平成27年度卒業論文

インターリービングモデルの分散アルゴリズムシミュレータ

平成28年2月12日

指導教員 濱田幸弘

明石工業高等専門学校

電気情報工学科

報告者 E1110 唐澤弘明

目 次

第1章　はじめに

分散システムは複数のコンピュータが通信ネットワークを介して相互に情報を伝達しあうことで処理を進めるコンピュータシステムである。昨今のコンピュータの低価格化と通信速度の向上は分散システムの応用範囲を劇的に拡大させた。実際に分散システムはコンピュータネットワーク上のルーティング制御やWWW(World Wide Web)、航空機制御、コンピュータグラフィクスの分散描画などの様々な分野で活用されている。

このような事情で分散システムとそのシステム上で実行されるアルゴリズム、すなわち分散アルゴリズムは近年非常に活発に研究されている。ところが分散アルゴリズムの研究には避けられない問題が付随する。それは分散システムが生来的に持ち合わせている複雑さである。分散システムを考える際に考慮しなければならない項目は、プロセス数や通信遅延、プロセスの故障、メッセージ配送のランダム性など多数存在する。これらのパラメータはアルゴリズムの実行過程を複雑にし、同一のアルゴリズムが様々な実行結果を生じる原因となる。それらをすべて手作業で検証することは非常に困難である。そこで考え出されたのが分散アルゴリズムシミュレータという類のソフトウェアである。分散アルゴリズムシミュレータは分散アルゴリズムを実行するための環境、つまり分散システムをユーザに提供し、アルゴリズムの検証を容易にする。このソフトウェアにより研究者はアルゴリズムそのものの考案・実装に専念できるようになり、煩わしい分散システムの実装をせずに済むようになった。

既存の分散アルゴリズムシミュレータにはBen-AriのDAJ(Distributed Algorithms in Java)[1]などがある。DAJは分散アルゴリズムを記述するためのJavaのクラスライブラリを提供し、そのライブラリを利用したアルゴリズムのシミュレーションと可視化を行う。また、弘田らはD^2AS(Distributed Distributed Algorithms Simulator)という分散アルゴリズムシミュレータを開発した[7]。このシミュレータは大規模な分散システムのシミュレーションを行うためのプログラムで、イーサネットにより結合された複数のコンピュータを実際に使用してシミュレーションを行うことができる。

本研究室では平成18年度からH-DAS(Hamada group - Distributed Algorithms Simulator)という分散アルゴリズムシミュレータの開発が行われてきた。H-DASは単一のコンピュータ上で動作するGUIプログラムで、仮想的な分散システム上で分散アルゴリズムの動作を視覚的に検証することができる。本研究ではH-DASの機能追加と不具合修正を行っている。

本論文の構成は次のとおりである。第2章では分散システムと分散アルゴリズムについて概説する。第3章では分散アルゴリズムシミュレータの役割を述べ、既存の分散アルゴリズムシミュレータの機能とH-DASの機能についてまとめる。第4章ではH-DASに行った機能追加と不具合修正を説明する。第5章ではH-DASの使用方法と既存のアルゴリズムを解説する。第6章では本研究のまとめとH-DASの今後の発展を述べる。最後に付録Aでシミュレータとサンプルアルゴリズムからなる付録CDの中身を説明する。

第2章　分散システムと分散アルゴリズム

2.1　分散システム

分散システムは、通信ネットワークにより結合された複数のプロセス上で、共通の目的の計算をするためのシステムである。分散システムは通常は複数のコンピュータを用いて構成されるため、物理的な共有メモリを持たない。さらにプロセス間のメッセージ通信には遅延があり、それがどれほどの大きさかもわからない。したがって2つのプロセスでそれぞれ実行されたイベントは、それらが同一メッセージの送信イベントと受信イベントでなければ、どちらが物理的に先に実行されたものなのかは区別できない。ここでいうイベントとは、プロセスが行う意味的にまとまった処理のことを指し、たとえば、関数の局所計算やプロセス間を行き交うメッセージの送信・受信などが当てはまる。

このようなモデルを用いる利点は以下のとおりである。

- 負荷分散、機能分散により処理を高速化できる。
- システムを構成しているいくつかのコンピュータが故障してもシステム全体で処理を継続できる。
- 複数の低価格なコンピュータでシステムを構成するため、低コストである。
- コンピュータの追加、削除を容易に行えるため拡張性が高い。

2.1.1　同期システムと非同期システム

分散システムは、プロセスの実行速度、通信遅延、局所時計の非同期性の程度によって、同期システムと非同期システムに大別される。以下の3つの条件を満たす分散システムを同期システムと呼ぶ[6]。

- プロセスの実行速度 (単位時間あたり実行される命令の数) に定数の下限が存在する。
- 通信にかかる通信遅延に定数の上限が存在する。
- 分散システム全体に共通して流れる時間において、すべてのプロセス P_i の局所時計の値 $clock_i$ は等しい。

これらの条件を1つでも満たさない分散システムが非同期システムであるが、その非同期さには程度の差が大きく、特に以下の3つの条件を満たす非同期システムを完全非同期システムと呼ぶ。

- プロセスの実行速度に対してどのような仮定も設けない。ただし、プロセスが命令を実行しようとしたときには、有限時間内にその命令が実行されることだけは仮定する。
- 通信遅延に対してどのような仮定も設けない。ただし通信遅延は有限である。
- 局所時間の差に対してどのような仮定も設けない。ただし、局所時間の修正は可能である。

分散システムが実現される通信ネットワークは様々であり、同期システムの条件を仮定することは難しいので、本研究では分散システムを完全非同期システムでない非同期システムと仮定している。H-DAS の分散システムの非同期さは以下のように特徴づけられる。

- プロセスの実行速度はメッセージの配送速度に比べ非常に高速であると仮定する。このとき、イベントは実行が開始されたならば瞬時に実行が終了する。
- 通信にかかる通信遅延に定数の上限が存在する。
- 局所時間の差に対してどのような仮定も設けない。ただし、局所時間の修正は可能である。

2.1.2 分散計算のモデル

分散計算は分散システム上で生起するイベントの集合としてモデル化できる。イベントの集合には順序関係が定義され、その関係の種類により分散計算のモデルは以下の 3 つに分けられる [6]。

Interleaving Model 分散システム上で生起するすべてのイベントの間に全順序が定まる。

Happend Before Model 異なるプロセスで生起するすべてのイベントの間に半順序が、同一のプロセスで生起するすべてのイベントの間に全順序が定まる。

Potential Causality Model 分散システム上で生起するすべてのイベントの間に半順序が定まる。

H-DAS では Interleaving Model を採用している。 Happend Before Model や Potential Causality Model の分散計算も Interleaving Model を用いて実現可能である。なぜなら、Happened Before Model の 1 つの計算結果は有限個の Interleaving Model による計算結果と等価であり、 Potential Causality Model の 1 つの計算結果は有限個の Happened Before Model による計算結果と等価であるからである。

2.2 分散アルゴリズム

分散アルゴリズムは、ある共通の目的を達成するために分散システム上のプロセスが行う計算の手順を定めたものである。分散システムには共有メモリが存在しないため、アルゴリズム実行開始直後の各プロセスは自身の局所情報しか知ることができない。そのため分散アルゴリズムでは、各プロセスがメッセージの送受信により互いの情報を交換することで処理を進める。

分散アルゴリズムを実行中のプロセスの状態は、命令を実行中の実行状態と、他のプロセスからのメッセージを待っている待機状態の 2 つに分けられる。分散アルゴリズムの実行は、1 つ以上のプロセスが自発的に処理を始めることによって開始される。自発的に処理を始めるプロセスのこと始動プロセスという。始動プロセス以外のプロセスは他のプロセスからのメッセージを受け取ってからアルゴリズムの実行を開始する。

2.2.1 分散アルゴリズムの評価

分散アルゴリズムの性能を評価するための尺度として、通信複雑度と時間複雑度がある。アルゴリズムの実行開始から実行終了までにプロセスの間で交換されたすべてのメッセージの総和をメッセージ複雑度といい、プロセス間で交換されるメッセージに含まれるすべてのビット数の総和をビット複雑度という。両者を合わせて通信複雑度という [6]。時間複雑度は実行開始から実行停止までに生じ

たメッセージの最長鎖に属するメッセージの総数である。ここで鎖とは最初のメッセージを除く他のすべてのメッセージが直前のメッセージの受信によって生じたようなメッセージ列のことをいう。

良い分散アルゴリズムであればあるほど、問題を解決するために必要な情報を効率的に交換していると予想されるので、通信複雑度がアルゴリズムの良さの尺度として適切であることがわかる。

アルゴリズムの実行速度も評価のための重要な尺度である。逐次アルゴリズム (分散アルゴリズムでない普通のアルゴリズム) ならば、アルゴリズムの実行速度はアルゴリズムが終了するまでの基本演算の数によって評価できる。しかし、分散アルゴリズムではプロセスが他のプロセスからのメッセージを待つ状態にあることも多く、単純に基本演算の数で実行速度を評価することはできない。そこで考えだされたのが時間複雑度である。時間複雑度は、プロセスが一度に可能なかぎりのメッセージを送信するという仮定のもとで、アルゴリズムの実行時間の評価とみなすことができる。

2.2.2 論理時計

分散システムを構成する複数のコンピュータは共有時計を持たない。そのため、システム上で生起したイベントの発生順をとらえるには工夫が必要となる。そこで、各イベントに整数あるいは整数のベクトルで表現される時刻印を付与し、それをイベントの生起順序の判別に用いる。それが論理時計である。

論理時計には様々な種類が存在するが、H-DAS では Direct-Dependency Clock を用いている。この時計を用いる場合、プロセスは通信のたびにメッセージにただ 1 つの整数だけを付与するため、通信パケットの大きさがほとんど変化しない。以下に Direct-Dependency Clock のアルゴリズムを Java 言語により記述する。

Code 2.1: Direct-Dependency Clock

```
1  public class DirectDependencyClock {
2      public int[] clock;
3      int myId;
4
5      public DirectDependencyClock(int numberOfProcess, int id) {
6          myId = id;
7          clock = new int[numberOfProcess];
8
9          for (int i = 0; i < numProc; i++) {
10             clock[i] = 0;
11         }
12
13         clock[myId] = 1;
14     }
15
16     // internal action
17     public void tick() {
18         clock[myId]++;
19     }
20
21     public void sendAction() {
22         // sentValue = clock[myId];
23         tick();
24     }
25
26     public void receiveAction(int sender, int sentValue) {
27         clock[sender] = Util.max(clock[sender], sentValue);
28         clock[myId] = Util.max(clock[myId], sentValue) + 1;
29     }
30 }
```

論理時計に Direct-Dependency Clock を用いるならば、プロセスのクラスは `DirectDependencyClock` クラスをメンバに持つ。プロセスは内部イベント (メッセージの送信・受信を伴わないイベント) を実行するとき `DirectDepandencyClock` クラスの `tick` メソッドを実行し、送信イベントを実行する際には `sendAction` メソッドを、受信イベントを実行する際には `receive` メソッドを実行させる。

この時計を用いるとき、状態 s にあるプロセス P_i と状態 t にあるプロセス P_j を考え、状態 t は状態 s より因果的に後の状態なのかどうかを知りたければ、P_i の時計を $clock_i$、P_j の時計を $clock_j$ とし、$clock_i[i] \leq clock_j[i]$ が成り立つかどうかを見ればよい。もちろん、状態 s が状態 t より因果的に後の状態なのであれば、$clock_j[j] \leq clock_i[j]$ が成り立つ。どちらも成り立たないというときは状態 s と状態 t は因果的に関係ない状態ということである。

図 2.1 は Direct Dependency Clock の実行例である。黒点の上の (x, y, z) は各プロセスのその時点での論理時計の内容を表している。つまり s_0 の上のものは $clock_0[0] = 1, clock_0[1] = 0, clock_0[2]$=0 ということ表している。また、ななめの矢印はメッセージ送信を表しており、そのラベルはメッセージに付加される時刻印である。この例では、s_0 の後に s_3 の状態になっているので、$clock_0[0] \leq clock_1[0]$ が成り立っている。どちらが因果的に先に生じたともいえない s_2 と s_3 では、$clock_2[2] \leq clock_1[2]$ も $clock_1[1] \leq clock_2[1]$ も成り立たない。

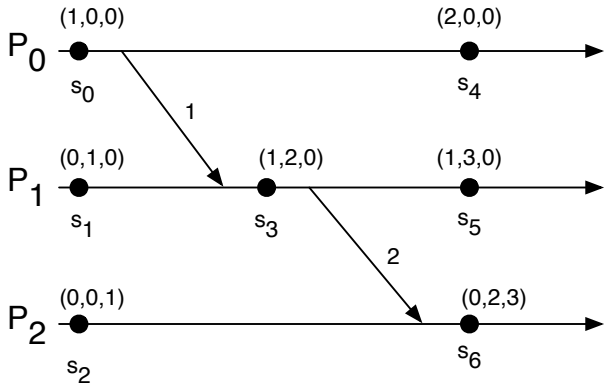


図 2.1: Direct-Dependency Clock の実行例

2.2.3 コータリー

分散システム上の複数のプロセスがある 1 つの資源を占有的に使用したいと同時に望むとき、資源を使用できるプロセスはただ 1 つに決定されなければならない。このような問題を相互排除と呼ぶ。コータリーは純粋に数学的な概念で、分散アルゴリズムではこれを相互排除のために使う。

U をプロセスの集合とし、以下の 2 条件を満たす U の部分集合 Q_i(コーラムと呼ばれる) の集合をコータリーと呼ぶ。

1. すべての対 i と $j(1 \leq i, j \leq N)$ に対して、$Q_i \cap Q_j \neq \phi$
2. すべての対 i と $j(1 \leq i, j \leq N, i \neq j)$ に対して、$Q_i \nsubseteq Q_j$

コータリー C を用いた相互排除では、プロセスはあるコーラム $Q \in C$ に含まれるすべてのプロセスから資源の使用許可を得る必要がある。C に含まれるすべての 2 つのコーラムは少なくとも 1 つのプロセスを共有するので、どのプロセスも同時に複数のプロセスに資源使用の許可を与えることはないという約束をすれば、複数のプロセスが同時に資源を使用できるという状況にはならない。

コータリーに基づく相互排除のアルゴリズムは耐故障性に優れている。通信ネットワークが完全グラフで、通信リンクが絶対に故障しないような理想的な分散システムでは、少なくとも 1 つのコーラムに属するすべてのプロセスが正常であるならば相互排除のアルゴリズムは正常に動作するからである。

H-DAS では Majority、Singleton、CWlog の 3 つのコータリーを使用できる。以下にそれぞれのコータリーと、CWlog の説明に用いる Crumbling Walls というコータリーについて説明する。U をプロセスの集合、n をプロセスの数とする。

Majority サイズが $k = \lceil n/2 \rceil$ である U のすべての部分集合からなる集合

Singleton $P \in U$ を任意のプロセスとし、$\{\{P\}\}$ なる集合

Crumbling Walls プロセスをいくつかの行に分けて、その行を縦に並べる。ただし行の長さはそれぞれ違ってもよい。コーラムは、ある行に含まれるすべてのプロセスに、その行の下にあるすべて行から 1 つずつプロセスを加えて構成される (図 2.2)。

CWlog Crumbling Walls の 1 つ。行に含まれるプロセスの個数が i 行では $\log_2 2i$ 個と、一番上の行から下の行にかけて対数的に増加する。

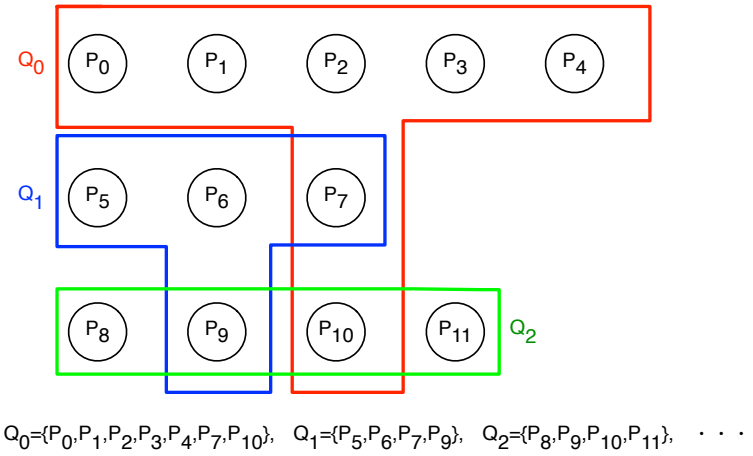


図 2.2: Crumbling Walls の例

第3章　分散アルゴリズムシミュレータ

3.1　分散アルゴリズムシミュレータの役割

第1章で述べたとおり、分散アルゴリズムは社会のいたるところで活用されている。しかし、分散アルゴリズムが生来的に備える複雑さのために、それを考案・検証するのは大変に難しい。分散アルゴリズムシミュレータはそのような問題に応え、分散アルゴリズムの開発者にその検証環境を提供するものである。

分散アルゴリズムの検証は、どのようなアルゴリズムであっても概ね次のような作業からなる。すなわち、

1. 初期大域状態の設定
2. イベントの生起
3. プロセス・通信リンクの状態遷移
4. 大域状態の確認

である。アルゴリズムが複雑になると、このような作業を手作業で行うのは難しくなる。そのため、分散アルゴリズムが開発者の想定通りに動いているかを検証するために、開発者を補助するツールが必要になる。それが分散アルゴリズムシミュレータである。

3.2　既存の分散アルゴリズムシミュレータ

既存の分散アルゴリズムシミュレータには単一のコンピュータ上で分散システムを模倣する Ben-Ari による DAJ(Distributed Algorithms in Java)[1] や、実際に複数のコンピュータ上で分散アルゴリズムのシミュレーションを実行できる弘田らの D^2AS(Distributed Distributed Algorithms Simulator)[7] などがある。ここでは H-DAS とシミュレーションの形式が似ている DAJ について説明する。

DAJ は教育目的に作成された分散アルゴリズムシミュレータで、分散アルゴリズムを対話的に動作させてその様子を動的に観察することができる。アルゴリズム例として 14 のアルゴリズムのソースコードが付属しており、自作のアルゴリズムを動作させることもできるようになっている。ユーザが記述するアルゴリズムは非常にシンプルなプログラムインターフェイスのクラスライブラリを利用して簡単に作成できる。DAJ が開発される以前は、PVM[3] や MPI[4] などの高機能だが複雑で巨大なライブラリが多かった。このようなライブラリは低レイアの技術的な問題を含み、また内部の情報が隠蔽されていて観察できないなど、教育目的の利用には適さなかったため DAJ が開発された。

図 3.1 は DAJ の外観である。画面を四等分しているブロックのそれぞれが分散システムを構成するプロセスの状態を表している。また、各プロセスの状態の下にある Request や Reply などのボタンで、各プロセスのイベントを実行できる。さらに最も下部に配置されているボタンでは以下のことができる。

FileX, Step, Log on/off, Auto on/off Buttons ログの記録、及びログのステップ実行と自動実行

Prompt on/off 各プロセスがメッセージを送信できるプロセスを表示する行の表示・非表示の切り替え

Trace on/off 実行されたイベントをテキストとして表示するウィンドウの表示・非表示の切り替え

Graphics on/off プロセスの関係をグラフ化して表示するウィンドウの表示・非表示の切り替え

上記の機能を含めてまとめると DAJ ではおおまかに以下のことができる。

- プロセス数を 1 から 6 個の間で自由に設定できる。
- 自作のアルゴリズムの実行
- アルゴリズムの実行例の保存
- アルゴリズムの実行例の再生
- イベントの実行履歴のテキスト表示
- プロセスの関係のグラフ化 (一部のアルゴリズムに限られる)

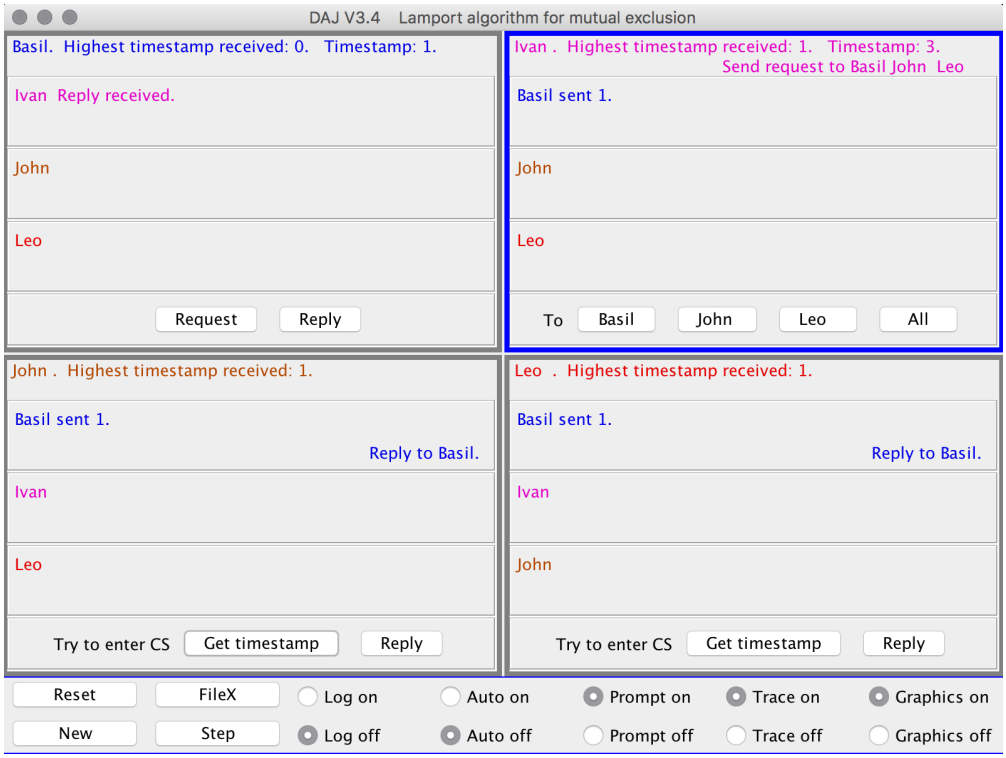


図 3.1: DAJ の外観

3.3 本研究室で開発されてきた分散アルゴリズムシミュレータ

本研究室では平成 18 年度から H-DAS(Hamada group - Distributed Algorithms Simulator) という分散アルゴリズムシミュレータが開発されてきた。H-DAS は分散アルゴリズムを記述するためのデスクリプタというプログラムと、出来上がった分散アルゴリズムをシミュレートするシミュレータというプログラムからなる。

図 3.2 はデスクリプタの外観である。このプログラムを用いてユーザは以下の 4 つの工程を行い、シミュレータで利用するための das という拡張子のファイルを生成する。

1. アルゴリズムのモデル情報の入力
2. テンプレートコードの出力
3. アルゴリズムの実装
4. ビルド

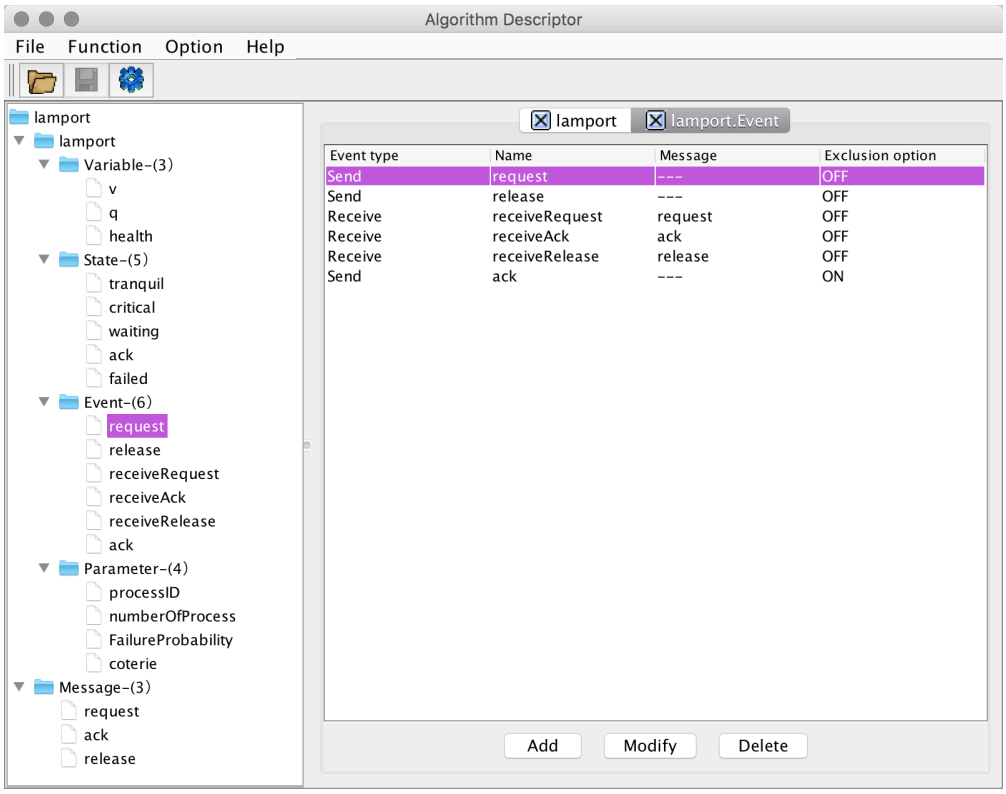


図 3.2: H-DAS の Descriptor の外観

図 3.3 はシミュレータの外観である。このプログラムでデスクリプタにより生成した das ファイルを読み込みシミュレーションを開始する。3.3.1 項で簡単に機能の紹介を行う。 詳細な使用方法の説明は第 5 章で行う。

3.3.1 機能

昨年度までに開発された H-DAS の主な機能について述べる。

- ユーザが Java 言語によるアルゴリズムを記述できる。
- 記述されたアルゴリズムのコンパイルが可能、コンパイル成功時にクラスファイルとアルゴリズムに関するファイル (拡張子 das) を作成する。
- プロセスの個数を 1 から 1000 までの間で設定できる。
- メッセージを自動で配送できる。

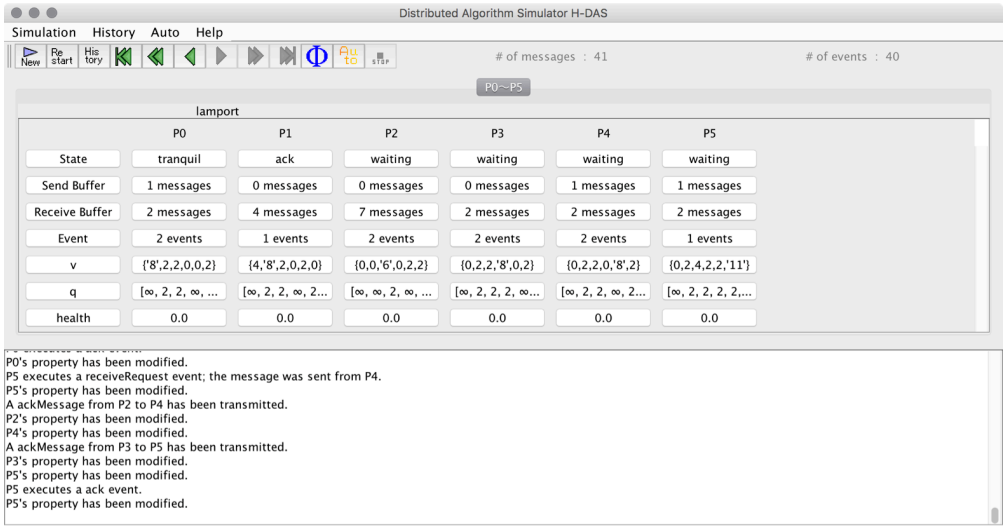


図 3.3: H-DAS の Simulator の外観

- メッセージの通信遅延を模倣する。
- 通信遅延の上限は 0 から 100 まで設定できる。
- メッセージの総数を集計し表示する。
- 通信路の FIFO 性を設定できる。
- 実行したシミュレーションの履歴をテキスト形式で保存できる。
- 実行したシミュレーションの履歴をコマンド形式で保存できる。
- コマンド形式で保存されたシミュレーションを再現できる。
- イベント発生時にテキストエリアに結果を表示する。
- 空イベントを実行できる。
- シミュレータとデスクリプタのヘルプマニュアル (英語) を参照できる。
- 実行したイベントを取り消し、元の状態に戻すことができる。
- イベントの生起を自動化できる。
- プロセスの故障を模倣できる。
- コータリーの選択ができる。

3.3.2 問題点

本研究で改善を加えた H-DAS の問題点を述べる。

- シミュレーションを開始する際の設定画面が逐次的に出てくるので、一覧性がない。
- イベントの生起回数が表示されないので、シミュレーションが開始してからの進度がわかりにくい。

- 大量のイベントを発生させ続けるとプログラムが強制終了する。
- 通信路に FIFO 性を設定したときに通信遅延の制約が守られない。
- 大量のプロセスからなるシステムをシミュレートすると実行速度が極端に低下する。
- 特定の環境でプロセスの状態を表す色付けがなされない。

第4章　今年度改良した部分

本研究では本研究室で開発されてきた分散アルゴリズムシミュレータ H-DAS の機能追加と不具合の修正を行った。この章ではその詳細について述べる。

4.1　機能追加

4.1.1　シミュレーション開始時の設定画面の統合

H-DAS は平成 18 年から開発されてきたソフトウェアである。そのため、初期のヴァージョンと比べると多数の機能が存在し、シミュレーションの自由度も向上している。しかし、多くの機能が追加されたことで、シミュレーションを開始する前に設定しなければならない項目も増えてしまった。今やその項目はプロセス数・プロセスの故障確率・メッセージの通信遅延の上限・コータリーの要否・使用するコータリーの種別・FIFO 性の有無の 6 項目となった (図 4.1)。このように多くの項目を別個で設定するのはユーザにとって無意味な手間であるし、シミュレーションを開始する際に項目のチェックができないのでミスを生みやすい。このような事情により、設定画面を統合しユーザがストレスなくシミュレーションを開始できるようにする必要があった。

図 4.2 は改良されたシミュレータの統合された設定画面である。このように、シミュレーションを開始する際の設定項目は一目瞭然となった。

4.1.2　イベント発生回数の表示

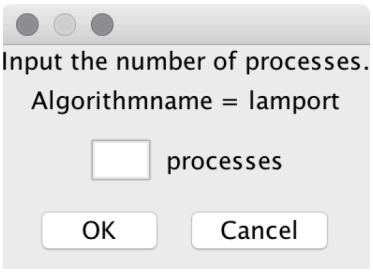
図 4.3 はシミュレータの上部を切り取ったものである。ここに表示されている`# of events: [number]` と書いてあるのがイベント数を表している。この数値はユーザがプロセスのイベントを実行するか空イベントを実行するかすると 1 ずつ上昇する。この数値はユーザの操作のたびに上昇するため、シミュレーションを開始してからどれほどの時間が経過したかを示す一つの目安となる。この数値を利用すれば、たとえば、実行したシミュレーションを保存しそれを履歴から再実行する際に、この数値を見て動作を確認したい箇所の目算をつけることなどができる。

4.2　不具合修正

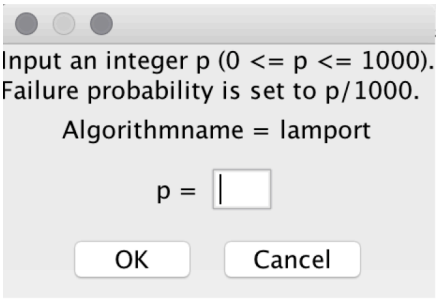
4.2.1　大量のイベントの連続発生でプログラムが強制終了する不具合

従来のシミュレータには多数のイベントを生起させ続けるとプログラムが強制終了してしまうという問題があった。この問題はイベント生起の自動化時に特に顕著で、たとえば、プロセス数を 16 個に限定しても、30 分ほど自動実行させているだけでプログラムが停止してしまうほどであった。強制終了ではそのときのプロセスの状態なども全くわからないので大変に不便である。

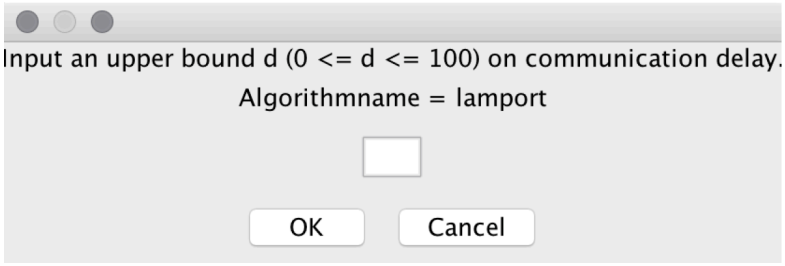
このような不具合が生じる原因は、実行したイベントを初期状態になるまで取り消すことができるという機能にある。この機能を実現するために H-DAS では実行されたイベントの情報をすべて主記



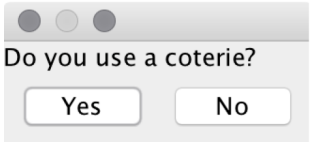
(a) プロセス数



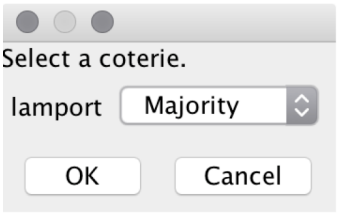
(b) プロセスの故障率



(c) メッセージの通信遅延の上限



(d) コータリーの要否



(e) 使用するコータリーの種別

(f) FIFO性の有無

図 4.1: 従来のシミュレータで入力する必要のあったウィンドウの一覧

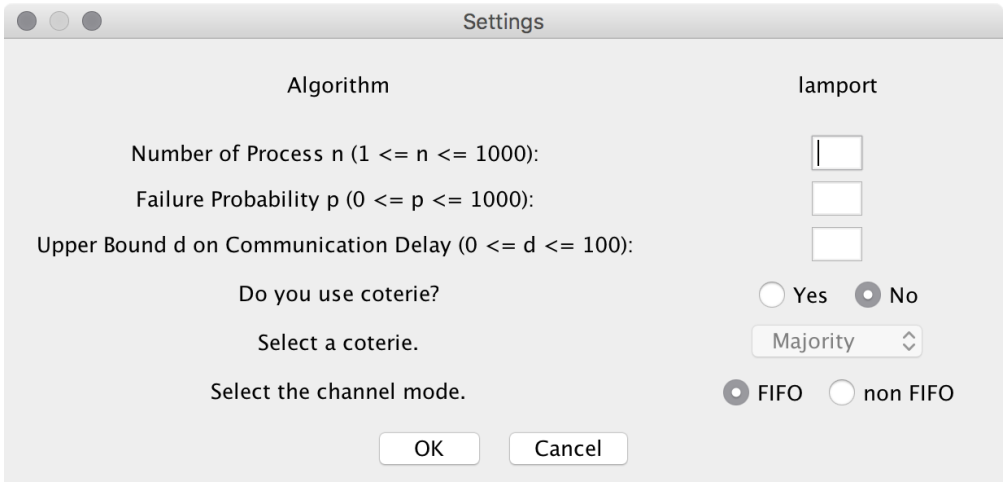


図 4.2: 改良されたシミュレータで 1 つに統合された設定画面

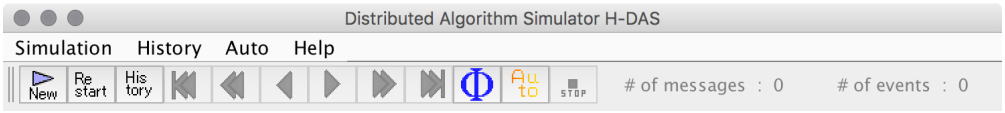


図 4.3: シミュレータ上部の`# of events: [number]` はイベントの発生回数を表している。

憶上に記憶している。そのため、大量のイベントを生起させると主記憶の容量が足りなくなってしまう。こうして発生したメモリ不足によりプログラムの強制終了が起きてしまうのであった。

この問題を解決するためのアプローチは、1 つ 1 つのイベントが持つ情報量自体を少なくする方法と、主記憶上に保存するイベントの量を制限し残りは補助記憶に保存するという方法の二通りが考えられる。本研究では両方の解決法を施すこととし、主記憶上に保存しなければならない情報を制限すると同時に、補助記憶に保存しなければならない情報もなるべく少なくするという方法を取った。

まず各イベントの情報量を削減する方は、イベントが持つプロセスの集合を表すデータ構造を整理することで簡単に実現できた。もともとプロセスの集合は `java.util.HashSet` として実装されていたが、プロセスには分散システムのプロセス数を N として 0 から $N-1$ までの連番が振られているので、データ構造を `java.util.ArrayList` に変更することで、使用するメモリ容量を半分に減らすことができた。本研究では実施していないが、各イベントが持つイベントを実行する前と後の 2 つの大域状態の情報をイベントを制御するクラスに持たせることで情報量は更に半分まで減らせる。これは、あるイベントの持つそれを実行した後の大域状態とそのイベントの次に実行されたイベントが持つ実行前の大域状態は全く同じであるからである。また、分散システムのプロセスのほとんどはただ 1 つのイベントを実行しただけでは状態が変化しない。そのため、状態が変化したプロセスだけを抜き出して保存すれば必要な記憶容量はなおいっそう少なくなる。

次にイベントを補助記憶に書き出す方法だが、これも問題なく実装することができた。主記憶に保存されるイベントの数は 256 個とし、それらは `loadedEvents` という表に登録される。もし生起したイベントの数が 256 個を超えたら、表にあるイベントのうち登録された時刻が最も古いものから順に補助記憶に書き出される。イベントの書き出し先はプログラムの実行ディレクトリに作られる dump ディレクトリとした。プログラム実行中にユーザが dump ディレクトリを削除するなどして補助記憶にあるイベントの情報にアクセスできなくなると、イベントを取り消したり、シミュレーションの履歴を保存するなどの操作が出来なくなってしまうので注意が必要である。

以上のような改善を施したことで、プログラムが主記憶を使い果たし強制終了することはなくなった。加えて、イベント自体の情報量を削減したことで、シミュレーションを再実行可能な形式で保存する場合に必要なデータ容量も少なくなった。この形式はデータのほぼすべてがイベントの集合によるもので占められていたからである。シミュレーションの結果を保存するのは主記憶ではなく補助記憶なので、プログラムが強制終了するといった重大な問題とはいえないが、この形式は場合によっては数 GB の大きさになることもあり、また複数のシミュレーションが保存される状況が想定されるので、データ量はなるべく削減するのが好ましい。

4.2.2 通信路に FIFO 性を付与したときに通信遅延の制約が守られない不具合

H-DAS ではシミュレーション開始時の設定で選択することによって通信路に FIFO 性を付与することができる。FIFO 性を付与した通信路では、プロセス P_i からプロセス P_j に向けて送信されたメッセージは必ずメッセージの発生順に到着する。また、H-DAS では FIFO の設定と同時に通信遅延の上限も設定する必要がある。従来のシミュレータでの FIFO を設定した際の通信遅延の決め方は、プロセス P_i からプロセス P_j に向けて送信されたメッセージの内、最も後に送信されたメッセージの通信遅延に通信遅延上限以内の正の整数をランダムで選び足し合わせたものを新たな通信遅延とするというものだった。これでは時刻 t と時刻 $t+1$ にプロセス P_i からプロセス P_j に向けて送信されたメッ

セージの後者の方は通信遅延が最大 $2D-1$(D を通信遅延上限とする) となってしまい、通信遅延の制約を満たさない。

この問題は、メッセージの発生順と到着時刻を別々に管理することで解決できた。この方法では、メッセージ m が発生した時刻を t、通信遅延上限を D とし、メッセージの発生と同時に $t_m = t + d(1 <= d <= D)$ という値を生成し、m は待ち行列 $Q_{message}$ に、t_m は優先度付き待ち行列 Q_{time}(小さい値が優先される) にそれぞれ保存する。ある時刻にプロセスに到着するべきメッセージがあるかどうかは Q_{time} の先頭の値を見て判断し、もしあるならば $Q_{message}$ からメッセージを取り出してプロセスに届ける。このとき Q_{time} からも先頭の値を取り除く。このようにすると、メッセージはその発生順に $Q_{message}$ から取り出されるので通信路の FIFO 性は保証される。また、メッセージ m_i が $Q_{message}$ の先頭から i 番目にあるとすると、Q_{time} の i 番目の値 t_i は、m_i が発生すると同時に生成された値よりも小さくなることはあっても大きくなることはありえない。したがって、通信遅延上限の制約も満たされる。

図 4.4 にプロセス P_i から P_j に向けて送られるメッセージの様子を示す。

	t	$Q_{message}$	Q_{time}
メッセージm_0発生, t_{m_0}=3 →	0	[m_0]	[3]
メッセージm_1発生, t_{m_1}=5 →	1	[m_0,m_1]	[3,5]
メッセージm_2発生, t_{m_2}=4 →	2	[m_0,m_1,m_2]	[3,4,5]
メッセージm_0到着,到着時刻=3 →	3	[m_1,m_2]	[4,5]
メッセージm_1到着,到着時刻=4 →	4	[m_2]	[5]
メッセージm_2到着,到着時刻=5 →	5	[]	[]

到着順(括弧内は実際の到着時刻): m_0(3), m_1(4), m_2(5)

このとおりFIFO性もあり通信遅延の上限の制約も満たしている。

図 4.4: プロセス P_i からプロセス P_j に向けて送られるメッセージの様子

以上のようにして通信路の FIFO 性と通信遅延の制約を満たすことができたが、この実装方法には問題があった。それはプロセス P_i からプロセス P_j に向かう複数のメッセージが同時に到着してしまうということである。H-DAS では通信ネットワークをすべての頂点の間に辺があるグラフ、つまり完全グラフだと仮定しており、プロセス P_i からプロセス P_j に向かうすべてのメッセージは同じ通信路を用いるので、メッセージは必ず順番に到着し、同時に到着するということはありえない。厳密にいえば、H-DAS の内部ではメッセージは同時ではなく順番に到着し、処理されている。しかしこれではユーザには大域時計を 1 進めただけで 2 つのメッセージの到着処理がなされているように見えてしまうので不便である。

この問題は、複数のメッセージの発生時刻が一致してしまったとき、一番初めに発生したものだけを本来の到着時刻に到着させ、それ以外のものの到着時刻を大域時刻で 1 だけ遅らせることで解決できた。遅らせた到着時刻でも同様の状況に陥る可能性はあるが、そのときも同じ処理を繰り返し実行すればよい。このようにすれば、メッセージは明らかに発生順に到着するので、FIFO 性については問題ない。また、通信遅延の上限の制約も問題なく満たされる。プロセス P_i からプロセス P_j に向け

て送信されるメッセージは同時に 2 つ以上発生することはなく、それぞれのメッセージが到着しなければならない上限の時刻は一致しない。よって、P_i から P_j に向けて送信された複数のメッセージが同時に到着するとき、その時刻が通信遅延の上限である可能性のあるメッセージは一番初めに発生したメッセージのみであり、これはその時刻に到着するので通信遅延の上限の制約を破ることはない。

4.2.3 大量のプロセスからなるシステムをシミュレートする際に実行速度が極度に低下する不具合

従来のシミュレータではプロセス数が多くなると実行速度が極端に低下する問題があった。プロセス数が多くなればなるほどイベントの実行毎にしなければならない処理が増えるので、これはある程度までは当然の動作であるのだが、従来のシミュレータのそれはあまりに極端なものであった。

この問題は自動実行時にイベントを実行するプロセスの選択方法に原因があった。その選択方法とは、分散システム上のすべてのプロセスからランダムでプロセスを選択し、そのプロセスに実行できるイベントがあればそれを実行するというものであった。実行するアルゴリズムにもよるが、これは大変に非効率的である。なぜならプロセスの状態によっては実行できるイベントが 1 つもない場合もあるからである。もし、分散システム上の多くのプロセスで実行できるイベントが 1 つもなかった場合、何度も何度もイベントを実行するプロセスの抽選を行うことになり、無駄に時間がかかってしまう。

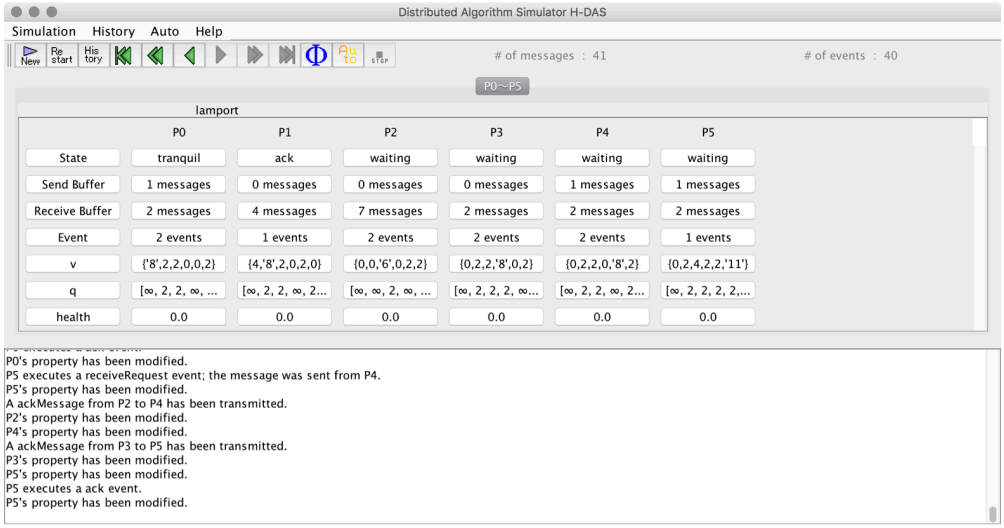
この問題は予めイベントを実行できるプロセスを抽出し、条件を満たすプロセスが 1 つもなければ何も実行せず、そのようなプロセスが存在すればそれらの中から選出を行うことで解決できた。

4.2.4 特定の環境でプロセスの色付けがなされない不具合

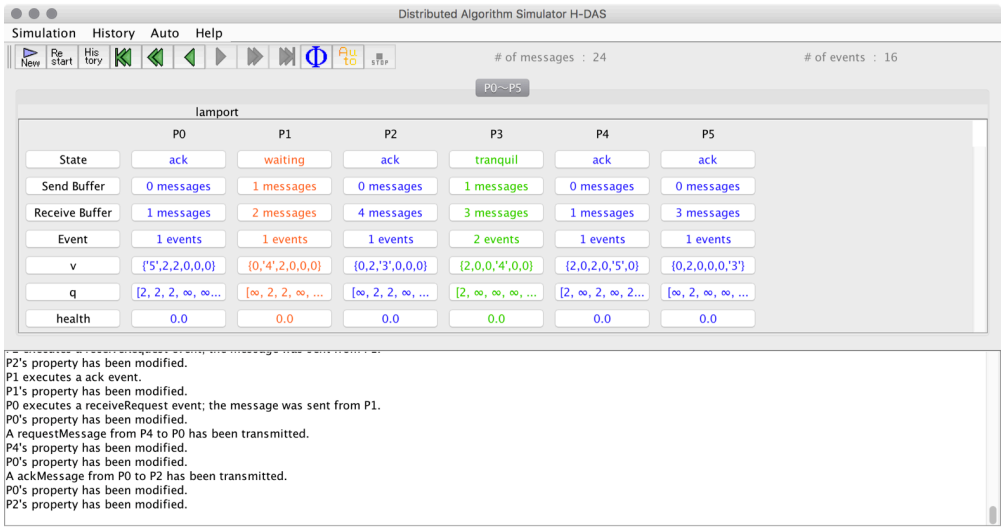
H-DAS ではプロセスを表すセルをプロセスの状態によって色付けするという機能があり、これは複数のプロセスの状態をひと目で判別でき、便利である。しかし従来のシミュレータでは特定の環境で色付けがなされないという問題があった。H-DAS は Java の実行環境のあるすべてのマシンで実行される可能性があると想定しているので、この不具合を修正する必要があった。

結論から言うと、この問題の原因は `javax.swing.JComponent` クラスの `opaque` フラグにあった。このフラグはコンポーネントの背景を透過するかどうかを表している。シミュレータのセルは `javax.swing.JButton` クラスで表現しているが、一部の環境では `opaque` フラグがデフォルトで `false` になっているようだった。

この問題はセルの初期化時に `opaque` フラグを `true` に設定することで簡単に解決できた。問題の解決前と解決後のシミュレータの画面を図 4.5 に示す。



(a) プロセスの色が表示されない問題を解決する前



(b) プロセスの色が表示されない問題を解決した後

図 4.5: シミュレータの画面

第5章　シミュレータの使用方法

5.1　動作環境

H-DAS は Java の実行環境 JRE(Java Runtime Environment) で動作するプログラムである。また、ユーザが記述したアルゴリズムをコンパイルするためには JDK(Java Development Kit)8[5] が必要となる。JDK は Oracle 社の Web サイトからダウンロードすることができる。

5.2　インストール

H-DAS のインストールは、付録の CD より H-DAS を適当なディレクトリにコピーすれば完了する。

5.3　アンイストール

H-DAS のアンインストールは、H-DAS のディレクトリを削除すれば完了する。

5.4　シミュレータの起動

H-DAS は、アルゴリズムを記述するためのデスクリプタとシミュレーションを行うためのシミュレータからなる。共に共通の画面から起動でき、起動画面を呼び出すには H-DAS ディレクトリ内の bin ディレクトリからコマンドラインに `java startup.Controller` と入力する。すると図 5.1 の起動画面が呼び出される。

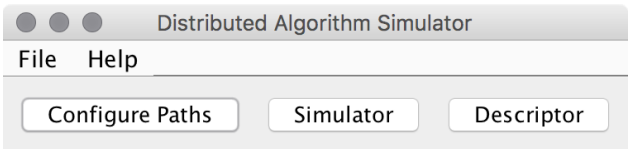


図 5.1: H-DAS の起動画面

シミュレータを起動する場合には [Simulator] を、デスクリプタを起動する際には [Descriptor] をクリックすればよいが、その前に Configure Paths で設定をする必要がある。図 5.2 はパスの設定画面である。ここで、tools.jar 及びシミュレーションを行う das ファイルのあるディレクトリの指定をしなければならない。なお tools.jar はデスクリプタで、das ファイルのあるディレクトリはシミュレータでしか使わないので、どちらか片方しか使用しない場合は使わない方の設定はしなくても問題ない。

5.5　分散アルゴリズムの記述

5.5.1　モデル情報の入力

分散アルゴリズムの記述に使用するデスクリプタの初期画面を図 5.3 に示す。

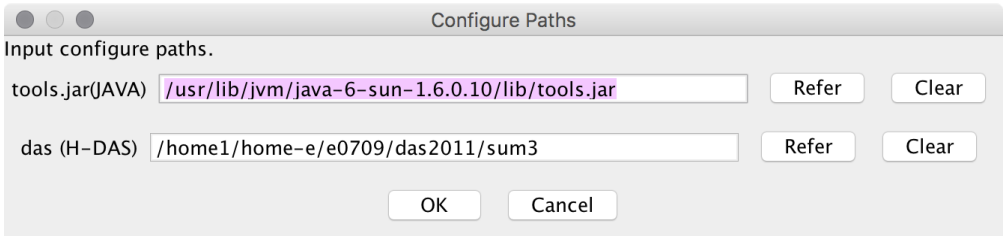


図 5.2: Configure Paths

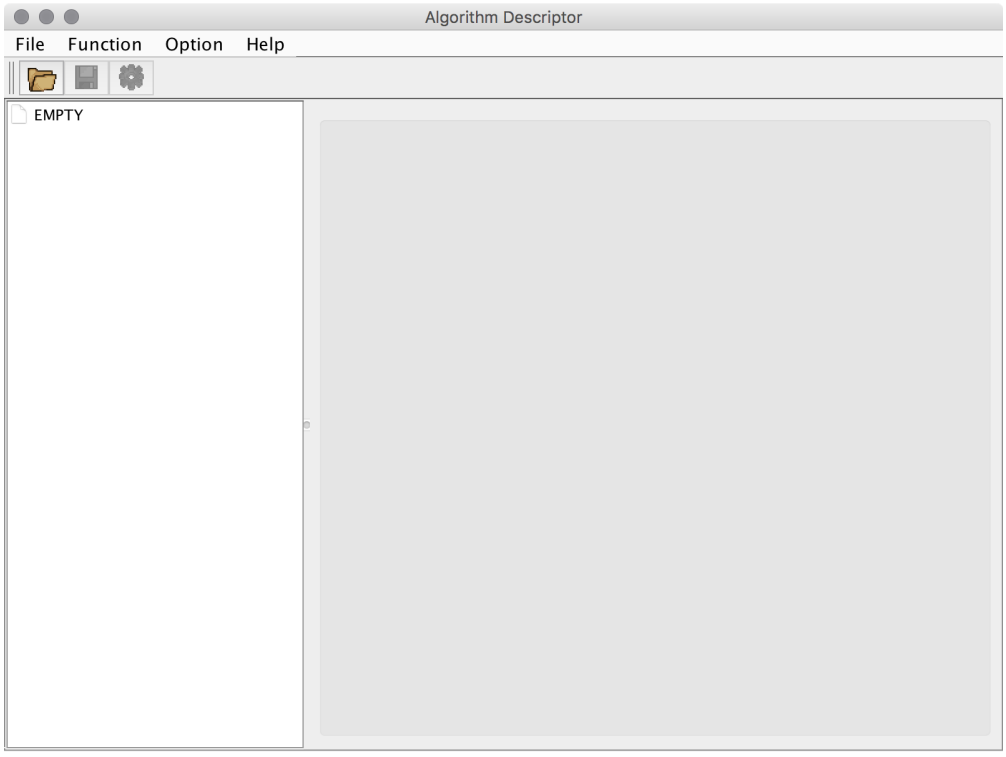


図 5.3: デスクリプタの初期画面

メニューから [File] → [New] を選択し、出現するダイアログでシステムディレクトリを指定する。次に出現するダイアログでは作成するアルゴリズムのシステム名を入力する。そうすると、図 5.4 のような画面になる。ここで、画面左のツリーからノードを選択すると、画面右にモデル情報タブが表示されるので、画面下部の [Add]、[Modify]、[Delete] ボタンからモデル情報を編集する。

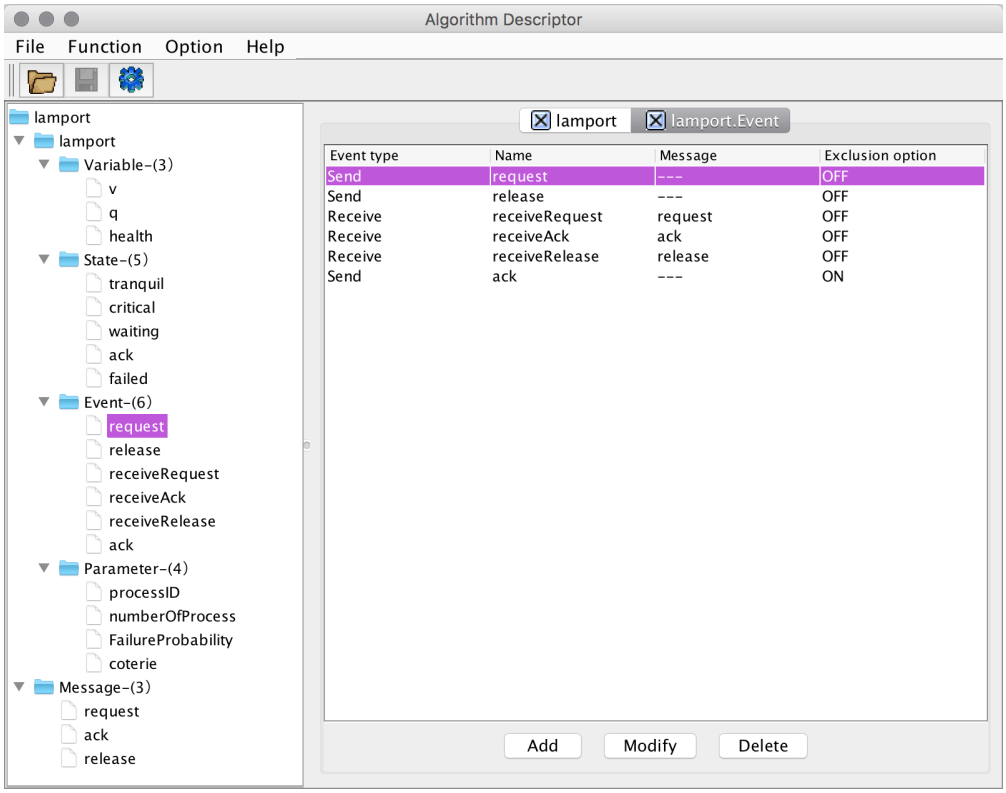


図 5.4: デスクリプタの使用画面

モデルはアルゴリズムごとに Variable モデル、State モデル、Event モデル、Parameter モデル、及び Message モデルがある。Event モデルの入力時に受信イベントで受信するメッセージの種別は、Message モデルとして入力したものから選択することになる。

デフォルトのパラメータとして、Variable モデルには `Health`、State モデルには `tranquil`、`critical`、`failed`、Parameter モデルには `processID`、`numberOfProcess`、`FailureProbability`、`coterie` がある。特に、State モデルの `failed` と Parameter モデルのすべてパラメータはシステムに必須であるため変更・削除してはならない。

Parameter モデルのパラメータの変数の型として選択できるのはラッパークラスの Integer 型と Double 型のみであるので注意が必要である。また、Message モデルのパラメータには int 型や double 型などの基本データ型が使用できる。

モデル情報の入力が完了したら、メニューから [Function] → [Output a skeleton code] を選択し、つづいて出現するダイアログでスケルトンコードを生成したいアルゴリズムを選択する。これによりシステムディレクトリ及び、その内部にスケルトンコードが生成される。ユーザはプロセス処理の記述に移行する前に、入力したモデル情報に誤りがないかを確認するため、図 5.5 のビルドボタンをクリックしビルドを実行することが推奨される。

図 5.5: ビルドボタン

5.5.2 プロセス処理の記述

プロセス処理の記述は、スケルトンコードの作成によってシステムディレクトリ内に生成された xxxProcess.java(xxx はアルゴリズム名) ファイルの内容を編集することによって行う。記述する内容は、デスクリプタで入力したイベントごとにその動作と、イベントの前提条件が整っているかどうかを表す precondition メソッド、受信イベントにおける対応メッセージが受信バッファに存在するか否かという条件である。

また、デッドロックを自動的に検出できるようにするには、各メッセージが送信されるときに返信を要求するかどうかを明確にする必要がある。メッセージが返信を要求するならば、スケルトンコードの xxxMessage.java(xxx はメッセージ名) の内容のうち、

```
public class xxxMessage extends Message implements Serializable
```

という行を

```
public class xxxMessage extends WantResponse implements Serializable
```

と書き換えなければならない。返信を要求しないメッセージについてはそのままでよい。

ユーザがプロセス処理を記述するにあたって使用することのできるメソッドのうち主要なものを表 5.1 に示す。

表 5.1: ユーザが使用可能な主なメソッド

メソッド名	所有クラス	処理内容
getProcessID()	MutualExclusionAlgorithm	プロセス番号の取得
getNumberOfProcess()	MutualExclusionAlgorithm	プロセスの総数の取得
sendEvent()	DirectDependencyClock	ベクトル時計の自身の時刻成分を 1 増分
dispatchMessage()	MessagePost	第 1 引数で指定したプロセス番号に、第 2 引数で指定したメッセージを送信
dispathToOthers()	MutualExclusionAlgorithm	メソッドを呼び出したプロセス以外の全プロセスに、dispathMessage を呼び出して第 2 引数で指定したメッセージを送信
getXXX()	メッセージモデルのクラス	メッセージモデルの保持変数を取得
receiveEvent()	DirectDependencyClock	受信したメッセージについて、ベクトル時計の送信者の時刻成分と自身の時刻成分を比較して更新
getAlgorithmName()	MutualExclusionAlgorithm	プロセスのアルゴリズム名を取得

5.5.3 ビルド

プロセス処理の記述が完了したら、デスクリプタの [Build] ボタンによりビルドを実行する。一度ビルドしたアルゴリズムについても編集の都度ビルドボタンをクリックすることでアルゴリズムを更新できる。ビルドをするとダイアログが出現し、プロセス処理の記述に含まれるエラーやシステムファイルのバージョンエラーが表示される。エラーが生じた場合、システムは直前のビルド成功後の状態から更新されない。ビルド成功時のダイアログを図 5.6 に示す。

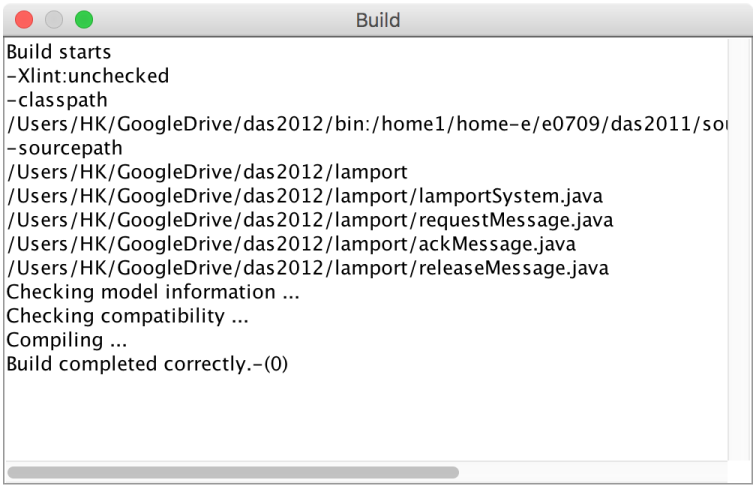


図 5.6: ビルド成功時の様子

5.6 シミュレーション

5.6.1 シミュレーションの開始

シミュレーションの開始は以下の手順で行う。

1. 図 5.7 の [New] ボタンをクリックする。または、メニューから [Simulation] → [Start] → [New simulation] を選択する。
2. 図 5.8 のダイアログで、シミュレーションを行いたいアルゴリズムの das ファイルを選択する。
3. 図 5.9 のダイアログで、プロセス数を設定する。
4. つづいて同じダイアログで、プロセスの故障率を設定する。入力した p の 1/1000 がプロセスの故障率として設定される。各プロセスが持つ変数 health がプロセスの故障率を下回ったとき、プロセスは故障状態になる。
5. 更に同じダイアログで通信遅延の上限を設定する。
6. 次にコータリーを使用するか否かと、使用するコータリーの選択を行う。
7. 次に通信路の FIFO 性の選択を行う。
8. 図 5.10 のダイアログでプロセスのパラメータを設定する。[Use the same configuration as that of the previous one] を選択すると、設定したパラメータ以外にベクトル時計も直前のプロセスと同じ初期値で生成される。[Use similar configuration for other processes] にチェックを入れ

た場合は、入力されたパラメータはすべてのプロセスに反映され、ベクトル時計はプロセス毎に正しく生成される。

以上の手順を完了すると、シミュレーションを開始できる状態となる。その状態の画面を図 5.11 に示す。

図 5.7: 新規シミュレーションボタン

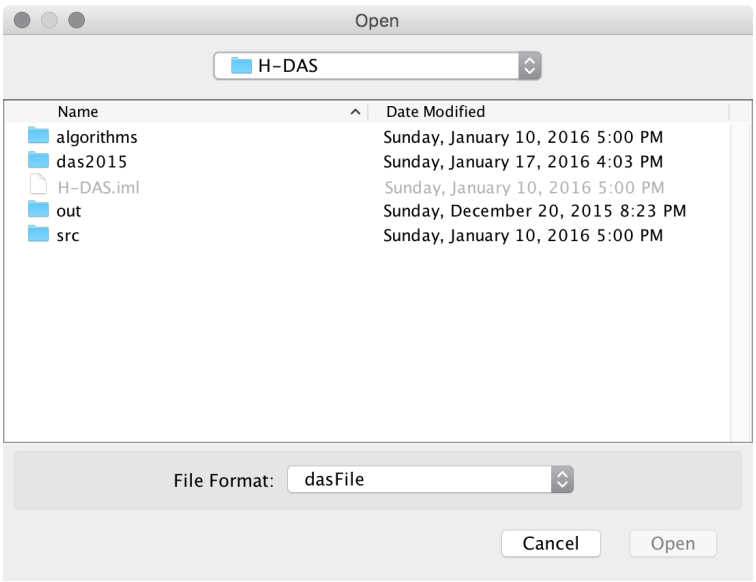


図 5.8: das ファイル選択ダイアログ

5.6.2 イベントの実行

送信または受信イベントの実行は以下の手順で行う。

1. イベントを生起させるためにプロセスの [Event] セルをクリックする。[Event] セルには、現在そのプロセスが実行可能なイベントの件数が示されている。
2. 図 5.12 のダイアログで、プロセスが実行可能なイベントを選択できるので、実行したいイベントを選択し、それが受信イベントの場合はメッセージの送信元のプロセスを選択する。
3. ダイアログ下部の [Execute] をクリックして実行する。[Cancel] をクリックすると何もせずダイアログを閉じる。

プロセスはイベントを実行すると、アルゴリズムにしたがって自身の論理ベクトル時計の自身の成分を 1 進める。それと同時に仮想大域時計も 1 進む。本シミュレータにおいてメッセージ配送は通常、システムの仮想大域時計が進むことによって自動で行われる。送受信イベントが生起出来ない場合や、新たに送受信イベントを行わずにメッセージ配送を行いたい場合には、論理時計は進めずに仮想大域時計のみを 1 ずつ進めることで送信バッファにあるメッセージの配送が行える。この機能は 5.13 に示す [Execute a null event] ボタンをクリックするか、メニューで [Auto] → [Execute a null event] を選択することで利用できる。

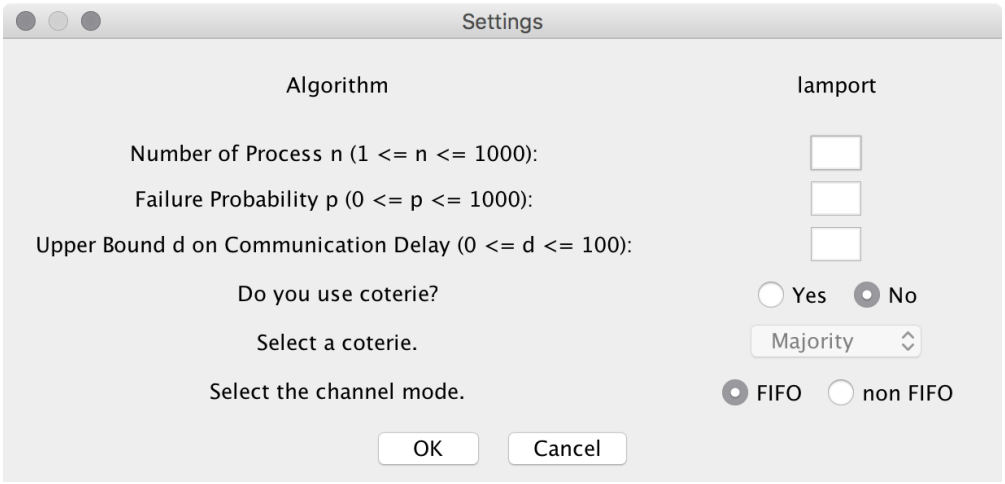


図 5.9: パラメータ入力ダイアログ

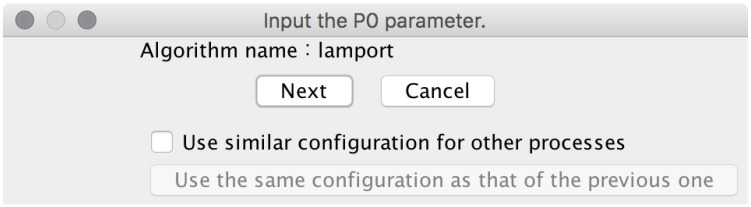


図 5.10: プロセスのパラメータ設定ダイアログ

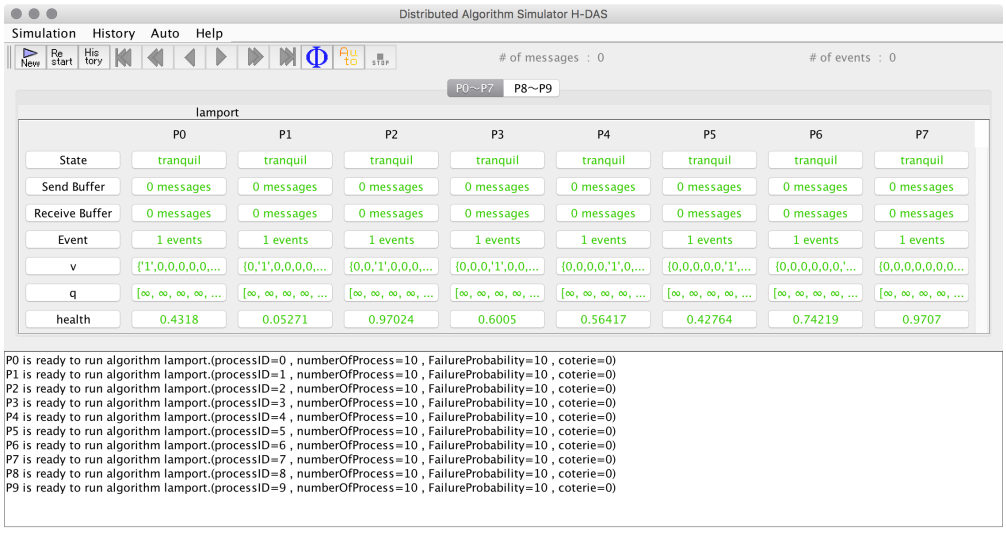


図 5.11: シミュレーション開始時のシミュレータの様子

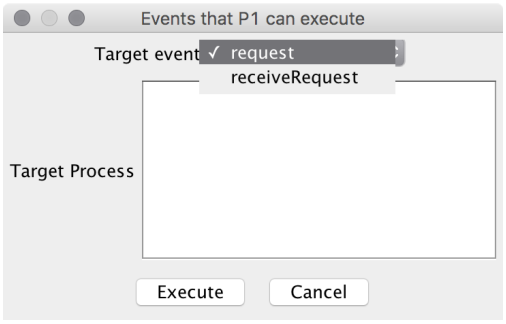


図 5.12: 実行するイベントを選択するダイアログ

図 5.13: Execute a null event ボタン

5.7 シミュレータの機能

5.7.1 実行されたイベントの取り消し機能

図 5.14 の [Undo a previous event] ボタン、[Undo several previous events] ボタン、[Undo all of previous events] ボタンにより、それぞれ 1 つ、任意回数、すべての直前イベントを取り消すことができる。任意回数戻る場合には図 5.15 のダイアログで取り消すイベント数を指定する。イベントを取り消すと、すべてのプロセスのパラメータも当該イベント実行前の状態に戻る。このとき、新規に実行されたシミュレーションでは、進む機能を使用してイベントの取り消しを取り消すことはできない。履歴からシミュレーションを行っているときは、進む機能を使用することができる。

図 5.14: イベントを取り消すボタン群

5.7.2 イベント生起の自動化機能

イベント生起の自動化機能は、以下の手順で行う。

1. 図 5.16 の [Auto] ボタンをクリックする。または、メニューから [Auto] → [Execute events automatically] を選択する。
2. 図 5.17 のダイアログでイベントの生起回数と実行速度を設定する。
3. ダイアログ下部の [Start] をクリックして自動化を開始する。

また、イベント生起の自動化を実行中に、意図的にイベント生起を止めたい場合には、図 5.18 に示す [Stop] ボタンをクリックする。または、メニューから [Auto] → [Stop the execution] を選択する。

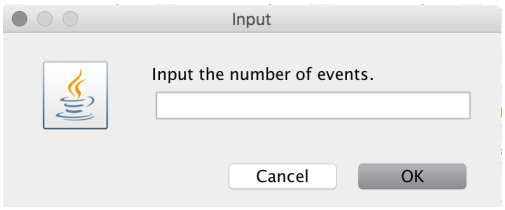


図 5.15: 取り消すイベント数を指定するダイアログ

図 5.16: Auto ボタン

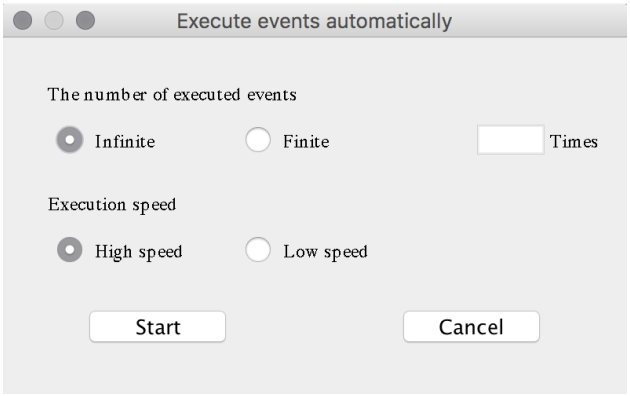


図 5.17: イベント生起の自動化設定ダイアログ

図 5.18: Stop ボタン

5.7.3　実行したシミュレーションの保存・再生

本シミュレータの履歴機能にはコマンド履歴とテキスト履歴がある。コマンド履歴はシミュレーションを再現するため、テキスト履歴はシミュレーションの様子をテキストから読み取るためのものである。

コマンド履歴の保存は以下の手順で行う。コマンド履歴ファイルは.cmd という拡張子で保存される。

1. メニューから [Simulation] → [Export] → [Event history] を選択する。
2. つづいて現れるダイアログで保存するディレクトリを指定し、[Save] をクリックする。

テキスト履歴の保存は以下の手順で行う。テキスト履歴ファイルは.txt という拡張子で保存される。

1. メニューから [Simulation] → [Export] → [Event history in text format] を選択する。
2. つづいて現れるダイアログで保存するディレクトリを指定し、[Save] をクリックする。

コマンド履歴からシミュレーションを再現は以下の手順を行う。

1. 図 5.19 の [History] ボタンをクリックする。または、メニューから [Simulation] → [Start] → [Simulation from history] を選択する。
2. 図 5.8 のダイアログで、再現するシミュレーションのアルゴリズムの das ファイルを選択する。
3. 図 5.20 のダイアログで、再現するシミュレーションの cmd ファイルを選択する。
4. イベントを生起させ、シミュレーションを進める。

図 5.19: History ボタン

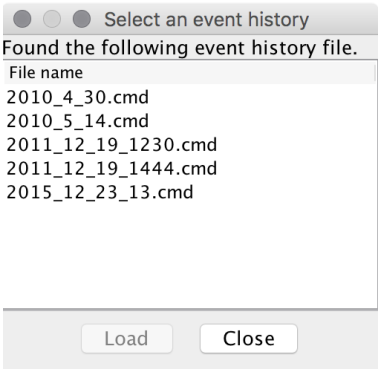


図 5.20: cmd ファイル選択ダイアログ

イベントを生起させるには、図 5.21 の [Execute a next event] ボタン、[Execute several next events] ボタン、[Execute all of next events] ボタンにより、それぞれ 1 つずつ、任意回数、すべてのイベントを実行することができる。

図 5.21: コマンド履歴のイベントを実行するボタン群

また、コマンド履歴からシミュレーションを再現しているときに [Event] セルからイベントを生起させると、新たな別のシミュレーションとみなされる。そのため、戻る機能を実行してもシミュレーションの再現に戻ることはできない。

コマンド履歴からシミュレーションを再現する際には、5.4 節で述べたシミュレータの起動方法に注意する。

5.7.4 実行中のシミュレーションと同じ設定での再シミュレーション

同一アルゴリズムの再シミュレーションは、以下の手順で行う。

1. 図 5.22 の [Restart] ボタンをクリックする。または、メニューから [Simulation] → [Start] → [Restart simulation] を選択する。
2. 図 5.23 のダイアログでシミュレーションを終了するか確認されるので、現在行っているシミュレーションをこのまま終了してよいならば [Yes] をクリックする。
3. シミュレーションの新規開始と同様の設定を行う。

図 5.22: Restart ボタン

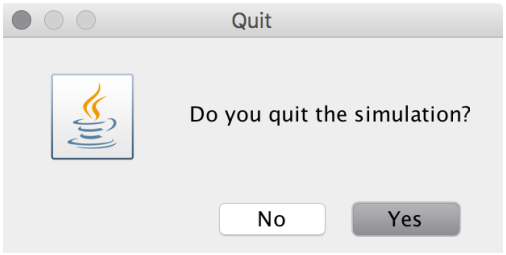


図 5.23: シミュレーションの終了確認ダイアログ

5.7.5 デッドロックの検出機能

デッドロックの検出機能は、シミュレーションを行っているとき自動で働いているため、特別な操作を行う必要はない。デッドロックを検出すると、図 5.24 に示すダイアログが表示される。

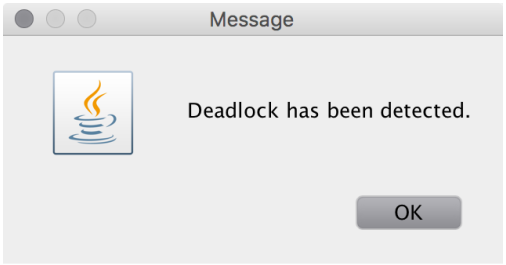


図 5.24: デッドロック検出ダイアログ

5.7.6 ヘルプマニュアル

本シミュレータにはシミュレータ及びデスクリプタについてのヘルプマニュアルが備わっている。それぞれメニューから [Help] → [Simulator]、[Help] → [Descriptor] を選択することで規定のブラウザ上で閲覧できる。

5.8 サンプルアルゴリズム

ここではサンプルアルゴリズムとして実装された Lamport のアルゴリズムと前川のアルゴリズムについて説明する。

5.8.1 Lamport のアルゴリズム

Lamport のアルゴリズムは、分散システムの相互排除問題を解決するためのアルゴリズムである。

アルゴリズムの動作をおおまかに述べる。資源を使用したいと望むプロセスは、自身も含めすべてのプロセスに時刻印を付与した要求メッセージを送信する。要求メッセージを受け取ったプロセスは、そのメッセージを自身が持つ待ち行列に加える。プロセスは自身の待ち行列の先頭に自身の要求メッセージがあり、かつ他のすべてのプロセスから承認メッセージを受け取ったならば資源を利用できる。

サンプルして実装したアルゴリズムでは、待ち行列の代わりに Direct-Dependency Clock $v[0..N-1]$ と、配列 $q[0..N-1]$ を使う。ここで N はプロセスの個数である。v の初期値はプロセス P_i では $v[i] = 1, v[j] = 0 (j \neq i)$ で、q の初期値はプロセスによらず $q = [\infty, \infty, \cdots, \infty]$ である。プロセスは $q[i]$ にプロセス P_i から受け取った時刻印を記録する。初期値の ∞ は P_i から要求メッセージを受け取っていないということを示している。

以下にアルゴリズムの手順を述べる。なお、以下の手順では逐一述べていないが、メッセージの送信と受信の際には 2.2.2 小節に述べた Direct-Dependency Clock のアルゴリズムも同時に実行される。また、$(a, b) < (c, d)$ は、$a < c \vee (a = c \wedge b < d)$ を省略して書いたものである。

1. 資源の使用を望むプロセス P_i は自分自身を含めたすべてのプロセスに自身の時刻印 $v[i]$ をつけた要求メッセージを送信し、自身の $q[i]$ に $v[i]$ を代入する。

2. 要求メッセージを受け取ったプロセス P_j は、自身の $q[j]$ に受け取った時刻印を代入し、承認メッセージを送信元のプロセスに返す。

3. P_i は $\forall j : j \neq i : (q[i], i) < (v[j], j) \wedge (q[i], i) < (q[j], j)$ を満たしたとき資源の使用を許可される。

4. 資源を開放するプロセス P_i は、自身の $q[i]$ を ∞ で初期化し、他のすべてのプロセスに資源の使用が終了したことを知らせる解放メッセージを送信する。

5. P_i から解放メッセージを受信したプロセスは、自身の $q[i]$ を ∞ で初期化する。

Lamport のアルゴリズムは、メッセージが FIFO で届くと仮定した場合、個々の要求はその発生順で承認されるという公平性を満たす。また、資源を使用したいと望むプロセスは、そのプロセス以外のすべてのプロセスから承認のメッセージを受け取らなければならないので、分散システムに 1 つでも故障したプロセスがあると、それが回復しないかぎり資源を利用することができない。よって、Lamport のアルゴリズムは耐故障性がないといえる。

5.8.2 前川のアルゴリズム

前川のアルゴリズムは、コータリーを用いる相互排除アルゴリズムである。

このアルゴリズムはコータリー C を用い、各プロセス P_i はコーラム $Q \in C$ 中のすべてのプロセスから資源の使用を許可されれば資源を使用する。すなわち、各プロセス P_i はコーラム Q 中のすべてのプロセスに要求メッセージ req を送信する。この req メッセージを受信したプロセスは、もし他のプロセスに許可を送信していない場合、即座に許可メッセージ locked を送信する。許可を送信している場合は送信先のプロセスにロックされているという。コーラム中のすべてのプロセスから許可メッセージ locked を受信すれば資源を使用できる。資源の使用終了後は資源の使用終了メッセージ release をコーラム中のすべてのプロセスに送信し、ロックを解除する。

しかし、上記のアルゴリズムはデッドロックに陥る可能性がある。そこで、前川のアルゴリズムでは、要求メッセージ req に Lamport の論理時計による時刻印を付加することで、要求に優先度を付ける。他のプロセスにロックされているプロセスが優先度の高い要求メッセージを受信した場合、先に送信した許可メッセージを取り消し、優先度の高い要求メッセージを送信したプロセスに許可を送信する。これによりデッドロックを解除する。

以下にアルゴリズムの詳細な手順を述べる。また図 5.25 はあるプロセス P_j の状態に着目して前川のアルゴリズムの動作を説明した図である。

1. 資源の使用を望むプロセス P_i はコーラム $Q \in C$ のすべてのプロセスに要求メッセージ req(TS_i,P_i) を送信する。

2. req を受けとったプロセス P_j は、もし他のプロセスに許可を送信していないならば、即座に許可メッセージ locked を P_i に送信する。

3. 他のプロセス P_k が P_j をロックしていたときには、P_j は req を待ち行列 $QUEUE_j$ に挿入する。$QUEUE_j$ にはいくつかの req が時刻印順に保持されている。

    a. もしも P_k の req または $QUEUE_j$ に保存されている req のどれかが、P_i の req よりも古い時刻印を持つならば P_j は P_i にメッセージ failed を返す。

    b. そうでなければ、P_j は P_k にメッセージ inquire を送信し、P_k が failed を受信しているかどうかを照会する。

4. P_k が P_j に release を送信しておらず、かつ P_k が failed を受信しているならばメッセージ relinquish を P_j に返す。

5. P_j が P_k の relinquish を受信したならば

1. req(TS_k, P_k) を $QUEUE_j$ に挿入する。その前から $QUEUE_j$ にあった req の中で最古の時刻印を持つ req(TS_h, P_h) を取り出し、そのメッセージを送信したプロセス P_h に対し locked を送る。
2. $P_h \neq P_i$ ならば P_i に failed を送信する。

6. P_i が Q に属するすべてのプロセスから locked を受信したならば、資源を使用する。
7. 資源の使用が終了したら、P_i は Q に属するすべてのプロセスに release を送信する。
8. P_j が P_i から release を受信したら、P_i によるロックを解除する。待ち行列 $QUEUE_j$ が空でなければ

1. 待ち行列 $QUEUE_j$ にある req の中で最古の時刻員を持つ req(TS_h,P_h) を取り出し、プロセス P_h に locked を送る。
2. もしも受信した release が、プロセス P_l の求めに応じて P_j が P_i に送信した inquire に対する応答の代わりであるときに、$P_h \neq P_l$ ならば P_l に failed を送信する。

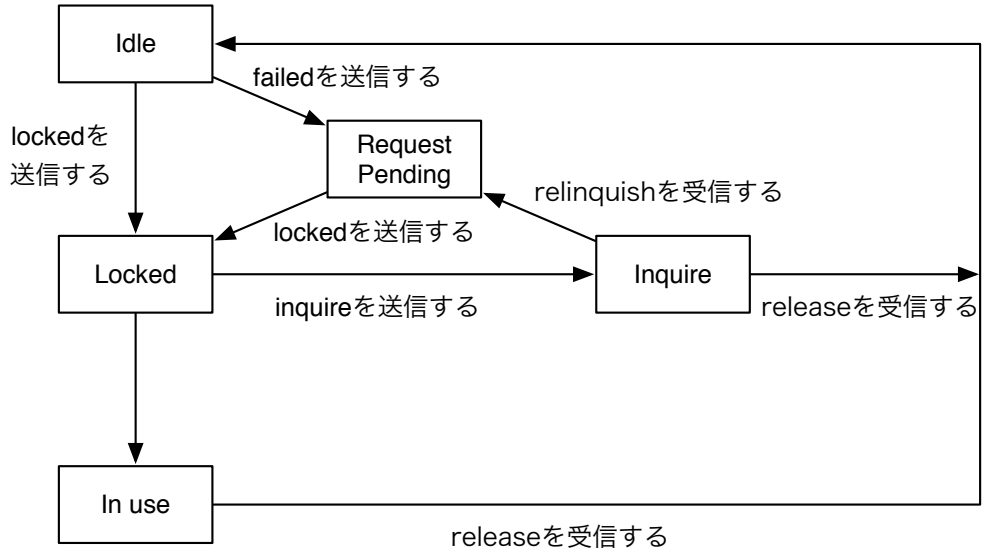


図 5.25: 前川のアルゴリズムにおけるプロセス P_j の P_i に関する状態遷移

前川のアルゴリズムは Lamport のアルゴリズムに比べて耐故障性が優れている。Lamport のアルゴリズムではシステム上のすべてのプロセスが正常に動作している必要があったが、前川のアルゴリズムでは少なくとも選択したコーラムに含まれるプロセスが動作していれば十分であるからである。

第6章　おわりに

本研究では、これまで本研究室で開発されてきた分散アルゴリズムシミュレータの機能追加と不具合の修正を行った。今回新たに追加した機能は、シミュレーション開始時の設定画面の統合とイベント発生回数の表示の 2 つである。また修正した不具合は、大量のイベントの連続発生でプログラムが強制終了する不具合、通信路に FIFO 性を付与したときに通信遅延の制約が守られない不具合、大量のプロセスからなるシステムをシミュレートする際に実行速度が極度に低下する不具合、特定の環境でプロセスの色付けがなされない不具合の 4 つである。

シミュレーション開始時の設定画面を統合したことで、設定項目の一覧性は向上し、ユーザのストレスを軽減することができた。

イベントの発生回数を表示するようにしたことで、イベント発生回数をシミュレーションの進度を見るための指標として用いることができるようになった。

これらの追加機能に加え、4 つの不具合に修正を施した。特に大量のイベントを発生させた際にプログラムが強制終了する問題は深刻であったが、これを解決することで H-DAS をシミュレータとして完成形とみなせるようになった。

しかし、H-DAS にはいまだ改善の余地が残されている。本シミュレータの完成度を高めるための方策を以下に提案する。

- 補助記憶の使用量の更なる削減
- 通信路故障の模倣
- ビット複雑度、時間複雑度の解析
- ユーザが決めた条件でシミュレーションの自動実行を停止する

補助記憶の使用量の削減は、シミュレーションの履歴を保存する.cmd ファイルの容量の削減に直結する。現状では.cmd ファイルは数 GB の大きさになることも多く、これを複数保存するユーザにとっては負担が大きい。4.2.1 節で示したように、補助記憶の使用量を半減させる方法もあるので、それを実現しユーザの負担を軽減することが望まれる。

H-DAS では通信路の故障は起きないものと仮定してシミュレーションを実行している。しかし、実際の通信ネットワークでは様々な理由で通信路が使用できなくなることがある。よって、通信路の故障を模倣することにより、より実際的なシミュレーションができるようになると思われる。このような機能を実装する際には、通信路の状態をグラフィカルに表示するインターフェイスを実装したり、プロセス間のメッセージの配送可能性について考慮する必要がある。

H-DAS は分散アルゴリズムの正しさを検証するソフトウェアではあるが、分散アルゴリズムの評価の尺度であるビット複雑度や時間複雑度を計算できれば、アルゴリズムの効率まで含めて計測できるようになり便利である。

分散アルゴリズムのシミュレーションでは数十から数百個のプロセスを使用することもあり、実際 H-DAS ではプロセスを 1000 個まで使用できる。大量のプロセスを使用した場合、プロセスの状態を目視で確認するのは現実的ではない。もし、プロセスの状態がある状態になったときに自動実行を停止するような機能があれば、大量のプロセスを使用した場合でもアルゴリズムの検証が容易になる。

参考文献

[1] M. Ben-Ari. *Distributed Algorithms in Java*. In Proceedings of the 2nd Conference on Integrating Technology into Computer Science Education, pp. 62–64, 1997.

[2] V. K. Garg. *Elements of Distributed Computing*. John Wiley & Sons, 2002.

[3] J. Dongarra et al. *PVM: Parallel Virtual Machine*. URL: `http://www.csm.ornl.gov/pvm/pvm_home.html`.

[4] Message-Passing Interface Forum. *MPI: A Message-Passing Interface Standard* (*Version 3.1*). URL: `http://www.mpi-forum.org/docs/docs.html`.

[5] Oracle. *Java SE Development Kit 8 Downloads*. URL: `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`.

[6] 亀田恒彦, 山下雅史. **分散アルゴリズム**. 近代科学社, 1994.

[7] 弘田暢幸, 梅本秀樹, 相原玲二, 山下雅史, 阿江忠. **分散環境で動作する分散アルゴリズムシミュレータ**. 情報処理学会論文誌 vol. 34, no. 11, pp. 2421–2430, 1993.

[8] 梶原将大, 長谷川貴紀. **分散アルゴリズムシミュレータの改良**. 明石工業高等専門学校卒業研究論文, 2012.

[9] 小島弘樹. **頂点故障を模倣する分散アルゴリズムシミュレータの構築**. 明石工業高等専門学校卒業研究論文, 2011.

[10] 中村文士. **汎用性のある分散アルゴリズムシミュレータの自動化**. 明石工業高等専門学校卒業研究論文, 2010.

[11] 吉田斉史, 和田渚. **汎用性のある分散アルゴリズムシミュレータの改良**. 明石工業高等専門学校卒業研究論文, 2009.

[12] 安田朋広. **汎用性のある分散アルゴリズムシミュレータの開発**. 明石工業高等専門学校卒業研究論文, 2007.

付録A 付録CD

付録 CD に H-DAS のクラスファイル、ソースファイル、及びサンプルアルゴリズムのソースファイルを収録した。CD 内のディレクトリ構造は図 6.1 に示すとおりである。

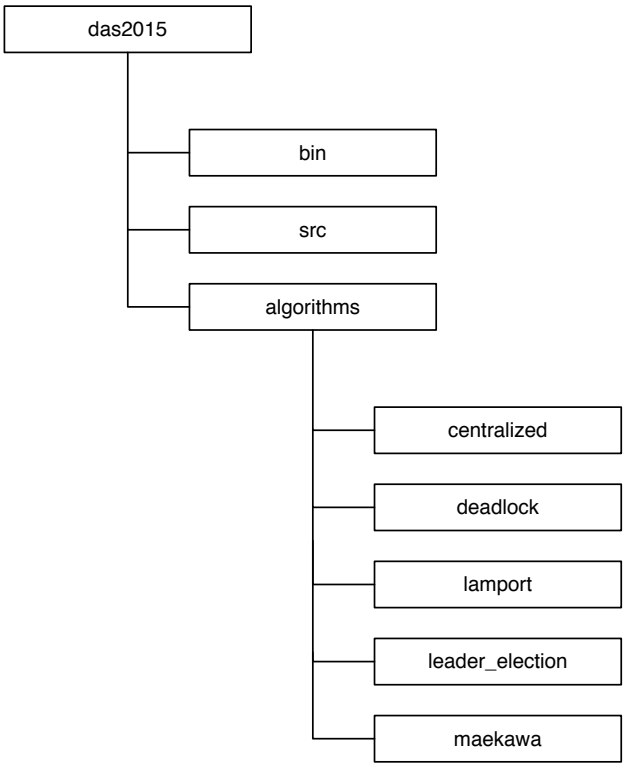


図 6.1: ディレクトリ構造

bin ディレクトリにはクラスファイルが、src ディレクトリにはソースファイルが格納されている。クラスファイルとソースファイルはパッケージ構造に対応したディレクトリ構造で格納されているが、図では省略している。algorithms ディレクトリには 5 つのサンプルアルゴリズムのソースファイルが格納されている。

各サンプルアルゴリズムがどのようなアルゴリズムであるかを簡単に説明する。centralized アルゴリズムは、1 つのプロセスを coordinator とし、coordinator が各プロセスに共有資源を割り当てていくアルゴリズムである。deadlock アルゴリズムはデッドロックを引き起こすアルゴリズムである。lamport アルゴリズムはメッセージの時刻印に基づいて相互排除を実現するアルゴリズムである。leader_election アルゴリズムはリーダー選出問題を解決する Chang-Roberts のアルゴリズムを実現しており、最もプロセス番号が大きいものをリーダーとして決定するアルゴリズムである。maekawa アルゴリズムは、コータリーを用いて相互排除を実現するアルゴリズムである。