Vol.2010-SLDM-145 No.10
2010/5/20

情報処理学会研究報告
IPSJ SIG Technical Report

# 充足可能性判定に基づくシステムレベルデバッグ支援手法におけるバグモデルの導入による効率化

原田裕基[†1]　西原佑[†2]
松本剛史[†3]　藤田昌宏[?,†4]

現在デバッグの作業において，バグの位置を特定する作業に特に時間が費やされている．本研究では，システムレベル設計を対象とした，バグの位置を特定する手法を提案する．さらに，この手法を効率化するために本稿ではバグモデルを導入する．バグモデルは設計記述において起こり得る記述誤りを形式的に定義したものである．このバグモデルの導入により，バグの具体的な修正方法が明らかになり，よりデバッグに有用な情報を得る事が可能になる．実験結果により，本手法によってバグの位置を大幅に絞り込め，デバッグに有用な情報を得る事ができる事を示す．またケーススタディにより，バグモデルを複合的に利用することによって，ある程度複雑なバグに対してもデバッグ支援を行える事を示す．

# Improving the Efficiency of System-Level Debug Support based on Satisfiability Problem by Introduing Bug Models

Hiroki Harada,[†1] Tasuku Nishihara,[†2]
Takeshi Matsumoto[†3] and Masahiro Fujita [†3,†4]

If an errorneous behavior is detected by simulation of hardware designs, it is very difficult and time-consuming to identify and fix the bug. In this paper, we propose a method to identify the location of bugs by utilizing SAT solvers, for a given system-level design and a set of failing patterns that make the outputs of the design incorrect. Also, to get possible solutions to fix bugs, we introduce bug models into the SAT-based proposed method. In this method, bug models are introduced in the design under debugging, and the method tries to find which bug models can make the design outputs correct for a given failing patterns. Experimental results show that the proposed method can provide the possible locations of bugs and possible solutions to fix them. Also, we describe that the proposed method can provide the debug solutions for real bugs.

## 1. はじめに

近年，組み込みシステムや SoC(System-on-chip) の設計においては，抽象度の高い段階でシステム全体を設計するシステムレベル設計から設計を始める場合が増えている．従来の RTL(Resister Transfer Level) に比べ，抽象度の高い設計記述が可能になるため，システムレベル設計の導入により，設計効率が高まることが期待されている．システムレベルから設計を行う場合，バグが抽象度の低い段階で検出されると手戻りの手間が発生するため，可能な限り多くのバグをシステムレベルにおいて検出・修正する必要がある．システムレベル設計における検証は主にシミュレーションによって行われるが，形式的手法（動作合成前後の等価性検証やプロパティ検証等）も多く提案されている．一方，検証によってバグがあることが分かった場合，そのバグの場所の特定や修正には多くの時間とコストがかかるが，有効な手法はほとんど提案されていない．そこで本研究では，システムレベル設計における検証によって得られた反例から，設計記述中のバグ位置を特定し，その修正候補を提示する手法を提案する．

本研究で提案する手法では，システムレベル設計においてバグが検出された場合に得られる誤った出力値に対して，同じ入力値を与えて正しい出力値を得るためには設計のどの部分を修正すべきか，を解析することによってバグ位置の特定を行う．そのため，反例の入力値に対する出力値，設計記述が必要になる．システムレベルでは SpecC，SystemC といった C ベース設計記述言語が用いられるため，本研究では，設計はそれらの C ベース言語で記述されているものとする．

†1 東京大学大学院工学系研究科電気系工学専攻
Dept. of Electrical Engineering and Information Systems, ,
The University of Tokyo
†2 東京大学大学院工学系研究科電子工学専攻
Dept. of Electronics Engineering and Information Systems,
The University of Tokyo
†3 東京大学大規模集積システム設計教育研究センター
VLSI Design and Education Center, The University of Tokyo
†4 科学技術振興機構戦略的創造研究推進事業
Core Research for Evolutional Science and Technology, Japan Science and Technology Agency

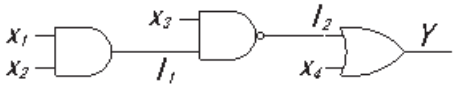


**図 1** ゲートレベル回路図の例
Fig. 1 Example of circuit

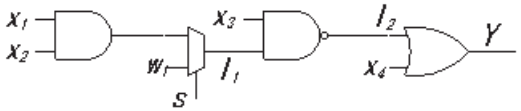


**図 2** ゲートレベル回路図におけるマルチプレクサの挿入例
Fig. 2 Example of circuit with multiplexer

与えられた誤った出力値を正しい出力値にするために変更すべき回路の箇所を求める手法は，ゲートレベルにおいては既に提案されている．本研究ではまず，その既存手法をシステムレベルで扱えるように，ワードレベル変数を含む論理式の充足可能性判定問題として定式化した．また，バグの可能性のある箇所と修正方法をより狭い範囲に絞り込むために，バグモデルを導入している．既存手法では，任意の修正によって出力値を正しくする箇所を求める事ができるが，具体的な修正方法を求める事が出来なかった．本研究では，バグモデルを導入することにより，誤った出力値を正しい値にするために修正すべき箇所と同時に，その修正方法も設計者に提示することができる．

本稿では，前述のように，システムレベル設計記述に対するデバッグ支援手法として，与えられた誤った出力値を正しくするために修正すべき箇所とその修正方法を求める手法の提案と評価を行う．まず，修正箇所と修正方法を求める手法を，ワードレベル変数を含む論理式の充足可能性判定問題として定式化し，評価を行う．また，修正方法の候補として用いる 2 種類のバグモデルを定義し，それらの組み合わせによってどのようなバグが特定・修正可能であるかを例題への適用を通して議論する．

## 2. 関連研究：ゲートレベルにおける，SAT ソルバを用いたバグ位置特定手法[1)2)3)]

ゲートレベルにおけるバグ位置特定手法は既に提案されている．本節では，この手法について詳しく説明する．

(1) マルチプレクサの挿入
ここでは，例題として，図 1 の回路を用いる．この回路は，$x_1$，$x_2$，$x_3$，$x_4$ を入力とし，$y$ を出力とする組み合わせ回路である．この手法ではまず，図 2 のように，回路の信号線のうちバグの可能性のある信号線にマルチプクサを挿入する．

(2) 制約式の導出
次に，マルチプレクサを挿入した図 2 の回路に対する制約式を作成する．具体的には，各回路素子に対する制約式を求め，それら全ての積を求める．さらに，この制約式に入力パタン及びそれに対する仕様に対して正しい出力パタンによる制約を制約式に書き加える．この例の場合，入力である $x_1$，$x_2'$，$x_3$，$x_4'$ 及び出力の $y$ が書き加えられることになる．

(3) バグ位置の特定
以上により得られた制約式を SAT ソルバに与えて解かせる事により，バグの位置を特定する．具体的には，解において，あるエッジに挿入されたマルチプレクサのセレクタ変数の値が 1 であった場合，その信号線の値を適当な値に変更することにより，回路が正しい出力値を出すことができるようになる．

## 3. ワードレベル充足可能性判定問題を利用したシステムレベルデバッグ支援手法

本節では，前節で述べた手法をシステムレベル設計に拡張する形で高位設計記述におけるバグ位置特定手法を提案する．ゲートレベルではデータをビットレベルで扱っているが，システムレベル設計では C ベース言語を用いて記述されるため，データは一般にはワードレベルである．このため，手法の中でもデータをビットレベルではなくワードレベルで扱う必要がある．データをワードレベルで扱うために，本研究では DFG(Data Flow Graph) を導入する．ここで，DFG に展開するにあたり，条件分岐等があると DFG への一意的な展開が出来ないため，ダイナミックスライシングを用いてあらかじめ必要な部分のみを抽出する必要がある．

次に，このようにして得られた DFG のエッジ上にマルチプレクサを挿入する．さらに，このマルチプレクサ付きの DFG が持つ変数が満たすべき制約式を生成する．この制約式を解くことによりバグの位置を特定する．

### 3.1 対象とする設計

本手法では，以下のような前提を満たす設計記述のみを対象として扱うものとする．

- 動作合成可能なシステムレベル記述
  これは，ポインタ，再帰呼び出し等のハードウェアでは扱えない記述を含まないという

```
int main() {
  int a, b, c, d, e;
  d = a + b;
  e = c * d;
}
```

**図 3** 対象とする設計例
Fig. 3 Design example

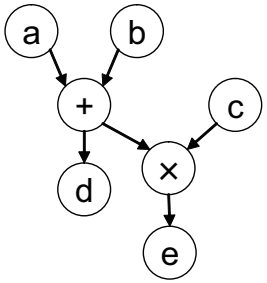


**図 4** 設計例の DFG
Fig. 4 DFG of the design

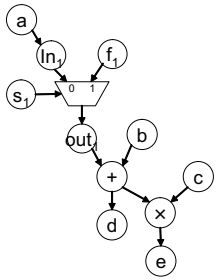


**図 5** DFG へのマルチプレクサの挿入例
Fig. 5 DFG with multiplexers

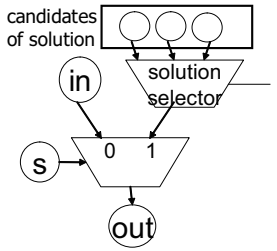


**図 6** バグモデルの導入
Fig. 6 Introduction of bug model

ことを示す.

- SSA(Static Single Assignment) 形式に変換されている
  SSA は，記述内の各変数が一度のみ代入される形式である．この形式に変換することにより，各変数の値が一意的に決定されることになり，一意的に制約式を生成することが可能になる．

### 3.2 ダイナミックスライシングによるデータフローの抽出

ダイナミックスライシングは，設計中から，ある入力パタンにおいてある出力変数に影響を与える記述を抽出する技術である．ダイナミックスライシングを用いることにより，今用いている入力パタンによる実行におけるデータの流れを抽出することができる．ダイナミックスライシングにはいくつか種類がある[6)7)] が，ここではデータスライシングを用いる．データスライシングは，データ依存だけを辿る手法である．文 s1 において定義された変数が，文 s2 において使用されている時，s2 は s1 にデータ依存があると言う．

ダイナミックスライシングにより抽出されたデータの流れは，DFG により表現される．DFG とは，プログラムのデータの流れを表現した有向グラフである．例えば，図 3 のようなプログラムがあった時，このプログラムを DFG により表現すると図 4 のようになる．ここでは，$a$ のノードと $b$ のノードから $+$ のノードにエッジを繋ぎ，そのノードから $d$ のノードにエッジを繋ぐことにより，演算 $d = a + b$ を表現している．さらに，この $+$ のノードから $\times$ のノードにエッジを繋ぐことにより，$a + b$ の答えである $d$ が $e = c * d$ という演算に用いられることを表現している．

### 3.3 制約式の作成

次に，作成された DFG のエッジ上に擬似的にマルチプレクサを挿入する．例えば，図 4 のエッジのうち 1 つにマルチプレクサを挿入すると，図 5 のようになる．

さらに，作成されたマルチプレクサ付きの DFG から，この中で用いられている変数に対する制約式を求める．具体的には，各マルチプレクサによる制約と，各演算に対する制約全ての論理積である．例えば，図 5 のマルチプレクサに対しては，$out_1 = s1?(f_1 : in_1)$ が成立する時に 1 となるような式を加えることになる．

さらに，ここに入力パタンと，それに対する求められる出力による制約式を加える．以上のようにして作成された制約式を SAT ソルバに解かせることによりバグの位置を特定する．なお，SAT ソルバとしては boolector[4)5)] を用いた．

### 3.4 手法の問題点

この手法の問題点として，具体的なデバッグの方法が明らかにならない点が挙げられる．この手法では，あるエッジの値を適当な値に修正すれば正しい出力値を得られるという情報が得られる．ところがこの情報だけでは実際のコード上でどう修正すればいいのかが分からず，デバッグが行えない．

この問題を解決するため本稿では，バグモデルを導入する．バグモデルとは，設計上で起こりうる記述誤りを形式的に定義したものである．具体的な修正方法をバグモデルの範囲内から選択することで，具体的なデバッグの方法を明らかにすることが可能となる．

## 4. バグモデルの導入によるデバッグ効率化

バグモデルを用いたデバッグは，一般的には図 6 のように，自由変数の代わりに修正候補を選択するマルチプレクサを導入することで実現される．自由変数のかわりに，バグモデルの範囲内に存在する修正候補から具体的な修正方法を選択することにより，デバッグ対象の

```
1 int main() {
2   int a, b, c, d, mean,
3       a2, b2, A;
4   mean = (a + b) >> 1;
5   c = a - mean;
6   d = b - mean;
7   a2 = c ^ 2;
8   b2 = d ^ 2;
9   A = (a2 + b2) >> 1;
10}
```

**図 7** 実験に用いた例題
Fig. 7 Design example used in experiment

設計記述が与えられた反例に対して正しい出力値をだすことが可能になる．以下では，2 つのバグモデルを提案し，それぞれに対する実験結果を示す．

次節で述べるバグモデルは，ソフトウェアプログラムやハードウェア設計の mutation テストにおける mutation のサブセットである．mutation テストでは与えられたテストパタンによって設計中の mutation が検出できる (出力値が異なる) かどうかを調べるのに対し，本研究では与えられた反例に対して，誤った出力値を仕様を満たす特定の値にすることができるバグモデル (mutation) の集合を求めることが目的となっている．

### 4.1 バグモデルの例

#### 4.1.1 演算における引数の誤り

これは，演算ノードを終点とするエッジに，引数となっている変数以外の変数を選択するマルチプレクサを挿入することで実現される．元の設計における，エッジの始点となるノードを $in$，終点となるノードを $out$ とすると，このバグモデルの挿入は以下の 2 式を満たす．

$$out = \begin{cases} in & (if \quad !s) \\ v_1 & (if \quad s \wedge s_1) \\ \cdot & \\ \cdot & \\ \cdot & \\ v_n & (if \quad s \wedge s_n) \end{cases}, \quad \sum_{i=1}^{n} s_i = 1$$

ここで，$s$ はそのエッジを修正すべきかどうかを判断するマルチプレクサの制御信号，$v_1$，$v_2 \cdots v_n$ は，設計中に存在する各変数，$s_1$，$s_2 \cdots s_n$ は，修正候補を選択するマルチプレクサの制御信号である．

**表 1** 演算における引数の誤りのバグモデルを用いた実験結果
Table 1 Experimental result using bug model " description mistake of parameter "

| 元の設計における記述 | バグ挿入後の記述 | バグ位置候補数 | 修正候補数 | | |
|---|---|---|---|---|---|
| $a2 = c^2$ | $a2 = b^2$ | 2 | 4 | 2 | 3 |
| $c = a - mean$ | $c = b - mean$ | 4 | 9 | 4 | 4 |
| $A = (a2 + b2)/2$ | $A = (c + b2)/2$ | 2 | 4 | 1 | 2 |
| $d = b - mean$ | $d = a - mean$ | 8 | 16 | 6 | 6 |
| $c = a - mean$ | $c = a + mean$ | 1 | 2 | 1 | 2 |
| $A = (a2 + b2)/2$ | $A = (a2 * b2)/2$ | 1 | 1 | 1 | 1 |
| $A = (a2 + b2)/2$ | $A = (a2 * b2) - 2$ | 3 | 3 | 1 | 1 |

**表 2** バグモデルを複数用いた実験結果
Table 2 Experimental result using bug multiple model

| 用いたバグモデル | A | B | A | A | A | A | A | B | A | B |
|---|---|---|---|---|---|---|---|---|---|---|
| 位置の候補数 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | |
| 修正数 | 1 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | | |
| 用いたバグモデル | A&A | A&A | A&B | A&B | A&B | A&B | A&B | A&B | | |
| 位置の候補の数 | 1 | 1 | 1 | 1 | 3 | 4 | 1 | 11 | 2 | 5 |
| 修正候補数 | 1 | 1 | 1 | 1 | 3 | 4 | 1 | 11 | 2 | 5 |

#### 4.1.2 演算子の誤り

演算ノードを始点とするエッジに，元の設計とは異なる演算結果を選択するマルチプレクサを挿入して，演算種類の誤りを表現する．このバグモデルの挿入は，以下の 2 式を満たす．

$$out = \begin{cases} op(if \quad v_1 \cdots, v_m) & (!s) \\ op_1(if \quad v_1 \cdots, v_m) & (s \wedge s_1) \\ \cdot & \\ \cdot & \\ \cdot & \\ op_n(if \quad v_1 \cdots v_m) & (s \wedge s_n) \end{cases}, \quad \sum_{i=1}^{n} s_i = 1$$

ここで，$s$ はそのエッジを修正すべきかどうかを判断するマルチプレクサのセレクタ変数，$v_1$，$\cdots$，$v_n$ は，今注目している演算ノードにおいて用いられている各変数，$s_1$，$\cdots$，$s_n$ は，修正候補を選択するマルチプレクサのセレクタ変数である．

### 4.2 実 験 結 果

以上 2 つのバグモデルに対する実験を，図 7 のような SpecC による設計例に対して行った．この結果を表 1 に示す．ここで，上 4 つは演算の引数の誤りによるバグモデルを用いた例，下 3 つは演算子の誤りによるバグモデルを用いた例である．

この表において，2 列目，3 列目は，どのようなバグを挿入したかを示している．例えば表 1 の一つ目のバグは，6 行目の $a2 = c^2$ という記述を $a2 = b^2$ と書き換えたものである．3 列目，4 列目は入力パタンを一つだけ用いた場合の結果である．3 列目はバグの位置の候補を示しており，4 列目は修正候補の数を示している．5 列目，6 列目は入力パタンを複数

入れた場合の結果である．今回の実験では，バグの位置の候補の総数は演算の引数の誤りによるバグモデルを用いた例おいては 10，演算子の誤りによるバグモデルを用いた実験においては 8 である．また，修正候補の総数は演算の引数の誤りによるバグモデルを用いた例おいては 70，演算子の誤りによるバグモデルを用いた実験においては 48 である．なお今回の実験では，入力パタンが二つの場合と三つ以上の場合では結果の違いは観測されなかった．

バグモデルを導入する前の手法と異なり，この実験では具体的な修正方法が提示されており，デバッグに有用な情報が得られた事が確認された．また，いずれの場合もデバッグの候補の数を大きく減らす事に成功しており，本手法がバグの位置を特定する際に有効であることが確認された．

以上の実験とは別に，図 7 の例題に対して 20 種類のバグを用意してそれぞれに対して実験を行った．うち 10 種類は記述の 1 箇所のみをバグモデルの範囲内でランダムに改変し，残り 10 種類は記述の 2 箇所を同時にランダムに改変することでバグを挿入した．この結果をまとめたものを表 2 に示す．ここで，1 行目及び 4 行目において，A は演算における変数の誤りによるバグモデル，B は演算の誤りによるバグモデルを指す．いずれの場合も，正しい出力値を得るための修正候補が得られた．これにより，本手法がデバッグ支援に有効であること，バグが複数存在した場合にも有効であることが示された．

バグを 1 箇所のみに挿入した例においては，バグの位置としては 1 箇所のみが示された．ただし，修正方法としては複数示された例が存在し，これらはいずれも演算における変数の誤りによるバグモデルを用いた例である．これは，図 7$a$2 と $b$2 のように実装上常に同じ値を持つ変数の組が存在していたため，他の変数に置き換えても全く同じ計算結果になる場合が存在するためである．

一方で，複数箇所にバグを挿入した例においては，多くの例においてバグの位置が複数示された．得られた修正方法いずれも，その修正方法に従えば現在の入力パタンに対しては正しい出力値が得られる．この例題のように単純な演算のみの設計であれば，これらの修正方法のうちいずれを用いることでも設計が仕様を満たすように修正できる．

## 5. ケーススタディ

本節では，いくつかの設計例を通して，バグモデルを導入したバグ位置特定手法の有効性を検証する．上で述べたバグモデルはそれぞれは簡単なものだが，組み合わせたり複数を同時に適用することでより複雑なバグを修正できる可能性があることを示す．

```
1   x3 = x1 + x3;
2   x3 = x4 - x2 + x3;
3   x5 = x5 - x6 + x2;
4   x4 = x2 + x4 + x7;
5   x6 = x5 + x7;
6   x6 = x1 + x2 * x3;
```

**図 8** ケース 1：改変前
Fig. 8 Case 1 : Before change

```
1   x3 = x1 + x3;
2   x6 = x5 + x7;
3   x3 = x4 - x2 + x3;
4   x5 = x5 - x6 + x2;
5   x4 = x2 + x4 + x7;
6   x6 = x1 + x2 * x3;
```

**図 9** ケース 1：改変後
Fig. 9 Case 1 : After change

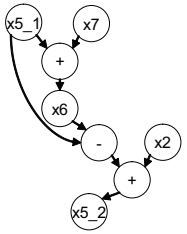


**図 10** ケース 1 の DFG
Fig. 10 DFG of case 1

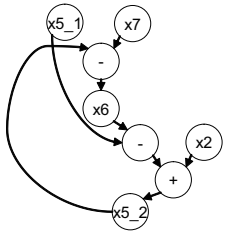


**図 11** ケース 1 の DFG に対する修正方法
Fig. 11 Solution for DFG of case 1

### 5.1 演算の順序変更

図 8 のような設計に対して，図 9 のように 5 行目の演算の位置を 2 行目に移動させることによるコード移動を想定する．この変更を行うと，図 8 の 3 行目と 5 行目に，x5 に関して依存関係があるため，最終的な計算結果が変わってしまう．つまり，この改変によってバグが挿入されてしまったと言える．この例に対して本手法を適用すると以下のようになる．

まず，設計を DFG に変換する前段階として，図 9 を SSA 形式に変換する．簡単のため，問題となる図 9 の 2 行目と 4 行目だけを抽出して考えると，以下のようになる．

x6 = x5_1 + x7;

x5_2 = x5_1 - x6 + x2;

これを DFG に変換すると，図 10 のようになる．これに対して提案手法を適用すると，図 11 のように，2 つのバグモデルを同時に適用する事でバグが直るという結果が得られる．これにより，演算の順序を逆にすれば正しい出力値を得られるようになることが分かる．

```
   INPUT : a, b, c;
1  if(b + c > 3){
2    b = c - 5;
3    a = b + c;
4  else{
5    d = b - 2;
6    d = d - b + c;}}
```

**図 12** ケース 2：改変前
Fig. 12 Case 2 : Before change

```
   INPUT : a, b, c;
1  A = b + c;
2  if(A > 3){
3    b = c - 5;
4    a = A;
5  else{
6    d = b - 2;
7    d = d - A;}}
```

**図 13** ケース 2：改変後
Fig. 13 Case 2 : After change

### 5.2 投機的実行

図 12 の設計において，$b+c$ という記述が複数回表れるため，図 13 のように 1 箇所にまとめる共通部分式の括り出しを想定する．ところが，図 12 の 2 行目で b に代入が行われているため，最終的な計算結果が変わってしまうことになり，この場合もソースコードの改変によってバグが挿入されてしまっていると言える．この例に対して本研究の手法を適用すると以下のようになる．

この例の場合，初期入力である $b$，$c$ の値によって実行内容が変わるため，DFG を生成する前にダイナミックスライシングにより出力 $d$ に影響を与える記述を抽出する．改変前の記述と $d$ の値が変わってしまうのは，if 文で正の分岐側が選ばれた場合だけである．よって，ダイナミックスライシングの結果，図 13 の 1，3，4 行目が抽出される．提案手法によって制約式を作成してソルバに解かせると，ケース 1 と同様の流れで $A = b - c$ は $b = c - 5$ よりも後で実行されなければならないという結果が得られる．

## 6. まとめと今後の課題

本稿では，システムレベル設計を対象とするバグ位置特定手法を提案した．また，この手法における，バグの具体的な修正方法が明らかにならないという問題点に対し，バグモデルを導入した手法を提案した．バグモデルを導入することにより具体的なバグ修正の候補を提示することが可能になり，より効率的にバグの位置候補を絞ることが可能になった．今回は二つのバグモデルについて，バグの位置候補，修正候補を大幅に減らすことが可能になったことを実験により示した．また，ケーススタディを通して，複数のバグモデルを同時に用いたり，複数箇所のバグモデルを同時に用いたりすることにより，実際に起こり得るより複雑なバグを特定・修正可能となることを示した．

今後は，上位設計記述で用いられる並列動作・通信に関するバグモデルを提案することにより，さらに複雑なケースに対して対応できるような手法の改良・評価を行う予定である．また，より広い範囲のバグを扱えるようにするため，利用するスライシング手法の再検討，バグモデルを用いない手法とバグモデルを用いる手法を組み合わせて適用する手法の検討を行う予定である．

## 参 考 文 献

1) A. Smith, A. Veneris, M. F. Ali, A. Viglas, “ Fault diagnosis and logic debugging using Boolean satisfiability, ”In *Proc. Microprocessor Test and Verification*, Vol. 4, pages 60-65, May 2003.
2) S. A. Safarpour, “ Formal Methods in Automated Design Debugging, ”PhD Thesis, University of Toronto, 2008.
3) M. F. Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, M. Abadir, “ Debugging Sequential Circuits Using Boolean Satisfiability, ”In *Proc. International Conference on Computer Aided Design*, Vol. 15, pages 204-209, Nov 2004.
4) R. Brummayer, A. Biere, “ Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays, ”In *Proc. 15th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, vol. 5505, pages 174-177, May 2009.
5) R. Brummayer, A. Biere, F. Lonsing, “ BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking ”, In *Proc. 1st Intl. Workshop on Bit-Precise Reasoning*, Vol. 367, pages 33-38, Jul 2008.
6) J. Krinke, “ Advanced slicing of sequential and concurrent programs, ”Master Thesis, *Universitätof Passau*, 2003.
7) X. Zhang, H. He, N. Gupta, R. Gupta, “ Experimental evaluation of using dynamic slices for fault location ”, In *Proc. international symposium on Automated analysis-driven debugging*, Vol. 6, pages 33-42, Sep 2005.