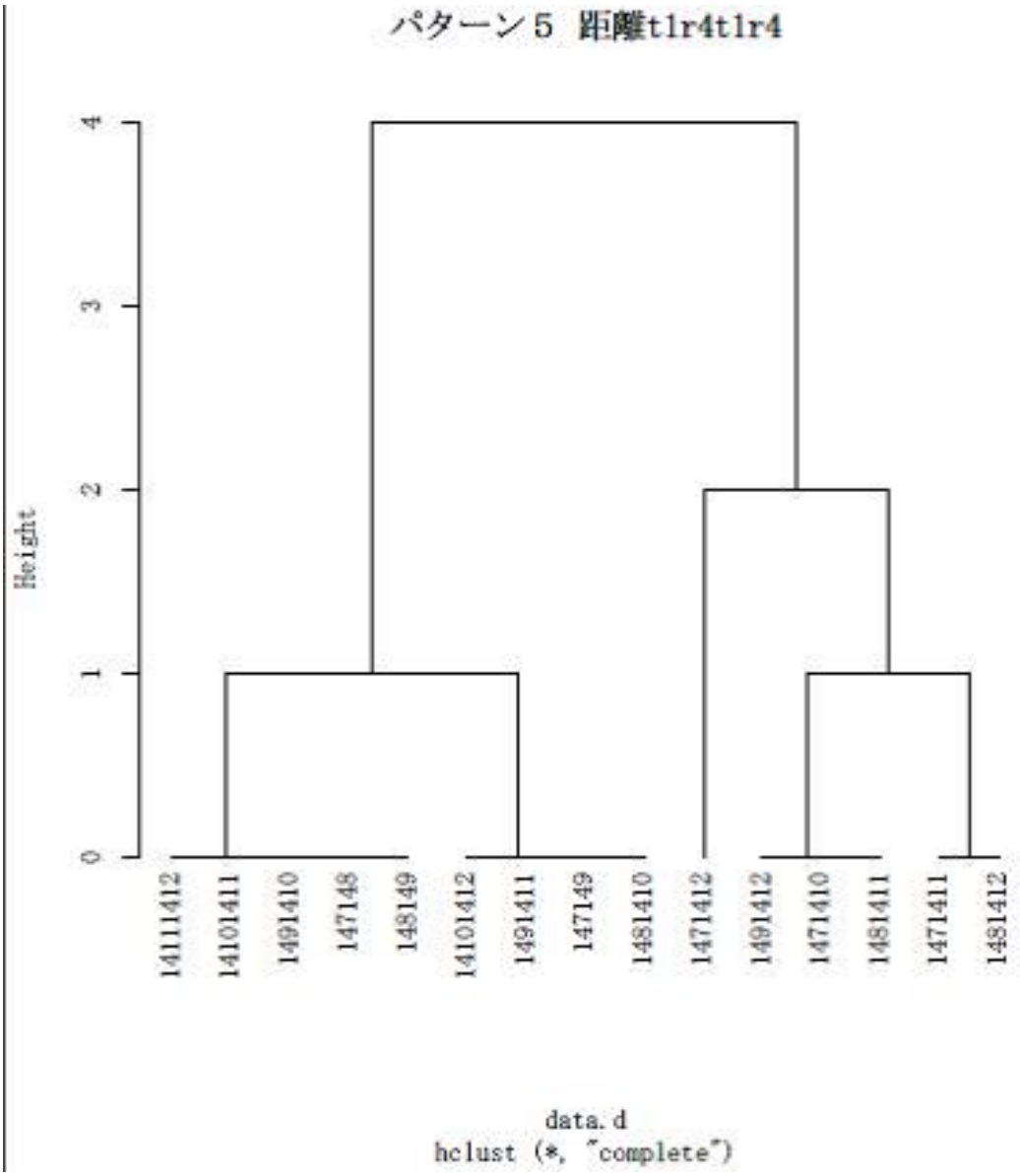
付録D

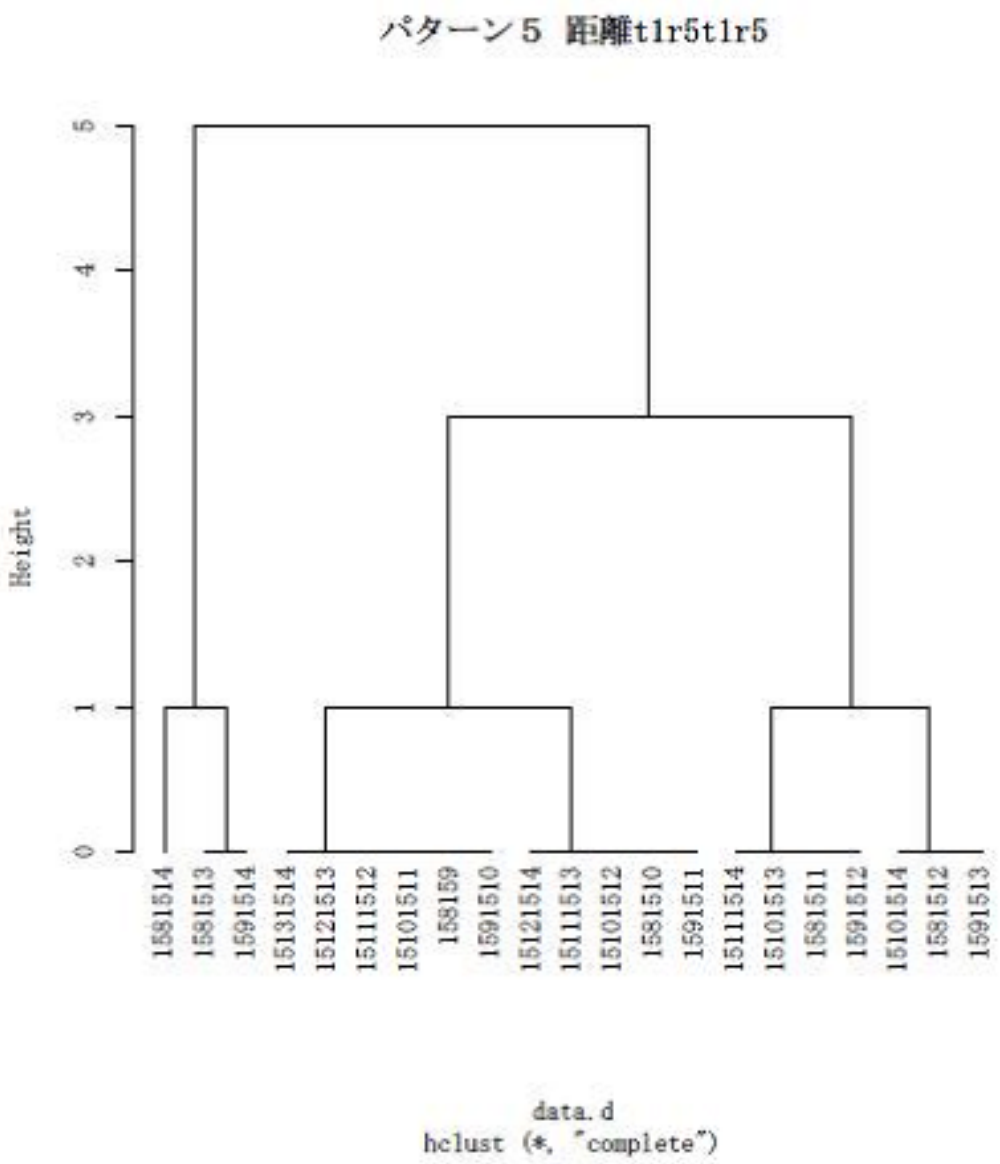
5章のクラスター分析の適用結果を示す。

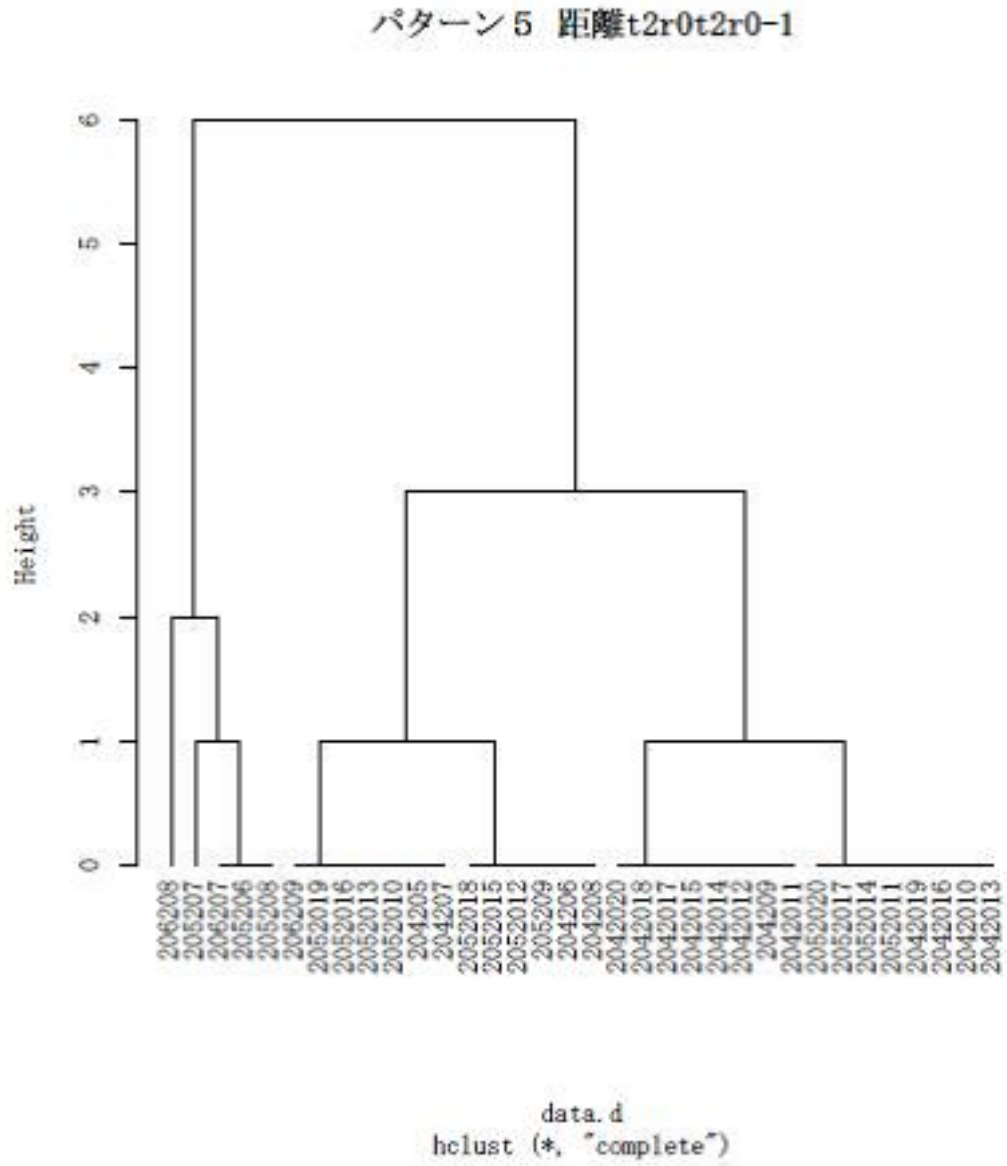



パターン５ 距離t1r1t1r1
Height
0.0
0.2
0.4
0.6
0.8
1.0
114116
114115
115116
data.d
hclust (*, "complete")

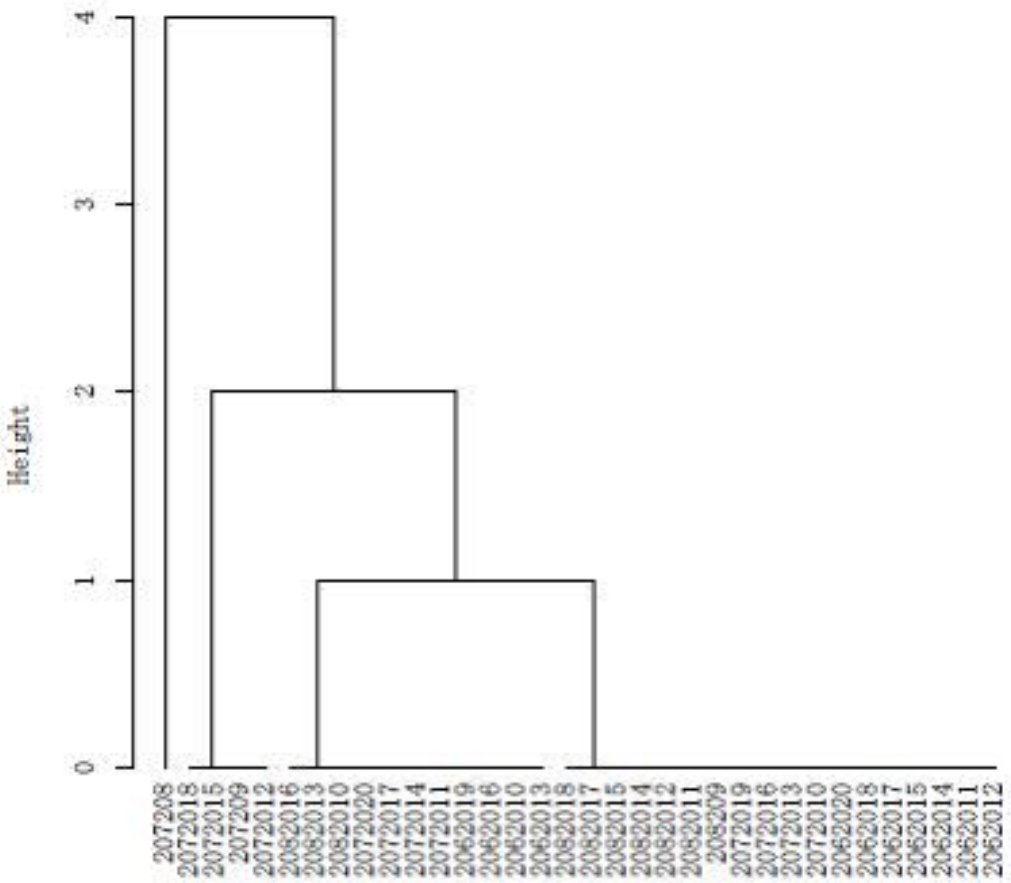

パターン５ 距離t1r3t1r3
Height
0.0
0.5
1.0
1.5
2.0
2.5
3.0
1391310
138139
136137
137138
1381310
136138
137139
1361310
136139
1371310
data.d
hclust (*, "complete")


パターン５ 距離t1r4t1r4
Height
0
1
2
3
4
14111412
14101411
1491410
147148
148149
14101412
1491411
147149
1481410
1471412
1491412
1471410
1481411
1471411
1481412
data.d
hclust (*, "complete")

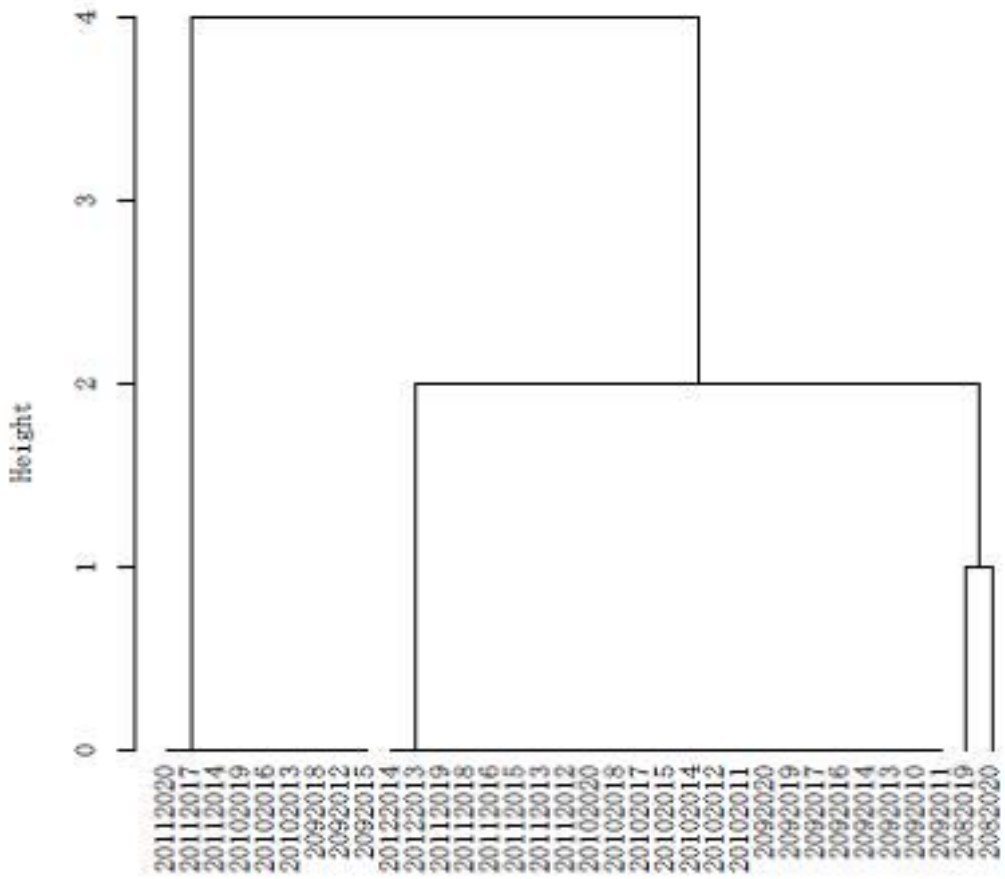

パターン５ 距離t1r5t1r5
Height
0
1
2
3
4
5
1581514
1581513
1591514
15131514
15121513
15111512
15101511
158159
1591510
15121514
15111513
15101512
1581510
1591511
15111514
15101513
1581511
1591512
15101514
1581512
1591513
data.d
hclust (*, "complete")


パターン５ 距離t2r0t2r0-1
Height
0
1
2
3
4
5
6
206208
205207
205207
205206
205208
206209
2052019
2052016
2052013
2052010
204205
204207
2052018
2052015
2052012
205209
204206
204208
2042020
2042018
2042017
2042015
2042014
2042012
204209
2042011
2052020
2052017
2052014
2052011
2042019
2042016
2042010
2042013
data.d
hclust (*, "complete")

パターン５ 距離t2r0t2r0-2

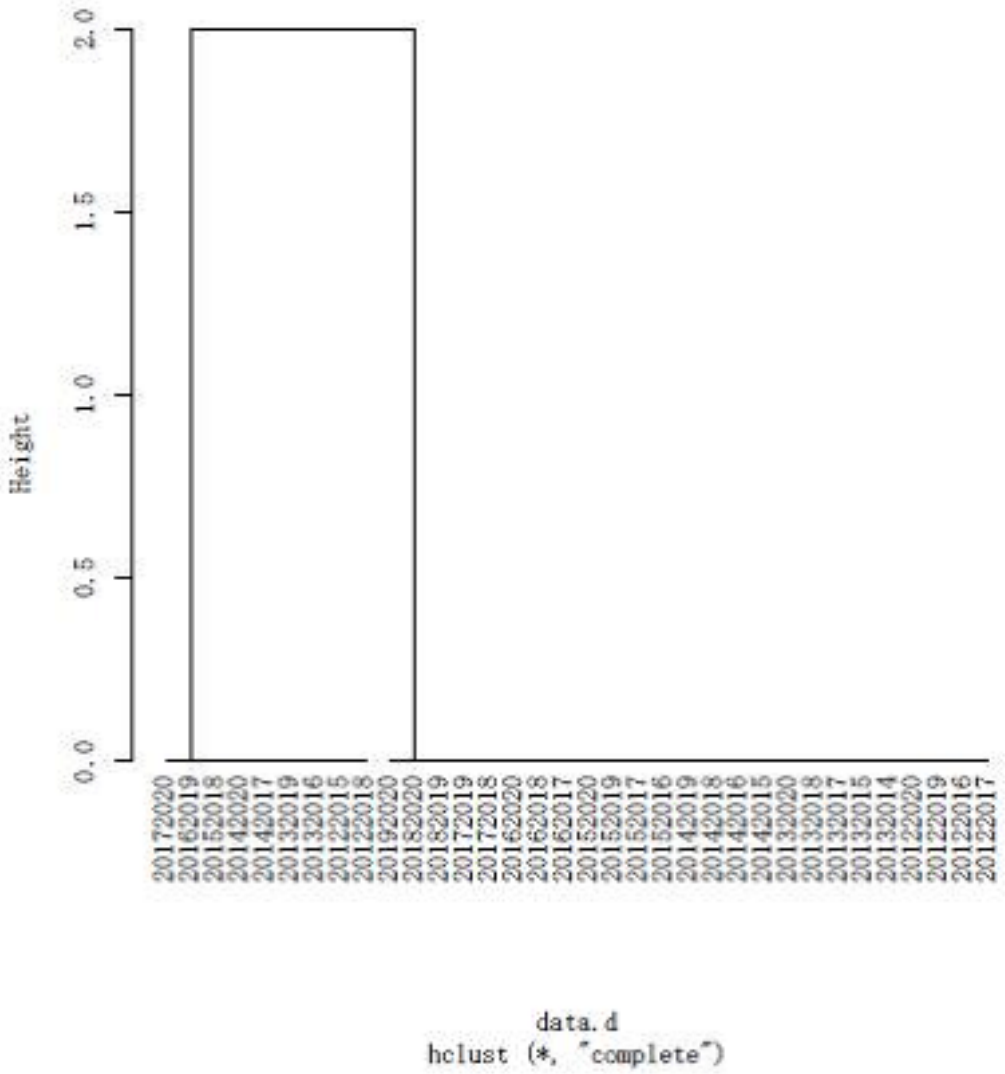


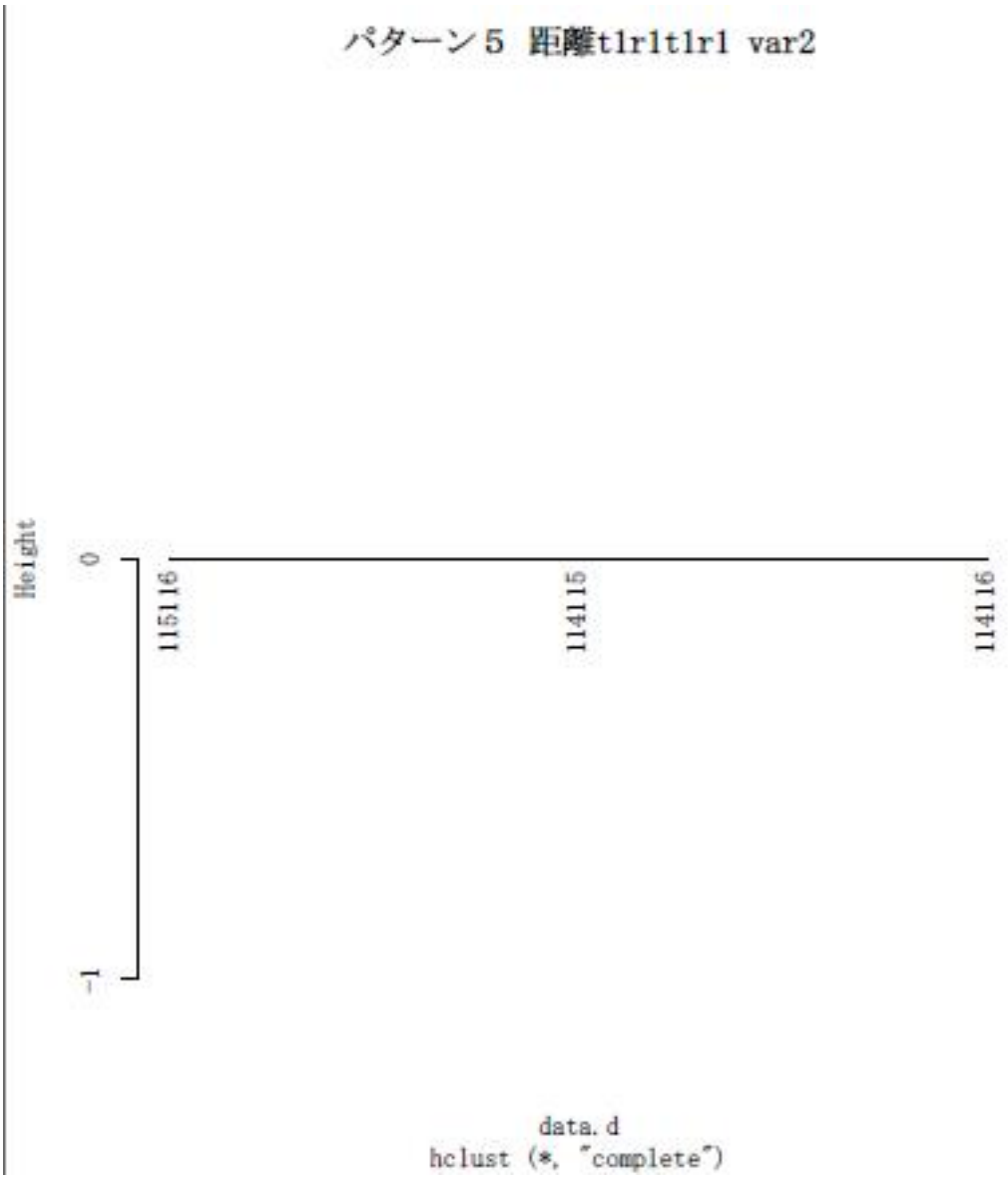
パターン５ 距離t2r0t2r0-3



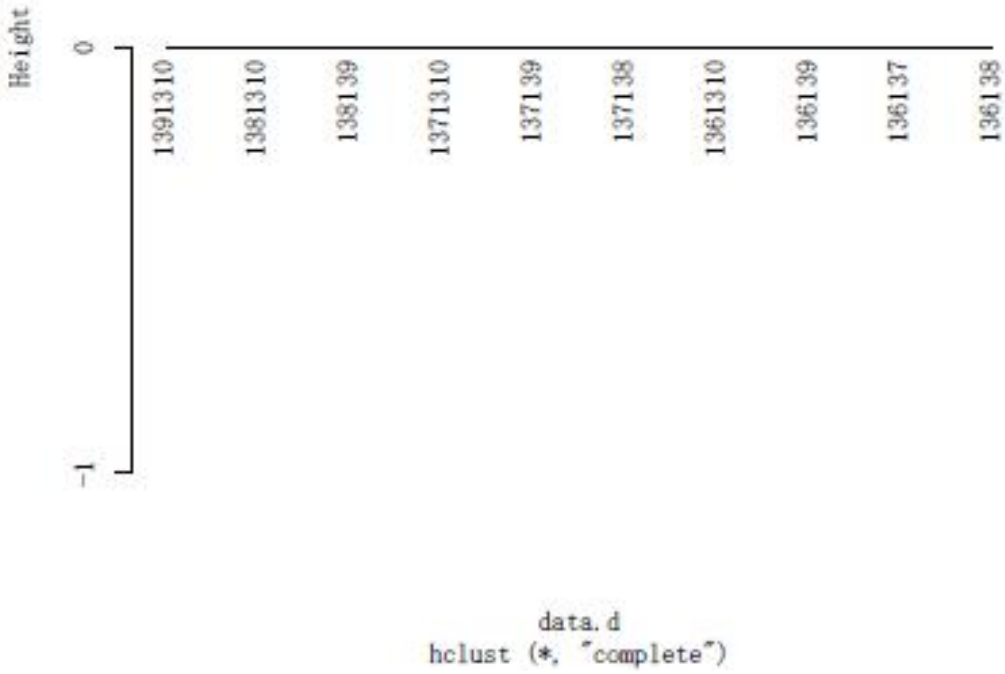
data.d
hclust (*, "complete")

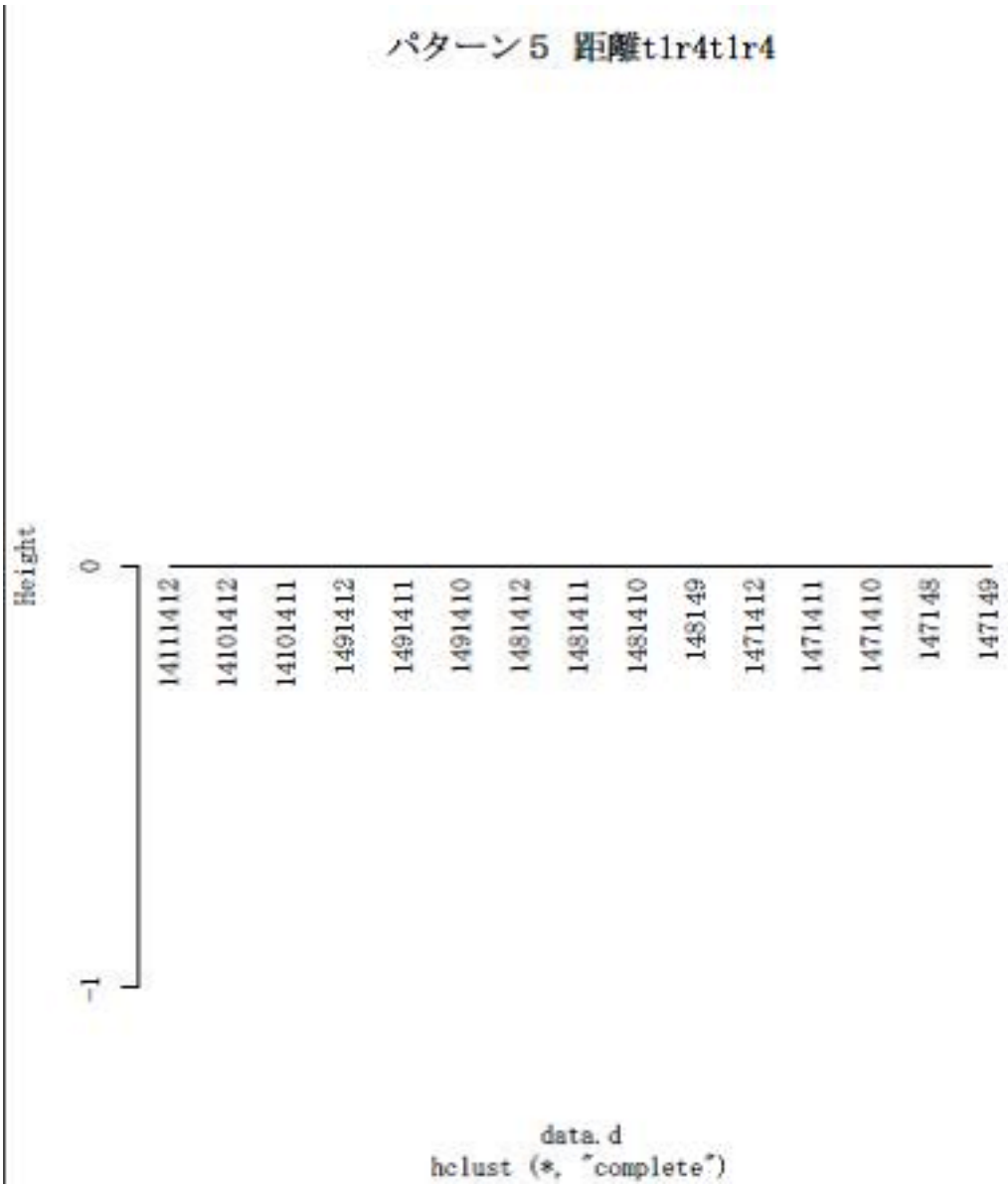
パターン５ 距離t2r0t2r0-4



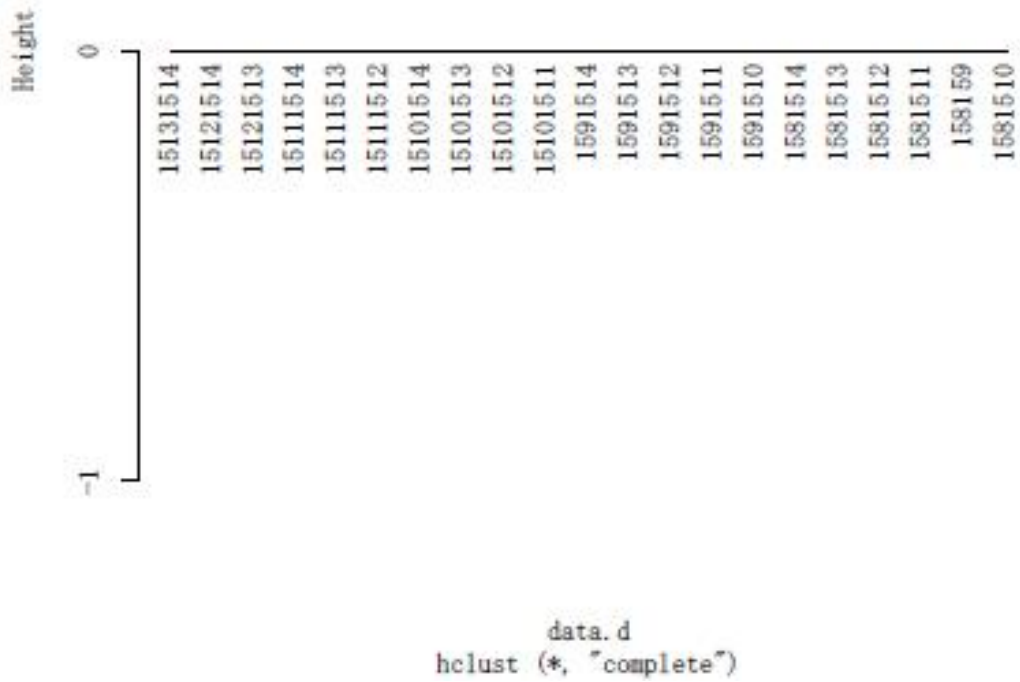

パターン５ 距離t1r1t1r1 var2
Height
0
-1
115116
114115
114116
data.d
hclust (*, "complete")

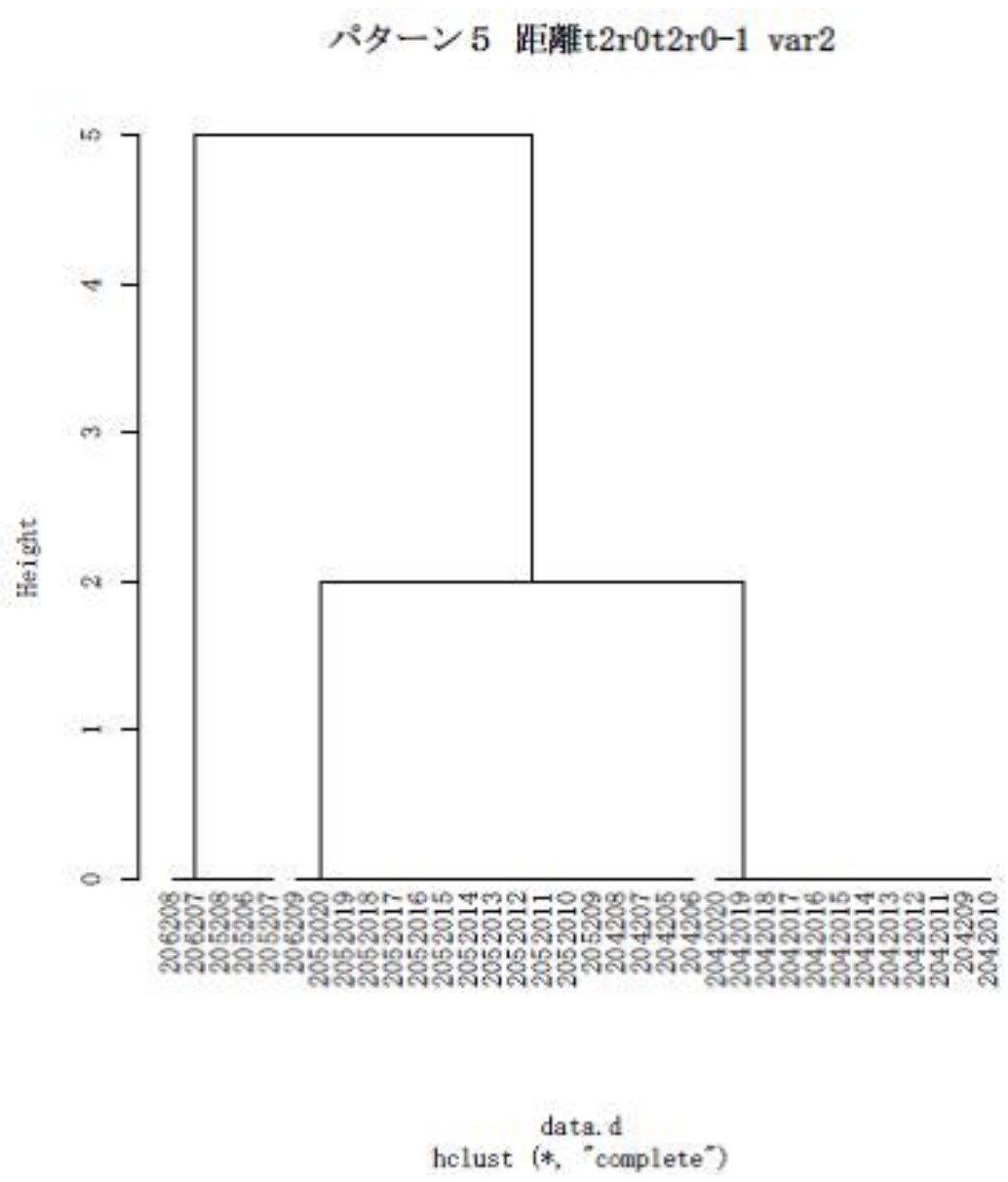
パターン５ 距離t1r3t1r3 var2

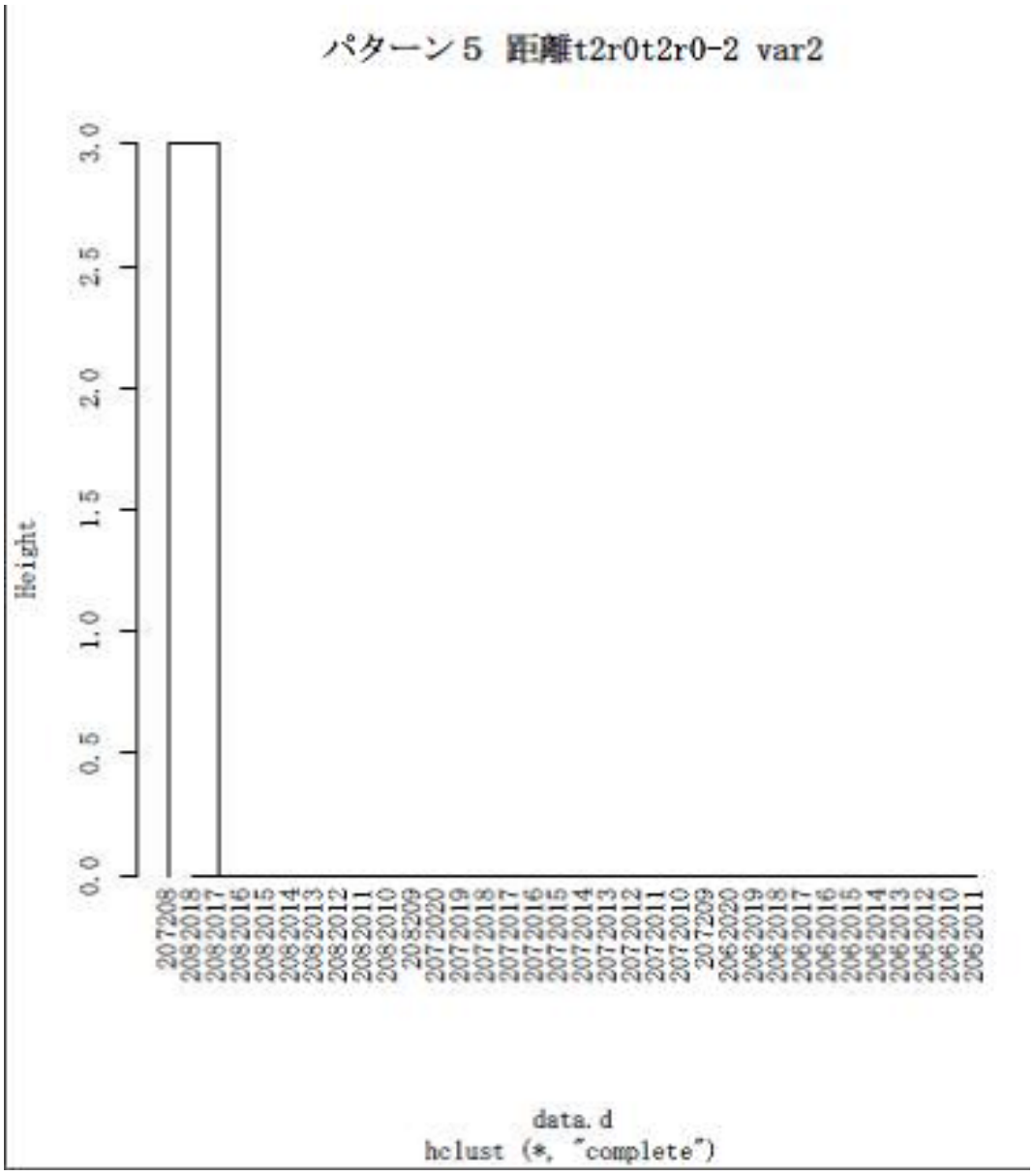


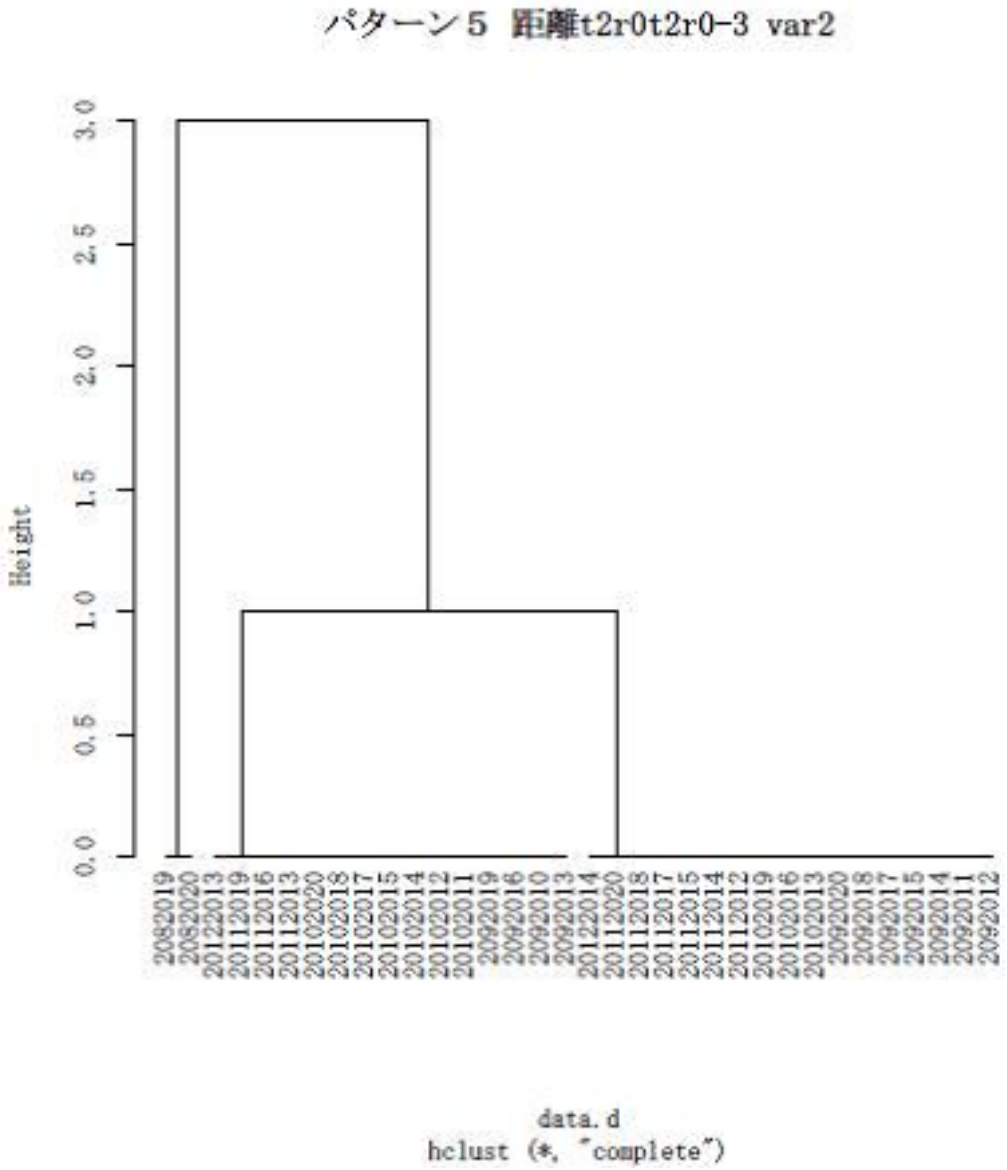

パターン５ 距離t1r4t1r4
Height
0
-1
14111412
14101412
14101411
1491412
1491411
1491410
1481412
1481411
1481410
148149
1471412
1471411
1471410
147148
147149
data.d
hclust (*, "complete")

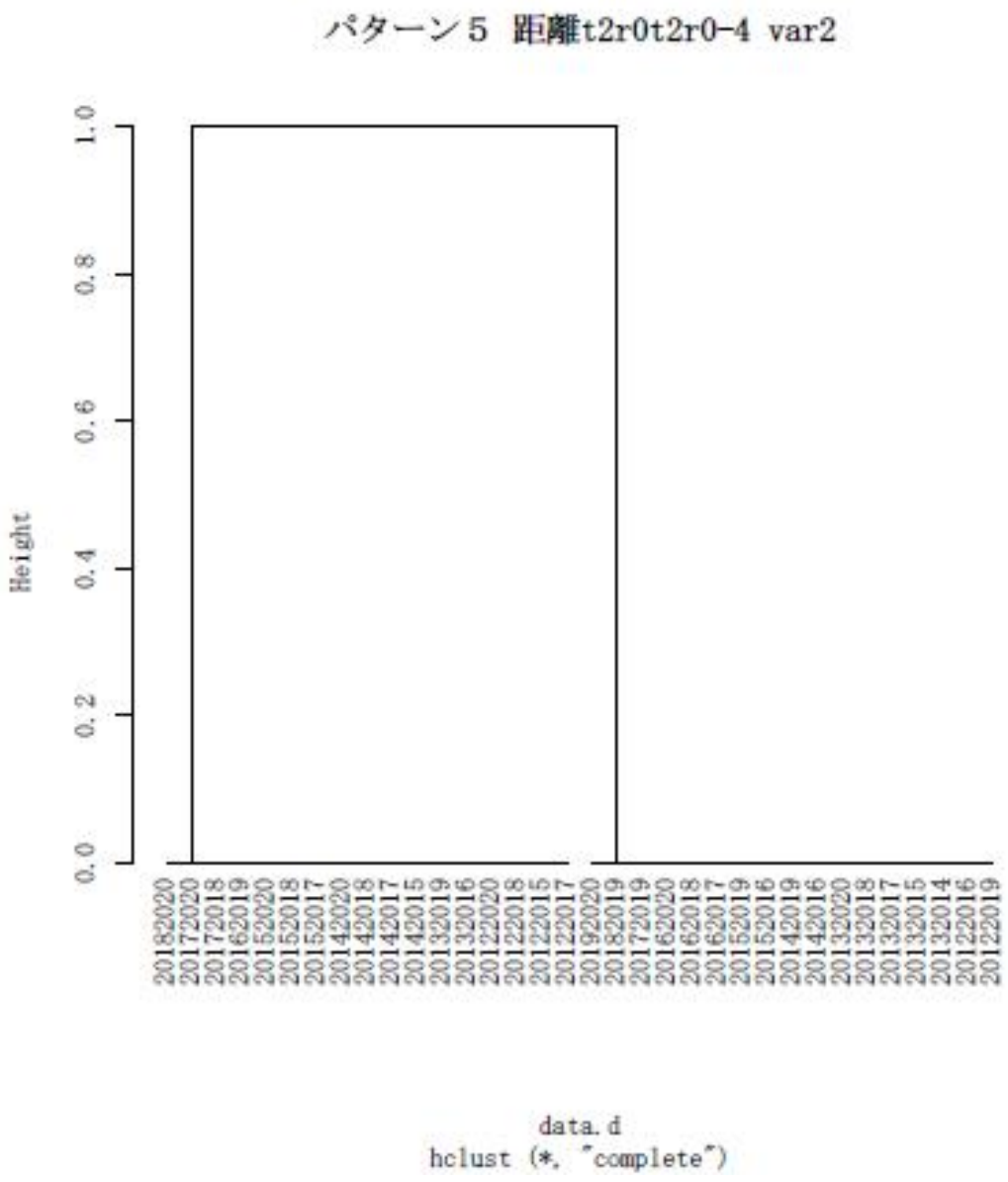
パターン５ 距離t1r5t1r5




パターン5 距離t2r0t2r0-1 var2
Height
0
1
2
3
4
5
206208
206207
205208
205206
205207
206209
205202O
205201
205201
205201
205201
205201
205201
205201
205201
205201
204207
204205
204206
204202O
204201
204201
204201
204201
204201
204201
204209
204210
data.d
hclust (*, "complete")


パターン５ 距離t2r0t2r0-2 var2
Height
0.0
0.5
1.0
1.5
2.0
2.5
3.0
207208
208 2018
208 2017
208 2016
208 2015
208 2014
208 2013
208 2012
208 2011
208 2010
208 209
207 2020
207 2019
207 2018
207 2017
207 2016
207 2015
207 2014
207 2013
207 2012
207 2011
207 2010
207 209
206 2020
206 2019
206 2018
206 2017
206 2016
206 2015
206 2014
206 2013
206 2012
206 2010
206 2011
data.d
hclust (*, "complete")


パターン５ 距離t2r0t2r0-3 var2
Height
0.0
0.5
1.0
1.5
2.0
2.5
3.0
20820 19
20820 20
20122013
20112019
20112016
20112013
20102020
20102018
20102017
20102015
20102014
20102012
20102011
20920 19
20920 16
20920 10
20920 13
20122014
20112020
20112018
20112017
20112015
20112014
20112012
20102019
20102016
20102013
20920 20
20920 18
20920 17
20920 15
20920 14
20920 11
20920 12
data.d
hclust (*, "complete")


パターン５ 距離t2r0t2r0-4 var2
Height
0.0
0.2
0.4
0.6
0.8
1.0
20182020
20172020
20172018
20162019
20152020
20152018
20152017
20142020
20142018
20142017
20142015
20132019
20132016
20122020
20122018
20122015
20122017
20192020
20182019
20172019
20162020
20162018
20162017
20152019
20152016
20142019
20142016
20132020
20132018
20132017
20132015
20132014
20122016
20122019
data.d
hclust (*, "complete")