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

2006－SLDM－124 (3)
2006／3／16

SpecC言語の依存グラフを利用したプログラムチェッカ

佐々木俊介† 西原 佑† 安藤 大介†† 藤田 昌宏†††

† 東京大学大学院工学系研究科電子工学専攻
文京区本郷 3-7-8
†† 東京大学工学部電子工学科
文京区本郷 3-7-8
††† 東京大学大規模集積システム設計教育研究センター
文京区弥生 2-11-16
E-mail: †{shun,tasuku}@cad.t.u-tokyo.ac.jp, ††ando@cad.t.u-tokyo.ac.jp , †††fujita@ee.t.u-tokyo.ac.jp

あらまし SoCのシステムレベル設計において, バグを早期に発見し修正することは手戻りを減らすために重要である. そのための支援手法として提案されたSpecCを対象としたシステム依存グラフを利用したプログラムチェッカについて, ヌルポインタのチェックを行う手法と, 条件式の解釈を行うことによって未初期化変数の参照の検出とヌルポインタのチェックの精度を向上する手法を提案し, 実験結果を示す.

キーワード SpecC, システム依存グラフ, プログラムチェッカ

SpecC Program Checker Using System Dependence Graph

Shunsuke SASAKI†, Tasuku NISHIHARA†, Daisuke ANDO††, and Masahiro FUJITA†††

† Department of Electronics Engineering, Graduate School of Engineering, University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo, 113-8656 Japan
†† Department of Electronics Engineering, Undergraduate School of Engineering, University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo, 113-8656 Japan
††† VLSI Design & Education Center, University of Tokyo
Yayoi 2-11-16, Bunkyo-ku, Tokyo, 113-0032 Japan
E-mail: †{shun,tasuku}@cad.t.u-tokyo.ac.jp, ††ando@cad.t.u-tokyo.ac.jp , †††fujita@ee.t.u-tokyo.ac.jp

Abstract In system-level design of SoC, it is important to find and fix bugs for reduction of backtracking of design flow. In this paper, we propose program checking methods for SpecC descriptions using System Dependence Graphs (SDGs) . First, we propose a checking method of nil-pointer references. Second, we propose a method to improve accuracy of checking results of uninitialized variables and nil-pointer references with interpreting conditional expressions.

Key words SpecC, System Dependence Graph, Program Checker

1. 研究の背景

集積回路は 18 ヶ月に 2 倍の割合で年々高集積化し, 回路規模が年々増大している. それに伴い, 回路のデバッグはますます困難になり, 検証にかかる時間も長くなっており, 設計期間の70~80% を検証の時間が占めることも多いと言われている. [1]

また, 高集積化と, 製品の市場投入までの要求される期間が短くなってきていることにより, 設計資産の再利用の重要性も次第に高くなっている.

このような問題を解決するため, システムレベル設計が提案され, 使用されるようになりつつある. システムレベル設計では, 設計記述に SpecC [2] や SystemC [3] などの C 言語ベースのシステムレベル設計言語を用い, ハードウェアとソフトウェアを含むシステム全体を同時協調的に設計することで, 既存のソフトウェアとして実装された設計資産の再利用や, 設計のトレードオフの検討などをより効率的に行えると共に, 従来ソフトウェアの開発で使用されてきた設計生産性向上のための支援手法などが利用可能になる. その例として, 設計の中からバグの疑いのある箇所を検出する Lint 系ツール [4] や, 設計記述上の依存関係を解析するプログラムスライシング [5], [6] などが

ある.

筆者らは, システムレベル設計の設計フロー中でシステム依存グラフ (SDG) [7] を用いてプログラムのチェックを行いバグを早期に発見することで, バグによる手戻りを減らす手法を提案するとともに, そこで使用する依存グラフを用いた未使用箇所と未初期化変数参照の検出手法を提案した.

本稿では, ヌルポインタ参照の検出手法についても提案すると共に, SDG のノード中の条件式を解釈する事によって, これらの手法による誤検出を減らし, 精度を向上する手法を提案する.

2. 関連研究

2.1 システム依存グラフ

プログラムのチェックや並列化には依存解析が重要になる. そのために利用できる解析手法として, システム依存グラフ (System Dependence Graph : SDG) を使用する手法がある.

SDG は, 主にプログラムスライシング手法で利用されてきたデータ構造で, プログラム記述中のステートメントをノード, 依存関係をエッジとして, 静的に解析した依存関係をグラフ構造にした物である.

C 言語の SDG は, 各関数毎に生成される, 関数内の依存関係を表す Procedure Dependence Graph (PDG) と, その間を繋ぐ関数呼び出しを表現するためのエッジとノードから成る.

PDG には, データ依存エッジと制御依存エッジがあり, コントロールフローグラフ (CFG) とその各ノード上での変数の使用/代入の情報を用いて, 以下のようにして生成される.

- データ依存

文 s2 が文 s1 にデータ依存しているのは,

(1) 文 s1 において変数 v に値を代入し,

(2) 文 s2 において v を使用していて,

(3) s1 から s2 に至る実行可能なパスが存在していて, かつその過程で v に値が再代入されていない場合である.

- 制御依存

文 s2 が文 s1 に制御依存しているのは,

(1) 文 s1 がプログラムの開始点あるいは, 条件分岐文 (if 文など) かループ文 (while 文など) で,

(2) 文 s2 が実行されるかどうかが s1 の実行結果に直接依存する場合である.

また, Java や C++等のオブジェクト指向言語を対象とした拡張 [8]~[10] も提案されており, 後に述べる SpecC 言語の SDG もこれを基にしている.

この SDG 上での到達可能性を判定することによって, ある行に影響を与える行を抽出するバックワードスライシングや, ある行から影響を与える箇所を抽出するフォワードスライシングなどが可能である.

また, 本研究で使用した, システムレベル記述言語の一つである SpecC 言語 [2] についても, 依存グラフの生成とスライシングの手法が提案されており [11], ANSI-C に対し SpecC で拡張が成されている, 構造の階層化, 並列実行性, タイミング, 同期, 通信, 状態遷移などの記述について, オブジェクト指向言語を対象とした手法をもとに, SDG の表現方法を定義している. 本研究ではその手法に基づいて生成した SDG を利用した.

```
1  behavior Bhvr1(
2    int x,int y){
3    void main(void){
4      int a;
5      a=x+1;
6      y=a;
7    }
8  };
9
10 behavior Bhvr2(
11   int y,int z){
12   void main(void){
13     int a;
14     a=y*2;
15     z=a;
16   }
17 };
18
```

```
19 behavior Main{
20   int x,y,z;
21   Bhvr1 b1(x,y);
22   Bhvr2 b2(y,z);
23
24   void main(void){
25     x=1;
26     par{
27       b1.main();
28       b2.main();
29     }
30   }
31 };
32
33
34
35
36
```

図 1 An example source code : Use of uninitialized variables.

2.2 既存のプログラムチェッカ

C 言語などのプログラミング言語を対象としたプログラムチェッカは幾つかあり, コンパイラによる警告や, 以下で説明する Lint, UNO など, 色々なツールがあり, 主に構文, Syntax Tree , Control Flow Graph , Call Graph による解析に基づいている.

しかし, 現状では

- 並列動作などを含むシステムレベル記述を対象としたプログラムチェッカは存在しない
- C++に変換した SpecC や, C++にライブラリを追加することで実現している SystemC に対して C++用のプログラムチェッカを適用するのは, 階層構造や並列処理, 同期等のハードウェア特有の概念に対して対応しきれない
- C/C++用のプログラムチェッカでも, 誤検出が多かったり, 検出しきれない場合が有るなど, 十分ではない
- CFG を使用する物では, 長いパスを辿るとチェックに掛かる時間が長くなるため, グローバルなチェックを避け, ローカルなチェックにとどまる傾向にある.

といった問題が有った.

2.3 依存グラフを用いたプログラムチェッカ

2.2 節の問題に対し, 筆者らは SpecC 記述の SDG を用いた, 未使用箇所と未初期化変数の参照をチェックするプログラムチェッカを提案した. [12], [13] このうち, 未初期化変数の参照の検出手法について説明する.

並列実行を含むことの多いシステムレベル記述では, 並列実行している 2 つのプロセス間で, 一方で代入を, 一方で参照を行うという事が発生する. この場合に, 代入が参照よりも後に行われると, 初期化前に変数が参照されることになり, 正しく動作しない. また, 依存関係が有って並列に実行できない箇所を並列に実行しようとした場合にも同様の現象が起こり得る. この手法は, そのようなバグを検出する事を目的としている.

図 1 と図 2 に検出可能な例を示す. この例では, 依存関係のある 2 つのビヘイビア Bhvr1 と Bhvr2 を並列に実行しているため, 実行順によっては未初期化の変数が発生する.

この手法では, データ依存エッジを辿ることによって初期化を行っている可能性のある箇所を抽出し, その箇所についてコ

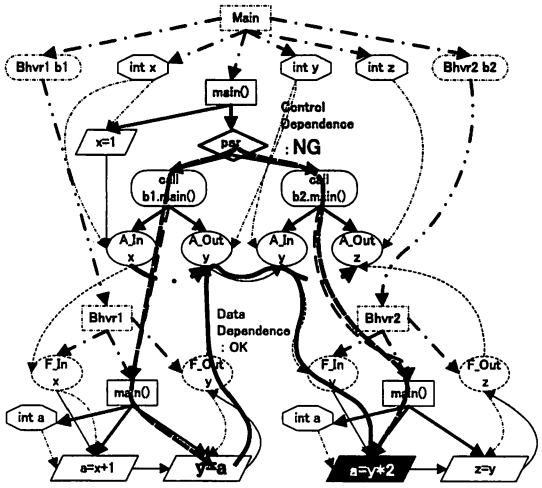

図 2 SDG of example source code : Use of uninitialized variables.

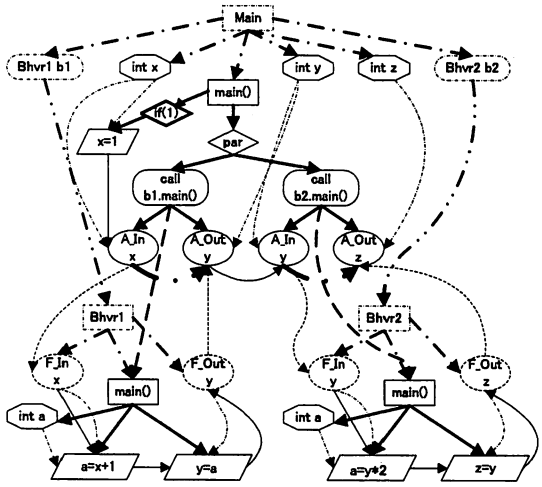

図 3 SDG of example source code : Use of uninitialized variables (False Warning).

ントロール依存エッジを辿ることによって確実に先に実行される事が保証されるかどうかを確認するという 2 段階で未初期化の可能性が有るかどうかを判定する.

以下, この手法を用いて図 1 の 12 行目にある "a=y*2" (図 2 の右下) が未初期化になる可能性の有る箇所であると判定される手順を示す.

(1) 変数 y について, データ依存エッジを辿る. "F_In y", "A_In y" を経由して, さらに "A_Out y", "F_Out y" を経由して, Bhvr1 の "y=a" に到達する. 従って, "y=a" で初期化を行っている可能性があることが判る.

(2) Bhvr1 の "y=a" が Bhvr2 の "a=y*2" よりも先に必ず実行されている事を確認する.

- "y=a" と "a=y*2" の両方からコントロールエッジとコールエッジのみを経由して逆方向に辿った場合に, 到達可能な共通の祖先ノードは, Main ビヘイビアの "par" である.
- 初期化を行っている可能性のある "y=a" に対するこのノードからの経路上 (このノード自身も含む) には, この "par" ノードが存在する.
- よって, 正しく初期化されない可能性がある事が判る. 従って, このノード "a=y*2" について警告を出す.

以上のように, この例題では正しく判定可能である.

この手法では, SDG を使用することで, CFG のみの場合と比較し, 辿るべきノード数が少なくて済む. 例えば, ある箇所で代入が行われた後, 1000 ステップ後にその変数が初めて使用されるという場合, CFG ではその 1000 ステップ分のノードを辿る必要があるが, SDG ではそれらのノード間にエッジが直接引かれるため, エッジを 1 段階辿るだけで良い. そのため, CFG のみでは計算量が掛かるため困難であったグローバルなチェックも短い時間で行うことができるという点がこの手法の特長である.

しかしながら, この手法では誤警告が多い. 図 3 に示す例のように, 初期化を行っている側が常に実行されるような if 節の中に有る場合や, if - else 等による分岐の全ての場合で初期化が行われているような場合には, 初期化が行われているにもかかわらず警告が出るという問題が有る.

3. 提案手法

本節では, まず 3.1 節で従来の SDG を用いたプログラムチェッカと同様に条件式を解釈しない範囲でのヌルポインタ参照の検出手法を提案し, 次に, 3.2 節では, 条件式を解釈することで到達可能性の判定を行い, 未初期化変数の参照とヌルポインタ参照の検出について誤検出を減らし精度を向上する手法を提案する.

3.1 ヌルポインタ参照の検出

ポインタ変数を使用する際, そのポインタが何も指していないヌルポインタであるにもかかわらず使用してしまうというバグがある.

通常, ポインタを使用する際は, 変数のアドレスをポインタ変数に代入することで初期化を行ってから使用するが, 条件分岐や並列実行などによってその初期化を行わないうちに使用してしまう事があり, バグの原因となる.

この検出のための手法として, 以下のアルゴリズムを提案する.

(1) ポインタ変数は定義された時点で明示的に NULL を代入するように設計者が記述する (代入されていない場合は未初期化変数のチェックで発見される)

(2) SDG を生成する

(3) ポインタの参照を行っている箇所それぞれについて, 以下のチェックを行う

(a) データ依存エッジで接続されている親ノードについて, 注目しているポインタ変数に NULL を代入しているかどうかを調べる

(b) もし NULL を代入している場合, ヌルポインタ参照の可能性があるので警告を出す

```
1 void main(void){
2   int b,c;
3   int *a=NULL;
4   b=1;
5   if(cond){
6     a=&b;
7   }
8   c=*a;
9 }
```

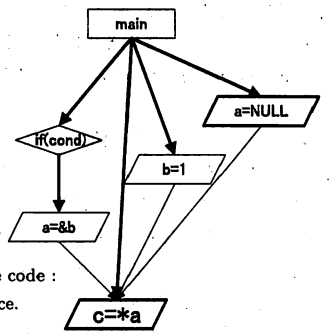

図 4 An example source code : Nil-pointer reference.

図 5 SDG of Example source code : Nil-pointer reference.

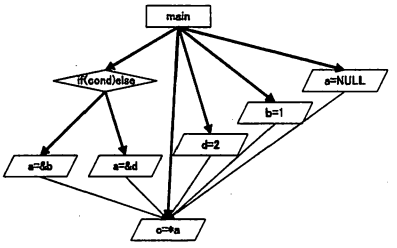

図 6 SDG of Example source code : Nil-pointer reference(2).

図 4・図 5 にヌルポインタ参照の検出が可能な例を示す. この例では, if 節の中でポインタ変数 a の初期化が行われているため, if 節の中が実行されない場合には初期化が行われず, ヌルポインタ参照の可能性がある. NULL を代入しているノードからポインタを使用している "c=*a" にデータ依存エッジがあることから, ヌルポインタ代入の可能性があると判定できる.

しかしながら, この手法にも 2.3 節の未初期化変数の参照の検出と同様の問題が有る.

例えば, 図 6 のように, 必ず実行される条件になっている if 文が有る場合や, if-else 文による分岐の両方で初期化が行われていて必ず初期化されるような場合には, 誤警告が発生する.

3.2 条件式の解釈による精度の向上

3.2.1 到達可能性の判定

本節では, SDG 上のノードに記述された条件式の解釈を行うことで, 記述中の特定の箇所に到達する可能性の有無を判定する手法を提案する. この手法は, 3.2.2 節及び 3.2.3 節の手法で利用可能である.

まず, 一つの関数内で関数の Entry Node から選択された検証対象のノード V に到達可能かどうかを判定する式を得るためのアルゴリズム LocalReachability() を, 図 7 のように定義する.

つまり, 対象とするノードから, SDG のコントロール依存エッジを EntryNode に辿り着くまで逆方向に辿り, if 文の else 側の場合は否定を, それ以外の場合は式をそのまま, 条件式に追加していくことで, 判定するための式を得ることができる.

そして, プログラム開始箇所からの到達可能性を判定するには, このアルゴリズム (LocalReachability(V)) を利用して, 図

```
expression LocalReachablility(node V){
  expression ex = "true";
  bool reach_end = false;

  while(!reach_end){  /*  ControlDependenceEdge を辿る  */
    switch(getNodeType(V=ParentViaControlEdge(V))){
      /* ノードの種類で場合分け */
      case ENTRY: // entry ノードなら終了する
        reach_end=true;break;
      case IF: // if 文の then 側なら
      case WHILE: // while 文なら
      case FOR: // for 文なら
        /* そのまま条件式に追加 */
        ex = (ex && getExpr(V)); break;
      case ELSE: // if 文の else 側なら
        /* 否定を取って条件式に追加 */
        ex = (ex && !getExpr(V)); break;
      case PAR: // par なら
        /* 条件を追加しない */
        ex = ex; break;
    }
  }
  return ex;
}
```

図 7 Pseudocode of local reachability checking algorithm.

```
expression GlobalReachablility(node V){
  expression ex;
  bool reach_end = false;

  ex = true;
  while(!reach_end){
    switch(getNodeType(V)){
      /* ノードの種類で場合分け */
      case MAIN_CALL_SITE:
        // main 関数の call-site ノードなら終了する
        reach_end=true;break;
      default:
        ex = (ex && LocalReachability(V)); break;
    }
    V=GetCaller(GetEntry(V));
  }
  return ex;
}
```

図 8 Pseudocode of global reachability checking algorithm.

8 に示すアルゴリズムを用いることで求められる.

すなわち, 対象とするノードから, main 関数を呼び出している call-site ノードに辿り着くまでコントロールエッジを遡り, その際, 対象とするノードと各 Call-Site ノードについて, それぞれ LocalReachability() を求め, それらを条件式に追加していくことで求められる.

このようにして得られた条件式を, 論理妥当性判定ツールで判定することで, 到達可能かどうかの判定が可能である. ここでは CVC [14] を利用した.

ただし, ノードに記述された式の解釈はコントロールノードについてのみ行っているため, 条件分岐の途中で条件式に使用している変数への代入が発生する場合などには, 正しく判定できない. そのような場合には, 代入が行われるたびに新しい変数名を与えるように前処理を施すことで対処できる.

具体例を示す. 図 9・図 10 に例題のソースコードと SDG を示す. 13 行目の "d=4" に到達する条件は, SDG を用いて得られる "b==2","!(a!=1)","a!=1" から, " (b==2) && (!(a!=1)) && (a!=1) " となる.

これを論理妥当性判定ツール CVC で判定するために, 図 11 に示す CVC の入力記述を生成する. CVC の QUERY 文では, 与えた条件式が変数の値に関わらず恒等的に真で有る場合に Valid を, 反例が有れば InValid を返す. そこで, 判定したい条

```
1  behavior Main{
2    void main(void){
3      int a,b,c,d;
4      a=1;
5      b=2;
6      if(a!=1){
7        b=1;
8      }
9      else{
10       c=3;
11       if(b==2){
12         if(a!=1){
13           d=4;
14         }
15       }
16     }
17   }
18 };
```

図 9 Example code of Reachability Analysis.

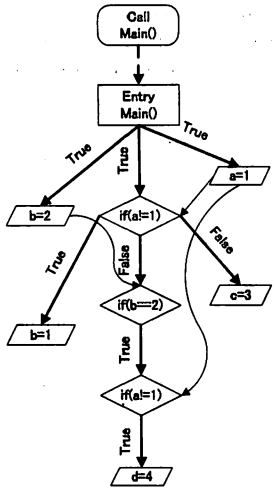

図 10 Example SDG of Reachability Analysis.

```
1  a,b:INT;
2  QUERY NOT((NOT(a/=1))AND((a/=1))AND((b=2)));
```

図 11 Example CVC code of Reachability Analysis.

件式の否定を与えることで, InValid が返れば条件を満たす解が存在する, つまり到達可能であり, Valid が返れば条件を満たす解は存在しない, すなわち到達不可能であると判定される.

この例では, Valid, すなわち, そのような条件を満たす各変数は存在しないという結果が得られ, 従って, 13 行目には到達可能ではないと判定される.

3.2.2 未初期化変数の参照の検出

2.3 節の手法に於いて, コントロール依存エッジを利用した因果関係の判定の代わりに, 3.2.1 を利用したより精度の高い藩邸手法を提案する.

アルゴリズムは以下の通りである. なお, **Reachability**(*V*) や **LocalReachability**(*V*) は 3.2.1 節の手法を用いて求められるとする.

- *V1* と *V2* が同じ関数内にある場合

CFG 上で *V1* から *V2* に到達可能かつ (**LocalReachability**(*V2*) → **LocalReachability**(*V1*)) として求められる. これを AlwaysExecute(V1,V2) とする.

- *V1* と *V2* が同じ関数内に無い場合

(1) SDG 上で *V1* と *V2* の共通の祖先ノード Va を求め, *V1* から Va に SDG を遡る際に最後に経由する call-site を V1cs, 同様に *V2* から Va に SDG を遡る際に最後に経由する call-site を V2cs とする. (そのような call-site が無い場合はそれぞれ *V1*, *V2* を V1cs,V2cs とする)

(2) Va が実行された場合に *V1*, *V2* がそれぞれ実行される条件 Cond1a,Cond2a を以下のように求める

- Cond1a=(**Reachability**(*Va*) → **Reachability**(*V1*))
- Cond2a=(**Reachability**(*Va*) → **Reachability**(*V2*))

(3) 求める条件は, AlwaysExecute(V1cs,V2cs) && (Cond2a → Cond1a) で得られる

(つまり, 呼び出し側の関数内で必要な順序で実行されていて, かつ, 呼びだされている側についても *V2* が実行されていれば *V1* が実行されている, ということを判定している)

具体例を用いて示す. 図 9 で, 「13 行目の "d=4" が実行される場合にはその前に 10 行目の "c=3" が必ず実行されている」かどうかを判定する.

上記の手法に従うと, "(((!(a!=1))&&(b==2)&&(a!=1)) → !(a!=1)" という式が得られ, これを論理妥当性判定ツール CVC で判定することで, "c=3" が実行されている事が保証されると判定できる.

3.2.3 ヌルポインタ参照の検出

3.1 節で提案したヌルポインタ参照の検出手法に, 初期化の箇所が実行される条件の解析と, 参照箇所の到達可能性の判定を追加することで誤警告を減らす手法を以下のように提案する.

(1) ポインタ変数は定義された時点で明示的に NULL を代入する

(2) SDG を生成する

(3) ポインタの参照を行っている箇所それぞれについて, 以下のチェックを行う

(a) Reachability() を用いて, その箇所に到達する可能性があるか判定する.

- 到達する可能性が無い場合, 警告の対象とせず, 次の箇所を調べる.
- 到達する可能性が有る場合, 以降のチェックを行う.

(b) データ依存エッジで接続されている親ノードについて, 注目しているポインタ変数に NULL を代入しているかどうかを調べる

- もし NULL を代入している箇所が無い場合, 警告の対象とせず, 次の箇所を調べる.
- もし NULL を代入している箇所が有るならば, さらに以降のチェックを行う.

(c) データ依存エッジで接続されている親ノードについて, 注目しているポインタ変数に NULL 以外の値を代入しているかどうか調べる

- もし代入している箇所があった場合, その行が実行される条件を Reachability() によって求め, 初期化の条件式に加える
- 代入を行っている全ての箇所について同様に繰り返す.

(d) 求められた初期化の条件式を全て OR で繋いだものについて, CVC を用いて常に真であるかどうかチェックする.

- 真であれば, 確実に初期化されていると判定されるので, 次の箇所を調べる.
- 偽であれば, ヌルポインタ参照の可能性があるので, 警告を出す.

この手法では, 3.1 節で誤警告が発生する例として示した図 6 のように if 文と else 文の両方で初期化を行っている場合でも, 初期化される条件は "cond || !cond" となり常に真であるので警告の対象と成らず, 正しく判定される.

表 1 Infomation of SpecC Testcase Codes.

Name	Brief Exmplanation	Lines	# of behaviors	# of nodes in SDG	time to generate SDG
IDCT_ROW	Part of Behavioral description of Inverse Discrete Cosine Transformation	135	2	389	1.685 sec
DWT	Behavioral description of Discrete Wavelet Transformation	202	3	1474	3.312 sec

表 2 Experimental Result of Program Checkers.

Type of Check	node interpretation	Testcase	Warnings	Real Errors	False Warnings	Miss	Time (sec)	# of CVC callings
Uninitialized	No	IDCT	48	2	46	0	0.082	-
		DWT	28	1	27	0	0.228	-
	Yes	IDCT	3	2	1	0	1.341	77
		DWT	11	1	10	0	1.119	56
Nil-pointer	No	IDCT	3	2	1	0	0.067	-
		DWT	2	1	1	0	0.169	-
	Yes	IDCT	2	2	0	0	0.103	2
		DWT	1	1	0	0	0.207	2

4. 実験結果

例題として表 1 に示すような SpecC 記述を用意し, これらの例題に, それぞれのチェックに応じたバグを複数, 手作業で混入したものを使用して評価を行った.

評価に使用した計算機は, CPU が Xeon 3.2GHz Dual, メモリが 2GB の PC/AT 互換機である.

なお, DWT のノード数が IDCT_ROW と比較して行数の割に非常に大きいのは, 標準 C ライブラリのヘッダファイルに相当する部分が有るためである.

実験の結果を表 2 に示す.

この結果から,

- いずれの場合も混入したバグの見逃しが発生していない
- 条件式の解釈を行うことで誤警告が減少している. 特に未初期化変数の検出では, DWT の例で 46 個有った誤検出が 1 個に, IDCT の例では 27 個から 10 個にと, 従来手法に比べ大幅に減少している.
- その反面, 条件式の解釈を行う場合は, CVC の呼び出しがボトルネックになっている
- プログラムチェッカ自体の処理時間に対し, SDG の生成に掛かる時間が無視できない

ということが判る.

5. まとめと今後の課題

本稿では, SDG を用いて SpecC 記述中のバグの疑いのある箇所を検出する手法の一つとしてヌルポインタ参照の検出手法を提案し, さらに, SDG のノード上の条件式を解釈することで未初期化変数とヌルポインタ参照の検出精度を向上する手法を提案した. また, 実験によって従来手法と比較して誤検出が大幅に減少したことを確認した.

今後の課題としては, 配列の範囲外参照の検出など他のプロパティの検出手法, wait / notify による同期を利用した誤検出の削減, この手法とアサーションの追加とシミュレーションを併用したデバッグ手法, 記述を一部修正した場合の効率的な SDG の再生成手法等が挙げられる.

文 献

[1] Semiconductor Industry Association: "International Technology Roadmap for Semiconductors 1999 EDITION". http://public.itrs.net/.

[2] "SpecC". http://www.cecs.uci.edu/~specc/.

[3] "SystemC". http://www.systemc.org/.

[4] David Evans, John Guttag, Jim Horning and Yang Meng Tan: "LCLint: A Tool for Using Specifications to Check Code", SIGSOFT Symposium on the Foundations of Software Engineering (1994).

[5] "CodeSurfer". http://www.grammatech.com/products/codesurfer/.

[6] M. Weiser: "Program slicing", IEEE Transactions on Software Engineering, **10**, 4, pp. 352–357 (1984).

[7] S. Horwitz, T. Reps and D. Binkley: "Interprocedural slicing using dependence graphs", Vol. 12, ACM Press, pp. 26–60 (1990).

[8] Donglin Liang and Mary Jean Harrold: "Slicing Objects Using System Dependence Graph", pp. 358–367 (1998).

[9] Zhenqiang Chen and Baowen Xu: "Slicing Object-Oriented Java Programs", ACM SIGPLAN, **36**, 4, pp. 33–40 (2001).

[10] Loren Larsen and Mary Jean Harrold: "Slicing object-oriented software", Proceedings of the 18th international conference on Software engineering, IEEE Computer Society, pp. 495–505 (1996).

[11] K. Tanabe, S. Sasaki and M. Fujita: "Program Slicing for System Level Designs in SpecC", Proc. of the IASTED, International Conference on Advances in Computer Science and Technology, pp. 252–258 (2004).

[12] 佐々木 俊介, 田辺 健, 藤田 昌宏: "SpecC 記述のプログラムスライシングを利用した未初期化変数・未使用変数の検出", 電子情報通信学会技術研究報告 Vol.104, No.708 (2005 年).

[13] S. Sasaki, T. Nishihara, M. Fujita: "Slicing - based Hardware / Software Co-design Methodology From Functional Specifications", Preliminary Proceedings of FSEN05 (2005).

[14] A. Stump, C. Barret, and D. Dill: "CVC: a Cooperating Validity Checker", Proc. of 14th International Conference on Computer-Aided Verification (2002).