JAIST Repository
https://dspace.jaist.ac.jp/

Title	リアルタイム・タスクスケジューリング・シミュレーション結果を可視化するGUI 分析ツールの開発 [課題研究報告書]
Author(s)	鈴木, 和佳子
Citation	
Issue Date	2017-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/14182
Rights	
Description	Supervisor:田中　清史, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

課題研究報告書

リアルタイム・タスクスケジューリング・シミュレーション結果を可視化するGUI分析ツールの開発

北陸先端科学技術大学院大学
情報科学研究科

鈴木　和佳子

平成29年3月

課題研究報告書

リアルタイム・タスクスケジューリング・シミュレーション結果を可視化するGUI分析ツールの開発

1310706　鈴木　和佳子

主指導教員　田中　清史
審査委員主査　田中　清史
審査委員　金子　峰雄
審査委員　井口　寧

北陸先端科学技術大学院大学
情報科学研究科

提出年月: 平成 29 年 2 月

Copyright © 2017 by Suzuki Wakako

概 要

今日、様々なリアルタイム・システムが稼動しており、リアルタイム・システムで実行されているタスクの中には、極めて重要性が高く応答遅延が人命にかかわるような致命的な影響を及ぼすリアルタイムタスクも存在する。そのため、全てのタスクでデッドラインミスなく指定された期限内に実施可能であるかを正確に計算し、スケジュール可能性を保証することはますます重要になってきている。しかし、タスクスケジューリングのシミュレーション結果はタスクの実行ログなどによって提供されることが多く、このようなテキストベースの情報から実際にスケジューリング可能性を確認することは困難である。特にタスクの実行回数が多く対象となる時間が長い場合には、スケジューリング可能性の確認は非常に多くの時間を要する作業となる。しかしながら、このようなタスク・スケジューリングのシミュレーション結果を実行ログなどの情報から図やグラフとして可視化するツールは現在のところあまり提供されていないのが現実である。

本研究ではリアルタイム・タスクスケジューリングのシミュレーション結果をグラフで可視化する GUI ツールを作成した。また、グラフの作成だけでなく、ズームイン/ズームアウト、デッドラインミスの検索機能やツールで作成したグラフを画像ファイルとしてエクスポートする機能も提供している。これによってタスクの実行回数が多く対象となる時間が長くても、スケジュール可能性を即座に確認することを可能にしているほか、論文などでシミュレーション結果をグラフとして図示することも容易にしている。また、本研究では Java SE に同梱して提供されている JavaFX を用いてツールを開発しているため、極めて簡単な準備で Windows や Linux, Mac OS などマルチプラットフォームでツールを実行することが可能になっている。

本稿では開発したリアルタイム・タスクスケジューリングのシミュレーション結果を可視化する GUI ツールの機能を説明し、開発に使用した JavaFX の機能や実際のコードについても説明する。

目 次

第 1 章　はじめに **1**
1.1　研究の背景 1
1.2　研究の目的 1
1.3　論文の構成 2

第 2 章　GUI ツールの概要 **3**

第 3 章　GUI ツールの開発 **6**
3.1　開発方法 6
3.2　入力ファイルの読み取り 6
3.3　グラフの描画 11
3.4　チャートを部分表示する機能の実装 17
3.5　スクロール・バーの表示とズームイン/ズームアウト機能の実装 24
3.6　チャートを画像ファイルとしてエクスポートする機能の実装 26
3.7　デッドラインミスを検索する機能の実装 29
3.8　開発時に発生した問題点と解決法 35
3.9　ツールのファイル構成およびソフトウェア規模 38

第 4 章　実際のシミュレーション結果における適用例 **41**

第 5 章　まとめ **47**

第1章　はじめに

1.1　研究の背景

今日、様々な組込みシステムにおいて日常的にリアルタイム・タスクが実行されている。組み込みシステムでのリアルタイム性への要求は増大し、その中には車のブレーキ制御装置・エアバッグ制御装置などデッドラインミスが致命的になりうるシステム（ハードリアルタイムシステム）も存在する。リアルタイムシステムにおいてそのシステムが実行するタスクが重要であればあるほど、タスクセットが真にスケジュール可能であることを保証することが重要になる。また、複数のリアルタイム・タスクが並行して実行されている時に、それらのタスクが真にスケジューリング可能であるかを検証することがますます重要になってきている。

システム開発においてシミュレーションあるいは実行ログの精査によってスケジュール可能性を検証することが有効であるが、テキストベースのシミュレーション結果／実行ログからスケジュール可能性や応答時間の変動を判断することは困難であり、時間のかかる作業となる。この作業をより円滑に実行するため、タスク・スケジューリングの結果を可視化するツールが有用であるが、現在のところ広く使用可能な形態で提供されていないのが実情である。

1.2　研究の目的

リアルタイム性を確保するためには、デッドラインミスを防ぎ応答時間を短縮するためのリアルタイム・タスクスケジューリング方式が重要である。リアルタイム・タスクスケジューリング方式に対する評価指標として代表的なものは「デッドラインミス発生の有無/頻度」と「応答時間」である [1]。高いリアルタイム性を達成するためには既存スケジューリング方式や新しいスケジューリング方式によるタスク実行の様子を観測し、デッドラインミスの発生状況や応答時間の変化を確認することが重要であるが、実行ログやシミュレーション出力は通常テキストベースであり、タスク実行状況を確認することが困難である。また、リアルタイムスケジューリングの研究において、新規に考案したスケジューリングアルゴリズムが有効に働くケースを的確に図示することが困難である場合がある。長い実行（シミュレーション）の中でアルゴリズムの効果が現れる個所を発見して論文のための例図とすることが有効であるが、このためには実行結果のスケジューリング図の取得が必要である。さらに、タスクの実行結果を可視化することによって、描画され

たタスクの実行結果の矛盾や問題点から、シミュレータのバグやシミュレータ上に実装されたスケジューリングアルゴリズムのバグを発見することも可能になる。
本研究では、実行ログやリアルタイム・タスクスケジューリング・シミュレーションの結果を可視化するツールを作成し、結果の分析を簡易化してリアルタイムシステム開発をサポートすることを目的とする。マルチプラットフォームでの利用を考慮し、Java 言語によるツール開発を行う。作成するツールはシミュレーション結果をグラフで図示する GUI ツールであり、以下のタスクスケジューリングの評価指標を可視化する。

1. 応答時間： タスクの起動要求から実行終了までの時間が応答時間となる。
2. デッドラインミスの有無/発生頻度： 実行がデッドラインまでに終了しない場合、デッドラインミスとなる。
3. スループット： 各タスク、および全タスクによるの CPU 使用率をスループットとする。

1.3 論文の構成

本課題研究報告では以下の構成をとる。2 章では本研究で開発した GUI ツールの概要について述べる。3 章では GUI ツールの開発方法や機能について、実際のコードや使用した JavaFX の機能も交えながら説明する。4 章では実際のタスク・シミュレーション結果を本研究の GUI ツールで分析した結果について記述する。最後に 5 章でまとめとして本研究を総括する。

第2章　GUIツールの概要

本研究ではCSV形式のファイルを入力とする。本ツールで使用するCSVファイルはタスク・スケジューリングのシミュレーション・ツールが出力することを前提とし、本研究で開発したツールでのCSV形式のフォーマットに合わせて出力すれば、シミュレーション結果を容易に可視化して分析することができる (図2.1)。各行には以下の項目をカンマ区切りで記述する。

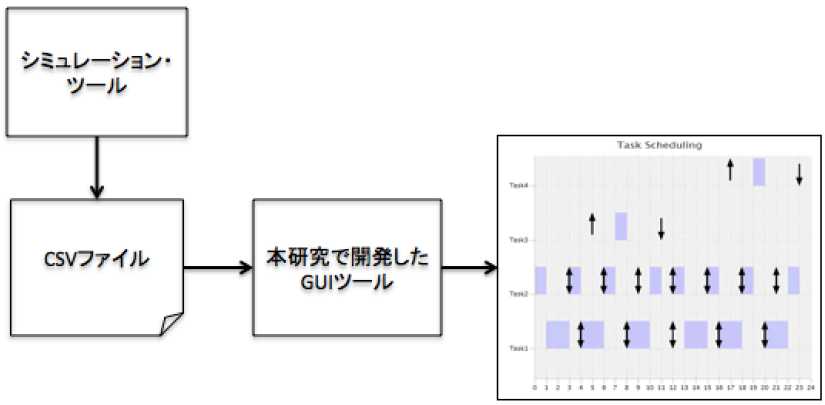

図 2.1: GUIツールの入力と出力

1. 各実行インスタンスのイベント時刻
2. タスク番号
3. ジョブ番号
4. イベント内容（起動要求、実行開始、実行終了、デッドライン、デッドラインミス）

出力するグラフは以下を表示する。

1. タスクの起動要求： 上矢印
2. タスクのデッドライン： 下矢印

3. タスク実行の CPU 使用時間： 四角

4. デッドラインミスの発生箇所： ×印

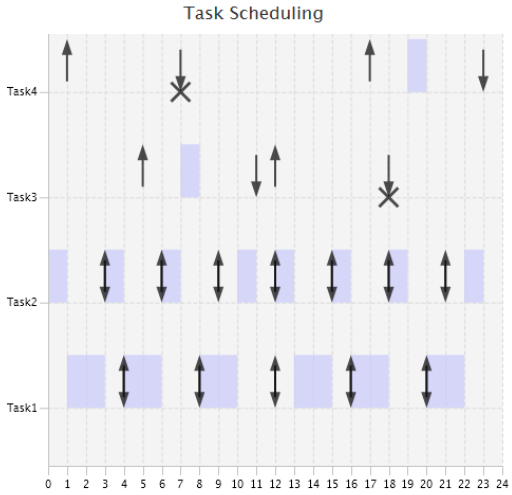

図 2.2: 作成した GUI ツールによる描画

サンプルの実行結果は図 2.2 のとおりである。

また、機能として以下を開発した。

1. 描画した結果を画像ファイルとして保存する。

2. スクロールして表示する。

3. デッドラインミスの発生箇所を検索し、発生箇所までジャンプして表示する。

4. ズームイン/ズームアウトして表示する。

また、JavaFX 8 はマルチプラットフォームをサポートしており、Mac OS や Linux もサポート対象である [2]。本 GUI ツールも図 2.3 や図 2.4 のように Mac OS 10 や Cent OS 7.2 で稼動する。

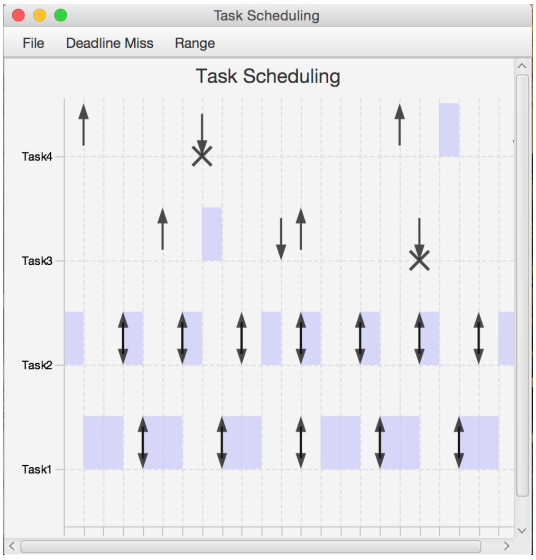

図 2.3: Mac OS 10 上で本 GUI ツールを稼動させた画面ショット

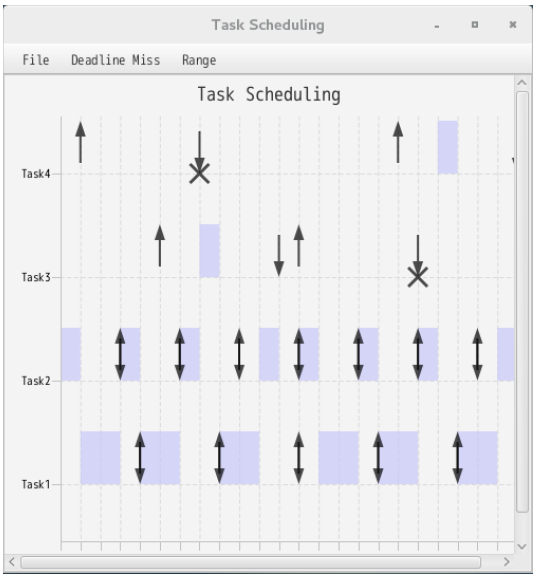

図 2.4: Cent OS 7.2 上で本 GUI ツールを稼動させた画面ショット

第3章　GUIツールの開発

3.1　開発方法

開発には JavaFX[3] を使用する。JavaFX はグラフィック、UI 機能を実現する Java のリッチ・クライアント・プラットフォームであり、シーン・グラフ [4] によって描画する個々の要素を階層ツリーで管理しており、これによって複雑な描画やアニメーションの開発を容易にしている。グラフ描画や図形描画、グラフィカル・ユーザー・インターフェースの機能を開発するための API を提供しているフレームワークや言語は Qt や HTML5, C# WPF など複数あり、ズームやスクロールの機能は実装することができるものもある。今回は通常の棒グラフではなく、ガントチャートのように途中で中断されるタスクの実行を描画する必要があり、横棒以外にもタスクの起動要求を示す上矢印、デッドラインを示す下矢印、デッドラインミスを示す×印を加えて描画する必要があった。調査を行ったところ、JavaFX ではチャートのデータに X 座標や Y 座標だけでなく、extraValue として汎用情報を付加することが可能 [5] であり、この API を利用することで単純な棒グラフではなく、ガントチャートの描画や上矢印や下矢印、×印を表示するといったカスタマイズを行うことができたため、JavaFX を採用した。JavaFX はまた稼動環境の構築が容易である点、マルチプラットフォームをサポート可能であるという点でも優位性がある。Java 7 Update 5 以前では JDK や JRE を導入するだけでなく、外部ライブラリである jfxrt.jar にクラスパスを通す必要があったが、Java 8 では JavaFX は JRE に同梱されており、JDK や JRE を導入するだけで JavaFX アプリケーションを稼動させる環境を構築することが可能になった。したがって、使用するバージョンは JDK1.8 に同梱されている JavaFX 8 とした。開発環境には NetBeans 8.1 を使用した。

3.2　入力ファイルの読み取り

入力ファイルは CSV 形式で項目は以下である。

1. 各実行インスタンスのイベント時刻
2. タスク番号
3. ジョブ番号

4. イベント内容（タスクの起動要求, 実行開始, 実行終了, デッドライン, デッドラインミス）

イベントの内容は以下で記述する。

1. タスクの起動要求： RL
2. 実行開始： ST
3. 実行終了： ED
4. デッドライン： DL
5. デッドラインミス： DM

Listing3.1 が CSV ファイルのサンプルである。

Listing 3.1: CSV ファイルのサンプル

```
0,2,1,ST
1,2,1,ED
1,1,1,ST
1,4,1,RL
3,1,1,ED
3,2,1,DL
3,2,2,RL
3,2,2,ST
...
7,4,1,DM
```

GUI ツールの起動直後の初期状態は以下である (図 3.1)。

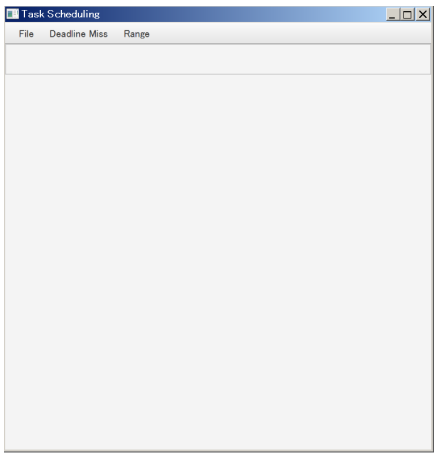

図 3.1: GUI ツール起動直後の状態

GUI ツールでファイルの読み取りを行う際には、GUI ツールのメニューから File → Open を選択し (図 3.2)、ファイル選択ダイアログから入力ファイルを 1 つ選択する (図 3.3)。

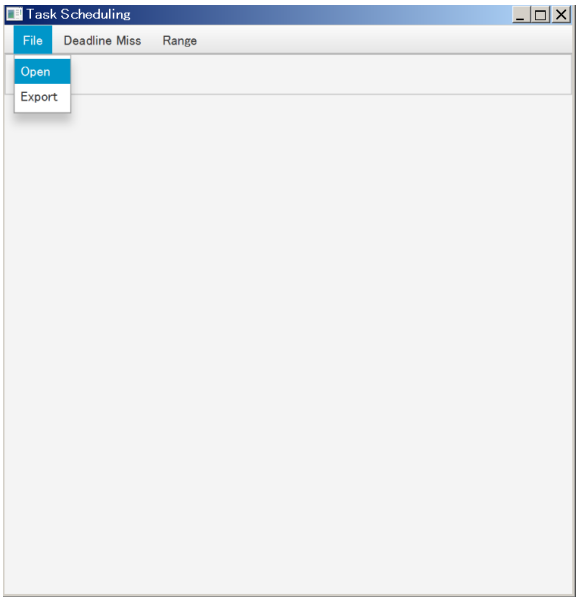

図 3.2: メニューから入力ファイルの選択

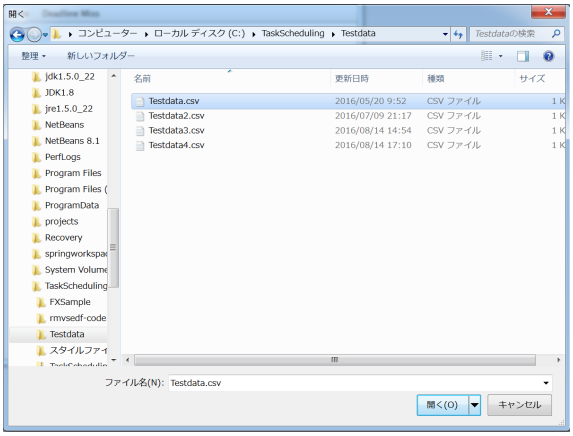

図 3.3: ファイル選択ダイアログの表示

ファイル選択ダイアログの表示にはJavaFXで標準で提供されているFileChooserクラスを使用した。ファイル選択ダイアログの表示はGUIの表示部分を担うChartCreaterクラスで行うが、ファイルに記述されているタスクスケジューリングのデータを読み取る部分はChartCreaterクラスのreadFile()メソッドで記述している。実装した部分をListing3.2に示す。

Listing 3.2: ファイル選択ダイアログの表示

```
public class ChartCreater extends Application {
...
   @Override
   public void start(Stage stage) {
       stage.setTitle("Task Scheduling");

       MenuBar menuBar = new MenuBar();
       Menu menuFile = new Menu("File");
       MenuItem open = new MenuItem("Open");
       open.setOnAction(new EventHandler<ActionEvent>() {
           public void handle(ActionEvent t) {
               open(stage);
           }
       });
...
    public void open(Stage stage) {

...
       try {
           final FileChooser fc = new FileChooser();
           importFile = fc.showOpenDialog(stage);
           taskList = readFile(importFile);
       } catch (Exception e) {
           Alert fileOpenError = new Alert(AlertType.ERROR);
           fileOpenError.setTitle("Error");
           fileOpenError.setContentText("Could not open a file.");
           fileOpenError.showAndWait().ifPresent(response -> {
               if (response == ButtonType.OK) {
                   return;
               }
           });
       }
```

ChartCreaterクラスのreadFile()メソッドは入力のCSVファイルを1行ずつ読み取り、カンマで区切られた各項目(イベント時刻, タスク番号, ジョブ番号, イベントの種類)をイベント情報を格納するクラスEventを作成して、それをArrayListに格納している。同時にデッドラインミスが発生した時刻のリスト、タスク番号のリストも作成している。

createTaskList()メソッドの実装はListing3.3である。

Listing 3.3: TaskList オブジェクトの作成

```
public ArrayList<Event> readFile(File file) throws IOException {
        FileReader fileReader = null;
        BufferedReader bufReader = null;
        String strLine;
        ArrayList<Event> taskList = new ArrayList<Event>();
        taskNoList = new ArrayList<Integer>();
        deadlineMissTimeList = null;
        initializeEventTypeList();
        try {
            fileReader = new FileReader(file);
            bufReader = new BufferedReader(fileReader);
            while ((strLine = bufReader.readLine()) != null) {
                String[] line = strLine.split(DELIMITER);
                String time = line[0];
                String taskNo = line[1];
                String jobNo = line[2];
                String eventType = line[3];
                int intTime = java.lang.Integer.parseInt(time);
                int intEventType = (enumEventList.get(eventType)).
                   intValue();
                Event event = new Event(intTime, java.lang.Integer.
                   parseInt(taskNo), java.lang.Integer.parseInt(jobNo),
                   (enumEventList.get(eventType)).intValue());
                taskList.add(event);
                if (deadlineMissTimeList == null) {
                    deadlineMissTimeList = new ArrayList<Integer>();
                }
                if (event.getEventType() == Event.DM_CODE && !
                   deadlineMissTimeList.contains(event.getTime())) {
                    deadlineMissTimeList.add(event.getTime());
                }
                if (!taskNoList.contains(new Integer(taskNo))) {
                    taskNoList.add(new Integer(taskNo));
                }
            }
            Collections.sort(deadlineMissTimeList);
            Collections.sort(taskNoList);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            fileReader.close();
            bufReader.close();
        }
        return taskList;
    }
```

3.3　グラフの描画

グラフの描画には stackoverflow に掲載されていたガントチャート表示のプログラム [6] をカスタマイズした。stackoverflow に掲載されていたガントチャート表示は図 3.4 のように系列と始点となる座標、終点となる座標を指定して長方形を描き、ガントチャートを作成するものである。

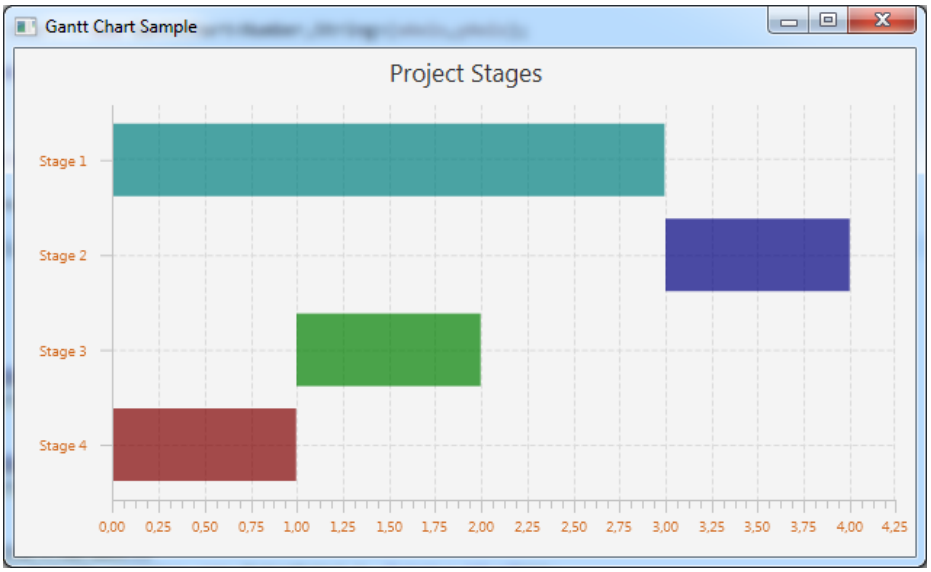

図 3.4: stackoverflow のガントチャート表示プログラムによるサンプル [6]

今回作成する GUI で出力するグラフはタスク実行の CPU 使用時間を示す長方形だけでなく、起動要求を示す上矢印、デッドラインを示す下矢印、デッドラインミスを示す×印も表示する必要がある。そのため、Listing3.4 のように ExtraData クラスにイベントの種類を示す type を追加した。Event クラスの変数 length は CPU 使用時間の長さ、styleClass は描画に使用するスタイルシートの設定、type はイベントの種類を格納する変数である。

Listing 3.4: ExtraData クラス

```
public class ExtraData {

    private long length;
    private String styleClass;
    private int type;

    public static final int EXEC = 0;
    public static final int START = 1;
    public static final int END = 2;
    public static final int DEADLINE_MISS = 3;

    public ExtraData(long lengthMs, String styleClass, int type) {
        this.length = lengthMs;
```

```
        this.styleClass = styleClass;
        this.type = type;
    }

    public long getLength() {
        return length;
    }

    public void setLength(long length) {
        this.length = length;
    }

    public String getStyleClass() {
        return styleClass;
    }

    public void setStyleClass(String styleClass) {
        this.styleClass = styleClass;
    }

    public int getType() {
        return type;
    }

    public void setType(int type){
        this.type = type;
    }

    }
```

グラフの描画はChartCreaterクラスのcreateChart()メソッドで行われる。ChartCreaterクラスのopen()メソッドで同じくChartCreaterクラスのreadFile()メソッドを呼び出してデータを読み込み、イベントのリストをArrayListを作成した(Listing3.2)後、リストからイベントのデータを一つずつ取り出し、イベント内容に合わせた描画を行うが、実際の描画を行う前に、TaskListから読み込んだデータを描画用のデータに変換する部分が必要になる。イベントのデータを描画データとして格納するのが先程述べたListing3.4のExtraDataクラスである。 タスクの実行のCPU使用時間を四角で描画するためのデータ格納はListing3.5である。開始時刻を起点として終了時刻–開始時刻の長さの四角を描くという形でデータを格納する。

Listing 3.5: タスクの実行のCPU使用時間の描画データへの変換

```
public class ChartCreater extends Application {
...
public void createChart(int minTime, int maxTime, Stage stage) {
...
        for (int i = startIndex; i <= endIndex; i++) {
...
            switch (eventType) {
```

```
                case (Event.ST_CODE):
                    Process p = new Process(event.getTaskNo(), event.
                        getJobNo());
                    inFlight.put(p.getKey(), new Integer(event.getTime()
                        ));
                    break;
                case (Event.ED_CODE):
                    Process proc = new Process(event.getTaskNo(), event.
                        getJobNo());
                    int startTime = minTime;
                    if (inFlight.get(proc.getKey()) != null) {
                        startTime = (inFlight.get(proc.getKey())).
                            intValue();
                        inFlight.remove(proc.getKey());
                    }
                    int endTime = event.getTime();
                    series.getData().add(new XYChart.Data<Number, String
                        >(startTime, machine, new ExtraData(endTime -
                        startTime, "status-gray", ExtraData.EXEC)));
                    break;
            }
            seriesArray.put(event.getTaskNo() - 1, series);
...
```

同様に、タスクの起動要求、デッドラインやデッドラインミスの描画データを作成している部分は以下となる (Listing3.6)。タスクの実行の CPU 使用時間の描画 (Listing3.5) と分けて描画しているのは、タスクの起動要求、デッドラインやデッドラインミスを後から描画することで、タスクの実行の CPU 使用時間と重なっても、タスクの実行の CPU 使用時間よりも前面に表示されるようにするためである。

Listing 3.6: タスクの起動要求、デッドラインやデッドラインミスの描画データへの変換

```
        for (int i = startIndex; i <= endIndex; i++) {
            Event event = taskList.get(i);
            XYChart.Series series = seriesArray.get(event.getTaskNo() -
                1);
            String machine = "Task" + (event.getTaskNo());
            if (series == null) {
                series = new XYChart.Series();
                machineList.put(event.getTaskNo() - 1, machine);
            }
            int eventType = event.getEventType();
            if (maxTime < event.getTime()) {
                maxTime = event.getTime();
            }
            switch (eventType) {
                case (Event.RL_CODE):
                    series.getData().add(new XYChart.Data<Number, String
                        >(event.getTime(), machine, new ExtraData(event.
                        getTime(), "status-black", ExtraData.START)));
                    break;
```

```
                case (Event.DL_CODE):
                    series.getData().add(new XYChart.Data<Number, String
                        >(event.getTime(), machine, new ExtraData(event.
                        getTime(), "status-black", ExtraData.DEADLINE)));
                    break;
                case (Event.DM_CODE):
                    series.getData().add(new XYChart.Data<Number, String
                        >(event.getTime(), machine, new ExtraData(event.
                        getTime(), "deadline-miss", ExtraData.
                        DEADLINE_MISS)));
                    break;
            }
            seriesArray.put(event.getTaskNo() - 1, series);
        }
...
         for (int i = 0; i < seriesArray.size(); i++) {
            chart.getData().add(seriesArray.get(i));
        }
```

グラフの軸や系列部分の描画データは Listing3.7 のように作成した。

Listing 3.7: グラフの軸や系列の描画データ作成

```
        NumberAxis xAxis = null;
        if (maxTime - minTime > 2000) {
            xAxis = new NumberAxis(minTime, maxTime, 10);
        } else {
            xAxis = new NumberAxis(minTime, maxTime, 1);
        }
        xAxis.setLabel("");
        xAxis.setTickLabelFill(Color.BLACK);
        xAxis.setMinorTickCount(1);
        xAxis.setAutoRanging(false);

        final CategoryAxis yAxis = new CategoryAxis();
        chart = new GanttChart<>(xAxis, yAxis);

        yAxis.setLabel("");
        yAxis.setTickLabelFill(Color.BLACK);

        chart.setTitle("Task Scheduling");
        chart.setLegendVisible(false);
        chart.setBlockHeight(50);
        ArrayList taskNoList = getTaskNoList();
        chart.setMinHeight(120 * taskNoList.size());

        chart.getStylesheets().add(getClass().getResource("ganttchart.
            css").toExternalForm());
        ArrayList<String> yAxisCategoryList = new ArrayList();
        for (int i = 0; i < taskNoList.size(); i++) {
            yAxisCategoryList.add("Task" + (i + 1));
            if (seriesArray.get(i) != null) {
                chart.getData().add(seriesArray.get(i));
```

```
            } else {
                chart.getData().add(new XYChart.Series());
            }
        }
        yAxis.setAutoRanging(false);
        yAxis.setCategories(FXCollections.<String>observableArrayList(
            yAxisCategoryList));
        yAxis.invalidateRange(yAxisCategoryList);
```

この描画データを元に実際に描画を行うのは GanttChart クラスである。GanttChart クラスも stackoverflow に掲載されていたもの [6] を今回開発した GUI ツール用にカスタマイズして使用している。GanttChart クラスは JavaFX で提供されている XYChart を継承しており、layoutPlotChildren() メソッドでガントチャートの四角形の描画を行っている。GanttChart クラスの layoutPlotChildren() メソッドは stackoverflow に掲載されていたものではガントチャートの四角形のみの描画だったが、カスタマイズを行い、ExtraData クラスに格納したイベントの種類や描画データを使用し、タスクの実行の CPU 使用時間・タスクの起動要求・デッドラインやデッドラインミスの描画を行うよう変更した。タスクの起動要求を示す上矢印、デッドラインを示す下矢印、デッドラインミスを示す×印は javafx.scene.shape.Polygon クラスを使用して描画している。Polygon クラスを使用して図形の描画を行う際には図形の各頂点の座標を指定する。Listing3.8 は上矢印の図形の描画部分である。

Listing 3.8: Polygon クラスを使用した上矢印の描画

```
Double[] arrowShape = new Double[]{0d, -50d, 5d, -35d, 1d, -35d, 1d, -10
    d, -1d, -10d, -1d, -35d, -5d, -35d};
Polygon polygon = new Polygon();
polygon.getPoints().addAll(arrowShape);
```

実際にイベントの種類に応じた図形の描画を行っているのが GanttChart クラスの layoutPlotChildren() メソッドで、実装は Listing3.9 である。

Listing 3.9: グラフの軸や系列の描画データ作成

```
public class GanttChart<X, Y> extends XYChart<X, Y> {
...
 @Override
   protected void layoutPlotChildren() {

       for (int seriesIndex = 0; seriesIndex < getData().size();
           seriesIndex++) {

            Series<X, Y> series = getData().get(seriesIndex);

            Iterator<Data<X, Y>> iter = getDisplayedDataIterator(series)
                ;
            while (iter.hasNext()) {
```

```
                Data<X, Y> item = iter.next();
                double x = getXAxis().getDisplayPosition(item.getXValue
                    ());
                double y = getYAxis().getDisplayPosition(item.getYValue
                    ());
                if (Double.isNaN(x) || Double.isNaN(y)) {
                    continue;
                }
                Node block = item.getNode();
                int type = ((ExtraData) (item.getExtraValue())).getType
                    ();
                Rectangle ellipse;

                if (block != null) {
                    if (block instanceof StackPane) {
                        StackPane region = (StackPane) item.getNode();
                        if (type == ExtraData.EXEC) {
                            if (region.getShape() == null) {
                                ellipse = new Rectangle(getLength(item.
                                    getExtraValue()), getBlockHeight());
                            } else if (region.getShape() instanceof
                                Rectangle) {
                                ellipse = (Rectangle) region.getShape();
                            } else {
                                return;
                            }
                            ellipse.setWidth(getLength(item.
                                getExtraValue()) * ((getXAxis()
                                instanceof NumberAxis) ? Math.abs(((
                                NumberAxis) getXAxis()).getScale()) : 1))
                                ;
                            ellipse.setHeight(getBlockHeight() * ((
                                getYAxis() instanceof NumberAxis) ? Math.
                                abs(((NumberAxis) getYAxis()).getScale())
                                : 1));

...
                            region.setShape(null);
                            region.setShape(ellipse);
                            region.setScaleShape(false);
                            region.setCenterShape(false);
                            region.setCacheShape(false);

                            block.setLayoutX(x);
                            block.setLayoutY(y - getBlockHeight());
                        } else if (type == ExtraData.START) {
                            Double[] arrowShape = new Double[]{0d, -50d,
                                5d, -35d, 1d, -35d, 1d, -10d, -1d, -10d,
                                -1d, -35d, -5d, -35d};
                            Polygon polygon = new Polygon();
                            polygon.getPoints().addAll(arrowShape);
                            region.setShape(polygon);
                            region.setScaleShape(false);
```

```
                        region.setCenterShape(false);
                        region.setCacheShape(false);

                        block.setLayoutX(x);
                        block.setLayoutY(y);
                    } else if (type == ExtraData.END) {
                        Double[] arrowShape = new Double[]{0.0, 0.0,
                            -5.0, -15.0, -1.0, -15.0, -1.0, -40.0,
                           1.0, -40.0, 1.0, -15.0, 5.0, -15.0};
                        Polygon polygon = new Polygon();
                        polygon.getPoints().addAll(arrowShape);
                        region.setShape(polygon);
                        region.setScaleShape(false);
                        region.setCenterShape(false);
                        region.setCacheShape(false);

                        block.setLayoutX(x);
                        block.setLayoutY(y);
                    } else if (type == ExtraData.DEADLINE_MISS) {
                        Double[] crossShape = new Double[]{0d, 2d, 8
                           d, 10d, 10d, 8d, 2d, 0d, 10d, -8d, 8d,
                           -10d, 0d, -2d, -8d, -10d, -10d, -8d, -2d,
                            0d, -10d, 8d, -8d, 10d};
                        Polygon polygon = new Polygon();
                        polygon.getPoints().addAll(crossShape);
                        region.setShape(polygon);
                        region.setScaleShape(false);
                        region.setCenterShape(false);
                        region.setCacheShape(false);

                        block.setLayoutX(x);
                        block.setLayoutY(y);
                    }
                }
            }
    ...
    }
```

3.4 チャートを部分表示する機能の実装

入力ファイルは10000ティック程度までの長いデータを想定しており、一度に全部のデータを表示すると見づらくなる場合が考えられるので、表示するティックの範囲を区切って表示を行う部分表示の機能が必要となる。部分表示を行うためには表示するティックの範囲に該当するイベントをリストから検索する必要がある。リスト内にはイベントがティック順に格納されているので、表示範囲の最初のティック以降の先頭のイベントと表示範囲の最後のティック以前の最後のイベントを検索することで、部分表示に必要なイベントのリスト内でのインデックスの範囲を取得することができる。表示範囲の最初のティック

以降の先頭のイベントの検索は ChartCreater クラスの binarySearchMinIndex() メソッド (Listing3.10 参照) で、表示範囲の最後のティック以前の最後のイベントは ChartCreater クラスの binarySearchMaxIndex() メソッド (Listing3.11 参照) で実装した。

Listing 3.10: 表示範囲の最初のティック以降の先頭のイベントの検索

```
    public int binarySearchMinIndex(int minTime) {
        int left = 0;
        int right = taskList.size() - 1;
        Event event = null;
        int currentIndex = 0;
        while (true) {
            currentIndex = (int) (left + right) / 2;
            event = taskList.get(currentIndex);
            if (event.getTime() > minTime) {
                if (taskList.get(currentIndex - 1).getTime() < minTime)
                    {
                    break;
                }
                right = currentIndex;
            } else if (event.getTime() < minTime) {
                left = currentIndex;
            } else {
                break;
            }
        }
        while (true) {
            if (taskList.get(currentIndex).getTime() >= minTime &&
                taskList.get(currentIndex - 1).getTime() < minTime) {
                break;
            }
            currentIndex--;
        }
        return currentIndex;
    }
```

Listing 3.11: 表示範囲の最後のティック以前の最後のイベントの検索

```
public int binarySearchMaxIndex(int maxTime) {
    int left = 0;
    int right = taskList.size() - 1;
    Event event = null;
    int currentIndex = 0;
    while (true) {
        currentIndex = (int) (left + right) / 2;
        event = taskList.get(currentIndex);
        if (event.getTime() > maxTime) {
            right = currentIndex;
        } else if (event.getTime() < maxTime) {
            if (taskList.get(currentIndex + 1).getTime() > maxTime)
               {
                break;
            }
            left = currentIndex;
        } else {
            break;
        }
    }
    while (true) {
        if (taskList.get(currentIndex).getTime() <= maxTime &&
           taskList.get(currentIndex + 1).getTime() > maxTime) {
            break;
        }
        currentIndex++;
    }
    return currentIndex;
}
```

入力ファイル内の最後のイベントの時刻が 50 を超える場合、初期表示は先頭から 50 ティックとし、メニューから Range → Next50 を選択することで、次の 50 ティックを表示できるようにした (図 3.5 参照)。前に戻る場合はメニューから Range → Prev50 を選択することで、前の 50 ティックを表示できるようにした (図 3.6 参照)。現在の先頭のティックが 50 未満にもかかわらず、前 50 ティックの表示が呼び出された場合は、先頭から 50 ティックを表示する。最後のティックから数えて 50 未満のティックであるにもかかわず、次の 50 ティックの表示が呼び出された場合は、最後の 50 ティックを表示する。全体が 50 ティックに満たない場合は全体を表示する。また、50 ティック区切りによる表示だけでなく、Range → Show All を選択して部分表示から全体表示へ切り替える機能 (図 3.7 参照) や、Range → Specify Range を選択し、任意の範囲を表示できる機能も実装した (図 3.8 参照)。範囲指定によるグラフの表示結果は図 3.9 である。

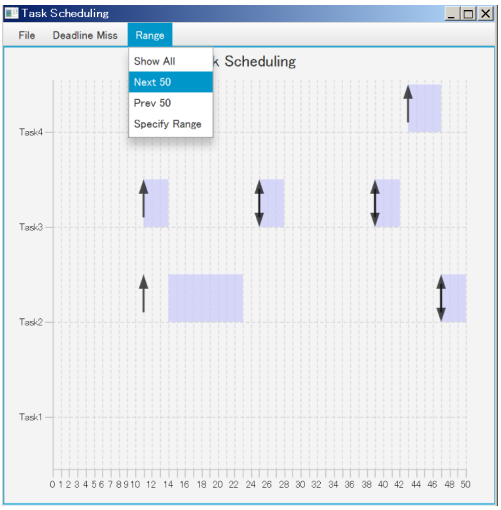

図 3.5: 次 50 件表示のメニュー選択

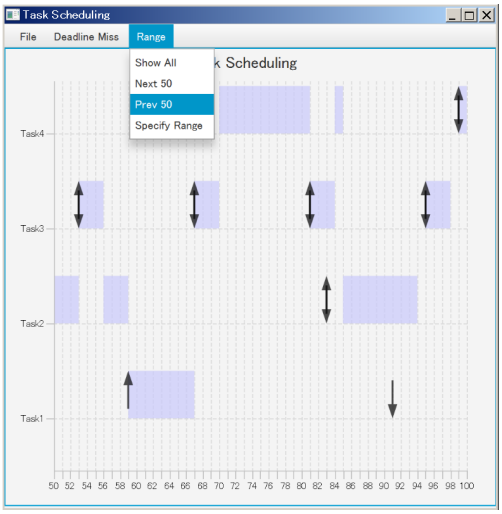

図 3.6: 前 50 件表示のメニュー選択

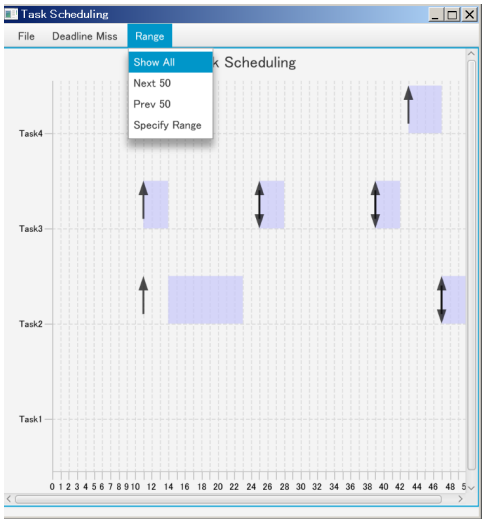

図 3.7: 全体表示のメニュー選択

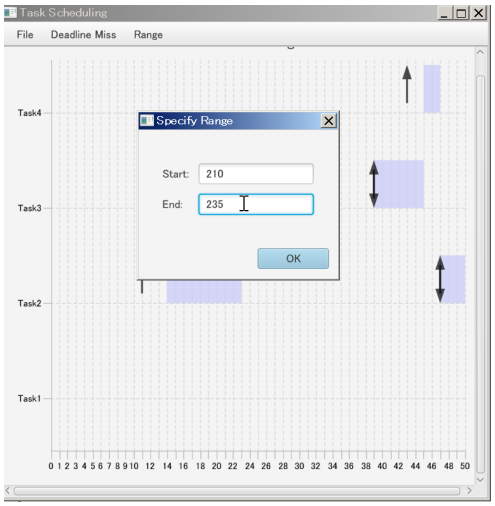

図 3.8: 表示するティックの範囲指定

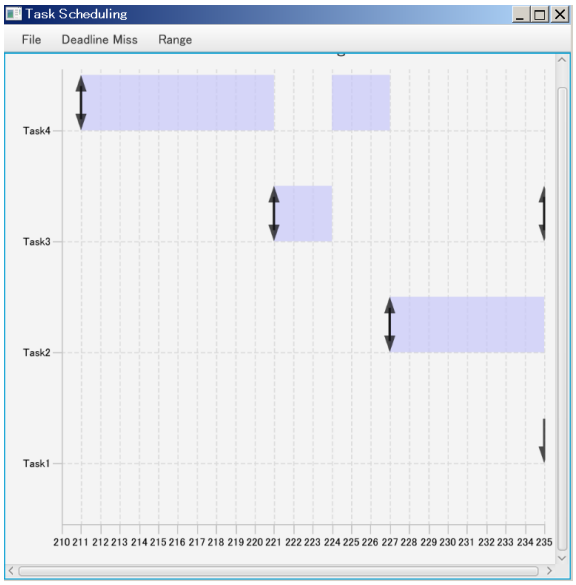

図 3.9: ティックの範囲を指定したグラフの表示結果

チャートを表示する際に、タスク実行の CPU 使用時間を表示するためには、タスク実行の開始時刻と終了時刻をマッチさせる必要がある。この役割を担うために本 GUI ツールでは HashMap を使用し、タスク実行の開始のイベントがあった場合に HashMap にレコードを追加し、タスク実行の終了のイベントがあった場合に HashMap にタスク番号とジョブ番号をキーにタスク実行開始のレコードを HashMap に取得しに行き、そのレコードを読み取った後 HashMap から削除する動作になっている。しかし、チャートの部分表示を行う場合にはタスク実行の開始時刻が表示範囲に含まれるがタスク実行の終了時刻が表示範囲に含まれない場合、逆にタスク実行の終了時刻は表示範囲に含まれるが、タスク実行の開始時刻が表示範囲に含まれない場合が生じてくる。タスク実行終了のイベントに対応するタスク実行開始のイベントが表示範囲に存在しない場合、タスク実行の開始時刻を表示範囲の最初のティックとしてチャートを表示する。タスク実行の終了のイベントを読み取った際に、タスク実行開始のレコードを HashMap から消しているので、タスク実行の開始のイベントに対応するタスク実行の終了時刻がない場合は、HashMap にタスク実行開始のレコードが残ることになる。この場合、削除されずに残ったタスク実行開始のレコードに対してはタスク実行の終了時刻として表示範囲の最後のティックを設定しチャートに表示する。図 3.12 の inFlight という HashMap がタスク番号とジョブ番号を結合した文字列をキーにタスク実行開始のレコードを格納する HashMap である。

Listing 3.12: タスク実行開始時刻とタスク実行終了時刻のマッチング

```
    public void createChart(int minTime, int maxTime, Stage stage) {

...
        inFlight = new HashMap();
...
        for (int i = startIndex; i <= endIndex; i++) {
...
            switch (eventType) {
                case (Event.ST_CODE):
                    Process p = new Process(event.getTaskNo(), event.
                       getJobNo());
                    inFlight.put(p.getKey(), new Integer(event.getTime()
                       ));
                    break;
                case (Event.ED_CODE):
                    Process proc = new Process(event.getTaskNo(), event.
                       getJobNo());
                    int startTime = minTime;
                    if (inFlight.get(proc.getKey()) != null) {
                        startTime = (inFlight.get(proc.getKey())).
                           intValue();
                        inFlight.remove(proc.getKey());
                    }
                    int endTime = event.getTime();
                    series.getData().add(new XYChart.Data<Number, String
                       >(startTime, machine, new ExtraData(endTime -
                       startTime, "status-gray", ExtraData.EXEC)));
                    break;
            }
            seriesArray.put(event.getTaskNo() - 1, series);
        }
        if (inFlight.size() > 0) {
            Set<String> procKeySet = inFlight.keySet();
            Iterator<String> procKeyItr = procKeySet.iterator();
            while (procKeyItr.hasNext()) {
                String procKey = procKeyItr.next();
                String[] process = procKey.split("-");
                int taskNo = Integer.parseInt(process[0]);
                int jobNo = Integer.parseInt(process[1]);
                int startTime = (inFlight.get(procKey)).intValue();
                int endTime = maxTime;
                String machine = "Task" + taskNo;
                seriesArray.get(taskNo - 1).getData().add(new XYChart.
                   Data<Number, String>(startTime, machine, new
                   ExtraData(endTime - startTime, "status-gray",
                   ExtraData.EXEC)));
            }
        }
...
```

3.5 スクロール・バーの表示とズームイン/ズームアウト機能の実装

今回の GUI ツールでは初期表示のサイズを 500 ピクセル× 500 ピクセルとし、作成したグラフがそのサイズを超える場合は自動的にスクロール・バーを表示する動作とした。また、図 3.10 のようにグラフの任意の箇所を右クリックし、Zoom In または Zoom Out を選択することにより、任意の場所でズームイン/ズームアウトができるように実装を行った。

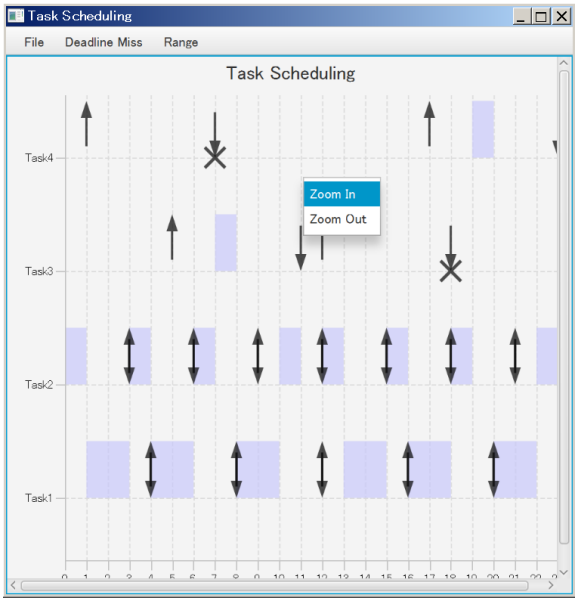

図 3.10: ズームイン/ズームアウトの選択

スクロール・バーの表示には ScrollPane クラスを使用し、ズームの実装は Pixel Duke[7] に掲載されていた方法を参照し、スクロール対象のコンテンツを Group に入れて Group を ScrollPane のコンテンツとして設定することで、ScrollPane 内の全てのコンポーネントをズームイン/ズームアウトできるようにした。スクロール・バーの表示は以下 Listing3.13 のように行っている。

Listing 3.13: スクロール・バーの表示

```
public class ChartCreater extends Application {
...
ScrollPane sp = new ScrollPane();
Group group = new Group();
...
    @Override
   public void start(Stage stage) {
 ...
        sp.setContent(group);
       Scene scene = new Scene(new VBox(), 500, 500);
       ((VBox) scene.getRoot()).getChildren().addAll(menuBar, sp);
       stage.setScene(scene);

       stage.show();
   }
 ...
    public void open(Stage stage) {
 ...
       group.getChildren().add(chart);
       sp.requestLayout();
 ...
   }
```

ズームイン/ズームアウトについてはチャートの表示を行う ChartCreater クラスの createChart() メソッドで Listing3.14 のように実装を行い、チャートの横幅を 1.25 倍にすることでズームイン、チャートの横幅を 0.8 倍にすることでズームアウトを実現している。

Listing 3.14: ズームイン/ズームアウトの実装

```
public class ChartCreater extends Application {
  ...
    public void createChart(int minTime, int maxTime, Stage stage) {
  ...
       final ContextMenu contextMenu = new ContextMenu();
       MenuItem zoomIn = new MenuItem("Zoom In");
       MenuItem zoomOut = new MenuItem("Zoom Out");

       chart.addEventHandler(MouseEvent.MOUSE_CLICKED, new EventHandler
          <MouseEvent>() {
           @Override
           public void handle(MouseEvent e) {
               if (e.getButton() == MouseButton.SECONDARY) {
                   contextMenu.show(chart, e.getScreenX(), e.getScreenY
                      ());
               }
           }
       });
       contextMenu.getItems().add(zoomIn);
       contextMenu.getItems().add(zoomOut);
       group.getChildren().add(chart);
       sp.setContent(group);
```

```
        sp.requestLayout();

        zoomIn.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent t) {
                chart.setPrefWidth(chart.getWidth() * 1.25);
                sp.requestLayout();
            }
        });

        zoomOut.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent t) {
                chart.setPrefWidth(chart.getWidth() * 0.8);
                sp.requestLayout();
            }
        });
    }
      ...
```

実際にズームインを行った場合の画面ショットが以下である。図 3.11 がズーム前、図 3.12 がズーム後の画面ショットである。

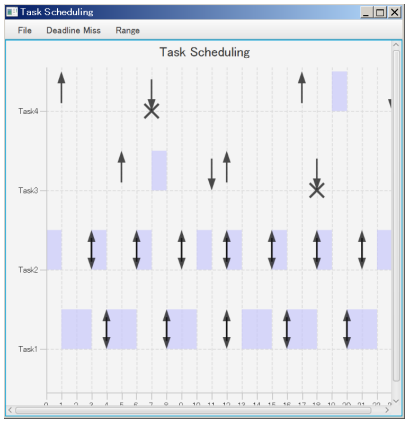

図 3.11: ズームイン前の画面ショット

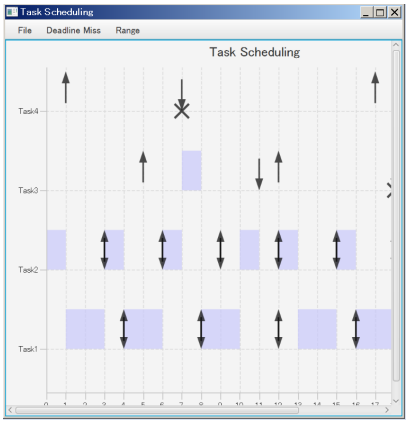

図 3.12: ズームイン後の画面ショット

3.6 チャートを画像ファイルとしてエクスポートする機能の実装

画像ファイルのエクスポートには JavaFX で提供されている Node クラスの snapshot() メソッド [8] を使用した。GanttChart クラスは XYChart クラスを継承しており、XYChart クラスが Node クラスを継承しているので、GanttChart クラスからこのメソッドを呼び出すことができる。なお、実装の際には Oracle 社のチュートリアル [9] や stackoverflow[10] のサンプルを参照した。今回の GUI ツールでは入力ファイルを開き、チャートを表示し

た状態でメニューバーから File → Export を選択することで表示しているチャートを png 形式の画像ファイルとしてエクスポートできる。本 GUI ツールでは非常に長いティック数の画像を数十回程度ズームしてからダウンロードするような大きな描画データをエクスポートする場合も想定し、Platform.runLater() メソッドを使用して画像エクスポートを行うスレッドを JavaFX アプリケーション本体のスレッドと別にしている。図 3.13 がその画面ショットである。

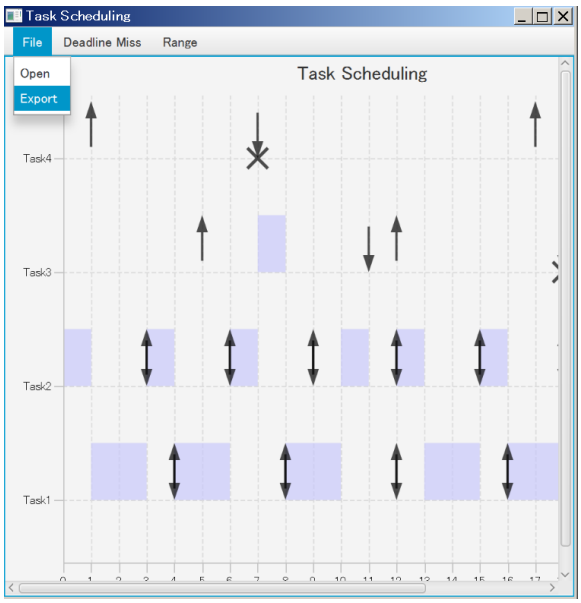

図 3.13: チャートを画像ファイルとしてエクスポート

保存先のファイルの選択には、入力ファイルの読み取りの際と同様 FileChooser クラスを使用し、ダイアログを表示して保存先のディレクトリを GUI で選択させ、ファイル名を入力できるようにした。また、拡張子の付け忘れや入力ミスなく必ず png ファイルとして保存するよう拡張子フィルタを追加した。図 3.14 が画像エクスポートの際に表示される拡張子フィルタをつけたファイル選択ダイアログである。エクスポートのメニューの表示は ChartCreater クラスの start() メソッドで、エクスポート処理の実装は ChartCreater クラスの export() メソッドで行っている。実装は Listing3.15 のとおりである。

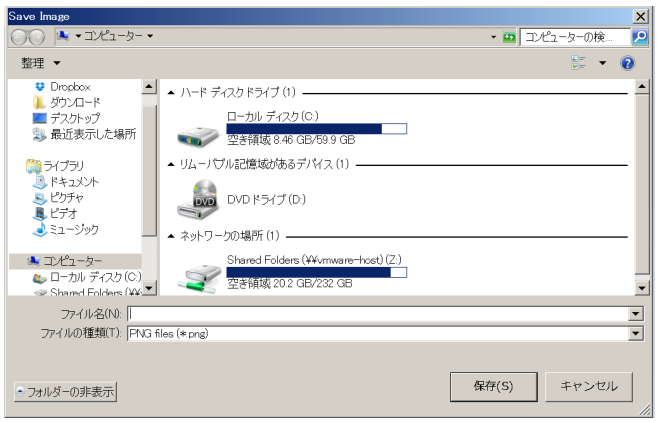

図 3.14: 画像エクスポート時に表示される拡張子フィルタをつけたファイル選択ダイアログ

Listing 3.15: 画像ファイルとしてのエクスポートの実装

```
public class ChartCreater extends Application {
...
    @Override
   public void start(Stage stage) {
...
        MenuBar menuBar = new MenuBar();
        Menu menuFile = new Menu("File");
...
        MenuItem download = new MenuItem("Export");
        download.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent t) {
                Thread thread = new Thread(() -> {
                    Platform.runLater(() -> export(stage));
                });
                thread.start();
            }
        });
        menuFile.getItems().addAll(open, download);
        menuBar.getMenus().addAll(menuFile);
...
}
...
   public void export(Stage stage) {
        FileChooser fc = new FileChooser();
        FileChooser.ExtensionFilter extFilter
            = new FileChooser.ExtensionFilter("PNG files (*.png)", "*.
               png");
        fc.getExtensionFilters().add(extFilter);
        fc.setTitle("Save Image");
        File file = fc.showSaveDialog(stage);
        WritableImage image = chart.snapshot(new SnapshotParameters(),
           null);
```

```
        try {
            ImageIO.write(SwingFXUtils.fromFXImage(image, null), "png",
                file);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

エクスポートを行う際には、現在表示されているチャートがそのままエクスポートされる。表示範囲やズーム回数なども反映してエクスポートを行うことができる。実際に表示範囲として 6700〜6730 ティックを指定し、ズームを 2 回行った状態でエクスポートした画像が図 3.15 である。

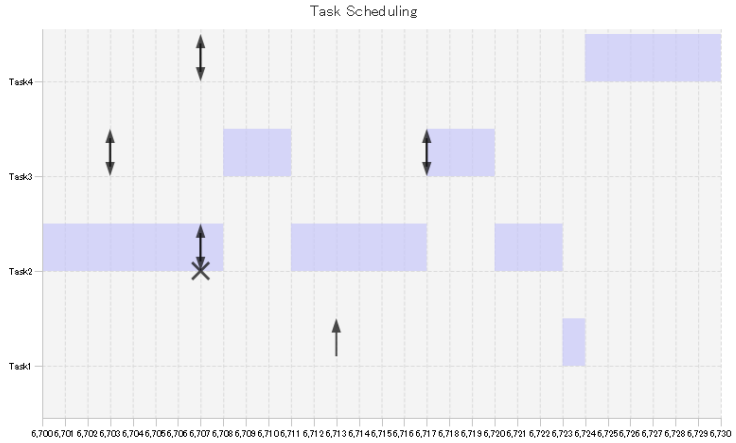

図 3.15: 表示範囲を指定しズームしてエクスポートした画像

3.7 デッドラインミスを検索する機能の実装

デッドラインミスの検索は前方と後方の 2 方向での検索が可能になるよう実装した。チャートを開いた状態でメニューから Deadline Miss → Previous で前方への検索、Deadline Miss → Next で後方への検索を行うことができる (図 3.16)。検索を行った結果、デッドラインミスは図 3.17 のように表示される。

デッドラインミスの検索を行う前に、入力ファイルを選択してチャートが表示されている必要がある。入力ファイルが選択されておらず、チャートがまだ表示されていない場合には図 3.18 のエラーメッセージが表示される。図 3.16 のメニュー表示及び入力ファイル未選択のエラー表示の実装部分は以下 Listing3.16 である。

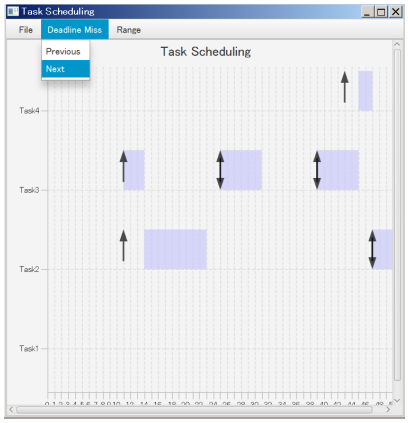

図 3.16: デッドラインミス検索のメニュー

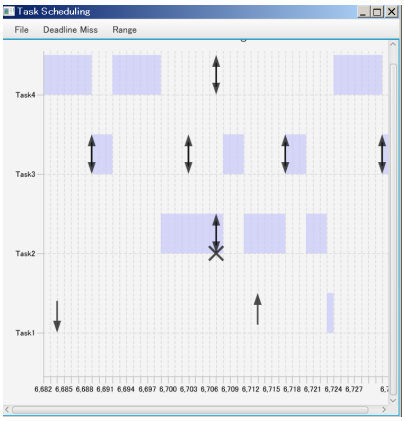

図 3.17: 検索されたデッドラインミスの表示

図 3.18: 入力ファイル未選択のエラー表示

Listing 3.16: デッドラインミス検索のメニュー表示の実装

```
...
public class ChartCreater extends Application {
...
    @Override
    public void start(Stage stage) {
    ...
        Menu menuDeadlineMiss = new Menu("Deadline Miss");
        MenuItem next = new MenuItem("Next");
        next.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent t) {
                try {
                    next(stage);
                } catch (Exception e) {
                    Alert chartNull = new Alert(AlertType.ERROR);
                    chartNull.setTitle("Error");
                    chartNull.setContentText("Open a file first.");
                    chartNull.showAndWait().ifPresent(response -> {
                        if (response == ButtonType.OK) {
                            return;
                        }
                    });
                }
            }
```

```
        });
        MenuItem previous = new MenuItem("Previous");
        previous.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent t) {
                try {
                    previous(stage);
                } catch (Exception e) {
                    Alert chartNull = new Alert(AlertType.ERROR);
                    chartNull.setTitle("Error");
                    chartNull.setContentText("Open a file first.");
                    chartNull.showAndWait().ifPresent(response -> {
                        if (response == ButtonType.OK) {
                            return;
                        }
                    });
                }
            }
        });
        ...
```

3.2 節で記述した入力ファイルの読み取りの部分でデッドラインミスが発生した時刻の重複を排除した昇順のリストをあらかじめ作成しておく。これによって現在グラフに描画されてい部分にデッドラインミスが発生していなくても、デッドラインミスの発生箇所にジャンプすることができる。メニューから Deadline Miss → Next を選択した際に呼び出される ChartCreater クラスの next() メソッドの実装は以下 Listing3.17 である。前述した入力ファイルが未選択の場合のエラー処理もここに記述されている。デッドラインミスが発生した時刻のリストを参照し、次のデッドラインの時刻を中央に表示するグラフを表示する。また、最後のデッドラインミスであるにもかかわらず、Next が呼び出された場合には図 3.19 のようなエラーメッセージを表示する。

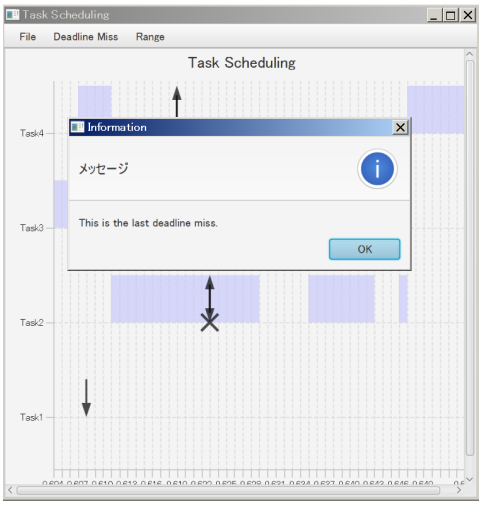

図 3.19: 最後のデッドラインミスであることを示すエラー表示

Listing 3.17: デッドラインミスの後方への検索

```
...
public class ChartCreater extends Application {
...
    public void next(Stage stage) throws Exception {

        if (chart == null) {
            throw new Exception();
        }
        int currentIndex = 0;
        ArrayList<Integer> deadlineMissTimeList =
           getDeadlineMissTimeList();
        if (deadlineMissTimeList == null || deadlineMissTimeList.isEmpty
           () == true || deadlineMissTimeList.size() == 0) {
            Alert noDeadlineMiss = new Alert(AlertType.INFORMATION);
            noDeadlineMiss.setTitle("Information");
            noDeadlineMiss.setContentText("There is no deadline miss.");
            noDeadlineMiss.showAndWait().ifPresent(response -> {
                if (response == ButtonType.OK) {
                    return;
                }
            });
        } else {
            if (deadlinemissIndex < deadlineMissTimeList.size() - 1) {
                deadlinemissIndex++;
                int lastTime = taskList.get(taskList.size() - 1).getTime
                   ();
                int deadlineMissTime = deadlineMissTimeList.get(
                   deadlinemissIndex);
                if (lastTime <= 50 || deadlineMissTime <= 25) {
                    createChart(0, lastTime, stage);
                    currentMinTime = 0;
                    currentMaxTime = lastTime;
                } else if (lastTime >= 50 && deadlineMissTime >=
                   lastTime - 25) {
                    createChart(lastTime - 50, lastTime, stage);
                    currentMinTime = lastTime - 50;
                    currentMaxTime = lastTime;
                } else {
                    createChart(deadlineMissTime - 25, deadlineMissTime
                       + 25, stage);
                    currentMinTime = deadlineMissTime - 25;
                    currentMaxTime = deadlineMissTime + 25;
                }
            } else {
                int lastTime = taskList.get(taskList.size() - 1).getTime
                   ();
                int deadlineMissTime = deadlineMissTimeList.get(
                   deadlinemissIndex);
                if (lastTime <= 50 || deadlineMissTime <= 25) {
                    createChart(0, lastTime, stage);
                    currentMinTime = 0;
```

```
                    currentMaxTime = lastTime;
                } else if (lastTime >= 50 && deadlineMissTime >=
                    lastTime - 25) {
                    createChart(lastTime - 50, lastTime, stage);
                    currentMinTime = lastTime - 50;
                    currentMaxTime = lastTime;
                } else {
                    createChart(deadlineMissTime - 25, deadlineMissTime
                        + 25, stage);
                    currentMinTime = deadlineMissTime - 25;
                    currentMaxTime = deadlineMissTime + 25;
                }
                Alert noNextDeadlineMiss = new Alert(AlertType.
                    INFORMATION);
                noNextDeadlineMiss.setTitle("Information");
                noNextDeadlineMiss.setContentText("This is the last
                    deadline miss.");
                noNextDeadlineMiss.showAndWait().ifPresent(response -> {
                    if (response == ButtonType.OK) {
                        return;
                    }
                });
            }
        }
    }
...
```

メニューから Deadline Miss → Previous を選択した際に呼び出される ChartCreater クラスの prev() メソッドの実装は以下 Listing3.18 である。また、最初のデッドラインミスであるにもかかわらず、Previous が呼び出された場合には図 3.20 のようなエラーメッセージを表示する。

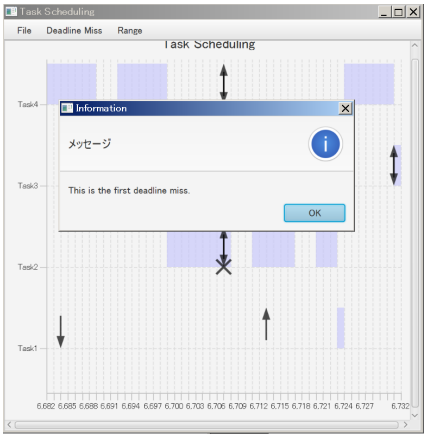

図 3.20: 最初のデッドラインミスであることを示すエラー表示

Listing 3.18: デッドラインミスの前方への検索

```
...
public class ChartCreater extends Application {
...
    public void prev(Stage stage) throws Exception {
        if (chart == null) {
            throw new Exception();
        }

        ArrayList<Integer> deadlineMissTimeList =
           getDeadlineMissTimeList();
        if (deadlineMissTimeList == null || deadlineMissTimeList.isEmpty
           () == true || deadlineMissTimeList.size() == 0) {
            Alert noDeadlineMiss = new Alert(AlertType.INFORMATION);
            noDeadlineMiss.setTitle("Information");
            noDeadlineMiss.setContentText("There is no deadline miss.");
            noDeadlineMiss.showAndWait().ifPresent(response -> {
                if (response == ButtonType.OK) {
                    return;
                }
            });
        } else {
            if (deadlinemissIndex > 0) {
                deadlinemissIndex--;
                int lastTime = taskList.get(taskList.size() - 1).getTime
                   ();
                int deadlineMissTime = deadlineMissTimeList.get(
                   deadlinemissIndex);
                if (lastTime <= 50 || deadlineMissTime <= 25) {
                    createChart(0, lastTime, stage);
                    currentMinTime = 0;
                    currentMaxTime = lastTime;
                } else if (lastTime >= 50 && deadlineMissTime >=
                   lastTime - 25) {
                    createChart(lastTime - 50, lastTime, stage);
                    currentMinTime = lastTime - 50;
                    currentMaxTime = lastTime;
                } else {
                    createChart(deadlineMissTime - 25, deadlineMissTime
                       + 25, stage);
                    currentMinTime = deadlineMissTime - 25;
                    currentMaxTime = deadlineMissTime + 25;
                }
            } else {
                deadlinemissIndex = 0;
                int lastTime = taskList.get(taskList.size() - 1).getTime
                   ();
                int deadlineMissTime = deadlineMissTimeList.get(0);
                if (lastTime <= 50 || deadlineMissTime <= 25) {
                    createChart(0, lastTime, stage);
                    currentMinTime = 0;
                    currentMaxTime = lastTime;
```

```
                } else if (lastTime >= 50 && deadlineMissTime >=
                    lastTime - 25) {
                    createChart(lastTime - 50, lastTime, stage);
                    currentMinTime = lastTime - 50;
                    currentMaxTime = lastTime;
                } else {
                    createChart(deadlineMissTime - 25, deadlineMissTime
                        + 25, stage);
                    currentMinTime = deadlineMissTime - 25;
                    currentMaxTime = deadlineMissTime + 25;
                }
                Alert noNextDeadlineMiss = new Alert(AlertType.
                    INFORMATION);
                noNextDeadlineMiss.setTitle("Information");
                noNextDeadlineMiss.setContentText("This is the first
                    deadline miss.");
                noNextDeadlineMiss.showAndWait().ifPresent(response -> {
                    if (response == ButtonType.OK) {
                        return;
                    }
                });
            }
        }
    }
```

なお、前方への検索でも後方への検索でもデッドラインミスが存在しない場合は図 3.21 のように表示される。

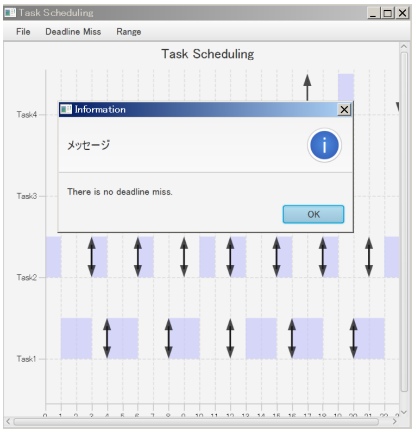

図 3.21: デッドラインミスが存在しないことを示すメッセージ表示

3.8 開発時に発生した問題点と解決法

本節では開発時に発生した問題点とその解決法について記述する。本節で開発した GUI ツールを実行し、検証を行った際に以下の 2 つの問題点が発生した。

1. 仮想マシン上で非常に長いティック範囲のグラフを表示し、ズームを繰り返すと表示が崩れる

2. 非常に長いティック範囲のグラフを画像としてエクスポートすると、例外が発生してエクスポートできない

Mac OS 10 上で VMWare を使用して Windows 7 の仮想環境を構築して検証を行っていたところ、非常に長いティック範囲のグラフを表示し、ズームを繰り返すと表示が崩れる問題が発生した。その際には Listing3.19 のように GUI ツールを実行していた。

Listing 3.19: 問題発生時の GUI ツールの実行方法

```
java taskscheduling.ChartCreater
```

図 3.22 が 9997 ティック分のイベントをグラフ表示した直後の状態、図 3.23 がその後 8 回ズームを行った状態である。

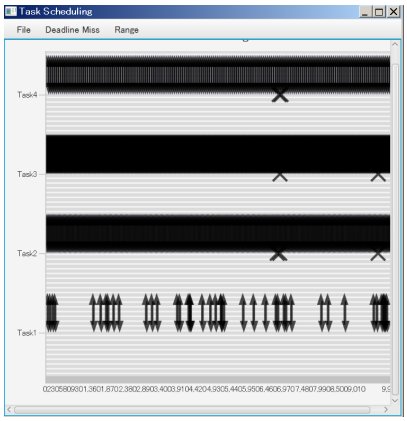

図 3.22: 仮想マシン上で 9997 ティックを表示

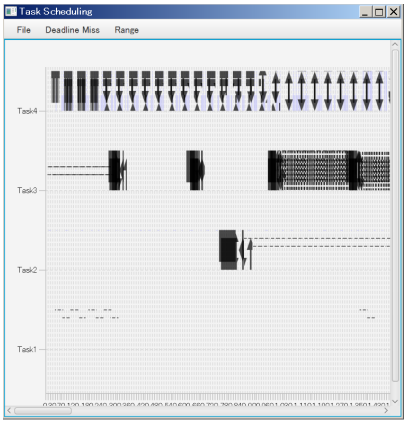

図 3.23: 仮想マシン上で 9997 ティックを表示した後 8 回ズームを行ったグラフ

JavaFX ではパフォーマンス上の観点から可能な場合はハードウェア側で画像のレンダリングを行う動作となっている [11] が、JavaFX は正式に仮想環境をサポートしておらず、仮想環境ではハードウェアでの画像レンダリングに問題が発生してしまうため、このように表示が崩れる状態になっていた。このため、Listing3.20 のように JVM 引数 prism.order を使用してソフトウェア・レンダリングを強制的に実行させるような設定を行い、問題を解決することができた。

Listing 3.20: 問題解決後の GUI ツールの実行方法

```
java -Dprism.order=sw taskscheduling.ChartCreater
```

また、非常に長いティック範囲のグラフを画像としてエクスポートすると、例外が発生してエクスポートできない問題も発生していたが、同様に Listing3.20 で画像をソフトウェア・レンダリングで生成させることにより解決することができた。この問題は非仮想環境でも発生しており、9997 ティックのグラフを全範囲で表示し、その後 15 回ズーム (図 3.24 参照) してから画像エクスポートを行うと、Listing3.21 の例外が発生した。ズーム回数を減らした場合には発生しなかった。

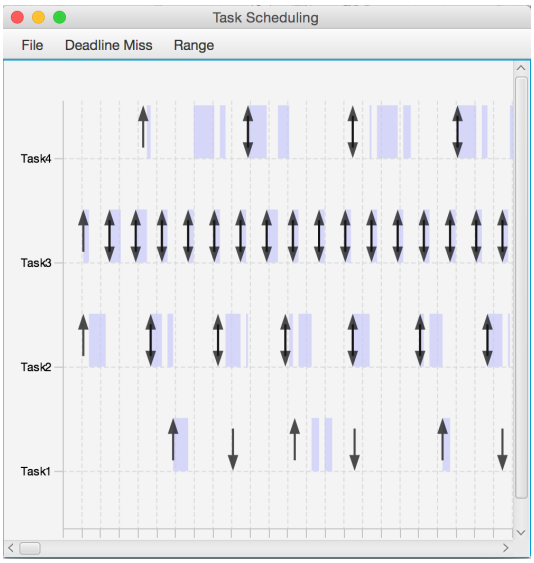

図 3.24: 非仮想環境で 9997 ティック表示後 15 回ズームした画面

Listing 3.21: 非仮想環境で 9997 ティック表示後 15 回ズームしたした後画像エクスポートした際に発生した例外

```
java.lang.RuntimeException: Requested texture dimensions (17764x480)
   require dimensions (0x480) that exceed maximum texture size (16384)
        at com.sun.prism.es2.ES2RTTexture.create(ES2RTTexture.java:220)
        at com.sun.prism.es2.ES2ResourceFactory.createRTTexture(
           ES2ResourceFactory.java:157)
        at com.sun.prism.es2.ES2ResourceFactory.createRTTexture(
           ES2ResourceFactory.java:153)
        at com.sun.javafx.tk.quantum.QuantumToolkit$QuantumImage.getRT(
           QuantumToolkit.java:1284)
        at com.sun.javafx.tk.quantum.QuantumToolkit$5.run(QuantumToolkit
           .java:1421)
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors
           .java:511)
        at java.util.concurrent.FutureTask.runAndReset(FutureTask.java
           :308)
        at com.sun.javafx.tk.RenderJob.run(RenderJob.java:58)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(
           ThreadPoolExecutor.java:1142)
```

```
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(
           ThreadPoolExecutor.java:617)
        at com.sun.javafx.tk.quantum.QuantumRenderer$PipelineRunnable.
           run(QuantumRenderer.java:125)
        at java.lang.Thread.run(Thread.java:745)
Exception in thread "JavaFX Application Thread" java.lang.
   IllegalArgumentException: Unrecognized image loader: null
        at javafx.scene.image.WritableImage.loadTkImage(WritableImage.
           java:240)
        at javafx.scene.image.WritableImage.access$000(WritableImage.
           java:46)
        at javafx.scene.image.WritableImage$1.loadTkImage(WritableImage.
           java:51)
        at javafx.scene.Scene.doSnapshot(Scene.java:1236)
        at javafx.scene.Node.doSnapshot(Node.java:1864)
        at javafx.scene.Node.snapshot(Node.java:1942)
        at taskscheduling.ChartCreater.export(ChartCreater.java:401)
        at taskscheduling.ChartCreater$2.lambda$null$0(ChartCreater.java
           :107)
        at com.sun.javafx.application.PlatformImpl.lambda$null$173(
           PlatformImpl.java:295)
        at java.security.AccessController.doPrivileged(Native Method)
        at com.sun.javafx.application.PlatformImpl.lambda$runLater$174(
           PlatformImpl.java:294)
        at com.sun.glass.ui.InvokeLaterDispatcher$Future.run(
           InvokeLaterDispatcher.java:95)
```

この例外は画像生成に必要なテキスチャ・サイズがハードウェアで提供しているテクスチャ・サイズを超えたために発生しているので、こちらもハードウェアによる画像レンダリングを行わず、ソフトウェア・レンダリングを使用することで解決できた。

3.9 ツールのファイル構成およびソフトウェア規模

本 GUI 分析ツールを構成するファイルとその行数、メソッドの一覧を示したものが表 3.1 である。

表 3.1: 本ツールのファイル構成

ファイル名	クラス名	行数	メソッド名
ChartCreater.java	ChartCreater	790	public void start(Stage) public int binarySearchMaxIndex(int) public int binarySearchMinIndex(int) private void checkRangeAndCreate-Chart(Range, Stage)

			public void createChart(int, int, Stage) public void export(Stage) public ArrayList getDeadlineMissTimeList() private void initializeEventTypeList() public static void main(String[]) public void next(Stage) public void prev(Stage) public void open(Stage) public ArrayList<Event>readFile(File) throws IOException public void start(Stage)
GanttChart.java	GanttChart	240	public GanttChart(Axis, Axis) public GanttChart(Axis, Axis, ObservableList) public ExtraData getExtraValue() public void setExtraValue(ExtraData) private static String getStyleClass(Object) private static double getLength(Object) protected void layoutPlotChildren() public double getBlockHeight() public void setBlockHeight(double) protected void dataItemAdded(Series, int, Data) protected void dataItemRemoved(Data, Series) protected void dataItemChanged(Data) protected void seriesAdded(Series, int) protected void seriesRemoved(Series) private Node createContainer(Series, int, Data, int) protected void updateAxisRange()
ExtraData.java	ExtraData	56	ExtraData(long, String, int) public long getLength()

			public void setLength(long) public String getStyleClass() public void setStyleClass(String) public int getType() public void setType(int)
Event.java	Event	62	Event(int, int, int, int) public int getEventType() public int getJobNo() public int getTaskNo() public int getTime() public void setEventType(int) public void setJobNo(int) public void setTaskNo(int) public void setTime(int)
Process.java	Process	49	Process(int, int) public boolean equals(Process) public long getJobNo() public String getKey() public int getTaskNo() public void setJobNo(int) public void setTaskNo(int)
Range.java	Range	39	Range(String, String) throws NullPointerException, NumberFormatException public boolean equals(Process) public int getStart() public int setEnd()

第4章　実際のシミュレーション結果における適用例

本研究で作成したGUIツールを実際のタスク・シミュレーション結果に適用してみた。タスク・シミュレーションではタスクセットは周期タスクと非周期タスクの両方を含み、タスク数は4〜7で、CPU使用率60%、65%, 70%, 75%, 80%, 85%, 90%, 95%で、それぞれデッドラインミスがある場合とない場合の両方のシミュレーション結果に適用した。まず、CPU使用率60%でタスク数6, デッドラインミスなしの場合について記述する。このシミュレーションで実行したタスクは以下である。

- タスク1 : 非周期タスク
- タスク2 : 周期タスク (周期64 実行時間2)
- タスク3 : 周期タスク (周期43, 実行時間5)
- タスク4 : 周期タスク (周期84, 実行時間18)
- タスク5 : 周期タスク (周期99, 実行時間10)
- タスク6 : 周期タスク (周期45, 実行時間6)

シミュレーション結果であるCSVを本ツールで開いた直後の状態が図4.1で、画面の縦横のサイズを変更して全てのタスクが表示されるようにした画面が図4.2である。

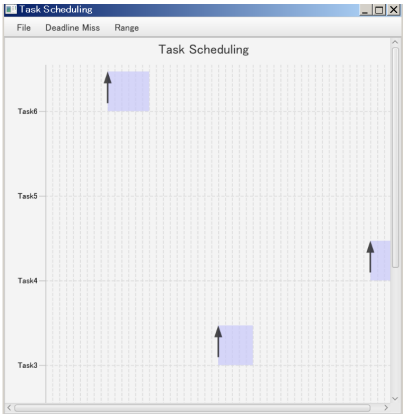

図 4.1: ファイルを開いた直後の画面

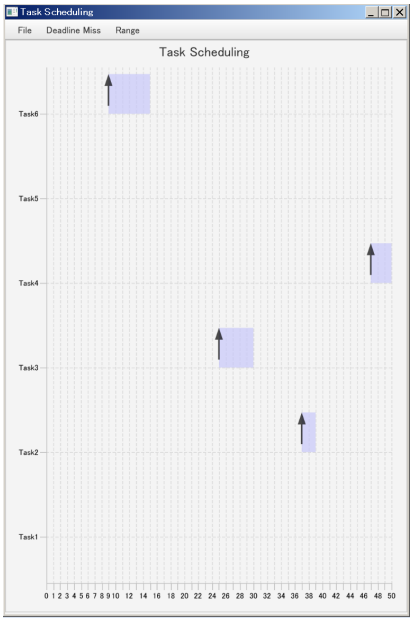

図 4.2: 画面の縦横のサイズの変更し、全てのタスクが表示された画面

その後ティック 45〜177 から画像エクスポートした結果は図 4.3 である。

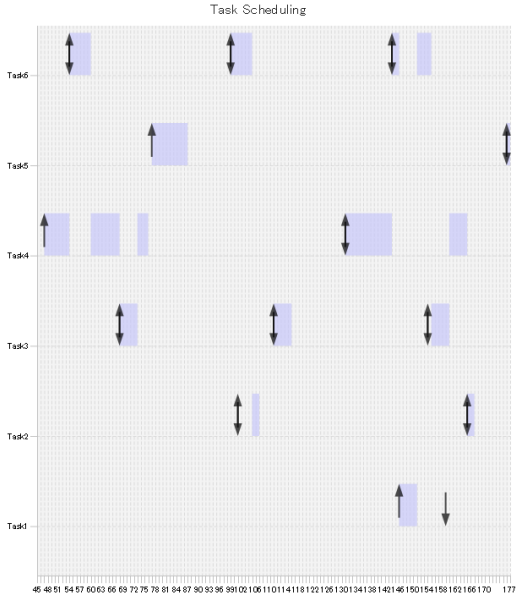

図 4.3: ティック 45〜177 を範囲指定して実行した結果

この実行結果において、周期タスクであるタスク 2〜6 は 2 周期目以降からは前の周期のデッドラインと次の周期の開始が重なるため、周期の開始が上下矢印で表現されており、非周期タスクであるタスク 1 と見分けることができる。タスク 4 は実行時間が長いため、1 周期内で何度か他のタスクに優先順位が移り、タスクの実行が中断されその後再開されている。その状態は入力ファイルでは Listing4.1 のように同じジョブ番号でジョブのレコードが複数件あることで表現されている。図 4.3 でもタスク 4 が 1 周期内で中断・再

開されている部分も正しく表示されていることが分かる。

Listing 4.1: タスク 4 が 1 周期内で中断と再開が発生している部分

```
47,4,1,RL
47,4,1,ST
54,4,1,ED
...
60,4,1,ST
68,4,1,ED
...
73,4,1,ST
76,4,1,ED
131,4,1,DL
131,4,2,RL
```

このシミュレーションはシングル CPU を前提として実行しているが、どの部分でもタスク間で CPU 実行時間が重なっていないことから、シミュレーションが正しく実行されたことが確認できる。またシミュレータに問題がある場合、タスクの起動要求やデッドラインの数が合わなかったり、周期タスクの周期の開始・終了が正しく表現できない等の問題が発生する可能性があるが、この表示結果ではそのような問題は発生していないことが確認できる。また、このシミュレーション結果ではデッドラインミスが存在しないため、デッドラインミス検索を行うと、図 4.4 のようにデッドラインミスが存在しないというメッセージが表示される。

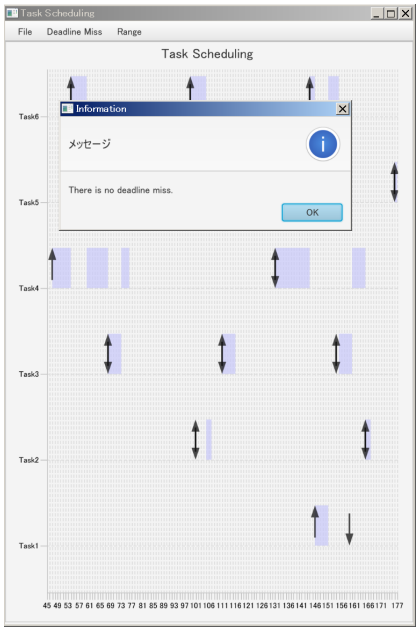

図 4.4: デッドラインミスが存在しないことを示す表示

次に CPU 使用率 80%でタスク数 5, デッドラインミスありの場合について記述する。

- タスク 1 : 非周期タスク (実行時間 3)
- タスク 2 : 周期タスク (周期 5, 実行時間 2)
- タスク 3 : 周期タスク (周期 15, 実行時間 3)
- タスク 4 : 周期タスク (周期 70, 実行時間 2)
- タスク 5 : 周期タスク (周期 70, 実行時間 12)

このシミュレーション結果で最初に表示されるティック 0〜50 のデータを表示した画面が図 4.5 である。この段階ではまだタスク 1, タスク 4, タスク 5 は起動されていない。

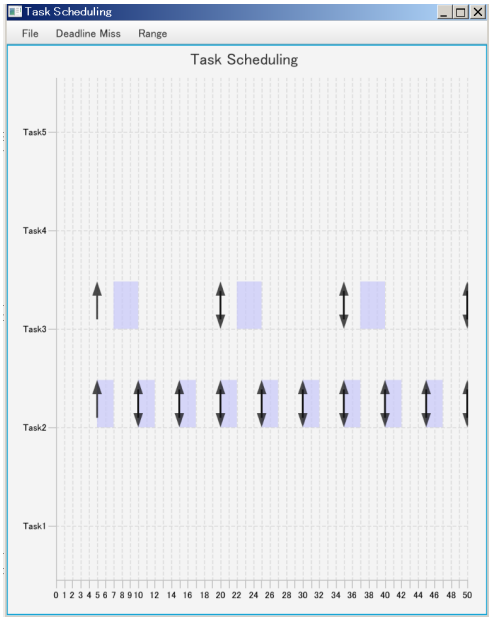

図 4.5: 初期表示のティック 0〜50 のデータが表示されている画面

次にティック 63〜140 までを範囲指定して表示し、ズームしたものを画像エクスポートしたものが図 4.6 である。この図ではタスク 2〜タスク 5 が周期タスク, タスク 1 が非周期タスクであることが確認できる。

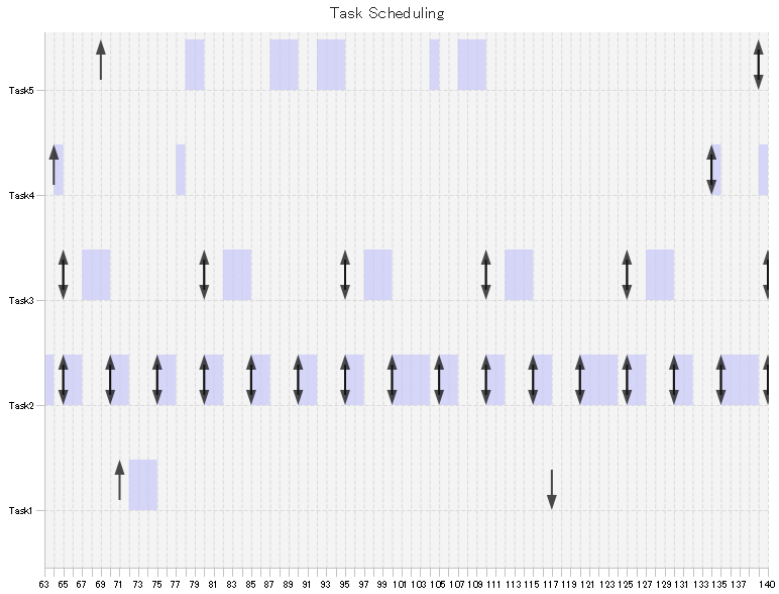

図 4.6: ティック 63〜140 を範囲指定して実行した結果

さらにメニューから Deadline Miss → Next を選択してデッドラインミス検索を行うと図 4.7 のように最初のデッドラインミスが中央に表示される。最初のデッドラインミスはティック 703 で発生しており、デッドラインミス検索によりティック 0 から順に確認する必要はなく、一度にデッドラインミスが発生している箇所にジャンプすることが可能である。再度メニューから Deadline Miss → Next を選択してデッドラインミス検索を行うと図 4.8 のように 2 番目のデッドラインミスが中央に表示される。

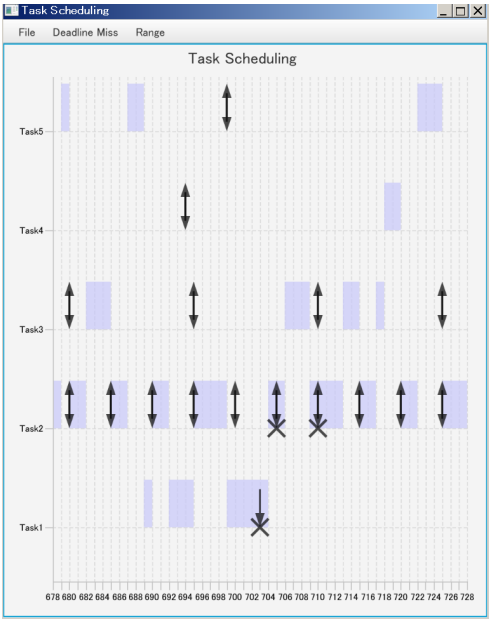

図 4.7: デッドラインミス検索で最初のデッドラインミスが表示された画面

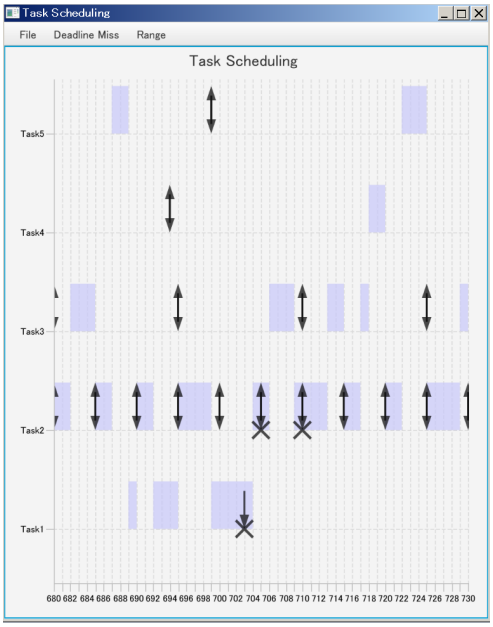

図 4.8: デッドラインミス検索で 2 番目のデッドラインミスが表示された画面

このように実際のシミュレーション結果でもデッドラインミス検索の機能が有効であることを確認した。

第5章　まとめ

本研究ではリアルタイム・タスク・シミュレーション結果を可視化して分析を容易にすることを目的として、JavaFX を使用してリアルタイム・タスク・シミュレーション結果をグラフとして表示する GUI 分析ツールを開発した。入力ファイルは CSV ファイルとし、入力ファイルをファイル選択ダイアログから選択する機能、各イベント (タスク実行、起動要求、デッドライン、デッドラインミス) のグラフ表示やスクロール表示、ズームイン/ズームアウト、画像ファイルへのエクスポート、デッドラインミス検索の機能を開発した。10000 ティック程度の長い範囲のデータを想定し、表示するティックの範囲を絞ってグラフを部分表示する機能も開発した。開発時に非常に長いティック範囲のデータをグラフ表示してズームインや画像ファイルへのエクスポートを行った際に問題が発生したが、画像のレンダリングをソフトウェア側で行うように指定する JVM 引数を追加することで解決できた。さらに、本 GUI 分析ツールを実際のタスク・シミュレーション結果に適用し、周期タスク・非周期タスクともにイベントが正しく表示できることを確認し、ズームイン/ズームアウトでイベントを見やすく表示したり、画像ファイルへのエクスポートを行ったり、デッドラインミス検索を行い、実際のタスク・シミュレーション結果でも本 GUI ツールが有用であることを確認した。今後は長いティック範囲でグラフを表示した場合にグラフだけでなく軸も一緒にスクロールしてタスク番号が常に見えるように改良したい。

参考文献

[1] “Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications”, 3rd Edition, Springer, 2011

[2] “Oracle JDK 8 and JRE 8 Certified System Configurations Contents”, http://www.oracle.com/technetwork/java/javase/certconfig-2095354.html, 2017 年 1 月 29 日アクセス

[3] “API ドキュメント - Overview (JavaFX 8)”, https://docs.oracle.com/javase/jp/8/javafx/api/toc.htm, 2017 年 1 月 29 日アクセス

[4] “JavaFX: JavaFX シーン・グラフの操作”, https://docs.oracle.com/javase/jp/8/javafx/scene-graph-tutorial/scenegraph.htm, 2017 年 1 月 31 日アクセス

[5] “Node (JavaFX 8)”, https://docs.oracle.com/javase/jp/8/javafx/api/javafx/scene/chart/XYChart.Data.html#Data-X-Y-java.lang.Object-, 2017 年 1 月 31 日アクセス

[6] “Gantt chart from scratch”, http://stackoverflow.com/questions/27975898/gantt-chart-from-scratch, 2017 年 1 月 29 日アクセス

[7] “Zooming inside a Scrollpane”, https://pixelduke.wordpress.com/2012/09/16/zooming-inside-a-scrollpane/, 2017 年 1 月 29 日アクセス

[8] “Node (JavaFX 8)”, https://docs.oracle.com/javase/jp/8/javafx/api/javafx/scene/Node.html#snapshot-javafx.util.Callback-javafx.scene.SnapshotParameters-javafx.scene.image.WritableImage-, 2017 年 1 月 29 日アクセス

[9] “Using the Image Ops API”, http://docs.oracle.com/javafx/2/image_ops/jfxpub-image_ops.htm, 2017 年 1 月 29 日アクセス

[10] “How to generate chart image using JavaFX chart API for export without displying first”, http://stackoverflow.com/questions/29721289/how-to-generate-chart-image-using-javafx-chart-api-for-export-without-displying, 2017 年 1 月 29 日アクセス

[11] “JavaFX スタート・ガイド”, https://docs.oracle.com/javase/jp/8/javafx/get-started-tutorial/jfx-architecture.htm#A1106308, 2017 年 1 月 29 日アクセス