一般社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

アスペクト指向プログラミングによる高性能・低消費電力化

鷲崎　弘宜‡　大河原　洸太†　原　章浩†　深澤　良彰†

†早稲田大学基幹理工学部情報理工学科
早稲田大学グローバルソフトウェアエンジニアリング研究所
〒169-8555 東京都新宿区大久保 3-4-1
E-mail:　†washizaki@waseda.jp

あらまし　特定の品質要求を満足するためのプログラムコードはしばしば、プログラムの基本的なモジュール構成に対して横断的に散らばることが知られている。そのような横断的関心事のモジュール化を通じて保守性を維持あるいは高めることに有効な技術として、アスペクト指向プログラミング（Aspect-Oriented Programming: AOP）がある。本稿では、AOP の実行性能や品質向上に関する応用として、マルチコア環境における単体テスト実行時間の短縮を目的とした単体テストフレームワーク JUnit の AspectJ（Java 言語用の AOP 処理系）によるマルチスレッド化と、ロボット制御プログラムの低消費電力化を目的としたハードウェア制御方式やアルゴリズムの AspectC（C 言語用の AOP 処理系）による追加および変更の事例をそれぞれ報告する。

キーワード　アスペクト指向プログラミング，モジュール，マルチスレッド，低消費電力

High Performance and Low Energy Consumption by Aspect-Oriented Programming

Hironori Washizaki†　Kota Ohkawara†　Akihiro Hara†　and　Yoshiaki Fukazawa‡

†Waseda University, Dept Computer Science and Engineering, Global Software Engineering Laboratory
3-4-1, Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
E-mail:　†washizaki@waseda.jp

Abstract　In this paper, we report two case studies of applying aspect-oriented programming for achieving high performance and low energy consumption while keeping or improving maintainability: multi-threading existing unit test framework (JUnit) to reduce testing time on multi-core processing environments, and, adding and changing hardware control strategies and algorithms to reduce energy consumption while controlling robotics.

Keyword　Aspect-oriented programming, Modularity, Multithread, Low energy consumption

1. はじめに

プログラムの実装において、特定の品質要求を満足するためのコードはしばしば、プログラムの基本的なモジュール構成に対して横断的に散らばることが知られている。例えば、テスト容易性の向上や、セキュリティ対策として証跡を残すことを目的としたログ出力コードは、プログラム中でログ出力の必要な至る所に埋め込まれることとなる。そのような品質要求（およびその満足のための実現手段）は横断的関心事であり、モジュール化せずに横断的に実装したままとすると、当該品質要求を満足するがためにかえってプログラム全体の見通しが悪くなり、以降の開発効率や保守性の低下を招きかねない。

横断的関心事のモジュール化に有効な技術として、アスペクト指向プログラミング（Aspect-Oriented Programming: AOP）[1][2]がある。AOP とは，要求を本質と横断的関心事の集合として分離して捉え，本質を表す箇所の修正なしに、結合ルールによって一つのプログラムへと纏め上げるプログラミング手法である．AOP は様々な種類の品質の効率的かつ変更や拡張に優れた作りこみに有効なことが知られており、例えば Web アプリケーション開発における広義のディペンダビリティ（可用性およびその影響要因としての信頼性や保守性、セキュリティ等の能力や程度全般）の確保について様々な手法や実践が知られている[3]。

本稿では、そのような品質の作りこみにおける AOP の応用として、我々による異なる以下の二つの事例を報告する。

- 単体テスト実行時間の短縮（およびそれを通じた各種ソフトウェア開発における効率的な信頼性向上）を目的として、AOP による効率的かつ修正や拡張に優れた単体テストフレームワ

ークのマルチスレッド化

- ロボット制御プログラムの低消費電力化を目的として、AOP による効率的かつ修正や拡張に優れた制御アルゴリズムの追加および変更

2. 単体テストフレームワークのマルチスレッド化によるテスト時間短縮

ソフトウェアの大規模、複雑化により以前に比べ開発に時間を要するようになった。また、これにともなって、ソフトウェアのテスト工程における単体テストのテストケースの数が増えテストにも時間を要するようになっている。

テストを補助するための単体テストフレームワークが存在する。これは、テストケースの作成から、テストの実行とその結果の表示までの補助を一貫して行う。例えば JUnit[4]や TestNG[5]がそれに当たる。JUnit のテストケースは、テストクラスとテストの内容を記述するテストメソッドからなる。

単体テストの実行時間短縮の方法の一つとして、計算機環境のマルチコア CPU のリソースを活用した単体テストの並列実行が挙げられる。ここで、単体テスト実行の基盤となる単体テストフレームワークがマルチスレッド化されていれば、マルチコア環境のリソースが活かされることが期待できる。しかし、JUnit 4.x のようにマルチスレッド化が行われていない単体テストフレームワークが存在する。つまり、テストクラスに存在するテストメソッドは、上から記述してある順番に逐次実行される。これでは、現在のマルチコア CPU のリソースを活かし切れない。また、マルチスレッド化がなされていても、テストクラス内部の個々のテストメソッド間の依存関係を考慮して並列実行ができなければ、意図したテストが行われない可能性がある。

そこで我々は、AOP 処理系である AspectJ[2]を用いて単体テストフレームワーク JUnit を修正せずに、テストクラス内の依存関係を考慮した上で、既存の単体テストフレームワークをマルチスレッド化し、テストを並列実行するための拡張を行った。以降でその仕組みおよび結果を示す。

2.1. マルチスレッド化アスペクト

マルチスレッド化を行うために、J.L.Sobral の提案手法[6]を用いる。具体的には AspectJ を用いて[6]の手法における Concurrency アスペクトを用意して JUnit 内部に手を加えることなく自動拡張する。Concurrency アスペクトにおける具体的なマルチスレッド化の処理については、マルチスレッド化用ライブラリ Fork/Join Framework[7]を用いた。Fork/Join Framework はワーカースレッドにキューが存在し、個々のワーカースレッドのキューにタスクが無くなったら他のワーカースレッドのキューからタスクを奪う形をとる。マルチコア環境においては、Fork/Join Framework は柔軟にタスクの振り分けを行うことができ効率がよい。アスペクトの主要コードを以下に示す。

(1) 最初に、JUnit の ParentRunner クラスの runChildren メソッドについて以下の本体コード（抜粋）を持つ around アドバイスを適用して以降の並列テスト実行の準備を行う。これは結果として、各 BaseRunnerScheduler の schedule メソッドおよび finished メソッドを呼び出すこととなる。

```
final ParentRunner pr =
(ParentRunner)thisJoinPoint.getThis();
RecursiveAction rootra = new RecursiveAction() {
        protected void compute() {
                proceed(notifier);
        }
};
pr.fFJpool.invoke(rootra);
rootra.join();
pr.fFJpool.shutdown();
```

(2) BaseRunnerScheduler クラスの schedule メソッドに対しては以下の本体コード（抜粋）を持つ around アドバイスを適用して、並列化を達成するスケジューリングに向けた fork を実行する。

```
RunnerScheduler scheduler =
(BaseRunnerScheduler)thisJoinPoint.getTarget();
RecursiveAction ra = new RecursiveAction() {
        protected void compute() {
                proceed(childStatement);
        }
};
ra.fork();
scheduler.fRAarray.add(ra);
```

(3) BaseRunnerScheduler クラスの finished メソッドに対しては以下の本体コード（抜粋）を持つ around アドバイスを適用して、スケジューリングの動機を取るための join を実行する。

```
RunnerScheduler scheduler =
(BaseRunnerScheduler)thisJoinPoint.getTarget();
for(RecursiveAction ra : scheduler.fRAarray) {
        ra.join();
}
```

2.2. 依存関係の設定

テストを並列実行するに当たり、テストメソッド間の依存関係が重要になる。我々は依存関係をテストケース作成者が明示的に指定する形を採用し、指定方式としてはテストケース内のアノテーションおよびテストケース外の XML ファイルの両方をサポートすることとした。それらをテストケース作成者が記述し、依存関係読込み系に渡し、依存関係を考慮した上で並列実行を可能にする。その全体構成を図 1 に示す。

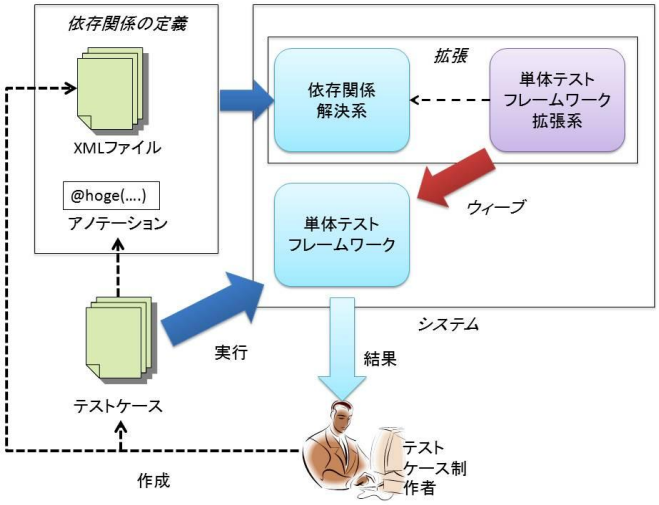

図 1: 単体テストフレームワークのマルチスレッド化の仕組みの全体像

2.3. テスト時間短縮効果の評価

AOP の適用を通じたマルチスレッド化によるテスト時間短縮の効果を評価した。具体的には、(a)マルチスレッド化せずにテストを逐次実行した場合、(b)Executor を用いて並列化を行っている Parallel Computer を用いて並列実行した場合（関連研究で説明する）、(c)我々の方法で Fork/Join Framework を用いて並列実行した場合でのテストが終了するまでの時間を測定した。

デュアルコア CPU である Intel Core 2 Duo T7500 2.2GHz、Memory 4GB のテスト実行環境上であるテストケースをそれぞれ実行した結果、5 回実行の平均時間は(a)36.0 秒、(b)26.2 秒、(c)24.6 秒となり、我々の方法がマルチスレッド化および Fork/Join Framework の採用によりテスト時間短縮に優れていることを確認した。

2.4. 保守性の評価

さらに保守性の評価として、横断的関心事の実装および変更にかかるコスト[8]を評価した。具体的には、ある機能の実装と変更にあたり、AOP を適用しない場合と適用した場合で、そのコード記述量、他の関心事のコードを含まずに連続記述したコードの集合であるコード行集合、変更に伴ってコードを修正したファイル数を測定するものである。結果的に、その機能の変更終了後に修正したファイル数が少なければ保守性が向上していることになる。

最初に Executor を用いて実装し、その後 Fork/Join Framework に変更する場合を考える。また、拡張を実装する際に追加したクラス数と実装の段階から修正を行ったファイル数も測定した。

結果を図 2 に示す。図 2 より、アスペクトを用いるとコード記述量は幾らか増加するが、変更ファイル数を削減できていることが分かる。アスペクトを使用した方が未使用の場合に比べて、ライブラリの更新が合った場合に、変更を行うファイルの数が少ないので保守性の向上につながると共に、追加するファイルの数も少ないので、クラスの関連があまり複雑にならないと考えられる。

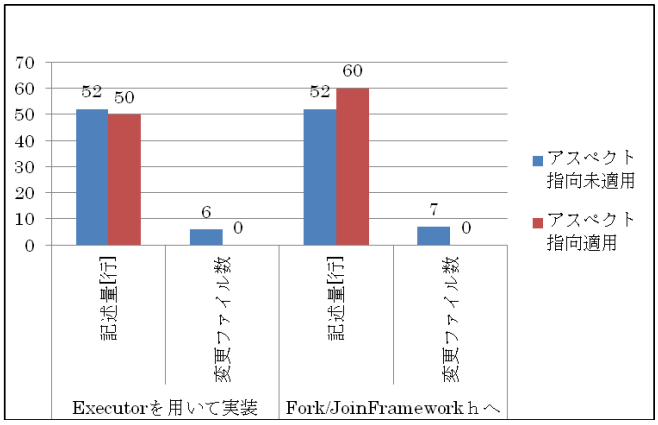

図 2: 記述量および変更ファイル数の比較

2.5. 関連研究

Parallel Computer は、JUnit 4.X の experimental パッケージに付属しているクラスである。JUnit のテストは JUnit 内部で Java の Runnable インターフェースの実装によって実行されているが、その Runnable インターフェースを実行、同期するクラスを Parallel Computer 独自のクラスに置き換えることによって並列実行を可能にしている。この独自のクラスは JDK 1.5 で追加された Executor を用いてマルチスレッド化を行っている。

Java の単体テストおよび結合テストを行うフレームワークである。テストの記述の形式もアノテーションを用いるもので JUnit 4.X とほぼ同一である。ただし、提供しているメソッドに互換性はなく種類も異なる。、異なる点として、テストの実行時に設定ファイルとして XML を用意する。TestNG は並列化に対応しており、XML ファイルの suite 要素に parallel 属性を加えること

によって並列に実行が可能になる。ただしテスト間の複雑な依存関係の記述と把握には適していない。

JUnit によるテスト実行の並列化の別アプローチとして、グリッドやクラウド技術を応用し、並列分散実行する手法が提案されている[9][10]。我々の提案は、並列実行の環境が異なること、および、並列実行の仕組みの実現にあたり AOP の適用により JUnit そのものの改変を避けている点が根本的に異なる。JUnit の改変を避けることでテスト環境全体の保守性を保っており、さらには今後の JUnit の改訂におけるアスペクトの再利用を通じた効率的なマルチスレッド化が期待できる。

3. 制御方式やアルゴリズムの追加・変更による低消費電力化

組込みソフトウェアシステムの開発にあたり、バッテリ駆動の場合や環境性の考慮などにより、省電力性が求められることがある。組込みソフトウェアの低消費電力化を実現するにあたり、コンパイル技術、アプリケーション設計技術、OS による電力管理技術などが挙げられる[11]。

我々は、アプリケーション設計を変更し、ハードウェアを効率的に制御することでソフトウェアの低消費電力化を実現することを考える。しかし、そのようなハードウェアの制御方式・アルゴリズムの変更は、しばしばプログラムの多数の箇所を変更する必要性を生じ、結果としてプログラムの見通しが悪くなり保守性や拡張性を低下させる可能性がある。

そこで我々は、AOP の適用によりハードウェアの制御方式・アルゴリズムをモジュール化した形で柔軟に追加・拡張・変更することを可能として、低消費電力化と高い保守性の維持の両方を同時に達成することを試みた。以降においてその仕組みと結果を示す。

3.1. 低消費電力化アスペクト

教育用レゴマインドストームで組み立てられたロボットを、床にひかれたラインに沿って自動走行させて荷物を搬送するライントレース制御アプリケーションプログラムを対象とする（参考[12]）。もともとのプログラムの記述言語は C 言語である。そこで AOP 処理系としては AspectC[13]を用いた。

最初に、省電力性を検討せずに、荷物搬送やライントレースといったユースケース・機能要求を満足するためのシンプルなプログラム実装を用意した。

続いて低消費電力化の戦略として、より効率的なライントレース方式へと変更すること、および、効率的な光センサ制御の二つを検討し、それぞれもとのプログラムの複数のファイルに対して作用する横断的なアスペクトとして実装した。もとのプログラムに対する作用の散らばりの様子を図 3 に示す。

前者については、もとの簡単な走行方式に置き換えて、比例制御（P 制御）によりラインに滑らかに沿って効率的に走行させる方式を、インタータイプ宣言と around アドバイスを用いて実現するアスペクトとして実装した。

後者については、アプリケーション全体の中で光センサを転倒させる必要のない状態（例えばライントレースせずに荷物の積み込みを待機している状態）を識別し、そのような状態において光センサを自動消灯させる方式を、複数の before アドバイスにより実現するアスペクトとして実装した。

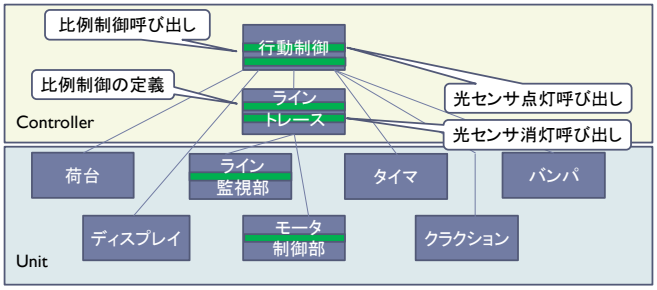

図 3: もとのプログラムに対する低消費電力化戦略実現コードの散らばり

3.2. 低消費電力化効果の評価

低消費電力化の効果について、プログラム実行中のバッテリ電圧の推移をもって代替的に評価することとした。放電特性に伴い、電力消費に応じてバッテリ電圧が減少していくことが知られている。低消費電力を考慮しないもともとのプログラム実行におけるバッテリ電圧の推移を図 4 に示す。対して、AOP により効率的なライントレース方式（P 制御）への変更、効率的な光センサの制御を組み入れたプログラム実行の場合を図 5 に示す。左縦軸はバッテリ電圧初期値からのバッテリ電圧の降下した値を示している。右縦軸はどの状態にあるかを示している。そして、横軸は時間を示している。

両図の比較により、低消費電力化を図ったプログラムの方がプログラムを実行する際にバッテリ電圧の低下が少なく、低消費電力化は成功したといえる。

ライントレースを行っているのは、グラフにおける状態 1、5 である。その際のバッテリ電圧は低消費電力を考慮していないもとのプログラムの方がより降下している。

またアスペクトにより光センサの消灯をすることにしたのは、状態 2、3、6、7、8 である。図 5 ではライントレース方式も変更しているため光センサの消灯により効果があったのかどうかが判断しにくい。そこ

で、アスペクトにより効率的な光センサの制御のみを組み込んだプログラムを別途用意して（図 6）、確かに低商品電力化に寄与していることを確認した。

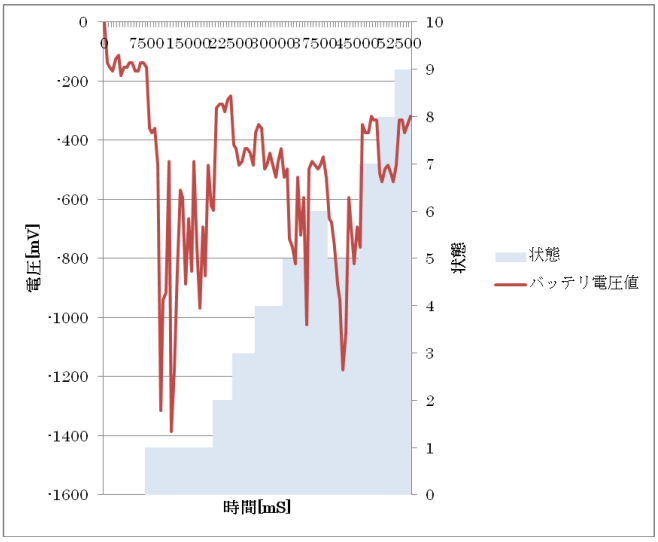

図 4: 省電力性を考慮しないもとのプログラム実行における電圧推移

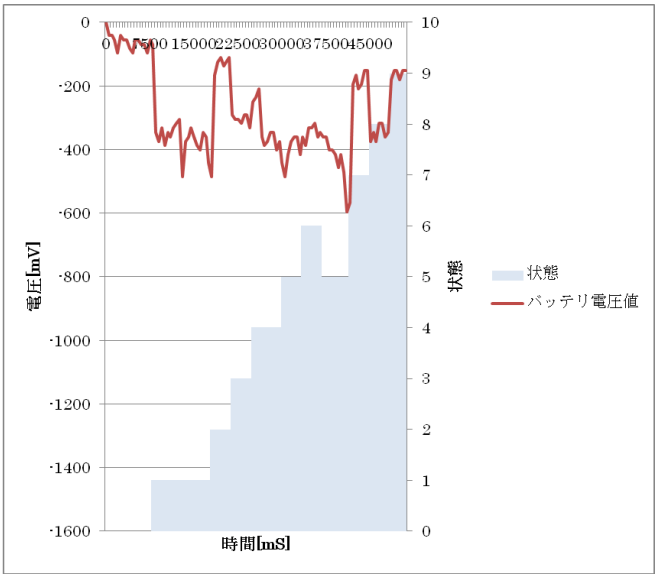

図 5: 両アスペクトを適用したプログラム実行における電圧推移

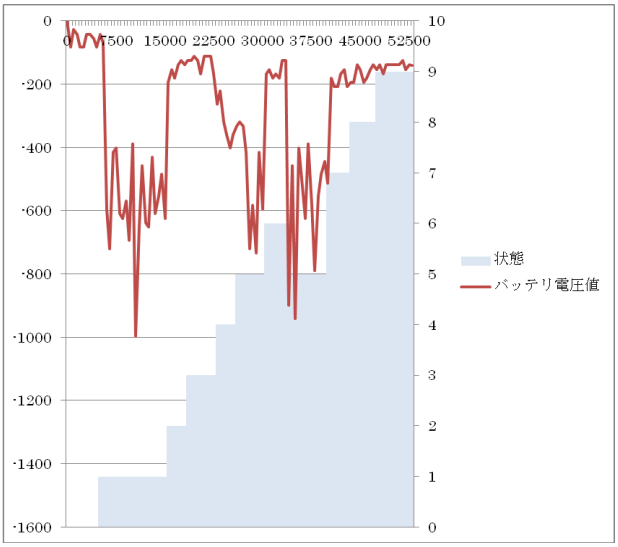

図 6: 光センサ制御アスペクトのみを適用したプログラム実行における電圧推移

3.3. 保守性の評価

低省電力化のための二つの戦略(ライントレース方式変更、光センサ制御）をそれぞれ、もとのプログラムに対して AOP を用いずに直接実装した場合と、AOP を用いて実装した場合で、その実装に必要なファイル数およびコードのまとまり数（チャンク数）を測定比較した。その結果を表 1 に示す。

AOP において、実現ファイルをアスペクト唯一つにまとめられることになったため、従ってコードのまとまりも単一となり、低消費電力化を実現するコードが散らばることなくモジュール化できていることが分かる。

このように低消費電力化を実現するコードを集約できたことで、プログラムの修正、拡張が必要な場合はアスペクトのみを見ればよく、多くのファイルを見ることがないので保守がしやすくなるといえる。また、もとのプログラムは改変してないため、プログラムの再利用性は高いといえる。ライントレース方式等の低消費電力化のアスペクトに関してもポイントカットを書き直せば、他のプログラムにも適用可能であり、再利用できる可能性がある。

表 1: 戦略実現コードの散らばりの比較

戦略の実装方法	ライントレース		光センサ	
	ファイル数	コードまとまり（チャンク）数	ファイル数	コードまとまり（チャンク）数
直接	4	5	2	3
AOP	1	1	1	1

4. おわりに

本稿では、品質の作りこみにおける AOP の応用として、単体テスト実行時間の短縮を目的とした単体テストフレームワークのマルチスレッド化と、ロボット制御プログラムの低消費電力化を目的とした制御アルゴリズムの追加および変更の事例をそれぞれ報告した。

いずれの事例においても、品質要求の満足・品質向上を達成しつつ、保守性を維持あるいは向上させることに成功しており、AOP が効率的かつ将来においても保守に優れた品質の作りこみの手段として有効なことを確認できたと考えられる。

今後の課題として、並行性・並列性・効率性や低消費電力化以外の品質要求の AOP による扱いの検証や、それらの実現における共通および特有の課題や定石の蓄積と整理が挙げられる。また、異なる品質特性間のトレードオフのような影響についても追加検証したい。

文　献

[1] Gregor Kiczales, et al., “Aspect-Oriented Programming,” Proc. European Conference on Object-Oriented Programming, pp.220-242, 1997.

[2] 長瀬嘉秀, 天野まさひろ, 鷲崎弘宜, 立堀道亜昭, "AspectJ によるアスペクト指向 プログラミング入門", ソフトバンク パブリッシング, 2004.

[3] 鷲崎弘宜, 高橋竜一, 村上真一, 大橋昭, 吉岡信和, 石川冬樹, 久保淳人, 山本里枝子, 小高敏裕, 錠尚史, 鹿糠秀行, 杉本信秀, “ディペンダビリティ確保にむけたアスペクト指向技術動向”, 情報処理学会第 168 回ソフトウェア工学研究発表会, 2010.5.

[4] JUnit, http://www.junit.org/

[5] Parallel JUnit, http://testng.org/doc/index.html

[6] J. L. Sobral, "Incrementally Developing Parallel Applications with AspectJ," 20th International on Parallel and Distributed Processing Symposium (IPDPS), 2006.

[7] Doug Lea, “A Java Fork/Join Framework,” Proceedings of the ACM conference on Java Grande, 2000.

[8] 大橋昭, 久保淳人, 水町友彦, 江口和樹, 村上真一, 高橋竜一, 鷲崎弘宜, 深澤良彰ほか, “AOJS：JavaScript のためのアスペクト指向プログラミング・フレームワーク”, コンピュータソフトウェア, 岩波書店, Vol.28, No.3, pp114-131, 2011.

[9] Alexandre Duarte, et al., "GridUnit : Software Testing on the Grid," Proceedings of the 28th International Conference on Software Engineering (ICSE), pp.779-782, 2006.

[10] 和田祐介, 大森洋一, 日下部茂, 荒木啓二郎, “ JUnit 向け単体テストを対象とした MapReduce 型並列分散実行フレームワークの提案”, 情報処理学会研究報告. ソフトウェア工学研究会報告 2010-SE-168(5), 1-7, 2010.

[11] 石原亨,冨山宏之, ” 低消費電力化ソフトウェア技術” , 組込みソフトウェアシンポジウム論文集, 2005(12), pp.188-190,2005.

[12] 赤山聖子, 久保秋真, 久住憲嗣, 二上貴夫, 北須賀輝明, “ ソフトウェア初学者へのモデリング教育における MDD の活用”, 組込みシステムシンポジウム 2011 論文集, 2011.

[13] Michael Gong, Charles Zhang, and Hans-Arno Jacobsen, “AspeCt-oriented C for Systems Programming with C,” AOSD 2007 Software Demonstration, 2007. http://www.aspectc.net/