トレースログ可視化ツール
TraceLogVisualizer（TLV）

後藤 隼弐　本田 晋也　長尾 卓哉　高田 広章

マルチコア環境で動作するソフトウェアのデバッグには，トレースログの解析が有効であるが，開発者がトレースログを直接扱うのは非効率的である．そのため，トレースログを可視化することにより解析を支援するツールが存在するが，これらは汎用性や拡張性に乏しいという問題がある．そこで我々は，これらの問題を解決したトレースログ可視化ツール（TLV）を開発した．TLV は，変換ルールによりトレースログを標準形式に変換することで汎用性を実現し，可視化ルールにより可視化表示項目の追加・変更を可能にすることで拡張性を実現している．TLV を用いて形式の異なる 3 種類のトレースログを可視化し，汎用性と拡張性を確認した．

It is effective to debug software executed in multi-core processors by analyzing a trace log, but it is inefficient that developers analyze the trace log directly. Therefore, some visualization tools for the trace log had been developed before now, but there are two problems, lacks of general versatility and expandability. To that end, we developed TraceLogVisualizer(TLV), a visualization tool for the trace log that solved those problems. In this paper, we will discuss requirements to achieve general versatility and expandability, and how we achieved them.

1　はじめに

近年，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．マルチコア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，マルチコアプロセッサ環境で動作するソフトウェアのデバッグを行うのは，シングルプロセッサ環境に比べ困難であるという問題がある．これは，各コアが非同期的に並列動作するため，タイミングに依存した再現性の低いバグが発生する可能性があることや，ハードウェアの制約からコア間の同期精度が低い場合があり，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はない場合があるからである．

一方，マルチコア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ある．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を行う際に必要な情報である，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を知ることができるからである．しかしながら，開発者がトレースログを直接解析するのは効率が悪い．図 1 に，2 コア上で動作する RTOS（Real-Time Operating System）が出力するトレースログの例を示す．図 1 の例では，約 430μs の間に 5 行のトレースログが出力されており，トレースログを収集する時間が長くなれば膨大な量になることを示唆している．取得する情報の粒度を細かくしたり，処理の複雑さが増すと，出力されるトレースログ

Visualization Tool for Trace Log. TraceLogVisualizer（TLV）

Junji Goto, Hiroaki Takada, 名古屋大学 大学院情報科学研究科, Graduate School of Information Science, Nagoya University. Shinya Honda, Takuya Nagao, 名古屋大学 大学院情報科学研究科 附属組込みシステム研究センター, Center for Embedded Computing Systems, Graduate School of Information Science, Nagoya University

コンピュータソフトウェア, Vol.16, No.5 (1999), pp.78–83.
[ソフトウェア論文] xx 年 x 月 xx 日受付.

[*time(μs)*]:[*core_id*]: *information*

```
[60692484]:[1]: task 4 becomes WAITING.
[60692586]:[2]: leave to sns_ctx state=0.
[60692708]:[1]: dispatch from task 4.
[60692798]:[2]: enter to get_pid p_prcid=304.
[60692914]:[1]: dispatch to task 1.
```

図 1　2 コア上で動作する RTOS が出力するトレースログの例

の量はさらに増大するため，デバッグを行う際に必要な情報を探し出すのが困難となる．また，各コアのトレースログは時系列に分散して記録されているため，コア毎の動作を解析するのが困難である．

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あり，これまでにいくつか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 [1] [2] [3] [4] や Unix 系 OS のトレースログプロファイラ [5] [6] などが存在する．しかしながら，これら既存のツールは，特定の環境（OS やデバッグハードウェア）を対象としているため，可視化対象となるトレースログの形式が固定されている．そのため，他の環境が出力するトレースログを可視化することができず，汎用性に乏しいという問題がある．また，可視化表示する項目やその表現方法が固定である場合が多く，拡張性に乏しいという問題もある．

そこで我々は，これらの問題を解決し，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．開発にあたり，我々は，まず，TLV 内部でトレースログを一般的に扱えるよう，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．さらに，任意の形式のトレースログを標準形式に変換するためのルールを変換ルールとして形式化し，ユーザが外部から指定できる仕組みを提供した．次に，トレースログの可視化表示項目やその表現方法を指示する仕組みの抽象化を行い，可視化ルールとして形式化した．また，可視化ルールをユーザが外部から指定できる仕組みを提供した．TLV では，環境毎に変換ルールと可視化ルールを外部ファイルとして用意することで様々な環境に対応することが可能であり，これにより，汎用性と拡張性を実現している．

本論文は次のように構成される．2 章では，汎用性と拡張性を実現するためのトレースログ可視化ツールの要件と，TLV での実現方針について述べる．3 章では，要件と実現方針に基づく TLV の設計について述べ，4 章では，その実装について述べる．5 章では，複数のトレースログの可視化表示や，可視化表示項目の変更，追加を行い，TLV の汎用性と拡張性を示す．最後に，6 章でまとめと今後の課題，展望を述べる．

2　要件と実現方針

本章では，トレースログ可視化ツールが汎用性と拡張性を備えるための要件と，TLV での実現方針についてそれぞれ説明する．

まず，トレースログ可視化ツールの定義，ならびにその汎用性と拡張性を明確にする．本研究では，トレースログ可視化ツールを，時系列に記録されたプログラムの実行履歴をテキスト形式でファイル化したトレースログファイルを読み込み，その内容を GUI を通じて時系列に可視化表示するツールとする．次に，トレースログ可視化ツールの汎用性を，入力するトレースログファイルの形式を制限しないこととする．これは，ユーザがツールのバイナリを変更することなく，様々な形式のトレースログファイルを可視化表示できることを指す．また，拡張性を，ユーザがツールのバイナリを変更することなく，容易に可視化表示項目の追加・変更・削除ができることとする．可視化表示項目とは，トレースログの内容や出力元の環境，解析の目的などに基づき区分される可視化表示する情報の単位である．具体的には，出力元の環境が OS の場合，“タスクの状態遷移” や “タスクのコア占有率” などが該当する．

2.1　汎用性の実現

はじめに，本研究では，汎用性を実現するため，トレースログの標準形式を提案する．そして，任意の形式のトレースログを，標準形式に変換するためのルールを変換ルールとして形式化し，ユーザが外部から指定できる仕組みを提供する．

トレースログ可視化ツールが汎用性を実現するた

めには，可視化表示の仕組みが特定のトレースログの形式に依存しないようにしなければならない．そのためには，ツールがトレースログを一般的に扱えなければならず，トレースログの形式を標準化する必要がある．標準化されたトレースログの形式を標準形式と呼ぶ．つまり，汎用性を実現するためには，可視化表示の仕組みが標準形式にのみ依存するようにし，任意の形式のトレースログを標準形式のトレースログに変換する仕組みを提供すればよい．

標準形式への変換処理は，逐次的なテキストの置き換えといった，単純なテキスト処理では要件を満たせない場合がある．たとえば，変換元のトレースログが暗に含む情報を，可視化のために標準形式として出力する必要がある場合，機能的に不十分になる場合が考えられる．具体例として，RTOS が出力するトレースログを，標準形式に変換する場合を考えてみる．RTOS においては,「あるタスクがディスパッチされた」という情報（以下，A とする）は，暗に,「実行中のタスクがプリエンプトされた」という情報（以下，B とする）を含んでいる．しかしながら，RTOS の実装によっては，トレースログを出力するコストを減らすため，A のみをトレースログとして出力し，B を出力しない場合がある．このとき，B を標準形式トレースログとして出力したい場合，A のトレースログを変換する際に，A が出力された時刻に実行状態であるタスクが存在するかどうか，また，存在する場合はどのタスクであるのか，という情報が必要となる．そのため，これらの情報を，変換元のトレースログファイルの内容から判断する必要がある．つまり，標準形式への変換処理においては，変換元のトレースログが暗に含む情報を出力するため，それを出力する条件の制御，また条件を制御するために必要な情報を取得する手段が必要である．

2.2 拡張性の実現

拡張性を実現するためには，汎用的な可視化表示の仕組みを提供し，ユーザが可視化表示項目に合わせてそれを制御できればよい．つまり，可視化表示項目毎に可視化表示の仕組みを用意するのではなく，汎用的な可視化表示の仕組みに対して，可視化表示項目を実現するパラメータを指定することで可視化表示を行う．そして，ユーザがパラメータを指定する手段を実現すればよい．その際，汎用性の実現のため，汎用的な可視化表示の仕組みは，標準形式トレースログにのみ依存するようにしなければならないことを前節で述べた．

また，指定するパラメータは，描画処理に必要な GUI フレームワークや表示デバイスに依存しないことが要求される．これは，プログラムの可搬性を損なわないためである．

本研究では，汎用的な可視化表示の仕組みを実現するために，まず，トレースログの内容から可視化表現を決定し，それを表示するまでの流れを抽象化し，本質的な処理を洗い出す．次に，ユーザが，どの処理に，どのようなパラメータを指定できればよいかを判断し，可視化ルールとして形式化する．そして，ユーザが外部から可視化ルールを指定できる仕組みを提供する．

3 設計

本章では，前章で汎用性を実現するための手段として挙げた標準形式と変換ルールと，拡張性を実現するための手段である可視化表示の仕組みの設計について説明する．

3.1 標準形式

トレースログを一般化するため，RTOS や Unix 系 OS，ISS（Instruction Set Simulator）などのトレースログの形式の調査を行った．その結果から，次のようにトレースログを一般化した．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した．イベントとはイベント発生源の事象であり，イベント発生源の属性の変化，または振る舞いとした．ここで，イベント発生源をリソースと呼称し，固有の識別子として名前をもつとした．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である．また，リソースは，型により属性，振る舞いを特徴付けられるとした．ここで，リソースの型をリソースタイプと呼称し，固有の識別子として名前をもつとした．属性は，リソースが固有にもつ文

```
1  TraceLog = { TraceLogLine,"\n" };
2  TraceLogLine = "[",Time,"]",Event;
3  Time = /[0-9a-zA-Z]+/;
4  Event = Res,".",(AttrChange|BhvrHappen);
5  Res = ResName|ResTypeName,"(",AttrCond,")";
6  ResName = Name;
7  ResTypeName = Name;
8  AttrCond = BoolExp;
9  BoolExp = Bool|CompExp
10    |BoolExp,[{LogclOpe,BoolExp}]
11    |"(",BoolExp,")";
12 CompExp = AttrName,CompOpe,Value;
13 Bool = "true"|"false";
14 LogclOpe = "&&"|"||";
15 CompOpe = "=="|"!="|"<"|">"|"<="|">=";
16 AttrName = Name;
17 AttrChange = AttrName,"=",Value;
18 Value = /[^"\\]+/;
19 BhvrHappen =  BhvrName,"(",Args,")";
20 BhvrName = Name;
21 Args = [{Arg,[","]}];
22 Arg = /[^"\\]*/;
23 Name = /[0-9a-zA-Z_]+/;
```

図 2　標準形式の定義

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した．それぞれは固有の識別子として名前をもつ．

例として，RTOS が出力する，「TASK1 という名前のタスクの状態が，実行状態に遷移した」という情報のトレースログを，一般化した方法で考えてみる．この場合，TASK1 がリソースであり，タスクがリソースタイプであると考えることができる．また，タスクの状態は属性にあたり，その属性が実行状態という値に変化したというイベントであると言える．

図 2 に，一般化の結果を形式化したトレースログの標準形式の定義を示す．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ている．正規表現はスラッシュ記号（/）で挟み記述している．

標準形式において，リソースの記述方法（図 2：5 行目）は，リソースの名前（ResName）による直接指定か，リソースタイプの名前と属性の条件（ResTypeName(AttrCond)）による条件指定の 2 通りの方法で指定できるとした．条件指定では，リソースタイプとリソースの属性の値から，特定のリソース，または複数のリソースを表現することができる．条件指定は，2.1 節で述べた，変換元のトレースログが暗に含む情報の表現を可能にするため導入した．属性の値の変更と振る舞いの発生の記述方法は，Java や C++などのオブジェクト指向言語における，メンバ変数への代入と，メソッドの呼び出しの記法と同様である．

図 3 に，RTOS が出力するトレースログを標準形式に変換した標準形式トレースログの例を示す．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 行目は，時刻 2403010 に，MAIN_TASK という名前のリソースが leaveSVC という振る舞いを，ena_tex と ercd=0 を引数として発生したことを表現している．これは，タスクである MAIN_TASK が，ena_tex というシステムコールを，エラーコード 0 でリターンしたことを意味している．

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．3 行目は，時刻 4496802 に，その時点でリソースタイプが TASK で属性 state が RUNNING であるリソースの，属性 state が READY になったことを表現しており，実行状態であるタスクが実行可能状態になった，というイベントを意味している．

3.2　変換ルール

変換ルールは，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ためのルールであり，これらの対応関係を定義したものである．標準形式への変換は，変換ルールに基づき，テキストの変換処理を行う処理である．

2.1 節において，変換処理の要件として，変換元のトレースログが暗に含む情報を出力するため，それを出力する条件の制御，また条件を制御するために必要な情報を取得する手段が必要であることを述べた．TLV では，出力する条件の制御を，指定された時刻の特定リソースの有無や数，属性の値，属性値変更イベントであれば変更後の値，振る舞いイベントであれば，その引数などを用いて論理式を記述することで行う．論理式の記述は，それらの値を取得できるマクロを用いて行い，その際の記法には標準形式を用いる．たとえば，図 3 の 3 行目を出力する条件が，「時刻 4496802 にリソースタイプ

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

図 3 標準形式トレースログの例

が TASK で属性 state が RUNNING であるリソースが存在する場合に出力する」である場合，条件には，$EXIST{[4496802]TASK(state==RUNNING)}と記述すればよい．このとき，$EXIST{*resource*}が，リソース *resource* の有無を真偽値で取得するマクロである．

3.3 可視化表示の仕組み

可視化表示の仕組みは，汎用性の実現のため，標準形式にのみ依存するようにしなければならない．また，拡張性の実現のためには，可視化表示の仕組みとして汎用的なものを提供し，ユーザが可視化表示項目に依存するパラメータを指示できる仕組みを提供すればよい．

本節では，はじめに，トレースログの内容から可視化表現を決定し，それを表示するまでの流れを抽象化し，可視化表示の仕組みとして本質的な処理を洗い出す．そして，ユーザが，どの処理に，どのようなパラメータを指定すればよいかを判断する．

トレースログ可視化ツールにおいて，可視化表示とは，可視化表示項目毎に要求される表現（以下，可視化表現とする）を，トレースログの内容に従い時系列の図として画面に描画する処理であると考えられる．ここで，画面への描画は，GUI フレームワークに依存するため，可視化表示の仕組みとしては本質的ではない．そのため，可視化表示の仕組みは，可視化表現を，時間と高さを次元にもつ仮想的な座標に割り当てる処理であると抽象化できる．ここで，可視化表現は，複数の図形で構成されるものとする．図 4 に，抽象的な可視化表示の仕組みを示す．

図形は，楕円や四角などの複数の基本図形で構成されるものと考えることができる．このとき，図形を定義する座標系をローカル座標系と呼称する．また，図形をトレースログの内容に従い配置する座標である，時間と高さを次元にもつ仮想的な座標をワールド座標系と呼称する．ワールド座標系へ割り当てられた図形は，表示開始時刻と単位時間あたりのピクセル数により表示デバイスに表示される．このとき，表示先の座標系をデバイス座標系と呼称する．単位時間あたりに，何ピクセル数でワールド座標系をデバイス座標系に割り当てるかという処理が，表示上の拡大，縮小の処理となる．

以上の，可視化表示の仕組みの抽象化から，ユーザが任意の可視化表示項目を実現するために，汎用的な可視化表示の仕組みに対して与える指示としては，どのような図形を，どの時間領域に割り当てるかということであると考えられる．その際，図形を定義するには，形状や大きさ，位置，色，透明度などが指示できればよい．一方，図形をどの時間領域に割り当てるか，つまり，図形をワールド座標系のどの領域に割り当てるかの指示は，割り当て先の領域に対して一般的でなければならない．つまり，割り当て先の個々の領域をすべて指定するのではなく，ルールに基づいて割り当てが行われるように指定されるべきである．このルールを可視化ルールと呼称する．

TLV では，可視化ルールとして，表示領域の時間をイベントを用いて指定することとした．その際，イベントの記述に標準形式を用いる．つまり，時間領域の指定に，開始時刻と終了時刻ではなく，指定の開始イベントと終了イベントを記述することで行う．標準形式トレースログに可視化ルールを適用する際に，指定されたイベントに一致するイベントを標準形式トレースログから探し，そのイベントの時刻を割り当て先の時間領域として採用することで，ワールド座標系への割り当てを行う．

4 実装

本章では，前章で行った設計を元に，TLV をどのように実装したかを述べる．具体的には，全体の構成と処理の流れから，変換ルールや可視化ルールをどのように形式化したか，また，それらをどの段階で適用するのかを説明する．TLV の実装言語は C#3.0 であり，統合開発環境として Microsoft 社の Visural Studio 2008 Professional Edition を用いた．

TLV は，2 つの主たる処理と 6 種の外部ファイルによって構成される．図 5 に TLV の構成を示す．

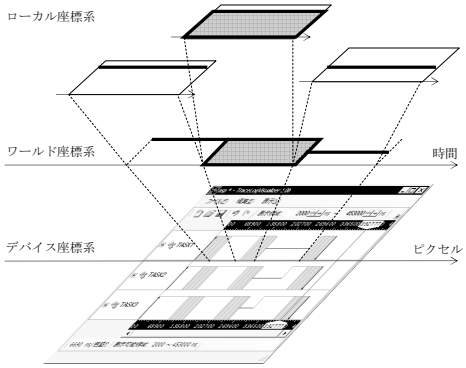

図 4　抽象的な可視化表示の仕組み

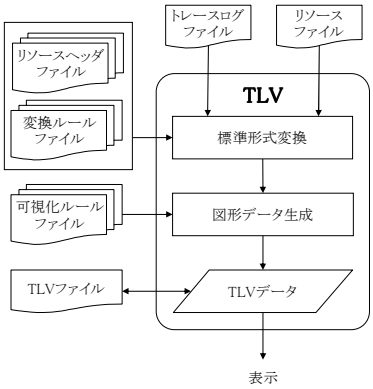

図 5　TLV の構成

トレースログを可視化表示する際，ユーザは，トレースログファイルとリソースファイルを用意する．トレースログファイルは，可視化対象となる環境が出力した，任意の形式のトレースログをファイル化したものである．また，リソースファイルは，トレースログの形式や，トレースログに登場するリソースの定義，トレースログの時刻の単位や基数，適用する変換ルールと可視化ルールの指定などを記述したファイルである．リソースファイルの例を図 6 に示す．

リソースヘッダファイルと変換ルールファイルは，トレースログを出力する環境毎に用意する．リソースヘッダファイルはリソースタイプの定義を記述したファイルであり，変換ルールファイルは，変換ルールを記述したファイルである．リソースヘッダファイルの例を図 7 に示す．変換ルールファイルの例を図 8 に示す．

可視化ルールファイルは，可視化表示項目毎に図形と可視化ルールの定義を記述したファイルであり，複数の可視化表示項目を一つのファイルでまとめて定義することができる．可視化表示項目を追加したければ，新たな可視化ルールファイルを用意すればよい．

TLV を構成する 2 つの主たる処理は，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，任意の形式のトレースログに対して，変換ルールを適用することで標準形式のトレースログに変換する処理である．この処理では，ユーザがトレースログファイルとリソースファイルを入力し，リソースファイルの定義によりリソースヘッダファイル，変換ルールファイルが自動で読み込まれる．

図形データの生成は，変換され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リソースファイルの定義に従い読み込まれる．

それぞれの処理で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，2 つの処理を省略して可視化表示することができる．

TLV が扱う 6 種のファイルのうち，トレースログファイル以外は JSON（JavaScript Object Notation）[8] 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 の特徴は，構文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が容易であり，コンピュータにとっても解析が容易であることを意味する．これにより，ユーザの記述コスト，習得コスト，またファイルの解析を行う実装コストを低減することができる．

```
1  { "TimeScale" :"us", "TimeRadix" :10,  /* 時刻の単位と基数 */
2    "ConvertRules"   :["asp"],                 /* 変換ルール */
3    "VisualizeRules" :["toppers","asp"],     /* 可視化ルール */
4    "ResourceHeaders":["asp"],               /* リソースヘッダ */
5    "Resources":{                      /* リソースの定義始まり */
6      "TASK1":{          /* TASK1 という名前のリソースを定義 */
7        "Type":"Task",                       /* リソースタイプ */
8        "Color":"ff0000",        /* リソース固有の色 (GUI 用) */
9        "Attributes":{                 /* 属性の初期値を指定 */
10         "id"    :1,
11         "atr"   :"TA_NULL",
12         "pri"   :10,
13         /* ... 省略 ... */
14         "state" :"DORMANT"
15       }
16     },
17     /* ... 省略 ... */
```

図 6　リソースファイルの例の一部

```
1  { "asp":{
2      "Task":{                        /* リソースタイプの名前 */
3        "DisplayName":"タスク",       /* 表示名 */
4        "Attributes":{                /* 属性の定義始まり */
5          "id":{                      /* 属性名 */
6            "VariableType"  :"Number",/* 属性の型 */
7            "DisplayName"   :"ID",    /* 属性の表示名 */
8            "AllocationType":"Static",/* 値は動的か静的か */
9            "CanGrouping"  :false /* 同じ属性値同士でグループ */
10         },                        /* を構成できるか (GUI 用) */
11         /* ... 省略 ... */
12       },                            /* 属性の定義終わり */
13       "Behaviors":{                 /* 振る舞いの定義開始 */
14         "start":{"DisplayName":"起動"},/* タスクの起動 */
15         "exit":{"DisplayName":"終了"}, /* タスクの終了 */
16         /* ... 省略 ... */
17         "enterSVC":{          /* システムコールの呼び出し */
18           "DisplayName":"サービスコールに入る",
19           "Arguments":{"name":"String","args":"String"}
20         },
21         /* ... 省略 ... */
```

図 7　リソースヘッダファイルの例の一部

4.1　標準形式変換

標準形式への変換は，変換ルールを変換元のトレースログに適用することで行う．このとき，変換ルールの定義は，変換ルールファイルに記述する．

標準形式へ変換する際，リソースファイルを読み込むことでトレースログの形式や，扱うリソースの情報，適用するリソースヘッダや変換ルールなどを取得する．また，扱うリソースのリソースタイプを定義したリソースヘッダファイルを読み込み，リソースの属性の初期値や，属性の型などの情報を得る．

図 7 に，RTOS におけるタスクを，リソースタイプとして定義しているリソースヘッダファイルの例の一部を示す．2 行目から，タスクを `Task` という名前のリソースタイプとして定義している．4 行目から属性の定義をしており，13 行目から振る舞いの定義を記述している．ここでは，数値型の `id` という属性とタスクが起動したことを表す `start`，終了したことを表す `exit`，システムコールに入ったことを表す `enterSVC` という振る舞いが定義されていることがわかる．

図 6 に，図 7 で示される `Task` というリソースタイプのリソースを定義した，リソースファイルの例の一部を示す．6 行目に `TASK1` という名前のリソースが定義されており，9 行目から属性の初期値の値が定義されている．

既存のテキスト処理するためのスクリプト言語として，awk などがあるが，2.1 節で述べた要件を満たすためには，リソースヘッダファイルで定義されるリソースの属性の初期値や型などの情報を参照したり，リソース毎に属性の値の遷移を監視する必要があるなど，多くの複雑な記述をする必要があり，実用的ではない．そのため，TLV では，3.1 節の設計で述べたとおり，特定リソースの有無や数，属性の値の遷移などから標準形式トレースログの出力を制御する仕組みを変換ルールに提供した．具体的には，指定された時刻の特定リソースの有無や数，属性の値などを参照することができるマクロを提供した．このマクロを用いて標準形式トレースログの出力を制御することで要件を満たすことができる．図 8 に変換ルールファイルの例の一部を示す．

図 8 の 2 行目は，元のトレースログファイルから任意のトレースログを検索するための正規表現であり，一致した場合に 3〜6 行目で指定される標準形式トレースログを出力する．3 行目は，出力する条件であり，条件中に`$EXIST{`*resource*`}`というマクロを用いてリソース *resource* の有無を判定している．条件が真の場合に 4 行目の標準形式トレースログが出力される．5 行目の標準形式トレースログは，2 行目の正規表現に一致した際に必ず出力される．

4.2　図形データ生成

図形データの生成とは，3.2 節で述べたとおり，標準形式トレースログの内容に従い，ワールド座標系に図形を割り当てる処理である．図形データの生成は，

```
{"asp":{
  "\[(?<t>\d+)\] dispatch to task (?<id>\d+)\.":[{
    "$EXIST{[${t}]Task(state==RUNNING)}":
      "[${t}]$RES_NAME{[${t}]Task(state==RUNNING)}.state=READY
    },
    "[${t}]$RES_NAME{Task(id==${id})}.state=RUNNING"
  ],
  /* ... 省略 ... */
```

図 8　変換ルールファイルの例の一部

```
{ "toppers":{
    "Shapes":{                          /* 図形の定義始まり */
      "runningShapes":[{
          "Type":"Rectangle",           /* 形状の指定（長方形）*/
          "Size":"100%,80%",            /* 大きさの指定 */
          "Pen":{"Color":"ff00ff00","Width":1},/* 線種の指定 */
          "Fill":"6600ff00"             /* 塗りつぶしの指定 */
        }],
      /* ... 省略 ... */
    },                                  /* 図形の定義終わり */
    "VisualizeRules":{          /* 可視化ルールの定義始まり */
      "taskStateChange":{
        "DisplayName":"状態遷移", /* 可視化ルールの表示名 */
        "Target":"Task",          /* 適用するリソースタイプ */
        "Shapes":{                  /* 表示項目の指定 */
          "stateChangeEvent":{
            "DisplayName":"状態", /* 表示項目の表示名 */
            "From":"${TARGET}.state", /* 開始イベント */
            "To"  :"${TARGET}.state", /* 終了イベント */
            "Figures":{                 /* 表示する図形 */
              "${FROM_VAL}==RUNNING" :"runningShapes",
              "${FROM_VAL}==READY":"readyShapes"
              /* ... 省略 ... */
            }
          },
          /* ... 省略 ... */
        }
      },                            /* 可視化ルールの定義終わり */
      /* ... 省略 ... */
```

図 9　可視化ルールファイルの例の一部

図形の定義と可視化ルールの定義に従い，標準形式トレースログに可視化ルールを適用することで行う．

図形と可視化ルールの定義は可視化ルールファイルに記述する．図 9 に可視化ルールファイルの例の一部を示す．2 行目から 10 行目までが図形の定義であり，それ以降が可視化ルールの定義である．

図 9 では，`taskStateChange` という可視化ルールが定義されており，開始イベントと終了イベントとして `Target` で指定されたリソースタイプのリソースの属性の値変更イベントを指定している．そして，ワールド座標系に割り当てる図形を，変更後の属性の値によって場合分けしている．

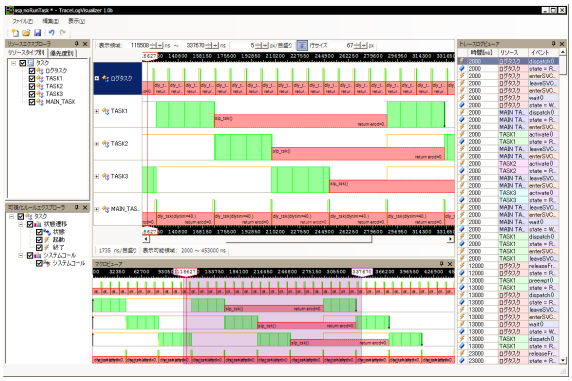

図 10　TOPPERS/ASP カーネルが出力するトレースログを可視化表示した TLV の実行結果のスクリーンショット

5　評価

図 10 に，シングルプロセッサ用 RTOS である TOPPERS/ASP カーネル [9] が出力するトレースログを，TLV を用いて可視化表示した結果のスクリーンショットを示す．タスクの状態遷移やシステムコールの出入り，その際の引数，返値を可視化表示できていることがわかる．このとき，変換ルールは約 28 行であり，可視化ルールは 168 行であった．また，リソースヘッダファイルは 44 行であった．標準形式への変換処理に要した時間は，トレースログファイルが 353 行の場合で約 2 秒であった．

次に，TOPPERS/ASP カーネルをマルチコアプロセッサ用に拡張した TOPPERS/FMP カーネル [9] が出力するトレースログの可視化表示を行った．TOPPERS/FMP カーネルが出力するトレースログの形式は，TOPPERS/ASP カーネルが出力するトレースログの形式に，コアの番号を付加した形式になっている．また，TOPPERS/FMP カーネルの可視化表示項目は，TOPPERS/ASP カーネルの可視化表示項目に加え，実行コアを背景色で区別するように拡張したものが要求される．そのため，TOPPERS/ASP カーネル用の変換ルールと可視化ルールをベースに拡張を行った．その結果，変換ルールファイルは 32 行，可視化ルールファイルは 232 行，リソースヘッダファイルは 50 行となった．図 11 に，TOPPERS/FMP

図 11　TOPPERS/FMP カーネルが出力するトレースログを可視化表示した TLV の実行結果のスクリーンショット

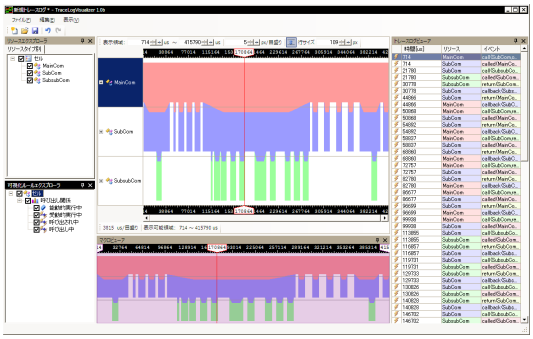

図 12　TECS のトレースログを可視化表示した TLV の実行結果のスクリーンショット

カーネルが出力するトレースログを，TLV 用いて可視化表示した結果のスクリーンショットを示す．タスクの状態遷移を表示している箇所に，背景色で実行コアを区別し表現できていることがわかる．これにより，拡張性が確認できた．

最後に，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）[10] のトレースログを採用した．図 12 に，組み込みコンポーネントシステムである TECS のトレースログを TLV 用いて可視化表示した結果のスクリーンショットを示す．図 12 より，コンポーネントの呼び出し関係が可視化表示できていることがわかる．このとき，変換ルールは 15 行であり，可視化ルールは 88 行であった．また，リソースヘッダファイルは 47 行であった．

6　開発プロセス

6.1　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創造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た．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，筆者を含む学生 3 名と企業出身者 2 名が開発実務を行った．プロジェクトの進捗，成果物，情報の管理は，グループウェアやバグトラッキングシステム，ソースコードリポジトリ，Wiki といったネットワークアプリケーションを用いて行われた．また，週に一度ミーティングを行い，開発実務者による進捗の報告や，プロジェクトマネージャによる開発スケジュールの調整や指導が行われた．

プロジェクトは約 9ヶ月を通じて 3 フェーズに分けて実施された．フェーズを分割した主な理由は，段階的な要求取得，目標設定や人員入れ替え時期，学期との同期のためである．フェーズ 1 はプロトタイプの実装として約 3ヶ月かけて行われ，GUI の評価や要求の再取得，アプリケーションドメインを通じた設計方法の探索を行うことを目的に行われた．

フェーズ 1 は，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し，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など，目的の達成に必要な要件のみの実装に努めた．フェーズ 1 での実装成果物の総行数は約 9000 行 (有効行数 5000 行) であった．フェーズ 1 の成果物は TLV の開発関係者と RTOS 開発者および利用者の数

名に利用され，意見や要求の収集を行った．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．フェーズ 1 では，プロトタイプの実装という位置づけであったため，開発プロセスを設定せずに開発を行った．その結果，実装中心の開発になってしまい，進捗管理が曖昧になってしまった．

フェーズ 2 は約 5ヶ月かけて行われ，標準形式変換ルールファイルによる標準形式トレースログへの変換や，可視化ルールファイルによる可視化表現の外部プラグイン化の実装など，汎用性，拡張性を実現する主機能の実装を目的に行われた．フェーズ 2 では，フェーズ 1 での反省を踏まえ，開発プロセスを導入した．導入した開発プロセスはユースケース駆動のアジャイルソフトウェア開発であり，詳細は次節で後述する．

フェーズ 3 は約 1ヶ月かけて行われ，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の充実を図るとともに，TLV の汎用性，拡張性の確認を行うことを目的として行われた．

6.2 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，ユースケース駆動のアジャイルソフトウェア開発プロセスを用いて行われた．

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としてイテレーションを反復して実施した．図 13 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．

ユースケース駆動アジャイルソフトウェア開発の工程は，機能要求からユースケースを洗い出すことからはじまる．そして，ユースケース毎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，

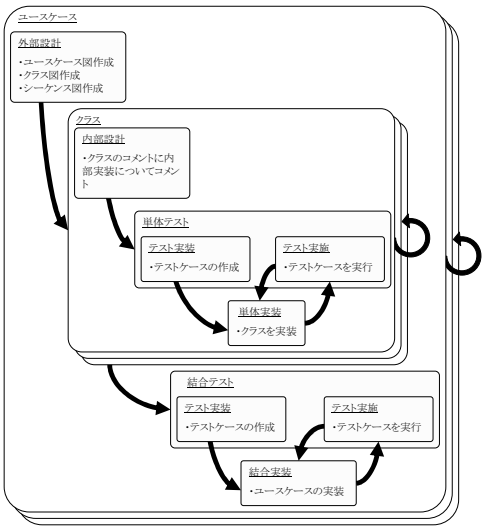

図 13　ユースケース駆動アジャイルソフトウェア開発

単体テスト，実装を行う．内部設計では，文書化せずに，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ソースコードスケルトンを作成し，ソースコード内のコメントとして機能，入出力を記述することで行った．TLV のアジャイル開発では，実装の前にテストケースを作成し，それをパスするように実装を行うテストファースト方式を採用した．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，実施を行った．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追記することで行う．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パスするかどうかを試しながら行う．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をすべてをパスしたら，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．

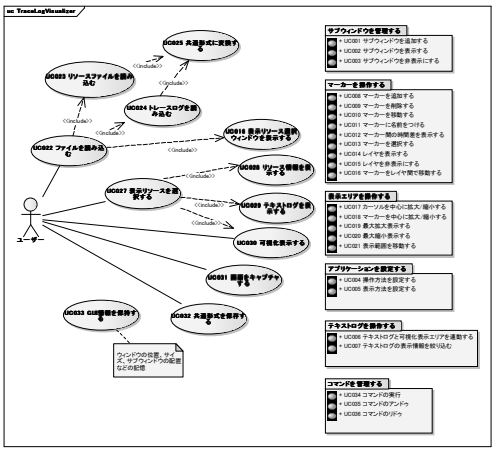

図 14 TLV のユースケース図の例

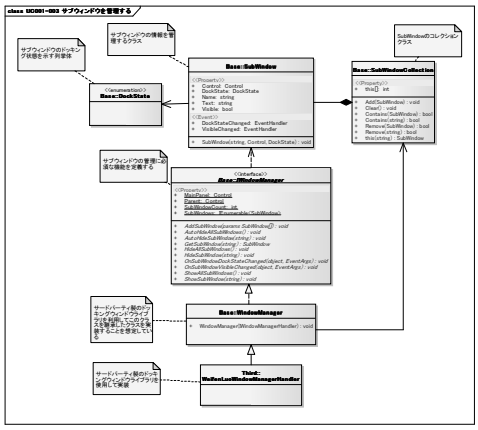

図 15 TLV のクラス図の例

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．

6.2.1 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 14 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，複数のメンバーでユースケースや工程で分担して実施した．

6.2.2 設計

図 13 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の定義をクラス図として，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 15 に作成したクラス図の例を示す．また，図 16 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スケルトンクラスを記述し，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といった説明を記述する．

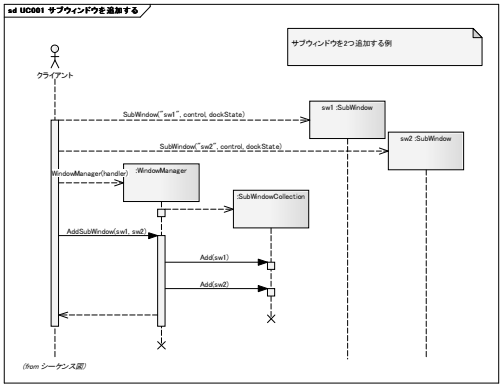

図 16 TLV のシーケンス図の例

6.2.3 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．

6.2.4 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，ユニットテストを全部パス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 1 に示す．

表 1 TLV のソースコードメトリクス

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
インタフェース数	22
列挙体	15
デリゲート	2

7 おわりに

本論文では，トレースログ可視化ツールである TLV について，背景と既存ツールの問題点，それを解決するための要件，要件を満たすための方法とその実装，評価について述べた．

既存ツールの問題点としては汎用性と拡張性の欠如を挙げ，それを解決する要件として，トレースログの標準化と標準形式への変換の仕組みの提供，また可視化表示の仕組みの汎用化とパラメータ指示の形式化が必要であることを述べた．

そして，TLV における要件の実装方法として，標準形式変換と図形データの生成，変換ルールファイル，可視化ルールファイル，リソースヘッダファイル，リソースファイルについて説明した．

最後に 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，変換ルールファイルと可視化ルールファイル，リソースヘッダファイルの拡張によりそれらが実現可能であることを示した．

今後の課題としては，現状の変換ルールや可視化ルールの記述では実現できない，計算の必要な可視化表示項目について検討することが挙げられる．たとえば，RTOS における CPU 使用率やタスクのコア占有率の可視化表示などがある．これには，可視化ルールを現状の単純なイベントの指定ではなく，イベントの状態遷移で指定できるようにする方法や，変換ルールをスクリプト言語化する方法などが考えられる．

参 考 文 献

[1] JTAG ICE PARTNER-Jet, http://www.kmckk.co.jp/jet/.
[2] WatchPoint デバッガ, https://www.sophia-systems.co.jp/.
[3] QNX Momentics Tool Suite, http://www.qnx.co.jp/products/tools/.
[4] eBinder, http://www.esol.co.jp/embedded/ebinder.html.
[5] Mathieu Desnoyers and Michel Dagenais, "OS Tracing for Hardware, Driver and Binary Reverse Engineering in Linux," CodeBreakers Journal Article, vol.4, no.1, 2007.
[6] OpenSolaris Project: Chime Visualization Tool for DTrace, http://opensolaris.org/os/project/dtrace-chime/.
[7] RFC3164 The BSD syslog Protocol, http://www.ietf.org/rfc/rfc3164.txt.
[8] RFC4627 The application/json Media Type for JavaScript Object Notation (JSON), http://tools.ietf.org/html/rfc4627.
[9] TOPPERS Project, http://www.toppers.jp/.
[10] Takuya Azumi, Masanari Yamamoto, Yasuo Kominami, Nobuhisa Takagi, 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.