平成 22 年度　　修士論文

トレースログ可視化ツールにおける
統計情報表示機能

提 出 日　平成 23 年 2 月 14 日

主査　山本　晋一郎 教授

副査　戸田　尚宏 教授

副査　太田　淳 准教授

所　属　愛知県立大学
情報科学研究科
情報システム専攻

学生番号　2009834003

市原　大輔

目次

第 1 章

はじめに

近年，プロセッサにおけるクロック周波数が実質上限に達し，これ以上の高速化は難しくなってきた．また，高クロックによる高消費電力，高発熱により，バッテリの消費速度やマシンへの影響も懸念される．そこで，マルチプロセッサシステムによる処理の並列化により，1 つ 1 つのプロセッサのクロック周波数を抑え，消費電力や発熱を抑えつつ，処理の高速化を実現する動きが活発になっている．

こうしたマルチコアプロセッサが代表するマルチプロセッサ環境では，処理の並列性からプログラムの挙動が非決定的になり，プログラムの挙動が実行する毎に異なる．そのため，ブレークポイントやステップ実行を用いたシングルプロセッサ環境で用いられているデバッグ手法を用いることができない．

このような理由から，マルチプロセッサ環境では，プログラム実行履歴であるトレースログの解析を解析することでデバッグを行う．しかしながら，トレースログは量が膨大な上に，マルチプロセッサ環境ではプロセッサ毎にログが分散し，そのテキストを直接目で追うなどしてデバッグを行うのは困難である．

この解析作業を支援するために，トレースログを可視化するツールが開発された．しかしながら，このツールは，デバッグのターゲットとなるシステム毎に用意される事が多く，ターゲットシステムに対する汎用性に乏しい．また，可視化する情報や方法がツールで限定されてしまい，ユーザが望む情報の可視化を実現することが難しく，拡張性に乏しい．

これらを解決すべく，OJL(On the Job Learning)[7, 8] の開発テーマとして OJL1 期生により，トレースログ可視化ツール **TraceLogVisualizer (TLV)** が開発された．TLV は，汎用性を実現するために，入力されたトレースログを自身が可視化処理で扱う形式のトレースログへ変換する仕組みがある．また，拡張性を実現するために，可視化の際に用いられる図形データをユーザが定義できる仕組みがある．これらの仕組みを形式化したの

が，変換ルール，可視化ルールである．

この TLV の開発は，OJL2 期生，3 期生，4 期生と続けて行われている．OJL2 期生は，OJL1 期生が作り上げた TLV に対する単体テストとシステムテストを実施し，問題解決を行った．また，ユーザから要求を収集し，複雑な処理を実現するスクリプト拡張機能と，アプリケーションが出力したメッセージをトレースログと同様に可視化するアプリログ機能を追加した．その後，TLV1.1 が一般向けに公開された．

そして OJL3 期生である筆者は，TLV に対する要求に対応した．収集した要求のうち，“CPU 利用率などの統計情報のグラフ表示” と “TLV の高速化” に対する要求が強かったので，これらの実現を試みた．

トレースログの解析において、あるイベントが何回行われたのか，ある状態がどれほど続いたのか，といったトレースログの統計情報が必要になる．TLV では，トレースログの時系列情報を扱うツールであり，こうした統計情報は，手作業や独自に開発したスクリプトなどで得る必要がある．手作業で行う場合，トレースログを可視化したとしてもコストがかかり，また，見落としなどで正確性に欠ける．さらに，得た統計情報をグラフ化するためにグラフ作成ツールを使用する必要があり，解析作業が非効率である．

こういった背景から，“CPU 利用率などの統計情報のグラフ表示” する機能が求められた．これを実現するため，統計情報の生成とそれをグラフ表示する仕組みとして，新たに統計情報の生成とグラフの設定を行うルールを追加し，グラフを描画するウィンドウを実装した．統計情報の生成方法を複数用意することで，様々なユースケース，様々な統計情報に対応した．

また，TLV の保守に支障をきたすコードのリファクタリングを行った．対象コードは，複雑な正規表現である．それをパーサ・コンビネータという技法を用いて作成したパーサでリファクタリングを行った結果，可読性の向上の他に，微少ながら高速化に成功した．

最後に，本報告書の構成を述べる．

2 章では，TLV の設計について述べる．

3 章では，ユーザから得られた TLV に対する要求について述べる．

4 章では，統計情報を生成してグラフ表示する機能について述べる．

5 章では，高速化と保守性向上のためのリファクタリングについて述べる．

6 章では，第 3 期 OJL における筆者の実績について述べる．

最後に 7 章で所感と本報告書のまとめ，今後の課題について述べる．

第 2 章

トレースログ可視化ツール (TLV)

2.1 TLV における汎用性と拡張性

TLV は，汎用性と拡張性を実現することを目標としている．

汎用性とは，可視化表示したいトレースログの形式を制限しないことであり，可視化表示の仕組みをトレースログの形式に依存させないことによって実現する．具体的には，まず，トレースログを抽象的に扱えるように，トレースログを一般化した標準形式トレースログを定義する．そして，任意の形式のトレースログを標準形式トレースログに変換する仕組みを，変換ルールとして形式化する．変換ルールの記述で任意のトレースログが標準形式トレースログに変換することができるため，あらゆるトレースログの可視化に対応することが可能となる．

拡張性とは，トレースログに対応する可視化表現をユーザレベルで拡張できることを表し，トレースログから可視化表示を行う仕組みを抽象化し，それを可視化ルールとして形式化して定義することで実現する．可視化ルールを記述することにより，トレースログ内の任意の情報を自由な表現方法で可視化することが可能になる．

2.2 TraceLogVisualizer の設計

TLV の主機能は，2 つのプロセスと 6 種の外部ファイルによって実現される．TLV の全体像を図 2.1 に示す．

2 つのプロセスとは，標準形式への変換と，図形データの生成である．標準形式への変換は，任意の形式をもつトレースログを標準形式トレースログに変換する処理である．この処理には，外部ファイルとして変換元のトレースログファイル，標準形式トレースログ

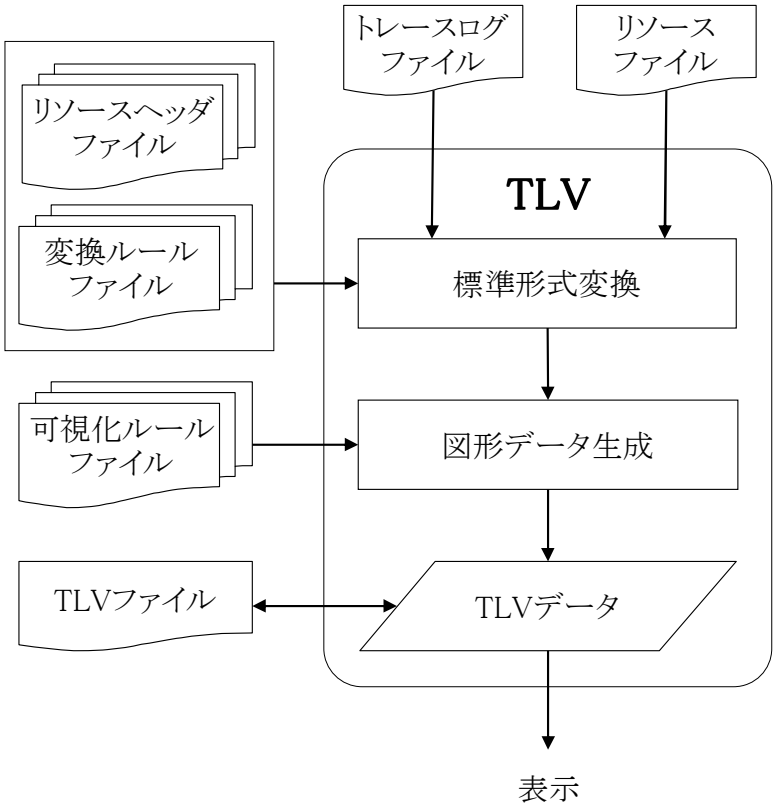

図 2.1　TLV の全体像

に登場するリソースを定義したリソースファイル，リソースタイプを定義したリソースヘッダファイル，標準形式トレースログへの変換ルールを定義した変換ルールファイルが読み込まれる．

また，図形データの生成は，標準形式トレースログに対して、可視化ルールと呼ぶ

を適用し図形データを生成する処理である．この処理には外部ファイルとして可視化ルールファイルが読み込まれる．可視化ルールファイルとは，図形と可視化ルールの定義を記述したファイルである．

生成された標準形式トレースログと図形データは，TLV データとしてまとめられ可視化表示の元データとして用いられる．TLV データは TLV ファイルとして外部ファイルに保存することが可能であり，TLV ファイルを読み込むことで，標準形式変換と図形データ生成の処理を行わなくても可視化表示できるようになる．

図 2.2 に，TLV のスクリーンショットを示す．

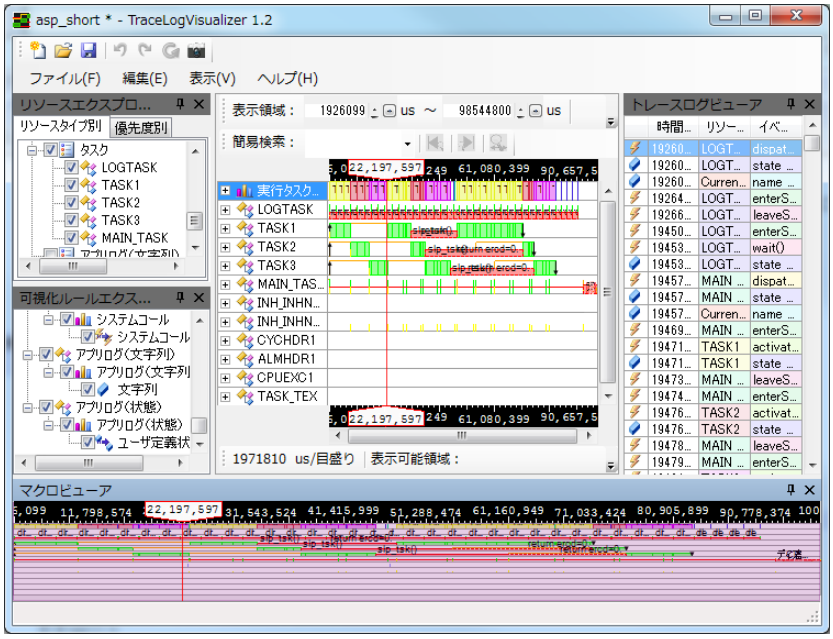

図 2.2　TLV のスクリーンショット

2.3　標準形式トレースログ

2.3.1　トレースログの抽象化

標準形式トレースログを提案するにあたり，トレースログの抽象化を行った．結果を以下にまとめる．

トレースログ

時系列にイベントを記録したもの．

イベント

リソースの属性の値の変化，リソースの振る舞い．

リソース

イベントの発生源．固有の名前，属性をもつ．

リソースタイプ

リソースの型．リソースの属性，振る舞いを特徴付ける．

属性

リソースが固有にもつ情報．文字列，数値，真偽値のいずれかで表現されるスカ

ラーデータで表される．

振る舞い

リソースの行為．主に属性の値の変化を伴わない行為をイベントとして記録するために用いることを想定している．振る舞いは任意の数のスカラーデータを引数として受け取ることができる．

2.3.2 標準形式トレースログの定義

本小節では，前小節で抽象化したトレースログを，標準形式トレースログとして形式化する．標準形式トレースログの定義は，EBNF(Extended Backus Naur Form) および終端記号として正規表現を用いて行う．正規表現はスラッシュ記号 (/) で挟むものとする．

前小節によれば，トレースログは，時系列にイベントを記録したものであるので，1つのログには時刻とイベントが含まれるべきである．トレースログが記録されたファイルのデータを TraceLog，TraceLog を改行記号で区切った 1 行を TraceLogLine とすると，これらは次の EBNF で表現される．

```
TraceLog = { TraceLogLine,"\n" };
TraceLogLine = "[",Time,"]",Event;
```

TraceLogLine は”[”,”]”で時刻を囲み，その後ろにイベントを記述するものとする．

時刻は Time として定義され，次に示すように数値とアルファベットで構成するものとする．

```
Time = /[0-9a-Z]+/;
```

アルファベットが含まれるのは，10 進数以外の時刻を表現できるようにするためである．これは，時刻の単位として「秒」以外のもの，たとえば「実行命令数」などを表現できるように考慮したためである．この定義から，時刻には，2 進数から 36 進数までを指定できることがわかる．

前小節にて，イベントを，リソースの属性の値の変化，リソースの振る舞いと抽象化した．そのため，イベントを次のように定義した．

```
Event = Resource,".",(AttributeChange|BehaviorHappen);
```

Resource はリソースを表し，AttributeChange は属性の値の変化イベント，

BehaviorHappen は振る舞いイベントを表す．リソースはリソース名による直接指定，あるいはリソースタイプ名と属性条件による条件指定の 2 通りの指定方法を用意した．

リソースの定義を次に示す．

```
Resource = ResourceName
         | ResourceTypeName,"(",AttributeCondition,")";
ResourceName = Name;
ResourceTypeName = Name;
Name = /[0-9a-Z_]+/;
```

リソースとリソースタイプの名前は数値とアルファベット，アンダーバーで構成されるとした．AttributeCondition は属性条件指定記述である．これは次のように定義する．

```
AttributeCondition = BooleanExpression;
BooleanExpression = Boolean
   |ComparisonExpression
   |BooleanExpression,[{LogicalOpe,BooleanExpression}]
   |"(",BooleanExpression,")";
Boolean = "true"|"false";
ComparisonExpression = AttributeName,ComparisonOpe,Value;
LogicalOpe = "&&"|"||";
ComparisonOpe = "=="|"!="|"<"|">"|"<="|">=";
```

属性条件指定は，論理式で表され，命題として属性の値の条件式を，等価演算子や比較演算子を用いて記述できるとした．

AttributeName は属性の名前であり，リソース名やリソースタイプ名と同様に，次のように定義する．

```
AttributeName = Name;
```

イベントの定義にて，AttributeChange は属性の値の変化，BehaviorHappen は振る舞いとした．これらは，リソースとドット”.”でつなげることでそのリソース固有のものであることを示す．リソースの属性の値の変化と振る舞いは次のように定義した．

```
AttributeChange = AttributeName,"=",Value;
Value = /[^"\\]+/;
BehaviorHappen =  BehaviorName,"(",Arguments,")";
BehaviorName = Name;
Arguments = [{Argument,[","]}];
Argument = /[^"\\]*/;
```

属性の変化イベントは，属性名と変化後の値を代入演算子でつなぐことで記述し，振る舞いイベントは，振る舞い名に続けてカンマで区切った引数を括弧”()”で囲み記述するとした．

2.3.3 標準形式トレースログの例

標準形式トレースログの定義を元に記述した，標準形式トレースログの例を図 2.3 に示す．

```
1 [2403010]MAIN_TASK.leaveSVC(ena_tex,ercd=0)
2 [4496099]MAIN_TASK.state=READY
3 [4496802]TASK(state==RUNNING).state=READY
```

図 2.3　標準形式トレースログの例

1 行目がリソースの振る舞いイベントであり，2 行目，3 行目が属性の値の変化イベントである．1 行目の振る舞いイベントには引数が指定されている．

1 行目，2 行目はリソースを名前で直接指定しているが，3 行目はリソースタイプと属性の条件によってリソースを特定している．

2.3.4 JSON

リソースファイル，リソースヘッダファイル，変換ルールファイル，可視化ルールファイルは，JSON(JavaScript Object Notation)[5] と呼ばれるデータ記述言語を用いて記述する．

JSON は，主にウェブブラウザなどで使用される ECMA-262，revision 3 準拠の JavaScript（ECMAScript）と呼ばれるスクリプト言語のオブジェクト表記法をベースと

しており，RFC 4627 として仕様が規定されている．JSON は Unicode のテキストデータで構成され，バイナリデータを扱うことはできない．また，JSON ではシンタックスのみの規定がなされ，セマンティクスは規定されていない．

JSON の特徴は，シンタックスが単純であることである．これは，人間にとっても読み書きし易く，コンピュータにとっても解析し易いことを意味する．また，複数のプログラミング言語で JSON ファイルを扱うライブラリが実装されており，異なる言語間のデータ受け渡しに最適である．JSON が利用可能なプログラミング言語としては，ActionScript, C, C++, C#, ColdFusion, Common Lisp, Curl, D, Delphi, E, Erlang, Haskell, Java, JavaScript (ECMAScript), Lisp, Lua, ML, Objective CAML, Perl, PHP, Python, Rebol, Ruby, Scala, Squeak などがある．

TLV の各ファイルのフォーマットに JSON を採用した理由はこれらの特徴による．シンタックスが単純であることにより，ユーザの記述コスト，習得コストを低減させることができ，また，複数のプログラミング言語でパース可能であることによりファイルに可搬性を持たせることができるからである．

JSON で表現するデータ型は以下のとおりであり，これらを組み合わせることでデータを記述する．

- 数値 (整数，浮動小数点)
- 文字列 (Unicode)
- 真偽値 (true，false)
- 配列 (順序付きリスト)
- オブジェクト (ディクショナリ，ハッシュテーブル)
- null

JSON の文法を EBNF と正規表現を用いて説明する．

JSON は，次に示すようにオブジェクトか配列で構成される．

```
JSONText = Object | Array;
```

オブジェクトは複数のメンバをカンマで区切り，中括弧で囲んで表現する．メンバは名前と値で構成され，名前と値はセミコロンで区切られる．メンバの名前は値であり，データ型は文字列である．オブジェクトの定義を次に示す．

```
Object = "{",Member,[{",",Member}],"}";
Member = String,":",Value;
```

配列は複数の値を持つ順序付きリストであり，値をコンマで区切り，角括弧で囲んで表現する．次に配列の定義を示す．

```
Array = "[",Value,[{",",Value}],"]";
```

値は，文字列，数値，オブジェクト，配列，真偽値，null のいずれかである．文字列はダブルクオーテーションで囲まれた Unicode 列である．数値は 10 進法表記であり，指数表記も可能である．値の定義を次に示す．

```
Value = String|Number|Object|Array|Boolean|"null";
String = /"([^"\]|\n|\"|\\|\b|\f|\r|\t|\u[0-9a-fA-F]{4})*"/;
Boolean = "true"|"false";
Number = ["-"],("0"|Digit1-9,[Digit]),[".",Digit],Exp;
Exp = ["e",[("+"|"-")],Digit];
Digit = /[0-9]+/;
Digit1-9 = /[1-9]/;
```

図 2.4 に JSON におけるオブジェクトを定義した例を示す．

```
{
  "Image":{
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail":{
      "Url": "http://www.example.com/image/481989943",
      "Height": 125,
      "Width": "100"
    },
    "IDs": [116, 943, 234, 38793]
  }
}
```

図 2.4　JSON におけるオブジェクトを定義した例

図 2.5 に JSON における配列を定義した例を示す．

```
[
  {
    "City": "SAN FRANCISCO",
    "State": "CA",
    "Zip": "94107",
    "Country": "US"
  },
  {
    "City": "SUNNYVALE",
    "State": "CA",
    "Zip": "94085",
    "Country": "US"
  },
  {
    "City": "HEMET",
    "State": "CA",
    "Zip": "92544",
    "Country": "US"
  }
]
```

図 2.5　JSON における配列を定義した例

2.4　TraceLogVisualizer のその他の機能

本節では，標準形式変換と可視化データ生成の他に TLV が備える機能について述べる．

2.4.1　マーカー

TLV では，可視化表示部にマーカーと呼ぶ印を指定の時刻に配置することができる．注目するべきイベントの発生時刻にマーカーを配置することで，可視化表示内容の理解を補助することができる．

2.4.2　可視化表示部の制御

可視化表示ツールでは，可視化表示部を制御する操作性が可視化表示ツール全体の操作性に大きく影響するため，TLV では状況に合わせて様々な操作で制御を行えるようにした．TLV では，可視化表示部の制御として，表示領域の拡大縮小，移動を行うことができ

る．これらの操作方法として，キーボードによる操作，マウスによる操作，値の入力による操作の 3 つの方法を提供した．

マウスによる操作は，クリックによる操作，ホイールによる操作，選択による操作，ドラッグによる操作がある．クリックによる操作はカーソルを虫眼鏡カーソルに変更してから行う．左クリックでカーソル位置を中心に拡大，右クリックで縮小を行う．また，左ダブルクリックを行うことでクリック箇所が中心になるように移動する．ホイールによる操作は，コントロールキーを押しながらホイールすることで移動を行い，シフトキーを押しながら上へホイールすることでカーソル位置を中心に拡大，下へホイールすることで縮小する．選択による操作では，マウス操作により領域を選択し，その領域が表示領域になるように拡大する．ドラッグによる操作は，表示領域をドラッグすることで表示領域の移動を行う．

キーボードによる操作は，可視化表示部において方向キーを押すことで行う．左キーで表示領域を左に移動し，右キーで右に移動する．

値の入力による操作では，より詳細な制御を行うことができる．可視化表示部の上部にはツールバーが用意されており，そこで表示領域の開始時刻と終了時刻を直接入力することができる．

2.4.3 マクロ表示

TLV の要求分析を行った際，可視化表示部で拡大した場合に全体の内どの領域を表示しているのかを知りたいという要求があった．そのため，TLV では，マクロビューアというウィンドウを実装した．

マクロビューアでは，トレースログに含まれるイベントの最小時刻から最大時刻までを常に可視化表示しているウィンドウで，可視化表示部で表示している時間を半透明の色で塗りつぶして表示する．塗りつぶし領域のサイズをマウスで変更することができ，それに対応して可視化表示部の表示領域を変更することができる．

マクロビューアでは可視化表示部と同じように，キーボード，マウスにより，可視化表示部の表示領域を拡大縮小，移動の制御を行うことができる．

2.4.4 トレースログのテキスト表示

TLV では，標準形式トレースログをテキストで表示するウィンドウを実装した．ここでは，トレースログの内容を確認することができる．

可視化表示部とテキスト表示ウィンドウは連携しており，テキスト表示ウィンドウ上でカーソルを移動させると，カーソル位置にあるトレースログの時刻にあわせて可視化表示部のカーソルが表示されたり，ダブルクリックすることで対応する時刻に可視化表示部を移動することができる．また，可視化表示部でダブルクリックすることで，ダブルクリック位置にある図形に対応したトレースログが，テキスト表示ウィンドウの先頭に表示されるようになっている．

2.4.5 可視化表示項目の表示/非表示切り替え

TLV では，可視化表示する項目を可視化ルール単位，リソース単位で切り替えることができる．この操作は，リソースウィンドウと可視化ルールウィンドウで行う．リソースウィンドウでは，リソースファイルで定義されたリソースを，リソースタイプやグループ化可能な属性毎にツリービュー形式で表示しており，チェックの有無でリソース毎に表示の切り替えを行う．同じように，可視化ルールウィンドウでは，可視化ルールごとに表示の切り替えを行える．

2.4.6 スクリプト拡張機能

標準形式トレースログへの変換処理，図形生成処理を，外部スクリプトで行えるようにすることで，変換・可視化ルールでは記述できない複雑な可視化を実現する．

また，TLV でもっとも処理時間を要する変換処理は，変換ルールによる処理より外部スクリプトによる処理の方が高速である場合がある．

2.4.7 アプリログ機能

アプリケーションにおける，いわゆる printf デバッグを行うための機能である．これにより，printf デバッグによる出力を TLV のタイムライン上に表示させることによって，メッセージが出力された時のタスク状態等が一目でわかるようになる．

2.4.8 リロード機能

変換ルール・可視化ルールを記述する際に，記述を変更するたびにトレースログを開き直す必要があり不便である，という要望があった．

そこで，現在開いているトレースログを再度，変換・可視化を行なうリロード機能を追

加した.

2.5 第 3 期 OJL で追加した機能

第 3 期 OJL で追加した機能について述べる.

2.5.1 統計情報表示機能

ユーザから,「統計情報は，特に長時間のトレースログにおいて，どのコアでアイドルが多いのかなどの解析に役立つので，統計情報を生成してグラフ表示する機能がほしい」という要望があった.

そこで，統計情報の生成方法と軸ラベルなどのグラフの設定を指定することで，統計情報を生成してグラフ表示する機能を実装した.

詳細は，4 章で述べる.

2.6 活用事例

名古屋大学大学院情報科学研究科附属組込みシステム研究センター (NCES) [*1]内の 7 件のプロジェクトのうち，2 件のプロジェクトによって利用されている．また，同 NCES のコンソーシアム型共同研究においても利用されている.

2.7 過去の実績

2.7.1 開発プロセス

TLV は，OJL の開発テーマとして開発された.

TLV はプロジェクトベースで開発が行われ，前年度までは企業出身者 1 名と教員 2 名がプロジェクトマネージャを務め，学生累計 3 名が開発実務を行ってきた．進捗の報告は，週に 1 度のミーティングと週報の提出により行った.

[*1] http://www.nces.is.nagoya-u.ac.jp/

2.7.2 フェーズ分割

TLV の開発は複数フェーズに分割して行われている．フェーズ 1,2,3 は主に OJL1 期生によって実施された．フェーズ 3,4,5 は主に OJL2 期生によって実施された．本報告書ではフェーズ 5 以降について述べる．

第 3 期 OJL 以前のフェーズの内容は次に述べる．

2.7.2.1 フェーズ 1,2,3

2007 年 5 月〜2008 年 3 月までに，OJL1 期生によって実施され，TLV の主要機能の実装が行われた [1].

2.7.2.2 フェーズ 3,4,5

2008 年 10 月〜2010 年 3 月までに，OJL2 期生によって実施された [3, 4]．主な実施内容を次に示す.

- 単体テストおよびシステムテスト
- 要求の収集と対応
- TLV 1.0 の TOPPERS[6] 会員への公開の後，TLV 1.1 を一般向けに公開
- リファクタリング方針の検討

第 3 章

TLV に対する要求

3.1　ユーザミーティングでの要求収集

TLV は，現在，基本的な機能が完成し，機能追加および保守の段階にある．よって，TLV を開発するにあたり，ユーザミーティングを開催し，TLV に対する要求を収集した．収集した要求の中でも，「CPU 利用率などの統計情報のグラフ表示」および「TLV の高速化」は，過去にも要求があったので，これらに対応することにした．

3.2　CPU 利用率などの統計情報のグラフ表示

統計情報とは，データを調査して，そのデータが持つ特性を数量的に表したものである．トレースログの解析においても，統計情報は有用である．例えば，ユーザが求めるマルチプロセッサ対応 RTOS のトレースログにおける統計情報は，次のようなものがある．

- タスクの CPU 利用率
- 各 CPU のアイドル時間
- イベント数
- タスクの実行回数
- プリエンプト回数
- ディスパッチ回数
- タスクが実行可能状態になってから実行状態になるまでの時間
- デッドラインミス回数
- デッドラインまでの時間

これらの統計情報について，求められる理由をいくつか紹介する．例えば，各 CPU のアイドル時間は，マルチプロセッサ環境で生じる CPU 毎のアイドル時間の偏りを発見でき，各 CPU へのタスク割り当てが効率よく行われているか確かめることができる．また，デッドラインミス回数は，時間制約の強い RTOS において，タスクの完了期限であるデッドラインをオーバーすることは許されないのでチェックする必要がある．

こうした統計情報は，多くの場合，グラフとして表現する．そうすることで，統計情報のもととなるデータの様々な特性が視覚的に読み取れるようになる．

しかし，現状の TLV では，トレースログの統計情報を扱うことはできない．なぜなら，現状の TLV が扱うデータは，トレースログという時系列情報だからである．統計情報を扱うツールがない場合，次のような問題がある．

まず，統計情報を得るコストが大きい．ユーザは，可視化されたトレースログを手作業で数える必要があり，また，この作業は見落としが大いに考えられ，正確性に欠ける．手作業による統計情報の取得の他に，スクリプトを利用する方法もある．この方法は，スクリプトの開発コストはかかるものの，開発後は手作業よりも作業コストを抑えられ，正確性も向上する．しかしながら，TLV を用いた解析作業中に，ある特定区間における統計情報が必要になった場合，TLV 以外のアプリケーションに目的の区間情報を入力して実行する必要がある．また，統計情報をグラフ化するためにグラフ作成ツールの利用も考えられる．このように，スクリプトを利用する方法では，複数のアプリケーションを利用する必要があるので，解析作業が非効率である．

したがって，低コストで信頼性の高い統計情報を得られ，また，作業を効率的に行えるよう，統計情報を得てグラフとして表示する機能が求められた．そこで我々は，これらの要求を実現する機能を統計情報表示機能と名付けて実装した．この機能について，4 章で説明する．

3.3 TLV の高速化

TLV は，トレースログの可視化に時間がかかる．例えば，約 7,000 行[*1]のトレースログに対して約 3 分の時間を要する．約 50,000 行のトレースログに対しては，10 時間以上という実用に耐えない時間を要した，という報告もある．

この問題に対して，OJL2 期生が調査を行った [3]．結果，複雑な処理を行うメソッドが

[*1] プロセッサを 2 個利用したシステムで，マルチプロセッサ対応 RTOS である TOPPERS/FMP カーネル [10] が約 252 秒の間に出力した行数である．

存在しており，ソースコードを複雑化させていたことが判明した．このような状況では，保守や高速化が難しいので，リファクタリングをすることになった．

問題となっているのは，標準形式トレースログのパーシングにおける正規表現の多用と可視化ルールにおける暗黙的な木構造である．

まず，前者について説明する．正規表現は，文字列のマッチングを柔軟に行えるが，どのような文字列をマッチングするのか解読するコストを要する．そのため，正規表現を多用すると，コードの可読性が低下する．

次に，後者について説明する．可視化ルールを表現するクラス群の中に，概念的に木構造となるクラスがあるが，クラス設計よって木構造で表現されていない．そのため，木構造をトラバースするためのコードが多数の場所に存在することになり，保守性と可読性が低下している．

正規表現は，標準形式トレースログを表すクラスで多用されている．このクラスは，TLV の処理で多用されるので，この問題を先に解決することにした．このリファクタリングについて，5 章で説明する．

第 4 章

統計情報表示機能

4.1　概要

統計情報表示機能は，トレースログなどの情報から統計情報を得て，それをグラフとして表示する機能である．統計情報表示機能の概要図を図 4.1 に示す．様々な統計情報生成方法とグラフ設定を提供し，ユーザの要求に対して柔軟に答えられる機能を目指す．

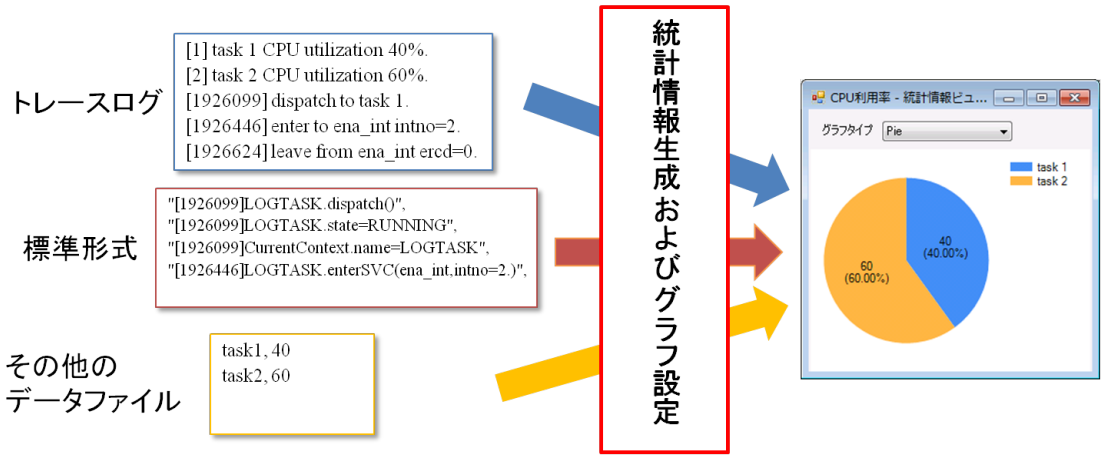

図 4.1　統計情報表示機能の概要図

4.2　設計

統計情報をグラフとして表示するには，統計情報と，グラフの種類や目盛幅などのグラフの設定が必要である．よって，統計情報表示機能の処理として，統計情報を取得・生成

するプロセスとグラフ設定をするプロセスが必要となる．これらのプロセスを制御するため，統計情報の生成方法とグラフの設定を指定する仕組みを，変換・可視化ルールに同様に，統計情報生成ルールとして形式化して定義することで実現する．

本節では，統計情報とグラフに関する要求から，グラフの設定，統計情報の取得・生成方法を検討した結果を述べた後，統計情報とグラフの設定の保存方法について検討した結果を述べる．

4.2.1 統計情報のグラフ表示

統計情報によって適切なグラフやグラフの設定はさまざまである．したがって，どのようなグラフが利用でき，どのようなグラフ設定ができるとよいかをユーザに対してアンケート調査を実施した．アンケートにご協力頂いたのは，TLV ユーザである NCES 研究員 1 名と名古屋大学大学院の学生 1 名の計 2 名である．「どのようなグラフでどのような統計情報を表現したいか」という問いに対する回答をまとめたものが表 4.1 ,「どのようなグラフ設定が行えるとよいか」という問いに対する回答をまとめたものが表 4.2 である．

表 4.1　問：どのようなグラフでどのような統計情報を表現したいか

統計情報	円	縦棒	横棒	折れ線	ヒストグラム	ろうそく
タスクの CPU 利用率	○					
メモリアクセス率	○					
アイドル時間	○					
CPU の処理時間割合	○	○	○			
タスクの合計実行時間		○	○			
タスクの実行時間分布					○	
タスクの切替回数		○		○ (時系列)		
デッドラインミス回数		○		○ (時系列)		
タスクの起動回数		○		○ (時系列)		
タスク起動要求キューイング数		○		○ (時系列)		
タスクの応答時間					○	○
割込み禁止時間					○	○
ロック獲得時間と獲得待ち時間					○	○
セマフォ獲得時間と獲得待ち時間					○	○

表 4.2　問：どのようなグラフ設定が行えるとよいか

グラフ設定	円	縦棒	横棒	折れ線	ヒストグラム	ろうそく
タイトル	○	○	○	○	○	○
X 値のラベル	○	○	○	○	○	○
Y 値のラベル	○	○	○	○	○	○
軸のタイトル		○	○	○	○	○
凡例		○	○	○	○	○
グリッド線の有無		○	○	○	○	○
目盛幅		○	○	○	○	○

グラフの種類では，割合を表現する円グラフ，値比較を表現する棒グラフ，値変化を表現する折れ線グラフ，分散を表現するヒストグラム，最小・最大・平均を表現するグラフ[*1]が必要なことがわかった．分布図も候補に入れていたが，該当する統計情報は今回得られなかった．

グラフの設定では，質問する際にあらかじめ一般的な設定項目を提案して，適宜，追加・削除を行って頂いた．しかしながら，追加も削除もされていなかったため，一般的な設定項目があれば，問題ないということがわかった．よって，表 4.2 に挙げたグラフの設定を用意することになるが，これらは次のように分類できる．なお，設定の名前とグラフにおける反映場所についてを図 4.2 に示す．

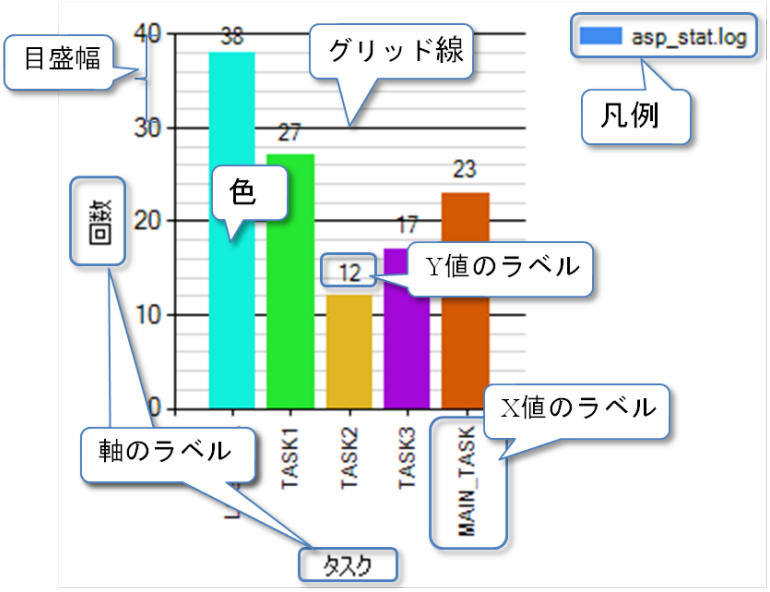

図 4.2　グラフ設定の名前と反映場所

データポイントに依存しない設定

タイトル，軸のタイトル，凡例，グリッド線の有無，目盛幅

データポイントに依存する設定

X 値のラベル，Y 値のラベル，色

ここで，データポイントとは，統計情報の各データのことである．よって，トレースログなどの統計情報のソースが異なれば，データポイントが変化する．具体的には，例えば

[*1] 該当するグラフは複数あるので，表 4.1 では仮に「ろうそく」としている

タスクの起動回数を求めてグラフに表示するとき，X 軸にタスク 1，タスク 2，... と並べて表示するためのものが X 値のラベルであり，プロットされた各データポイント付近にそれぞれ求めた回数の数字を表示するためのものが Y 値のラベルである．

データポイントに依存しない設定は，X 値と Y 値がグラフごとに異なっていても，統計情報の種類が同じであればグラフ間で異なることはない．よって，データポイントに依存しない設定は，特定の統計情報をグラフ表示する際の設定テンプレートとして再利用することが可能である．これを実現することで，設定の冗長を防ぎ，同じ種類の統計情報に関する統計情報生成ルールを記述する手間を軽減できる．

4.2.2 統計情報の生成・取得

4.2.2.1 統計情報のソースと取得・生成方法

統計情報を取得・生成するプロセスでは，まず，統計情報のソースが必要である．考えられるソースとして，トレースログとそれ以外の形式のファイルである．それ以外の形式のファイルとは，システムの出力，または，それ以外の方法で得られたファイルのことである．

次に，統計情報を取得・生成するプロセスでは，取得・生成する方法が必要である．ここでいう取得とは，何らかの形で得られた統計情報を TLV が読み取ることをいい，生成とは，ソースを解析したり，ソースにあるデータを演算を加えたりすることで統計情報を得ることをいう．ソースがトレースログの場合，トレースログにおける時系列情報を解析して統計情報を得ることが考えられる．また，システムが内部で得られた統計情報をトレースログとして出力する場合，それを読み取ることで得ることが考えられる．ソースがトレースログ以外の形式のファイルの場合も，トレースログと同様に，ファイル内のデータを解析する，ファイルに記述されている統計情報を読み取る，といったことで統計情報を得ることが考えられる．

よって，取得・生成する具体的な方法を 3 種類考えた．それらを次に示す．

実現方法 1：正規表現によるデータ取得

文字列のマッチングなので，トレースログやそれ以外のファイルにも対応できる．イメージを図 4.3 に示す．

実現方法 2：外部プロセスによる取得・生成

スクリプトやアプリケーションなどを外部プロセスとして利用することで，複雑な

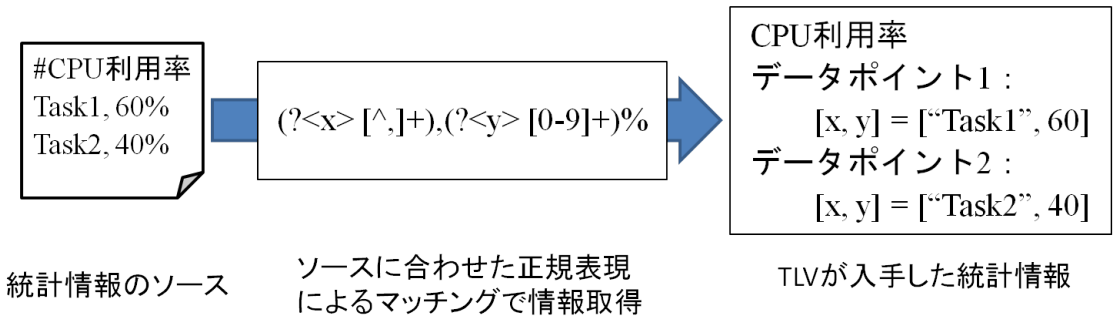

図 4.3　実現方法 1 のイメージ

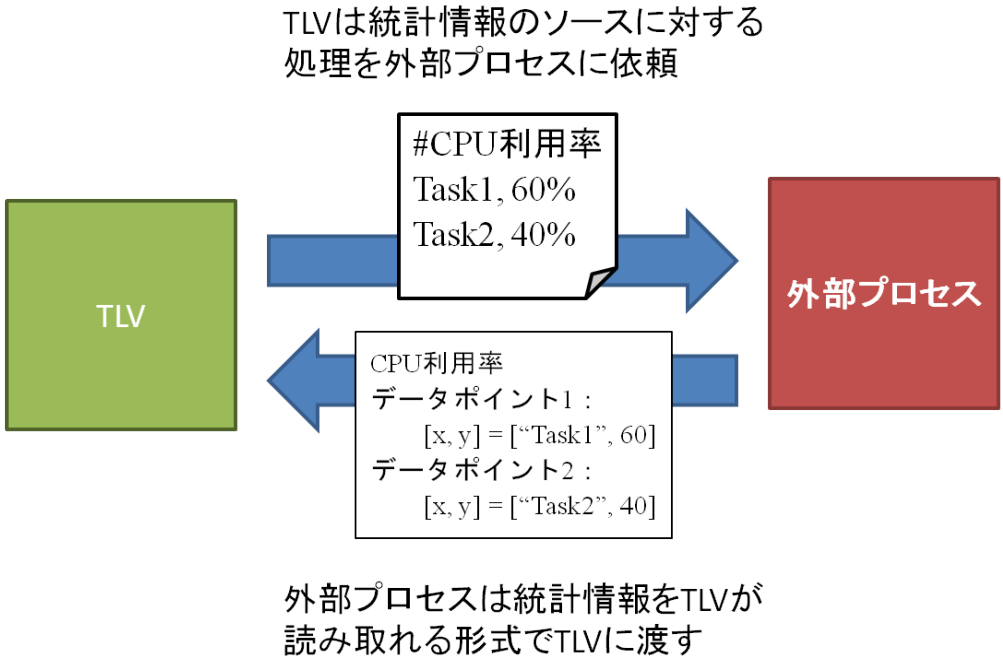

図 4.4　実現方法 2 のイメージ

計算が必要な統計情報も得られる．また，利用するアプリケーションさえ対応していれば，様々なファイルから統計情報が得られる．イメージを図 4.4 に示す．

実現方法 3：TLV に実装した簡単な解析ができるメソッドによる生成

外部スクリプトでは，開発するコストがどうしてもかかってしまう．そのため，TLV に標準形式トレースログを簡単な解析ができるメソッドを実装することで，外部スクリプトを利用せずに単純な統計情報が生成可能になる．イメージを図 4.5

に示す．

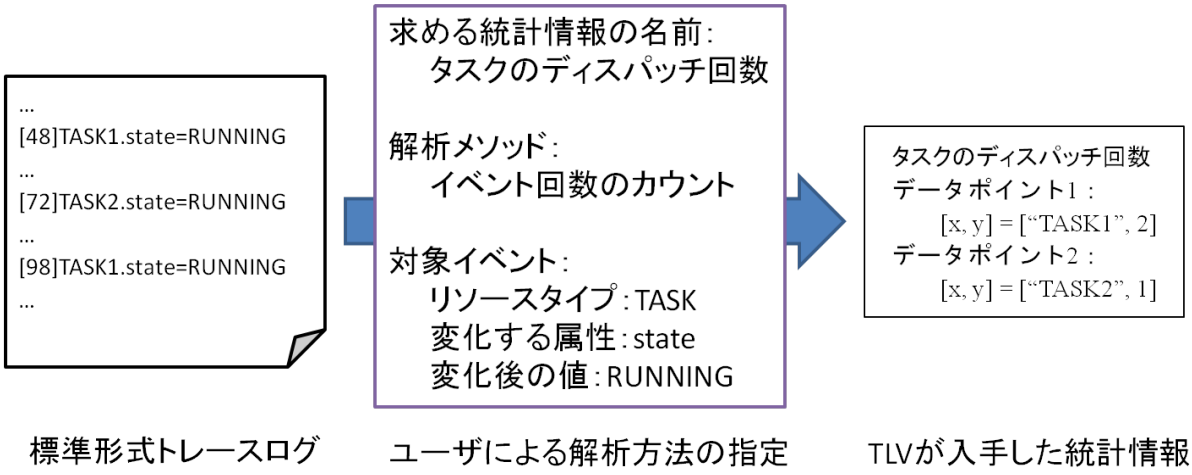

図 4.5　実現方法 3 のイメージ

実現方法 3 に関して，前述の説明だけでは理解しづらいので，詳細を述べる．

この手段をユーザに提供するに当たり，いくつかの簡単な解析ができるメソッドを TLV に用意する必要がある．TLV では，このメソッドを解析メソッドと呼ぶ．4.2.1 節に示す表 4.1 から，トレースログにおける統計情報の取り方について分析した結果，必要となるメソッドは，イベント回数のカウント，イベント間隔の測定，測定したイベント間隔のカウントの 3 つであることがわかった．イベント回数のカウントは，タスクの起動回数などのイベントが発生した回数を数えるものである．イベント間隔の測定は，始点となるイベントから終点となるイベントまでの時間を測定するものである．割込み禁止時間などの時間に関する統計情報を得る際に必要となる．測定したイベント間隔のカウントは，ヒストグラムの生成を想定しており，イベント間隔の測定を行って得られた時間が何回あるか数えるものである．例えば，タスクの実行時間分布などで必要となる．

統計情報の取り方について分析してわかったことは他にもある．各メソッドは，対象となるイベントを指定する必要があるが，得たい統計情報によっては，事前条件も必要となるケースがある．例えば，タスクの起動は，タスクが休止状態から実行可能状態になるときのことをいう．よって，タスクの起動回数を求める際の対象イベントは，タスク (TASK) の状態 (state) が実行可能状態 (RUNNABLE) に遷移するイベント (標準形式トレースログ形式で表現すると “TASK.state=RUNNABLE”) となる．しかし，このイベントは，タスクが実行状態である時など複数の状態において発生する可能性がある．よっ

て，タスクの起動回数を求めるには，対象イベントの他に「タスクが休止状態である」という前提条件が必要になる．こういったケースはどの統計情報にもありうるというわけではないので，オプション設定として用意する．

4.2.2.2 統計情報を得るトレースログ上の時刻区間とその指定方法

トレースログやその他のファイルから統計情報を取得する場合，情報が静的であるので，ソースが変わらない限り，情報は不変である．

しかし，統計情報を生成する場合，ソース内のデータの取り方によって，数値が変わってくる．トレースログにおける時系列情報を解析する場合，解析する時刻区間によって異なる．

解析する時刻区間を指定するタイミングとして，トレースログの解析作業前と解析作業中の 2 通りが考えられる．

まず，トレースログの解析作業前とは，TLV にトレースログなどの入力を与える前のことである．この状況で区間指定可能であることは，すなわち，指定する区間がすでに決まっている状態である．具体的には，区間としてトレースログ全体を指定したい場合などである．このような場合，統計情報を得ようとするたびに区間指定を行うのは非効率である．よって，外部ファイルで区間を指定しておき，TLV に変換・可視化処理と同じタイミングで統計情報を取得・生成できるようにすることで，効率化する．

次に，トレースログの解析作業中とは，TLV でトレースログを可視化している状態である．トレースログの解析作業中には，解析を容易にするために，ある一部区間のみの統計情報が必要になることがある．よって，可視化後も区間を入力して統計情報を得られるようにする．また，このケースでは，可視化されたトレースログを扱っているため，GUI を用いてグラフィカルに指定できるようにすることで，利用しやすくする．

トレースログ以外のファイルでも，ソース内のデータの取り方によって数値が変わってくるが，ソース内のデータや形式が一意に定まらないため，トレースログの解析と同様の機能は TLV で実現しないことにする．

4.2.3 統計情報とグラフ設定の保存方法

取得・生成した統計情報とグラフ設定の保存方法について 3 つの候補を挙げて検討した．図 4.6 に示す．以下にそれぞれの利点・欠点を述べ，検討した内容を示す．

候補 1 は，データポイントに依存しないグラフ設定と統計情報をそれぞれ別のファイルで保存する方法である．関連付用ファイルは，統計情報とグラフ設定のそれぞれを保存す

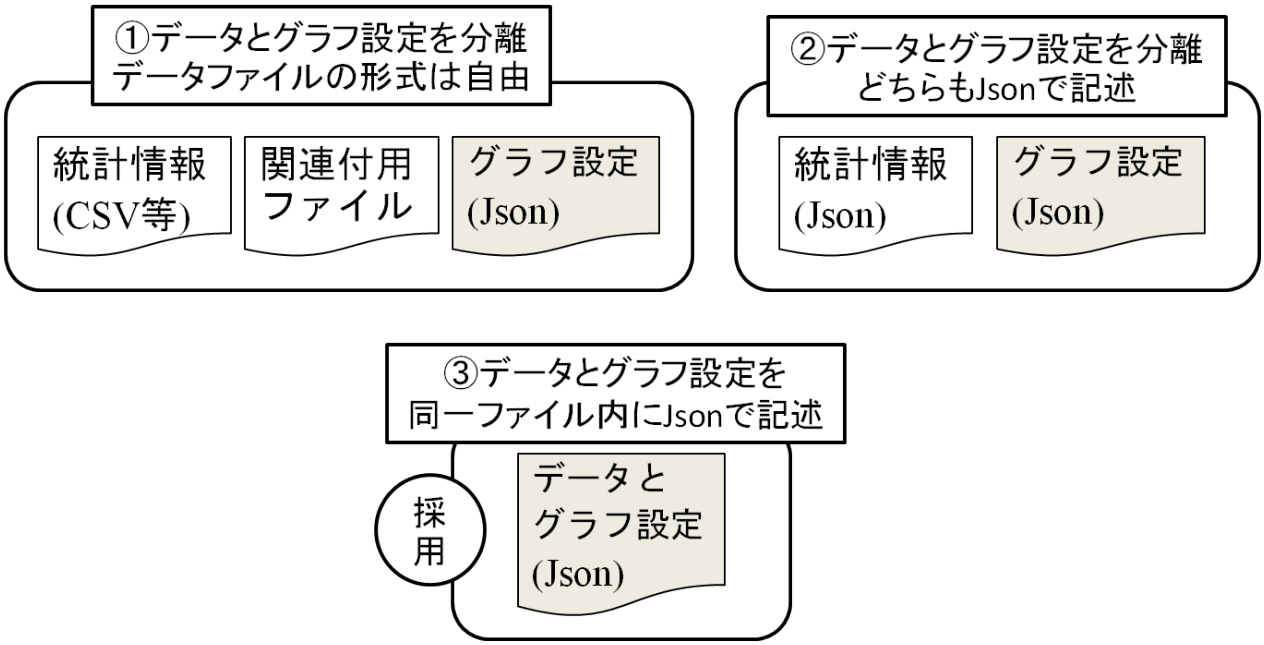

図 4.6　統計情報とグラフ設定の保存方法の候補

るファイルのパスまたは識別子を記述し，統計情報を保存するファイルからデータを取り出すときに使用する正規表現を記述する．データポイントに依存するグラフ設定は，統計情報を保存するファイルにデータポイントと関連付けて記述する．この方法についての利点と欠点を次に示す．

■候補 1 の利点

- 統計情報を保存するファイルのファイル形式を様々な方法で保存できるようにすることで，TLV 以外のアプリケーションでも統計情報を利用できるようになる．
- データポイントに依存しないグラフ設定を再利用できるようになり，グラフ設定を保存するファイルを 1 つ変更するだけで，その変更が関係する統計情報すべてに反映される．

■候補 1 の欠点

- ファイルを分散させることで，統計情報がどのようなグラフ設定で表示されるかが把握しづらい．
- ファイル移動に弱い．関連付用ファイルを更新しなければならない．

- 管理するファイルが候補 2 候補 3 より多いので保守性が低下する．
- 統計情報を保存するファイルを他のアプリケーションで使用する場合，統計情報とは無関係の情報であるデータポイントに依存するグラフ設定が処理の障害となる．
- TLV の使用環境により，グラフ設定を保存するファイルの内容が異なる場合があるので，統計情報を保存するファイルだけを異なる環境へ移すと，移動前の環境と同じグラフ設定でグラフ．
- 統計情報の種類によっては，目盛幅などのグラフ設定をログごとに設定したいケースがあるので，その対応が難しい．
- 正規表現は可読性が低く，難しい形式のファイルには対応しづらい．

候補 2 は，候補 1 同様にファイル分割を行うが，どちらも JSON で記述する方法である．候補 1 での関連付用ファイルの代わりに，統計情報を保存するファイルにグラフ設定を保存するファイルのパスまたは識別子を指定する．この方法についての利点と欠点を次に示す．

■候補 2 の利点

- データポイントに依存しないグラフ設定を再利用できるようになり，グラフ設定を保存するファイルを 1 つ変更するだけで，その変更が関係する統計情報すべてに反映される．
- 候補 1 の欠点であった保守性の懸念とデータポイントに依存するグラフ設定の問題が緩和される．

■候補 2 の欠点

- ファイルを分散させることで，統計情報がどのようなグラフ設定で表示されるかが把握しづらい．
- 統計情報を保存するファイルを他のアプリケーションで利用できない．
- TLV の使用環境により，グラフ設定を保存するファイルの内容が異なる場合があるため，異なる環境でのグラフ設定が保障されない．
- 統計情報の種類によっては，目盛幅などのグラフ設定をログごとに設定したいケースがあるので，その対応が難しい．

候補 3 は，統計情報もグラフ設定も 1 つのファイルに保存する方法である．この方法についての利点と欠点を次に示す．

■候補 3 の利点

- 統計情報がどのようにグラフ表示されるか把握しやすい.
- 1 つのファイルのみで完結するので，ファイルパスなどの整合性を気にしなくて良い.
- 統計情報毎にグラフ設定ができる.
- 実装が楽である.

■候補 3 の欠点

- 統計情報を保存するファイルを他のアプリケーションで利用できない.
- 同じグラフ設定が複数のファイルに出現して冗長する．修正する場合に時間がかかる.

候補 1 は，他のアプリケーションの使用を最低限にする目的からすると，欠点に比べ利点が弱いので，残りの 2 つに絞られる．候補 2 と候補 3 では，実装の観点からすると，どちらもすべて JSON で記述するので，候補 2 から候補 3 への変更，または，その逆の変更が容易に行えると考えられる．よって，実装時間を考慮して，まずは容易に実装でき，欠点も少ない候補 3 を採用した．この統計情報とグラフ設定を含んだファイルのことを統計情報ファイルと呼ぶことにする.

4.3 実装

本節では，4.2 節の設計に基づいた実装について述べる.

4.3.1 統計情報表示機能の主なプロセス

統計情報表示機能は，統計情報をグラフ表示するためのファイルを生成するプロセスと 6 種類のファイルによって実現される．図 4.7 に統計情報表示機能の主なプロセスを示す.

統計情報ファイルを生成するプロセスでは，統計情報生成ルールファイルに記述された統計情報生成ルールに従い，指定したソースから統計情報を取得・生成する．そして，得られた統計情報と，統計情報生成ルールに従ったグラフの設定を 1 つのファイルに書き込む，といった処理を行う．このプロセスでは，統計情報のソースとなる 3 つのファイル

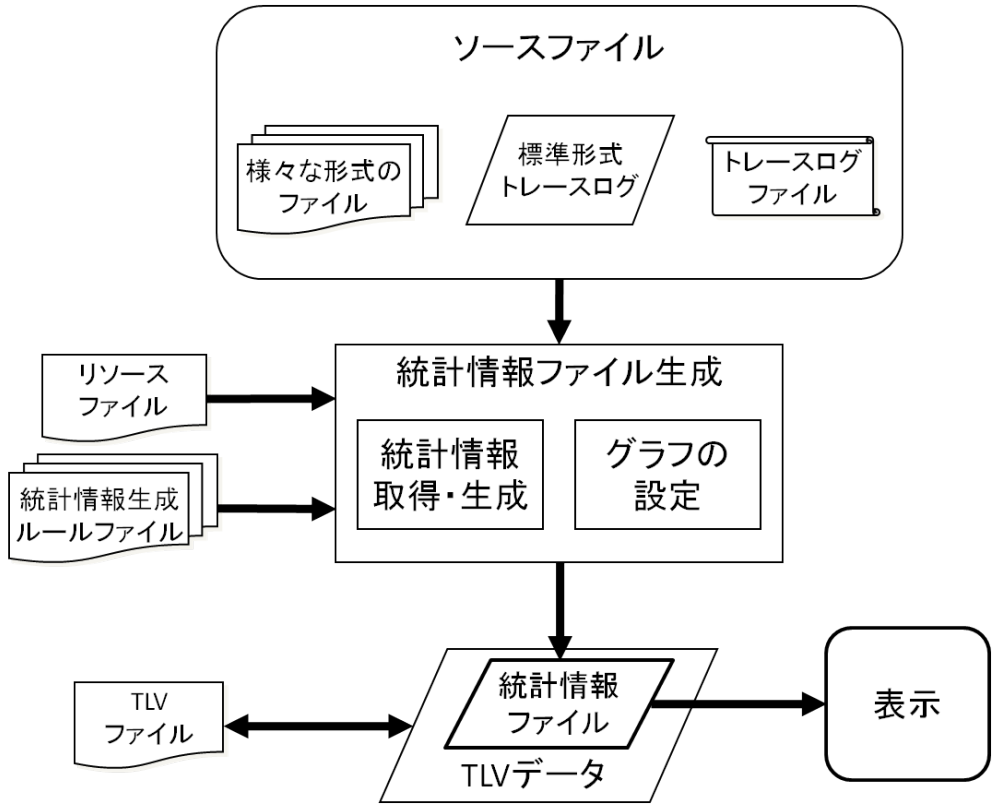

図 4.7　統計情報表示機能の処理プロセス

(変換元のトレースログ，標準形式トレースログ，トレースログ以外のファイル) のうちのどれか 1 つと，統計情報生成ルールファイル，統計情報の生成やグラフの設定に関係するリソースファイルが読み込まれる．

図 4.7 では，統計情報を取得・生成するプロセスとグラフ設定を行うプロセスを，統計情報ファイルを生成するプロセスのサブプロセスとして表現しているが，これは 2 つのプロセスがそれぞれ独立して動作するとは限らないことを示している．たとえば，統計情報を生成しつつ，各データを表す色を個別に設定するといったケースが該当する．

生成した統計情報ファイルは，統計情報ビューアと呼ばれる統計情報をグラフ表示するためのウィンドウで利用される．また，TLV データ内に格納することで，TLV ファイルとして外部ファイルに保存でき，同じトレースログを扱う際に再度，統計情報ファイルを生成しなくてもよくなる．

4.3.2 統計情報表示機能を実装した TLV の処理プロセス

必要な統計情報を得るためには，ユーザが TLV に統計情報生成ルールを入力する必要がある．そこでまず考えられる方法は，トレースログとリソースファイルを入力する方法と同様に，GUI で指定する方法である．しかしながら，ターゲットシステムをデバッグする際，トレースログを何度も取り直し，TLV で何度も変換・可視化処理をすることになるので，そのたびに統計情報生成ルールを入力することになる．そのため，この方法だけでは，入力作業が非効率になる．よって，まず，従来の TLV の処理プロセスで統計情報ファイル生成プロセスを実行する方法を実現する．

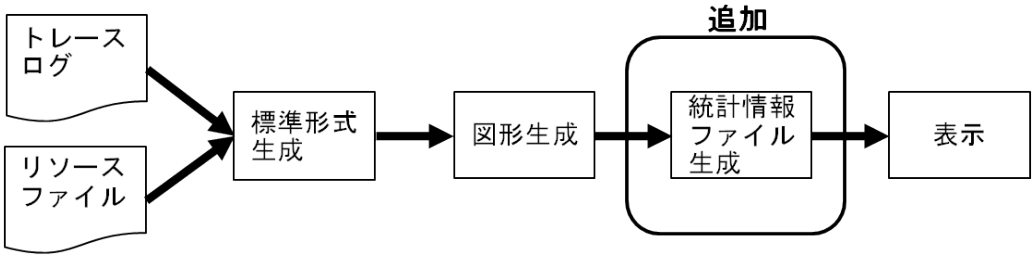

図 4.8　統計情報表示機能を実装した TLV の処理プロセス

従来の TLV の処理プロセスは，2.2 節で示した通り，トレースログとリソースファイルを読み込み，トレースログを標準形式トレースログへ変換し，図形データ生成するプロセスとなっている．変換と図形データ生成に使用する変換ルールと可視化ルールは，リソースファイルで定義されている．

統計情報表示機能を実装した TLV の処理プロセスを図 4.8 に示す．従来のプロセスにある図形データ生成プロセスの後に，統計情報表示機能の処理プロセスである統計情報ファイル生成プロセスを追加する形となる．入力する統計情報生成ルールをリソースファイルで定義することで，ユーザが直接入力するファイル数を増やさずに済む．また，図 4.9 の`"StatisticsGenerationRules"`のように，他のルールと同じ指定方法にすることで，統計情報表示機能に対する学習コストを軽減する効果もある．

4.3.3 統計情報ファイル

統計情報ファイルとは，グラフの設定と統計情報が形式的に記述されたファイルのことである．TLV では，統計情報ファイル生成プロセスで生成され，統計情報ビューアにグ

```
{
        "TimeScale" :"us",
        "TimeRadix" :10,
        "ConvertRules" :["asp"],
        "VisualizeRules" :["toppers","asp"],
        "ResourceHeaders":["asp"],

        "StatisticsGenerationRules":["task_dispatch_count"],

        "Resources": {...}
}
```

図 4.9　統計情報表示機能を利用するリソースファイルの例

ラフを表示するときに使用する．また，ユーザは，TLV ファイルを通して外部出力することにより，統計情報を数値データとして得ることができる．

　統計情報ファイルには，1 つの統計情報に関する統計情報ファイルが記述される．統計情報ファイルの例を図 4.10 に示す．

```
{
  "task_dispatch_count": {
    "Setting":{
      "Title":"タスクのディスパッチ回数",
      "AxisXTitle":"タスク",
      "AxisYTitle":"回数",
      "DefaultType":"Column",
      "MajorTickMarkInterval":10.0,
      "MinorTickMarkInterval":5.0,
      "MajorGridVisible":true,
      "MinorGridVisible":true
    },
    "Series": {
      "Points": [
        {
          "XLabel": "task 1",
          "XValue": 0,
          "YValue": 40
        },
        {
          "XLabel": "task 2",
          "XValue": 0,
          "YValue": 60
        }
      ]
    }
  }
}
```

図 4.10　統計情報ファイルの例

Setting

説明 グラフ設定の定義

値 オブジェクト．メンバの説明は以下の通りである．

Title

説明 グラフのタイトル．主に GUI 表示に用いられる．

値 文字列

省略時 統計情報生成ルール名が適用される．

AxisXTitle

説明 X 軸のタイトル．

値 文字列

省略時 設定されない．

AxisYTitle

説明 Y 軸のタイトル．

値 文字列

省略時 設定されない．

DefaultType

説明 最初に表示するグラフの種類．

値 文字列 (現在，"Pie"：円，"Column"：縦棒，"Bar"：横棒，"Line"：折れ線，"Histogram"：ヒストグラムのいずれか)

省略時 "Pie"

MajorTickMarkInterval

説明 主目盛の間隔．

値 数値

省略時 データに合わせて自動設定．

MinorTickMarkInterval

説明 補助目盛の間隔．

値 数値

省略時 データに合わせて自動設定．

MajorGridVisible

説明 主目盛のグリッド線を表示するかどうか．ここで true を指定された場合，グリッド線が表示される．

値 真偽値

省略時 false

MinorGridVisible

説明 補助目盛のグリッド線を表示するかどうか．ここで true を指定された場合，グリッド線が表示される．

値 真偽値

省略時 false

Series

説明 系列.

値 オブジェクト．唯一のメンバである Point は値としてオブジェクトの配列をとる．配列に格納されるオブジェクトはデータポイントを表す．データポイントを表すオブジェクトのメンバの説明は以下の通りである．

XLabel

説明 X 軸に並べるデータポイントのラベル.

値 文字列

省略時 設定されない.

YLabel

説明 データポイント上に表示するラベル．円グラフとヒストグラムで表示するときは反映されない.

値 文字列 (次のマクロが使用可能．"#VALX"：X 値，"#VALY"：Y 値，"#PERCENT"：系列全体に対する Y 値の割合，"#TOTAL"：系列の全ての Y 値の合計)

省略時 グラフに合わせて自動的に設定される．

XValue

説明 X 値.

値 数値

省略時 0

YValue

説明 Y 値.

値 数値

省略時 0

Color

説明 グラフにプロットしたデータポイントの色.

値 文字列 ("#AARRGGBB"というフォーマットに従う．A：アルファ値，R：赤，G：緑，B：青とし，16 進数で表現する)

省略時 グラフライブラリの初期値に従う．

4.3.4 統計情報の取得・生成

統計情報の取得・生成は，4.2.2.1 節にて，3 通りの手段を考えた．それらの実現方法を実現し，統計情報のソースとなるファイルや用途に合わせて選択できるようした．それらの実現方法を用いて統計情報を取得・生成して統計情報ファイルを生成する手段を生成モードと呼ぶ．生成モードは，次の 4 種類である．

- データ読み取りモード (4.2.2.1 節における実現方法 1)
- スクリプト拡張モード (4.2.2.1 節における実現方法 2)
- 基本解析モード (4.2.2.1 節における実現方法 3)
- 統計情報ファイル入力モード

これらの生成モードは，統計情報生成ルールで設定する．

生成モードでは，基本的に統計情報の取得・生成とデータポイントに依存するグラフ設定のみを行うが，スクリプト拡張モードと統計情報ファイル入力モードに関してはその限りではない．詳しくは，それぞれのモード説明で述べる．

4.3.4.1 データ読み取りモード

対象となるソースファイル

トレースログ，標準形式トレースログ，その他のファイル

データ読み取りモードとは，4.2.2.1 節における実現方法 1 を実現する生成モードである．正規表現のマッチングの際に名前付きグループ化構造体を用いて部分文字列をキャプチャし，X 値と Y 値を取得する．必要に応じて，グラフ上の各データの色を設定できるようにすることで，グラフを見やすくできるようにした．

正規表現は，複数定義できるようにした．こうすることで，同じ統計情報に対して異なる表現がされているケースに対応でき，また，マッチングする正規表現によって異なる色を設定できるようになる．色の設定に関する具体的なケースは，X 軸がリソースとなるときにリソース別に色を設定するケース，Y 値がある数値以上となるデータのみをほかの色に設定するケースなどがある．

4.3.4.2 スクリプト拡張モード

対象となるソースファイル

トレースログ，標準形式トレースログ，その他のファイル

スクリプト拡張モードとは，4.2.2.1 節における実現方法 2 を実現する生成モードである．

この生成モードの名前は，変換ルール，可視化ルールにおけるスクリプト拡張機能に由来する．名前以外にも，入出力に多少の差異がある程度で，それ以外はほぼそれらに合わせた実装にした．これは，すでにある機能に合わせることで，使用方法の習得を容易にするためである．また，開発の観点では保守性を高めることにつながる．

主にスクリプトを利用することを想定しているので，ここではそれに従って処理プロセスを説明する．まず，TLV は統計情報生成ルールで指定されたスクリプト処理系とスクリプト本体を外部プロセスとして起動する．次に，起動した外部プロセスの標準入力に対し，リソースファイルと対象となるソースファイルを出力する．外部プロセスは，標準入力に書き込まれた 2 つのファイルを読み込んで処理を行い，標準出力に統計情報ファイルを書き込む．そして，TLV が外部プロセスの標準出力に書き込まれた統計情報ファイルを読み込み，生成モードの処理を終える．

外部プロセスとして起動するアプリケーションに関しては，標準入力でリソースファイルと対象となるソースファイルを読み込み，標準出力に統計情報ファイルを書き込む，という定められた入出力仕様を満たしていればよい．

以上のように，この生成モードでは外部プロセスから取得・生成した統計情報を統計情報ファイルという形で受け取る．これは，取得・生成した統計情報とデータポイントに依存するグラフ設定を結び付けたデータを外部プロセスから受け取りやすくするほかに，データポイントに依存しないグラフ設定も外部プロセス内で設定したいケースに対応するためである．例えば，目盛間隔を微調整したいケースや，4.3.4.1 節で挙げた色に関するケースである．

4.3.4.3 基本解析モード

対象となるソースファイル

標準形式トレースログ

基本解析モードとは，4.2.2.1 節における実現方法 3 を実現する生成モードである．この生成モードは，ユーザが解析メソッドとそれに必要な情報を与えることで，統計情報を生成する．

この生成モードが対象とするソースファイルは標準形式トレースログのみである．これは，TLV が様々なトレースログを扱えるように，トレースログを標準形式トレースログという中間形式にするので，これを利用することで汎用的な解析メソッドを単純な形で提供できるからである．様々な形式に対応したものにしてしまうと，正規表現によるマッチングが必要になる．その結果，生成モードの指定が複雑な形となるうえ，統計情報生成ルールファイルに記述する際に可読性も低くなり，この生成モードの主旨とは異なるものになってしまう．

第 3 期 OJL では，測定したイベント間隔のカウントの実装を行っていない．実装の優先順位を検討したとき，実装が複雑になることが予想されたので，他の実装よりも優先順位を低くした．その結果，時間に余裕がなくなり，実装できなくなったためである．また，同様の理由でオプション設定の実装も行っていない．

4.3.4.4 統計情報ファイル入力モード

対象となるソースファイル

統計情報ファイル

統計情報ファイル入力モードとは，ユーザが何らかの方法で入手した統計情報ファイルを TLV に入力する生成モードである．この生成モードは，他の生成モードと違い，統計情報を取得・生成ではなく，統計情報ファイル生成プロセスの最終成果物となるファイルを取得するモードである．そのため，4.2.2.1 節における実現方法を実現するものではない．この生成モードの発想は，統計情報の比較を行うために，以前生成した統計情報ファイルを利用する手段が求められるのではないか，という分析がもとになっている．

グラフ設定は，この生成モードを指定した統計情報生成ルールの設定ではなく，入力する統計情報ファイルの設定を利用する．これは，入力する統計情報ファイルには，それに記録された統計情報をグラフとして表現する際の最適なグラフ設定がされているはずだからである．

4.3.5 統計情報生成ルールファイル

統計情報生成ルールファイルには，1 つ以上の統計情報生成ルールが記述される．図 4.11 に例を示す．統計情報生成ルールファイルは，1 つのオブジェクトで構成され，オブジェクトのメンバに統計情報毎の統計情報生成ルールを記述する．メンバ名に統計情報生成ルール名を記述し，値としてオブジェクトを与え，そのオブジェクトに統計情報生成

ルールの定義を記述する．統計情報生成ルールの定義は，大きく分けて統計情報の取得・生成方法とグラフ設定の 2 つを記述することで行う．それぞれについての説明は小節で述べる．

統計情報生成ルールファイルのフォーマットは，メンバ名を含め，変換・可視化ルールを参考にしている．こうすることで，利用方法をわかりやすくし，開発の面でも再利用が可能な部分を増やしている．

```
{
  "task_dispatch_count" : {
    "Setting":{
      "Title":"タスクのディスパッチ回数",
      "AxisXTitle":"タスク",
      "AxisYTitle":"回数",
      "DefaultType":"Column",
      "MajorTickMarkInterval":10.0,
      "MinorTickMarkInterval":5.0,
      "MajorGridVisible":true,
      "MinorGridVisible":true
    },
    "UseResorceColor":true,
    "Mode":"Basic",

    "RegexpRule":{
      "Target":"C:/data/asp/task_dispatch_count.csv",
      "(?<task>[^,]+),\s*(?<num>\d+),\s*(?<color>\d+)" : {
        "XLabel" : "${task}",
        "YValue" : "${num}",
        "Color" : "#${color}"
      }
    },

    "ScriptExtension":{
      "Target":"standard",
      "FileName":"C:/cygwin/bin/ruby.exe",
      "Arguments":"statisticsGenerationScript/dispatch_counter.rb"
    },

    "BasicRule":{
      "Method":"Count",
      "When":{
        "ResourceType":["Task"],
        "AttributeName":"state",
        "AttributeValue":"RUNNING"
      }
    },

    "InputRule":{
      "FileName":"C:/data/asp/asp_stats-task_dispatch_count.sta"
    }
}
```

図 4.11　統計情報生成ルールファイルの例

4.3.5.1 統計情報の取得・生成方法

統計情報の取得・生成方法の定義は，図 4.11 の 15 行目にある"Mode"以降のもので，生成モードと生成モードの動作の定義をすることで行う．生成モードは"Mode"で定義し，生成モードの動作の定義は図 4.11 の 17 行目以降に定義されたオブジェクトで行う．

生成モードの動作の定義は，"Mode"で定義しなかった生成モードに関する定義も記述しておけるようにした (省略も可能)．これにより，生成モード別に統計情報生成ルールを記述する必要がなくなり，グラフ設定の冗長を抑えることができる．また，1 つの統計情報生成ルール内に同じ統計情報を求める生成モードを集約することで，統計情報生成ルールの管理が容易になる．

次に，統計情報生成ルールのオブジェクトにおける統計情報の取得・生成方法の定義に関するメンバを説明する．

Mode
- **説明** 生成モードの定義.
- **値** 文字列 ("Regexp"：データ読み取りモード，"Script"：スクリプト拡張モード，"Basic"：基本解析モード，"Input"：統計情報ファイル入力モードのいずれか)
- **省略時** 省略不可能.

RegexpRule
- **説明** データ読み取りモードの動作定義.
- **値** オブジェクト．Target 以外のメンバは，メンバ名に適用する正規表現，値に正規表現がマッチしたときのデータポイントの定義をオブジェクトで行う．統計情報ファイルで説明したデータポイントのオブジェクトと同じであるので，そのオブジェクトのメンバの説明は省略し，Target の説明のみ次に示す.
- **省略時** "Mode"で"Regexp"と定義した場合，省略不可能.

  Target
  - **説明** 対象となるソースファイル.
  - **値** 文字列 ("standard"：標準形式トレースログ，"nonstandard"：変換前のトレースログ，”ファイルパス”：トレースログ以外のファイルのいずれか)
  - **省略時** 省略不可能.

ScriptExtension
- **説明** スクリプト拡張モードの動作定義.
- **値** オブジェクト．オブジェクトのメンバの説明は以下の通りである.
- **省略時** "Mode"で"Script"と定義した場合，省略不可能.

Target
説明 対象となるソースファイル.
値 文字列 ("standard"：標準形式トレースログ，"nonstandard"：変換前のトレースログ，”ファイルパス”：トレースログ以外のファイルのいずれか)
省略時 省略不可能.

FileName
説明 使用するスクリプト処理系またはアプリケーション.
値 文字列 (絶対パス，または，TLV 実行ディレクトリからの相対パス)
省略時 省略不可能.

Arguments
説明 使用するスクリプトまたはアプリケーションの引数.
値 文字列 (ファイルパスの場合，絶対パス，または，TLV 実行ディレクトリからの相対パス．"{0}"とした場合，TLV 内で一時ファイル名に置き換えられる．)
省略時 省略不可能．ただし，空文字列を定義することは可能.

Script
説明 一時ファイルに記述するスクリプト文．"FileName"で指定したスクリプト処理系に依存する．"Arguments"で"{0}"としたときのみ有効.
値 文字列
省略時 空文字列

BasicRule
説明 基本解析モードの動作定義.
値 オブジェクト．オブジェクトのメンバの説明は以下の通りである.
省略時 "Mode"で"Script"と定義した場合，省略不可能.

Method
説明 解析メソッド.
値 文字列 ("Count"：イベント回数のカウント，"Measure"：イベント間隔の測定のいずれか)
省略時 省略不可能.

When
説明 対象となるイベント．ある 1 つのイベントのみを対象としたいときに定義する．よって，From，To とは同時に定義できない.
値 オブジェクト．オブジェクトはイベントを表し，メンバはイベントを構成する要素でできている．オブジェクトのメンバの説明は以下の通りである.
省略時 From，To が定義されていない場合，省略不可能.

ResourceType
説明 対象となるリソースタイプ．定義されたリソースタイプに属するリソースを対象とする．
値 文字列の配列
省略時 ResourceNames が定義されていない場合，省略不可能．
ResourceNames
説明 対象となるリソースの名前．
値 文字列の配列
省略時 ResourceType が定義されていない場合，省略不可能．
AttributeName
説明 属性の名前．属性の変化に関するイベントを定義したいときに用いる．振る舞いに関するイベントの定義とは同時に定義できない．
値 文字列
省略時 振る舞いに関する定義をしていない場合，省略不可能．
AttributeValue
説明 属性の値．属性の変化に関するイベントを定義したいときに用いる．振る舞いに関するイベントの定義とは同時に定義できない．
値 文字列
省略時 振る舞いに関する定義をしていない場合，省略不可能．
BehaviorName
説明 振る舞いの名前．振る舞いに関するイベントを定義したいときに用いる．属性の変化に関するイベントの定義とは同時に定義できない．
値 文字列
省略時 属性の変化に関する定義をしていない場合，省略不可能．
BehaviorArg
説明 振る舞いの引数．振る舞いに関するイベントを定義したいときに用いる．属性の変化に関するイベントの定義とは同時に定義できない．
値 文字列
省略時 属性の変化に関する定義をしていない場合，省略不可能．

From
説明 始点となるイベント．To とペア．When とは同時に定義できない．
値 オブジェクト．オブジェクトはイベントを表し，メンバはイベントを構成する要素でできている．オブジェクトのメンバは When を同じなので説明は省略する．
省略時 When が定義されていない場合，省略不可能．省略不可能．

To

説明 終点となるイベント．From とペア．When とは同時に定義できない．

値 オブジェクト．オブジェクトはイベントを表し，メンバはイベントを構成する要素でできている．オブジェクトのメンバは When を同じなので説明は省略する．

省略時 When が定義されていない場合，省略不可能．省略不可能．

InputRule

説明 統計情報ファイル入力モードの動作定義．

値 オブジェクト．オブジェクトのメンバの説明は以下の通りである．

省略時 "Mode"で"Regexp"と定義した場合，省略不可能．

FileName

説明 入力する統計情報ファイル．Data と同時に定義した場合，こちらが優先される．

値 文字列 (絶対パス)

省略時 Data が定義されていない場合，省略不可能．

Data

説明 入力する統計情報ファイルの内容．FileName と同時に定義した場合，FileName が優先される．

値 文字列

省略時 FileName が定義されていない場合，省略不可能．

4.3.5.2 グラフの設定

グラフの設定は，統計情報生成ルールのオブジェクトのメンバにである"Setting"と"UseResorceColor"が該当する．"Setting"は，統計情報ファイルの"Setting"と同じなので説明を省略する．

UseResorceColor

説明 リソースファイルに定義されたリソースの名前がデータポイントの XLabel に設定されているとき，リソースファイルに定義されたリソースの Color を利用するかどうか．true を指定された場合，それを利用する．Color が定義されていないリソースの色は，グラフライブラリの初期値に従う．

値 真偽値

省略時 false

グラフの設定には，どの設定が反映されるかという優先順位を設けた．これは，生成モードによってグラフの設定がされる場合に対応するためである．図 4.12 に示す．

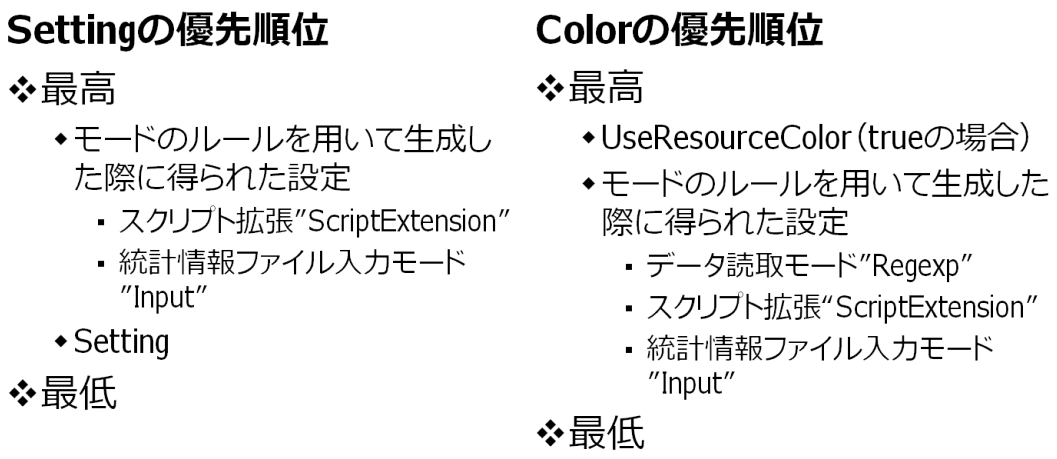

図 4.12　グラフ設定の優先順位

"Setting"は，生成モードによって統計情報ファイルが得られる場合があるので優先順位をつける必要がある．4.3.4.4 節でも述べたように，生成モードによって統計情報ファイルが得られる場合，得られた統計情報ファイルには統計情報生成ルールで定義した設定よりも最適なグラフ設定がされている可能性が高い．なぜなら，統計情報生成ルールに定義するグラフの設定は，変更回数の軽減をするために汎用性の高い設定が望まれるからである．よって，図のような優先順位とした．

生成モードによって"Color"が設定される場合があるので，"Color"にも優先順位をつける必要がある．優先順位は，"Setting"とは逆のような形になっている．これは，リソースの色が関係するからである．リソースファイルで定義されるリソースの色は，TLV の可視化表示部に表示する際に使用される場合がある．それを統計情報表示機能でも利用できるようにすることで，色によるリソースの識別を可能にし，グラフを読みやすくなるよう工夫した．その ON/OFF を実現しているのが"UseResorceColor"である．これは，"UseResorceColor"を"true"にするのは，ユーザがそのような効果を狙っている可能性が高いからである．よって，図のような優先順位とした．

4.3.6　統計情報をグラフ表示するための GUI

本節では，統計情報表示機能の実装に伴って追加された 2 種類のウィンドウについて説明する．該当するウィンドウは，統計情報をグラフ表示する統計情報ビューアとユーザが表示させる統計情報を選択する統計情報エクスプローラである．例を図 4.13 に示す．

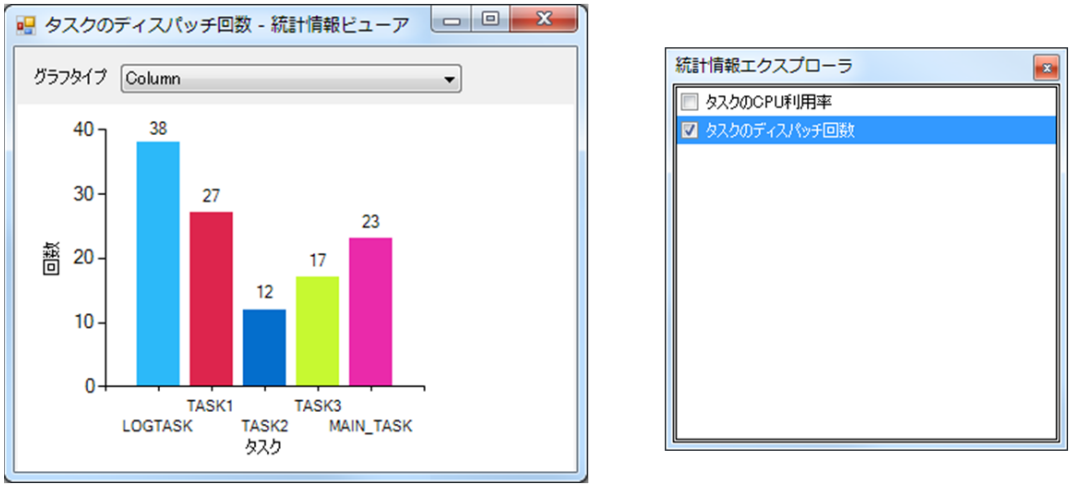

図 4.13　統計情報ビューア (左) と統計情報エクスプローラ (右)

4.3.6.1　統計情報ビューア

統計情報ビューアは，1 つのウィンドウに 1 つのグラフを表示する．異なる種類のグラフで表現したいときのために，グラフの種類が簡単に切り替えられる機能を実装している．また，ウィンドウのサイズを変えることで，グラフの拡大縮小が可能である．

このウィンドウは，他のウィンドウを違い，TLV のメインウィンドウにドッキングができないようになっている．他のウィンドウと同じように実装してしまうと，図 4.14 に示すウィンドウのリストにウィンドウ名が列挙されることになる．統計情報ファイルは，1 つのログに対して複数生成することが可能であるので，このリストが利用しづらくなる．

グラフを表示させる部分の実装について述べる．グラフを描画するためのライブラリを調査したが，.NET Framework 3.5 の標準ライブラリとして用意されておらず，ユーザ側にも開発側にも新たにライブラリをインストールする必要があった．そのライブラリは，.NET Framework 3.5 の追加ライブラリと.NET Framework 4 の 2 種類あり，それぞれの特徴を検討して適切な方を選択した．

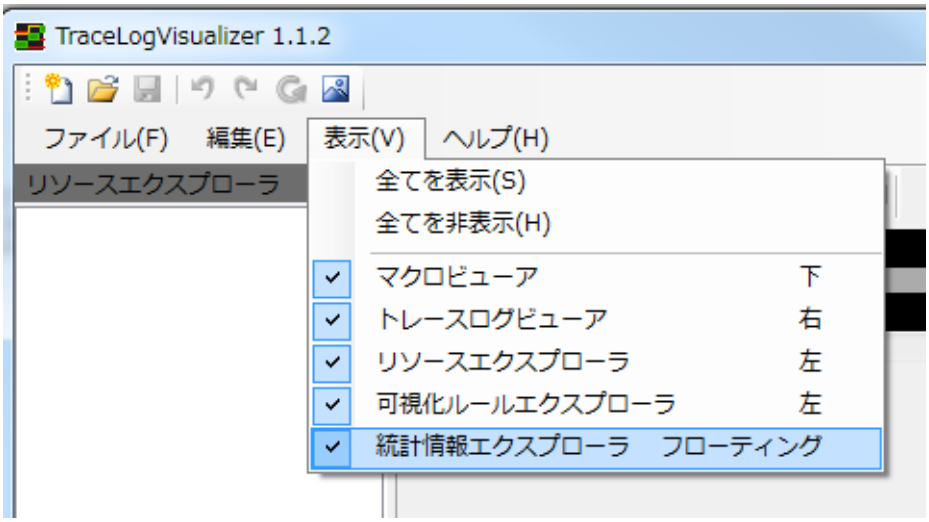

図 4.14　ドッキング可能なウィンドウがリストアップされているところ

.NET Framework 3.5 の追加ライブラリは，Microsoft Chart Controls for Microsoft .NET Framework 3.5 をユーザ側も開発側もインストールする必要がある．これは，.NET Framework のバージョンアップによるコード改変や不具合調査などを行わずに済む．しかしながら，このライブラリを利用するケースは TLV 以外にあまり考えられず，TLV のためだけにインストールすることになりかねない．

一方，.NET Framework 4 では，グラフを描画するライブラリが標準ライブラリとして用意されている．よって，TLV 以外で不要となりうるライブラリをインストールする必要がない．また，現在の TLV が Microsoft Parallel Extentions to .NET Framework 3.5 という Community Technology Preview，つまりテスト版を利用しているが，これについても標準ライブラリとして用意されている．.NET Framework 3.5 と.NET Framework 4 では互換性が完全でないので，不具合調査や開発環境の再構築を行う必要があるが，前述した利点に加え，開発が進むにつれていずれは.NET Framework 4 へ移行する必要がでてくると考え，.NET Framework 4 で実装することにした．

4.3.6.2　統計情報エクスプローラ

統計情報エクスプローラは，表示させる統計情報を選択するためのウィンドウである．これを使用することで，見たい統計情報に関する統計情報ビューアのみを表示させることが可能になる．

その他の役割として，コンピュータとユーザにかかる負荷の軽減がある．トレースロ

グの可視化処理の終了後，全ての統計情報ビューアを同時に立ち上げてしまうと，コンピュータに負荷がかかり，TLV の応答性が低下する．そのうえ，ユーザにとって不要な統計情報ビューアまで立ち上がる可能性がある．よって，最初は統計情報ビューアのインスタンス化せず，必要になった時に初めてインスタンス化して表示することで，これらの問題も解決できる．また，インスタンス化したものは，プーリングすることで，その後のウィンドウ開閉を円滑に行えるようにした．

4.4　統計情報表示機能の使用事例

本節では，TOPPERS/ASP カーネル [9] のトレースログにおける統計情報表示機能の利用例を 2 通り示し，統計情報表示機能の有用性について述べる．

例で使用する TOPPERS/ASP カーネルのトレースログは，TLV のパッケージ内に含まれるサンプルである asp_short.log である．このログのリソースファイル asp_short.res には，タスクとして，LOGTASK，TASK1，TASK2，TASK3，MAIN_TASK が定義されている．このリソースファイルに対して，例に示す 2 つの統計情報生成ルールを指定する．トレースログを可視化し，タスクに関してのみ表示したものを図 4.15 に示す．

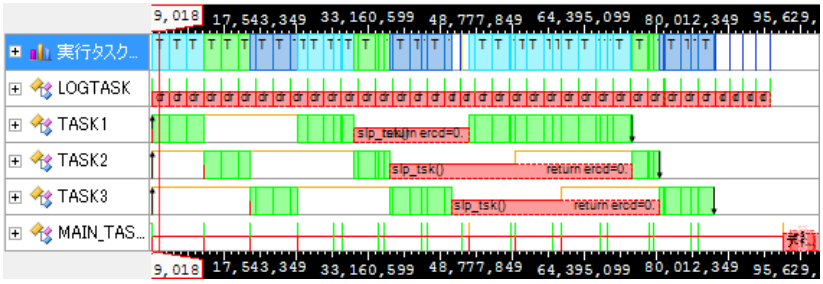

図 4.15　例に示す TOPPERS/ASP カーネルのトレースログ asp_short.log の可視化

4.4.1　基本解析モードを用いた各タスクのディスパッチ回数の生成

TOPPERS/ASP カーネルにおける各タスクのディスパッチ回数は，タスクが実行状態 (RUNNING) になるイベントを数えることで求めることができる．標準形式トレースログで表すと`"Task.state=RUNNING"`であり，`"Task"`の部分に各タスク名が入ったログを数えることで統計情報を生成できる．よって，基本解析モードを利用する場合，統計情報生成ルールファイルは 4.3.5 節で示した図 4.11 のようになる．複数の生成モードに対

するルールが記述されていても，"Mode"で"Basic"を指定することで，基本解析モードを利用できる．このファイルに定義した統計情報生成ルール"task_dispatch_count"を用いて統計情報を生成し，グラフ表示したものを図 4.16 に示す．グラフの色と図 4.15 の実行タスクのラインにある図形の色を比較すると，同じような値になっていることがわかる．これが"UseResourceColor"を true にしたときの動作である．図 4.15 の方が薄いのは，色のアルファ値が別の値に設定されているためである．

この統計情報をこの機能を利用せず求める場合，図 4.15 の各タスクのラインに表示されている緑色の長方形が立ち上がる箇所を手作業で数えたり，トレースログに対して各タスク毎に grep を行ったりする必要がある．さらに，統計情報をグラフ表示するためには，入手した統計情報をグラフ作成ツールへ入力する必要がある．

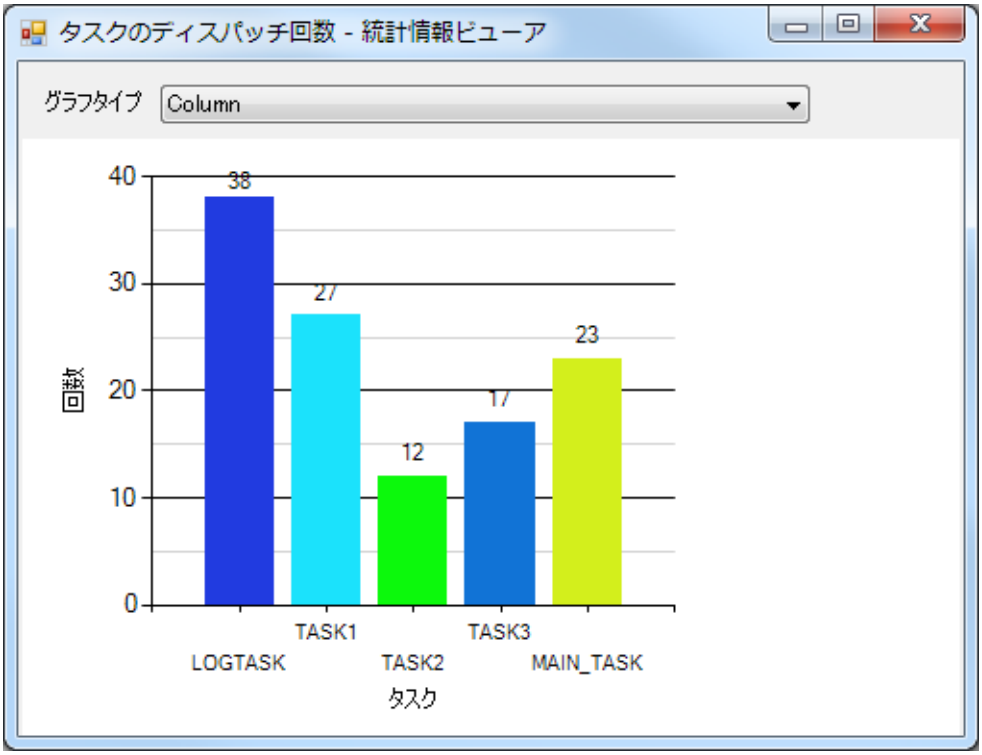

図 4.16　asp_short.log における各タスクのディスパッチ回数のグラフ表示

4.4.2　スクリプト拡張モードを用いた各タスクの CPU 利用率の生成

TOPPERS/ASP カーネルにおける各タスクの CPU 利用率は，タスクが実行状態である時間を求め，その時間をトレースログの記録時間で割ることで求めることができる．実

行状態である時間は，タスクの状態が実行状態に遷移してから，他の状態に遷移するまでの時間で求めることができ，CPU 利用率を出すには，その合計を求める必要がある．また，トレースログの記録時間は，トレースログの最後のログの時間から最初のログの時間を引くことで求めることができる．

以上のように，様々な値を導出し，それらを演算することは，基本解析モードではサポートしない．このような複雑な計算を要する統計情報には，スクリプト拡張モードを利用する．各タスクの CPU 利用率を生成して統計情報ファイルとして出力する Ruby スクリプト cpu_utilization.rb を用意し，統計情報生成ルールファイルに定義したものを図 4.17 に示す．このスクリプトは，汎用性を高くするため，標準形式トレースログを処理する．

```
{
  "cpu_utilization" : {
    "Setting":{
      "Title":"タスクの CPU 利用率",
      "AxisXTitle":"タスク",
      "AxisYTitle":"割合 [%]",
      "DefaultType":"Pie"
    },
    "UseResourceColor":true,
    "Mode":"Script",
    "ScriptExtension":{
      "Target":"standard",
      "FileName":"c:/cygwin/bin/ruby",
      "Arguments":"F:/TLV/statisticsGenerationScript/cpu_utilization.rb"
    }
  }
}
```

図 4.17　cpu_utilization.rb を利用した統計情報ファイルの生成

このファイルに定義した統計情報生成ルール"cpu_utilization"を用いて統計情報を生成し，グラフ表示したものを図 4.18 に示す．値が上下 2 段になっているが，上の値が求めた統計情報であり，下の値が円グラフ化したときの割合を表している．このように，グラフによって，適切な表示方法をする．また，各タスクの CPU 利用率を円グラフで表したのにもかかわらず，上下の値が一致しないのは，全体のアイドル時間の割合を含めていないためである．その上，LOGTASK と MAIN_TASK の値が小さすぎて重なってしまっている．このような，グラフの種類を変更して表示したい場合，統計情報ビューアの

上部にあるグラフタイプとラベリングされたコンボボックスを操作することで，グラフの種類を変更できる．横棒グラフに変更した例を図 4.19 に示す．

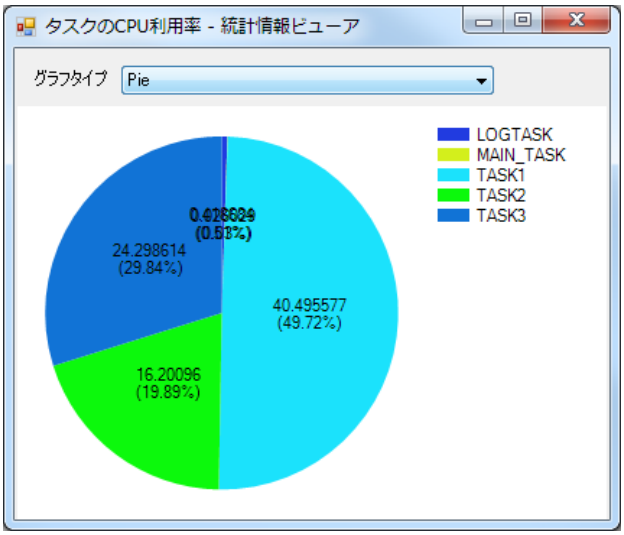

図 4.18　asp_short.log における各タスクの CPU 利用率のグラフ表示 (初期状態)

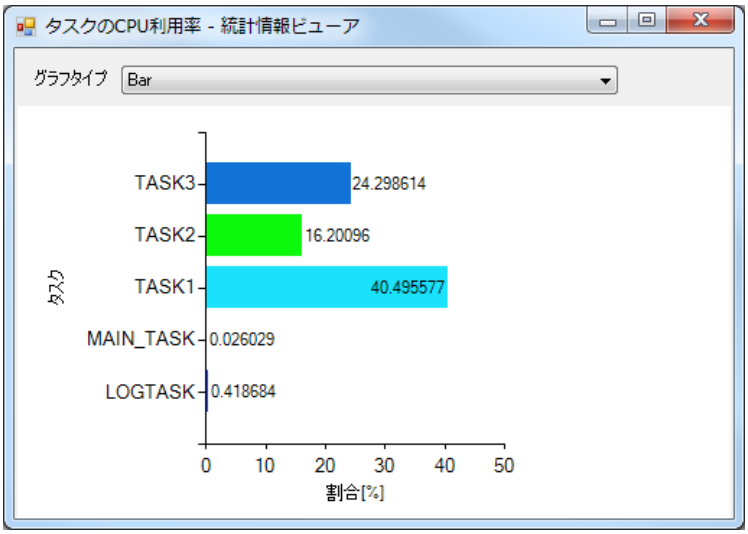

図 4.19　asp_short.log における各タスクの CPU 利用率のグラフ表示 (グラフタイプ変更後)

第 5 章

トレースログパーサのリファクタリング

5.1 概要

5.1.1 実施理由

標準形式トレースログのパースにおいて，図 5.1 のような複雑な正規表現を計 8 回適用している．これは，Object などの要素 1 つにつき 1 行の正規表現で取得するためである．こうすることで，Time や Value 等の有無に関わらず[*1]パースできるようにしている．しかし，このような複雑な正規表現は可読性が低いので，リファクタリングを実施する．

リファクタリング後の TLV は，標準形式トレースログのパースを専用のパーサに任せる．パーサを構成するコードを，2.3 節 に示す EBNF のように記述することで，開発者が構文を直感的に理解できるようになる．コード例を図 5.2 に示す．

また，重複したコードおよび生成規則変更時の変更箇所も減らすことができるため，保守性も向上する．例えば，Time を囲む記号`"["`,`"]"`を`"<"`,`">"`にするとき，従来の正規表現を用いた方法では，8 行すべての正規表現を変更する必要があるが，リファクタリング後は 1 箇所で済むようになる．生成規則を追加する場合でも，従来の手法では 8 行の正規表現の中から変更箇所を探さねばならないが，リファクタリング後は，EBNF のように記述されているため，変更箇所が明確であるので見落としを減らせる．しかしながら，このような構文の変更は稀であるため，一番のメリットは構文を直感的に理解できるように

[*1] 可視化ルールファイルには，Time がない，Value がない，AttributeName がない等といった不完全な標準形式トレースログが記述されている．

なることである．

```
m = Regex.Match(_log,
      @"^(\[[^\]]+\])?(?<objectType>[^\[\]\(\)\.]+)\([^\)]+\)(\.[^\s]+)?$");

if (m.Success)
       ObjectType = m.Groups["objectType"].Value;
HasObjectType = m.Success;
```

図 5.1　リファクタリング前のコード例

```
var object_ =
       ObjectTypeName().Char('(').AttributeCondition().Char(')')
       .OR().
       ObjectName();
```

図 5.2　リファクタリング後に表現したいコード例

5.1.2　対象

リファクタリング対象は，TraceLog クラス，特にそのコンストラクタである．この TraceLog クラスは，標準形式トレースログを構成するトークンを保持し，標準形式トレースログを用いた処理を容易にする．

現在，標準形式トレースログのパースに正規表現を用いている箇所をパーサを用いるように，TraceLog コンストラクタを変更する．

5.1.3 変更内容

5.1.3.1 クラス構成

Regex クラスと Match クラスを用いた正規表現によるパースを廃止し，標準形式トレースログ用のパーサを作成する．そして，TraceLog クラスは，そのパーサへパース処理を委譲する．これにより，現在はパースの実装が TraceLog クラスに直書きされているためパースの実装方法を変更するのが困難であるのが，TraceLog クラスとパースの実装を分離することでパーサの交換が容易になり，パーサの実装方法変更が容易になる．

今回の変更に伴う影響範囲は，TraceLog クラス以外に存在しない．リファクタリング前後のクラス図を図 5.3,5.4 に示す．

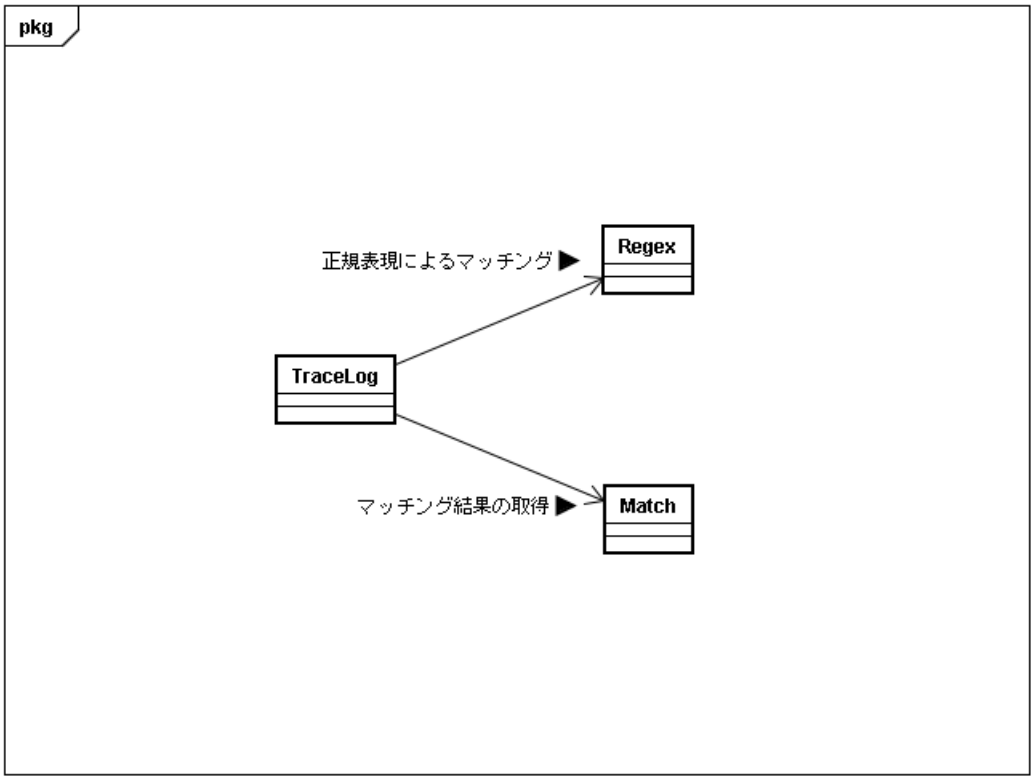

図 5.3　リファクタリング前のクラス構成

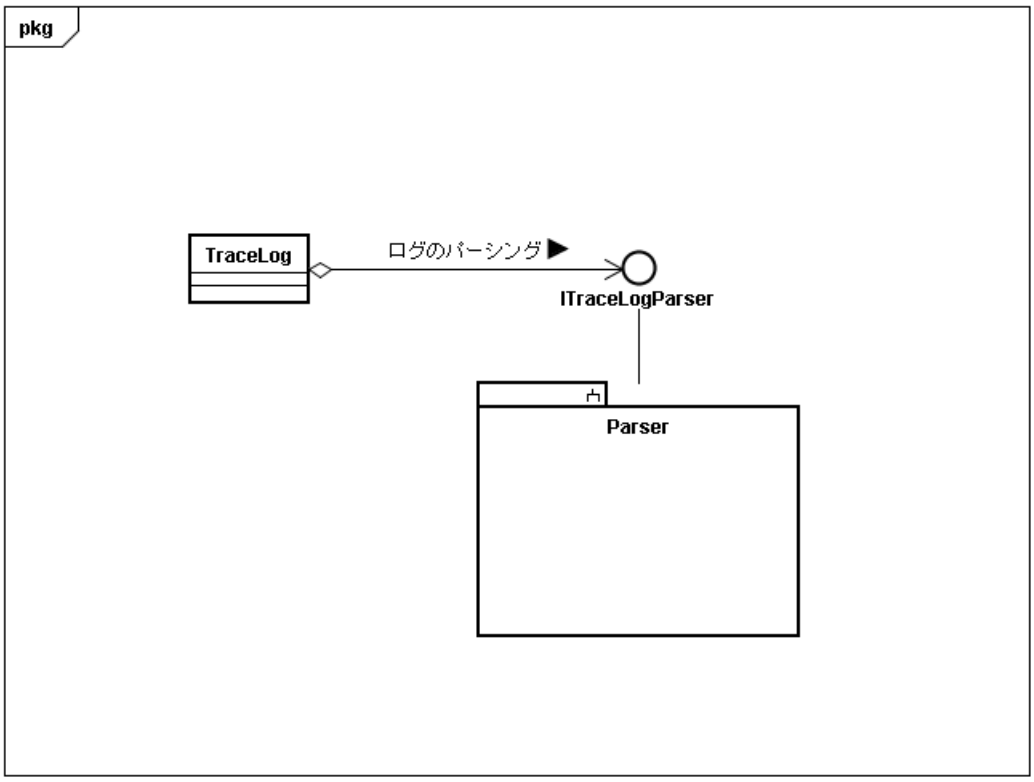

図 5.4　リファクタリング後のクラス構成

5.1.3.2　パーサの構成

パーサの実装は，コードで EBNF を表現し，直感的に理解できるようにする．

例として，Haskell のパーサコンビネータライブラリ Parsec を参考にしたパーサ [11] が挙げられる．このコードを使用するか，自作など他の実装方法を用いるかは，デバッグの容易さ，実装のしやすさ，可読性，実行速度などの要素により決定する．

5.2 詳細仕様

本節は，フェーズ 5 で行ったリファクタリングにより変更・追加された箇所，および，リファクタリング後のシーケンスを説明し，今回採用しなかった手法との比較や今回の実装の採用理由を記述する．

5.2.1 コード設計

ここでは，リファクタリングの目標であった，EBNF を表現して直感的に理解できるコードの紹介と，小節にて実装に用いた手法・概念および主要メソッドの説明をする．

まず，EBNF を表現したコードの例を図 5.5 に示す．

```
1 var line = Char('[').Time().Char(']').Event();
2 ...
3 var object_ =
4         ObjectTypeName().Char('(').AttributeCondition().Char(')')
5         .OR().
6         ObjectName();
7 ...
8 var typeName = Many1(() => AnyCharOtherThan('(', ')', '.'));
```

図 5.5　コードで EBNF を表現したコード例

これは，次の EBNF をパースすることを示している．

```
1 TraceLogLine = "[", Time, "]", Event;
2
3 Resource = ResourceTypeName, "(", AttributeCondition, ")"
4         | ResourceName;
5
6 ResourceTypeName = /[0-9a-Z_]+/;
```

図 5.6　図 5.5 が表す実際の EBNF

このように，図 5.5 は EBNF を表現しており，正規表現によるパーシングより可読性が向上している．図 5.5 は，実際には図 5.7 のような解析メソッド内に記述され，それを組み合わせることで図 5.5 を実現している．

```
1 public ITraceLogParser ObjectTypeName()
2 {
3         Begin();
4
5         var typeName = Many1(() => AnyCharOtherThan('(', ')', '.'));
6
7         typeName.ObjectTypeValue = Result();
8         return (ITraceLogParser)typeName.End();
9 }
```

図 5.7　解析メソッド例

5.2.1.1　利用した概念・手法

本節では，利用した概念と手法について述べる．

再帰下降パーサ

今回，実装したパーサは，スタックを利用した再帰下降パーサであり，バックトラッキングを用いている．これは，EBNF のようにトップダウンに記述されているものを表現するのに適しているが，バックトラッキングによる処理効率低下が懸念される．

パーサ・コンビネータ

パーサ・コンビネータとは，パーサとパーサを組み合わせることのできるコンビネータを指す．具体的には，Many メソッド，Many1 メソッド，OR メソッドが挙げられる．これにより，EBNF の表現や LL(k) 文法のパーシングを可能にしている．しかしながら，LL(k) 文法を扱うため生成規則から左再帰性の除去が必要である．

NullObject パターン

インタフェースを実装しているが，何もしないクラスを用いるデザインパターンである．多態性を利用したもので，Null かどうかを判別するコードを排除でき，処理の流れが明確になる．今回，これを用いることで EBNF のような記述を実現している．解析メソッドは，パーサクラスもしくはパーサクラス用の NullObject を返すことで次の解析メソッドを実行するかを決定する．これにより，if 分をはさまずにメソッドチェーンのみで生成規則を表現できるようになる．他に，意図しない Null 参照による例外発生などの，Null による問題を防止できるというメリットも

ある．

5.2.1.2 基本メソッドの説明

本節では，パーサを記述する上で基本となるメソッドについて述べる．

Begin メソッド

解析メソッドの処理で最初に実行されるメソッドである．

現在は，スタックへのパース結果およびポインタ[*2]の保存領域確保を行っている．

End メソッド

解析メソッドの処理で最後に実行されるメソッドである．基本的に，パースを実行，つまり，生成規則を表すメソッドチェーンを実行して得られた戻り値であるパーサオブジェクトまたは NullObject から呼ばれる．現在は，スタックのポップ，直前のスタックにあるパース結果との結合，NullObject のステータス初期化を行っている．パーサクラスか NullObject クラスかで動作が変わる．

Result メソッド

パースを実行して得られた文字列を返すメソッドである．

OR メソッド

EBNF の記号"|"を表すメソッドである．生成規則を表すメソッドチェーンで使用する．このメソッドより前の生成規則に当てはまらなかった場合，このメソッド内でバックトラッキングして，次の生成規則の適用を試みる．

Many メソッド

EBNF の記号"*"を表すメソッドである．生成規則を表すメソッドチェーンで使用する．引数に与えられた解析メソッドを 0 回以上適用させる．

Many1 メソッド

EBNF の記号"+"を表すメソッドである．生成規則を表すメソッドチェーンで使用する．引数に与えられた解析メソッドを 1 回以上適用させる．

5.2.2 クラス設計

リファクタリング後のクラス図を図 5.8 に示す．TraceLog クラス以外のクラス群が，5.1.3.1 節の図 5.4 にあるサブシステムに相当する．

[*2] パーシング中の文字列の，解析の現在位置 (文字) を示すもの．

パーサは，基本的な機能を備えたクラスとパース対象に特化したクラスにわけることで，基本的な機能を備えたクラスの再利用ができるようになっている．そのため，それに該当する Parser クラスおよび NullObjectForParser クラスを継承する派生クラスを定義し，ITraceLogParser クラスのような特化したインタフェースを定義してそれぞれの派生クラスが実装することで，様々なパーサが作成可能である．

次に，図 5.8 の各クラスについて説明する．

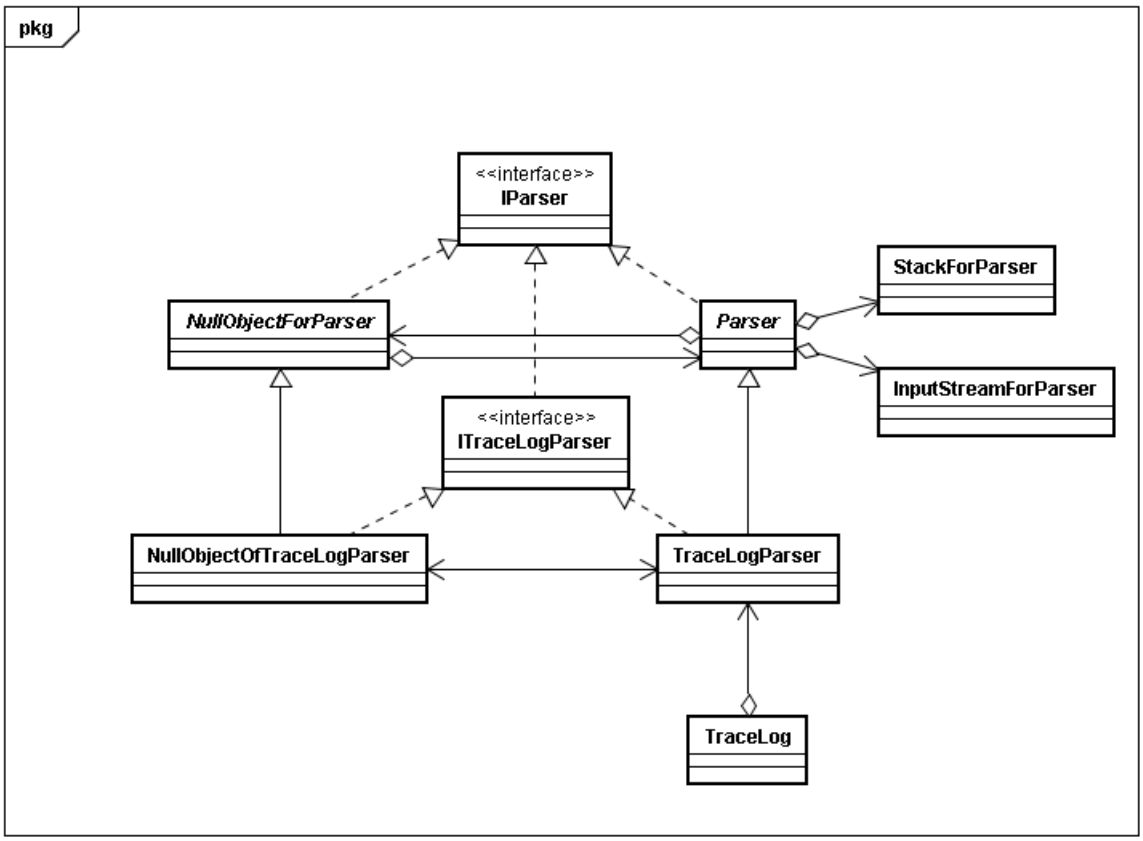

図 5.8　リファクタリング後の詳細なクラス構成

5.2.2.1　既設クラス

本節では，リファクタリング以前から存在するクラスについて述べる．

TraceLog クラス

標準形式トレースログをコード上で扱うためのクラスである．今回，そのコンストラクタをリファクタリングし，可読性の向上を行った．また，新たに

TraceLogParser オブジェクトを保持する静的フィールドを追加し，パーサの生成を一回で済むようにした．

5.2.2.2 新設クラス

本節では，リファクタリングにより新しく追加したクラスについて述べる．

Parser クラス

パーサを実現するための基本的な機能を備えたクラスである．これを継承し，解析メソッドを記述することで専用のパーサを作成する．5.2.1.2 に示したメソッドの他に，アルファベットや数字の解析メソッドも有する．これらのメソッドは，単純に継承しただけでは戻り値の型が合わず，使用できない．よって，派生クラスに，処理を Parser クラスに委譲し，その結果を使用するインタフェース型にキャストして return する同名 (もしくは，それとわかる名前) のメソッドを用意して使用する．

StackForParser クラス

Parser クラスが利用する専用スタックである．パーサが今までのパース結果やポインタを退避させるのに用いる．できるだけ効率よく管理するために用意した．

InputStreamForParser クラス

パース対象を格納し，管理するクラスである．ポインタの制御やポインタの示す文字の提供などを行う．.NET Framework の標準ライブラリである StringReader クラスは，Read メソッドで文字を消費してしまって復元が難しいため，専用のクラスを用意した．

NullObjectForParser クラス

Parser クラスの NullObject を実現するための基本的な機能を備えたクラスである．これを継承し，解析メソッドを含むインタフェースを実装することで，専用のパーサ用 NullObject を作成する．

TraceLogParser クラス

標準形式トレースログ用のパーサクラスである．標準形式トレースログをパースするための解析メソッドや結果を取得するためのプロパティがある．

NullObjectOfTraceLogParser クラス

標準形式トレースログ用のパーサクラス用の NullObject クラスである．メソッドは基本的に何もしないが，効率やアルゴリズム上の理由などで処理を行う場合がある．ただし，NullObject としては異例なため，そのような場合は保守性が低下す

る恐れがある．

IParser クラス

Parser クラスと NullObjectForParser クラスを結ぶインタフェースである．これにより，メソッドチェーンによる EBNF のような表現を実現している．

ITraceLogParser クラス

TraceLogParser クラスと NullObjectOfTraceLogParser クラスを結ぶインタフェースである．これにより，メソッドチェーンによる EBNF のような表現を実現している．

5.2.3 シーケンス

例として，"[.]Task.state=="RUNNING""という誤ったログをパースするシーケンスの一部を図 5.9 に示す．解析メソッドの適用に成功すれば次の解析メソッドを試み，失敗すれば以降の解析メソッドは NullObject のものを呼ぶことでパーサの解析メソッドを適用しないようになっている．

また，失敗した状態で OR メソッドが呼ばれるとバックトラックし，OR メソッド以降の解析メソッドにより再度パースが行われる．その他の状態で呼ばれた時の OR メソッドの処理を図 5.10 に示す．3 つ目の場合は，例えば hoge().OR().fuga().OR().piyo() という生成規則があった場合，hoge() が成功した時の fuga() と piyo() の間にある OR() の挙動を示している．

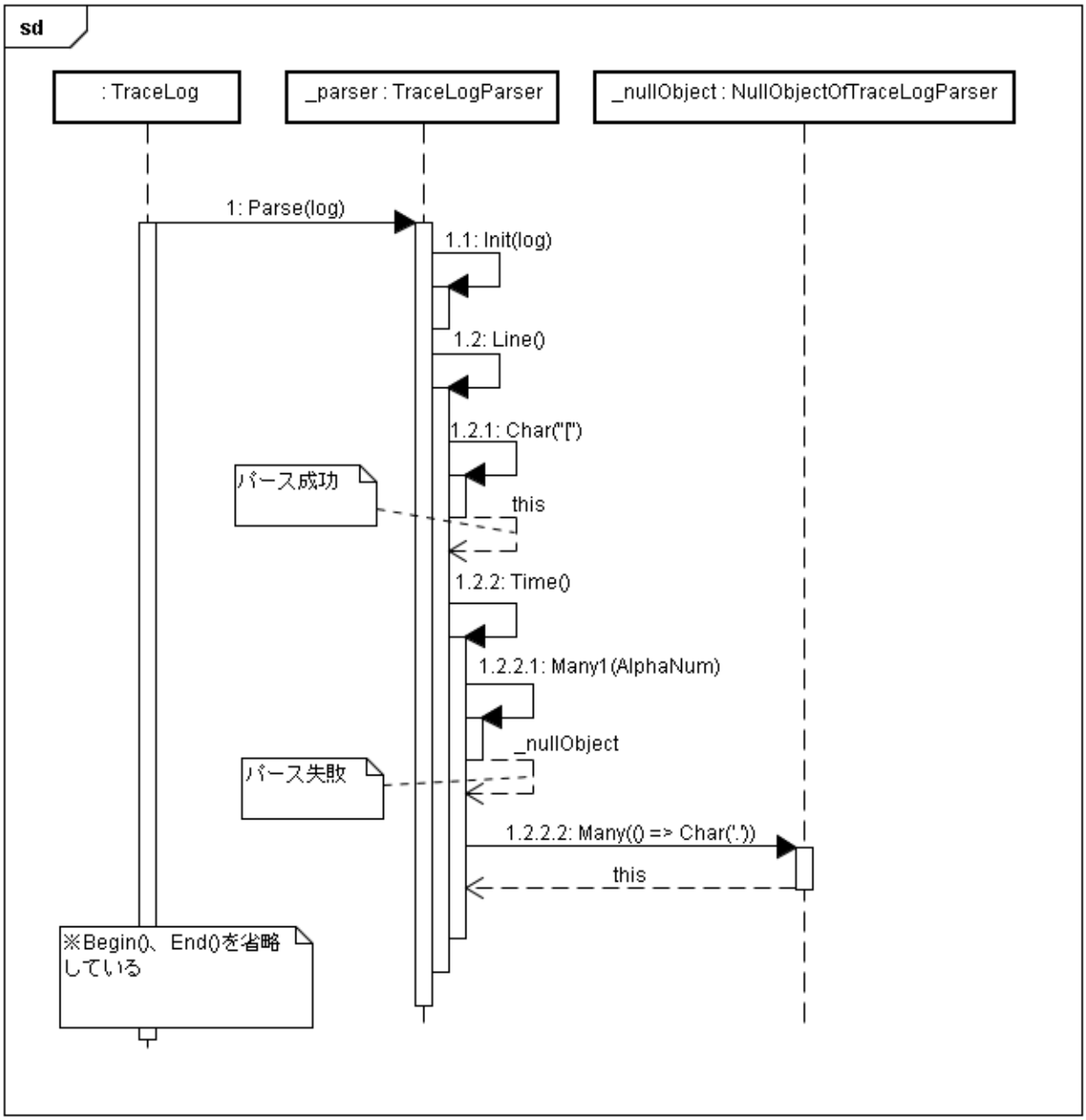

図 5.9　パーシング時のシーケンスの一部

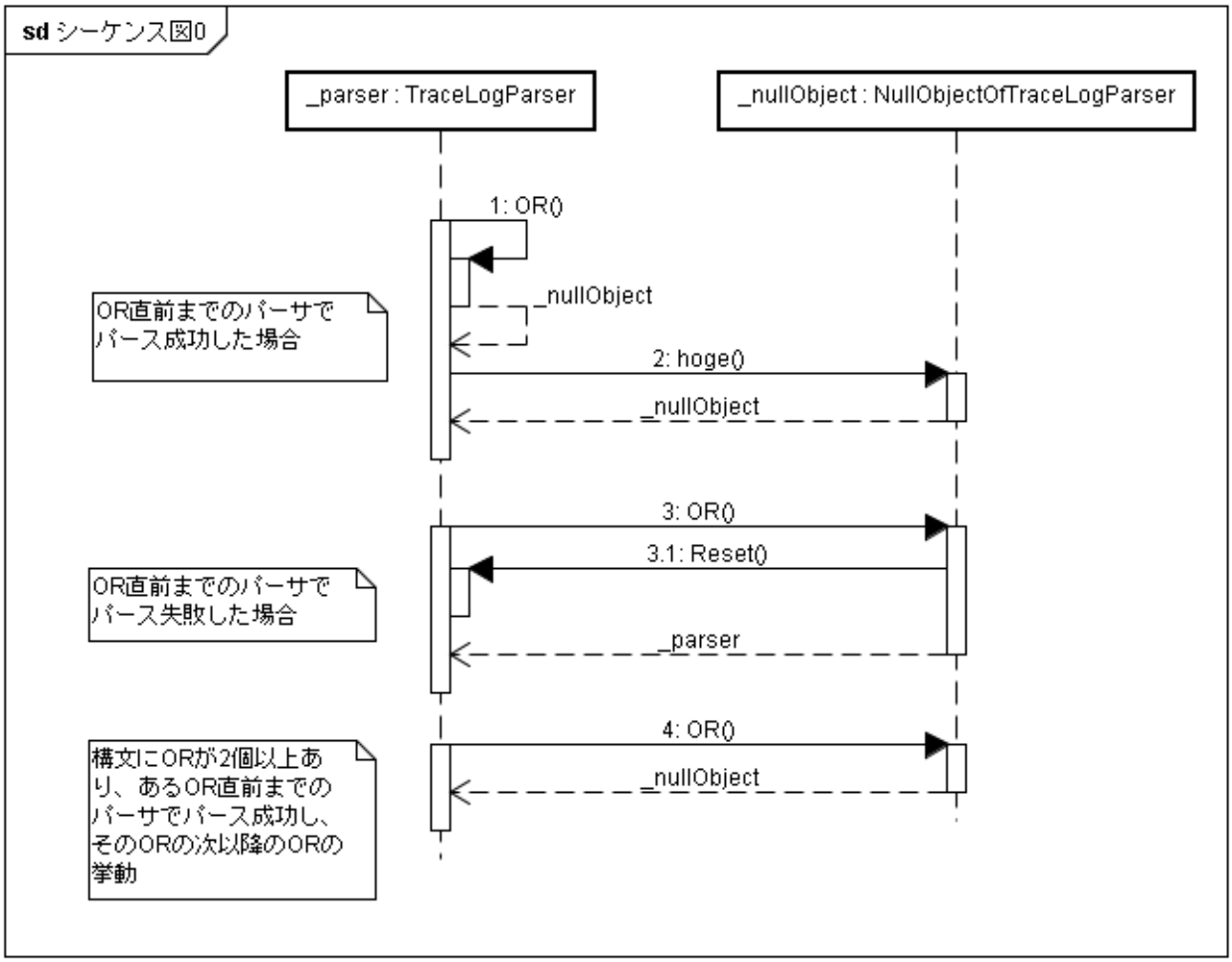

図 5.10　各状態での OR メソッドの処理

5.3 評価

本節は，5.1.3.2 で紹介した [11] を用いたパーサ (以降，C#版 Parsec)，5.2 節で紹介したパーサ (以降，自作パーサ) を比較し，自作パーサを選択した理由を記述する．また，それら二つのパーサおよび従来のパース手法で性能比較を示す．

5.3.1 可読性

まず，C#版 Parsec のコードを図 5.12 に示す．このパーサは，Haskell の Parsec を模したコードになっており，図**??**のように置き換えることができる．すなわち，図 5.12 では，`in` より右の解析メソッドを上から順に見たものが生成規則を表している．しかしながら，この記述は，コード上でクエリ式を記述できる LINQ を利用しているため，初見では理解しにくいということが現在の開発メンバとのレビューで出てきた．また，右の解析メソッドを上から順に見たものが生成規則を表していることがわかったとしても，`select` が何をするのか，何が記述されているのかが理解しにくい．そのため，Haskell 経験者であればよいとは思われるが，そうでない者にとっては可読性が従来のものより低くなってしまう可能性がある．

自作パーサは，生成規則を表す解析メソッドのチェーンが EBNF に近いため，自作パーサの方が可読性が高いというレビュー結果が得られた．

```
1  OBject = (from otn in ObjectTypeName
2            from u1 in Char('(')
3                   from ac in AttributeCondition
4                   from u2 in Char(')')
5                   select Base(otn).Append(u1).Append(ac).Append(u2).ToString())
6                  .OR(
7                  (from on in ObjectName
8                   select Base(on).ToString()));
9
10 // Base メソッドは，引数に基づいて StringBuilder クラスを新規作成して返すメソッド
11 // StringBuilder クラスに関しては，.NET Framework ライブラリを参照
```

図 5.11　C#版 Parsec のコード例

```
1  OBject = do{ otn <- objectTypeName
2             ; u1 <- char '('
3             ; ac <- attributeCondition
4             ; u2 <- char ')'
5             ; return  $ otn ++ u1 ++ ac ++ u2
6             }
7           <|>
8           do{ on <- objectName
9             ; return $ on
10            }
```

図 5.12　図 5.12 を Haskell に置き換えたコード例

5.3.2　保守性

C#版 Parsec のクラス構造は，機能ごとによく分割・整理されており，また，文字列以外にも対応できるように設計されているため汎用性が高い．この C#版 Parsec およびパーサを変更するには，Parsec の実装に関する知識が必要になる．それに伴い，Haskell の知識が必要になる場合がある．よって，チューニングまで行おうとすると，事前知識がなければ学習コストがかかってしまう．

自作パーサは，文字列のパーシングに特化したものであり，文字列化できないもののパーシングはほぼ不可能に近い．しかしながら，TLV で用いる場合，文字列以外は現状では考えられないため，問題にならない．変更するコストに関しては，自作パーサでも，NullObject との連携や解析メソッドのチェーンの理解に学習コストがかかる．

次に，生成規則の変更について考える．これについては，両方とも EBNF を表現したものであるため，基本的な手順や特徴は同じである．記述変更すべき箇所は，目的の生成規則を表現している箇所をみればよいので，従来の手法より変更箇所を発見しやすく，変更箇所の個数も削減できる．変更には，生成規則の追加，削除がある[*3]ので，この順序で見ていく．まず追加時には，次の手順を踏む．

1. 新規の生成規則にしたがって解析メソッドを作成する
2. 新規の解析メソッドを既存の生成規則に挿入する
3. C#版 Parsec の場合：新規の解析メソッドを挿入した既存の生成規則の select 句を書き換える

[*3] 入れ替えは，追加・削除を組み合わせたものと考えられるため除外

削除時には，次の手順を踏む．

1. 既存の生成規則から目的の解析メソッドを削除する
2. C#版 Parsec の場合：解析メソッドを削除した生成規則の select 句を書き換える

以上のように，C#版 Parsec の場合は，自作パーサより多く記述しなければならない場合があるので，変更箇所数でいえば，自作パーサの方が優れる場合がある．

5.3.3 デバッグの容易性

C#版 Parsec は，かなりの割合をデリゲートが占める．そのため，デバッガによるステップ実行では，どのデリゲートが適用されているのかがわかりづらく，結果，処理の流れが把握しづらいためデバッグが難しい．また，流れ自体も実装と関わっているため，Parsec などの知識が必要となる．

自作パーサは，コードを見るよりもデバッガによるステップ実行を行った方がデバッグが容易である．特に，取得したいパース結果の格納時の，図**??**のコードの※ 1，2 である．

```
1  var object_ =
2          ObjectTypeName().Char('(').AttributeCondition().Char(')')
3          .OR().
4          ObjectName();
5
6  object_.ObjectValue = Result(); // ※ 1
7
8  // HasObjectTypeValue は，ObjectTypeName() が真でも,ほかで失敗すれば偽である．
9  object_.HasObjectTypeValue = false; // ※ 2
```

例えば※ 1，2 は，`object_`が何を指すのかを推測して記述する必要がある．デバッガによるステップ実行を行うことで，`object_`が明確となり，処理が間違っていないか確認できる．また，C#版 Parsec のようにデリゲートを多用しておらず，使用していてもシンプルなシーケンスであるため，どのメソッドまたはデリゲートを適用させているのかが C#版 Parsec よりシーケンスを追いやすい．

5.3.4 性能比較

5.3.4.1 処理速度

各手法間で処理速度の違いが出るかを調査した．TLV は，ユーザから高速化の要求を頂いているため，処理速度がリファクタリングにより著しく低下することは望ましくなく，逆にリファクタリングにより高速化することが望ましい．

以下の方法で行った．

用いるもの　各パーサを実装した TLV(計 3 つ)，ストップウォッチ
計測開始　「新規作成ウィザード」の「OK」ボタンを押した状態から離したとき
計測終了　「初期化中」という窓に切り替わったとき
計測順序　自作パーサ 3 回→ C#版 Parsec3 回→従来 3 回
入力 (.log)　6958 行 (TLV 付属のサンプルファイル fmp_long.log)
入力 (.res)　リソース数 28 (TLV 付属のサンプルファイル fmp_long.res)
備考　1 回の測定につき TLV の起動・終了を行う

実施環境は次の通り．

OS　Windows 7 Professional　x64
CPU　Core2Duo E8400(3.0GHz)
メモリ　DDR2-800　1GBx2+2GBx2

結果を表 5.1 に示す．

表 5.1　処理速度の測定結果

	1 回目	2 回目	3 回目
自作パーサ	2 分 26 秒	2 分 21 秒	2 分 22 秒
C#版 Parsec	2 分 40 秒	2 分 38 秒	2 分 38 秒
従来手法	2 分 25 秒	2 分 24 秒	2 分 24 秒

順位は自作パーサ，従来手法，C#版 Parsec の順であった．自作パーサは，処理時間が C#版 Parsec より約 11%，従来手法より約 2% 短く，処理速度面でも優れていることがわかる．

自作パーサが他より高速である理由は，次のものが考えられる．

- 文字列をそのまま扱うのではなく，文字列を文字に分解してコストが低い文字処理を行う
- 自作パーサ専用のスタックおよび入力ストリームのため無駄が少ない
- 文字列に特化することで，高速な文字列処理が行える StringBuilder クラスを採用できた

C#版 Parsec がこのような結果になった理由は，次のものが考えられる．

- クラスメソッド呼び出しよりコストのかかるデリゲートを多用している
- 汎用性を高めるために，要素の結合を配列の結合[*4]で行っている (Parsec の実装に似せている)
- new を多用し，多くのインスタンス化を行っている

5.3.4.2 解析能力

解析能力は, 3 種類全ての手法で変わらない．自作パーサおよび C#版 Parsec は LL(K) 文法を解析でき，従来の手法は.NET Framework の正規表現が通常の正規表現より柔軟かつ強力にできている [12] ため，標準形式トレースログのパーシングに問題はないからである．しかしながら，従来の手法で使用されていた正規表現では，括弧のネストを正確にパースするのに，現状以上に複雑な正規表現となってしまう[*5]．たとえば，次のコードは<,>のネストを正しくパースする正規表現 [13] だが，これを現状の正規表現上の括弧のネストが起こりうる箇所に挿入することになる．

```
1 ^[^<>]*(((?'Open'<)[^<>]*)+((?'Close-Open'>)[^<>]*)+)*(?(Open)(?!))
```

したがって，従来の手法では現状より可読性が低くなってしまう．よって，括弧のネストに対応するという将来性を考慮しても，自作パーサの有用性は高いと言える．

[*4] 配列同士の結合は文字列の結合よりコストが高い

[*5] 従来の手法では，括弧のネストのパースを実現していない．また，他の処理でも括弧のネストに対応しきれていない．

第 6 章

第 3 期 OJL の実績

6.1 フェーズ毎の実績

本節では，第 3 期 OJL で実施したフェーズ 5,6,7 について，筆者の実績を述べる．

6.1.1 フェーズ 5(2009 年度後期)

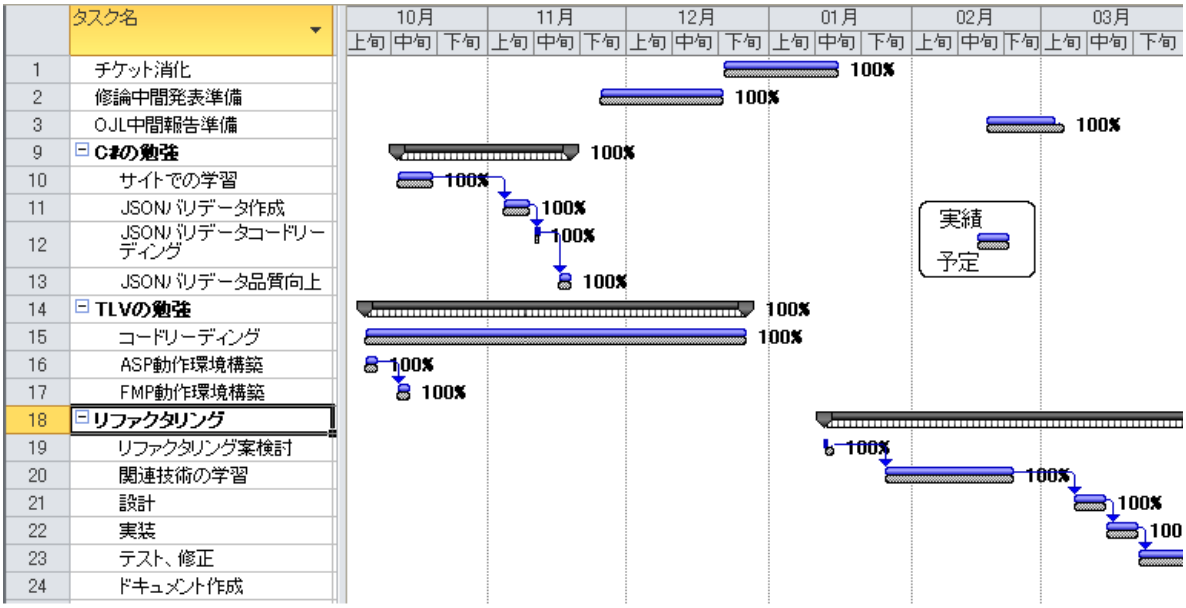

図 6.1　フェーズ 5 のスケジュール

6.1.1.1 実施内容

- 開発環境の構築
- JSON バリデータの作成を通した C#と JSON の学習
- 次の作業を通した TLV の学習
  - TLV のコードリーディング
  - TLV のマニュアル類を読んで不明点や誤りなどのチェック
- 不具合に関するチケットの担当
- リファクタリングの検討と実施

6.1.1.2 スケジュール

図 6.1 のように，第 3 期 OJL は，2009 年 10 月から開始した．まず，TLV の開発環境を整えた後，11 月末まで TLV とその開発言語である C#の学習を行った．TLV と C#についておおよその知識がついたところで，不具合に関するチケットを担当した．

1 月中旬に，OJL2 期生によるリファクタリング案のレビューに参加した後，リファクタリングを担当することになった．2 月末まで関連技術である Parsec と，それをライブラリとしてもつプログラミング言語である Haskell の学習を行った．技術を理解した後，その技術を参考にリファクタリングを行った．

特に目立ったスケジュールの遅延はなかった．

6.1.1.3 成果

5 章で述べたリファクタリングを行った．

また，JSON として正しく記述されているかチェックする JSON バリデータを作成した．学習の他に，変換ルールファイルと可視化ルールファイル用の JSON バリデータを作成する目的もあったが，それについては別の OJL3 期生が担当した．

6.1.2 フェーズ 6(2010 年度前期)

6.1.2.1 実施内容

- リファクタリング後のテスト・修正とドキュメント作成
- 統計情報表示機能の実現可能性調査と要件定義
- リソース属性表示機能の実現可能性調査と要件定義

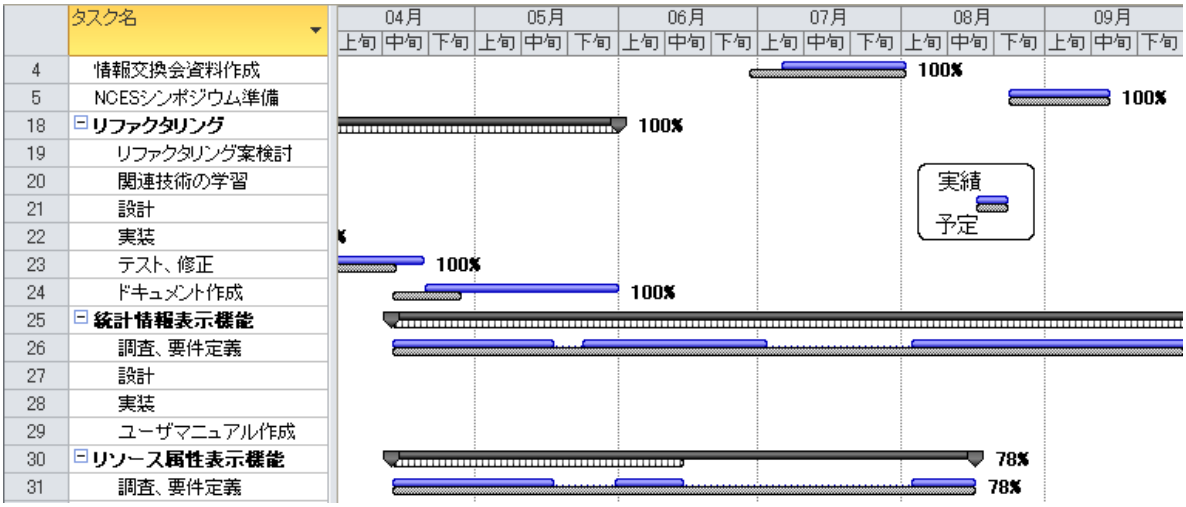

図 6.2　フェーズ 6 のスケジュール

6.1.2.2　スケジュール

図 6.2 のように，全体的にスケジュールの遅延が目立った．

リファクタリングのテストと修正での遅延は，予定を立てるときに，比較のために C#版 Parsec を実装した TLV を用意することを想定していなかったことにある．また，テスト方針を固めるのに時間がかかってしまったのも原因である．

ドキュメントの作成での遅延で考えられる原因を次に示す．

- ドキュメントの作成方針が作成途中で変更になったこと
- 筆者のドキュメント作成能力が低いこと
- 統計情報表示機能とリソース属性表示機能の実現可能性調査と同時進行していたこと

リソース属性表示機能とは，リソースのある時刻におけるセマフォなどの属性の値を表形式で表示する機能である．この機能もユーザの要望として挙がっている．統計情報表示機能と同様に，可視化表示部とは異なるウィンドウに情報を表示するものを想定していたので，実現可能性調査を統計情報表示機能と同時進行した．

調査開始当初，TLV プロトタイプにこの機能が実装されていたので，その名残があると予想した．そのため，実装が早く終わるだろうと判断し，統計情報表示機能より早く実装する予定であった．しかし，TLV にはコード以外の設計資料が乏しく，フェーズ 6 の時点では TLV の設計を理解しきれていなかったのもあり，調査が難航した．そのため，

より要求が強く，調査により実現可能性が見えてきていた統計情報表示機能の実現を優先させた．

実現可能性調査と要件定義に 6 か月という非常に多くの時間がかかってしまった．これは，要求分析の手順や方法が適切ではなかったこと，適切な図を用いていないなどの理由で考えがうまく相手に伝わらなかったこと，想定より多くの要件定義をする必要があったことが原因だと考える．

6.1.2.3 成果

リファクタリング報告書を作成した．

6.1.3 フェーズ 7(2010 年度後期)

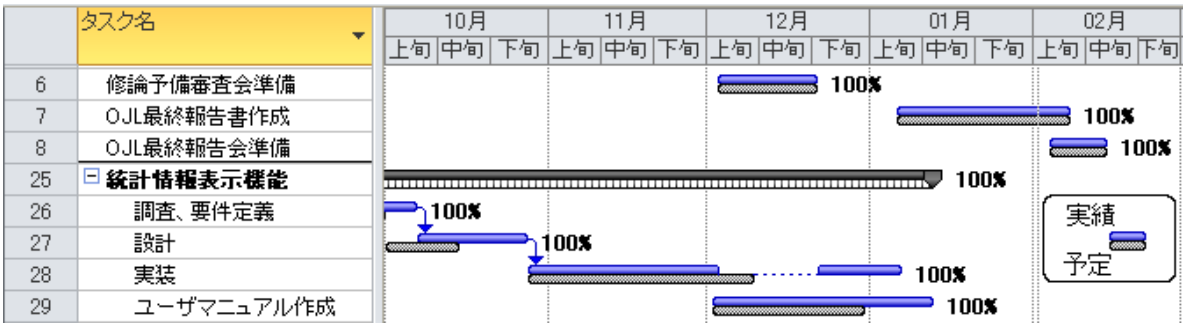

図 6.3　フェーズ 7 のスケジュール

6.1.3.1 実施内容

- 統計情報表示機能の実装とユーザマニュアル作成
- .NET Framework 4 への対応
- TLV ユーザ 1 名による統計情報表示機能のレビュー

6.1.3.2 スケジュール

図 6.3 のように，10 月から統計情報表示機能の設計，実装に着手した．予定では，12 月の上旬で実装がひと段落するはずだったが，.NET Framework 4 へのバージョンアップによる影響調査・対策に時間を取られたり，デバッグ時に見つかる不具合に対応したりすることで遅延が生じた．

12 月から，統計情報表示機能の 4.3 節に示したものを概ね実装し終えたので，ユーザマ

ニュアルを作成し始めた．また，TLV ユーザ 1 名に協力して頂き，統計情報表示機能のレビューを実施した．

6.1.3.3 成果

統計情報表示機能を実装した．また，統計情報表示機能のユーザマニュアルを作成した．

6.2 チケット数の推移

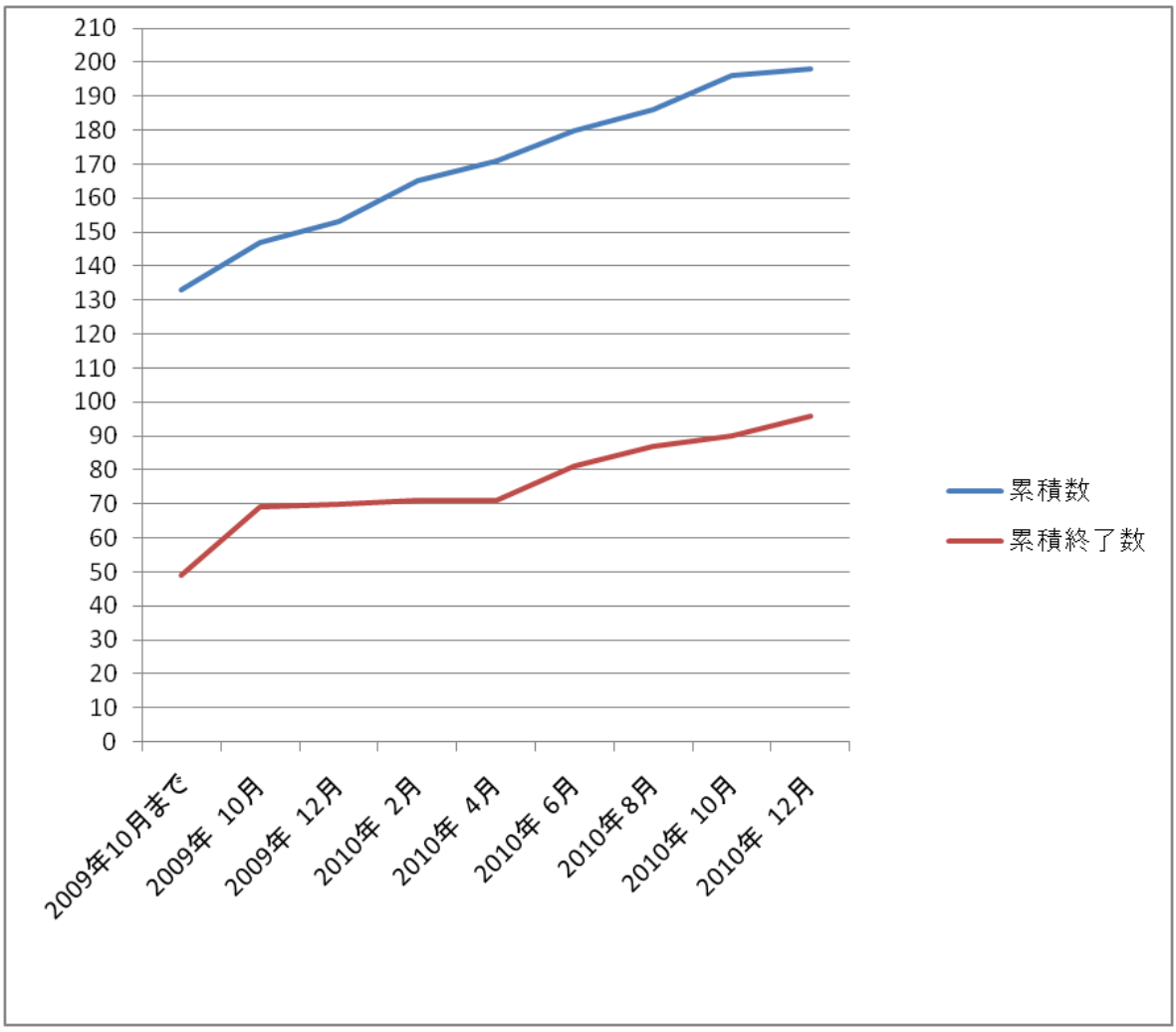

図 6.4 チケット累積件数

第 3 期 OJL で収集した要求・不具合はすべて Trac のチケットとして管理した．フェーズ 5 で新たに追加されたチケット件数は 32 件，終了したチケットは 22 件，フェーズ 6 で

新たに追加されたチケット件数は 21 件，終了したチケットは 16 件，フェーズ 7 で新たに追加されたチケット件数は 12 件，終了したチケットは 9 件であった．

チケットの累積件数・累積終了件数を図 6.4 に示す．図より，まだ成長段階で，成熟にはまだ時間を要することがわかる．

6.3 発表実績

2010 年 9 月 15 日に催された第 2 回 NCES シンポジウム[*1]にてポスター発表を行った．

[*1] http://www.nces.is.nagoya-u.ac.jp/news/sympo2010.html

第 7 章

おわりに

7.1 所感

OJL を経験して，大きく 4 つのことを学習した．

まず，コミュニケーションの大切さ，難しさである．統計情報表示機能における要件定義で痛感した．要求を分析する際，自分は TLV ユーザでなく，また，RTOS に関する研究を行ってきたわけでもないので，統計情報にどのようなものがあり，どのようにすれば得られるのか，どのように統計情報を扱いたいのかなどわからなかった．しかし，ユーザにアンケートという形でコミュニケーションを取ることで，要件定義の質を向上させることができた．また，口頭説明や文字のみのコミュニケーションは，多くの誤解を生み，自分の考えをうまく相手に伝えられなかった．図を使うにしても，使用する図が適切か，使用方法に誤りがないかに気を付けなければ，誤解を生んでしまう．

次に，ドキュメント整備の重要性である．TLV の設計・実装を調査するときに痛感した．TLV には，クラス図やシーケンス図などの設計書がなく，ソースコードにもコメントが少ない．そのため，各クラスがどのように連携し，クラスのメンバはそれぞれどのような役割があるのかをコードリーディングで把握する必要がある．結果，調査や実装の長期化につながってしまった．

次に，スケジュール作成の難しさである．まず，ソフトウェア開発経験が乏しいため，作業の洗い出しが困難であり，見積もりも曖昧であった．そのため，計画通りになかなか進められなかった．

最後に，企業目線での意見や考え方などにふれつつ作業ができた喜びである．これは，通常の講義や研究では滅多に味わえない体験であるからだ．通常の講義や研究では，学術的な部分が大きいが，企業の目線は，作業の効率化やプロジェクト管理の手法など，実用

的なものであるように感じた.

7.2 まとめ

第 3 期 OJL では，トレースログ可視化ツールである TLV の開発を継続して行い，TLV に対して統計情報表示機能の追加とトレースログパーサのリファクタリングを行った．これらは，ユーザから収集した要求のうち，特に要求が強かった「CPU 利用率などの統計情報のグラフ表示」と「TLV の高速化」に対応したものである．

統計情報表示機能の追加に伴い，統計情報生成ルールという新しいルールを導入した．これにより，トレースログやそれ以外の様々なファイルから統計情報を取得・生成し，その統計情報に適したグラフの設定が行える．新しいルールだが，既存のルールにある表現をこのルールでも使用する場合，既存のルールに合わせることで利用方法の習得のしやすさを向上させた．また，統計情報生成ルールに生成モードという統計情報の取得・生成するモードを 4 種類用意し，ユースケースによって使い分けられるようにした．表示に関しては，統計情報をグラフ表示するウィンドウを追加し，統計情報の取得・生成とグラフ表示を TLV 内で完結させ，他のアプリケーションを利用せずに済むようにした．

「TLV の高速化」を行うためには，標準形式トレースログへの変換と図形データの生成を高速化する必要がある．その際，それぞれの処理に関するソースコードが複雑化している点が障害となる．そこで，OJL2 期生によって立てられたリファクタリング方針を参考に，変換処理に関係するトレースログのパーサ部分のリファクタリングを行った．結果，パーサ・コンビネータを用いることで，コードで EBNF を表現できるようになり，可読性が向上し，若干ながら高速化も図れた．

7.3 今後の課題

第 3 期 OJL で実装した統計情報表示機能は，構想・設計段階において考えられていたいくつかの機能を実装し切れていない．それらの対応が主な課題となる．また，統計情報表示機能に対するユーザのレビューで得られたアイディアもあり，統計情報表示機能はまだ発展途上といえる．現在，確認している課題を以下に示す．

- 統計情報生成ルールファイルの入力を可視化後にも行える機能
- 平均・最小・最大を表現するグラフ (現在，積み重ねグラフを考えている)
- グラフ表示中にグラフ設定を変更できる「グラフ設定ウィンドウ」の実装

- 基本解析モードにおける「測定したイベント間隔のカウント」を行う解析メソッドの実装
- 基本解析モードにおけるオプション設定である事前条件の設定実装
- 可視化表示部またはマクロビューアで範囲指定した部分の統計情報を生成してグラフ表示する機能
- 統計情報同士を演算処理して新たな統計情報を生成する機能
- イベントの指定を GUI で行える機能 (統計情報表示機能の限りではない)

謝辞

第 3 期 OJL において，多くのご指導を頂きました山本晋一郎教授に深く感謝致します．お忙しい中，副査を引き受けてくださった戸田尚宏教授，太田淳准教授に深く感謝致します．

TLV を開発するにあたり，ご指導を頂きました名古屋大学大学院情報科学研究科情報システム学専攻組込みリアルタイムシステム研究室の高田広章教授に深く感謝致します．また，開発プロジェクトマネージャとして日頃より多くのご助言を頂きました名古屋大学大学院情報科学研究科附属組込みシステム研究センターの本田晋也准教授，企業出身者としての立場から実践的なご意見を頂きました同研究センターの鴫原一人研究員，同研究センターの長尾卓哉研究員に深く感謝致します．

統計情報表示機能に関して，アンケートにご協力下さいました名古屋大学大学院情報科学研究科附属込みシステム研究センターの松原豊研究員，同研究科情報システム学専攻組込みリアルタイムシステム研究室の安藤友樹氏に深く感謝致します．また，統計情報表示機能のレビューにご協力下さいました同研究室の佐野泰正氏に深く感謝致します．

参考文献

[1] 後藤隼弐『OJL によるトレースログ可視化ツールの開発』修士論文, 名古屋大学,2009

[2] 後藤隼弐, 本田晋也, 長尾卓哉, 高田広章, トレースログ可視化ツール TraceLogVisualizer(TLV), コンピュータ ソフトウェア, Vol.27, No.4, pp.8-23, Nov 2010.

[3] 水野洋樹『トレースログ可視化ツール (TLV) に対する機能追加とリファクタリング』OJL 報告書, 名古屋大学,2010

[4] 柳沢大祐『トレースログ可視化ツール (TLV) に対するアプリログ機能追加とプロファイリング』OJL 報告書, 名古屋大学,2010

[5] RFC4627 The application/json Media Type for JavaScript Object Notation (JSON), http://tools.ietf.org/html/rfc4627, 最終アクセス 2011 年 2 月 1 日

[6] TOPPERS Project, http://www.toppers.jp/, 最終アクセス 2011 年 2 月 1 日

[7] OJL による最先端技術適応能力を持つ IT 人材育成拠点の形成, http://www.ocean.is.nagoya-u.ac.jp/, 最終アクセス 2011 年 2 月 1 日

[8] 小林隆志, 沢田篤史, 山本晋一郎, 野呂昌満, 阿草清滋, " On the Job Learning: 産学連携による新しいソフトウェア工学教育手法 ", 電子情報通信学会信学技報 SS2009-28, vol.109, no.170, pp.95−100 2009

[9] TOPPERS プロジェクト／ ASP カーネル, http://www.toppers.jp/asp-kernel.html, 最終アクセス 2011 年 2 月 1 日

[10] TOPPERS プロジェクト／ FMP カーネル, http://www.toppers.jp/fmp-kernel.html, 最終アクセス 2011 年 2 月 1 日

[11] LukeH's WebLog : Monadic Parser Combinators using C# 3.0, http://blogs.msdn.com/lukeh/archive/2007/08/19/monadic-parser-combinators-using-c-3-0.aspx
, 最終アクセス 2011 年 2 月 1 日

[12] .NET Framework の正規表現,

http://msdn.microsoft.com/ja-jp/library/hs600312(v=VS.80).aspx，最終アクセス 2011 年 2 月 1 日

[13] グループ化構成体,
http://msdn.microsoft.com/ja-jp/library/bs2twtah%28VS.80%29.aspx
#BalancingGroupDefinitionExample，最終アクセス 2011 年 2 月 1 日