TLV変換ルール・可視化ルールマニュアル

平成21年8月26日

目 次

第 1 章　概要 **2**
1.1 TraceLogVisualizer の全体像 2
1.1.1 JSON 2

第 2 章　変換ルール **7**
2.1 標準形式トレースログ 7
2.1.1 トレースログの抽象化 7
2.1.2 標準形式トレースログの定義 10
2.1.3 標準形式トレースログの例 11
2.2 標準形式への変換 12
2.2.1 トレースログファイル 14
2.2.2 リソースヘッダファイル 14
2.2.3 リソースファイル 18
2.2.4 変換ルールファイル (JSON 形式) 20
2.2.5 変換ルールファイル (外部プロセス形式) 23

第 3 章　可視化ルール **24**
3.1 可視化表示の仕組みの抽象化 24
3.1.1 可視化表現 24
3.1.2 図形とイベントの対応 26
3.2 図形データの生成 27
3.2.1 可視化ルールファイル 28
3.2.2 可視化ルールファイル (外部プロセス形式) 36

第1章　概要

1.1　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 1.1 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 1.2 に，TLV のスクリーンショットを示す．

1.1.1　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)[1] と呼ばれるデータ記述言語を用いて記述する．

JSON は，主にウェブブラウザなどで使用される ECMA-262，revision 3 準拠の JavaScript（ECMAScript）と呼ばれるスクリプト言語のオブジェクト表記法をベースとしており，RFC 4627 としてで仕様が規定されている．JSON は Unicode のテキ

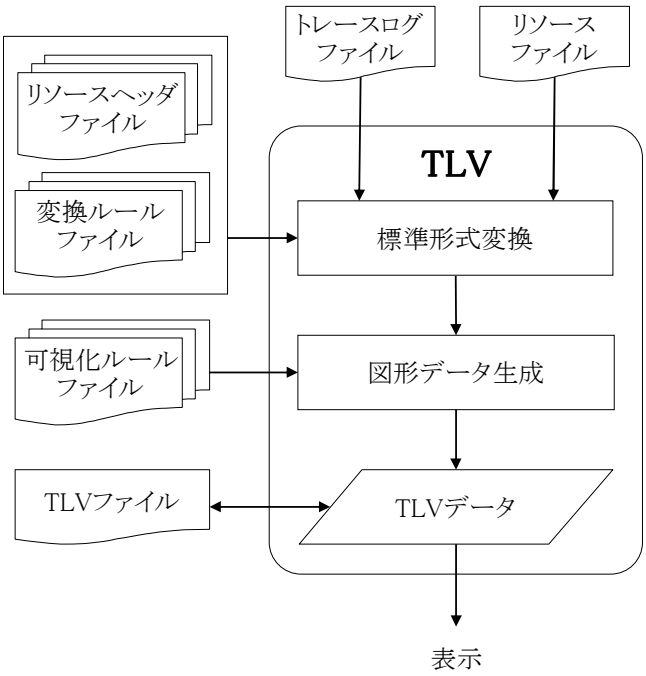

図 1.1: TLV の全体像

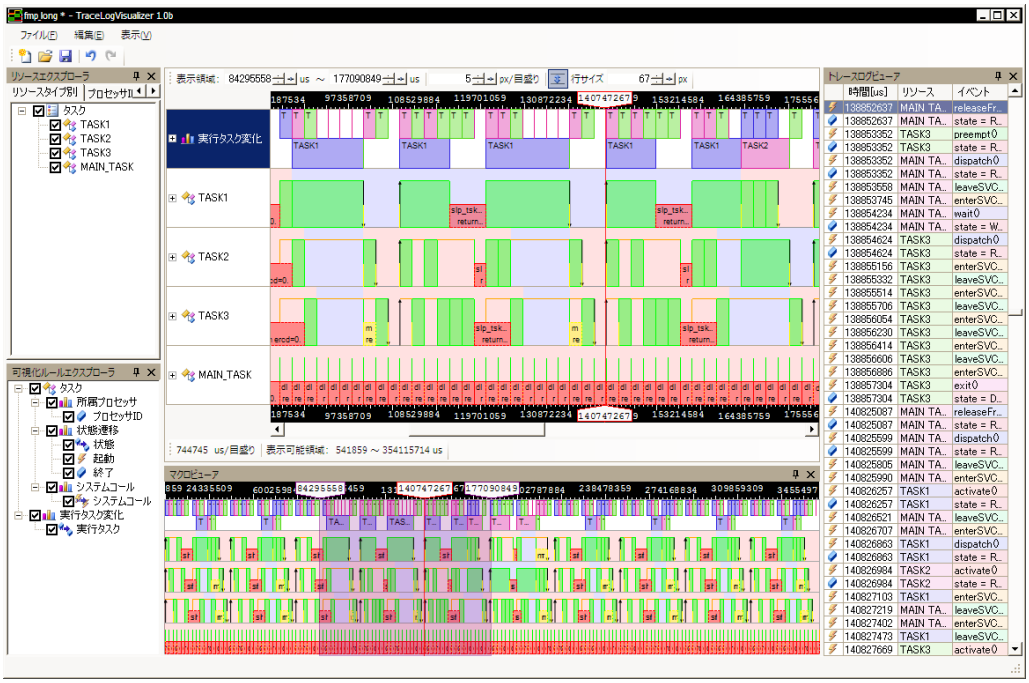

図 1.2: TLV のスクリーンショット

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 1.1 に JSON におけるオブジェクトを定義した例を示す.
表 1.2 に JSON における配列を定義した例を示す.

表 1.1: JSON におけるオブジェクトを定義した例

```
{
  "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 1.2: JSON における配列を定義した例

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

第2章　変換ルール

2.1　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．

2.1.1　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．

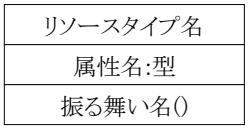

図 2.1: リソースタイプ

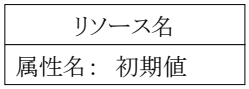

図 2.2: リソース

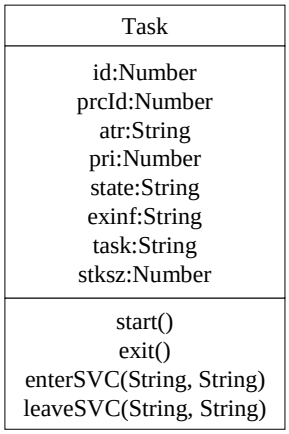

図 2.3: タスクをリソースタイプ Task として表現した例

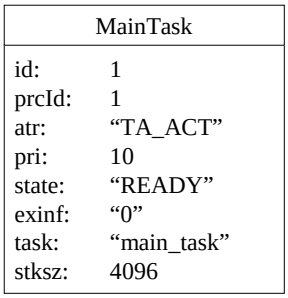

図 2.4: リソースタイプ Task のリソース MainTask の例

2.1.2 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．

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

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

表 2.1: 標準形式トレースログの例

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

表 2.2: 変換元となる TOPPERS/ASP カーネルのトレースログの例

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

2.2 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 2.2.4 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 [2] 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 2.2 に示す．

表 2.2 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 2.3 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 2.2 で示す元のトレースログが 3 行なのに対し，表 2.3 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 2.3 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 2.3 の 1 行目は，表 2.2 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 2.3 の 3 行目は，表 2.2 の 2 行目に対応しており，すでに

表 2.3: 表 2.2 を標準形式トレースログで表現した例

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

起動しているタスクが横取りされて状態が READY 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 2.3 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 READY 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 2.3 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 2.2.4 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 2.2.3 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 2.2.2 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．

表 2.4: TOPPERS/ASP カーネルのトレースログの例

```
 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.
```

2.2.1 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．

表 2.4 に，RTOS である TOPPERS/ASP カーネルのトレースログの例を示す．

2.2.2 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 2.5 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

表 2.5: リソースヘッダファイルの例

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

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．

2.2.3 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2.6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．

表 2.6: リソースファイルの例

```
{
  "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" :"READY"
      }
    }
  }
}
```

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

表 2.7: 変換ルールファイルの例

```
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=READY"
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 && ${state}==READY"
16           :"[${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 }
```

2.2.4 変換ルールファイル (JSON 形式)

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 2.7 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 2.7 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
```

2.2.5 変換ルールファイル (外部プロセス形式)

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外部プロセスも用いることも可能である．外部プロセスを指定する方法は，別途「スクリプト拡張マニュアル」において説明する．

第3章　可視化ルール

3.1　可視化表示の仕組みの抽象化

前章では，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依存するように設計されている．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について述べる．

3.1.1　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 3.1 に座標系の例を，図 3.2 にワールド変換の例を示す．

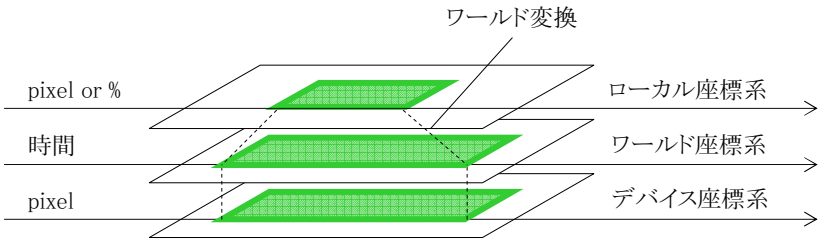

図 3.1: 座標系

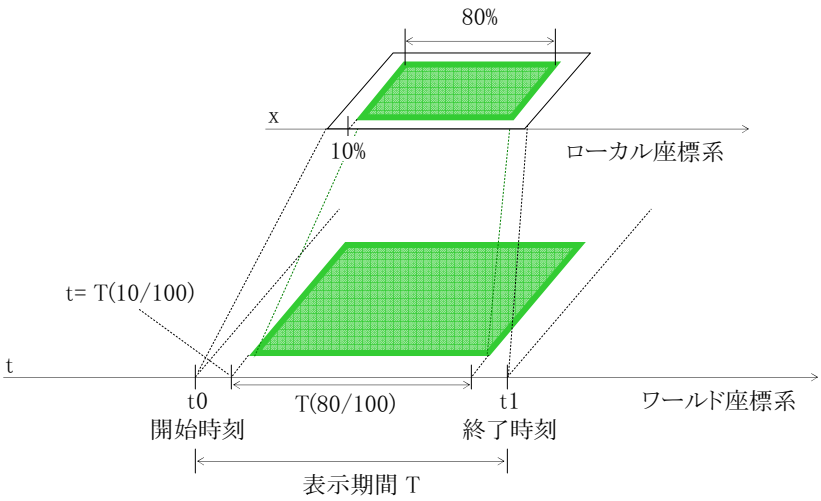

図 3.2: ワールド変換

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 3.3 に図形と図形群の例を示す．

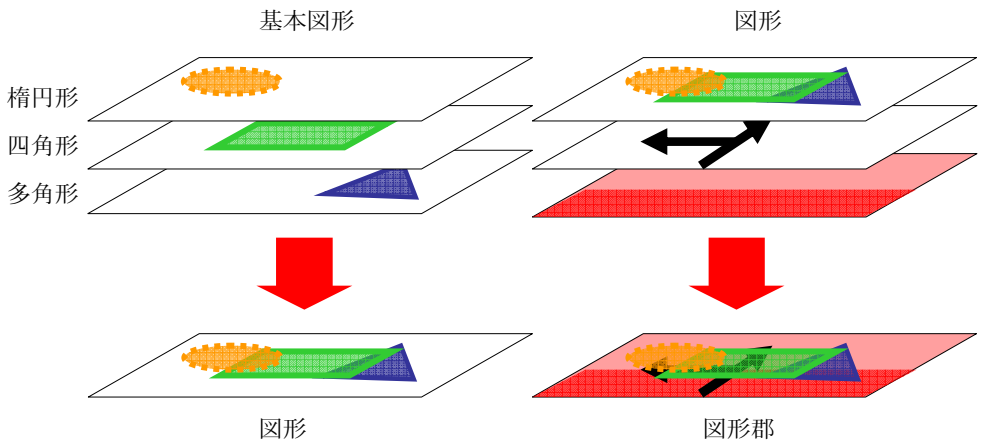

図 3.3: 図形と図形群

3.1.2 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 3.4 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 3.4 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．

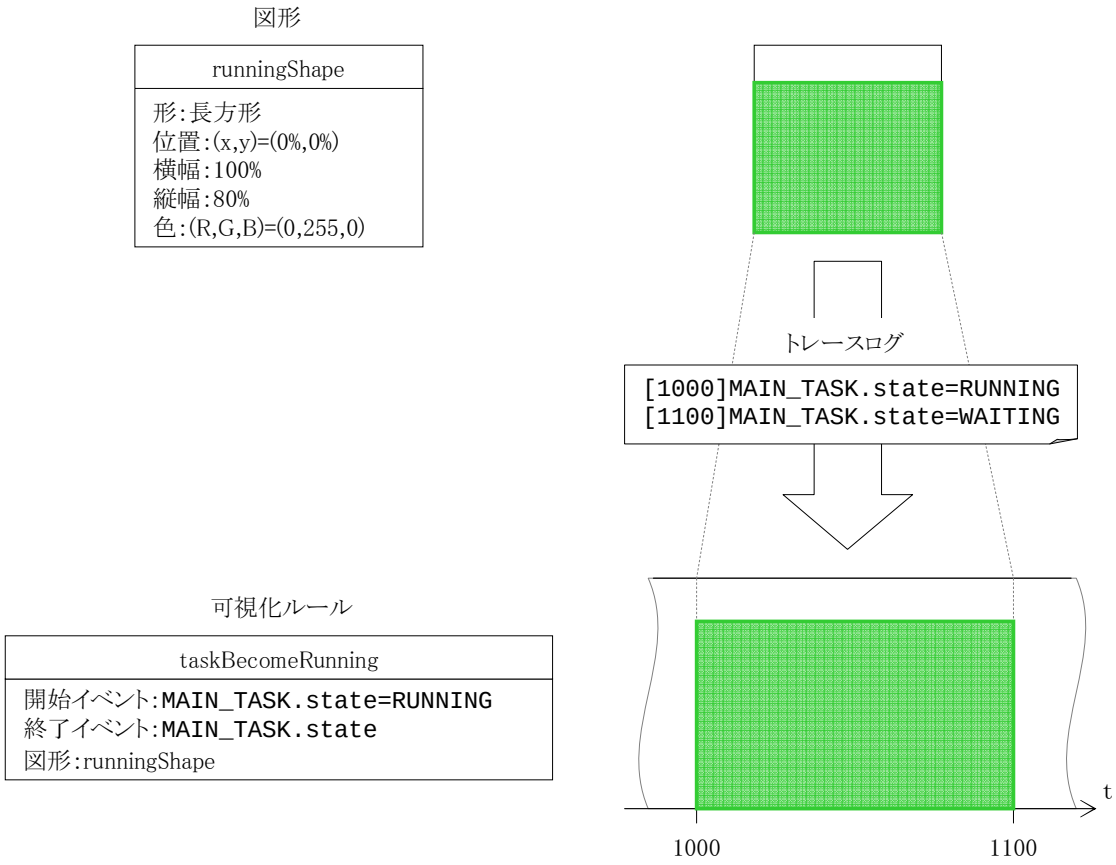

図 3.4: 可視化ルール

表 3.1: イベント期間を抽出するトレースログ

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

taskBecomeRunning を，表 3.1 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 3.4 の右下に示すものである．

3.2 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．

3.2.1 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，3.1.1 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3.2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 readyShapes，svcShapes の3つの図形を定義している．runningShapes と readyShapes 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

表 3.2: 可視化ルールファイルで図形を定義した例

```
{
  "toppers":{
    "Shapes":{
      "runningShapes":[
        {
          "Type":"Rectangle",
          "Size":"100%,80%",
          "Pen":{"Color":"ff00ff00","Width":1},
          "Fill":"6600ff00"
        }
      ],
      "readyShapes":[
        {
          "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}"
        }
      ]
    }
  }
}
```

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)?/
```

3.1.1 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 線の透明度．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

3.1.1 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，”ARGn”で参照できることとした．

表 3.2 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 (JSON 形式)

可視化ルールの定義は，3.1.1 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 3.3 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

表 3.3: 可視化ルールファイルで可視化ルールを定義した例

```
{
  "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}==READY":"readyShapes"
            }
          },
          "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})"
            }
          }
        }
      }
    }
  }
}
```

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 3.3 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_ARGn}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．

3.2.2 可視化ルールファイル (外部プロセス形式)

標準形式トレースログに図形に変換するため外部プロセスも用いることも可能である．外部プロセスを指定する方法は，別途「スクリプト拡張マニュアル」において説明する．

関連図書

[1] 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

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