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停滞や，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出来る．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出来るからである．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 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

Abstract

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

第1章　はじめに

1.1　開発背景

近年，PC，サーバ，組み込みシステム等，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停滞や，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出来る．組み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み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出来るからである．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 組み込みシステム向けデバッガソフトウェア

組み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．組み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，組み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．ここで，組み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．

組み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．

これら組み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，組み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 組み込み OS 向けの統合開発環境

QNX Software Systems 社は，自社の組み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

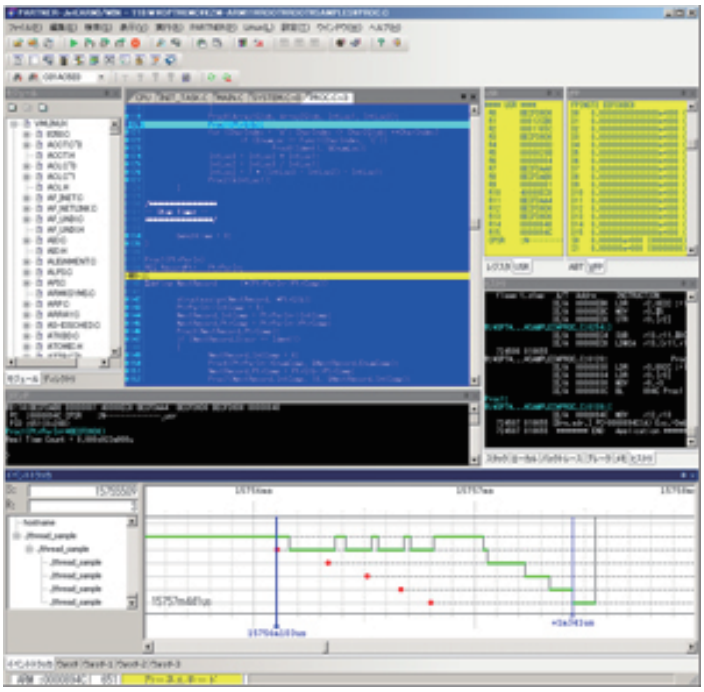

図 1.1: PARTNER イベントトラッカー

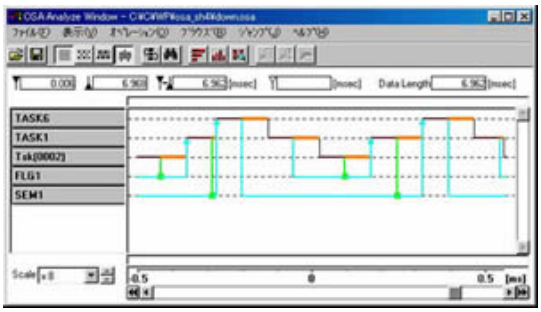

図 1.2: WatchPoint OS アナライザ

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-Kenel/μITRON 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出来る．図 1.4 に QNX System Profiler のスクリーンショットを示す．

このように，商用の組み込み 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されているものであり，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，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．

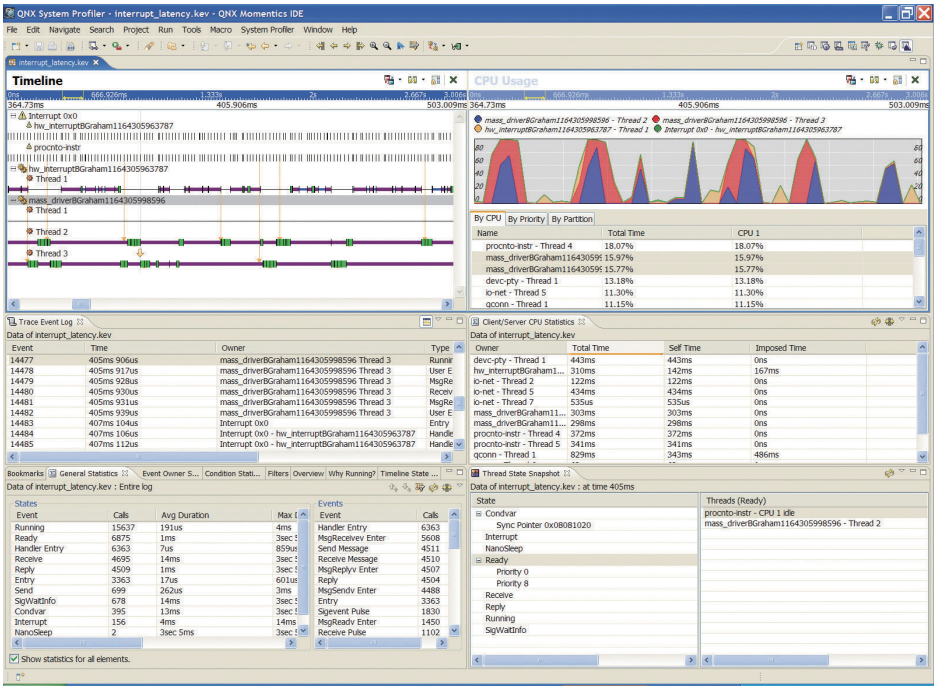

図 1.3: QNXSystemProfiler

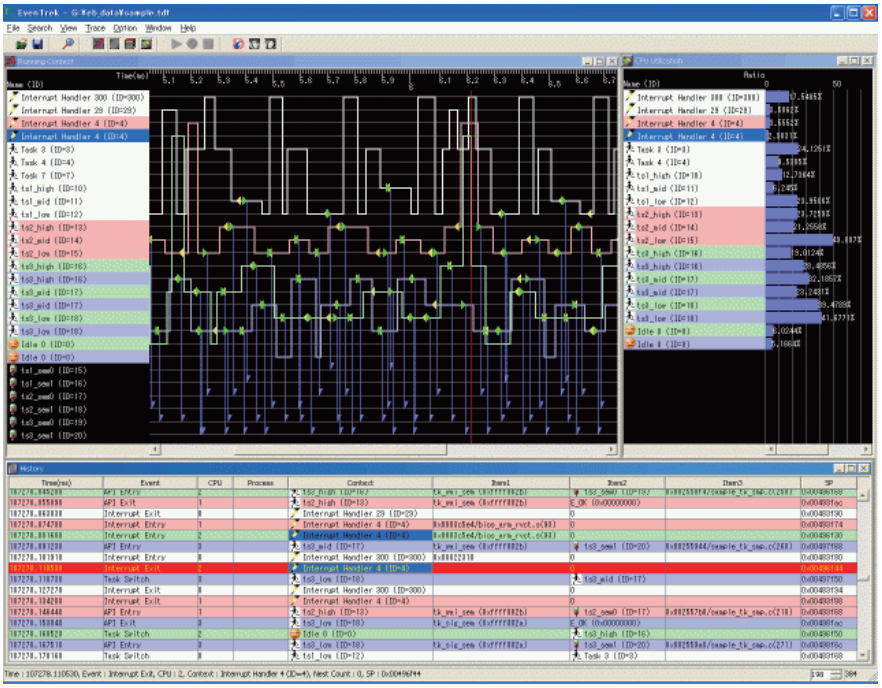

図 1.4: eBinder EvenTrek

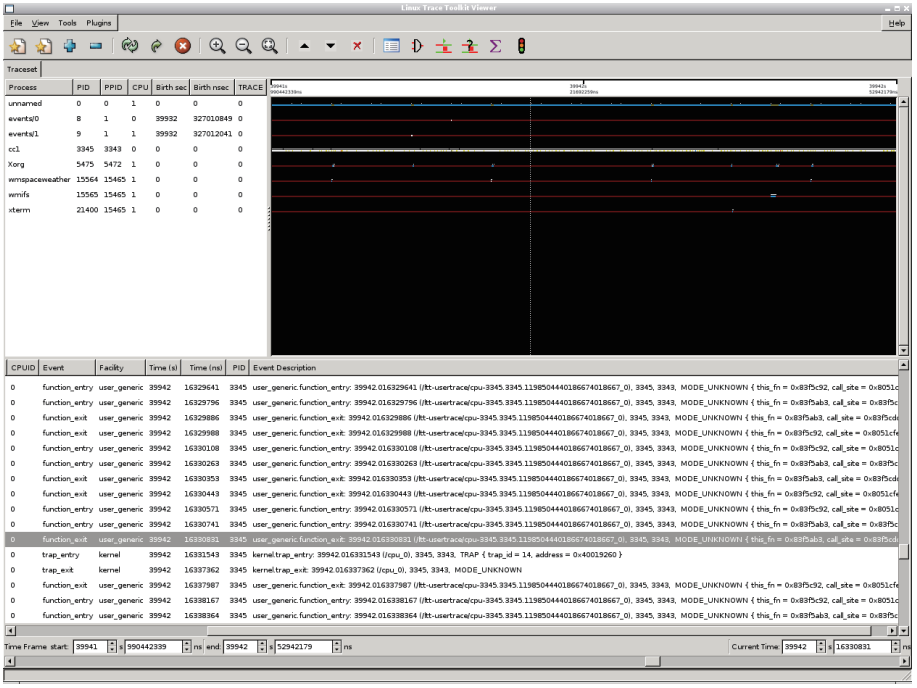

図 1.5: LTTV

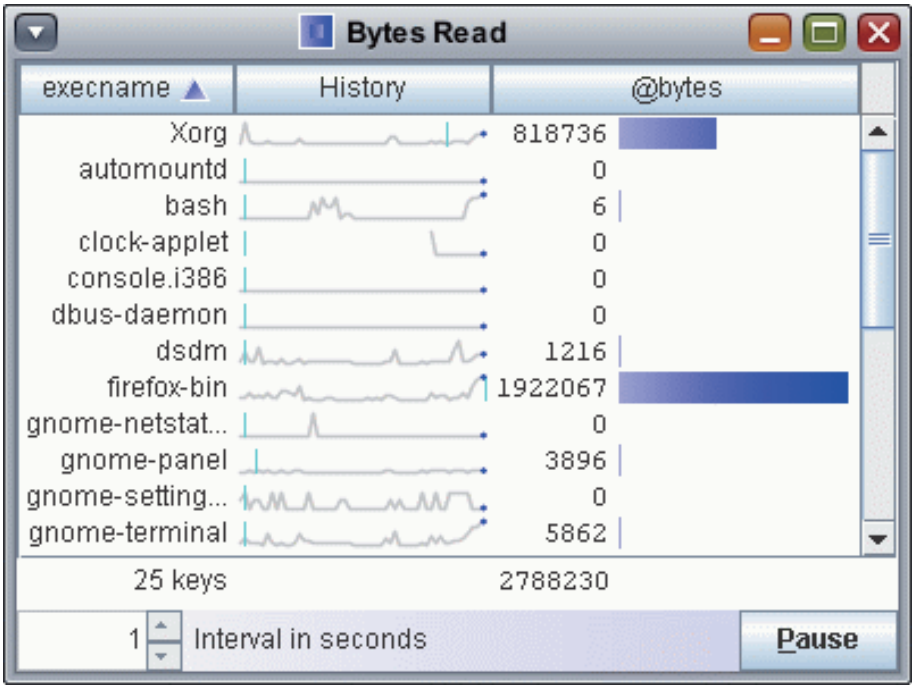

図 1.6: Chime

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．

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出来るようになる．

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．

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．

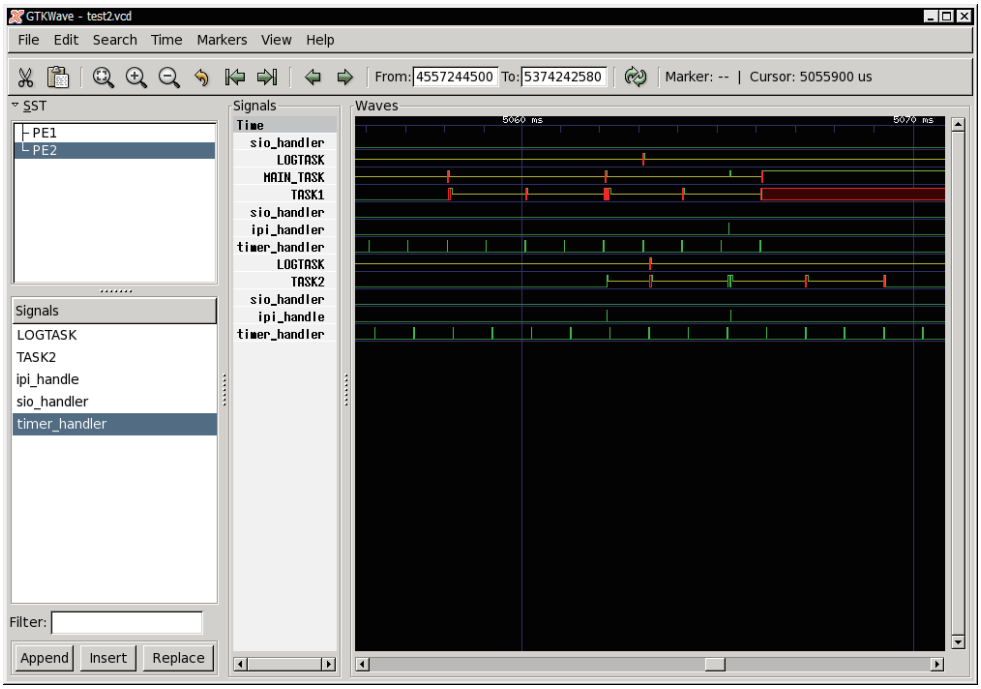
GTKWave - test2.vcd
File Edit Search Time Markers View Help
From: 4557244500 To: 5374242580
Marker: -- | Cursor: 5055900 us
SST
PE1
PE2
Signals
LOGTASK
TASK2
ipi_handle
sio_handler
timer_handler
Filter:
Append Insert Replace
Signals
Time
sio_handler
LOGTASK
MAIN_TASK
TASK1
sio_handler
ipi_handler
timer_handler
LOGTASK
TASK2
sio_handler
ipi_handle
timer_handler
Waves
5060 ms
5070 ms

図 1.7: GTKWave

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，汎用性と拡張性を備えることである．

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，可視化表示メカニズム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して定義出来る用にする．これにより，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出来ることを表し，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出来るように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出来る．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．属性と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，振る舞いとはリソースの行為であるとする．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．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，オブジェクト指向におけるメソッドの，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ものである．振る舞いの引数は，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．

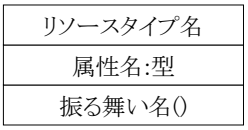

図 2.1: リソースタイプ

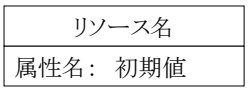

図 2.2: リソース

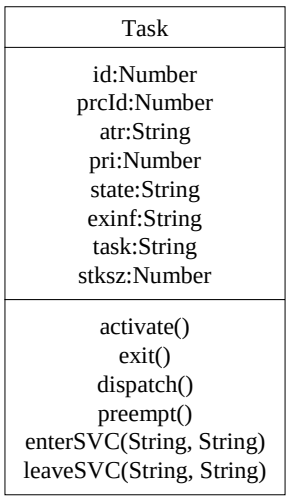

図 2.3: タスクをリソースタイプ Task として表現した例

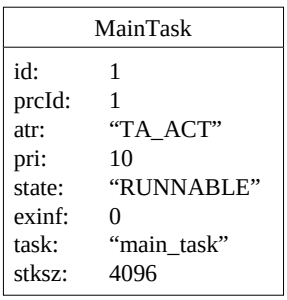

図 2.4: リソースタイプ Task のリソース MainTask の例

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定義する．標準形式トレースログの定義にはEBNF(Extended Backus—Naur Form) および終端記号として正規表現を用いる．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と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．

```
TraceLog = { TraceLogLine };
TraceLogLine = "[",Time,"]",Event,"\n";
Time = /[0-9a-Z]+/;
```

トレースログが記録されたファイルのデータをTraceLogとした．また，TraceLogを改行記号で区切った1行をTraceLogLineとし，トレースログの単位とした．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，数値とアルファベットで構成される．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」などを表現出来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，Eventを次のように定義した．

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

リソースはリソース名による直接指定、あるいは型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．

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

AttributeChangeは属性の値の変化を，BehaviorHappenは振る舞いを表現している．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;
AttributeName = Name;
```

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

リソースの条件指定の際の AttributeCondition 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 = "=="|"!="|"<"|">"|"<="|">=";
```

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次に示す.

```
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()
```

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.

2.3 可視化表示メカニズムの抽象化

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 の可視化表示メカニズムは，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とは，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抽象化している．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，トレースログにおける時系列を次元に持つ，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 2.7 にワールド変換の例を示す．

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出来るとする．この引数は，図形を構成する基本図形の，任意の属性に割り当てる．

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．

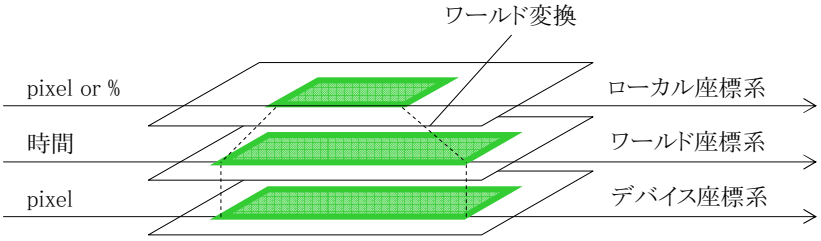

図 2.5: 座標系

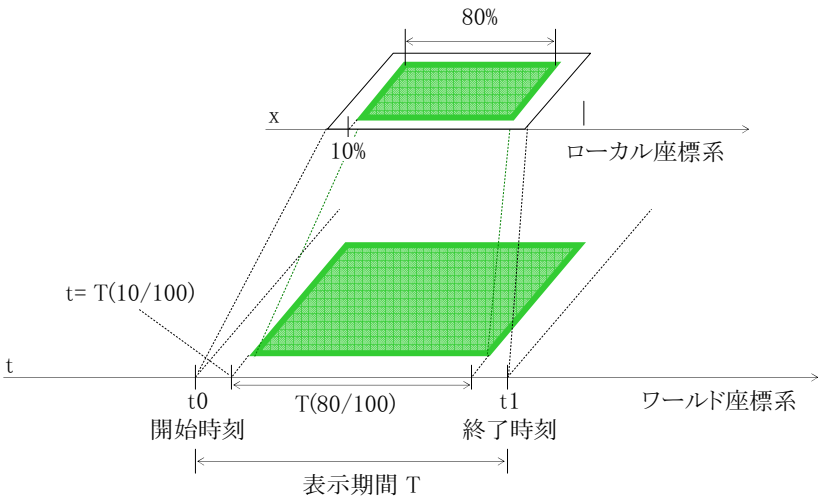

図 2.6: ワールド変換

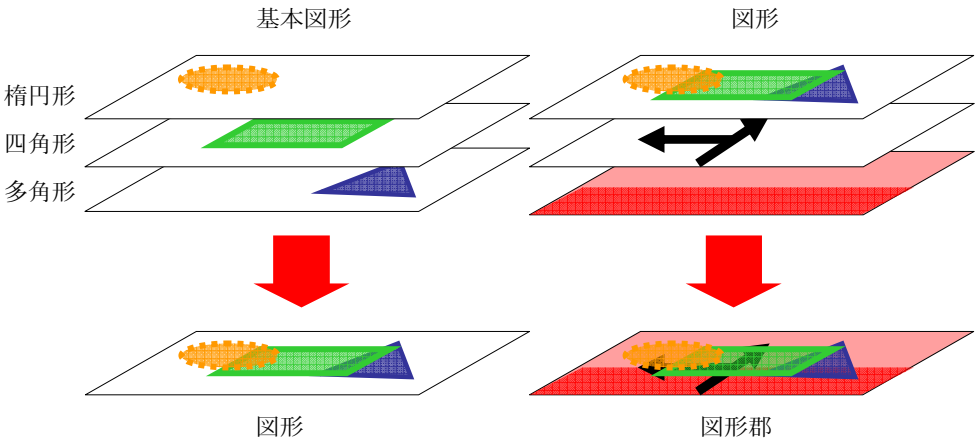

図 2.7: 図形と図形群

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た．

ここで，開始時刻と終了時刻に対応するイベントを開始イベント，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．ここで，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．

taskBecomeRunning を以下の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．

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

.

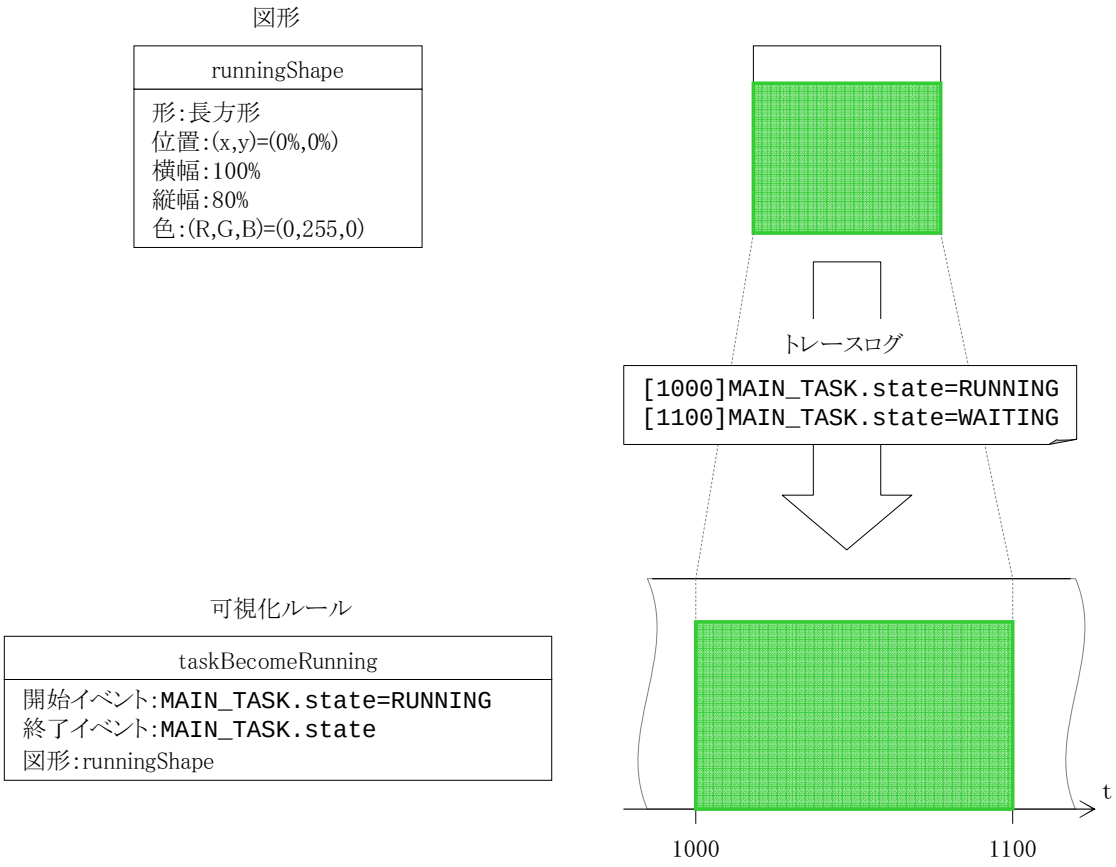
図形
runningShape
形:長方形
位置:(x,y)=(0%,0%)
横幅:100%
縦幅:80%
色:(R,G,B)=(0,255,0)
トレースログ
[1000]MAIN_TASK.state=RUNNING
[1100]MAIN_TASK.state=WAITING
可視化ルール
taskBecomeRunning
開始イベント:MAIN_TASK.state=RUNNING
終了イベント:MAIN_TASK.state
図形:runningShape
t
1000
1100

図 2.8: 可視化ルール

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，ターゲット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．

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．

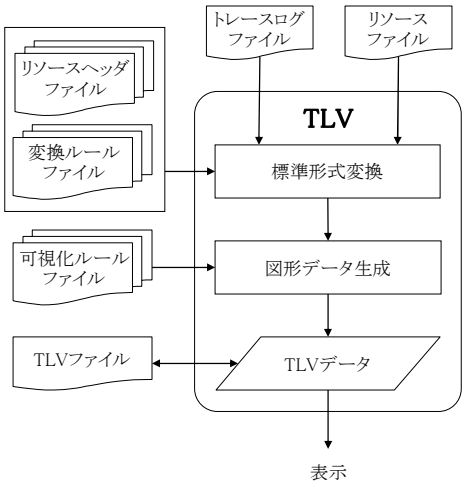

図 3.1: TLV の全体像

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出来ない．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パース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出来るからである．

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．メンバの名前は文字列である．

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

配列は複数の値をコンマで区切り，角括弧で囲んで表現する．

```
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．

```
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]/;
```

以下にオブジェクトの例を示す.

```
1  {
2    "Image":{
3      "Width":  800,
4      "Height": 600,
5      "Title":  "View from 15th Floor",
6      "Thumbnail":{
7        "Url":    "http://www.example.com/image/481989943",
8        "Height": 125,
9        "Width":  "100"
10     },
11     "IDs": [116, 943, 234, 38793]
12   }
13 }
```

以下に配列の例を示す.

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

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．また，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を次に示す.

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

上記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次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．

```
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
```

元のトレースログが 3 行なのに対し，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，標準形式トレースログの 1 行目と 3 行目が状況により追加される必要があるからである．標準形式トレースログの 1 行目は，元のトレースログの 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，標準形式トレースログの 3 行目は，元のトレースログの 2 行目に対応しており，すでに起動しているタスクがプリエンプトされて状態が 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．標準形式トレースログの 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出来ないためである．また，標準形式トレースログの 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出来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出来る必要がある．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．

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．

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

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

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出来る．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 を定義することが出来る．

以下に，ターゲット fmp 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．

```
{
  "asp":{
    "Task":{
      "DisplayName":"タスク",
      "Attributes":{
        "id":{
          "VariableType":"Number",
          "DisplayName":"ID",
```

```
 9            "AllocationType":"Static",
10            "CanGrouping":false
11          },
12          "atr":{
13            "VariableType":"String",
14            "DisplayName":"属性",
15            "AllocationType":"Static",
16            "CanGrouping":false
17          },
18          /* 省略 */
19          "state":{
20            "VariableType":"String",
21            "DisplayName":"状態",
22            "AllocationType":"Dynamic",
23            "CanGrouping":false,
24            "Default":"DORMANT"
25          }
26        },
27        "Behaviors":{
28          "preempt":{"DisplayName":"プリエンプト"},
29          "dispatch":{"DisplayName":"ディスパッチ"},
30          "activate":{"DisplayName":"起動"},
31          "exit":{"DisplayName":"終了"},
32          /* 省略 */
33          "enterSVC":{
34            "DisplayName":"サービスコールに入る",
35            "Arguments":{"name":"String","args":"String"}
36          },
37          "leaveSVC":{
38            "DisplayName":"サービスコールから出る",
39            "Arguments":{"name":"String","args":"String"}
40          }
41        }
42      }
43    }
44  }
```

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

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．

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．

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．また，他にも時間の単位や時間の基数，適用する標準形式変換ルール、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．

以下にリソースファイルの例を示す．

```
{
  "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"
      }
    }
  }
}
```

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

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．以下に，変換ルールファイルの例を示す．

```
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  }
```

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出来る．

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出来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出来る．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出来る．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．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．

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．

以下に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．

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

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．

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"：直線，"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** 文字列．サイズ指定形式

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

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** 文字列．位置指定形式

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出来るとした．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出来る．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”という指定が出来なくなってしまう．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出来ないからである．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．

3.4 TraceLogVisualizer の機能

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

4.1　組み込みRTOSのトレースログの可視化

4.1.1　シングルコアプロセッサ用RTOSのトレースログの可視化

4.1.2　マルチコアプロセッサ用RTOS対応への拡張

4.1.3　その他のシステムのトレースログの可視化

4.2　可視化表示項目の追加・カスタマイズ

第5章　開発プロセス

5.1　OJL

5.1.1　フェーズ分割

5.2　ユースケース駆動アジャイル開発

5.2.1　プロジェクト管理

5.2.2　設計

5.2.3　テスト

5.2.4　実装

5.3　開発成果物

第6章　おわりに

6.1　まとめ

6.2　今後の展望と課題

謝辞

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

OJLによるトレースログ可視化ツールの開発

350702101　後藤 隼弐