情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-SLDM-149 No.12
Vol.2011-EMB-20 No.12
2011/3/18

テストプログラム生成ツールのフロントエンドプロセッサの開発

阿部 真也[†1] 鴫原 一人[†2] 本田 晋也[†2]
山本 晋一郎[†1] 高田 広章[†2]

本論文では，マルチプロセッサ対応 RTOS のテストプログラム生成ツールのフロントエンドプロセッサについて述べる．フロントエンドプロセッサを開発したことにより，テスト実施の効率が上がり，誤った入力データの検出能力が向上した．また，テストプログラム生成ツール全体の保守性も向上した．

Development of a Front End Processor for a Test Program Generator

Shinya ABE,[†1] Kazuto SHIGIHARA,[†2]
Shinya HONDA,[†2] Shinichiro YAMAMOTO[†1]
and Hiroaki TAKADA [†2]

This paper describes a front end processor for a test program generator for multiprocessor RTOS. The use of this front end processor helped us improve the efficiency of testing and the capability of error detection on the input data. It also helped us improve the maintainability of the test program generator.

1. はじめに

近年，マルチプロセッサ対応リアルタイム OS（RTOS）の需要が高まっており，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（NCES）[1] では，マルチプロセッサ対応 RTOS の包括的なテストをコンソーシアム型の共同研究で行っている．その中で，テスト開発を効率化するために，テストシナリオからテストプログラムを生成するツールである TTG（TOPPERS Test Generator）[2] を開発して使用している．

TTG は，コード生成をする際に入力データであるテストシナリオのエラーチェック処理と，入力データを整形するフォーマッティング処理を行う．しかし，現状の TTG は，これらの前処理がコード生成処理と混在した状態で実装されているという問題がある．さらに，エラー処理が不十分であるという問題がある．TTG がテストの対象とする RTOS には，シングルプロセッサ対応 RTOS とマルチプロセッサ対応 RTOS がある．これらの異なる RTOS に対応するためには，TTG はそれぞれの RTOS に対応したコード生成処理を持つ必要があるため，ソースコードが複雑になる．

さらに，新たな要求としてターゲットシステムでテストケースが実行できるかどうかを判断するバリエーション判別機能があり，この要求に対応する場合，TTG のソースコードがさらに複雑となる．今後も TTG に対する新たな要求が生じる可能性があるが，保守性が低下している状態では対応することが困難である．

本研究では，これらの問題と要求に対応するために，コード生成処理以外の処理を切り離し，独立したモジュールであるフロントエンドプロセッサとした．フロントエンドプロセッサは，コード生成処理の前に行う処理を担当する．エラーチェック処理の強化をするために，チェック項目を効率的に管理する方法と実装方法を検討し，実施した．また，フォーマッティング処理を再設計や，新たな要求であるバリエーション判別を実装した．

本研究は,「OJL による最先端技術適応能力を持つ IT 人材育成拠点の形成」[3][4] の OJL（On the Job Learning）の開発テーマとして実施した．

本論文の構成は次のとおりである．第 2 章で TTG の概要と問題点，また，TTG に対する新たな要求について述べる．第 3 章では，フロントエンドプロセッサの概要と機能を定義する．第 4 章，第 5 章，第 6 章では，定義したフロントエンドプロセッサの機能を個別に述べる．第 7 章では，フロントエンドプロセッサにより得られた効果を評価する．そして，第 8 章で本論文をまとめる．

†1 愛知県立大学
Aichi Prefectural University
†2 名古屋大学
Nagoya University

2. TOPPERS Test Generator

本章では，TTG の概要と問題点，また，TTG に対する新たな要求について述べる．

ⓒ 2011 Information Processing Society of Japan

2.1 API テスト

TTG がテストの対象とする RTOS の 1 つは，TOPPERS プロジェクト[5] で開発および公開されている，マルチプロセッサ対応 RTOS の TOPPERS/FMP カーネル（以下，FMP カーネル）である．FMP カーネルは，シングルプロセッサ向け RTOS の TOPPERS/ASP カーネル（以下，ASP カーネル）を拡張して開発されている．ASP カーネルと FMP カーネルの仕様は，TOPPERS 新世代カーネル統合仕様書[6]（以下，統合仕様書）に準拠している．

実施するテストの観点の 1 つとして，API テスト[7] がある．API テストでは，ASP カーネルおよび FMP カーネルが用意した API が，統合仕様書に定められたとおりに正しく振る舞うかを確認するテストである．

API テストでは，最初に API 発行前のシステム状態を定義し，その状態からテスト対象となる API を発行した時に，システム状態が統合仕様書に定められたとおりに変化するかどうかを確認する．API 発行前のシステム状態を「前状態」，発行する API の内容を「処理」，API 発行後のシステム状態を「後状態」と呼ぶ．

API テストの実施方法を述べる．最初に，統合仕様書からテスト対象となる API の振る舞いを理解し，テストケースを抽出する．また，抽出したテストケースに対してソースコードの通るパスを確認し，条件網羅表を作成する．次に，抽出したテストケースを前状態・処理・後状態の 3 つの項目に分けて表現したテストシナリオを作成する．そして，テストシナリオを独自の形式化表現記法である TESRY（TEst Scenario for Rtos by Yaml）記法で記述した TESRY データを作成する．最後に，TESRY データを TTG に入力し，テストプログラムを生成して実行する．そして，実行結果を確認するとともに，コードカバレッジを取得し，条件網羅表で定義したパスを通っているか確認する．

対象とするタスクを起動する API である act_tsk のテストケースの例を**図 1**，そのテストケースをテストシナリオの形式で表現した結果を**図 2**，さらに，そのテストシナリオを TESRY 記法で記述した例を**図 3** に示す．

2.2 テストプログラム生成ツールの必要性

API テストで実施するテストケースの数は数千件に上り，それら全てのテストプログラムをハンドコーディングで作成するには工数がかかる．また，API テストのテストプログラムの実現方法は複数あるため，テストプログラムの作成者間で実装のばらつきが生じ，可読性が低下する恐れがある．

これらの問題を解決するため，API テストのテストプログラムを生成するツールとして

対象タスクの優先度が実行状態のタスクと同じ場合は，実行可能状態となり，同じ優先度のタスクの中で最小の優先順位になること．

図 1 テストケースの例
Fig. 1 Example of test case

前状態
　タイプ: タスク
　ID: TASK1
　状態: 実行

　タイプ: タスク
　ID: TASK2
　状態: 休止

処理
　TASK1がact_tsk(TASK2)を発行し，
　エラーコードとしてE_OKが返る．

後状態
　タイプ: タスク
　ID: TASK2
　状態: 実行可能

図 2 テストシナリオの例
Fig. 2 Example of test scenario

```
test1:
  pre_condition:
    TASK1:
      type   : TASK
      tskstat: running
    TASK2:
      type   : TASK
      tskstat: dormant

  do:
    id      : TASK1
    syscall: act_tsk(TASK2)
    ercd    : E_OK

  post_condition:
    TASK2:
      tskstat: ready
```

図 3 TESRY データの例
Fig. 3 Example of tesry data

TTG を開発した．TTG は，TESRY データを入力することでその内容に応じたテストプログラムを生成する．TTG の処理の論理的な流れを**図 4** に示す．テストプログラムをツールで生成する事により，テストプログラムの開発工数を削減するだけでなく，担当者間での実装のばらつきを無くすことができる．

図 4 TTG の処理の論理的な流れ
Fig. 4 Logical flow of processing of TTG

また，TESRY 記法のフォーマットは，スペースと改行により階層構造を表現する YAML（YAML Ain't a Markup Language）[8] を利用しているため可読性が高く，手作業で記述することも容易である．さらに，ユーザの利便性を考慮し，テストと関係のない情報の記述を省略できる．省略した内容は，TTG が TESRY データを読み込む際に適切に補完する．テストの管理対象をテストプログラムから，その入力データとなる TESRY データとすることで，可読性が高い状態でテストを管理でき，テストの保守性が向上する．

2.3 問題点と新たな要求

API テストの開発を進める中で，TTG に対する新たな要求が生じた．これまでは，その要求に対して，TTG を機能拡張する形で対応してきたが，設計後の機能拡張を行ったこと

ⓒ 2011 Information Processing Society of Japan

により TTG の保守性が低下した．保守性の低下を含めた TTG の問題点は 3 つあり，また，いくつかの機能追加の要求がある．

1 つ目の問題点として，入力データのエラーチェック処理がコード生成処理と混在していることで全体の見通しが悪くなっていることがある．2 つ目の問題点として，TESRY 記法をユーザの利便性を考えた設計にした結果，TESRY データの解析処理が複雑になっていることがある．3 つ目の問題点として，エラーチェック処理の保守性が低く，チェック項目が管理できていないため，エラーチェックが不足している問題がある．

新たな要求として，テストケースによっては，テストプログラムの実行可否がターゲットシステムに依存するため，ターゲットシステムを変更した場合にテストできる TESRY データを自動で取捨選択したいという要求がある．

3. フロントエンドプロセッサによる TTG の再設計

2.3 節に挙げた問題点の解決方法を述べる．

これらの根本的な原因は，コード生成処理とコード生成以外の処理が混在していることである．そのため，コード生成以外の処理をコード生成処理から分けることでこれらの問題を解決できると考えた．問題点と新たな要求に対応するために再設計した TTG をバージョン 3.0 と位置付ける．TTG3.0 は，コード生成の前処理を担当するフロントエンドプロセッサと，コード生成処理を担当するコード生成モジュールから構成される（**図 5**）．

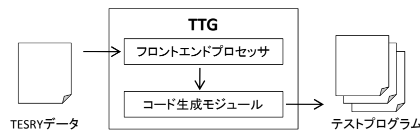

図 5 TTG3.0 の構成
Fig. 5 Composition of TTG3.0

TTG をフロントエンドプロセッサとコード生成モジュールに分け，それぞれのを独立させることにより，TTG の保守性が高くなる．また，フロントエンドプロセッサで入力した TESRY データをエラーチェックし，省略された情報を補完するなどの整形処理を行った結果をコード生成モジュールに入力する．そのため，コード生成モジュールは正しい入力データのみが入力される前提で実装できるので，ソースコードが簡潔となり保守性が高まる．

フロントエンドプロセッサの機能要件を，入力した TESRY データのエラーを検出するエラーチェック機能と入力した TESRY データの省略された情報の補完などの整形を行うフォーマッティング機能，入力した TESRY データの中からターゲットシステムで実行できるものだけを選択するバリエーション判別機能と定義する．また，エラーチェック機能では，チェック項目を効率的に管理するため，チェック項目を一定の法則の基にグループ分けする．

4. エラーチェック

エラーチェックは，入力した TESRY データが正しいかチェックを行う機能である．また，チェック項目を管理し，チェック内容に過不足がないかを効率的に管理する．エラーチェック機能の設計にあたって，チェック項目の整理と実装の検討を行う．

チェック項目の整理として，これまでの TTG からチェック項目を抽出し，不足しているチェック項目を追加してエラーチェックリストを作成する．各チェック項目には，全チェック項目を通してユニークな ID を割り付ける．また，そのチェック項目に該当するチェック処理を実装したソースコードにコメントで ID を記すことにより，エラーチェックリストとソースコードを対応付ける．

ID の割り付け以外に，チェック項目を内容によりグループ分けする．カテゴリ分けの基準として，設定ファイル内のチェックとして 1 つのカテゴリ，TESRY データのチェックとして参照する範囲によって 7 つのカテゴリを作成する．作成したカテゴリ一覧を**表 1** に示す．また，カテゴリ別の参照範囲を**図 6** に示す．以降，本文中でチェックカテゴリを指す場合は表中のカテゴリ ID で呼ぶ．

表 1 チェックカテゴリ一覧
Table 1 Check category list

カテゴリ ID	カテゴリ名	チェック内容	チェック項目数
T0	environment check	設定ファイル内の項目・マクロ	51
T1	basic check	TESRY データの構造	27
T2	attribute check	オブジェクト・処理の属性	220
T3	object check	オブジェクト内の依存関係	39
T4	condition check	コンディション内の依存関係	46
T5	scenario check	TESRY データ内の依存関係	17
T6	variation check	バリエーション指定との関係	14
T7	multiple check	TESRY データ間の依存関係	8

それぞれのカテゴリごとの参照範囲について述べる．T0 では，設定項目を参照する．T1

ⓒ 2011 Information Processing Society of Japan

Vol.2011-SLDM-149 No.12
Vol.2011-EMB-20 No.12
2011/3/18

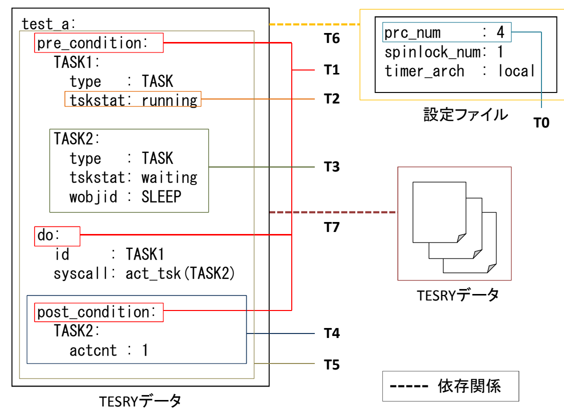

図 6 チェックカテゴリ別参照範囲
Fig. 6 Range of reference according to check category

では，TESRY データの構造を参照する．T2 では，各オブジェクトと処理に指定する 1 つ 1 つの属性を参照する．T3 では，各オブジェクトの属性同士を参照する．T4 では，前状態と後状態のそれぞれの状態で内部のオブジェクトを参照する．T5 では，1 つの TESRY データ全体を参照する．T6 では，1 つの TESRY データと設定ファイル内のターゲットシステムの設定を参照する．T7 では，入力する全ての TESRY データを参照する．

エラーチェックの実装方法として，YAML のスキーマバリデータの Kwalify[9] の利用を検討した．Kawlify は，外部に定義したスキーマの内容を基に YAML ドキュメントのエラーチェックを行う．TESRY データのエラーチェックにおいては，スキーマで TESRY データのエラーチェックを定義する事が不可能であるケースに遭遇したため利用を断念した．最終的に専用のエラーチェック処理を実装することに決定した．

5. フォーマッティング

フォーマッティングは，入力された TESRY データを解析し，コード生成に必要な整形を行う機能である．この機能により，ユーザが TESRY データを作成する際の利便性を向上させ，また，コード生成モジュールへの入力として適切に整形する事で，コード生成モジュールの実装を簡潔にする．

TESRY データを作成する時，オブジェクトを定義するために，テストの内容と関係のない属性も記述しなければならないとすると，TESRY データの情報が大きくなり可読性が低下する．また，テストの内容と関係のない属性の値を固定してしまうと，それらの属性の値を変更してテストを行いたい場合に TESRY データを修正する必要が生じる．これらの問題のため，TESRY 記法では属性の記述を省略することを可能とし，マクロを使用する事もできる．省略された属性は TTG が適切に補完し，マクロも TESRY データ読み込み時に対応する値に置換する．また，TTG は，複数の TESRY データを入力して 1 つのテストプログラムを生成する．TESRY データで登場するオブジェクト ID と変数名は生成するテストプログラム内で使用するため，それらの名前を重複させると不具合が生じる．そこで，テストケースごとに定義するユニークなテスト ID を，オブジェクト ID と変数名の先頭に付与することで，それらの名前を一意にする．フォーマッティングの例を**図 7** に示す．

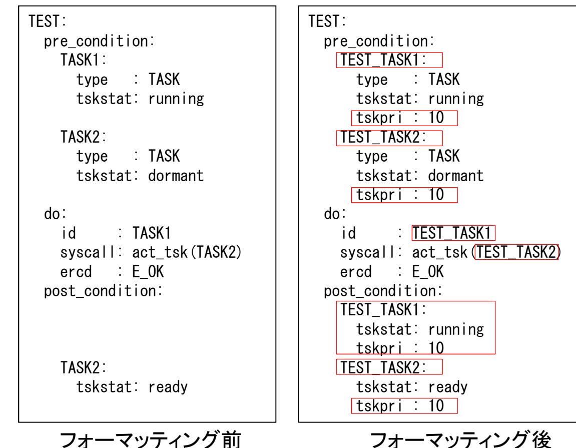

図 7 フォーマッティング例
Fig. 7 Example of Formatting

関連するツールに C 言語のプリプロセッサである cpp[10] があり，これはマクロの置換を行うことができる．しかし，TESRY データを読み込み，補完した結果からマクロの置換を行う必要があるため，最初から全てのマクロを記述したファイルを読み込む cpp では対応できない．また，TTG には条件によって動的に置換するマクロを選択する機能があるが，これも cpp では実現できない．パーサの自動生成ツールである yacc[11] を利用して TESRY

ⓒ 2011 Information Processing Society of Japan

データを解析する方法も考えられるが，省略して記述していない情報を解釈する必要があるため利用できない．

6. バリエーション判別

バリエーション判別は，入力する TESRY データがターゲットシステム上で実行できるかを判別する機能である．

ターゲット依存部の設定ごとに，実行できる TESRY データをリストで管理する方法もあるが，複数の設定の組み合わせを考慮すると組み合わせが膨大になる．また，リストの情報が古くならないよう管理する必要が生じる．

ターゲットシステムによって実行できるテストケースを判別するため，TESRY 記法を拡張して，テストに必要な API の情報や，要求するターゲットシステムの構成などの情報をバリエーション属性として記述可能とする．

フロントエンドプロセッサに TESRY データを入力する場合，バリエーション属性と入力する TESRY データの内容を解析し，指定したターゲットシステムで実行可能なテストケースかを判別する．そして，入力する TESRY データが指定したターゲットシステム上で実行できないと判別した場合は，その TESRY データをコード生成の対象から除外する．

バリエーション判別の観点を**表 2** に示す．

表 2 バリエーション判別に使用する観点
Table 2 Viewpoint used for variation distinction

観点	依存する内容
プロセッサ数	用意されたプロセッサの数
スピンロック数	用意されたスピンロックの数
タイマアーキテクチャ	時間管理方式
API のサポート	get_utm，ena_int，dis_int の API の有無
ターゲット依存関数	ターゲット依存部に用意する関数が利用できるか
自プロセッサへのプロセッサ間割込み	自身のプロセッサを対象にプロセッサ間割込みを発行できるか
CPU ロック中の CPU 例外発生のサポート	CPU ロック中に CPU 例外が発生しても元のコンテキストに戻れるか
時間動作判断	TTG がテストプログラムの時間を制御する必要があるか

7. 評　　価

本章では，開発したフロントエンドプロセッサにより得られた効果を機能ごとに評価する．

7.1 エラーチェック

これまでの TTG のチェック項目数は 124 件あったが，TTG3.0 で改めてチェック項目を抽出したところ，既存のチェック項目を含めて 422 件となった．そして，API テストに使用する全ての TESRY データ（ASP カーネル 1,667 件，FMP カーネル 2,542 件）を対象にエラーチェックを行った結果，ASP カーネルで 68 件，FMP カーネルで 69 件の TESRY データの不具合を検出した．エラー内容の内訳を**表 3** に示す．

表 3 TESRY データの不具合要因
Table 3 Defect factor in tesry data

エラー要因	ASP	FMP
YAML 構造の問題	18	0
マッピングのキーを二重に定義している	1	2
true・false の文字列が大文字になっている	14	0
真偽値を表す文字列が true・false ではない	5	2
エラーコードを確認するための属性の使用方法が不適切	0	8
post_condition で状態変化のないオブジェクトが記述されている	1	0
post_condition 内のオブジェクトに type 属性が定義されている	11	0
pre_condition で実行状態の処理単位が存在しない	2	4
実行状態でない処理単位に変数の value が定義されている	0	21
同一コンディション内で起動中の非タスクコンテキストが複数存在する	1	0
実行状態の処理単位が存在しない状態で CPU 状態を指定している	0	2
スピンロック中に CPU_STATE が定義されていない	0	2
エラーコード確認漏れ	13	14
確認できないエラーコード指定	2	13
プロセッサ番号を指定するマクロの組み合わせが不正	0	1

検出した不具合としては，YAML ライブラリに依存する YAML の誤記と，TESRY データ全体を見なければ検出できないエラーなどが多かった．また，生成するテストプログラム上で実際はチェックできていない属性が記述されている不具合もあった．

フロントエンドプロセッサで新たに不具合を検出できた理由としては，これまでの TTG では，エラーチェックはコード生成処理に支障が出ない範囲で行っていたことに対し，フロントエンドプロセッサのエラーチェックでは，生成されるコードが実行できて，かつ正しくテストが実施できるようにチェックしているからである．

7.2 バリエーション判別

バリエーション判別が必要となるテストケースは ASP カーネルで 548 件，FMP カーネルで 1,231 件になった．全バリエーション判別の項目ごとに該当するテストケースをまとめ

ⓒ 2011 Information Processing Society of Japan

た結果を**表 4** に示す.

表 4 バリエーション判別要因
Table 4 Variation distinction factor

判別要因	ASP	FMP
時間操作関数が必要なのに使用不可	532	1039
割込み発生関数が必要なのに使用不可	4	6
CPU 例外発生関数が必要なのに使用不可	9	10
CPU ロック中に CPU 例外発生のサポートがされていない	1	2
API の get_utm がサポートされていない	3	1
API の ena_int がサポートされていない	5	3
API の dis_int がサポートされていない	5	3
プロセッサ数が 3 つ以上必要	0	61
自プロセッサに対し割込みを発生させられない	0	204
ローカルタイマ方式でのみ実行可能	0	202
グローバルタイマ方式でのみ実行可能	0	17

それぞれの項目ごとの組み合わせを考えると，ターゲットシステムごとに実行できるテストケースの組み合わせは膨大な数になる．その組み合わせを考慮し，手動で選別する必要がなくなったことでテストの実行効率が上がったと言える．

7.3 考 察

現在，他の RTOS に対応するテストプログラム生成ツールが開発されているが，そのツールにおいてもコード生成処理の前処理をフロントエンドプロセッサで行う設計を採用している．また，エラーチェックのチェック項目においても同じ基準でカテゴリ分けをしているため，RTOS のテストプログラム生成ツールでフロントエンドプロセッサを用意する考えは有用であると言える．

8. ま と め

本研究では，テストプログラム生成ツールである TTG のフロントエンドプロセッサを開発した．

フロントエンドプロセッサを開発した背景として，TTG の問題点と新たな要求が生じたことがある．問題点として，TTG のソースコードの保守性が低下していることと，TTG への入力となる TESRY データのエラーチェック処理が不完全であることが挙げられる．新たな要求として，テストケースによっては，ターゲットシステムでテストプログラムが実行できない場合があるため，ターゲットシステムに応じて実行可能なテストケースを自動で取捨選択したいことがある．

これらの問題点と要求に対応するため，TTG3.0 の開発を計画した．保守性を高めるため，TTG3.0 の構成をフロントエンドプロセッサとコード生成モジュールに分割した．フロントエンドプロセッサの機能として，入力データのエラーチェック機能と，入力された TESRY データをソースコード生成モジュールで扱い易く整形するフォーマッティング機能，ターゲットシステムで実行可能なテストケースを取捨選択するバリエーション判別機能がある．エラーチェック機能でチェックする項目は，エラーチェックリストを作成し管理する．

フロントエンドプロセッサのエラーチェックによって，ASP カーネルと FMP カーネルの TESRY データで合わせて 137 件の不具合を検出した．また，バリエーション判別が必要なテストケース 1,779 件が TTG で取捨選択できるようになった．

謝辞 本研究にあたり，多くのご協力を頂いたコンソーシアム型研究の皆様に謹んで感謝の意を表する．

参 考 文 献

1) 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，http://www.nces.is.nagoya-u.ac.jp/.
2) 鴫原一人，眞弓友宏，本田晋也，高田広章：マルチプロセッサ対応 RTOS を対象としたテストシナリオ記述法とテストプログラム生成ツール，情報処理学会研究報告「組込みシステム (EMB)」(2010).
3) OJL による最先端技術適応能力を持つ IT 人材育成拠点の形成，http://www.ocean.is.nagoya-u.ac.jp/.
4) 小林隆志，沢田篤史，山本晋一郎，野呂昌満，阿草清滋：On the Job Learning: 産学連携による新しいソフトウェア工学教育手法，電子情報通信学会信学技報 SS2009-28, Vol.109, No.170, pp.95–100 (2009).
5) TOPPERS プロジェクト，http://www.toppers.jp/.
6) TOPPERS 新世代カーネル統合仕様書，http://www.toppers.jp/docs/tech/ngki_spec-120.pdf.
7) 鴫原一人，松浦光洋，眞弓友宏，本田晋也，山本雅基，高田広章ほか：組込みリアルタイム OS に対する API テストの実施，ソフトウェアテストシンポジウム 2010 予稿集，pp.46–53 (2010).
8) YAML, http://www.yaml.org/.
9) Kwalify, http://www.kuwata-lab.com/kwalify/.
10) The C Preprocessor, http://www.sra.co.jp/wingnut/gcc/cpp-j.html.
11) The Lex & Yacc Page, http://dinosaur.compilertools.net/.

ⓒ 2011 Information Processing Society of Japan