リアルタイムシステムのための正規表現を用いたログ検索システムの構築とその評価

森本亮太[†]　若林隆行[†]　高田広章[†]

近年組込みシステム開発は，機器の高度化・複雑化が進む一方で，市場に競争力のある製品を投入する為に開発期間の短縮化が要求されている．そのため，組込みソフトウェア開発においてはデバッグ工程の重要性が増している．

組込みシステムの中でも，通常のデバッグツールではデバッグが困難なシステムの検証には，実行時の動作履歴（以下，ログ）を詳細に記録し，終了後にログ解析を行うオフラインデバッグがよく用いられる．しかし，ログ解析は，正常な動作パターンを含む膨大なログの中から不正な動作パターンを探す作業が中心となり，利用者が手作業で解析するのは非常に困難であった．

本研究ではこの問題に対して正規表現で検索条件を記述する手法を提案し，ログ検索システムを構築して評価を行う．

Implementation and evaluation of a log search engine with a regular expression for real-time systems

Ryota Morimoto,[†] Takayuki Wakabayashi [†]
and Hiroaki Takada[†]

In the field of recent development of embedded systems, a period of product development (Time-to-market) is getting shorter to make more competitive products despite these products are getting more complicated. For these reasons, importance of debug process is getting greater to shorten its period.

Offline debugging is often used for checking validity of a embedded system which traditional debugging operations are not applicable to, for example, in case that it can not stop its execution by having reached a breakpoint. However, it is hard for developers to do offline-debugging without aid of a computer due to huge track histories because major work of offline debugging is to find illegal behavior of a target system.

In order to solve the problem, we describe a method of history search by using of regular expression and its evaluation in this paper.

1. 研究背景

近年組込み機器は高度な機能が次々と追加されて高機能になるとともに，従来複数の機器で実現されていた機能を一つの機器に統合するなど複合化が進んでいる．同時に，製品開発から市場に投入されるまでの時間（Time to market）が短期化されている．そのうえ，組込み機器は高い信頼性が要求され，製品出荷後に問題が見つかると製品回収につながる場合が多くある．

組込みソフトウェア開発においても同様に，ソフトウェアの生産性と信頼性の向上が要求されている．なかでもソフトウェア開発の大部分の時間が費やされているデバッグ工程の効率向上が強く要望されていた．

しかし，組込みシステムのなかには，通常のデバッグツールではデバッグが困難なシステムが存在する．このような機器の検証には，実行時の動作履歴（以下，ログ）を詳細に記録し，終了後にログ解析を行うオフラインデバッグがよく用いられる．しかし，ログ解析は，正常な動作パターンを含む膨大なログの中から不正な動作パターンを探す作業が中心となり，利用者が手作業で解析するのは非常に困難である．また，これまでログの解析手法についてはほとんど研究が行われおらず，一般にはシステムごとにそのシステムの動作に基づいて作成されたプログラムで解析を行っていた．

高度な処理を行うためにリアルタイム OS の導入が増加しているが，デバッグ工程においての信頼性やリアルタイム性の検証は非常に時間がかかる．そのため，

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

デバッグ環境の充実が現代の組込み機器開発では重要になっている．

2. 研究の目的

リアルタイムシステムのログ解析において，ログとして記録されているイベント同士の前後関係がシステムの動作を知る上で非常に重要になる．従来，ログ解析時にこれらイベント間の関係を表現する場合にはプログラミング言語を用いて記述がされていた．別の見方をすると，ログ解析のためのツールとその解析アルゴリズムは一体であり，分けることができないということである．

しかも，ログの動作パターンは組込みシステムごと，あるいは，動作状況ごとにまったく異なるため，ログ解析ツールは組込みシステムごとや検索項目ごとに新たに作成されていた．このような状況になっている根本的な原因は，解析ツールと独立にログのパターンを記述する方法が存在しなかったことにある

このようなことから，ログやログのパターンを表現する方法と，ログ解析のアルゴリズムを解析ツールのプログラムから分離する方法が求められていた．

3. 正規表現を用いたログ検索システム

3.1 概　　要

本研究では，ログのパターン表記に正規表現を用いることを提案する．

正規表現は簡単な表記で複雑なパターンを記述できることからテキスト検索の条件の記述などによく用いられている．この性質を利用し，正規表現でログのパターンを記述し検索を行うログ検索システムを構築する．

それと同時に，ログを解析するのに適した方法で扱う手法と，ログ解析後に行う処理についても考察する．また，構築したシステムで実際にログ検索を行い，従来の手法と比較し有用性を確認する．

3.2 提案するログ検索の手順

本研究で提案するログ検索システムの手法の手順を以下に示す．

(1) **ログファイルから個々のイベントを取り出す**

ログファイルにはテキスト形式でイベントが記録されているが，ログファイルは大きい場合は数十メガバイトにもなり，これをそのまま処理するのは困難である．一般に，各ログが持つ情報の大半は数値データであり，これらの情報はテキスト形式から数値に戻して処理を行ったほうが効率がよい．そこで，処理を行うのに適した形式に変換をおこなってメモリに読み込む．

まず，トレースログ種別ごとにクラスを定め，そのクラスのインスタンスとして個々のイベントをオブジェクト化する．イベントが持つ数値データは，クラスのメンバ変数に数値として保持する．そして，イベントのインスタンスは，ログファイルに記録された順序でメモリに保持する．

逆にイベントのクラスからは，そのイベントの持つ数値データを取り出す関数と，元のログを復元する関数を用意する．そうすることで，検索時の数値の比較や検索後の計算や処理が可能となる．

(2) **検索条件を正規表現で記述**

検索条件を正規表現で記述する．テキスト処理における正規表現は，リテラル文字とメタ文字からなる．ログ検索では一つのイベントがテキスト処理の一文字のリテラルと対応する．イベントはイベント種別，属性と数値データからなり，これらのパラメータを細かく指定するために，リテラル文字をあらわすクラスを作成することとした．イベント種別と属性は直接種類を指定し，数値データはデータごとにマッチさせるかさせないかを判定する関数を用意し，その関数のポインタをリテラルクラスに登録することであらわす．メタ文字は文字あるいは文字列で記述する．

(3) **ログ検索の実行**

検索条件が正規表現だけで記述されている場合，このログ検索システムの利用者がするのは検索の関数を実行させるだけである．

ログ検索システム内部では，正規表現にしたがって非決定性有限オートマトンを作成してイベントを順に入力し，受理されるかどうかを判定する．

(4) **後処理の実行**

ログ検索がマッチした場合とマッチしなかった場合でそれぞれの処理をおこなう．

マッチした場合，テキスト処理の正規表現と同様に，正規表現記述の丸括弧で囲まれたログを後方参照でグループとして取り出すことができる．例えば，時間間隔サーチなどではマッチした場合にグループの先頭と末尾のイベントの時間を求めるなどの処理を行う．

一方，正誤サーチではマッチするかしないかでそのとき組込みシステムが正しく動作していたかを判定する．

4. 評　　価

4.1 μITRON デバッギングインタフェース仕様

本研究では検索対象のログとして ITRON デバッギングインタフェース仕様 ver1.00[7]（以下，dbif）で定めているトレースログの標準実行履歴ファイル形式を対象とした．dbif とは，μITRON 仕様[5] に準拠する

カーネルとデバッガや ICE といったデバッグツールとの間のインタフェースを標準化するための仕様で，2001 年 5 月に定められた．

dbif におけるトレースログでは，**表 1** のトレースログ種別を定めている．

ログの属性で起動と終了があるものは，そのイベントの処理の開始と終了がログに記録される．ログは時刻やタスク ID などの数値データを含むが，数値データの数や意味はログ種別や属性ごとに異なる．以下にログの例を示す．

```
TSKSTAT : 1020 1 2 0 0;
SVC|LEAVE : 1030 -114 2 0 1;
TIMERHDR|LEAVE : 1040 17 1 0;
DISPATCH|ENTER : 1050 2 0;
```

4.2 検索項目の分類

リアルタイムシステムの挙動を調べる際に有用であると思われる一般的な検索項目について，内容別に**表 2** のように分類した．この他にも，そのシステム固有の性質を用いた検索など，分類困難な検索項目が考えられる．

4.3 実　　装

本検索システムは C++言語で構築し，正規表現を非決定性有限オートマトン（以下，NFA）に変換後，ログを入力し，受理されるかされないかで検索を行う．

ログとして記録されているイベントは時刻，種別ごとの数値データを含む．ログ検索において，これら数値データは詳細に条件判定を行えることが望ましい．また，検索終了後に後方参照で取り出したイベントの数値データを用いて計算を行ったり，再検索をおこなったりする場合が想定される．これらの要求からテキスト処理用の正規表現では表現力が不足していると考え，C++言語を用いて独自に実装を行った．

正規表現エンジンの実装には決定性有限オートマトン（以下，DFA）を用いる方法と，NFA を用いる方法が知られている．DFA は NFA より検索が高速であるが，オートマトンの状態数が多くなる為にメモリを多く消費する．また，DFA は状態とログが対応しないために後方参照の実装が困難である．一方，NFA は検索は一般に遅くなるが，後方参照の実装が比較的容易という特徴がある．本システムでは，検索結果から後方参照できることが必須である．また，本システムではリテラルの指定に条件式が指定できるなど個々のリテラルの自由度が高い．このため，DFA で実装を行うと状態数の爆発が起こることが予想される．これらのことから今回は NFA で正規表現エンジンを実装することにした．

サポートしている正規表現の機能を**表 3** に示した．基本的にはテキスト処理で用いられる記号と同じ意味であるが，後方参照にて用いる\1〜\9 は若干意味が異なる．テキスト処理の場合では以前に出てきた文字とまったく同じ文字が再度現れることはあるが，ログはログの記録された時刻も情報として持っているために，まったく同じログが現れることは一般にはない．また，時刻以外の数値データの値が異なる場合を同じログと見るか異なるログと見るかは場合によって異なる．そこで，これらの判定条件を指定できるようにした．

検索の対象とするログは，豊橋技術科学大学組込みリアルタイムシステム研究室で開発している μITRON 仕様 OS の TOPPERS/JSP の Windows シミュレータ環境 [8] に，dbif に準拠したトレースログを出力するように独自に改良を行い出力させた．

ログは一度メモリに読み込まれてから検索を行う．ログに含まれているイベントは dbif のトレースログでは表 1 の種類に分かれている．メモリに読み込んで処理を行うのに，イベントの基底クラスとして CLogType を作り，各イベントを表すクラスを CLogType の派生クラスとして実装する．CLogType はログの記録された時刻や属性など，全てのログが持っている共通の情報と，共通の関数を持つ．各イベントを表す派生クラスは，そのイベント固有の情報と情報を取り出す関数，そして，イベントごとに振る舞いの異なる関数を持つ．

ログファイル全体は，配列に各イベントのクラスのインスタンスのポインタを順に格納することで表す．イベントの集合や後方参照などイベントを指定する場合は，配列の位置で間接的に指定する．

各イベントの指定条件の方法として本システムでは，条件判定を行う関数を用意し正規表現のリテラルとして関数を登録することで詳細な条件を指定可能にした．また，正規表現の記述は左シフト演算子<<で記述する．以下に，プログラム記述の一部を示す．

```
/* 条件判定関数
   時刻が 2000 以上 5000 未満なら真 */
bool  func1( int  time ) {
    return (2000<=time && time<5000);
}

/* 正規表現のリテラルとして関数を登録 */
CLiteral
```

表 1 トレースログの種別

トレースログ種別	属性	意味	取得可能な情報
LOG_TYP_INTERRUPT	起動，終了	割込	時刻，割込ハンドラ番号
LOG_TYP_ISR	起動，終了	割込サービスルーチン	時刻，割込サービスルーチン ID，割込ハンドラ番号
LOG_TYP_TIMERHDR	起動，終了	タイムイベントハンドラ	時刻，種別，タイムイベント ID，拡張情報
LOG_TYP_CPUEXC	起動，終了	CPU 例外	時刻，タスク ID
LOG_TYP_TSKEXC	起動，終了	タスク例外	時刻，タスク ID
LOG_TYP_TSKSTAT		タスク状態	時刻，タスク ID，遷移先，待ち状態，待ち対象
LOG_TYP_DISPATCH	起動	タスクディスパッチ	時刻，実行状態にあったタスク ID，種別
LOG_TYP_DISPATCH	終了	タスクディスパッチ	時刻，実行状態になるタスク ID
LOG_TYP_SVC	起動，終了	サービスコール	時刻，機能コード，返値，パラメータ
LOG_TYP_COMMENT		コメント	時刻，コメント

表 2 ログ検索の分類

グループ	検索内容
時間間隔サーチ	2 つのイベント間の時間間隔などを測定
時刻サーチ	特定のイベントの発生時刻を検索
イベントサーチ	イベントそのものを検索
メッセージサーチ	メッセージの送受信に関する検索
比率・回数サーチ	CPU 使用率，起動回数等を対象とする検索
正誤サーチ	起動順序，起動周期間隔等が正しいか判断

表 3 サポートする正規表現

記号	意味
.	任意のログ 1 個とマッチ
|	「|」で隔てられたいずれかとマッチ
*, +, ?	欲張りな繰り返し制御
*?, +?, ??	非欲張りな繰り返し制御
[^ログ]	指定されたログ以外のログとマッチ
(...)	グループ化と後方参照のための格納
\1, \2, ..., \9	対応するグループでマッチしたログと同種のログマッチ

```
    svc( SVC|ENTER, func1 );

/* ログファイルを指定 */
CRegex  regex( "log.txt" );
/* 正規表現を記述（svc*） */
regex  <<  svc  << '*';

/* NFA 作成と検索実行 */
regex.matching();
```

ログファイルを指定し，正規表現を記述すると，内部で NFA を作成する．NFA は NFA 全体をコントロールする CRegex クラスと，NFA の状態一つと対応する CNFANode クラスからなる．正規表現のバッカスナウア記法（以下，BNF）をもとにトップダウン構文解析を行い，それに基づいて CNFANode の枝を繋ぎ，正規表現からオートマトンを作成する．ログ検索における BNF は次のようになる．

```
<正規表現> ::= <項>
<正規表現> ::= <項> | <正規表現>
<項> ::= φ
<項> ::= <因子><項>
<因子> ::= <一次子>
<因子> ::= <一次子>*
<因子> ::= <一次子>+
<因子> ::= <一次子>?
<因子> ::= <一次子>*?
<因子> ::= <一次子>+?
<因子> ::= <一次子>??
<一次子> ::= <イベント>
<一次子> ::= [^<イベント>]
<一次子> ::= ( <正規表現> )
```

先に示したプログラムで作成される NFA を**図 1** に示す．これは種別が LOG_TYP_SVC で時刻が 2000 以上 5000 未満であるログの任意個数の連続とマッチする．

4.4 検索条件の記述

ログ検索で有用な検索項目が正規表現でどのように表記されるかの例を示す．

(1) **時刻サーチ**

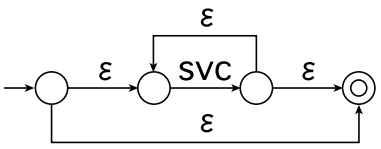

図 1 svc*の非決定性有限オートマトン

時刻サーチの例として，サービスコールがエラーを返した時刻を求める．
μITRON の仕様では原則としてサービスコールでエラーが発生した場合は負の値を返すことになっているから，負の値を返したサービスコールのイベントを探し，そのときの時刻を求めればよい．この正規表現は次のようになる．

```
/* 条件判定関数　返値が負ならマッチ */
bool  func2( int  ercd ) {
    return ( ercd < 0 );
}

/* 正規表現のリテラルとして関数を登録 */
CLiteral
    svc(SVC|LEAVE,NULL,NULL,NULL,func2);

/* ログファイルを指定 */
CRegex  regex( "log.txt" );

/* 正規表現を記述 */
regex << '(' << svc << ')';

/* NFA 作成と検索実行 */
if( regex.matching() ) {
    /* エラー発生時刻取り出し */
    cout << regex.startTime(1);
}
```

func2() で返値の正負を判定する．サービスコール終了イベント（LOG_TYP_SVC | LOG_LEAVE）のログは，時刻，機能コード，パラメータ数，返値の順で数値データが並んでいるので，CLiteral クラスのコンストラクタの 5 番目の引数で func2() のポインタを与えている．
該当するサービスコールが存在した場合はマッチした正規表現の 1 番目の丸括弧のグループの始まりの時刻を regex.startTime() で取り出し，出力している．この時刻がエラーの発生した時刻である．

(2) **時間間隔サーチ**

時間間隔サーチの例としてディスパッチからディスパッチまでの時間を求める．これはディスパッチの終了した時刻から次のディスパッチの開始の時刻の差を求めればよい．

```
CLiteral  dis_enter( DISPATCH | ENTER );
CLiteral  dis_leave( DISPATCH | LEAVE );

/* ログファイルを指定 */
CRegex  regex( "log.txt" );

/* 正規表現を記述 */
regex << '(' << dis_leave
      << '^' << dis_enter
      << '*' << dis_enter << ')';

/* NFA 作成と検索実行 */
if( regex.matching() ) {
    /* ディスパッチの時間の取り出し */
    cout << regex.time(1);
}
```

正規表現の部分は，
(a)　一つのディスパッチ終了のイベント
(b)　ディスパッチ開始以外のイベントの 0 回以上の繰り返し
(c)　一つのディスパッチ開始のイベント
をあらわしている．
このような記述が可能なのは，ディスパッチ開始とディスパッチ終了のイベントが必ず交互に発生するからである．このようなログの基本的な性質を使うことで正規表現の記述が容易になる．
もしも，ディスパッチ開始と終了が入れ子になるような関係であるならば，ディスパッチ開始以外のイベントの 0 回以上の繰り返し（^dis_enter*）にディスパッチ終了のイベントが含まれるケースが起こりうる．例えば，セマフォの獲得と返却の対応などは入れ子構造になる．しかし，正規表現ではこのような入れ子構造を記述できない．本システムで入れ子構造を処理するには，入れ子の入口と出口を別々の正規表現で記述して検索をおこない，マッチした入口と出口を個々に対応を調べるなどの方法が考えられる．これは，従来用いられていた検索の方法とほぼ同じである．

4.5 デッドラインミス検出の例

実際にログ検索システムでデッドラインミス検出のログ検索を実行し，必要とされるプログラムの記述量と検索の実行時間の比較を行った．

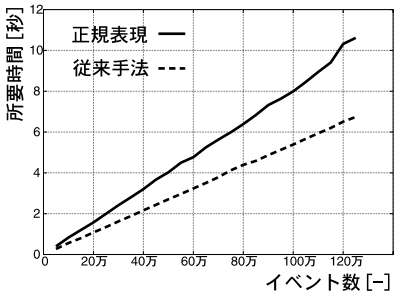

図 3 ログ検索時間の比較

従来行われていたように，デッドラインミス検出専用の検索プログラムを作成したところ，プログラム全体の行数は 1042 行となった．そのうちデッドラインミス検出に直接関係するプログラムの行数は 39 行であった．

次に同じことをするプログラムをログ検索システムを利用して記述を行った．**図 2** にログ検索システムを用いたプログラムを示す．

ログ検索システムを用いた場合のプログラム全体の行数は 19 行となった．そのうちデッドライン検出に直接関係するプログラムの行数は，正規表現の記述と検索の実行の 2 行だけであった．

なお，タスク ID が 1 のタスクのデッドラインミス検出の正規表現記述は次のようになる．

```
(WUP([^DL]*(DL[^SLP]*?DE))*[^DL]*
(DL[^SLP]*SLP[^DE]*DE))
```

一番外側の丸括弧に対応した後方参照のグループの開始と終了時刻の差が，別に定めたデッドラインよりも長ければデッドラインミスが発生していることとなる．ただし，各記号は次の意味である．

- WUP : タスク（ID=1）の起床とマッチ
- DL : タスク（ID=1）へのディスパッチとマッチ
- DE : タスク（ID=1）からのディスパッチとマッチ
- SLP : タスクの起床待ちとマッチ

次にデッドラインミス検出にかかった時間を測定し，結果を**図 3** に示した．検索にかかった時間は，正規表現の方が遅く，従来の手法の約 160%となった．

このように，全体の行数が数%になり大幅な記述量の削減が実現できた．なかでも検索のアルゴリズムに関する部分が 2 行になることは重要で，もし，検索条件が正規表現で記述可能ならばどんな複雑な検索アルゴリズムでも 2 行で記述できることを示している．

次に，検索後の処理であるが，図 2 のプログラムではグループの最初と終わりのログの時間差を取り出す regex.time() 関数を呼んでいる．本システムの正規表現では，検索後に丸括弧でかこまれている範囲のログをグループとして取り出すことが可能である．取り出したグループは，グループの最初のログ，終わりのログ，最初と終わりのログ，グループ内全てのログなど，利用する方法が限られている．これをパターン化することで検索後の処理も汎用化され，検索プログラムの再利用性が高まる．

以上の結果から，本システムは表記の面で様々な利点を持つため，検索に要する時間のオーバーヘッドは許容できると考えている．

5. まとめと今後の課題

リアルタイムシステムのオフラインデバッグにおいて，検索条件の記述に正規表現を使うことを提案した．そして実際にオフラインデバッグを実行するログ検索システムを構築し，これまで行われていた手法と比較を行った．検索の例としてデッドラインミス検出の場合で比較を行った結果，検出に約 1.6 倍時間がかかったが，全体の行数が数%となり，記述量が大幅に減少することがわかった．以上のことから，ログ検索に正規表現を用いたログ検索システムを利用することの有用性が確認できた．

今後の課題として，他の検索例による評価や，ログ検索用に正規表現を拡張することを考えている．

謝辞 本研究を行うにあたり御指導，御討論頂いた豊橋技術科学大学組込みリアルタイムシステム研究室の諸氏に感謝致します．

参 考 文 献

1) A.V. エイホ，R. セシィ共著，原田 賢一訳: ”コンパイラ I 原理 · 技法 · ツール”，サイエンス社 (1990).
2) 石畑 清: ”アルゴリズムとデータ構造”，岩波書店 (1990).
3) 小濱 一師: ”リアルタイム OS のログ検索システムに関する研究”，修士論文，豊橋技術科学大学情報工学系 (2001).
4) Robert Sedgewich 著 野上 浩平，星 守，佐藤 創，田口 東共訳: ”アルゴリズム C 第二巻 探索 · 文字列 · 計算幾何”，近代化学社 (1996).
5) 坂村 健監修，高田 広章編: ”μITRON4.0 仕様 Ver．4.00.00”，トロン協会 (1999).
6) 宿口 雅弘: ”組込みシステムのデバッグ技法”, 情報処理, Vol. 39, No. 10, pp. 886–891 (1997).
7) トロン協会 ITRON 部会 ITRON デバッギングインタフェース仕様ワーキンググループ: ”ITRON デバッギングインタフェース仕様 ver 1.00.00”，トロン協会 (2001).

```
bool funcw( int funcno ) {
  return ( funcno == TFN_IWUP_TSK );
}
bool funcs( int funcno ) {
  return ( funcno == TFN_SLP_TSK );
}
void main( void ) {
  CLiteral wup(SVC, 0, funcw, 0, taskID);
  CLiteral slp( SVC|LEAVE, 0, funcs);
  CLiteral de(DISPATCH, 0, taskID);
  CLiteral dl( DISPATCH|LEAVE, 0, taskID);
  CRegex regex( "log.txt" );
  regex << '(' << wup << "(^" << dl << "*(" << dl << '^' << slp << "*?" << de << "))*^"
        << dl << "*(" << dl << '^' << slp << '*' << slp << '^' << de << '*' << de << "))";
  while( regex.matching() ) {
    if( DEAD_LINE < regex.time(1) )
      cout<<"デッドラインミスが発生："<<regex.startTime(1)<<'\n';
  }
}
```

図 2 デッドラインミス検出プログラム

8) TOPPERS プロジェクト: "TOPPERS/JSP
カーネル ホームページ",
http://www.ertl.jp/TOPPERS/.