SYNOPSYS®

OpenVera アサーション

2002年4月

はじめに

バグの発見と除去に費やされる時間と労力は、設計自体にかかる時間/労力と比較してより顕著に増加している。また、チップのリスピンに伴うコストも増加している。チップの機能上の問題は、機能仕様の不完全さまたは不明確さ、あるいは単なる人為的ミスが原因である。しかし、現在の検証テクニックは、これらのエラーを発見および除去するのには不十分である。

あるコアをシステムオンチップ（SoC）に組み込んだケースを考えるとする。単に信号をチップ・レベルでチェックするだけでは、組み込まれたコアに対する不適切なプロトコルによって発生するバグを発見できないことがある。プロトコルが継続的にモニタされていれば、プロトコル違反が発生しても、シミュレーションは即座にアサーション・エラーに反応し、問題の真の原因を調査できる。

これに対応するため、デザインの動作仕様を記述したアサーションを作成して利用するアサーション・ベースの検証手法が注目されつつある。アサーションは、テスト・プランの完成に必要な不適切な動作または特定の動作を表すことができる。これらのアサーションは、ダイナミック・シミュレーションまたはフォーマル検証のどちらでもチェックできる。

OpenVera Assertions（OVA）は、 イベント・シーケンスを記述し、それらが正しく発生するかをテストするためのチェックのメカニズムを提供する。この手法は、Verilog-HDLなどのハードウェア記述言語によって提供される手続き型の記述より簡潔で、可読性に優れている。さらにOVAには、必要なコード記述量を最小限に抑える仕様になっている。こうしたOVAの表現力により、複雑なプロトコル・アサーションを、ハードウェア記述言語（HDL）ベースのアサーションよりはるかに少ないコードで記述できる。定義が明快かつコード記述量が少ないため、検証の生産性が向上する。

OVAはダイナミック・シミュレーションに用いることができるだけでなく、フォーマル解析ツールによる論理等価性検証にも用いることができる。従って、設計者は一度OpenVeraでデザインを定義すれば複数の検証環境で使用できるようになる。さらにOVAを使用すると、機能カバレッジをシミュレーションおよび測定するための機能を指定することもできる。

OVAは、すでにビルトインされたアサーションのライブラリを構築または再利用するための言語仕様を備えている。このマクロ機能は、再利用可能なアサーション・ライブラリを構築するためのメカニズムを提供する。構築されたアサーション・ライブラリは、複数の設計グループまたはOpenVeraコミュニティの間で共有できる。これらの共有アサーション・ライブラリを使用することで、設計者は既存の仕様を再利用して、仕様記述の抽象度を高めることができる。

OVAは、OpenVeraオープン・ソース・スタンダードの一部である。このオープン・ソース方式は、オープン・ソースに参画する複数のメンバーによって改善提案が行われるため、短期間でその内容を向上させられることが実証されている。

OVAの機能

OVAは、正規表現と線形時相論理の理論に基づく宣言型のセマンティクスを使用した言語である。これらの2つの理論は、シーケンス、不定量、有限ステートマシン演算などの共通のハードウェア・アクティビティを表現するための強力な組み合わせとなる。

OVAは、検証中に暗黙的およびコンカレントに評価される。ダイナミック検証では、アサーションは継続的に評価される。各アサーションはすべてのステップで評価され、そのステータスはシミュレーションの状態によって更新される。アサーションが満たされた場合、その結果をログに記録し、GUI環境に表示することができる。シミュレーション動作の評価は暗黙的に実行され、アサーションの開始、処理、および評価の段階でユーザの操作を必要としない。

OVAには、以下のものを表現するための機能が用意されている。
・ イベントの順序を指定する基本イベント、およびシーケンス
・ 過去と将来のイベントを参照するタイム・バウンド・シーケンス
・ 論理連結語（AND、ORなど）を使用した複合シーケンス
・ シーケンスの反復
・ 条件シーケンス、およびシーケンス中に保持する条件
・ ユーザ指定の単一または複数クロック
・ シーケンス中のデータの格納とチェック
・ パラメータ化された仕様ライブラリ

OVAは、階層的な検証手法をサポートしている。すべてのアサーションは、アサーションとしてチェックされるか、デザインのプロパティであると見なされる。この機能により、サブ・ブロックの検証に使用されるアサーション・チェックを、そのサブ・ブロックが上位階層でインスタンス化されるときのチェックとして再利用できる。これにより、設計とは独立したサブ・ブロック検証を可能にする階層証明が形成される。

OVAは、過去、現在、または将来のデータ値を参照する機能を提供する。これにより、データ値を特定の時刻でチェックできるようになる。この使用例として、複数の演算の後にFIFO出力が入力に対応しているかどうかのチェックなどがある。また、OVAは自由変数仕様と呼ばれる非決定的変数を持つ機能を提供する。これにより、フォーマル検証ツールはすべての変数代入を検証できる。これはすべての値をアサートする簡単な手法となる。

言語階層

OVA言語は、4つの主要セクションまたはレベルに分割される。最初のレベルはコンテキストで、アサーションの範囲とサンプリングするタイミングを定義するために使用される。次のレベルのディレクティブは、モニタまたはチェックするプロパティを指定するために使用される。3番目のレベルはブーリアン表現で構成される。最後のレベルは時相シーケンスを記述するイベント表現である。

内容

OVAは、アサーションをテスト対象デバイス（DUT）にバインドする3つのメカニズムを提供する。最初のメカニズムでは、moduleキーワードを使用する。このキーワードは、特定のアサーション・セットを設計単位にバインドする。2番目のメカニズムでは、scopeキーワードを使用する。このキーワードは特定のアサーション・セットを特定の回路インスタンスにバインドする。3番目のメカニズムでは、絶対パス名を使用し、回路信号を一意に参照する。

通常、サンプリング・クロックを定義して、いつ回路信号をサンプリングするかを指定する必要がある。OVAは、いつ設計信号をサンプリングするかを指定するためにclockキーワードを提供している。

ここで、単純な例を見てみる。図1は、対応するOVAコードで作成した小規模な8ビット・カウンタのVerilog-HDLコードである。カウンタは0で始まり、255に達すると再び0から始まる。ここでは、シミュレーション中にオーバフロー状態を検出するOVAコードを作成する。

図2に示すOVAコードは、「module counter_8bit」というモジュール指定で始まっている。この指定は、このモジュール定義のすべての文をモジュールcounter_8bitに適用する必要があることを示している。次の行は、「clock negedge（clock）」である。この指定は、このグループ内のすべてのイベントはclkの立下りエッジでその回路信号をサンプリングする必要があることを示している。次行は、イベント「e_overflow」を、時間tにおけるclkの立下りエッジでcntが8'hffと等しく、次に時間t+1においてcntが8'h00と等しいと宣言している。「#1」は、cnt が8'hffと等しくなるときと次に8'h00と等しくなるときの間に1クロック・サイクルのギャップが存在することを指定する。イベント宣言の後は、クロック・コンテキスト・セクションが閉じ括弧「}」で終了する。

次の行では、「assert」ディレクティブと「forbid」構文を使用する。このため、次の文は、このオーバーフロー状態がシミュレーション中に発生してはならないことを表明するために使用される。forbid構文は、不正な状態を記述する必要があるときに使用される。

```
assert a_overflow : forbid(e_overflow);
```

図3のタイミング・ダイアグラムに、出力cntとクロックclk、およびイベントe_overflowを示す。信号値は、クロックのネガティブエッジでサンプリングされる。時間2でcnt=255、時間3でcnt=0であり、オーバーフロー状態となっている。event e_overflowは時間2（上矢印で示されている）において開始時間を持ち、時間3（下矢印で示されている）において終了時間を持つ。このイベントが発生してはならないことを表明しているため、シミュレーション中にこのオーバーフロー状態が検出される。

```
module counter_8bit(rst, clk, cnt);
input rst, clk;
output [7:0] cnt;
reg    [7:0] counter;

always @(rst or posedge clk)
if (rst)
  counter <= 8'b0;
else
  counter <= counter + 1;

assign cnt = counter;
endmodule
```

図1：カウンタのVerilog-HDLコード

```
module counter_8bit {
 clock negedge (clk) {
    event e_overflow :  (cnt==8'hff)
#1 (cnt==8'h00);
  }
  assert a_overflow : forbid(e_overflow);
}
```

図2：カウンタのOVAコード

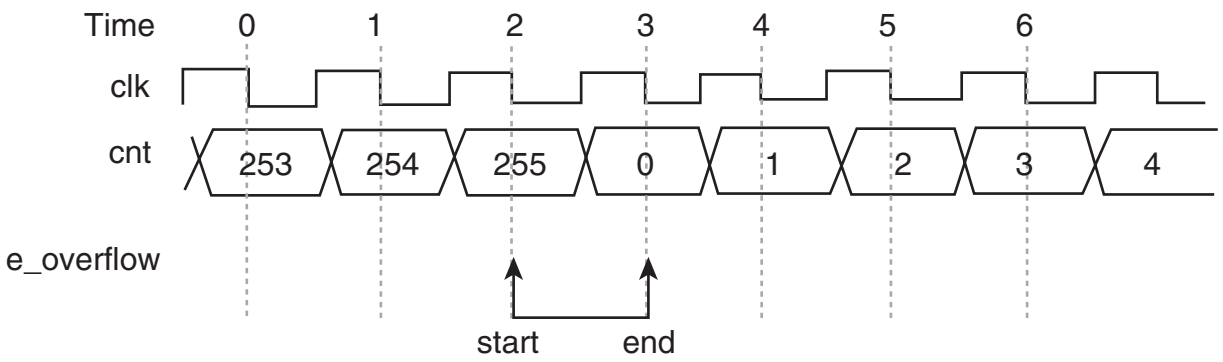

図3：カウンタの波形

ディレクティブ

ディレクティブは、シミュレーション中にモニタするプロパティを定義する文である。ディレクティブは、モジュールまたはインスタンスのコンテキストの中に配置する必要があり、クロック・グループの中には配置できない。2つの一般的なタイプのディレクティブを使用できる。最初のタイプは次のとおりである。

```
assert a_overflow : forbid(e_overflow);
```

このディレクティブは、オーバーフロー状態が発生してはいけないことを表明するものである。肯定のディレクティブ、つまり常に真である必要があるプロパティを記述することもできる。

```
module test {
 clock posedge (clk) {
    event e_mutex : !(a && b) ;
 }
 assert a_mutex : check(e_mutex);
}
```

図4：mutexのOVAコード

図4では、mutex（相互排他的）プロパティが定義されている。信号aおよびbは、同時にアサートされてはならない。 次のディレクティブは、シミュレーション全体にわたってmutexプロパティを適用することを表明している。

```
 assert a_mutex : check(e_mutex);
```

ブーリアン表現

ブーリアン表現は、プロパティを作成するためのビルディング・ブロックとして使用される。ブーリアン表現は、TrueまたはFalseのいずれかに評価される。OVAは0、1、x、zの4ステート値をサポートしている。これらの値は、ブール式で使用できる。OVA式では、Verilog-HDLのすべての論理演算子がサポートされる。

次に、メモリ定義のVerilog-HDL例（図5）を見てみる。この例では、メモリ・インデックスが有効であることを確認したいとする。図6に示すOVAコードは、ブーリアンvalid_addressを宣言している。これは、メモリのインデックスが最大値addrmaxを下回る限りtrueである。

```
module test (clock,rst, wen, wa, din, ra,
dout);
  parameter databits = 8;
  parameter addrbits = 8;
  parameter addrmax = (1<<addrbits) - 1;
  input  wen, clock,rst;
  input [addrbits-1:0] wa, ra;
  input [databits-1:0] din ;
  output [databits-1:0] dout;
  reg [databits-1:0] mymem [0:addrmax];

// Other Code Here

endmodule
```

図5：Verilog-HDLによるメモリ定義

```
module test {
 clock posedge clk {
   event e_valid_address : (wa < addrmax)
;
 }
 assert a_valid_address : check(e_valid_address);
}
```

図6：メモリのOVAコード

イベント表現

図6のオーバーフローの例に示したとおり、イベント表現は一定の時間にわたるイベントのシーケンスを記述するのに便利である。 OVAを使用すれば、イベント・シーケンスの説明を非常に効率的に記述できる。通常、デバッグの観点から、複雑なチェッカを分割して個別にチェックできるようにし、なおかつそれら全体が元のイベント・シーケンスを表すようにすることが望ましい。次に例を見てみる。

信号「a」がハイになるとき、信号「b」は1～4クロック・サイクル後にハイにならなければならず、信号「a」は信号「b」がハイになるまでハイのままでなければならない。

この例では、「a」が論理1のときには、「a」は「b」が論理1になるまでハイでなければならない。この場合、istrue構文が便利である。図7に、OVAコードを示す。

```
module test {
 clock posedge clk {
   event chk : if (a) then
               #1 istrue (a==1) in (#[0..3] b) ;
 }
 assert a_chk : check(chk);
}
```

図7：OVAコード

一致式

上の例では、イベントchkは要求の立上りエッジによってゲート化された。イベントを別のイベントでゲート化すると便利なことがある。イベントはブーリアン・データ・タイプではないので、一致関数を使用する必要がある。この関数はtrueまたはfalseのいずれかを返し、イベントが成功するとtrueになる。一致関数は、イベントの一致を次のクロック・ドメインに送るためにも使用される。図8は、プロトコルのOVAコード例である。

readyが断続的または連続的に3クロック・サイクルにわたってtrueの場合、その後4クロック・サイクル内にtransmitがtrueにならなければならない（その間にresetが発生した場合を除く）。

```
scope tb {
  clock posedge clk {
    event ready3: istrue (!transmit || !reset) in
(ready #[1..] ready #[1..] ready);
    event protocol: if (matched ready3) then
                        #[1..4] (transmit || reset);
  }
  assert c_protocol: check( protocol );
}
```

図8：プロトコルのOVAコード

チェックする初期シーケンスの「ready」は、3クロック・サイクルにわたって断続的または継続的にtrueである。まず、次の行を見てみる。

```
(ready #[1..] ready #[1..] ready)
```

このシーケンスは、readyが3クロック・サイクル内でハイである限り成功し、連続的である必要はない。図9に、2つの有効なシーケンスを示す。

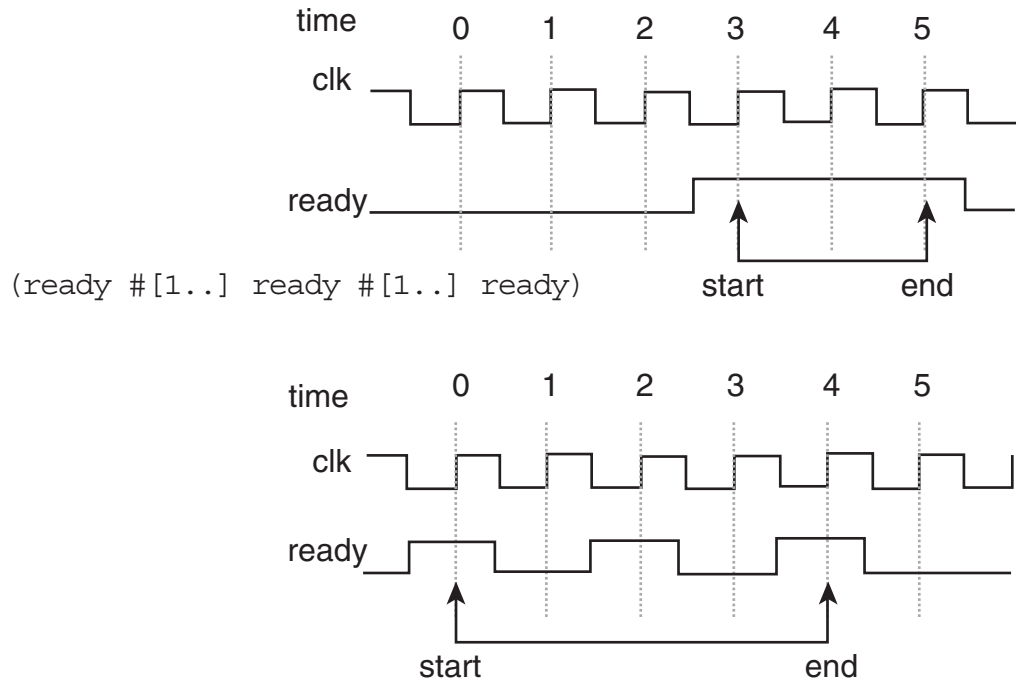

図9：プロトコルの波形図

最初の波形では、readyは連続する3クロック・サイクルにわたってハイである。イベントの開始時間と終了時間が示されている。2番目の波形では、readyは異なるサイクルでハイであるが、これも同様に有効なシーケンスである。これは必要条件ではあるが十分条件ではない。さらに、resetのトリガとtransmitのハイを制約する必要がある。これらの信号を制約するには、istrue構文を使用する。

```
istrue (!transmit || !reset) in (ready #[1..] ready #[1..] ready);
```

これで、transmitとresetはどちらもreadyシーケンス中に非アクティブとなる。このイベントが成功するか、または一致するたびに、transmitがハイになることをチェックする。そのためには、matched構文を使用する。これは、イベントready3の最後にtrueになる。

```
if (matched ready3) then #[1..4] (transmit || reset);
```

イベントready3が成功するたびに、1～4クロック・サイクル後にtransmitがハイになるか、またはresetがハイになる。ready3が一致しない場合、then条件はチェックしない。このチェックは、readyが一致した場合にのみ実行する。最後に、assertディレクティブを使用してシミュレーション中にこのチェックが実行されるようにする。

OVA 2.0の高度な機能

基本的な時相シーケンスおよびイベントを表現するための直感的な構文のほかにも、OVAには高度な用法とアプリケーションのための豊富な言語機能が用意されている。

複雑なシーケンス・チェックは、時相公式を使用して記述できる。時相公式は、時相シーケンスが相関してどのように発生しなければならないかを表明する場合に役立つ。時相公式は、以下の演算子を使用して構成できる。

・ followed_by：(X followed_by Yなど)。少なくともシーケンスXの後ろにシーケンスyが続かなければならない。
・ triggers：(X triggers Yなど)。Xのすべてのシーケンスの直後に少なくとも1つのYシーケンスが続かなければならない。
・ until, wuntil：(X until Yなど)。シーケンスXはシーケンスyが成功するまで保持されなければならない。
・ next, wnext：(next Yなど)。少なくともYの1つのシーケンスが次のクロック・サイクルで保持されなければならない。

トリガするブーリアン表現の先頭にACCEPTまたはREJECTを付加することにより、非同期セットおよびリセットを任意の式に追加できる。これにより、非同期状態が発生したときにアサーションを中止または完了するためのメカニズムが提供される。

まとめ

OpenVera Assertionsは、イベントの時相シーケンスを表現するための高級言語である。ダイナミック検証とフォーマル検証の両方の環境において複雑なプロトコルを記述およびチェックするための強力な構文が用意されている。OVAを使用すると、検証および設計エンジニアは設計のアサーションをチェックするためのコード量を減らすことができるので、生産性を高め、設計バグを発見しやすくなる。

OpenVeraはオープン・ソース言語であり、OpenVeraオープン・ソース・ライセンスに基づいて利用できる。OVA 2.0には、表現力とフォーマル検証サポートを強化するための新機能が組み込まれている。

OpenVera Assertionsは、アサーションベースのメソドロジをサポートする言語プラットフォームを提供する。アサーション知的資産（IP）のライブラリを開発および交換して、検証の抽象度と生産性を高めることができる。

日本シノプシス株式会社

http://www.synopsys.co.jp