# OJL によるトレースログ可視化ツールの開発

350702101　**後藤 隼弐**

## 要旨

近年，組込みシステムにおいても，マルチプロセッサの利用が進んでいる．その背景には，シングルプロセッサの高クロック化による性能向上効果の限界や，消費電力の増大がある．マルチプロセッサシステムでは，処理の並列性を高めることにより性能向上を実現するため，消費電力の増加を抑えることができる．しかし，マルチプロセッサ環境で動作するソフトウェアのデバッグは困難であるという問題がある．これは，処理の並列性からプログラムの挙動が非決定的になり，バグの再現が保証されないため，シングルプロセッサ環境で用いられているブレークポイントやステップ実行を用いた従来のデバッグ手法が有効でないからである．

一方，マルチプロセッサ環境で有効なデバッグ手法として，プログラム実行履歴であるトレースログを解析する手法が挙げられる．この手法が有効である理由は，並列プログラムのデバッグにおいて必要な情報である，各プロセスが，いつ，どのプロセッサで，どのように動作していたかということを，トレースログを解析することで知ることができるからである．しかしながら，開発者が直接トレースログを解析するのは効率が悪いという問題がある．これは，膨大な量となるトレースログから所望の情報を探し出すのが困難であることや，各プロセッサのログが時系列に分散して記録されるため，逐次的にログを解析することが困難であることが理由である．

トレースログの解析を支援する方法として，ツールによるトレースログの可視化が挙げられ，これまでに多くのトレースログ可視化ツールが開発されている．具体的には，組込みシステム向けデバッガソフトウェアや統合開発環境の一部，Unix 系 OS のトレースログプロファイラなどが存在する．しかしながら，これら既存のツールが扱うトレースログは，形式が標準化されておらず，環境（OS やデバッグハードウェア）毎に異なるため，可視化対象が限定されおり，汎用性に乏しい．さらに，可視化表示項目が提供されているものに限られ，追加や変更が容易ではないなど，拡張性に乏しいといった問題もある．

そこで我々は，これらの問題点を解決し，汎用性と拡張性を備えたトレースログ可視化ツールを開発することを目的とし，TraceLogVisualizer (TLV) を開発した．まず，TLV 内部でトレースログを抽象的に扱えるよう，トレースログを一般化した標準形式トレースログを定め，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した．次に，トレースログの可視化表現を指示する仕組みを抽象化し，可視化ルールとして形式化した．TLV では，変換ルールと可視化ルールを外部ファイルとして与えることで，汎用性と拡張性を実現した．

開発した TLV を用いて，シングルコアプロセッサ用 RTOS(Real-time operating system) やマルチコアプロセッサ用 RTOS，組込みコンポーネントシステムなど，形式が異なる様々なトレースログの可視化を試み汎用性の確認を行った．また，可視化表示項目の変更，追加を試み拡張性の確認を行った．その結果，変換ルールと可視化ルールの変更，追加でこれらが実現可能であることを示した．

TLV の開発は，OJL(On the Job Learning) 形式で行い，開発プロセスに，ユースケース駆動アジャイル開発を適用して実施した．

# Development of Visualization Tool for Trace Log by OJL

350702101 Junji Goto

## Abstract

In recent years, multiprocessors have been used even in embedded systems. The reason for this is limits of effectiveness that improve a performance through overclocking in a single processor. A multiprocessor system can reduce the increase in power consumption even though improving performance by parallel processing. However, there is a problem that it is difficult to debug software executed in multiprocessors. This means that a traditional debugging way using breakpoints and step excuting is ineffective, because a repeatability of bug is made low by a nondeterministic behavior of parallel processing.

On the other hand, an effective technique to debug software excuted in multiprocessors includes analyzing a trace log that is a program execution history. The reason for that the technique is effective is that it can know necessary information for a parallel program debugging by analyzing the trace log. The information, for example, are when, where or how long processes executed. However, it is inefficient that developeres analyze the trace log directly, because searching a desired information in tremendous quantities of trace log and analyzing time-series trace log recorded by a number of processors sequentially are difficult.

Techniques to support developeres to analyze the trace log includes visualizing trace log by tools. And, many visualizing tools for trace log have been developed before now. In particular, there are debugging software or integrated development environment for embedded systems and trace log profilers for Unix-like operating systems. However, these existing tools lack general versatility because they treat only own format trace log. And, they lack expandability because visualized information are limited to items provided by them.

To that end, we developed TraceLogVisualizer(TLV), a visualization tool forthe trace log, for the purpose of implementation of general versatility and expandability. First, we defined the standard-format-trace-log with generalizing a trace log. This is necessity for that the TLV treat the trace log abstractively. And, we provided a mechanism that any format trace logs convert to the standard-format-trace-log by writing a convert-rule-file. Next, we formalized a mechanism associating shapes and trace logs as visualize-rule-file through abstracting them.

We confirmed general versatility through attemptting to visualize trace logs of a wide variety of formats including the trace log of RTOS (Real-time operating system) for a singlecore processor and multicore processors, and embedded component systems. Also, we confirmed expandability through attemptting to add and change visualized information. In the result, they are achieved by writing convert-rule-files and visualize-rule-files.

Development of TLV was performed as OJL(On the Job Learning), and development process carried out by applying use case driven agile development.

修 士 論 文

# OJLによるトレースログ可視化ツールの開発

350702101　後藤 隼弐

名古屋大学 大学院情報科学研究科

情報システム学専攻

2009年1月

# 目 次

# 図 目 次

# 表 目 次

# 第1章　はじめに

## 1.1　開発背景

近年，PC，サーバ，組込みシステム等，用途を問わずマルチプロセッサの利用が進んでいる．その背景には，シングルプロセッサの高クロック化による性能向上効果の限界や，消費電力・発熱の増大がある．マルチプロセッサシステムでは処理の並列性を高めることにより性能向上を実現するため，消費電力の増加を抑えることができる．組込みシステムにおいては，機械制御と GUI など要件の異なるサブシステム毎にプロセッサを使用する例があるなど，従来から複数のプロセッサを用いるマルチプロセッサシステムが存在していたが，部品点数の増加によるコスト増を招くため避ける方向にあった．しかしながら，近年は，1 つのプロセッサ上に複数の実行コアを搭載したマルチコアプロセッサの登場により低コストで利用することが可能になり，低消費電力要件の強い，組込みシステムでの利用が増加している．

マルチプロセッサ環境でソフトウェアを開発する際に問題になる点として，デバッグの困難さが挙げられる．これは，処理の並列性からプログラムの挙動が非決定的になり，バグの再現が保証されないため，シングルプロセッサ環境で用いられているブレークポイントやステップ実行を用いた従来のデバッグ手法が有効でないからである．

一方，マルチプロセッサ環境で有効なデバッグ手法として，プログラム実行履歴であるトレースログを解析する手法が挙げられる．この手法が有効である理由は，並列プログラムのデバッグにおいて必要な情報である，各プロセスが，いつ，どのプロセッサで，どのくらい動作していたかということを，トレースログを解析することで知ることができるからである．しかしながら，開発者が直接トレースログを解析するのは効率が悪い．これは，膨大な量となるトレースログから所望の情報を探し出すのが困難であることや，各プロセッサのログが時系列に分散して記録されるため，逐次的にログを解析することが困難であることが理由である．

しかしながら，トレースログを開発者が直接扱うのは困難である場合が多い．これは，ログの情報の粒度が細かくなるほど単位時間あたりのログの量が増える傾向にあり，膨大なログから所望の情報を探し出すのが困難であることや，各コアのログが時系列に分散して記録されるため，逐次的にログを解析することが困難であるからである．そのため，トレースログの解析を支援するツールが要求されており，ログを解析し統計情報として出力したり，可視化表示することで開発者の負担を下げる試みが行われている．

## 1.2 既存のトレースログ可視化ツール

既存のトレースログ可視化表示ツールとしては，組込みシステム向けデバッガソフトウェアや統合開発環境の一部，Unix 系 OS のトレースログプロファイラ，波形表示用ツールの流用などがある．

本節では，これら既存のトレースログ可視化ツールについて説明した後，これらの問題点について指摘する．

### 1.2.1 組込みシステム向けデバッガソフトウェア

組込みシステム向けデバッガソフトウェアには，機能の１つとしてトレースログを可視化する機能が含まれている場合がある．組込みシステム向けデバッガソフトウェアとは，ICE（In-Circuit Emulator）や JTAG エミュレータなどの，組込みシステム向けデバッガに付属するデバッグ用のソフトウェアである．組込みシステム向けデバッガとは，ターゲットシステム上で動くプログラムをホストシステム上でデバッグを行えるようにするために，ターゲット CPU にアクセスする手段を提供する装置を指す．

組込みシステム向けデバッガソフトウェアとしては，京都マイクロコンピュータ株式会社の PARTNER[1] や，株式会社ソフィアシステムズの WatchPoint[2] などがあり，それぞれ，イベントトラッカー，OS アナライザというトレースログを可視化する機能を提供している．図 1.1 に，イベントトラッカーのスクリーンショットを，図 1.2 に OS アナライザのスクリーンショットを示す．

これら組込みシステム向けデバッガソフトウェアの一機能としての可視化ツールは，その性質からターゲットとなる OS やプロセッサが限定される．これは，組込みシステム向けデバッガは，通常，ターゲットとなるプロセッサが限られており，デバッガソフトウェアが対応する OS も限られているからである．また，可視化表示したい情報も提供されているものに限られるなど，可視化ツールとしては汎用性，拡張性に乏しい．

### 1.2.2 組込み OS 向けの統合開発環境

QNX Software Systems 社は，自社の組込みリアルタイムオペレーティングシステム QNX の統合開発環境として QNX Momentics を販売している．QNX Momentics にはシステムプロファイラとして，システムコールや割込み，スレッド状態やメッセージなどを可視化する QNX System Profiler[3] という機能を提供している．図 1.3 に QNX System Profiler のスクリーンショットを示す．

また，イーソル株式会社は T-Kernel/$\mu$ITRON ベースシステムの統合開発環境として eBinder[4] を販売している．eBinder にはイベントログ取得・解析ツールとして EvenTrek が付属しており，システムコール，割込み，タスクスイッチ，タスク状態遷移などを可視化することができる．図 1.4 に EvenTrek のスクリーンショットを示す．

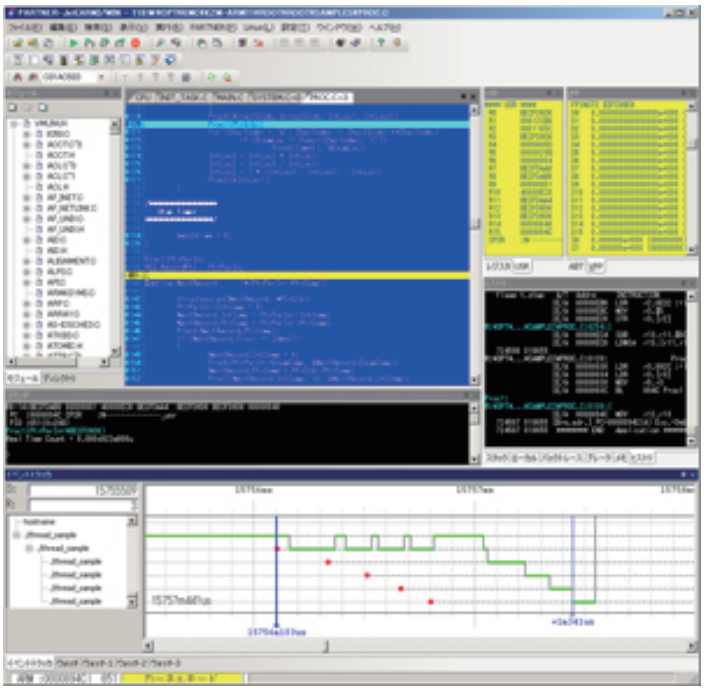

図 1.1: PARTNER イベントトラッカー

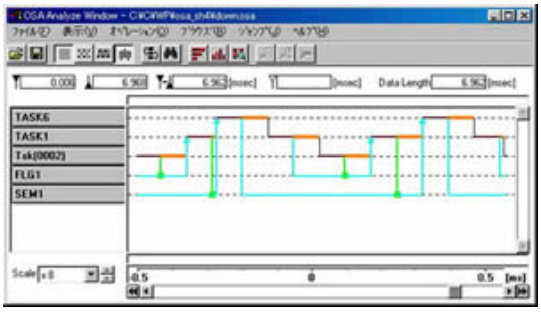


図 1.2: WatchPoint OS アナライザ

このように，商用の組込み OS 向けの統合開発環境には OS の実行履歴を可視化表示する機能が搭載されている場合がある．しかしながら，これらは，各ベンダーが自社 OS の競争力を高めるために提供しているものであり，当然ながら可視化表示に対応する OS は自社提供のものに限られている．また，可視化表示する情報も提供するものに限られており，表示のカスタマイズ機能もそれほど自由度は高くはない．

### 1.2.3 Unix 系 OS のトレースログプロファイラ

Unix 系 OS では，これまでに，パフォーマンスチューニングや障害解析を目的として，カーネルの実行トレースを取得するソフトウェアがいくつか開発されている．ここでは，単にこれをトレースツールと呼称する．

Linux 用のトレースツールとしては LKST[5]，SystemTap[6]，LTTng[7] などがあり，Solaris 用には Dtrace[8] がある．これらトレースツールが提供する主な機能は，カーネル内にフックを仕込みカーネル内部状態をユーザー空間に通知する機能と通知をログとして記録する機能である．

これらのトレースツールは，ログを分析，提示する，専用のプロファイラツールを提供している場合が多い．たとえば，LTTng には LTTV[9] が，DTrace には Chime[10] が，プロファイラツールとして提供されている．図 1.5 に LTTV のスクリーンショットを，図 1.6 に Chime のスクリーンショットを示す．

これら，プロファイラツールは，主に，カーネルの内部状態を統計情報として出力することにより，ボトルネックを探したり，障害の要因を探る目的で使用されるが，ソフトウェアのデバッグを目的に使用することもできる．DTrace などはログ出力のためのカーネルフックポイントを独自のスクリプト言語を用いて制御できるなど，任意の情報をソフトウェアの実行から取得することができる．しかしながら，取得したログを任意の図形で可視化する手段は提供されておらず，テキスト形式での確認となる．また，可視化表示する際のログの形式は，その OS のトレースツールが出力する形式に依存するため，他の OS のトレースを可視化することはできない．

### 1.2.4 波形表示ツールの流用

任意の OS，アプリケーションのトレースログを可視化表示する手段として，波形表示ツールを流用する方法がある．波形表示ツールとは，Verilog 等のデジタル回路設計用論理シミュレータの実行ログを波形で表示するソフトウェアのことを指す．

デジタル回路設計用論理シミュレータの実行ログには，VCD(Value Change Dump) 形式というオープンなファイルフォーマットが存在する．そのため，任意のログを VCD 形式として出力することにより，これらのツールで可視化表示することが可能になる．図 1.7 に，VCD 形式のログの可視化に対応した波形表示ツール GTKWave のスクリーンショットを示す．

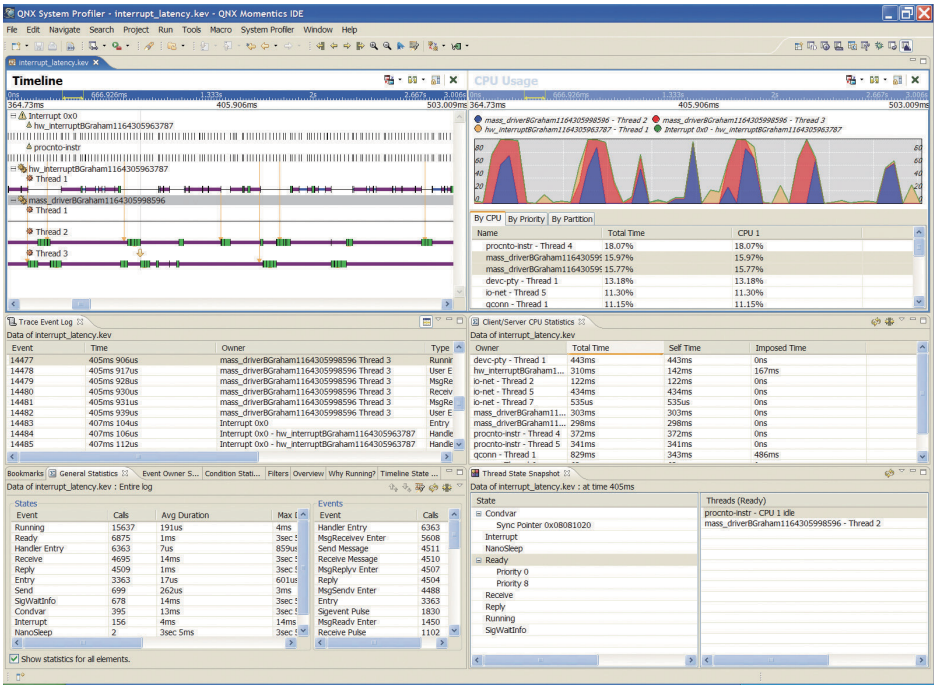


図 1.3: QNXSystemProfiler

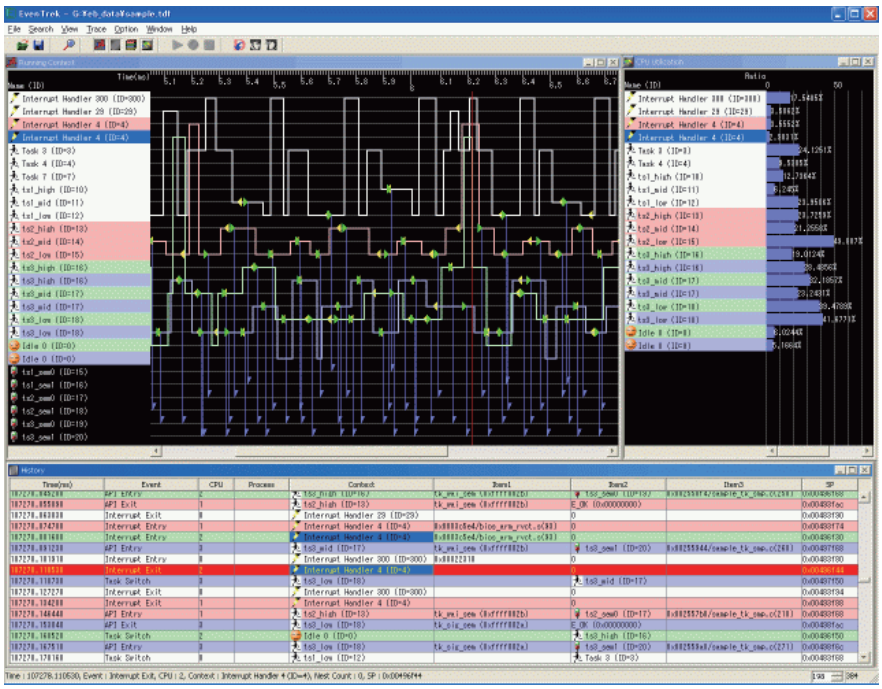


図 1.4: eBinder EvenTrek

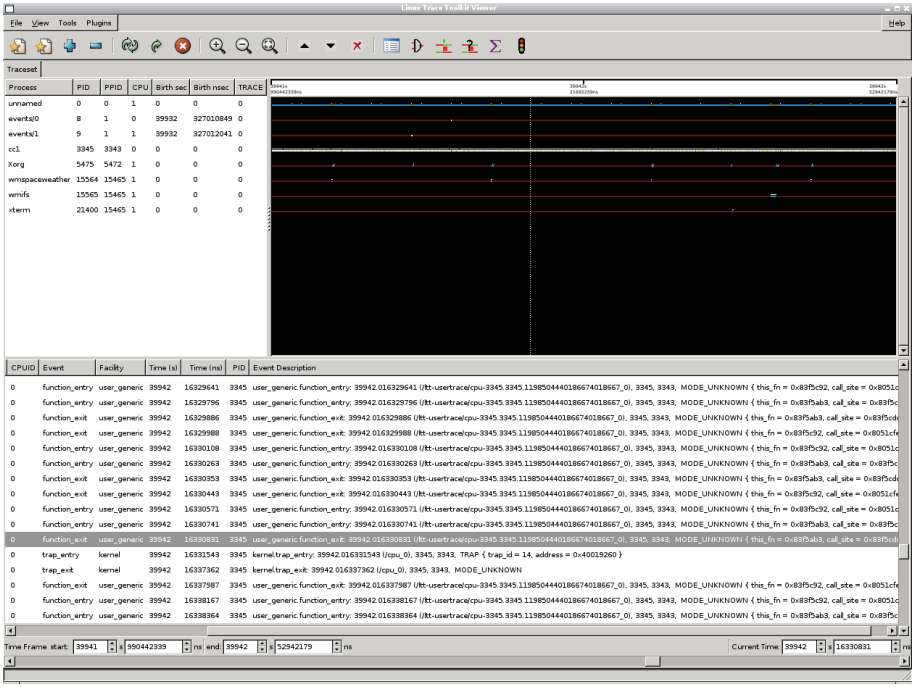

図 1.5: LTTV

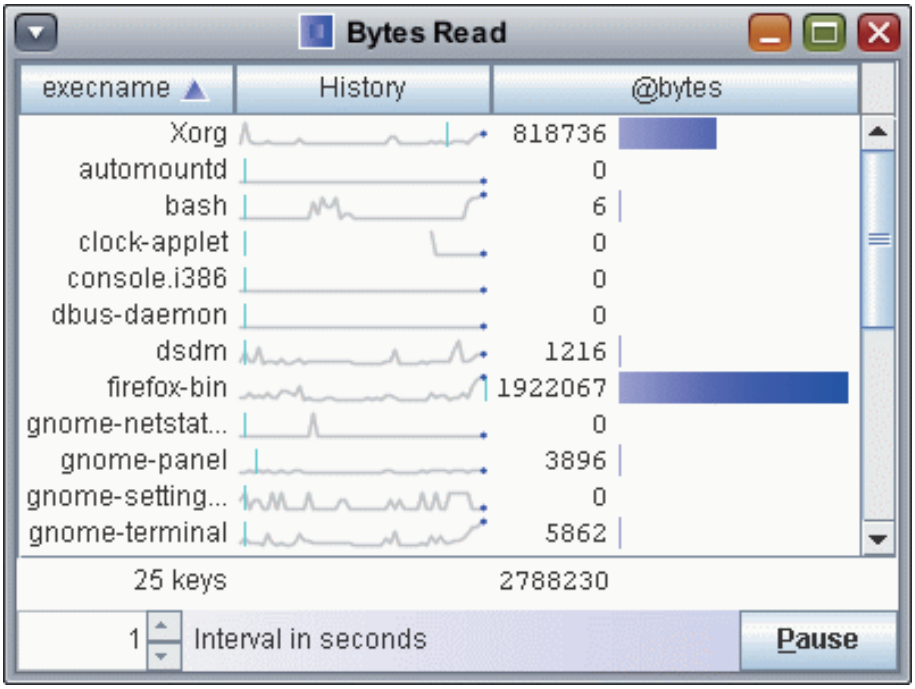


図 1.6: Chime

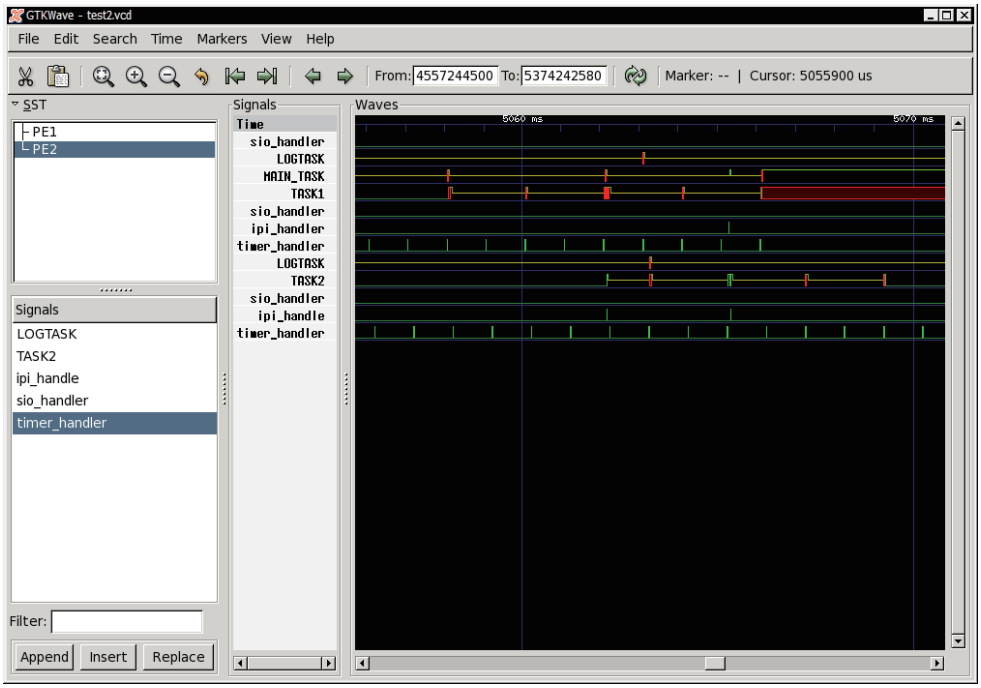


図 1.7: GTKWave

波形表示ツールを流用する方法では，任意のログをオープンフォーマットなファイル形式に変換することによりログの形式に依存せずに利用できる反面，表示能力に乏しく，複雑な可視化表現は難しいという問題がある．

### 1.2.5 既存のトレースログ可視化ツールの問題点

既存のトレースログ可視化ツールは，可視化ツールとして単体で存在しているわけではなく，トレースログを出力するソフトウェアの一部として提供されている．そのため，読み込めるトレースログの形式が出力ソフトウェアに依存しており，可視化対象となる OS，プロセッサが制限されてしまっている．

読み込むトレースログの形式を制限しないためには，波形表示ツールのようにトレースログ可視化ツール用に標準化されたトレースログ形式を定める必要があると考えられるが，現状，公表されているものではそのようなものは確認できていない．

ログ形式の標準化としては，syslog[11] と呼ばれる，ログメッセージを IP ネットワーク上で転送するための標準規格が RFC3164 として策定されており，その一部にログ形式について規定されている．しかしながら，syslog におけるログ形式は，時刻の最小単位が秒であり，デバッグを目的に利用するには粒度が粗く，また，メッセージ内容がフリーフォーマットであるなど，自由度が高すぎるため，可視化ツールが読み込むログの標準形式としては不適切である．

既存のトレースログ可視化ツールのもうひとつの問題点としては，可視化表示の項

目や形式が，提供されているものに限られていることが挙げられる．既存のトレースログ可視化ツールで可視化表示の項目を追加，変更したり，可視化表現をカスタマイズする機能を搭載するものは確認できていない．

## 1.3 開発目的と内容

前節で説明したとおり，既存のトレースログ可視化ツールには，標準化されたトレースログ形式がないことによりターゲットが限定されてしまっているという，汎用性に乏しい点と，可視化表示項目が提供されているものに限られているという，拡張性に乏しい点の２つの問題点があることを指摘した．

そこで我々は，これらの問題点を解決し，汎用性と拡張性を備えたトレースログ可視化ツールを開発することを目的とし，TraceLogVisualizer (TLV) を開発した．

はじめに，TLV の内部でトレースログを抽象的に扱えるよう，トレースログを一般化した標準形式トレースログを定めた．次に，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した．この変換ルールを外部ファイルとして与えることで，任意の形式のトレースログを読み込めるようになる．

また，トレースログの可視化表現を指示する仕組みを抽象化し，可視化ルールとして形式化した．この可視化ルールも，同じように外部ファイルとして与えることで，可視化表示項目の追加や変更，可視化の表現方法を自由に設定することができるようになる．

TLV では，このように，変換ルールと可視化ルールを外部ファイルとして与えることで，汎用性と拡張性を実現した．これらの外部ファイルは，ユーザーが記述することを想定し，読み書きのしやすいシンタックスを持つ Json(JavaScript Object Notation)[12] というデータ記述言語を採用し，ファイルの記述内容に合わせてセマンティクスを規定した．

## 1.4 論文の構成

本節では本論文の構成を述べる．

2 章では，TLV の設計について述べる．ここでは，問題解決のために開発方針をどのように設定したかを述べ，具体的な解決策をどのように設計したかを述べる．3 章では，TLV の実装について述べる．2 章で述べた設計をメカニズムとしてどのように実現しているかを説明する．4 章では，TLV を利用した例を示し，その有効性について言及している．5 章では TLV を開発するにあたり，どのような開発プロセスを用いたのかを述べている．最後に 6 章で本論文のまとめと今後の展望と課題について述べる．

# 第2章　トレースログ可視化ツール TraceLogVisualizer の設計

## 2.1　開発方針

TLV の開発目標は，汎用性と拡張性を実現することである．

ここで，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．

次に，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　標準形式トレースログ

本節では，標準形式トレースログを定義するために行ったトレースログの抽象化と，標準形式トレースログの定義について述べる．

### 2.2.1　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．

はじめに，トレースログを，時系列にイベントを記録したものと考えた．次に，イベントとはイベント発生源の属性の変化，イベント発生源の振る舞いと考えた．ここで，イベント発生源をリソースと呼称し，固有の識別子をもつものとする．つまり，リソースとは，イベントの発生源であり，名前を持ち，固有の属性をもつものと考えることができる．

リソースは型により属性，振る舞いを特徴付けられる．ここでリソースの型をリソースタイプと呼称する．

属性は，リソースが固有にもつ文字列，数値，真偽値で表されるスカラーデータとし，振る舞いはリソースの行為であるとする．

リソースタイプとリソースの関係は，オブジェクト指向におけるクラスとオブジェクトの関係に類似しており，属性と振る舞いはメンバ変数とメソッドに類似している．ただし，振る舞いはリソースのなんらかの行為を表現しており，メソッドの，メンバ変数を操作するための関数や手続きを表す概念とは異なる．

主に，振る舞いは，属性の変化を伴わないイベントを表現するために用いる．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，これは，図形描画の際の条件，あるいは描画材料として用いられることを想定している．

図 2.1 と図 2.2 に，リソースタイプとリソースを図で表現した例を示す．さらに，図 2.3 に，RTOS(Real-time operating system) におけるタスクの概念をリソースタイプとして表現した例を，図 2.4 に，リソースタイプ Task のリソースの例として MainTask を示す．

トレースログの抽象化を以下にまとめる．

**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．

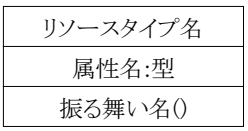


図 2.1: リソースタイプ

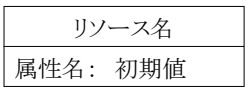


図 2.2: リソース

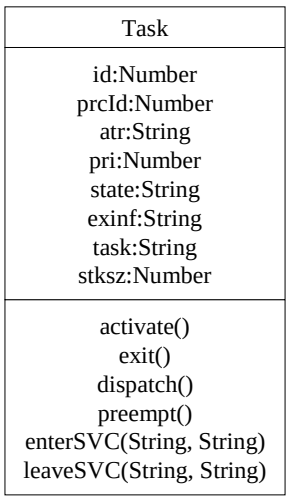


図 2.3: タスクをリソースタイプ Task として表現した例

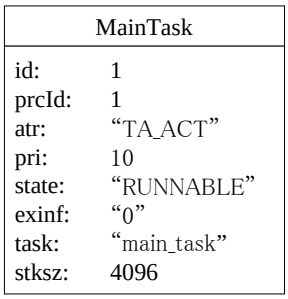


図 2.4: リソースタイプ Task のリソース MainTask の例

### 2.2.2 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,")";
ComparisonExpression = AttributeName,ComparisonOpe,Value;
Boolean = "true"|"false";
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 はリソースの名前であり，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 は属性の値の変化を，BehaviorHappen は振る舞いを表現しているとした．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．

### 2.2.3 標準形式トレースログの例

前小節の定義を元に記述した，標準形式トレースログの例を表 2.1 に示す．

1 行目，3 行目，5 行目がリソースの振る舞いイベントであり，2 行目，4 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おり，残りの振る舞いイベントには指定されていない．

1 行目，2 行目はリソースを名前で直接指定しているが，残り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．

```
1 [2403010]MAIN_TASK.leaveSVC(ena_tex,ercd=0)
2 [4496099]MAIN_TASK.state=RUNNABLE
3 [4496802]TASK(state==RUNNING).preempt()
4 [4496802]TASK(state==RUNNING).state=RUNNABLE
5 [4496802]TASK(id==2).dispatch()
```

表 2.1: 標準形式トレースログの例

## 2.3 可視化表示の仕組みの抽象化

前節では，トレースログを一般化し，標準形式トレースログとして定義した．TLV の可視化表示の仕組みは，この標準形式トレースログにのみ依存するように設計されなければならない．本節では，可視化表現と可視化表現とトレースログの対応を抽象化する．

### 2.3.1 可視化表現

TLV において，トレースログの可視化表現は，x 軸を時系列とした 2 次元直交座標系における図形の描画であるとした．本小節では，座標系と図形について詳述する．

#### 座標系

図形を定義する座標系と，表示する座標系は分離して考えるとする．これにより，図形を表示環境から独立して定義することが可能になる．図形を定義する座標系をローカル座標系，表示する座標系をデバイス座標系と呼称する．

また，TLV では，高さと時間を次元に持つ，ワールド座標系という座標系を導入した．ローカル座標系で定義された図形は，はじめに，ワールド座標系における，図形を表示すべき時間の領域にマッピングされ，これを表示環境に依存するデバイス座標系にマッピングすることで表示する．これにより，図形の表示領域を，抽象度の高い時刻で指定することが可能になる．ここで，ローカル座標系からワールド座標系へのマッピングをワールド変換と呼称する．また，表示する時間の領域を表示期間と呼称する．表示期間は開始時刻と終了時刻で表される時刻のペアである．

ローカル座標系において，図形の大きさと位置を定義する際は，pixel 単位による絶対指定か，ワールド座標系へのマッピング領域に対する割合を%で指定する相対指定かのいずれかを用いる．

図 2.5 に座標系の例を，図 2.6 にワールド変換の例を示す．

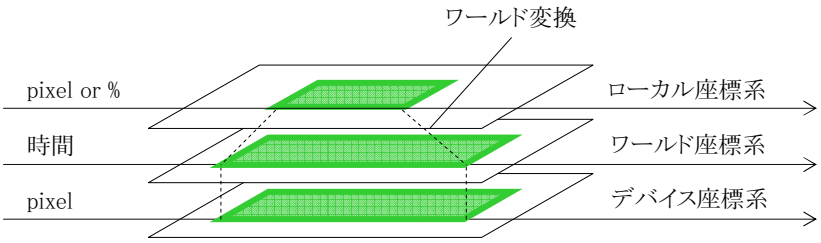


図 2.5: 座標系

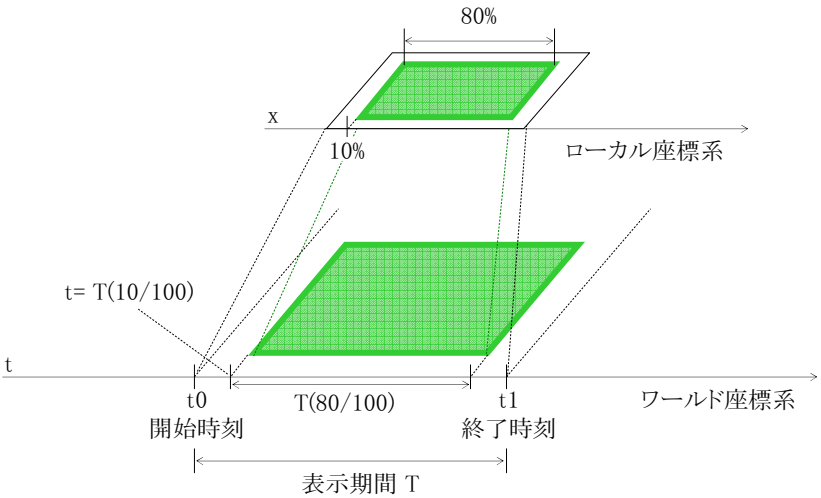


図 2.6: ワールド変換

### 基本図形と図形，図形群

可視化表現は，複数の図形を組み合わせることで実現する．この際，基本となる図形の単位を基本図形と呼称する．

基本図形として扱える形状は楕円，多角形，四角形，線分，矢印，扇形，文字列の 7 種類とする．基本図形は，形状や大きさ，位置，塗りつぶしの色，線の色，線種，透明度などの属性を指定して定義する．

複数の基本図形を仮想的に z 軸方向に階層的に重ねたものを，単に図形と呼称し，可視化表現の最小単位とする．図形は，構成する基本図形を順序付きで指定し，名前をつけて定義する．図形は名前を用いて参照することができ，その際に引数を与えることができるとする．この際，引数は，図形を構成する基本図形の，任意の属性に割り当てることを想定している．

複数の図形を仮想的に z 軸方向に階層的に重ねたものを図形群と呼称する．図 2.7 に図形と図形群の例を示す．

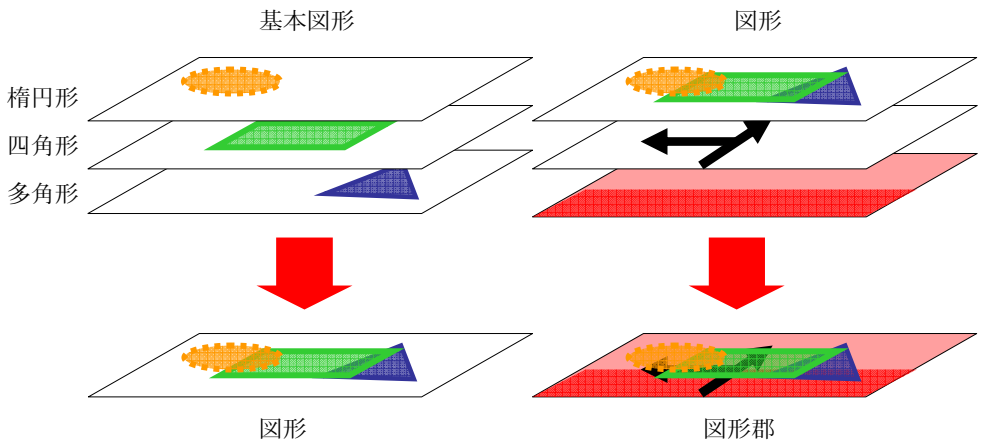


図 2.7: 図形と図形群

### 2.3.2 図形とイベントの対応

本小節では，前小節で述べた可視化表現とトレースログのイベントをどのように対応付けるのかを述べる．

#### 開始イベント，終了イベント，イベント期間

前小節において，可視化表現は，図形をワールド変換を経て表示期間にマッピングすることであることを説明した．ここで，表示期間の開始時刻，終了時刻を，イベントを用いて指定するとする．つまり，指定されたイベントが発生する時刻をトレースログより抽出することにより表示期間を決定する．このようにして，トレースログのイベントと可視化表現を対応付ける．ここで，開始時刻に対応するイベントを開始イベント，終了時刻に対応するイベントを終了イベントと呼称し，表示期間をイベントで表現したものをイベント期間と呼称する．

#### 可視化ルール

図形群と，そのマッピング対象であるイベント期間を構成要素としてもつ構造体を，可視化ルールと呼称する．図 2.8 に，標準形式トレースログを用いてイベント期間を定義した可視化ルールの例を示す．図 2.8 において，runningShape を，位置がローカル座標の原点，大きさがワールド座標系のマッピング領域に対して横幅 100%，縦幅 80%の長方形で色が緑色の図形とする．この図形を，開始イベント `MAIN_TASK.state=RUNNING`，終了イベント `MAIN_TASK.state` となるイベント期間で表示するよう定義したものが可視化ルール taskBecomeRunning である．開始イベント `MAIN_TASK.state=RUNNING` は，リソース `MAIN_TASK` の属性 `state` の値が `RUNNING` になったことを表し，終了イベント `MAIN_TASK.state` は，リソース `MAIN_TASK` の属性 `state` の値が単に変わったことを表している．

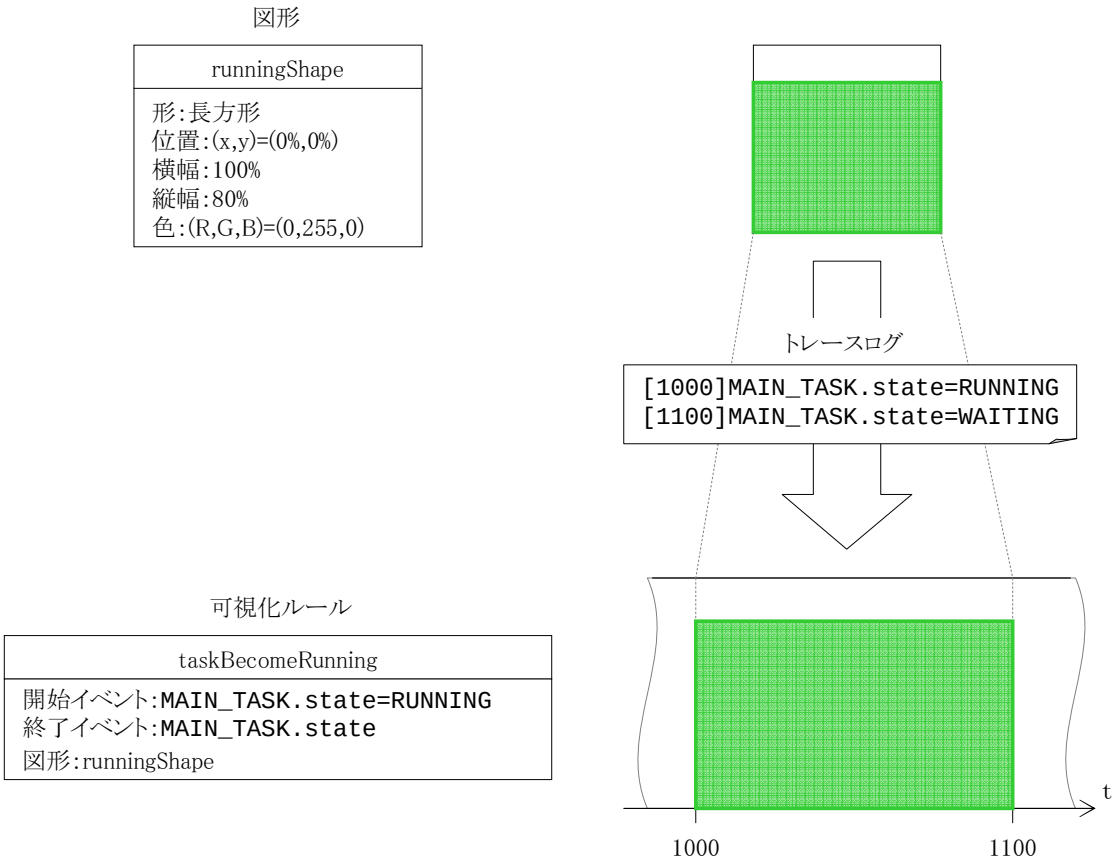


図 2.8: 可視化ルール

```
1 [1000]MAIN_TASK.state=RUNNING
2 [1100]MAIN_TASK.state=WAITING
```

表 2.2: イベント期間を抽出するトレースログ

taskBecomeRunning を，表 2.2 に示すトレースログからイベントを抽出して表示期間の時刻を決定し，図形のワールド変換を行った結果が図 2.8 の右下に示すものである．

# 第3章　トレースログ可視化ツール TraceLogVisualizer の実装

## 3.1　TraceLogVisualizer の全体像

TLV の主機能は，2 つの主たるプロセスと 6 種の外部ファイルによって実現される．図 3.1 に TLV の全体像を示す．

2 つの主たるプロセスとは，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，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，変換した標準形式トレースログに対して可視化ルール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．

TLV は，トレースログとリソースファイルを読み込み，トレースログの対象に対応したリソースヘッダファイル，変換ルールファイルの定義に従い，標準形式トレースログを生成する．生成された標準形式トレースログに可視化ルールファイルで定義される可視化ルールを適用し図形データを生成した後，画面に表示する．

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 3.2 に，TLV のスクリーンショットを示す．

### 3.1.1　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)[12] と呼ばれるデータ記述言語を用いて記述する．

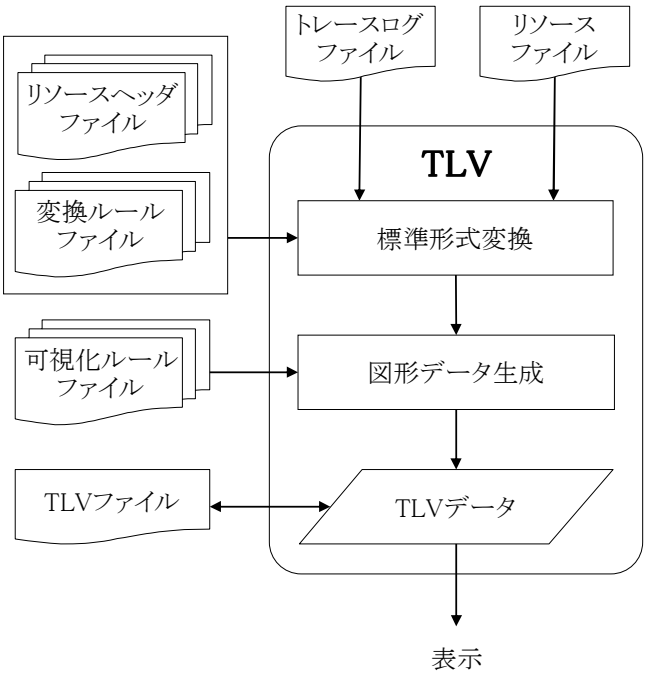


図 3.1: TLV の全体像

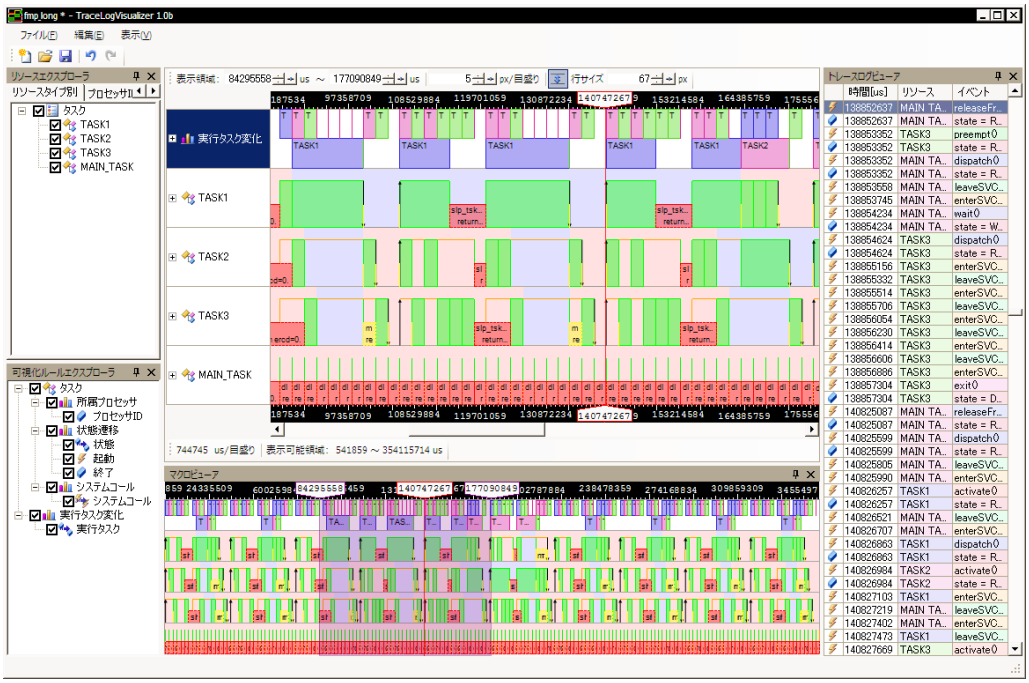

図 3.2: TLV のスクリーンショット

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 としてで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 言語, 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，名前のあとにはセミコロンが付く．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]/;
```

表 3.1 に Json におけるオブジェクトを定義した例を示す．
表 3.2 に Json における配列を定義した例を示す．

## 3.2 標準形式への変換

標準形式トレースログへの変換は，トレースログファイルを先頭から行単位で読み込み，変換ルールファイルで定義される置換ルールに従い標準形式トレースログに置換していくことで行われる．変換ルールファイルの詳細は 3.2.4 小節で説明する．

１つの置換ルールに対して複数の標準形式トレースログを出力可能である．しかし，所望の標準形式トレースログに変換する際，トレースログファイルの情報だけでは足りない場合がある．例として，TOPPERS/ASP カーネル [13] という RTOS のトレースログを標準形式トレースログに変換することを考えてみる．TOPPERS/ASP カーネルのトレースログの例を表 3.3 に示す．

表 3.3 のトレースログの内容を簡単に説明すると，時刻 1000 にタスク ID が 1 のタスクの状態が RUNNABLE になり，時刻 1005 に同タスクがディスパッチされ，時刻 1100 に同タスクの状態が WAITING になったことを示している．この場合，標準形式トレースログは表 3.4 のように出力されることが要求される．なお，説明のため簡略化しており，実際の変換結果とは異なる．

表 3.3 で示す元のトレースログが 3 行なのに対し，表 3.4 で示す要求される標準形式トレースログは５行となっている．これは，表 3.4 の 1 行目と 3 行目が状況により追加されるからである．表 3.4 の 1 行目は，表 3.3 の 1 行目に対応しており，起動 (`activate()`) というタスクの振る舞いを可視化したいという要求があるため追加される必要がある．また，表 3.4 の 3 行目は，表 3.3 の 2 行目に対応しており，すでに起動しているタスクが横取りされて状態が RUNNABLE になる，という情報が元のトレースログに存在しないため，追加される必要がある．

しかしながら，これら標準形式トレースログの追加が必要になるのは一定の条件下のみである．表 3.4 の 1 行目は，タスク ID が 1 のタスクの状態が，時刻 1000 未満のときに DORMANT である場合だけである．これは，起動という状態遷移を行うのが状

```
{
  "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]
  }
}
```

表 3.1: Json におけるオブジェクトを定義した例

```
[
  {
    "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"
  }
]
```

表 3.2: Json における配列を定義した例

```
1 [1000]task 1 becomes RUNNABLE
2 [1005]dispatch to task 1.
3 [1100]task 1 becomes WAITING
```

表 3.3: 変換元となる TOPPERS/ASP カーネルのトレースログの例

```
1 [1000]Task(id==1).activate()
2 [1000]Task(id==1).state = RUNNABLE
3 [1005]Task(state==RUNNING).state=RUNNABLE
4 [1005]Task(id==1).state = RUNNING
5 [1100]Task(id==1).state = WAITING
```

表 3.4: 表 3.3 を標準形式トレースログで表現した例

態が DORMANT から RUNNABLE に遷移するときだけであり，状態が DORMANT になっただけでは起動であるのかどうか判断できないためである．また，表 3.4 の 3 行目が必要なときは，時刻 1005 のときに状態が RUNNING のタスクが存在する場合だけである．

このように，リソース属性の遷移に伴うイベントや，元のトレースログに欠落している情報を補うイベントなど，元のトレースログの情報だけでは判断できないイベントを出力するには，特定時刻における特定リソースの有無やその数，特定リソースの属性の値などの条件で出力を制御できる必要がある．そのため，TLV の変換ルールでは，置換する条件の指定と，条件指定の際に用いる情報を置換マクロを用いて取得できる仕組みを提供した．具体的な記述例は 3.2.4 小節で述べる．

標準形式トレースログに含まれるリソースは，リソースファイルで定義されていなければならない．リソースファイルには，各リソースについて，その名前とリソースタイプ，必要であれば各属性の初期値を定義する．リソースファイルの詳細については 3.2.3 小節で述べる．また，その際に使用されるリソースタイプはリソースヘッダファイルで定義されていなければならない．リソースヘッダファイルには各リソースタイプについて，その名前と属性，振る舞いの定義を記述する．リソースヘッダファイルの詳細については 3.2.2 小節で述べる．

リソースヘッダ，変換ルール，可視化ルールは可視化するターゲット毎に用意する．その際のターゲットはリソースファイルに記述する．

### 3.2.1 トレースログファイル

標準形式トレースログに変換する元となるトレースログは，トレースログファイルとして読み込む．トレースログファイルはテキストファイルであり，行単位でトレースログが記述されていなければならない．これ以外のシンタックス，セマンティクスに関する制限はない．

```
 1 [11005239]: task 4 becomes RUNNABLE.
 2 [11005778]: dispatch from task 2.
 3 [11005954]: dispatch to task 4.
 4 [11006160]: leave to dly_tsk ercd=0.
 5 [11006347]: enter to dly_tsk dlytim=10.
 6 [11006836]: task 4 becomes WAITING.
 7 [11007050]: dispatch from task 4.
 8 [11007226]: dispatch to task 2.
 9 [11007758]: enter to sns_ctx.
10 [11007934]: leave to sns_ctx state=0.
11 [11008656]: enter to sns_ctx.
12 [11008832]: leave to sns_ctx state=0.
```

表 3.5: TOPPERS/ASP カーネルのトレースログの例

任意のトレースログファイルを標準形式トレースログに変換するには，ターゲットとなるトレースログの形式毎に変換ルールファイルを用意する必要がある．

表 3.5 に，RTOS である TOPPERS/ASP カーネルのトレースログの例を示す．

### 3.2.2 リソースヘッダファイル

リソースヘッダファイルにはリソースタイプの定義を記述する．リソースタイプの定義には，リソースタイプの名前，表示名，リソースタイプがもつ属性，振る舞いを記述する．

リソースヘッダは可視化するターゲット毎にリソースタイプを定義することができる．つまり，タスクを表すリソースタイプ Task を定義する際に，ターゲットとなる RTOS 毎に属性の内容を変えたい場合，RTOS 毎にリソースタイプ Task を定義することができる．

表 3.6 に，ターゲット asp のリソースタイプ Task を定義したリソースヘッダファイルの例を示す．リソースヘッダファイルは，1 つのオブジェクトで構成され，各メンバにターゲット毎のリソースタイプの定義を記述する．メンバ名にターゲット名を記述し，値としてそのターゲットに属する複数のリソースタイプを定義したオブジェクトを記述する．そのオブジェクトには，メンバ名にリソースタイプ名を，値にリソースタイプを定義したオブジェクトを記述する．以下に，リソースタイプを定義するオブジェクトのメンバの説明と値について説明する．

`DisplayName`
: **説明** リソースタイプの表示名．主に GUI 表示の際に用いられる
: **値** 文字列

```
{
  "asp":{
    "Task":{
      "DisplayName":"タスク",
      "Attributes":{
        "id":{
          "VariableType":"Number",
          "DisplayName":"ID",
          "AllocationType":"Static",
          "CanGrouping":false
        },
        "atr":{
          "VariableType":"String",
          "DisplayName":"属性",
          "AllocationType":"Static",
          "CanGrouping":false
        },
        /* 省略 */
        "state":{
          "VariableType":"String",
          "DisplayName":"状態",
          "AllocationType":"Dynamic",
          "CanGrouping":false,
          "Default":"DORMANT"
        }
      },
      "Behaviors":{
        "preempt":{"DisplayName":"プリエンプト"},
        "dispatch":{"DisplayName":"ディスパッチ"},
        "activate":{"DisplayName":"起動"},
        "exit":{"DisplayName":"終了"},
        /* 省略 */
        "enterSVC":{
          "DisplayName":"サービスコールに入る",
          "Arguments":{"name":"String","args":"String"}
        },
        "leaveSVC":{
          "DisplayName":"サービスコールから出る",
          "Arguments":{"name":"String","args":"String"}
        }
      }
    }
  }
}
```

表 3.6: リソースヘッダファイルの例

Attributes

**説明** 属性の定義

**値** オブジェクト．メンバ名に属性名，値に属性の定義をオブジェクトで記述する．その際のオブジェクトのメンバの説明は以下の通りである．

VariableType

**説明** 属性値の型

**値** 文字列 ("Number"：数値，"Boolean"：真偽値，"String"：文字列のいずれか)

DisplayName

**説明** 属性の表示名．主に GUI 表示の際に用いられる

**値** 文字列

AllocationType

**説明** 属性の値が動的か静的かの指定．ここで動的とは，属性値変更イベントが発生すること指し，静的とは発生しないことを指す．

**値** 文字列 ("Static"，"Dynamic"のいずれか)

CanGrouping

**説明** リソースをグループ化できるかどうか．ここで true を指定された場合，GUI でリソースの一覧を表示する際に初期値でグループ化され表示することができる．

**値** 真偽値

Behaviors

**説明** 振る舞いの定義

**値** オブジェクト．メンバ名に振る舞い名，値に振る舞いの定義をオブジェクトで記述する．その際のオブジェクトのメンバの説明は以下の通りである．

DisplayName

**説明** 振る舞いの表示名．主に GUI 表示の際に用いられる

**値** 文字列

Arguments

**説明** 振る舞いの引数

**値** オブジェクト．メンバ名に引数名，値に引数の型を記述する．

### 3.2.3 リソースファイル

リソースファイルには，主に，標準形式トレースログに登場するリソースの定義を記述する．他にも，時間の単位や時間の基数，適用する変換ルール，リソースヘッダ，可視化ルールを定義する．リソースの定義には，名前とリソースタイプ，必要があれば属性の初期値を記述する．

表3.7にリソースファイルの例を示す．リソースファイルは1つのオブジェクトで構成され，TimeScale，TimeRadix，ConvertRules，VisualizeRules，ResourceHeaders，Resources の 6 つのメンバを持つ．以下にそれぞれのメンバについて説明する．

```
{
  "TimeScale" :"us",
  "TimeRadix" :10,
  "ConvertRules"   :["asp"],
  "VisualizeRules" :["toppers","asp"],
  "ResourceHeaders":["asp"],
  "Resources":{
    "TASK1":{
      "Type":"Task",
      "Color":"ff0000",
      "Attributes":{
        "id"    :1,
        "atr"   :"TA_NULL",
        "pri"   :10,
        "exinf" :1,
        "task"  :"task",
        "stksz" :4096,
        "state" :"DORMANT"
      }
    },
    "TASK2":{
      "Type":"Task",
      "Color":"00ff00",
      "Attributes":{
        "id"    :4,
        "atr"   :"TA_ACT",
        "pri"   :5,
        "exinf" :0,
        "task"  :"task",
        "stksz" :4096,
        "state" :"RUNNABLE"
      }
    }
  }
}
```

表 3.7: リソースファイルの例

TimeScale
: **説明** 時間の単位
**値** 文字列

TimeRadix
: **説明** 時間の基数
**値** 数値

ConvertRules
: **説明** 適用する変換ルールのターゲット．複数のターゲットを指定可能
**値** 文字列の配列

VisualizeRules
**説明** 適用する可視化ルール．複数のターゲットを指定可能
**値** 文字列の配列

ResourceHeaders
**説明** Resources で定義されるリソースのリソースタイプを定義しているターゲット．複数のターゲットを指定可能
**値** 文字列の配列

Resources
**説明** リソースを定義
**値** オブジェクト．メンバ名にリソースの名前，値にリソースの定義をオブジェクトで記述．値として与えるオブジェクトで使えるメンバは以下のとおりである．

Type
**説明** 必須項目である．リソースタイプ名を記述
**値** 文字列

Color
**説明** リソース固有の色を指定．可視化表示の際に用いられる
**値** 文字列．RGB を各 8bit で表現したものを 16 進法表記で記述

Attributes
**説明** 属性の初期値．指定できる属性はリソースタイプで定義されているものに限る
**値** オブジェクト．メンバ名に属性名，値に属性の初期値を記述

### 3.2.4 変換ルールファイル

変換ルールファイルには，ターゲットとなるトレースログを標準形式トレースログに変換するためのルールが記述される．表 3.8 に，変換ルールファイルの例を示す．

変換ルールファイルは，1 つのオブジェクトで構成され，各メンバにターゲット毎の変換ルールを記述する．メンバ名にターゲット名を記述し，値としてそのターゲットが出力するトレースログを標準形式へ変換するためのルールをオブジェクトとして記述する．そのオブジェクトのメンバ名には，標準形式へ変換される対象となるトレースログを正規表現を用いて記述し，値には出力する標準形式トレースログを記述する．この際，値を文字列として記述すれば 1 行を，文字列の配列として記述すれば複数行を出力することができる．また，値としてオブジェクトを記述することで，そのメンバ名に出力する条件を記述し，値に出力する標準形式トレースログを記述すれば，条件が真のときのみ出力するように定義できる．また，このときの配列やオブジェクトはネストして記述することができる．

表 3.8 の例を用いて具体的な説明を行う．3 行目，13 行目，22 行目，26 行目が検索するトレースログの正規表現である．これらの正規表現に一致するトレースログが

```
1  {
2    "asp":{
3      "\[(?<time>\d+)\] dispatch to task (?<id>\d+)\.":[
4        {
5          "$EXIST{[${time}]Task(state==RUNNING)}":[
6            "[${time}]$RES_NAME{[${time}]Task(state==RUNNING)}.preempt()",
7            "[${time}]$RES_NAME{[${time}]Task(state==RUNNING)}.state=RUNNABLE"
8          ]
9        },
10       "[${time}]$RES_NAME{Task(id==${id})}.dispatch()",
11       "[${time}]$RES_NAME{Task(id==${id})}.state=RUNNING"
12     ],
13     "\[(?<time>\d+)\] task (?<id>\d+) becomes (?<state>[^\.]+)\.":[
14       {
15         "$ATTR{[${time}]Task(id==${id}).state}==DORMANT && ${state}==RUNNABLE"
16           :"[${time}]$RES_NAME{Task(id==${id})}.activate()",
17         "$ATTR{[${time}]Task(id==${id}).state}==RUNNING && ${state}==DORMANT"
18           :"[${time}]$RES_NAME{Task(id==${id})}.exit()",
19       },
20       "[${time}]$RES_NAME{Task(id==${id})}.state=${state}"
21     ],
22     "\[(?<time>\d+)\] enter to (?<name>\w+)( (?<args>.+))?\.?":{
23       "$EXIST{[${time}]Task(state==RUNNING)}"
24         :"[${time}][${time}]Task(state==RUNNING).enterSVC(${name},${args})"
25     },
26     "\[(?<time>\d+)\] leave to (?<name>\w+)( (?<args>.+))?\.?":{
27       "$EXIST{[${time}]Task(state==RUNNING)}"
28         :"[${time}][${time}]Task(state==RUNNING).leaveSVC(${name},${args})"
29     }
30   }
31 }
```

表 3.8: 変換ルールファイルの例

見つかったとき，対応する値の標準形式トレースログが出力される．3 行目，13 行目の正規表現に一致した場合は，標準形式トレースログが配列として与えられていて，複数の標準形式トレースログを出力する可能性がある．3 行目の正規表現に一致した場合は，10 行目，11 行目は必ず出力され，6 行目，7 行目の標準形式トレースログは 5 行目の条件が真の場合に出力される．13 行目の正規表現に一致した場合は，18 行目は必ず出力され，16 行目，18 行目の標準形式トレースログはそれぞれ 15 行目，17 行目の条件が真の場合に出力される．22 行目，26 行目の正規表現に一致した場合は，標準形式トレースログがオブジェクトとして与えられていて，それぞれ 23 行目，27 行目の条件が真の場合のみ標準形式トレースログを出力する．

検索するトレースログの正規表現において，名前付きグループ化構成体を用いると，入力文字列中の部分文字列をキャプチャすることができ，標準形式トレースログの出力や条件判定の際に使用できるようになる．名前付きグループ化構成体は”(?<*name*>*regexp*)”と記述する．このとき，*regexp* で表現される正規表現にマッチする部分文字列が *name* をキーとしてキャプチャされる．キャプチャされた文字列は，キーを用いて”${*name*}”と記述することで呼び出せる．また，名前付きでないグループ化構成体”(*regexp*)”を用いることもでき，その際は”$*n*”で呼び出せる．*regexp* に一致した部分文字列に 1 か

ら順に番号がつけられ，これを呼び出す際の番号 $n$ として用いる．

### 置換マクロ

標準形式トレースログをオブジェクトとして記述することで出力を条件で制御できることを上記で述べたが，条件判定の際に置換マクロを用いることで特定リソースの有無や数，属性の値を得ることができる．また，置換マクロは，条件判定のときだけでなく，出力する標準形式トレースログの記述にも用いることができる．置換マクロは”$*name*{*common-tracelog-syntax*}”という形式で記述する．*name* は置換マクロ名であり，*common-tracelog-syntax* は標準形式トレースログの文字列である．*common-tracelog-syntax* にはリソースや属性が指定され，時刻の指定も可能である．利用できる置換マクロは以下の通りである．

$EXIST{*resource*}

指定されたリソース *resource* が存在すれば true，存在しなければ false に置換される．リソースがリソース名ではなく，リソースタイプと属性の条件で記述されることを想定している．

**例** 時刻 1000 に属性 state の値が RUNNING であるリソースタイプ Task のリソースが存在する場合

入力
```
$EXIST{[1000]Task(state==RUNNING)}
```

出力
```
true
```

$COUNT{*resource*}

指定されたリソース *resource* の数に置換される．リソースがリソース名ではなく，リソースタイプと属性の条件で記述されることを想定している．

**例** 時刻 1000 に属性 state の値が WAITING であるリソースタイプ Task のリソースが 3 つ存在する場合

入力
```
$COUNT{[1000]Task(state==WAITING)}
```

出力
```
3
```

$ATTR{*attribute*}

指定された属性 *attribute* の値に置換される．リソースをリソースタイプと属性の条件で記述する場合は，条件に一致するリソースが 1 つになるようにしなければならない．

**例** 時刻 1000 にリソース MAIN_TASK の属性 state の値が WAITING である場合

入力
```
$ATTR{[1000]MAIN_TASK.state}
```

出力
```
WAITING
```

`$RES_NAME{`*resource*`}`

指定されたリソース *resource* の名前に置換される．リソースをリソースタイプと属性の条件で記述する場合は，条件に一致するリソースが 1 つになるようにしなければならない．

**例** 属性 `id` の値が `1` であるリソースタイプ `Task` のリソースの名前が MAIN_TASK のとき

入力
```
$RES_NAME{Task(id==1)}
```

出力
```
MAIN_TASK
```

`$RES_DISPLAYNAME{`*resource*`}`

指定されたリソース *resource* の表示名に置換される．リソースの表示名はリソースファイルで定義される．リソースをリソースタイプと属性の条件で記述する場合は，条件に一致するリソースが 1 つになるようにしなければならない．

**例** 属性 `id` の値が `1` であるリソースタイプ `Task` のリソースの表示名が”メインタスク”のとき

入力
```
$RES_DISPLAYNAME{Task(id==1)}
```

出力
```
メインタスク
```

`$RES_COLOR{`*resource*`}`

指定されたリソース *resource* の色に置換される．リソースの色はリソースファイルで定義される．リソースをリソースタイプと属性の条件で記述する場合は，条件に一致するリソースが 1 つになるようにしなければならない．

**例** 属性 `id` の値が `1` であるリソースタイプ `Task` のリソースの色が赤 (ff0000) のとき

入力
```
$RES_COLOR{Task(id==1)}
```

出力
```
ff0000
```

## 3.3 図形データの生成

標準形式変換プロセスを経て得られた標準形式トレースログは，可視化ルールを適用され図形データを生成する．ここで，図形データとは，ワールド変換が行われた全

図形のデータを指す．可視化ルールは可視化ルールファイルとして与えられ，適用する可視化ルールはリソースファイルに記述する．

図形データの生成方法は，標準形式トレースログを一行ずつ可視化ルールのイベント期間と一致するか判断し，一致した場合にその可視化ルールの表示期間をワールド変換先の領域として採用しワールド変換することで行われる．

### 3.3.1 可視化ルールファイル

可視化ルールファイルには，可視化ルールと，図形の定義を記述する．可視化ルールファイルは，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ターゲット毎の変換ルールを記述する．メンバ名にターゲット名を記述し，値としてオブジェクトを与え，そのオブジェクトに可視化ルールと図形の定義を記述する．

#### 図形の定義

図形の定義は，2.3.1 小節にて述べた抽象化した図形を形式化したものである．

図形の定義は Shapes というメンバ名の値にオブジェクトとして記述する．このオブジェクトのメンバ名には図形の名前を記述する．そして，その値に図形の定義を基本図形の定義の配列として与える．

表3.9にtoppersをターゲットとする図形を定義した例を示す．例では，runningShapes と runnableShapes，svcShapes の 3 つの図形を定義している．runningShapes と runnableShapes は 1 つの基本図形で構成され，svcShapes は 3 つの基本図形から構成される．

基本図形の定義に用いるメンバは，基本図形の形状により異なる．すべての形状に共通なメンバの説明を以下に示す．

Type
: **説明** 図形の形状．必須である．
**値** 文字列（"Rectangle"：長方形，"Line"：線分，"Arrow"：矢印，"Polygon"：多角形，"Pie"：扇形，"Ellipse"：楕円形，"Text"：文字列のいずれか）

Size
: **説明** 図形のサイズ．省略した場合，値は"100%,100%"となる
**値** サイズ指定形式の文字列

Location
: **説明** 図形の位置．省略した場合，値は"0,0"となる
**値** 位置指定形式の文字列

```
{
  "toppers":{
    "Shapes":{
      "runningShapes":[
        {
          "Type":"Rectangle",
          "Size":"100%,80%",
          "Pen":{"Color":"ff00ff00","Width":1},
          "Fill":"6600ff00"
        }
      ],
      "runnableShapes":[
        {
          "Type":"Line",
          "Points":["l(0),80%","r(0),80%"],
          "Pen":{"Color":"ffffaa00","Width":1}
        }
      ],
      "svcShapes":[
        {
          "Type":"Rectangle",
          "Size":"100%,40%",
          "Pen":{"Color":"${ARG0}","Width":1, "DashStyle":"Dash"},
          "Fill":"${ARG0}",
          "Alpha":100
        },
        {
          "Type":"Text",
          "Size":"100%,40%",
          "Font":{"Align":"TopLeft", "Size":7},
          "Text":"${ARG1}"
        },
        {
          "Type":"Text",
          "Size":"100%,40%",
          "Font":{"Align":"BottomRight", "Size":7},
          "Text":"return ${ARG2}"
        }
      ]
    }
  }
}
```

表 3.9: 可視化ルールファイルで図形を定義した例

Area

**説明** 図形の表示領域．サイズと位置を同時に指定する．省略した場合は，サイズに Size の値が，位置に Location の値が設定される．

**値** 文字列 2 つの配列．1 つ目の要素は位置指定形式，2 つ目の要素はサイズ指定形式を記述する

Offset

**説明** 図形のオフセット．省略した場合，値は"0,0"となる

**値** 位置指定形式の文字列

図形のサイズを指定する形式として，サイズ指定形式（ShapeSize）を次のように定めた．

```
ShapeSize = Width,",",Height
Width = /-?([1-9][0-9]*)?[0-9](\.[0-9]*)?(%|px)?/
Height = /-?([1-9][0-9]*)?[0-9](\.[0-9]*)?(%|px)?/
```

2.3.1 小節において，図形の大きさの指定方法として，絶対指定と相対指定の 2 つの方法を用いることができるとした．これらは px と%という単位を用いることで指定する．px が絶対指定であり，%が相対指定である．単位を省略した場合は絶対指定されたものとして解釈される．

また，図形の位置を指定する形式として，位置指定形式 (ShapeLocation) を次のように定めた．

```
ShapeLocation = X,",",Y
X = ("l"|"c"|"r"),"(",Value,")"|Value;
Y = ("t"|"m"|"b"),"(",Value,")"|Value;
Value = /-?([1-9][0-9]*)?[0-9](\.[0-9]*)?(%|px)?/
```

図形の位置も，サイズと同じように絶対指定と相対指定の両方で指定することができる．また，指定する際に基準とする位置を"*base*(*value*)"として指定できるようにした．この指定を基準指定と呼ぶ．ここで，*base* は，X の指定の場合，l か c か r であり，それぞれ領域の左端，横方向の中央，右端を指す．また，Y の指定の場合は，t か m か b であり，それぞれ領域の上端，縦方向の中央，下端を指す．

相対指定の際に基準指定することにより，同じ位置を様々な記述方法を用いて指定できる．例えば，l(100%) と r(0%)，c(50%) は領域の右端を指定し，l(50%) と r(-50%)，c(0%) は領域の横方向の中央を指す．同じように b(100%) と t(0%)，m(50%) は下端を指定し，b(50%) と t(-50%)，m(50%) は縦方向の中央を指す．

基準指定は，原点を基準とした指定しか行えない絶対指定のために導入した．基準指定が行えない場合，"右端から 5 ピクセル"という指定ができなくなってしまう．これは，図形をデバイス座標系へマッピングしない限り，図形の大きさをピクセル単位

で知ることができないからである．基準指定を行えば”右端から5ピクセル”という指定はr(5px)と記述することで行える．

図形の位置とサイズは描画領域を表し，この描画領域に内接するように図形が描画される．楕円形や扇形はこの描画領域の中心を円の中心として描かれる．

図の形状が線分(Line)，矢印(Arrow)のときは始点と終点，多角形(Polygon)は各頂点をPointsというメンバを用い座標を指定する．Pointsの説明を以下に述べる．

Points
: **説明** 線分(Line)，矢印(Arrow)は始点と終点，多角形(Polygon)は各頂点を指定する．これらの形状の時は必須である
: **値** 位置指定形式の文字列の配列

図の形状が扇形(Pie)のとき，扇形の開始角度，開口角度をArcというメンバを用いて定義する．Arcの説明を以下に述べる．

Arc
: **説明** 扇形の開始角度，開口角度を指定する．省略した場合は”0,90”となる
: **値** 数値2つの配列．1つ目の要素が開始角度，2つ目の要素が開口角度である

図の形状が文字列(Text)の場合，Textというメンバで描画する文字列を，Fontでフォントの設定を指定できる．Textの説明を以下に述べる．

Text
: **説明** 描画する文字列を指定する．省略した場合は文字列を描画しない
: **値** 文字列．

Font
: **説明** 描画する文字列の色，透明度，フォント，サイズ，スタイル，領域内での位置を指定する．
: **値** オブジェクト．オブジェクトのメンバとして以下が使える

    Color
    : **説明** 文字列の色．省略した場合，”000000”となる
    : **値** 文字列．ARGBあるいはRGBを各8bitで表現したものを16進法表記で記述する．Aは透明度である．

    Family
    : **説明** 文字のフォントを指定する．省略した場合，システムでSansSerifとして設定してあるフォント名になる
    : **値** 文字列．システムにインストールされているフォント名でなければならない．

    Style
    : **説明** 文字列のスタイルを指定する．
    : **値** 文字列．"Bold":太字，"Italic":斜体，"Regular":標準，"Strikeout":中央線付き，"Underline"：下線付きのいずれか

Alpha
**説明** 文字列の透明度．省略した場合，255 になる
**値** 数値．0〜255 の値

Size
**説明** 文字列のサイズ
**値** 数値．単位はポイントである

Align
**説明** 文字列の領域内での位置
**値** 文字列．"BottomCenter"：下端中央，"BottomLeft"：下端左寄せ，"BottomRight"：下端右寄せ，"MiddleCenter"：中段中央，"MiddleLeft"：中段左寄せ，"MiddleRight"：中段右寄せ，"TopCenter"：上端中央，"TopLeft"：上端左寄せ，"TopRight"：上端右寄せのいずれか

図の形状が文字列 (Text) 以外の場合，Fill というメンバで塗りつぶしの色，Alpha で塗りつぶしの透明度，Pen で図形の縁取り線を指定できる．これらの説明を以下に述べる．

Fill
**説明** 塗りつぶしの色．省略した場合，”ffffff”となる
**値** 文字列．ARGB または RGB を各 8bit で表現したものを 16 進法表記で記述する．A は透明度である．

Alpha
**説明** 塗りつぶしの色の透明度．省略した場合，255 となる
**値** 数値．0〜255 の値

Pen
**説明** 縁取り線を指定する．
**値** オブジェクト．オブジェクトのメンバとして以下が使える

Color
**説明** 線の色．省略した場合，”000000”となる
**値** 文字列．ARGB あるいは RGB を各 8bit で表現したものを 16 進法表記で記述する．A は透明度である

Alpha
**説明** 文字列の透明度．省略した場合，255 になる
**値** 数値．0〜255 の値

Width
**説明** 線の幅．省略した場合は，1 となる
**値** 数値

DashStyle
**説明** 線種の指定．省略した場合，”Solid”となる
**値** 文字列．"Dash"：ダッシュ，"DashDot"：ダッシュとドットの繰り返し，"DashDotDot"：ダッシュと 2 つのドットの繰り返し，"Dot"：ドット，"Solid"：実線，のいずれか

2.3.1 小節において，図形を参照する際に引数を与えることができ，任意の属性に引数の値を割り当てることができると述べた．そのため，与えられた引数は，”ARG$n$”で参照できることとした．

表 3.9 において，図形 svcShapes は 3 つの基本図形 (四角形，文字列，文字列) で構成されており，四角形の Pen の Color と Fill が”${ARG0}”，文字列のそれぞれの Text が”${ARG1}”，”${ARG2}”となっている．これは，図形 svcShapes が 3 つの引数をもつことを示している．たとえば，可視化ルールにおいて図形 svcShapes を参照する場合は，svcShapes(ff0000,act_tsk,ercd=0) のように記述する．この場合，1 つ目の引数として”ff0000”，2 つ目の引数として”act_tsk”，3 つ目の引数として”ercd=0”を与えており，それぞれが．${ARG0}，${ARG1}，${ARG2}と置き換えられる．引数をとらない図形を参照する場合は，図形の名前のみで参照できる．

このように，図形に引数を与えることで，図形の属性を外部から指定できるようになり，属性違いのために組み合わせ毎に図形を多量に定義しなければならないような状況を避けることができる．

### 可視化ルールの定義

可視化ルールの定義は，2.3.1 小節にて述べた可視化ルールを形式化したものである．

可視化ルールの定義は VisualizeRules というメンバ名の値にオブジェクトとして記述する．このオブジェクトのメンバ名には可視化ルールの名前を記述する．そして，その値に可視化ルールの定義を記述する．

表 3.10 に toppers をターゲットとする可視化ルールを定義した例を示す．例では，taskStateChange と callSvc の 2 つの可視化ルールを定義している．

可視化ルールの定義に用いるメンバは DisplayName，Target，Shapes である．これらについて，以下に詳述する．

DisplayName
**説明** 可視化ルールの表示名．省略した場合は可視化ルールの名前になる
**値** 文字列

Target
**説明** 可視化ルールを適用するリソースタイプ
**値** 文字列

```
{
  "toppers":{
    "VisualizeRules":{
      "taskStateChange":{
        "DisplayName":"状態遷移",
        "Target":"Task",
        "Shapes":{
          "stateChangeEvent":{
            "DisplayName":"状態",
            "From":"${TARGET}.state",
            "To"  :"${TARGET}.state",
            "Figures":{
              "${FROM_VAL}==RUNNING" :"runningShapes",
              "${FROM_VAL}==RUNNABLE":"runnableShapes"
            }
          },
          "activateHappenEvent":{
            "DisplayName":"起動",
            "When"   :"${TARGET}.activate()",
            "Figures":"activateShapes"
          }
        }
      },
      "callSvc":{
        "DisplayName":"システムコール",
        "Target":"Task",
        "Shapes":{
          "callSvcEvent":{
            "DisplayName":"システムコール",
            "From":"${TARGET}.enterSVC()",
            "To"  :"${TARGET}.leaveSVC(${FROM_ARG0})",
            "Figures":{
              "${FROM_ARG0}==slp_tsk||${FROM_ARG0}==dly_tsk"
                :"svcShapes(ff0000,${FROM_ARG0}(${FROM_ARG1}),${TO_ARG1})",
              "${FROM_ARG0}!=slp_tsk&&${FROM_ARG0}!=dly_tsk"
                :"svcShapes(ffff00,${FROM_ARG0}(${FROM_ARG1}),${TO_ARG1})"
            }
          }
        }
      }
    }
  }
}
```

表 3.10: 可視化ルールファイルで可視化ルールを定義した例

Shapes

**説明** 図形群の定義を記述する

**値** オブジェクト．メンバ名に図形群の名前，メンバの値として図形群の定義をオブジェクトとして与える．その際，以下のメンバが使える

DisplayName

**説明** 図形群の表示名．省略した場合は図形群の名前になる

**値** 文字列

From

**説明** 図形群を構成する図形のワールド変換に適用される開始イベント

**値** イベント指定形式文字列

To

**説明** 図形群を構成する図形のワールド変換に適用される終了イベント

**値** 標準形式トレースログの形式の文字列

When

**説明** 図形群を構成する図形のワールド変換に適用されるイベント．開始時刻と終了時刻が同じ場合に用いる

**値** イベント指定形式文字列

Figures

**説明** 図形群を構成する図形の定義

**値** 文字列，配列，オブジェクトのいずれか．文字列の場合は図形の参照，オブジェクトの場合はメンバ名に条件，値に図形の参照を記述する．配列の要素には，これら文字列かオブジェクトを記述できる．オブジェクトと配列はネストして記述することができる．

可視化ルールのイベント期間の指定には，イベント期間として適用するトレースログを，条件として必要な情報を標準形式トレースログの形式で記述する．たとえば，リソースの属性が変わったときをイベント期間に指定したいときは，標準形式トレースログの形式で属性名まで記述すればよい．また，属性の値が特定の値に変わったときをイベント期間に指定したいときは，標準形式トレースログの形式で値まで記述すればよい．リソースの振る舞いをイベント期間として指定したい場合は，標準形式トレースログの形式で振る舞い名と”()”を記述すればよく，特定の引数をとるときに条件を絞りたいときはその引数を”()”内に記述すればよい．

イベント期間の指定でリソースを指定する際，Target 指定がしてある場合は，${TARGET} という置換マクロで，リソースファイルで定義したリソースタイプ Target の各リソースの名前に適時置換することができる．

表 3.10 の taskStateChange を例に説明する．Target には Task が指定してある．このとき，リソースファイルにリソースタイプ Task のリソースとして Task1，Task2，Task3 が定義してある場合は，From と To の${TARGET}はこれらのリソース名に置換され，Task1.state，Task2.state，Task3.state となる．

Target 指定がないとき，To を指定する際に From で一致したトレースログのリソースを指定したい場合は，置換マクロ${FROM_TARGET}を記述すればよい．また，From で属性値の変化イベントを指定したとき，From で一致したトレースログの属性の値を To で指定したいときは置換マクロ${FROM_VAL}を記述すればよい．同じように，From で振る舞いの発生イベントを指定したとき，From で一致したトレースログの振る舞いの引数を To で指定したいときは置換マクロ${FROM_ARG$n$}を記述すればよい．$n$ には何番目の引数かを 0 からで記述する．

これらの置換マクロは図形の参照を記述する際にも利用できる．To で一致したトレースログの属性値変化の値は${TO_VAL}，振る舞いの引数は${TO_ARG$n$}という置換マクロで指定できる．また，Target 指定がないときに To で一致したトレースログのリソースを指定したいときは置換マクロ${TO_TARGET}を用いればよい．イベント期間指定が When のときは，一致したトレースログのリソースを${TARGET}，属性の値を${VAL}，振る舞いの引数を${ARG$n$}で指定する．

## 3.4 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詳述する．

### 3.4.1 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．

マーカーとマーカーの間には，その間の時間が表示されるので，ソフトウェアの計測を行うことができる．マーカーには名前を付けることができ，色の指定が可能である．また，マーカーはマウス操作で選択することができ，拡大縮小などの各種操作に利用される．マーカーは階層構造で管理し，階層ごとに表示の切り替え，マーカー間時間の表示などを行うことができる．

### 3.4.2 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使い勝手に大きく影響するため，TLV では目的や好み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，下へスクロ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．

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い，微調整するのに適している．左キーで表示領域を左に移動し，右キーで右に移動する．上キーでカーソル位置，または選択されたマーカーを中心に表示領域を縮小し，下キーで拡大する．

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．

### 3.4.3 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，マウスにより拡大縮小，移動の制御を行うことができる．

### 3.4.4 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でマウスを移動すると，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．

### 3.4.5　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により変更，追加することができるが，それらの表示を可視化ルールやリソースを単位で切り替えることができる．これらの操作は，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える．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．

# 第4章　トレースログ可視化ツール TraceLogVisualizer の利用

## 4.1　シングルコアプロセッサ用RTOSのトレースログの可視化

可視化するシングルコアプロセッサ用 RTOS として，TOPPERS/ASP カーネルを用いた．TOPPERS/ASP カーネルは，標準でトレースログを取得する機能を搭載しており，カーネルの動作開始と終了，処理単位の実行開始と終了，タスク状態の変化，ディスパッチャの実行開始と終了，サービスコールの入口と出口といった情報をトレースログ取得できる．処理単位とは，割込みハンドラ，割込みサービスルーチン，周期ハンドラ，アラームハンドラ，CPU 例外ハンドラ，タスク例外処理ルーチンを指す．TOPPERS/ASP カーネルのトレースログの例は表 3.5 で示している．

任意の形式をもつトレースログを TLV で可視化表示するために必要な作業は，可視化表示する項目の決定，リソースヘッダファイルの記述，変換ルールファイルの記述,，可視化ルールファイルの記述である．また，TLV では，トレースログに含まれるリソースをリソースファイルとして定義し，読み込まなければならないため，リソースファイルを記述する作業も必要になる．

### 4.1.1　可視化表示する項目の決定

可視化表示する項目は，タスクの状態遷移，システムコールとした．

タスクの状態遷移の可視化表現は，実行状態を緑色の四角形，実行可能状態を黄色い線分，待ち状態を赤い線，起動を上矢印，終了を下矢印とする．図 4.1 にタスクの状態遷移の可視化表現の例を示す．

システムコールの可視化表現は，赤色の四角形で，枠線を点線，高さを待ち状態の線と同じ高さとする．このとき，システムコール名，システムコールの引数を四角形の上左隅，返値であるエラーコードを下右隅に文字列で出力するとする．図 4.2 にシステムコールの可視化表現の例を示す．

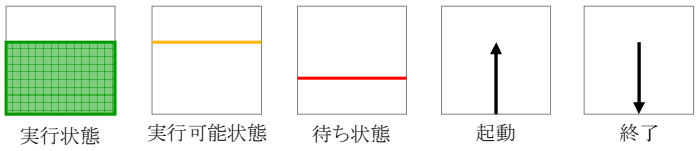


図 4.1: タスクの状態遷移の可視化表現例

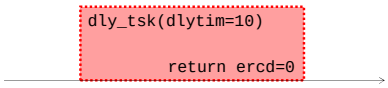


図 4.2: システムコールの可視化表現例

### 4.1.2 リソースヘッダファイルの記述

前述の可視化表示はタスクの挙動に関してであるので，リソースタイプとしてはタスクを定義すればよい．表 3.5 によると，トレースログ内においてタスクはタスク ID を指定して記録されている．また，状態の遷移を可視化表示したいため，タスクの属性としては，タスク ID と状態を考慮すればよい．

前述の可視化表現を描画するためには，以下のようなイベントを標準形式トレースログで出力しなければならない．そのため，それぞれを振る舞いとして定義する．

- システムコールに入った
- システムコールから出た
- タスクが起動した
- タスクが終了した

タスクのリソースタイプを `Task` として表 4.1 のようにリソースヘッダファイルに定義した．前述のシステムコールの可視化表現では，システムコールの名前と引数，返値のエラーコードを文字列として出力したいため `enterSVC` と `leaveSVC` に引数としてそれらを指定するようにした．

### 4.1.3 変換ルールファイルの記述

表 3.5 で示した TOPPERS/ASP カーネルのトレースログを標準形式トレースログに変換するための変換ルールファイルを記述する．

タスクの状態変化の可視化表示には，リソースタイプ `Task` の属性 `state` が変わったという属性変化イベントと，振る舞い `activate()` と `exit()` が発生したという振る舞いイベントをイベント期間として指定すればよい．

```
{
  "asp":
  {
    "Task":{
      "DisplayName":"タスク",
      "Attributes":{
        "id":{
          "VariableType"  :"Number",
          "DisplayName"   :"ID",
          "AllocationType":"Static",
          "CanGrouping"   :false
        }
        "state":{
          "VariableType"  :"String",
          "DisplayName"   :"状態",
          "AllocationType":"Dynamic",
          "CanGrouping"   :false,
          "Default"       :"DORMANT"
        }
      },
      "Behaviors"  :{
        "activate":{
          "DisplayName":"起動"
        }
        "exit":{
          "DisplayName":"終了"
        }
        "enterSVC":{
          "DisplayName":"サービスコールに入る",
          "Arguments"  :{
            "name":"String",
            "args":"String"
          }
        },
        "leaveSVC":{
          "DisplayName":"サービスコールから出る",
          "Arguments"  :{
            "name":"String",
            "args":"String"
          }
        }
      }
    }
  }
}
```

表 4.1: TOPPERS/ASP カーネル用リソースヘッダファイル

TOPPERS/ASP カーネルのトレースログは，タスクの状態遷移を表 4.2 のような形式で表現している．*time* は時間であり，時間の単位はマイクロ秒である．*taskId* はタスク ID を，*state* は遷移後の状態を表している．この形式に一致した際に出力するべき標準形式トレースログを表 4.3 に示す．

しかし，TOPPERS/ASP カーネルでは，実行状態と実行可能状態を内部では状態として区別しておらず，RUNNING への遷移と，RUNNABLE から RUNNING への遷移を上記の形式で記録しない．そのため，上記の形式のトレースログでタスクの状態遷移のすべてを標準形式トレースログで出力するのは不可能である．

TOPPERS/ASP カーネルでは，タスクが実行状態になったかどうかを表 4.4 のような形式のトレースログで記録する．これは，ディスパッチャから出てタスク ID が *taskId* のタスクの実行に遷移したことを表している．この形式のトレースログによりタスクが RUNNING へ遷移したことを標準形式トレースログで出力する．また，RUNNABLE から RUNNING への遷移は，この形式のトレースログに一致した際に，時刻がそのトレースログの *time* のときに状態が RUNNING のタスクが存在する場合に，そのタスクを RUNNABLE へ遷移したことを標準形式トレースログで出力することで行う．

タスクの起動の標準形式トレースログは，表 4.3 の形式のトレースログに一致した際に，時刻 *time* のときのタスク ID が *taskId* のタスクの状態が DORMANT から RUNNABLE になったときに表 4.5 で示す形式で出力ればよい．おなじように，タスクの終了は時刻 *time* のときのタスク ID が *taskId* のタスクの状態が RUNNING から DORMANT になったときに表 4.6 で示す形式で出力ればよい．

TOPPERS/ASP カーネルのトレースログは，システムコールに入ったことを表 4.7 で示す形式で記録する．また，システムコールから出たことを表 4.8 で示す形式で記録する．*svcName* はシステムコール名を，*args* はシステムコールの引数を，*ercd* はシステムコールの返値であるエラーコードを示す．これらの形式のトレースログに一致した場合，実行状態のタスクで，enterSVC(*svcName*,*args*)，leaveSVC(*svcName*,*ercd*) の振る舞いイベントを標準形式トレースログで出力すればよい．そのためには，一致したトレースログに含まれていない実行状態のタスクを知る必要があるが，置換マクロを用いることで取得することができる．

以上から，TOPPERS/ASP カーネルのトレースログを標準形式トレースログに変換するための変換ルールファイルを，表 4.9 のように記述した．表 4.9 に示す変換ルールファイルに従い表 3.3 で示した TOPPERS/ASP カーネルのトレースログを標準形式トレースログに変換した結果は，表 3.4 のようになる．

### 4.1.4 可視化ルールファイルの記述

4.1.1 小節で決めた，項目毎の可視化表現と，3.3.1 小節で定義した可視化ルールファイルの記述方法に従い，図形のデータを表 4.10 のように定義した．また，可視化ルールを表 4.11 のように定義した．

| [*time*] task *taskId* becomes *state*. |
|---|

表 4.2: TOPPERS/ASP カーネルのトレースログにおけるタスクの状態遷移を表す形式

| [*time*] Task(id==*taskId*).state=*state* |
|---|

表 4.3: タスクの状態遷移を表す標準形式トレースログ

| [*time*] dispatch to task *taskId*. |
|---|

表 4.4: TOPPERS/ASP カーネルのトレースログにおけるタスクが実行状態になったことを表す形式

| [*time*] Task(id==*taskId*).activate() |
|---|

表 4.5: タスクの起動を表す標準形式トレースログ

| [*time*] Task(id==*taskId*).exit() |
|---|

表 4.6: タスクの終了を表す標準形式トレースログ

| [*time*] enter to *svcName* *args*. |
|---|

表 4.7: TOPPERS/ASP カーネルのトレースログにおけるシステムコールに入ったことを表す形式

| [*time*] leave to *svcName* ercd=*ercd*. |
|---|

表 4.8: TOPPERS/ASP カーネルのトレースログにおけるシステムコールから出たことを表す形式

```
{
  "asp":
  {
    "\[(?<time>\d+)\] dispatch to task (?<id>\d+)\.":[
      {
        "$EXIST{Task(state==RUNNING)}":
          "[${time}]$RES_NAME{Task(state==RUNNING)}.state=RUNNABLE"
      },
      "[${time}]$RES_NAME{Task(id==${id})}.state=RUNNING"
    ],
    "\[(?<time>\d+)\] task (?<id>\d+) becomes (?<state>[^\.]+)\.":[
      {
        "$ATTR{Task(id==${id}).state}==DORMANT && ${state}==RUNNABLE"
          :"[${time}]$RES_NAME{Task(id==${id})}.activate()",
        "$ATTR{Task(id==${id}).state}==RUNNING && ${state}==DORMANT"
          :"[${time}]$RES_NAME{Task(id==${id})}.exit()"
      },
      "[${time}]$RES_NAME{Task(id==${id})}.state=${state}"
    ],
    "\[(?<time>\d+)\] enter to (?<name>\w+)( (?<args>.+))?\.?":
    {
      "$EXIST{Task(state==RUNNING)}"
        :"[${time}]$RES_NAME{Task(state==RUNNING)}.enterSVC(${name},${args})"
    },
    "\[(?<time>\d+)\] leave to (?<name>\w+)( (?<args>.+))?\.?":
    {
      "$EXIST{Task(state==RUNNING)}"
        :"[${time}]$RES_NAME{Task(state==RUNNING)}.leaveSVC(${name},${args})"
    }
  }
}
```

表 4.9: TOPPERS/ASP カーネル用変換ルールファイル

### 4.1.5 トレースログファイルとリソースファイルの生成

TLV では，トレースログに含まれるリソースをリソースファイルとして定義し，読み込まなければならない．TOPPERS/ASP カーネルでは，コンパイル過程においてこのリソースファイルを自動生成することができる．

TOPPERS/ASP カーネルでは，コンパイル過程において，タスクやセマフォなどのカーネルオブジェクトが静的 API により定義されたコンフィギュレーションファイルをコンフィギュレータが読み込み，C 言語ソースコードであるカーネル構成初期化ファイルを生成する．この際，カーネル構成初期化ファイルは，テンプレートファイルの記述に従い生成される．そのため，TLV のリソースファイルの形式に従うテンプレートファイルを記述し，これをコンフィギュレータに読み込ませることで，リソースファイルを生成することができる．

図 4.3 に，TOPPERS/ASP カーネルにおけるトレースログファイルとリソースファ

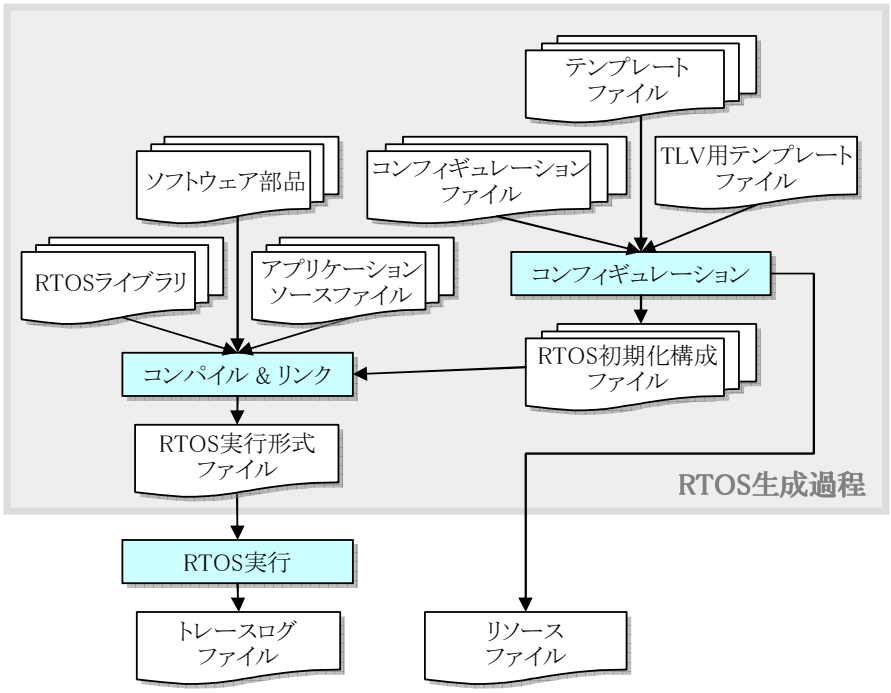


図 4.3: TOPPERS/ASP カーネルにおけるトレースログファイルとリソースファイルの生成過程

イルの生成過程を示す.

### 4.1.6 実行結果

TLV で TOPPERS/ASP カーネルのトレースログファイルと，リソースファイルを読み込み，可視化表示した実行結果のスクリーンショットを図 4.4 に示す.

図 4.4 より，可視化表示する項目が，想定した可視化表現で描画されていることが確認できる．たとえば，Task2 の状態が RUNNABLE，RUNNING，WAITING と遷移し，状態が WAITING なのは，slp_tsk というシステムコールを呼んで自ら待ち状態に遷移したから，ということがわかる.

## 4.2 マルチコアプロセッサ用 RTOS 対応への拡張

前節にて，シングルコアプロセッサ用 RTOS である，TOPPERS/ASP カーネルのトレースログを可視化表示できることを確認した．本節では，これを拡張し，マルチコアプロセッサ用 RTOS である，TOPPERS/FMP カーネル [13] のトレースログの可視化表示を試みる.

TOPPERS/FMP カーネルは，対称型 (SMP) またはそれに近いマルチコアプロセッサに対応し，リアルタイム性と柔軟性を両立させた RTOS である．タスクに対して

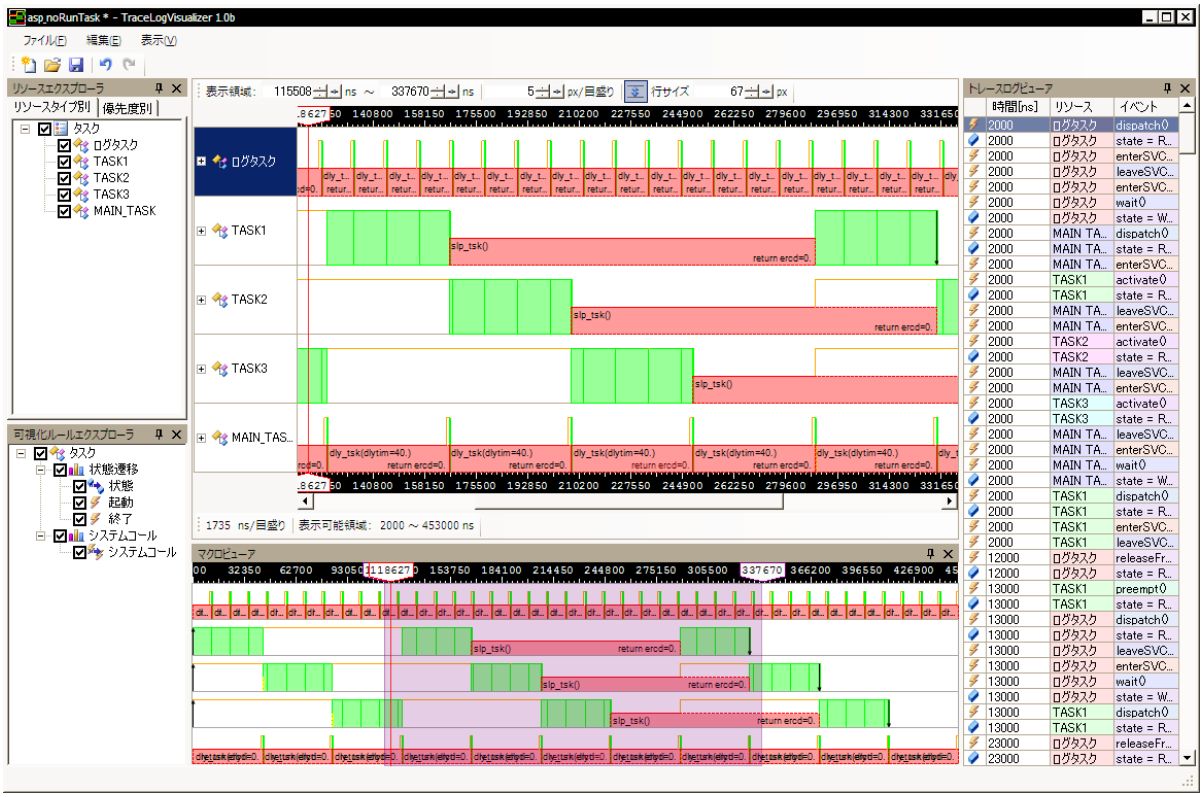


図 4.4: TOPPERS/ASP カーネルのトレースログを可視化した TLV 実行結果のスクリーンショット

コア間を移動させるシステムコールを実装しており，アプリケーションレベルで負荷分散を実装することができる．OS が負荷分散を行わないという仕様から，各コアはシングルコアプロセッサのときと同じスケジューリング方式を用いることができるため．リアルタイム性の確保が容易になる．

TOPPERS/FMP カーネルのトレースログの形式は，ASP カーネルのものに，トレースログを記録したプロセッサ ID を付与した形式になっている．表 4.12 に TOPPERS/FMP カーネルのトレースログの例を示す．

### 4.2.1 可視化表示する項目の追加

TOPPERS/FMP カーネルのトレースログを可視化するにあたり，可視化表示する項目を追加した．

FMP カーネルは，タスクが複数のコアで実行されるため，タスクがどのプロセッサに所属しているのかを可視化表示することにした．可視化表現としては，背景を所属プロセッサ毎に色分けすることにした．ここでは，プロセッサ 1 に所属している間は薄赤，プロセッサ 2 に所属している間は薄青とした．

### 4.2.2 リソースヘッダファイルの修正

所属プロセッサを可視化表現するため，リソースタイプ Task に所属プロセッサという属性を追加する必要がある．TOPPERS/ASP カーネル用のリソースヘッダをコピーし，表 4.13 に示すように修正した．修正内容は，ターゲットを fmp としたこと，Attributes の定義に prcId を追加したことである．

### 4.2.3 変換ルールファイルの修正

FMP カーネルのトレースログは表 4.14 のような形式になっている．*prcId* がプロセッサ ID であり，*tracelog* は ASP カーネルのトレースログの時刻以降と同じである．修正内容として，まず，検索対象となるトレースログの正規表現をプロセッサ ID を考慮して修正する．次に，実行状態のタスクを参照しているリソース記述 (Task(state==RUNNING)) について，プロセッサ ID を条件に追加する．これは，実行状態のタスクが最大プロセッサの数だけ存在するためである．ASP カーネル用の変換ルールファイルを元に記述した，FMP カーネル用の変換ルールファイルの一部を表 4.15 に示す．

### 4.2.4 可視化ルールファイルの記述

ASP カーネル用の可視化ルールファイルは，そのまま用いることができる．これは，可視化ルールファイルがトレースログの形式に依存していないからである．

4.2.1 小節で述べた，所属プロセッサの可視化表示のため，FMP カーネル用に，可視化ルールファイルを追加した．図 4.16 に TOPPERS/FMP カーネル用の可視化ルールファイルを示す．図 4.16 の内容は，まず，Shapes で prcIdShapes という所属プロセッサ ID を示す図形を定義している．表示領域の大きさの四角形で，引数で指定される色を塗りつぶしの色として指定している．また，Aplha で透明度を設定し，薄い色になるようにしている．

次に，VisualizeRules で taskPrcIdChange という名前で所属プロセッサを表示する可視化ルールを定義している．イベント期間としてリソースタイプ Task の所属プロセッサ ID を表す属性 prcId の変化イベントを指定している．その際の図形として，prcIdShapes を指定し，変化後の所属プロセッサ ID(prcId) の値で条件分けをして引数に与える色を変えている．ここでは，プロセッサの数を 4 つまで対応できるようにした，

### 4.2.5 実行結果

以上のファイルの修正・追加を行い，TOPPERS/FMP カーネルのトレースログを TLV で可視化した．図 4.5 にスクリーンショットを示す．

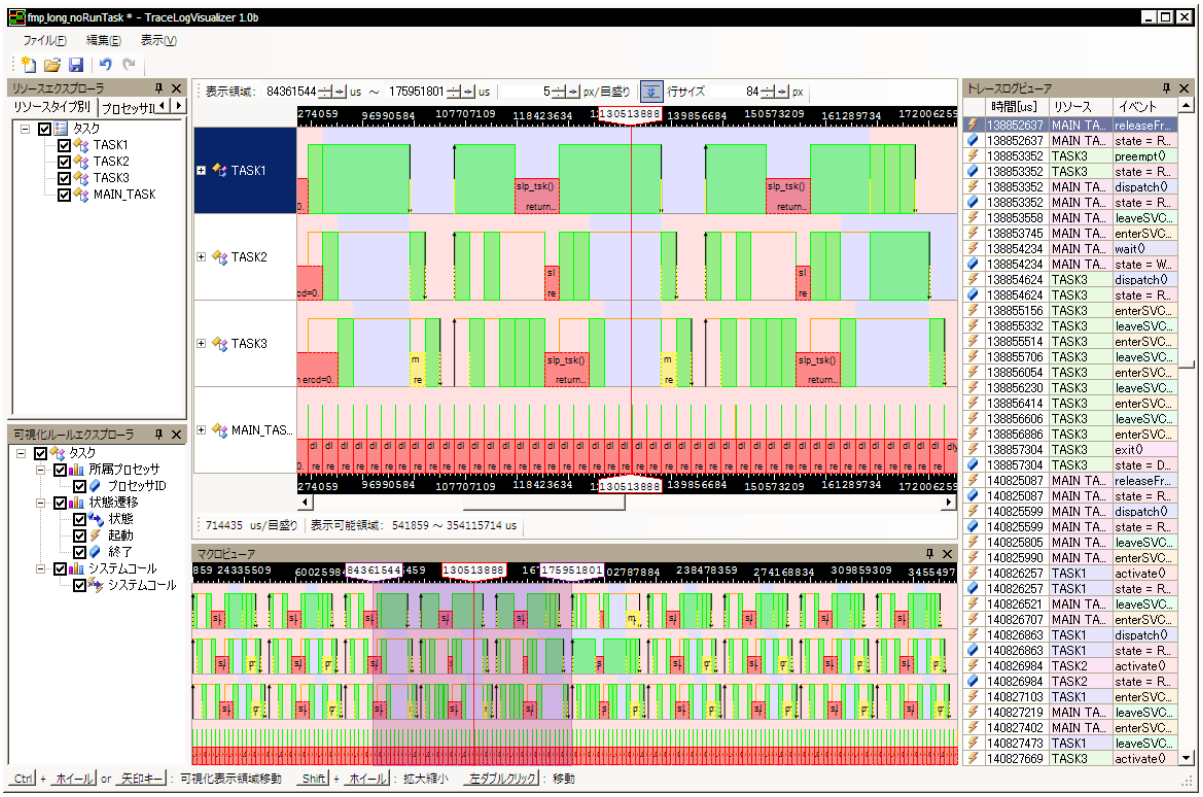


図 4.5: TOPPERS/FMP カーネルのトレースログを可視化した TLV 実行結果のスクリーンショット

図 4.5 より，各タスクの所属プロセッサにより，背景の色が分けて表示されていることがわかる．以上の結果から，シングルコアプロセッサ用 RTOS のトレースログの可視化に必要なファイルをそのまま利用し，マルチコアプロセッサ用 RTOS のトレースログの可視化に必要な可視化ルールファイルを追加するだけで可視化表示項目の追加に対応できることを示し，拡張性があることを示せた．

## 4.3 組込みコンポーネントシステムの可視化

本節では，これら RTOS のトレースログとは形式の異なるトレースログの可視化を行い，汎用性の確認を試みる．

RTOS のトレースログとは形式の異なるトレースログとして，ここでは，組込みコンポーネントシステム TECS(Toppers Embedded Component System)[14] のトレースログを採用した．

組込みコンポーネントシステム TECS とは，組込みに適したソフトウェアの部品化の仕組みで，セルと呼ばれるコンポーネントのインスタンス同士を接続することでソフトウェアを構築する．

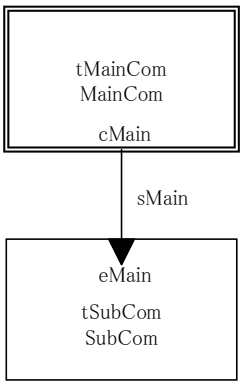


図 4.6: TECS のコンポーネント図

セルは呼び口，受け口を持ち，それぞれシグニチャを持つ．同じシグニチャ同士の呼び口と受け口を接続することができる．シグニチャは関数ヘッダの集合で，複数の関数インタフェースを持つ．図 4.6 に TECS のコンポーネント図を示す．

図 4.6 では，tMainCom というセルタイプ (セルの型) のセル MainCom の，sMain というシグニチャをもつ呼び口 cMain と，セルタイプ tSubCom のセル SubCom の，シグニチャsMain の受け口 eMain を接続していることを示している．ここで，sMain に `void print( [in,string]const char_t *str )` という関数インタフェースが定義してある場合，MainCom の実装で `cMain_print("HelloWorld")` と記述すれば，接続されている SubCom の eMain で実装されている `print("HelloWorld")` が呼び出される．SubCom を他のセルに切り替えることで，MainCom で呼ぶ `print` の実装を容易に変更することができる．

### 4.3.1 可視化表示する項目の決定

TECS のトレースログを可視化表示する項目として，セルの呼び出し関係を採用した．

簡略化のため，セルがシグニチャのどの関数を呼んだかどうかは可視化せず，どのセルが，どのセルを，どのくらいの期間呼び出していたかを可視化表示するとする．セルの呼び出し関係の可視化表現として，図 4.7 のように描画することとした．

### 4.3.2 リソースヘッダファイルの記述

セルをリソースタイプ `Cell` としてリソースヘッダファイルに定義した．表 4.17 に TECS 用リソースヘッダファイルを示す．

属性として，能動セルかどうかを表す `active` を定義した．振る舞いとして，セルの呼び出しを `call`，呼び出し先からのリターンを `callback`，セルが呼び出されるのを `called`，呼び出し元へのリターンを `return` として定義した．

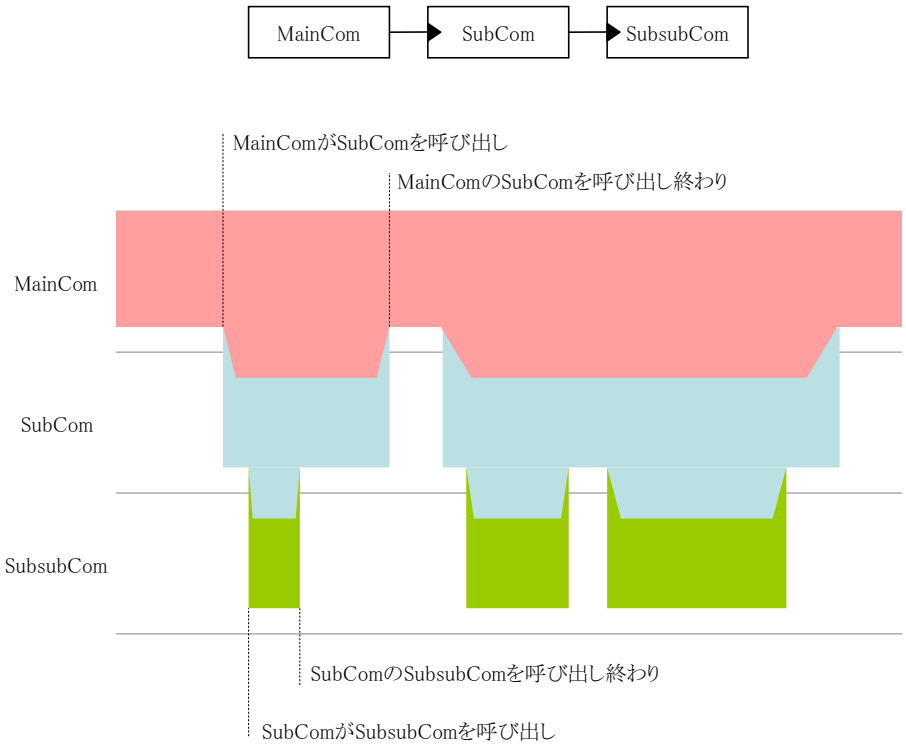


図 4.7: セルの呼び出し関係の可視化表現例

### 4.3.3 変換ルールファイルの記述

TECS のトレースログの形式は表 4.18 の用になっている．

*time* は時間であり，時間の単位はマイクロ秒である．*caller* は呼び出しセルを，*callee* は呼び出されるセルを表している．... には，シグニチャの名前や引数，返値が入るが，今回の可視化には必要がないので省略した．`.enter(...)` がセルの呼び出しを，`.leave(...)` が呼び出し先からのリターンを表している．

この形式を標準形式トレースログに変換するため，変換ルールファイルを表 4.19 のように記述した．

`.enter(...)` で，呼び出し元セルで振る舞い `call` のイベントを，呼び出し先セルで振る舞い `called` のイベントを出力するようにしている．また，`.leave(...)` で，呼び出し先セルで振る舞い `return` のイベントを，呼び出し元セルで振る舞い `callback` のイベントを出力するようにしている．

### 4.3.4 可視化ルールファイルの記述

4.3.1 小節にて，セルの呼び出し関係の可視化表現を決定した．これに従い，可視化ルールファイルの図の定義を表 4.20，と可視化ルールの定義を表 4.21 のように記述した．

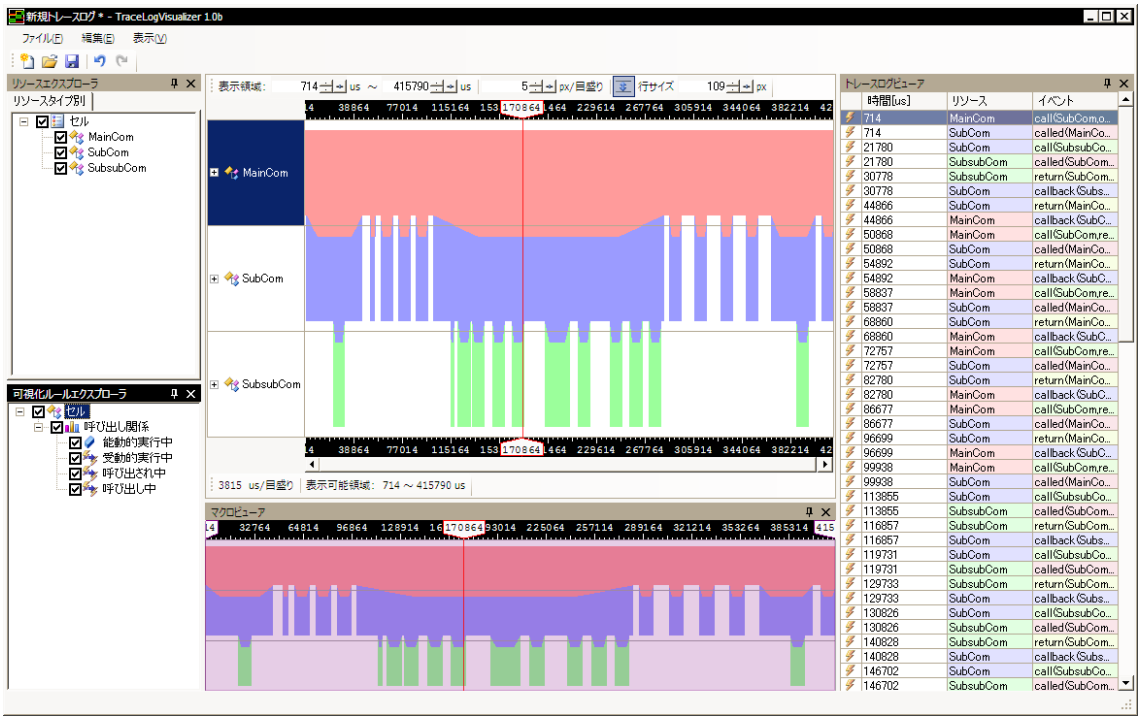


図 4.8: TECS のトレースログを可視化した TLV 実行結果のスクリーンショット

図 4.7 において，行間の接続を表現するために，多角形を用いているが，この方法では，行をまたぐセル間の呼び出し関係を表現することができない．4.3.5 小節にて，この点について言及する．

### 4.3.5 実行結果

以上のファイルの修正・追加を行い，TECS のトレースログを TLV で可視化した．図 4.8 にスクリーンショットを示す．

図 4.8 より，セルの呼び出し関係が想定した可視化表現どおりに可視化表示できていることがわかる．

これにより，TLV を用いることで形式の異なるトレースログを可視化できることが示せた．

しかしながら，前小節で述べたとおり，現状の方法では行をまたぐセル間の呼び出し関係を表現することができない．これは，TLV がリソース単位で行毎に可視化表示することを想定しているからである．

これを解決する手段として，可視化ルールファイルの図形定義の際に，図形を表示する行を指定させる方法が考えられる．

```
{
  "asp":{
    "Shapes":{
      "runningShapes":[{
          "Type":"Rectangle",
          "Size":"100%,80%",
          "Pen":{"Color":"ff00ff00","Width":1},
          "Fill":"6600ff00"
        }],
      "runnableShapes":[{
          "Type":"Line",
          "Points":["l(0),80%","r(0),80%"],
          "Pen":{"Color":"ffffaa00","Width":1}
        }],
      "waitingShapes":[{
          "Type":"Line",
          "Points":["l(0),40%","r(0),40%"],
          "Pen":{"Color":"ffff0000","Width":1}
        }],
      "activateShapes":[{
          "Type":"Arrow",
          "Points":["0,0","0,80%"],
          "Pen":{"Color":"ff000000","Width":1}
        }],
      "exitShapes":[{
          "Type":"Arrow",
          "Points":["0,80%","0,0"],
          "Pen":{"Color":"ff000000","Width":1}
        }],
      "svcShapes":[
        {
          "Type":"Rectangle",
          "Size":"100%,40%",
          "Pen":{"Color":"ff0000","Width":1, "Alpha":255, "DashStyle":"Dash"},
          "Fill":"99ff0000"
        },{
          "Type":"Text",
          "Size":"100%,40%",
          "Font":{"Align":"TopLeft", "Size":7},
          "Text":"${ARG1}"
        },{
          "Type":"Text",
          "Size":"100%,40%",
          "Font":{"Align":"BottomRight", "Size":7},
          "Text":"return ${ARG2}"
        }]}}
}
```

表 4.10: TOPPERS/ASP 用の図形を定義した可視化ルールファイル

```
{"asp":{
    "VisualizeRules":{
      "runningTaskChange":{
        "DisplayName":"実行タスク変化",
        "Events":{
          "runningTaskChangeEvent":{
            "DisplayName":"実行タスク",
            "From":"Task(state!=RUNNING).state=RUNNING",
            "To":"${FROM_TARGET}.state",
            "Figures":
 "runningTaskChangeShapes($RES_COLOR{${FROM_TARGET}},$RES_NAME{${FROM_TARGET}})"
          }}},
      "taskStateChange":{
        "DisplayName":"状態遷移",
        "Target":"Task",
        "Events":{
          "stateChangeEvent":{
            "DisplayName":"状態",
            "From":"${TARGET}.state",
            "To":"${TARGET}.state",
            "Figures":{
              "${FROM_VAL}==RUNNING" :"runningShapes",
              "${FROM_VAL}==RUNNABLE":"runnableShapes",
              "${FROM_VAL}==WAITING" :"waitingShapes"
            }},
          "activateHappenEvent":{
            "DisplayName":"起動",
            "When":"${TARGET}.activate()",
            "Figures":"activateShapes"
          },
          "exitHappenEvent":{
            "DisplayName":"終了",
            "When":"${TARGET}.exit()",
            "Figures":"exitShapes"
          }}},
      "callSvc":{
        "DisplayName":"システムコール",
        "Target":"Task",
        "Events":{
          "callSvcEvent":{
            "DisplayName":"システムコール",
            "From":"${TARGET}.enterSVC()",
            "To":"${TARGET}.leaveSVC(${FROM_ARG0})",
            "Figures":"svcShapes(${FROM_ARG0}(${FROM_ARG1}),${TO_ARG1})"
          }}}}}
}
```

表 4.11: TOPPERS/ASP 用の可視化ルールを定義した可視化ルールファイル

```
[27526775]:[1]: enter to dly_tsk dlytim=10.
[27527154]:[2]: enter to sns_ctx.
[27527284]:[1]: task 4 becomes WAITING.
[27527390]:[2]: leave to sns_ctx state=0.
[27527518]:[1]: dispatch from task 4.
[27527622]:[2]: enter to get_pid p_prcid=1386100.
[27527714]:[1]: dispatch to task 1.
[27527814]:[2]: leave to get_pid ercd=0. prcid=2
[27528162]:[2]: enter to sns_ctx.
[27528338]:[2]: leave to sns_ctx state=0.
[27528522]:[2]: enter to get_pid p_prcid=1386144.
[27528714]:[2]: leave to get_pid ercd=0. prcid=2
```

表 4.12: TOPPERS/FMP カーネルのトレースログの例

```
{
  "fmp":
  {
    "Task":{
      "DisplayName" :"タスク",
      "Attributes" :{
        "prcId":{
          "VariableType"  :"Number",
          "DisplayName"  :"プロセッサ ID",
          "AllocationType":"Dynamic",
          "CanGrouping"  :true
        },
        ...
```

表 4.13: TOPPERS/FMP カーネル用リソースヘッダファイルの一部

[*time*]:[*prcId*]: *tracelog*

表 4.14: TOPPERS/FMP カーネルのトレースログの形式

```
{
  "fmp":
  {
    "\[(?<time>\d+)\]:\[(?<pid>\d+)\]: dispatch to task (?<id>\d+)\." : [
      {
        "$EXIST{[${time}]Task(state==RUNNING && prcId==${pid})}"
          :"[${time}]$RES_NAME{[${time}]Task(state==RUNNING && prcId==${pid})}.state=RUNNABLE"
      },
      "[${time}]$RES_NAME{Task(id==${id})}.state=RUNNING"
    ],
    ...
```

表 4.15: TOPPERS/FMP カーネル用変換ルールファイルの一部

```
{
  "fmp":{
    "VisualizeRules":{
      "taskPrcIdChange":{
        "DisplayName":"所属プロセッサ",
        "Target":"Task",
        "Shapes":{
          "prcIdChangeEvent":{
            "DisplayName":"プロセッサ ID",
            "When":"${TARGET}.prcId",
            "Figures":{
              "${VAL}==1":"prcIdShapes(ff0000)",
              "${VAL}==2":"prcIdShapes(0000ff)",
              "${VAL}==3":"prcIdShapes(00ff00)",
              "${VAL}==4":"prcIdShapes(ff00ff)"
            }
          }
        }
      }
    },
    "Shapes":{
      "prcIdShapes":[
        {
          "Type":"Rectangle",
          "Location":"0,0",
          "Size":"100%,100%",
          "Fill":"${ARG0}",
          "Alpha":30
        }
      ]
    }
  }
}
```

表 4.16: TOPPERS/FMP カーネル用の可視化ルールファイル

```
{
  "tecs":
  {
    "Cell":{
      "DisplayName":"セル",
      "Attributes":{
        "active":{
          "VariableType"  :"Boolean",
          "DisplayName"   :"能動",
          "AllocationType":"Dynamic",
          "CanGrouping"   :false,
          "Default"       :false
        }
      },
      "Behaviors":{
        "call":{
          "Arguments" :{
            "cellName":"String"
          }
        },
        "callback":{
          "Arguments" :{
            "cellName":"String"
          }
        },
        "called":{
          "Arguments"   :{
            "cellerName":"String"
          }
        },
        "return":{
          "Arguments" :{
            "cellName":"String"
          }
        }
      }
    }
  }
}
```

表 4.17: TECS 用リソースヘッダファイル

```
time=timeus caller->callee.enter(...)
time=timeus caller->callee.leave(...)
```

表 4.18: TECS のトレースログの形式

```
{
  "tecs":
  {
    "time=(?<time>\d+)[^ ]+ (?<caller>[^-]+)->(?<callee>[^\.]+)\.enter\([^\)]*\)":
    [
      "\[${time}\]${caller}.call(${callee})",
      "\[${time}\]${callee}.called(${caller})"
    ],
    "time=(?<time>\d+)[^ ]+ (?<caller>[^-]+)->(?<callee>[^\.]+)\.leave\([^\)]*\)":
    [
      "\[${time}\]${callee}.return(${caller})",
      "\[${time}\]${caller}.callback(${callee})"
    ]
  }
}
```

表 4.19: TECS 用変換ルールファイル

```
{
  "tecs":{
    "Shapes":{
      "running":[{
          "Type":"Polygon",
          "Size":"100%,100%",
          "Points":["0%,90%","100%,90%","100%,10%","0%,10%"],
          "Fill":"${ARG0}",
          "Alpha":100
        }],
      "calledCalleeShape":[{
          "Type":"Polygon",
          "Size":"100%,100%",
          "Points":["0%,100%","10%,100%","20%,90%","80%,90%",
                    "90%,100%","100%,100%","100%,90%","0%,90%"],
          "Fill":"${ARG0}",
          "Alpha":100
        }],
      "calledCallerShape":[{
          "Type":"Polygon",
          "Size":"100%,100%",
          "Points":["10%,100%","90%,100%","80%,90%","20%,90%"],
          "Fill":"${ARG0}",
          "Alpha":100
        }],
      "callingCalleeShape":[{
          "Type":"Polygon",
          "Size":"100%,100%",
          "Points":["0%,0%","0%,10%","10%,0%","90%,0%","100%,10%","100%,0%"],
          "Fill":"${ARG0}",
          "Alpha":100
        }],
      "callingCallerShape":[{
          "Type":"Polygon",
          "Size":"100%,100%",
          "Points":["0%,10%","100%,10%","90%,0%","10%,0%"],
          "Fill":"${ARG0}",
          "Alpha":100
        }]
    }
  }
}
```

表 4.20: TECS 用の図形を定義した可視化ルールファイル

```
{
  "tecs":{
    "VisualizeRules":{
      "callRelative":{
        "DisplayName":"呼び出し関係",
        "Target":"Cell",
        "Shapes":{
          "activeRunning":{
            "DisplayName":"能動的実行中",
            "When":"${TARGET}.active",
            "Figures":{"${VAL}==True":"running($RES_COLOR{${TARGET}})"}
          },
          "passiveRunning":{
            "DisplayName":"受動的実行中",
            "From":"${TARGET}.called()",
            "To":"${TARGET}.return()",
            "Figures":"running($RES_COLOR{${TARGET}})"
          },
          "calledEvent":{
            "DisplayName":"呼び出され中",
            "From":"${TARGET}.called()",
            "To":"${TARGET}.return()",
            "Figures":[
              "calledCallerShape($RES_COLOR{${FROM_ARGO}})",
              "calledCalleeShape($RES_COLOR{${TARGET}})"
            ]
          },
          "callingEvent":{
            "DisplayName":"呼び出し中",
            "From":"${TARGET}.call()",
            "To":"${TARGET}.callback()",
            "Figures":[
              "callingCallerShape($RES_COLOR{${TARGET}})",
              "callingCalleeShape($RES_COLOR{${FROM_ARGO}})"
            ]
          }
        }
      }
    }
  }
}

```

表 4.21: TECS 用の可視化ルールを定義した可視化ルールファイル

# 第5章　開発プロセス

## 5.1　OJL

TLV は，OJL(On the Job Learning) の開発テーマとして開発された．OJL とは，企業で行われているソフトウェア開発プロジェクトを教材とする実践教育であり，製品レベルの実システムの開発を通じて想像的な思考力を身につけるとともに，単なる例題にとどまらない現実の開発作業を担うことにより，納期，予算といった実社会の制約を踏まえたソフトウェア開発の実際について学ぶことを目的としている．

TLV はプロジェクトベースで開発が行われ，企業出身者 2 名と教員 1 名がプロジェクトマネージャを務め，学生 3 名 (筆者含む) と企業出身者 2 名が開発実務を行った．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．

### 5.1.1　フェーズ分割

単年度で TLV を開発するにあたり，全体を 3 フェーズに分割した．各々のフェーズの内容は次の通りである．

#### フェーズ 1

**期間**

2008 年 5 月〜同年 8 月 (約 3ヶ月)

**目的・目標**

プロトタイプの実装を行い，そのプロセスと成果物から，GUI の評価や要求の再抽出，アプリケーションドメイン分析を通じた設計方法の探索を行う．

**実装内容**

機能を限定して実装を行う．

トレースログの対象を TOPPERS/ASP カーネルに絞り，可視化表示項目もタスクの状態遷移のみとする．

標準形式トレースログへの変換や，可視化ルールの適用による可視化表示は行わない．

**実施結果**

実装成果物の総行数は約 9000 行 (有効行数 5000 行) であった.

開発関係者 4 名と RTOS 開発者および利用者の 7 名に成果物を利用してもらい意見を収集した．その結果，ユーザインタフェース，追加の機能要求，可視化表現項目について意見が得られた.

実装中心の開発プロセスになってしまい，チーム開発がうまく機能しなかった.

### フェーズ 2

**期間**

2008 年 9 月〜2009 年 1 月 (約 5ヶ月)

**目的・目標**

標準形式トレースログの策定，可視化ルールの策定．標準形式トレースログへの変換，可視化ルールファイルによる可視化表示の外部プラグイン化の実装．ユースケース駆動アジャイル開発の導入.

**実装内容**

主要機能である，標準形式トレースログへの変換，可視化ルールファイルによる可視化表示の外部プラグイン化を実装する．また，フェーズ 1 にて再定義された機能要求を可能な限り実装する.

**実施結果**

実装成果物の総行数は約 18500 行 (有効行数 10300 行) であった.

標準形式トレースログ，可視化ルールファイル，またリソースファイル，リソースヘッダファイルの形式を本論文で述べたとおり定義した.

また，本論文で示したとおり，標準形式トレースログへの変換と可視化ルールファイルによる可視化表現項目の追加を実現できた.

ユースケース駆動アジャイル開発により，ユースケース毎にイテレーションを繰り返し実施し開発を行った．結果，36 個中 27 個のユースケースについて実装が完了した.

### フェーズ 3

**期間**

2008 年 2 月〜2009 年 3 月 (約 1ヶ月)

**目的・目標**

TLV で読み込めるトレースログ形式を増やす．可視化表現項目を増やす.

**実装内容**

フェーズ 2 の成果を元に，変換ルールファイルを追加・変更し対応するトレースログの形式を増やす．また，可視化ルールファイルを追加・変更し可視化表現項目の充実を図る．

**実施結果**

未実施である．

## 5.2 ユースケース駆動アジャイルソフトウェア開発

フェーズ 2 において，TLV の開発は，ユースケース駆動アジャイルソフトウェア開発という手法を用いて行われた．

アジャイルソフトウェア開発では，反復 (イテレーション) と呼ばれる短い期間を単位に反復して開発を行い，計画ではなく状況において適応的に対応することを重視してソフトウェア開発を行う．1 つのイテレーション内では 1 つの機能に対して設計，実装，テスト，文書化といった工程を完結して行う．TLV の開発ではユースケースを機能の単位にイテレーションを反復して実施した．図 5.1 にユースケース駆動アジャイルソフトウェア開発の例を示す．

はじめに外部設計としてユースケース図，ユースケース内のクラス図，シーケンス図を作成する．次に，クラス単位で内部設計，単体テスト，実装を行う．内部設計では，クラス図で定義したクラスに関してインタフェースのみを記述したスケルトンを作成し，コメントとして内部設計を記述する．単体テストは，メソッド単位のユニットテストとし，統合開発環境に付属するユニットテストフレームワークを用いて実装，実施する．単体テストの実装はテストケースの記述であり，ユニットテストフレームワークが出力するテストコードのスケルトンに必要な情報を記述することで行う．単体テストの実装が終わったらクラスの実装を行う．クラスの実装は，単体テストを実施しながら，テスト項目のをすべてを成功するかどうか試しながら行う．1 つのクラスの実装が終わればユースケース内の次のクラスの内部設計に移る．

このようにしてユースケース内のクラスをすべて実装したら，ユースケース内のクラスを結合してユースケースとして駆動する形に実装する．この際も，テストケースの実装を先に行ってから実装を開始する．結合テストのテスト項目のをすべてを成功したら，1 つのイテレーションの完了であり，次のユースケースの外部設計を開始する．

本節では，TLV の開発において，ユースケース駆動アジャイルソフトウェア開発の各工程で実践した内容について詳述する．

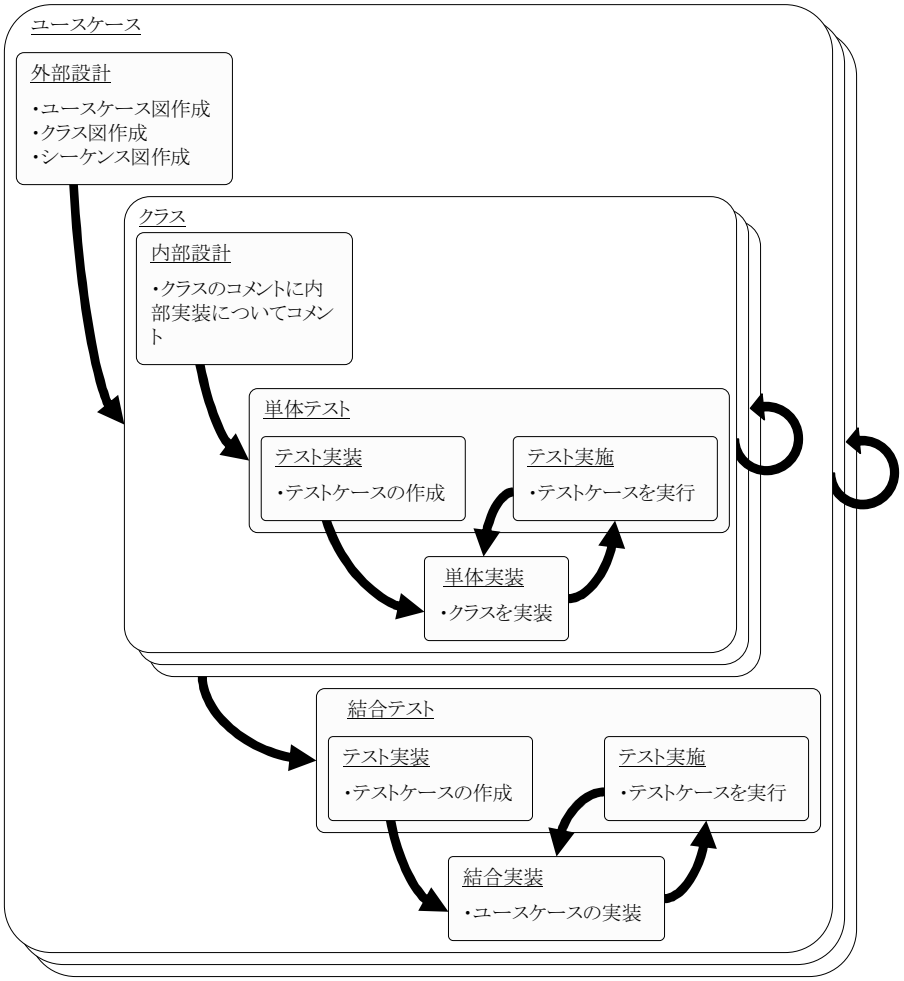


図 5.1: ユースケース駆動アジャイルソフトウェア開発

### 5.2.1 プロジェクト管理

フェーズ 2 のはじめの作業として，フェーズ 1 で行った評価の結果と，フェーズ 2 の実施概要について，プロジェクト計画書として文書化した．

次に，フェーズ 2 で実装する TLV の機能を，要求仕様書として文書化した．また，機能をユースケースを単位に定義し，ユースケース図を作成した．図 5.2 に TLV のユースケース図を示す．

TLV の機能をユースケースを単位に定義した結果，全ユースケース数は 36 個であった．これらのユースケースを工程毎に分けた進捗表を作成し，これを用いて進捗管理を行い，複数のメンバーで工程の分担，作業項目の同期を行った．

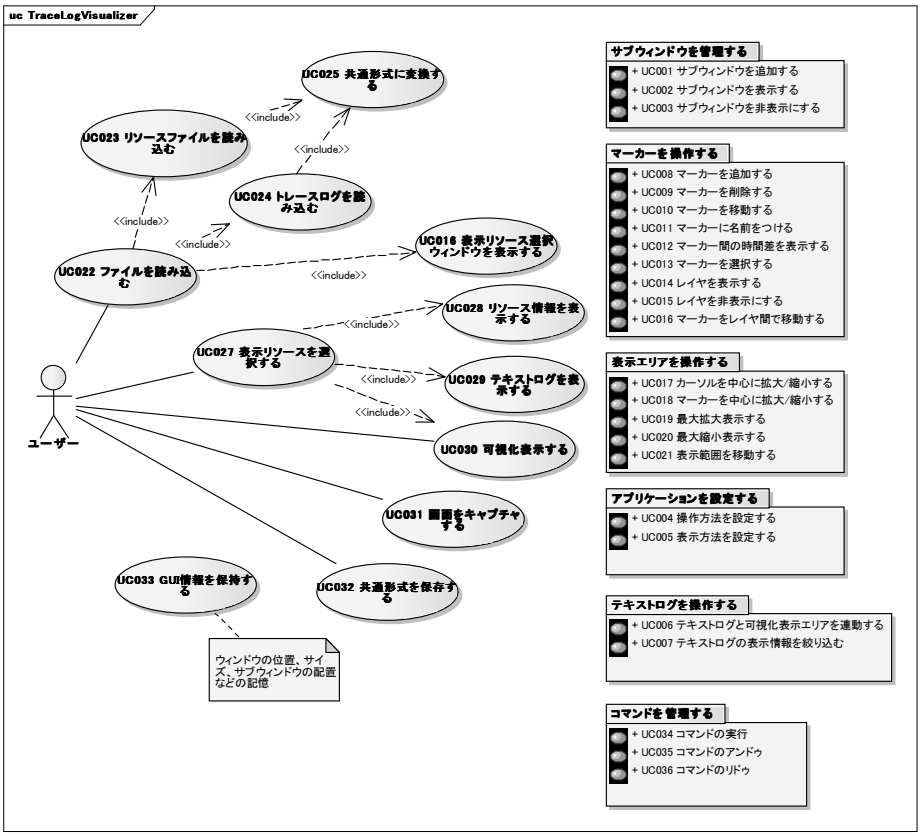


図 5.2: TLV のユースケース図

### 5.2.2 設計

図 5.1 に示すとおり，外部設計としては，ユースケース内のクラスについて，クラス図を，クラス間の連携の流れをシーケンス図として作成することで行う．図 5.3 に作成したクラス図の例を示す．また，図 5.4 に作成したシーケンス図の例を示す．

内部設計は文書化せずに，ソースコードのコメントを記述することで行った．外部設計で作成したクラス図を元に，インタフェースのみを記述したクラスを記述し，メソッドのコメントとして，引数，返値の説明，どのような内部実装を行うべきかの説明を記述する．

### 5.2.3 テスト

TLV のテストは，統合開発環境に付属するユニットテストフレームワークを用いて実装，実施した．ユニットテストフレームワークは，クラスのメソッドの定義を元に，テストケースのスケルトンを自動生成する仕組みを搭載しており，テスト工程作業の大幅な省力化を図ることができる．スケルトンは，引数の組み合わせと期待値を開発

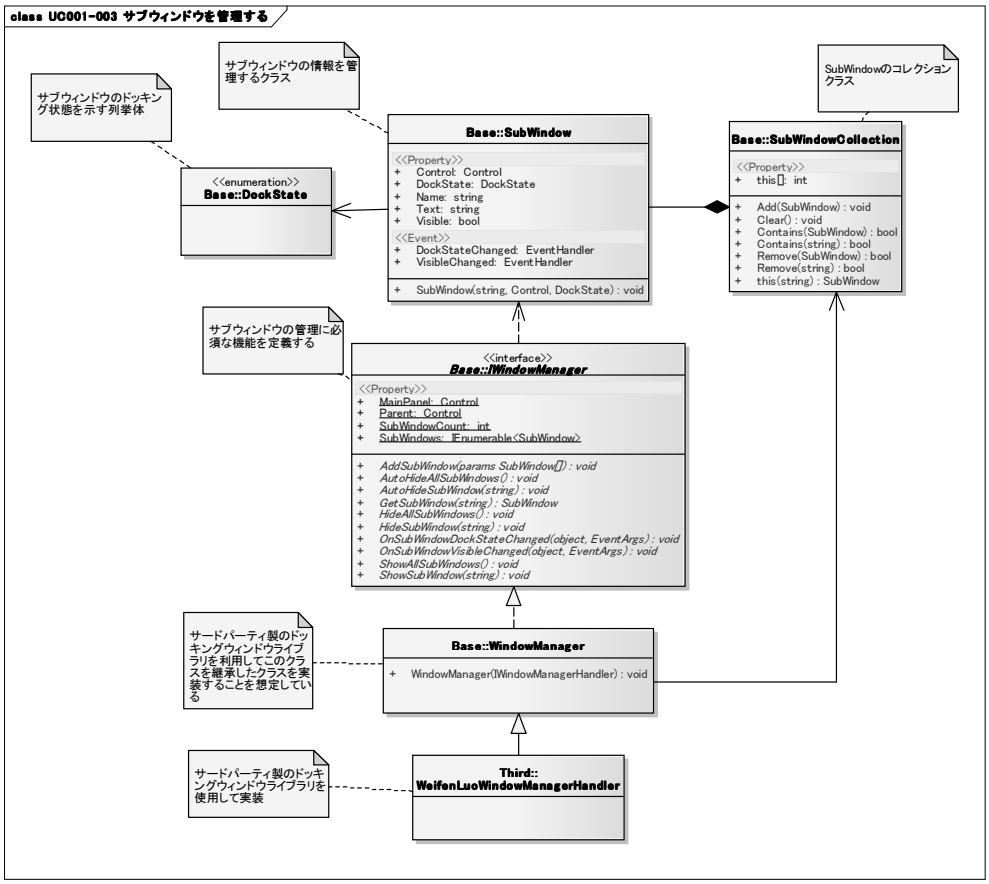


図 5.3: TLV の外部設計で作成したクラス図の例

者が入力するように生成されており，これらを入力する作業が実質的なテストケースの実装作業となる．

ユニットテストの単位はメソッドだが，単純なアクセサメソッドはテストの対象外とした．その結果，491 メソッドがユニットテストの対象となった．

### 5.2.4 実装

TLV の実装言語は，C♯ 3.0 である．統合開発環境として Microsoft Visual Studio 2008 Professional Edition を用いた．TLV の実行には .NET Framework 3.5 が必要である．

実装はクラス単位で行い，ユニットテストが全部成功することを目標に行う．その際，テストの実施とクラスの実装は反復して行う．テスト結果は記録されるため，文書化する必要がなく，素早くデバッグを行うことができる．

TLV のソースコードメトリクスを表 5.1 に示す．

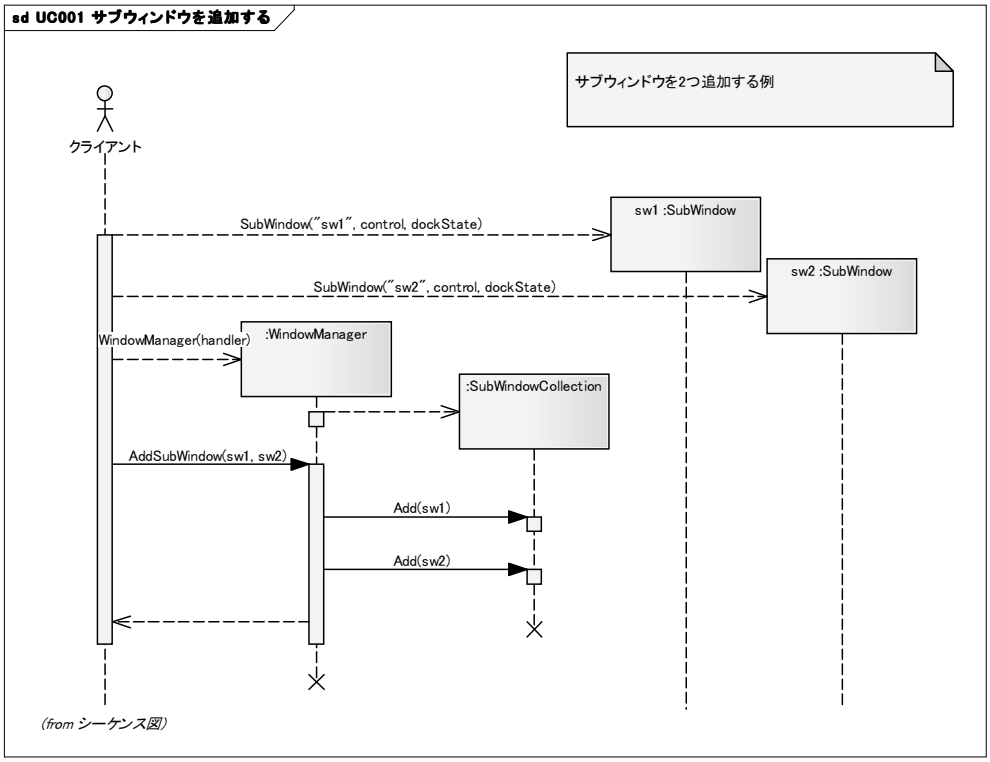


図 5.4: TLV の外部設計で作成したシーケンス図の例

| | |
|---|---|
| 総行数 | 18497 |
| ファイル平均行数 | 94.38 |
| 有効な行数 | 10332 |
| コメント行数 | 1432 |
| 空行 | 2224 |
| その他の行数 (中括弧など) | 4509 |
| ファイル数 | 196 |
| クラス数 | 186 |
| 構造体数 | 0 |
| インタフェース数 | 22 |
| 列挙体 | 15 |
| デリゲート | 2 |

表 5.1: TLV のソースコードメトリクス

# 第6章　おわりに

## 6.1　まとめ

本論文では，トレースログ可視化ツールである TLV の開発について，開発背景と既存ツールの問題点，設計と実装，利用例，また開発プロセスについて述べた．

TLV を開発するにあたり，動機となった背景として，組込みシステムにおいてもマルチコアプロセッサの利用が進んでおり，それに伴い従来のデバッグ方法が有効でなくなってきたことを述べた．これは，マルチコアプロセッサが各コアで並列処理を行うため，プログラムの挙動が非決定的になり，バグの再現が保証されず，従来のブレークポイントによるステップ実行ではバグを確実に捕らえることができないからである．

一方，マルチコアプロセッサ環境におけるデバッグで有効な方法として，実行中にデバッグを行うのではなく，実行後にトレースログを解析する手法があることを述べた．そして，開発者が直接トレースログを扱うのは効率が悪く，トレースログの解析を支援するツールが要求されており，その 1 つとして可視化表示ツールがあることを述べた．

既存のトレースログ可視化ツールは，標準化されたトレースログを扱っていないため，利用できるトレースログの形式が限られており，汎用性に乏しい点を指摘した．また，可視化表示項目が提供されているものに限られ，変更や追加を行う仕組みも提供されておらず，拡張性に乏しい点についても述べた．

こういった既存のトレースログ可視化ツールの欠点を解消するため，汎用性と拡張性のあるトレースログ可視化ツールとして TLV を開発することを目標として定めたことを述べた．

汎用性を確保するための解決策として，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し，TLV がこの形式のトレースログの可視化に対応するようにしたことを述べた．そして，任意の形式のトレースログは，標準形式トレースログへ変換することで TLV で扱うことができるようになることを述べた．また，トレースログを変換する仕組みを形式化した，変換ルールファイルを定義することで変換が行える仕組みを提供したことも述べた．

実際に，シングルコアプロセッサ用 RTOS のトレースログを標準形式トレースログへ変換し，可視化できたことを示し有効性を確認した．また，少量の変更でマルチコア用 RTOS のトレースログの可視化に対応できることを示し，拡張性があることを示した．最後に，RTOS のトレースログの形式とは全く異なる，組込みコンポーネン

トシステムのトレースログを標準形式トレースログへ変換し可視化表示できることを示し，汎用性があることを示した．

次に，拡張性を確保するため実施したこととして，可視化表現とトレースログを対応付ける仕組みを形式化した，可視化ルールファイルを定義したことを述べた．そして，可視化ルールファイルを記述することで，任意の可視化表示項目の追加や変更を行えることを述べた．

実際に，マルチコアプロセッサ用 RTOS のトレースログを可視化する際に，所属プロセッサ ID という可視化表示項目の追加を行ったが，新たにマルチコアプロセッサ用 RTOS の可視化ルールファイルを追加するだけで実現できたことを示し，拡張性があることを確認した．

TLV の開発は，OJL 形式で行い，実際のソフトウェア開発現場で用いられているプロセスを適用して行ったことを述べた．開発プロセスとしてユースケース駆動アジャイル開発を採用したことを述べ，各工程について作業内容を述べた，

## 6.2 今後の課題と展望

現在の TLV の課題としては次の 2 点がある．

1 つ目の課題は，4.3.5 小節で述べたとおり，可視化表示をリソース毎に行で行っているため，行をまたぐ描画指定ができないことである．これを解決する手段としては，可視化ルールファイルの図形定義の際に，図形を表示する行を指定させる方法が考えられる．

2 つ目の課題は，計算や制御を伴う可視化表示が対応できない点である．現在の変換ルールと可視化ルールは，置換マクロと条件を用いた指定はできるようになっているが，変数を用いて状態を保持したり，任意回数ループして出力させたり，計算結果を用いて出力内容を変更するなどといったことができない．このため，ある一定期間のイベントの統計情報を用いるような可視化表示を行うことができない．たとえば，CPU 使用率や，タスクの CPU 占有率などである．これを解決するには，可視化ルールや変換ルールをスクリプト言語を用いて記述できるように拡張する方法が考えられる．

今後の展望としては，上記 2 点の課題の克服と，フェーズ 3 の実施による，対応するトレースログの拡充と，すでに可視化表示することができている TOPPERS/ASP，FMP カーネル，TECS の可視化表示項目の追加を行う予定である．現在の TOPPERS/ASP，FMP カーネルの可視化表示項目としては，リソースとしてタスクのみを考慮しているので，セマフォやイベントフラグなど，他のカーネルオブジェクトの可視化表示項目を追加することを予定している．また，TECS に追加する可視化表示項目として，現在は，セルの呼び出し関係において，どんなシグニチャの呼び口の，どの関数を呼び出したかを考慮していないので，それらを考慮した可視化表現を考案したい．

# 謝辞

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，同研究室の冨山宏之准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同研究室の本田晋也助教，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科付属組込みリアルタイム研究センターの長尾卓哉研究員，同研究センターの杉山俊氏に深く感謝致します．

TLV のテストに関して，テスト仕様の作成から実施までご尽力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阿草・結縁研究室の水野洋樹氏，同専攻宮尾・八槇研究室の柳澤大祐氏に深く感謝いたします．

# 参考文献


[1] JTAG ICE PARTNER-Jet，http://www.kmckk.co.jp/jet/，最終アクセス 2009 年 1 月 14 日

[2] WatchPoint デバッガ，https://www.sophia-systems.co.jp/ice/products/watchpoint，最終アクセス 2009 年 1 月 14 日

[3] QNX Momentics Tool Suite，http://www.qnx.co.jp/products/tools/，最終アクセス 2009 年 1 月 14 日

[4] eBinder，http://www.esol.co.jp/embedded/ebinder.html，最終アクセス 2009 年 1 月 14 日

[5] LKST（Linux Kernel State Tracer) - A tool that records traces of kernel state transition as events，http://oss.hitachi.co.jp/sdl/english/lkst.html，最終アクセス 2009 年 1 月 14 日

[6] Prasad, V., Cohen, W., Eigler, F. C., Hunt, M., Keniston, J. and Chen, B.: Locating system problems using dynamic instrumentation. Proc. of the Linux Symposium, Vol.2, pp.49—64, 2005.

[7] Mathieu Desnoyers and Michel Dagenais.: The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In OLS (Ottawa Linux Symposium) 2006, pp.209—224, 2006.

[8] R. McDougall, J. Mauro, and B. Gregg.: Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris. Pearson Professional, 2006.

[9] Mathieu Desnoyers and Michel R. Dagenais.:Tracing for Hardware, Driver, and Binary Reverse Engineering in Linux. Recon 2006

[10] OpenSolaris Project: Chime Visualization Tool for DTrace，http://opensolaris.org/os/project/dtrace-chime/，最終アクセス 2009 年 1 月 14 日

[11] RFC3164 The BSD syslog Protocol, http://www.ietf.org/rfc/rfc3164.txt，最終アクセス 2009 年 1 月 14 日

[12] RFC4627 The application/json Media Type for JavaScript Object Notation (JSON)，http://tools.ietf.org/html/rfc4627，最終アクセス 2009 年 1 月 14 日

[13] TOPPERS Project，http://www.toppers.jp/，最終アクセス 2009 年 1 月 14 日

[14] Takuya Azumi and Masanari Yamamoto and Yasuo Kominami and Nobuhisa Takagi and Hiroshi Oyama and Hiroaki Takada.:A New Specification of Software Components for Embedded Systems. Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2007), pp.45-50, 2007

# OJLによるトレースログ可視化ツールの開発

350702101　後藤 隼弐