DEIM Forum 2011 C9-2

障害解析支援のためのオンラインログストリームエンジンの開発

五嶋　壮晃† 　中田　晋平† 　菅谷みどり† 　倉光　君郎†

† 横浜国立大学大学院工学府　〒240–8501 横浜市保土ヶ谷区常盤台 79-1
E-mail: †{goshima,shinpei}@ubicg.ynu.ac.jp

あらまし　近年, 自動車やロボットシステムに代表される高性能な組み込みシステムが登場してきている．これらのシステムに起きる複雑な障害を解決するため, システムの振る舞いを記録したログを解析する事が一般的であるが, 複数の障害を解析するためには障害ごとに障害解析器 (アナライザ) が必要となる．しかし, それぞれのアナライザが固有の実装を持つことは, 開発の効率や拡張性, 保守性の上から望ましくない．我々は, 効率良くアナライザを開発できるよう支援することを目的とし, ログストリームデータ解析エンジン, Stream Evidence Engine(SEE) を提案する．また, アナライザの動作やシステムの状態を視覚的に把握する事が可能な, SEE の補助ツールである Stream Evidence Engine Viewer(SEEV) を提案する．本論文では, 特定のログストリームデータに対して SEE と SEEV を適用し, 実験を行い, 評価により有効性を示した.

キーワード　ログ解析, アナライザ, ログストリームデータ解析

Development of online log stream engine to support analyzing of errors

Masaaki GOSHIMA†, Shinpei NAKATA†, Midori SUGAYA†, and Kimio KURAMITSU†

† School of Engineering, Yokohama National University Tokiwadai 79-1, Hodogaya-ku, Yokohama-shi, Kanagawa, 240-8501 Japan
E-mail: †{goshima,shinpei}@ubicg.ynu.ac.jp

Abstract　Recently, sophisticated embedded systems have been increasing. In general, way of analyzing logs that recorded behavior of systems is used to solve complicated errors in systems. We need analyzers as many as error of systems. However, it is not preferable because there is a possibility of preventing expandability and maintainability by having each analyzer a peculiar implementation. We aim to support efficient developing of analyzers, and propose Stream Evidence Engine (SEE) that analyzes streaming log data. We also propose Stream Evidence Engine Viewer (SEEV). SEEV is accessory tool for SEE. It makes to judge visually operation of analyzers and state of systems. In this paper, we do an experiment that uses SEE and SEEV for specific streaming log data, and show effectiveness by the evaluation.

Key words　Log analysis, Analyzer, Log Stream Data analysis

1. はじめに

近年, 自動車やロボットシステムに代表される高性能な組み込みシステムが登場してきている．これらのシステムでは起きる障害が複雑化しており, 事前の障害検知や, 事後の原因解析が一層重要な課題となっている．これらを解決するための手段として, システムの振る舞いを記録したログを解析する手法がある. [2] [3] この解析手法は大きく, 解析対象のデータをシステムが終了した後に解析するオフライン解析と, システムの動作と並行して解析するオンライン解析に分けられる．一般に, オフライン解析は大量のログを保存したストレージを用いて障害解析を行い, オンライン解析は, システムの動作中, 際限なく出力されるログをストレージに保存し続けることはできないため, ストレージを用いずにログを解析し, 障害解析を行う．ログ解析手法では, syslog や kernel log [5] などのシステムログが用いられており, 保存形式がそれぞれ異なる．オンライン解析を行う際は, 大量に出力されるシステムログによって解析プロセスのバッファをあふれさせないために, 任意のサイズ (window size) で解析するシステムログの量を制限し, 区切った window size 内でバッファの保存領域を再利用することで対処するスライド

窓モデルの実装が必要となる. また, システム管理者に対して, 障害が発生したことを迅速に通知するために, 解析結果を表示するための手法が必要となる. このため, 解析したい障害の種類によって異なるシステムログからの入力処理を実装しなければならず, オンライン解析時にはスライド窓モデルの実装も必要となるため, 障害検知や原因解析の効率の低下, 保守性の低下が考えられる. また, 解析結果が多様な表示形式で出力される事は, システム管理者がシステム全体に起きている障害を把握することを困難にしてしまう.

我々は, システムログの入力処理, 解析結果の出力, オンライン解析時に必要となるスライド窓モデルの実装を支援することを目的とし, ログストリーム解析エンジン, Stream Evidence Engine (SEE) を提案する. SEE は障害検知や原因解析を行うための障害解析器 (アナライザ) 開発者に対し, インターフェースを提供する. これにより, 開発者はシステムログの保存形式の差異を気にする事なく, 解析アルゴリズムの実装に注力する事ができる. また, SEE が行うオンラインな障害解析から得られる情報を基にして, 管理者がシステムの状態を把握することを支援するツールである Stream Evidence Engine Viewer(SEEV) を提案する. SEEV は SEE に登録されているアナライザとシステムの動作とを関連づけたグラフを自動生成するシステムの可視化ツールであり, 障害解析の結果, 問題が起きていればそれを視覚的に表示する.

本論文の構成を以下に示す.

２章で SEE と SEEV の設計と実装についてそれぞれ述べ, 3 章ではオンラインでの障害解析が重要となるロボットシステムに対して SEE と SEEV を運用した例を示す. 4 章で関連研究を示し, 5 章で全体のまとめと今後の課題について述べる.

2. 設計と実装

本章では, 我々の提案する SEE と SEEV の設計と実装について示す.

2.1 障害解析手順のモデル化

我々は, ロボットシステムの障害解析実験を通し (実験の詳細は 3 章で示す), 障害解析の手順が次の 3 過程に大別できることに着目した. SEE と SEEV によってそれぞれの過程を支援する.

（1） システムログの入力処理

（2） 解析処理

（3） 解析結果の通知

2.2 Stream Evidence Engine

2.2.1 設　　計

SEE では, 先に挙げた障害解析手順のうち, システムログの入力処理・解析処理を支援する. 以下で, それぞれについて具体的に述べる.

（1） システムログ入力の支援 : アナライザを開発するためには, 解析したいシステムログの保存形式の違いに応じて, 異なる入力処理を実装しなければならない. syslog のように以前の情報を消去せずに続きからログを書き込み, 保存する形式のものもあれば, /proc/meminfo の情報等のようにログファイルの状態を全てリセットし, 改めて書き込み, 保存するものもある. SEE ではこれらの書き込み形式の違いを吸収し, システムログの入力処理は, アナライザ開発者が解析対象のログファイル名を SEE に知らせるだけでよいとすることで, 対象とするシステムログがどのような形式で書き込まれていても, それを気にせずに開発することができる. また, オンライン解析においては, ログデータを受け付け始める処理や止める処理 を SEE が管理し, スライド窓モデルで解析することで, 開発者はこれらの処理を実装する手間を省くことができる.

（2） 解析支援 : システムログを解析する際, 大量のログデータの中から解析に必要な情報を抜き出すことや, 抜き出された情報に対してオンラインアルゴリズムを適用し, 障害が起きているかの判断をすることなどが必要になる. SEE ではこれらの処理を統一して記述できるように, 開発した情報抽出用の関数とアルゴリズム解析用の関数を SEE に登録する形で解析を行うようにしている. これにより, 解析対象が異なる障害に対しても, 同じインターフェースで実装することができるため, 開発物を保守する上で効率化が期待できる.

2.2.2 実　　装

SEE はプログラミング言語 Konoha [4] で実装されており, オブジェクト指向で記述されている. SEE は次に示す 4 つのクラスオブジェクトから構成され, アナライザ開発者に add, start, convert, send, stop の 5 つのメソッドを提供する.

- Receiver : ログストリームデータの入力管理
- Analyzer : ログストリームデータに対してオンラインアルゴリズムを適用し, 解析結果を得る
- Converter : 解析結果を任意のプロトコルでエンコードする
- Sender : エンコードされた解析結果を外部の DB や SEEV に通知する

以降でそれぞれのオブジェクト, メソッドについて述べる. SEE の全体図を図 1 に示す.

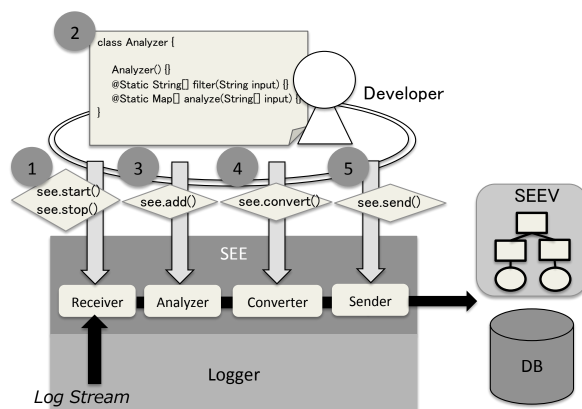


図 1 Stream Evidence Engine Overview

a） Receiver

ログストリームデータの入力管理を担当する. SEE で提供する start, stop メソッドを実行する事で, ログストリームデータ

の入力制御を行うことができる (図 1 の 1). 受け取るログの種類は大きく分けて追記型と更新型がある. 追記型は以前の情報を消去せずに続きからログを書き込んでいく方式であり, 更新型はログファイルの状態を全てリセットし, 改めて書き込む方式である. Receiver ではこれらのログファイルの状態を把握することで, ログファイルに何らかの変更があった時刻からログの内容を受け取り始める.

b) Analyzer

Receiver から得たログストリームデータに対してアナライザ開発者が作成したオンラインアルゴリズムを適用し, 解析結果を得る. オンラインアルゴリズムを Analyzer で利用するまでの手順は次のようになる.

(1) ログストリームデータから解析に必要な情報だけを抜き出すための filter メソッド, 抜き出された情報にオンラインアルゴリズムを適用して解析結果を得る analyze メソッドを作成し, これらを含む Analyzer Class を作成する (図 1 の 2)

(2) SEE のインスタンスを作成後, SEE で提供する add メソッドを用いて, 作成した filter, analyze メソッドを SEE に追加する. このとき, 一意に決まる Analyzer 名を指定する (図 1 の 3)

c) Sender

解析結果のデータ量はログファイルからの入力データ量に比べ, かなり削減される. このため, 外部リソースにデータを送信するコストは高くなく (実測値は評価の章で示す) , 解析結果を DB に保存することや, SEEV で可視化を行う際には, 解析に要するリソースを確保するためにも外部のリソースを用いる方が良い. Sender は外部にある DB や SEEV に対し, SEE を通して得られた解析結果を通知する役目を果たしている. また, デバッグ時など外部に通知する必要がないときのために, SEE の提供する send メソッド を用いることで, 解析結果を標準出力やテキストに出力することも可能である.

d) Converter

SEE が提供する convert メソッドを用いて, 予め Sender と外部間のプロトコルを SEE に対し指定しておくことで (図 1 の 4), Analyzer から得た解析結果を指定されたプロトコルでエンコードし, Sender によって通知された解析結果を外部で処理する際に必要となるデータのデコード処理を効率よく行うことができる.

以下に簡単な Analyzer Class と SEE を起動するためのソースコードを示す.

- Sample ソースコード [sample_analyzer.k]

```
using see.*;
LOGFILE = "/path/to/sample.log";
class Analyzer {
  Analyzer() {}
  @Static String[] filter(String input) {
      return getNeedsInformation(input);
  }
  @Static Map[] analyze(String[] input) {
      return judgeStateOfSystem(input);
  }
}
void main(String[] args)
{
  StreamEvidenceEngine see =
          new StreamEvidenceEngine();
  Analyzer a = new Analyzer();
  Func<String=>String[]> sample_filter =
          delegate(a, filter);
  Func<String[]=>Map[]> sample_analyzer =
          delegate(a, analyze);
  see.add("sample", sample_filter);
  see.add("sample", sample_analyzer);
  see.start("sample", LOGFILE);
}
```

初めに, SEE を使用するための処理とログファイルがある場所までのパスを記述する. filter メソッドは, 大量のログストリームデータから任意のデータを抽出するために行う. 上述のコードでは, 文字列型である String 型で受け取ったデータから, 必要なデータを抽出している. analyze メソッドでは, filter メソッドによって抽出されたデータに対してオンラインアルゴリズムを適用し, システムに障害が発生しているかを判断する. Konoha では main 関数が処理の起点となるため, main 関数内で SEE のインスタンスを作成した後, Analyzer Class の filter, analyze メソッドを SEE に加え, 起動する事で解析処理を行うことができる. また左記のコードでは SEE の提供する convert メソッドや send メソッドについて書かれていないが, これは特に指定しない場合は JSON 形式で SEEV との通信を行うためである.

2.3 Stream Evidence Engine Viewer

オンライン解析時において, 障害を一早く発見・理解し, 対処したいという要求がある. しかし, 多様な形式で出力される解析結果を統一的に管理する仕組みがなければ, システム管理者が障害を把握するまでに要する時間が増えてしまうことが考えられる. SEEV は, 予め SEE に登録されているアナライザの情報とシステムの構造を記録した D-Case [7] からシステムの状態管理グラフを自動生成し, SEE が行うオンラインな障害解析から得られる情報を基に, 障害解析の結果, 問題が起きていればそれを視覚的に表示することができる. これにより, 管理者は SEEV を監視し続けることで, システムに発生した障害を把握することができる. SEEV は先に挙げた障害解析手順のうち, 解析結果の通知を支援する.

2.3.1 設 計

SEEV は次の 4 つのコンポーネントから成る.

- Connector : SEE との通信部
- Generator : 構造生成部
- Visualizer : 表示部

● StateMachine：システムの状態管理部

以降でそれぞれのコンポーネントの設計について述べる．SEEV の全体図を図 2 に示す．

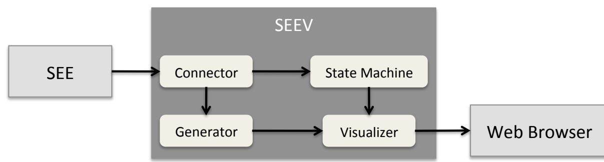


図 2 Stream Evidence Engine Viewer Overview

a） Connector

SEE から通知された情報を Generator, State Machine に渡す．

b） Generator

Connector から通知された情報から，各アナライザの名前を抜き出し，システムの構造情報を記録している D-Case の情報を合わせる事で，トップダウン形式でアナライザ同士の関係を構造化する．

c） Visualizer

Generator によって自動生成された構造化データを Web ブラウザ上に描画するための処理を行う．

d） StateMachine

システム運用中に Connector を通して通知される各アナライザの解析結果を管理しており，もしシステムの状態に変化が起こった場合は，それを Visualizer に通知する事で，Web ブラウザ上で視覚的にシステムの状態の変化を知ることができる．

2.3.2 実　装

SEEV の Connector を除くコンポーネントは，すべて JavaScript で実装されている． これは，Web Browser を表示媒体とすることでマルチプラットフォームに対応するためである．Connector は Konoha 言語で実装されており，SEE からのデータを受け取る Server となっている．通信プロトコルは全て JSON 形式をとっており，Generator, State Machine との通信は Web Socket を介して行っている．以下に SEE, Connector, Generator, State Machine 間の通信方法をまとめた図を示す．

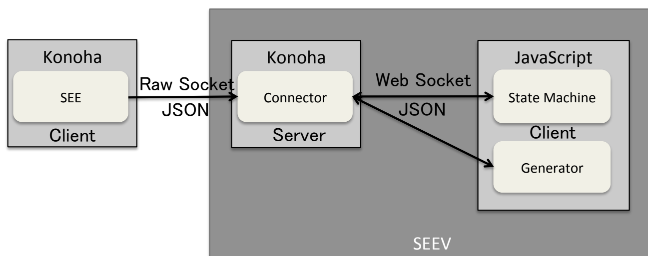


図 3 Communications Protocol

3. 実験・評価

本章では，リアルタイム OS である ART-Linux [6] 上で動作するロボットシステムに対し，LTTng を用いて取得したログ情報をもとに，SEE, SEEV を運用した例を示す．3.1 で実験の概要を示し，3.2 では SEE のパフォーマンスについて示す．3.3 で実験のまとめを行う．本実験で利用したシステムアーキテクチャ図を図 4 に示す．

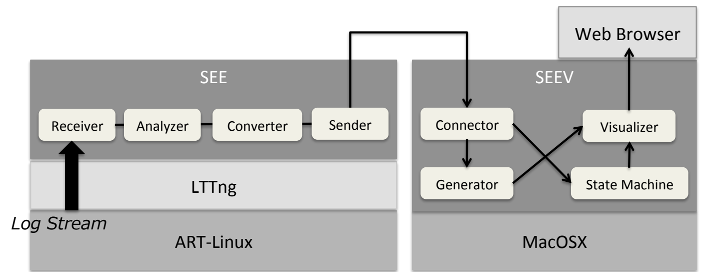


図 4 System Architecture Overview

3.1 実験の概要

本実験は，ロボットシステムのサーボプログラムに，本来呼ばれるはずの ART-Linux のシステムコールではなく Linux のシステムコールを用いてしまったというバグを，LTTng の Kernel Trace ログを解析することにより発見するというものである．

実験の手順は次の通りに行った．

（1） サーボプログラムにバグを埋め込む

（2） 誤ったシステムコールの使用を検知するアナライザを作成する

（3） (2) で作成したアナライザを SEE にセットし，SEE を起動する

（4） LTTng を走らせ，SEE にログを送る

（5） SEE を通して解析された結果を SEEV に送る

（6） SEEV によって解析結果を可視化する

それぞれについて詳しく説明する．

1: ART-Linux のシステムコールである art_wait() を呼ばずに，Linux のシステムコールである usleep() を呼ぶことで，Linux のタイマ割り込みのタイミングでタスクが実行されてしまうため，リアルタイム周期実行が阻害されてしまうというものである．概要を図 5 に示す．

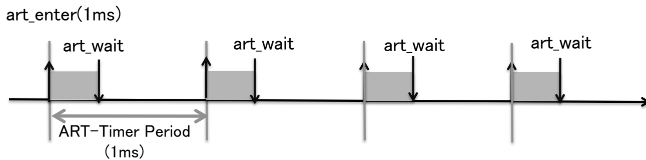
2: art_enter システムコールと art_wait システムコールが交互に呼ばれなければならないことに着目し，art_wait ではないシステムコールが呼ばれたことを判断するアルゴリズムを用いてアナライザを作成した．

4: LTTng のプローブポイントを ART-Linux の Kernel 内の任意の場所に追加し，システムコールの振る舞いを記録したログ情報を取得している．SEE にログ情報を通知する際は，LTTng で取得したログのバイナリデータをテキストダンプし，得られたテキストを追記型で SEE の読み込み対象のファイルに書き込む事で実現している．

5: WebSocket を用いて SEE と SEEV 間の通信を行っている．データ送受信の際のプロトコルは，JSON 形式をとった．

6: SEEV で解析結果を表示している図を図 6 に示す．

正しいAPI(art_enter, art_wait)を使用した場合の動作例



誤ったAPI(usleep)を使用してリアルタイム制御用を実装した場合の動作例
-usleep が依存するLinux のタイマ割り込みのタイミング(4ms)でサーボタスクが起床
→ 1/4の動作性能

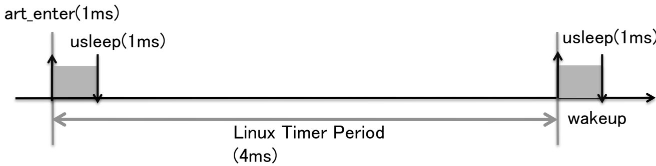


図 5 間違った API の使用による誤動作の例

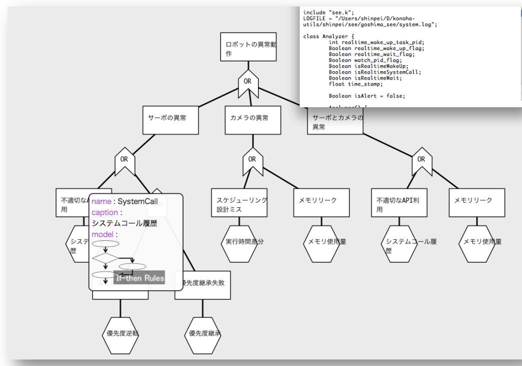


図 6 SEEV での可視化の例

3.2 SEE のパフォーマンス評価

実験で用いたシステムにおける, SEE の各コンポーネントの window size に対する処理時間と処理量, 処理速度の関係を表したたグラフを図 7, 図 8, 図 9 に示す.

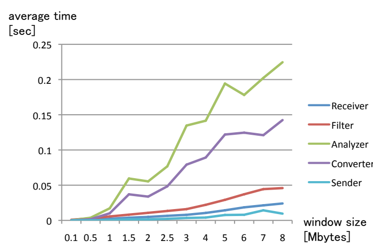


図 7 処理時間と window size の関係

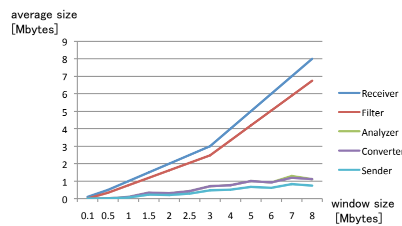


図 8 処理量と window size の関係

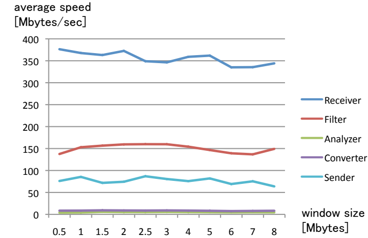


図 9 処理速度と window size の関係

図 7 では, SEE の各コンポーネントにおいて, windows size の増加によって 一周期に処理するデータ量が増えた場合, 処理時間がどのように変化するか表したものである. 図より, 測定誤差はあるものの, ほぼ比例的に処理時間が増加していることがわかる. これにより, window size の値は解析に影響を与えていない事がわかる. また, 処理時間の程度を各コンポーネントで比較すると, Analyzer, Converter において多くの時間がかかっていることがわかる. 同様に図 8, 図 9 を見ても, window size が解析に影響していない事がわかり, Analyzer, Converter における処理速度の遅さが目立った. また, 図 8 において, Receiver と Sender の処理量の間に大きな差があることから, 解析を行う過程で, 相当量のデータが削減できることがわかる.

3.3 実験のまとめ

本実験では, LTTng のログストリームデータを SEE を用いて解析し, SEEV によりロボットシステムに生じた障害を検知するところまでを示した. これにより, アナライザ開発者がオンラインアルゴリズムのみに注力するだけで障害解析を行うことができることが分かり, また図 9 から SEE を用いる事で障害解析にかかるパフォーマンスの低下は, Analyzer, Converter を高速化することで, 更に改善可能であることがわかった.

4. 関 連 研 究

ログを解析する手法を用いてオンライン障害解析を行い, できるだけ早い段階でのシステム復旧をサポートするための製品として, 日立の uCosminexus Stream Data Platform [2] がある. システム管理者が一早く障害を把握するために, 専用のシステムビューワが用いられており, 解析結果の通知を支援している点は共通しているが, 我々の研究との違いとして, 本研究ではアナライザ開発者を支援することに焦点を当て, システムログの入力処理やオンライン解析時のスライド窓モデルの実装を省略できるようにしている事が挙げられる.

5. お わ り に

我々は, 障害解析において, ログの入力・解析処理を支援するため, オンラインストリームログエンジン, Stream Evidence Engine を設計・実装した. また, ログの出力を支援するために, システムの構造を視覚的に管理する事が可能な, Stream Evidence Engine Viewer を設計・実装した. SEE, SEEV をロボットシステムの障害解析に用いることで, 障害解析の手段としてこれらが適している事を示し, また SEE のパフォーマンスを評価し, 課題を明らかにした. これらをもとに, 今後の課題と

して, より効率的にリソースを用いて高速な解析処理を行うために, 分散・並列処理の実装を考えている. また, 受け取るログの保存形式の差異についても吸収できるように, テキスト形式のものに加えてバイナリ形式で保存されているログ情報にも対応できるようにすることが考えられる.

文　献

[1] M. Sugaya, Y. Ohno, A. van der Zee, and T. Nakajima. A lightweight anomaly detection system for information appliances. Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on, 0:257?266, 2009.

[2] Y. Hayashida, N. Ioki, N. Arai, and I. Nishizawa. Real-Time Log Analysis Using Hitachi uCosminexus Stream Data Platform. 15. DASFAA 2010, pages 440-443, Tsukuba, Japan.

[3] James H. Andrews, Yingjun Zhang, General Test Result Checking with Log File Analysis, IEEE Transactions on Software Engineering, v.29 n.7, p.634-648, July 2003

[4] K. Kuramitsu. Konoha: Implementing a static scripting language with dynamic behaviors. In Proceedings of Workshop on Self-sustaining systems, page (to appear). ACM Press, 2010.

[5] Michel Dagenais. Mathieu Desnoyers. Lttng: Tracing across execution layers, from the hypervisor to user-space. In Proceedings of the Linux Symposium, Ottawa, Ontario Canada, 2008.

[6] Youichi Ishiwata, Satoshi Kagami, Koichi Nishiwaki and Toshihiro Matsui, ART-Linux 2.6 for Single CPU: Design and Implementation

[7] Yutaka Matsuno and Jin Nakazawa and Makoto Takeyama and Midori Sugaya and Yutaka Ishikawa, Toward a Language for Communication among Stakeholders, DCasePRDC2010, Proc. of the 16th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'10), 2010, pages 93-100