eXmotion
http://www.exmotion.co.jp/

C言語向けモデル検査ツール

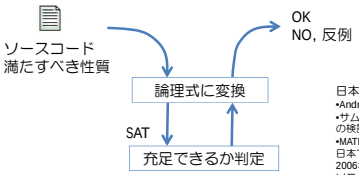
2010.07.10

概要

- 検証のメカニズム
- プログラムとしての正しさの検証
- 設計に対する正しさの検証
- その他の応用
  - 上流工程への展開
  - テスト・デバッグ支援

検証のメカニズム

- プログラムと満たすべき性質を論理式に変換してSATソルバーで満たすべき性質を満足するか調べる
  - 満たせない場合には反例を返す



日本以外では昨年あたりから使われている
•Androidの検証
•サムスンでフラッシュメモリのデバイスドライバの検証
•MATLAB/Simulinkで生成したCコードの検証
日本では
2006年にメモリリーク判定に利用されている(パナソニック)

SATソルバーは、SAT(satisfiability)問題を解くツールの総称
命題数が10^7個以上ある論理式が真になるか高速で判定できる
高速に解けるのでゲームプログラムに組み込まれたりしている

検証メカニズムの利用

- 検証メカニズムを利用することでC言語をSATソルバーのフロントエンドとして利用することができる
- SATソルバーは複雑な制約条件を満たす解を見つけることが本来の使い方
- 知りたい条件の否定を事後条件に指定する

```
int a1, a2, b1, b2, A1, A2, B1, B2;

A1 = 10*a1 + a2;
A2 = 10*a2 + a1;
B1 = 10*b1 + b2;
B2 = 10*b2 + b1;
assert((A1+B1) != (A2+B2));
```

満たすべき性質の否定

→ 42+46=64+24

反例として回文式を得ることができる

- 数独パズル、魔方陣、直交表の生成、リソース制約の解消など応用範囲が広い

プログラムとしての正しさの検証

- ソースコードを持ってきて検証ツールに掛けるだけ
  - PolySpaceやCoverityなど数百万のツールと同じことをフリーのツールで実施できる

任意のソースコード
部分だけでも良い

プログラムとして正しい
配列オーバーフロー,
ゼロの割り算, ポインタ異常等が無いことを保証

オプションを付けて実行するだけ

--bounds-check	enable array bounds checks
--div-by-zero-check	enable division by zero checks
--pointer-check	enable pointer checks
--overflow-check	enable arithmetic over- and underflow checks
--nan-check	check floating-point for NaN

設計に対する正しさの検証

- HoareやDijkustraが導入した {Q}S{R}の形の正しさ
  - Q: 事前条件、S:プログラム、R:事後条件
  - 事前条件、事後条件は関数仕様として前工程から与えられる
- この場合、Q∧S⇒Rが常に成立することをSATソルバーで示す

上限下限クリッピング関数

```
int Clip(int x, int min, int max)
{
  if (x < min) { x = min; }
  else if (x > max ) { x = max; }

  return x;
}
```

```
int Clip(int x, int min, int max)
{
  assume(min < max);

  if (x < min) { x = min; }
  else if (x > max ) { x = max; }

  assert(x >= min  &&  x <= max);
  return x;
}
```

Q:事前条件
S
R:事後条件

事前条件・事後条件を満たすことで
上位要求に対する正しさが証明される

工程間検証: 事前条件の導出

- 関数仕様としてQとRを上位から与えて、Q∧S⇒Rを満たすSを開発することが実装
  - 通常関数仕様としてR(事後条件)を指定することはできても、Q(事前条件)を正確に指定することは困難なことが多いが、検証しながら実装することでQを明確にすることができる
- Qを明確にすることで安全な再利用が可能になる
- Rのみ与えてSとQを作成していくことも可能
  - TDDと同様のアプローチ

```
int Clip(int x, int min, int max)
{
  assume(min < max);   Q

  assert(x >= min  &&  x <= max);   R
  return x;
}
```

関数仕様

```
int Clip(int x, int min, int max)
{
  assume(min < max);   Q

  if (x < min) { x = min; }
  else if (x > max ) { x = max; }   S

  assert(x >= min  &&  x <= max);   R
  return x;
}
```

Rのみ指定した場合はエラーになる。Qの指定またはSの変更が必要になる。

その他の応用例

- 仕様/プログラムの等価性の判定
- テストデータの生成
  - 網羅的生成
  - 特定の条件を満たすでーたの生成
- アルゴリズムの分割
  - 結合テスト支援
  - 詳細仕様記述への展開
- デバッグ支援
  - 論理ブレーク
- コードインスペクション支援
  - 経路分析

例題

- サンプルとして赤で示した部分の検証例を示す
  - 一定周期で温度を測定して、温度が上がりすぎないように管理をする

1. 以下の仕様は同じか?
   - 前回測定温度が200℃以上で，その次は200℃未満，だったら安全である
   - 二回連続して200℃以上だったら危険である
2. 論理式で書くと
   - $(T_1 \geqq 200) \rightarrow (T_2 < 200)$　なら安全
   - $(T_1 \geqq 200) \wedge (T_2 \geqq 200)$　なら危険
3. どちらの仕様が良いか?

等価性の判定

- 以下の論理式の等価性を確認すればよい

```
flg1 = (dataA>=200 && dataB>=200);  //危険
flg2 =(dataA<200 ||dataB>=200);   //安全

assert(flg1 != !flg2);
```


テストデータの生成

- 時系列データの生成

```
tempRead()
{
  static int i = 0, x;
  x = nondet_int();

  return data[i++] = x;
}
```

特定の特性を持ったデータを生成することもできる

```
assume(i != 2 || x > 300);
assume(i != 4 || x > 300);
```


アルゴリズムの分割

- 組込ソフトの場合は時系列処理することが多く、アルゴリズムを分割して実装することになる
- 必然的に結合テストが多くなる
- 分割する前と結果が等しいか確認することが重要になる
- 分割前アルゴリズムを検査式として検査する

検査式

```
for (i=0; i<N-1; i++){
  int flg = (data[i]>=200 && data[i+1]>=200);
  x = x || flg;
}
```

```
assert(alarm == x);
```

```
tempRead()
{
  static int i = 0, x;
  x = nondet_int();
  return data[i++] = x;
}
```

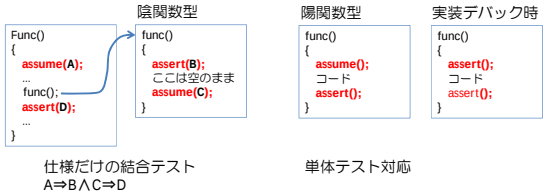
```
int check() {
  static int flg = 0;
  int cur_flg = tempRead() >= 200;

  alarm = alarm || (flg && cur_flg);
  flg = cur_flg;
}
```

```
…
  check();
…
```

タイマー

結合テストの考え方と仕様記述

- 陰関数型の関数を組み合わせることで、コードを全く書かない状態で仕様の検証ができる
  - VDMで作成したモデルと自動テストケース生成を合体させた使い方が可能になる



仕様だけの結合テスト
A⇒B∧C⇒D

単体テスト対応

デバッグ支援

- 何ができるのか
  - 任意の状況を再現できる
    - 任意の場所に到達するための条件の生成
    - ポインターを使用している場合のデバッグは条件設定が難しいが、条件式を設定するだけで状況を作ることができる
  - ソースコードの経路分析
    - コードインスペクションの支援
  - 論理ブレークポイントの設定
- 原因や解決策に至る推論のモレ・ヌケの検出
  - 副作用のないデバッグの保証

論理ブレーク

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int f(int v) {
  return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != 0)
      if (f(x) == p->v)
        if (p->next == p)
          ERROR;
return 0;
}
```

A

- A点に問題がある場合、デバッガを使用しても到達するための条件を設定するのはかなり面倒
- プログラム検証を利用すればassert文を入れるだけで到達できる
  - assert(0)
    - 無条件ブレーク
  - assert(p->v==3)
    - 特定の条件でブレーク
- この特性を利用することで、カバレッジを確保するデータを生成することができる
  - ヨーロッパの鉄道の安全規格ではsil4ソフトの場合、Branch coverage 100%が必要

まとめ

- 下流から形式検証を導入するアプローチ
- このアプローチでは、実装におけるプログラム自身の正しさの証明をほぼ自動化可能
- 上流工程で仕様が明確化されていれば設計に対する検証も可能
- C言語を用いることで、形式検証をテストやデバッグなどの領域でも利用できる