社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2008－SLDM－134／2008－EMB－8 (23)
2008 / 3 / 28

システムレベル設計記述に対する具体値・記号値シミュレーションによる入力パターンの自動生成手法

小島 慶久† 西原 佑† 松本 剛史† 藤田 昌宏††

† 東京大学 大学院 工学系研究科 電子工学専攻
〒 113-0032 東京都文京区弥生 2-11-16 武田先端知ビル 407
†† 東京大学 大規模集積システム設計教育研究センター
〒 113-0032 東京都文京区弥生 2-11-16 武田先端知ビル 407
E-mail: †{kojima,tasuku,matsumoto}@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

あらまし システムの大規模複雑化に伴い，シミュレーションベースの検証やデバッグにおいて，人手による入力パターンの作成が困難になってきている．そこで本研究では，設計記述を拡張システム依存グラフで表現し，設計記述に対して具体値および記号値によるシミュレーションを行い，アサーション違反を引き起こすような入力パターンの自動生成を行う手法を提案する．これにより，入力パターンの作成の自動化を支援し，また，ランダムシミュレーションでは扱いにくいようなコーナーケースの活性化および不具合の発見，コードカバレッジの改善を目指す．

キーワード 記号シミュレーション，記号・具体混合シミュレーション，入力パターン生成，検証，コードカバレッジ

A technique of automatic input pattern generation for system-level design descriptions by concrete and symbolic simulations

Yoshihisa KOJIMA†, Tasuku NISHIHARA†, Takeshi MATSUMOTO†, and Masahiro FUJITA††

† Electronic Engineering, the University of Tokyo
Takeda Bldg 407, 2-11-16 Yayoi, Bunkyo-ku, Tokyo 113-0032 Japan
†† VLSI Design and Education Center, the University of Tokyo
Takeda Bldg 407, 2-11-16 Yayoi, Bunkyo-ku, Tokyo 113-0032 Japan
E-mail: †{kojima,tasuku,matsumoto}@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

Abstract As the VLSI systems grow larger and more complicated, it becomes more difficult to manually prepare the input patterns on simulation-based verification and debugging. In this research, we try to automatically generate the input patterns which cause the assertion failure, by exercising concrete and symbolic simulations simultaneously on the design descriptions in the extended system dependence graphs. Our goals are i) to promote to automate the input pattern generation process, ii) to activate the corner-cases and discover the bugs hard to find by random simulation, and iii) to achieve a better code coverage.

Key words symbolic simulation, concrete/symbolic-hybrid simulation, input pattern generation, verification, code coverage

1. 導 入

近年，半導体の集積度の向上に伴い，大規模集積回路 (VLSI) の設計規模が増大している．設計規模の増大に対応して生産性を改善するために，SystemC [1] や SpecC [3] などのCライクなシステムレベル言語による高位設計が注目されている．

これらの高位設計のフローでは，ソフトウェアやハードウェアを含めたシステム全体の振舞いをシステムレベル言語で統一的に記述しておき，設計支援ツールや手作業によって，ソフトウェア・ハードウェア分割を含むアーキクチャ分割，最適化，並列処理の導入などを行なって設計記述の詳細度を上げ，最終的にアセンブリコードやレジスタ転送レベル (RTL) 記述などを作成する．

この詳細化の過程で，ツールの不具合やヒューマンエラーによって，設計誤りが混入することがある．詳細化の工程が進むにつれ，設計記述の量は増えるので，設計誤りの発見や修正は

より難しくなり，またそれに関わる時間や費用も大きくなるため，設計誤りをなるべく早い段階で発見することが重要である．そのために，設計記述が正しいことを証明し，あるいは設計誤りを発見するための検証やデバッグの技術が必要となる．

主な検証としては，設計記述がいかなる入力に対しても仕様(プロパティ)を常に満たすかどうかを調べるプロパティチェッキングや，二つの設計記述がいかなる入力に対しても同じように振る舞うかどうかを調べる等価性検証がある．また，その主な実現方法としては，入力パターンを人手もしくは自動生成によって用意し，設計記述をシミュレーションするというシミュレーションベースの手法と，設計記述を論理式などの形式的なモデルに変換し，数学的な手法によって証明を行う形式的手法がある．

シミュレーションは，大規模・複雑な記述でも動作する反面，正確に検査できるのは試した入力パターンについてのみであり，通常入力パターンの数は膨大なため，ありうる全ての状況を完全に網羅することは困難であり，特に，不具合がごくまれな状況で発生するコーナーケースを発見することは難しい．また，いったん不具合を発見できた場合は，記述に問題があると結論づけられるが，不具合を発見できなかった場合は，本当に不具合がないのか，あるいは単純に検査が不十分なのか区別しづらい．そのため，分岐カバレッジやパスカバレッジなどの指標を判断材料の一つとする．

形式的手法では，数学的に全て網羅できる(入力パターンを個別に用意する必要がない)ため，正確さの面で有利だが，複雑な記述はモデル化しづらい，また，自動定理証明器などのソルバの能力・性能に左右され，非線形式など複雑な式が解けないという問題がある．

従来，ゲートレベルや RTL の設計記述に対する形式的検証では，ビット幅がある一塊の(ワード)信号もビットレベルに展開し，ブール代数によって解く方法が確立されていた．しかし，システムレベル記述が一般化するにつれ，頻出するビット幅が大きい(例えば，int 型で 32 ビットなど)信号をビットレベルに展開することは，計算の効率上問題となってきている．

そのため，ワード信号を，ビットレベルに展開することなくワード信号のまま扱うワードレベルの検証手法が重要性を増してきているが，まだシステムレベル記述に対する検証技術は十分に整っていない．

近年，ワードレベルの形式的手法である記号シミュレーションが注目を浴びている．これは，従来の具体的な値を用いるシミュレーション(以後，具体シミュレーション)と異なり，演算の結果を記号値として伝播する．そのため，一つの記号値に対してシミュレーションすれば，ありうる全ての具体値についてシミュレーションしたことになる．ただし，具体値と異なり，記号値の演算結果は基本的には式となるため，演算とともに式の表現が大きくなってしまうこと，また，一般的には条件分岐における条件式の評価結果が定まらないため，then と else の両方の分岐に対して場合分けによる対処をしなければならず，パスの数だけシミュレーションを実行しなければならないという問題が残る．なお，記号による条件式の解決は，通常妥当性判定器の能力の制約を受ける．

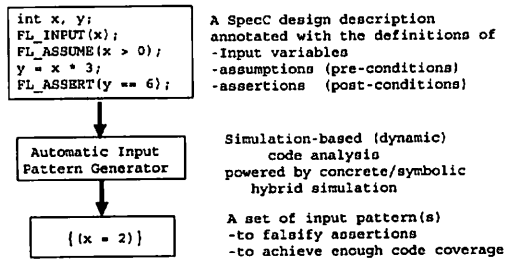

図 1 入力パターンの自動生成
Fig. 1 Automatic Input Pattern Generation.

さらに最近，ソフトウェアの単体テスト向けに，記号シミュレーションと具体シミュレーションを組み合わせて用いる手法が考案された[5]．この手法では，記号シミュレーションによって解けないような複雑な条件式は具体シミュレーションによって得られた具体値を用いて式を置き換えることで近似してしまう．また，具体シミュレーションによって得られたパスに対して記号シミュレーションを走らせ，そのパスを通るような入力値の条件式を求める．CUTE[11]では，さらに，その条件式の一部を変形して解くことで，現在通過したパスとは異なるパスを通るような入力値を求め，実現可能なパスを次々と列挙していくことができる．

本研究では，記号シミュレーションと具体シミュレーションを同時に行う記号・具体混合シミュレーションを実現するためのインタプリタを実装した．さらに，この混合シミュレーションフレームワークをシステムレベル記述を対象とした検証・デバッグに応用して，アサーション違反を起こすような入力パターンや，分岐カバレッジを改善するような入力パターンの集合を自動生成する手法を示す．

図 1 のように，本研究における自動入力パターン生成では，入力変数，事前条件(前提条件)，事後条件(アサーション条件)を定義された設計記述に対して，記号・具体混合シミュレーションを行い，得られた事後条件の記号式を形式的に評価することで，アサーション違反を起こす入力パターンや，分岐カバレッジなどのコードカバレッジを十分に達成するための入力パターンの集合を自動生成する．

以下，2. で混合シミュレーションのフレームワーク，3. で自動入力パターン生成について述べ，4. で予備的な実験結果を示し，5. でまとめる．

2. シミュレーションフレームワーク

本節では，実装した記号・具体混合シミュレーション環境の要素技術について述べる．SpecC によるシステムレベル設計記述のフロントエンド及び中間表現を用いるため，本ツールを FLEC(Fujita Laboratory Equivalence Checker) 上に構築した．ツールの全体像を図 2 に示す(斜線部分が今回実装した部分である)．

以下，説明のために，図 3 のような簡単なサンプルコードを

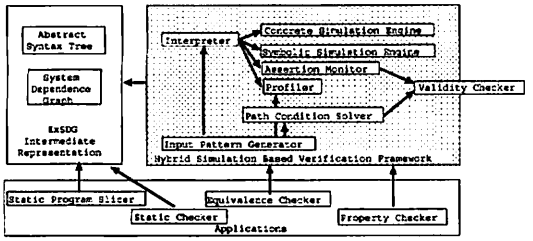

図 2 FLEC ツール全体図

Fig. 2 Overview of the FLEC tool.

```
1: int func(int x, int y) {
2:   int r = 0;
3:   if (x - y > 0) // B1-T
4:     r = x - y;
5:   else // B1-F
6:     r = y - x;
7:   return r;
8: }
```

```
1: int x, y, z;
2: FL_INPUT(x);
3: FL_INPUT(y);
4: FL_ASSUME(x >= 0);
5: FL_ASSUME(y >= 0);
6: z = func(x, y);
6: FL_ASSERT(z > 0);
```

図 3 サンプルコードとそのテストドライバ

Fig. 3 A sample code and its test driver.

用いる．本ツールは，以下の組込み関数を持ち，

- FL_INPUT(x) 入力変数 x
- FL_ASSUME(pred) 事前条件式 pred
- FL_ASSERT(pred) 事後条件式 pred

をそれぞれ本ツールに対して指示する．本ツールは，事前条件式を成立させ，かつ事後条件式を満たさないような入力変数の値の割り当てを探す．この例題では，変数 x と変数 y の差の絶対値を計算しており，入力変数 x，y の値が等しいときに，事後条件違反が発生する．

2.1 フロントエンドおよび中間表現

FLEC のフロントエンドは，SpecC による設計記述を，抽象構文木 (AST: Abstract Syntax Tree) に変換する．AST は，ソースコードをパースしたものであり，定数，変数，演算子，スコープ，コンパウンド，関数などをノードとした木構造を持つ．図 4 に，図 3 の AST を示す．

FLEC では，さらに AST に対して依存解析を行い，AST ノード間の静的なデータ依存関係，制御依存関係等を表現するシステム依存グラフ (SDG: System Dependence Graph) を構築する．この AST と SDG を統合した中間表現を，拡張システム依存グラフ (ExSDG: Extended System Dependence Graph) [10] と呼んでいる．

このため，FLEC では，静的な依存解析を用いるようなアプリケーションが比較的容易に実装できる．たとえば，FLEC には，ユーザに指定されたプログラム記述中のある変数に着目し，プログラム記述のどの部分から影響を受けうるか，もしくはその変数がプログラム記述のどの部分に影響を与えうるかを解析し，同等の結果を得られるサブセットの記述を抽出するプログラムスライシングが実装されている．

```
Function(4): func(args=2)
  Type(7): int
  Argument(5):
    Variable(6): x
      Type(7): int
  Argument(8):
    Variable(9): y
      Type(7): int
  Statement(10): compound
    Scope(11):
      Variable(34): r
        Type(7): int
    Statement(12): if else
      Expression(13): >
        Expression(15): -
          Expression(16): x
          Expression(17): y
        Expression(18): constant
          Constant(19): (int)0
```

```
Statement(20): expression
  Expression(21): =
    Expression(22): r
    Expression(23): -
      Expression(24): x
      Expression(25): y
Statement(26): expression
  Expression(27): =
    Expression(28): r
    Expression(29): -
      Expression(30): y
      Expression(31): x
Statement(32): return
  Expression(33): r
```

図 4 Fig.3 の抽象構文木

Fig. 4 Abstract syntax tree of Fig.3.

2.2 AST インタプリタ

AST インタプリタは，設計記述の AST をノードごとにトラバースし，具体シミュレーションエンジンもしくは記号シミュレーションエンジンに評価させることによって，インタプリタ的なプログラムの実行を実現する．これにより，具体シミュレーションと記号シミュレーションを統合的に実現できる．

a) 具体シミュレーション

AST ノードの式の値を具体値を用いて評価することで，従来のコンパイルと実行によるシミュレーションと同等の動作を実現できる．入力値として具体的な値 (例えば，int 型の変数 x に 3 を代入する等) を与えて実行するため，プログラムの実行状態は具体値によって表現される．

図 3 の例では，入力変数に具体値 (x=1, y=3) を与えて具体シミュレーションを行うと，3 行目の if 文の条件は false となり，r=2 となる (事後条件 $z > 0$ は成立する)．

ランダムシミュレーションでは，この入力変数の具体値を乱数により決定する．

b) インタプリタ方式のメリット

インタプリタ方式は，コンパイル方式に比べて一般に動作が遅いものの，検証・デバッグに便利な以下の機能を提供できる．

- ブレークポイント，ウォッチポイント
- 変数の値や値の変化のモニタ
- 値の伝播のトレース
- 分岐履歴のトレース
- プログラム状態の記録，再生
- 部分的な，あるいは不完全なコードの実行
- コードの一部を (条件を変化させながら) 繰り返し実行
- 対象のコードを動作中に動的に変更
- 並列性のスケジューリングなど，通常非決定的な要素になりがちなものの再現

なお，CUTE では CIL フレームワーク [9] によるインスツルメンテーションを行った後コンパイルする方式を採用している．

2.3 記号シミュレーション

記号シミュレーションでは，AST ノードの式の値を記号値を用いて評価する．具体シミュレーションと異なり，値は具体値

Path B1-T	*Path B1-F*
2: $(r_1 = 0)$	2: $(r_1 = 0)$
path-condition(B1-T):	path-condition(B1-F):
3: $(x - y > 0)$	3: $(x - y \leqq 0)$
4: $(r_2 = x - y)$	6: $(r_2 = y - x)$
pre-condition: $(x \geqq 0)$	pre-condition: $(x \geqq 0)$
pre-condition: $(y \geqq 0)$	pre-condition: $(y \geqq 0)$
post-condition: $(r_2 > 0)$	post-condition: $(r_2 > 0)$
Solve $\forall x, y(x \geqq 0)(y \geqq 0)$	Solve $\forall x, y(x \geqq 0)(y \geqq 0)$
$(x - y > 0)(r_1 = 0)$	$(x - y \leqq 0)(r_1 = 0)$
$(r_2 = x - y)$	$(r_2 = y - x)$
$\rightarrow (r_2 > 0)$.	$\rightarrow (r_2 > 0)$.

図 5　図 3 の記号シミュレーション結果
Fig. 5　Symbolic simulation results of Fig. 3.

ではなく記号的に取り扱うため，プログラムの実行状態は記号式である．

その後，結果として導出される事後条件の記号式が，ありうる全ての具体値に対して成立するかどうかを，妥当性判定器などのソルバを用いて判定する．

例えば，二つの記述 $x_1 = a; y_1 = b; z_1 = x_1 + y_1;$ と $x_2 = b; y_2 = a; z_2 = x_2 + y_2;$ が与えられたときに，+ 演算が可換 $(a + b = b + a)$ であるという知識があれば，$z_1 = z_2$ はいかなる具体値 a，b に対しても成立することが証明できる．

図 3 のコードのパス (分岐 B1-T を通るものと，分岐 B1-F を通るもの) それぞれに対して記号シミュレーションを行った結果を図 5 に示す．なお，複数回代入された変数の値を区別するために，静的単一代入形式 (Static Single Assignment Form [8]) という方法で変数の名前変更を行っている．記号シミュレーションによって生成された事後条件の記号式 (図 5 の最下行) を解き，もし式を成立させない入力変数 $(x,\ y)$ の割り当てが存在すれば，それが事後条件違反を引き起こす入力値となる．

c)　パス条件

記号シミュレーションによる実行では，一般には分岐条件の評価は静的に行えないため (入力の具体値によって成立する場合もあれば成立しない場合もある)，分岐では then と else とで場合分けを行い，それぞれの場合について分岐条件を憶えておく．パス全体について分岐条件を集めたものがパス条件である．図 5 の左側のパスの場合，パス条件は $(x - y > 0)$ である．パスによっては，いかなる入力値に対してもパス条件が成立しないことがあり，これはフォールスパス (false-path) と呼ばれている．パスがフォールスパスであるかどうかの判定にも，妥当性判定器を用いることができる．

d)　記号式の評価

記号シミュレーションによって得られた事後条件の記号式が，ありうる全ての値に対して成立するかどうかの判定や，収集したパス条件を満たすような具体値の発見には，自動定理証明器の一種である妥当性判定器 (Validity Checker) などのソルバを用いる．前者は妥当性判定問題，後者は充足可能性判定問題となる．

本ツールでは，妥当性判定器の実装である CVC [12] を用いて，妥当性や充足可能性を判定し，例外や充足解を具体値として得ている．

```
1: void test(int x, int y, int z) {
2:   if (x > 3) // B1
3:     if (y > 11) // B2
4:       if (z == y * y) // B3
5:         if (x < 5) // B4
6:           reach_me();
7: }
```

図 6　Concolic シミュレーションの例
Fig. 6　Example for Concolic simulation.

記号シミュレーションでは分岐条件の評価ができないため，ループなどがある場合，反復回数がわからない場合がある．この場合は，Bounded Model Checking [2] という手法を用いて，反復回数を有限回数に限定し，ループ展開を行う．ただし，展開回数を越えるような状況については正しく検証することができない．

等価検証用に特化した，等価クラスによる記号シミュレーションというものもあり，松本氏による実装 [6] [7] がある．

2.4　記号・具体混合シミュレーション

Larson らによって，記号シミュレーションと具体シミュレーションを組合わせるという手法が提案された [5]．この手法では，ユーザが与える具体値を元にシミュレーションを行いつつ，そのパスについて記号シミュレーションを行い，パス条件を求め，同一パスを通るような他の入力値について，何らかのアサーション違反が起きるかどうかを調べている．

一方，CUTE では，パス条件を一部変更して解くことで，別のパスを通るような入力パターンを生成している．図 6 のコードに対して，*reach_me*() の行に辿りつくための入力 (x, y, z) の値を求めたいとする．CUTE では，図 7 のように，

- まず乱数によって入力値を決め，具体シミュレーションを実行する
- 具体シミュレーションで実行されたパスについて，記号シミュレーションを実行し，そのパスを通るための条件式を構築する
- 分岐が目的のものから外れたら，その分岐の条件を否定した条件式を作成する
- その条件式を解き，次の入力値を決める

といったことを，目的のパスに辿りつくまで繰り返すことにより，パスのガイドを実現する．なお，CUTE は，単体テストを目的とするため，パスの探索順序は特に考慮せず，可能なパスを全て試そうとする．

また，妥当性判定器に解けないような非線形の式に遭遇した場合，非線形式を具体シミュレーションでの具体値に置換することによって，式を線形に保ち計算を続行する．近似によって式が不正確になるため，条件式を解いた値を入力として与えても，目的のパスに辿りつけるとは限らない．

3.　自動入力パターン生成

本節では，本ツールのフレームワークの応用として，記号・具

iteration	input x	y	z	path B1	B2	B3	B4	derived current path-condition	next path-condition to solve
1	0	0	0	F	-	-	-	$(x \leqq 3)$	$(x > 3)$
2	10	0	0	T	F	-	-	$(x > 3)(y \leqq 11)$	$(x > 3)(y > 11)$
3	10	20	0	T	T	F	-	$(x > 3)(y > 11)(z \neq y * y)$	$(x > 3)(y > 11)(z = 400)$
4	10	20	400	T	T	T	F	$(x > 3)(y > 11)(z = 400)(x \geqq 5)$	$(x > 3)(x < 5)(y > 11)(z = 400)$
5	4	20	400	T	T	T	T	$(x > 3)(x < 5)(y > 11)(z = 400)$	(target reached)

On 3rd iteration, non-linear formula $y * y$ has been replaced with the current concrete value 400.

図 7 図 6 の Concolic パスガイドの例

Fig. 7 Example of Concolic path-guidance of Fig.6.

体混合シミュレーションを用いたアサーションベース検証および分岐カバレッジの改善について説明する．アサーションベース検証では，事後条件違反を引き起こすような入力パターンを，分岐カバレッジの改善では，記述中に含まれる条件分岐のなるべく多くを最低一度は通るような入力パターンの集合を自動生成する．

3.1 アサーションベース検証

定義された入力変数 x に対し，与えられた事前条件 *assume_pred* を満たし，かつ事後条件 *assert_pred* を満たさないような値の割り当てを発見する．混合シミュレーションでは，入力変数に初期値 (ランダムな具体値) を割り当てて混合シミュレーションさせ，具体値によって条件分岐の評価をしてパスの選択を行い，パス条件 (*path_cond*) を蓄積し，同時に式の値を記号式でも評価する．最後に，記号式による事後条件で妥当性検査器を用いて評価し，例外が発見されたら，それがアサーション違反を起こす入力値である．具体的には，$assume\_pred(x) \wedge path\_cond(x) \rightarrow assert\_pred(x) = 0$ となるような x を探す．

混合シミュレーションによるアサーションベース検証では，具体値によって決まる特定パスにおいて，準形式的検証を行っている．純粋な形式的手法と異なり，完全ではないものの，形式的手法が使えないような複雑すぎる記述への対応ができること，また，静的な条件分岐の場合分け (if 文の場合分けやループの展開等) によるパス爆発を避けることができるという特徴がある．

3.2 コードカバレッジ改善

コードカバレッジとは，(異なる入力パターンによる) 一回以上のプログラムの実行によって，設計記述のうち，最低一回実行された部分が全体でどの程度あるか (逆に言えば，全く実行されなかった部分がどの程度存在するか) を計測したものである．ここでは，特に分岐カバレッジを扱う．

分岐カバレッジとは，プログラムの一回以上の実行において，記述中に含まれる全ての条件分岐の枝の数 (if 文，for 文，while 文，do-while 文なら 2 つ，switch 文なら case 文の数) のうち，分岐条件の評価によって一回以上選択された枝の数の割合を示したものである．

これは，作成したテストケースの入力パターンの集合の品質の目安となる．適切な入力パターンを与えることで到達可能なコードはカバーされるべきであり，テストによって一度も実行されていないコードが存在するとすれば，それはテストが不十分であるか，あるいは到達不可能なコードが存在することを意味する．

本研究では，混合シミュレーションによって，まず一回プログラムを実行し，そのパス条件を一部変更して解くことで，別の分岐を辿るような入力パターンを計算する．なるべく異なる分岐を通るようにパスを誘導することで，分岐カバレッジが高くなるような入力パターンの集合を作成する．また，上記のアサーションベース検証と組合せることで，アサーション違反のケースを見つけやすくなることも期待できる．

本ツールでは，以下のようにして新たなパス条件の導出を行う．条件分岐を n 個通過したあるプログラムの実行において，条件分岐 k における条件式が BC_k であるとすると，そのパス条件 PC は $PC = BC_1 \wedge BC_2 \wedge \cdots \wedge BC_k \wedge \cdots \wedge BC_n$ となる．ある条件分岐 k において，(まだカバーされていない) その分岐以外の分岐を通らせるためには，パス条件 PC' を $PC' = BC_1 \wedge BC_2 \wedge \cdots \wedge \neg BC_k$ として解き (k 番目以降のパスが異なるため，k より後の条件は不要になることに注意)，入力パターンの具体値を求める．ただし，このパス条件は解をもたない (フォールスパスになる) こともある．その場合は，節を削っていくことで条件式を緩和すればよいが，極端な場合は，ヒントとなるパス条件が消失してしまい，解空間が広くなりすぎて，ランダム探索とかわらなくなってしまう．新たなパス条件の導出によって次の入力パターンの生成し，分岐カバレッジが十分な値に達するまでこれを繰り返す．

なお，CUTE では同様のアルゴリズムを用いているが，CUTE は分岐カバレッジを見ずに，パスを全て列挙しようとするため，同じ分岐カバレッジを達成するまでのシミュレーション回数が増えてしまう傾向にある．

4. 実験結果

ここでは，本ツールを用いた予備的な実験結果について示す．図 3 の記述のほか，図 8，図 9(階乗の計算．関数の再帰呼出しやループを含む) の記述を用いて動作を確認した．

4.1 アサーションベース検証

これらの簡単な例題の範囲で，動作していることを確認した．

図 3 において，アサーション違反となる入力値 $(x = 0), (y = 0)$ を発見した．

図 8 において，アサーション違反となる入力値 $(x = 1), (y =$

```
1: behavior Test() {
2:   void f(int x, int y, int z) {
3:     if(x+y+z==6)
4:       if(2*x+7*y+3*z==25)
5:         if(-4*x-2*y+2*z==-2){
6:           FL_ASSERT(0);
7:         }
8:   }
```

```
9:   void main() {
10:    int x, y, z;
11:    FL_INPUT(x);
12:    FL_INPUT(y);
13:    FL_INPUT(z);
14:    f(x, y, z);
15:  }
16: };
```

図 8 実験例題 (1)

Fig. 8 An experimental example (1).

```
1: behavior Fact() {
2:   int fact_rec(int s) {
3:     if (s <= 1) {
4:       return s;
5:     } else {
6:       int t;
7:       int p;
8:       t = s * fact_rec(s - 1);
9:       return t;
10:    }
11:  }
12:  int fact_for(int s) {
13:    int i;
14:    int p;
15:    p = 1;
16:    for (i = 1; i <= s; i++) {
```

```
17:      p *= i;
18:    }
19:    return p;
20:  }
21:  void main() {
22:    int i, o1, o2;
23:    FL_INPUT(i);
24:    FL_ASSUME(i <= 10);
25:    o1 = fact_for(i);
26:    o2 = fact_rec(i);
27:    FL_ASSERT(o1 == o2);
28:  }
29: };
```

図 9 実験例題 (2)

Fig. 9 An experimental example (2).

2), ($z = 3$) を発見した.

図 9 において, 変数 i の値が 10 までに制限された中, 具体値 $i = 8$ について検証され, アサーション違反がないことが確認された.

ただし, 混合シミュレーションにおいて, 非線形記号式の具体値への置換がまだ実装されていないため, (これらの例題の中にはないが) 非線形の式が存在する場合, ソルバがうまく動かずに計算が進まなくなってしまう.

4.2 分岐カバレッジ

これらの簡単な例題の範囲で, 動作していることを確認した.

図 3 において, 2 回のシミュレーションで, 分岐カバレッジ 100%を達成した. 図 8 において, 4 回のシミュレーションで, 分岐カバレッジ 100%を達成した. 図 9 において, 1 回のシミュレーションで, 分岐カバレッジ 100%を達成した.

5. まとめと今後の課題

5.1 結　　論

本研究では, 記号・具体混合シミュレーション環境を構築し, それを使ったアサーションベース検証および分岐カバレッジを改善するための自動入力パターン生成手法を実装し, 簡単な例題によってその動作を確認した.

5.2 今後の課題

現在, ブランチカバレッジを改善しようとするものの, ユーザが指定する特定のステートメントのみを優先的に探索するような仕組みはまだ実装されていない.

今のところ, 本ツールのシミュレータは逐次記述にしか対応していないが, ハードウェアのシステムレベル記述では, 並列動作が記述されることがしばしばあり, それに対応する必要がある.

大規模記述に対応するためには, 記号シミュレーションにおいて, 記号式を効率的に管理する必要がある. たとえば, マルチプレクサを用いて複数パス間の共通部分式を共有する方法として, Maximally shared circuit graph [4] がある.

文　　献

[1] Systemc:. http://www.systemc.org/.

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[3] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[4] Alfred Koelbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *Int. J. Parallel Program.*, 33(6):645–666, 2005.

[5] Eric Larson and Todd Austin. High coverage detection of input-related security faculty. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[6] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. An equivalence checking method for c descriptions based on symbolic simulation with textual differences. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E88-A(12):3315–3323, 2005.

[7] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. Equivalence checking of c programs by locally performing symbolic simulation on dependence graphs. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 370–375, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[9] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Computational Complexity*, pages 213–228, 2002.

[10] T. Nishihara, D. Ando, T. Matsumoto, and M. Fujita. Exsdg : Unified dependence graph representation of hardware design from system level down to rtl for formal analysis and verification. In *the International Workshop of Logic and Synthesis*, pages 83–90, May 2007.

[11] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proc. of ESEC/SIGSOFT FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM Press.

[12] A. Stump, C. Barrett, and D. Dill. Cvc: a cooperating validity checker. In *14th International Conference on Computer-Aided Verification*, 2002.