Bluespec SystemVerilogを使用したベリフィケーション

概要

Bluespec SystemVerilog™ (BSV)は進化したハイレベルのハードウェア記述言語（HDL）で、抽象化や静的エラボレーション、静的チェックを強力にサポートして設計生産性を2倍以上に飛躍させます。この文書では、BSVのみで設計した場合と、既存のVerilog、SystemVerilog、VHDLおよびe言語、SystemCによるIPと組み合わせた、両方のBSVデザインにおいてどのようにベリフィケーションを行うのかを記載します。また、BSVのオプションを実行する方法や、なぜBSVがベリフィケーションを改善できるのか、BSV以外のブロックと共にBSVを実行する方法、テストベンチやDUTにおいてBSVとBSV以外のブロック間のインタフェース、テストベンチ内でBSVを使用する方法、実行の観測やBSVのデバッグ、BSVブロックにおけるアサーションの使用方法、についてのトピックも含みます。

Bluespec SystemVerilogのバックグランド

Bluespec SystemVerilog ™ (BSV)は、ASICやFPGAの設計に使用されるVerilogやVHDLなどの既存のHDLをハイレベルにおいて置き換えることを目的とした、進化したHDLです。主として設計に焦点がありますが、BSVの進化した機能は仕様（実行可能仕様書、アーキテクチャ探索）、再利用、ベリフィケーション、設計開始までの工程の短縮化についても改善し、仕様からネットリストまでの生産性を2倍以上に引き上げます。BSVの機能は以下のとおりです。

- ハイレベルで宣言型、複雑な並列性をもつ動作に良く馴染む仕様であるBSV Rule（RTL/SystemCの”processやevent”の代替となる）
- ハイレベルで並列動作を意識したモジュール化の仕様：BSV Interface（ポートリストの代替となる）
- ネスト化され、パラメータ化できるインタフェースによって複雑なインタフェースを簡単に構築可能
- 強力なリソースの指定や共有リソースへの多重アクセスの指定
- 複雑な並列性を管理する制御ロジックを自動合成
- 型のハイレベルな構造によるとてもフレキシブルな型のパラメータ化（ポリモーフィズム、オーバーロードなどで表される柔軟性）
- クロック規定の静的チェックを含むマルチクロックの強力なサポート
- 強力な静的エラボレーションによる、簡潔化、反復的な構造のパラメータ化記述、自動ステートマシン生成などが可能
- OVLライブラリもしくは直接のSystemVerilogアサーションを使用したテンポラルアサーション

BSVの備える型、オーバーローディング、パラメータ化、静的エラボレーション、抽象化は構造を指定する能力においてC++やSystemCに匹敵します。RuleやRuleベースのInterface Methodによって、BSVは複雑な並列性の記述においてRTLや動作記述のSystemCのどちらよりも優れた表現能力をもっています。

なぜBSVはベリフィケーションを改善できるのか

ハイレベルHDLに関してこのような発言を耳にすることがあります。「私の設計フローは設計ではなくベリフィケーションで費やされています。もしハイレベルHDLが私の設計時間を0にしても、全体の時間に大した影響はありません。」この誤った解釈は、ベリフィケーションはHDLの複雑度に全く依存しないということを前提としています。このセクションでは、HDL（ここではBSV）の機能がベリフィケーション時間まで劇的に削減することの要点を記載します。

RuleとInterfaceセマンティック

現在の設計における問題の中心は、複雑な並列性を正しく管理することです。この種のエラーは条件の競合、誤作動、状態の矛盾などとして現れます。この種のエラーはまた発見しにくく、原因を突き止めることが困難です。BSV Ruleはアトミックトランザクションのセマンティックであり、それゆえ、その記述によってそのような多くのエラーを完全に排除します。

RTLではインタフェースはポートリストとして指定されていますが、これらのインタフェースのプロトコルは、例えばタイミングダイアグラムや付属資料などの別の箇所だけで指定されます。これらのインタフェースプロトコルの仕様は非公式であったり、不完全、明らかに間違っていることが非常に一般的です。一方BSVではインタフェースはInterface Methodとして指定され、相互のプロトコルのロジックはコンパイラにより自動生成されます。したがって全ての種類のインタフェースエラーはBVSプログラムでは決して起こりません（例 ENABLE信号が有効なデータに対して間違ったクロックで駆動されていた、など）。

より複雑な例として、全てのサイクルでリクエストを受けるか、リクエストを生成するようなモジュールを想像してください。そして、このモジュールを改良して、レスポンスを生成するのと同時に次のレスポンスを受けることができるようにする、と仮定してください。BSVでは、この動作の違いは正式にRule/Methodセマンティックの一部です。コンパイラはスケジューリングの違いを認識し、追跡して、どちらの場合でも正しいクライアントロジックを生成します。

まとめると、複雑な並列性を管理するためのハードウェアが制御ロジックで、このロジックはBSVコンパイラによって、全ての種類のRTLエラーを排除して自動的に生成されます。

抽象型、型チェック、表記独立

BSVには、型をうっかり乱用しないことを確実にするため、とても強力な型チェックがあります。例えばもし2つのステートマシンがあり、両方のステートが別々の型の定義されている場合、ステートを保持するためにBSVの強力な型指定のレジスタを使用することで、最初のFSMのステートを第2のFSMのステートレジスタにうっかり格納しないことを確実にします。

構造と結合を定義する能力によって、アイデアを仕様に近い形で表現することを可能にし、例えばビットベクタの一部を取り出すのではなく、フィールド選択を使用することで、一般的なエラーを排除します。フィールドの追加やフィールドの次数の変更などの構造的な変更を加えるとき、フィールド選択はこのような変更にロバスト（強固）ですが、一方でビットベクタの一部取り出しでは脆弱です。

BSVはビット記述から、データ型の論理的な視点を切り離した強力なメカニズムをもっています。例えば、プロセッサ命令のビット記述が通常まさに直交（独立的）ではありません。すなわち、

あるフィールドの内容が他のフィールドのレイアウトを決定する可能性があります。RTLコードでは、これらの問題を管理することはコードの全体に及び、変更は追跡困難です。つまりエラーが生じやすい傾向があります。BSVでは型を簡単な直交の表現で定義でき、ワイヤやストレージエレメントのデータ型が必要とする非直交の表現を分けて定義することができます。

クロック抽象型と静的な強制クロック規律

BSVでは、クロックは抽象型です。そして信頼性のあるプリミティブのみから操作可能で、型チェックにより正確性が保証されています。さらに、静的チェックは同調器なしでクロックドメインを越えることができないことを確実にします。BSVクロックはパワーマネージメントのためのゲーテッドクロックでもあり、静的チェックは、クロックがゲート状態で安全に同時使用できることを確実にします。静的チェックはゲートがオフのモジュールに対して値を送ったり、ごみデータを読み込んだりできないことを確実にします。

これらの全ての機能によってクロックに関するエラーを排除します。

パラメータ化と強力な静的エラボレーション

BSVは極めて強力なパラメータ化機能と静的エラボレーションをもっています。本質的に全てのBSV構造（モジュール、インタフェース、ルール、関数）は他の構造や生成される構造のパラメータとなることができ、生成される機能はチューリング完全（チューリング機械と同じ計算能力）です。

RTLでは、このような機能を欠いており、しばしば多数の反復記述やカットアンドペーストコーディングによってエラーが生じる傾向があります。

さらに、パラメータ化はベリフィケーションも改善します。いくつかの構造で、すぐに実行できる小さなインスタンスで検証することで、どのようなサイズにおいてもデザインが動作する証拠を導くことができます。

変更へのロバスト性

上述の全てのベリフィケーションの問題は、デザインが変更されるときに拡大します。デザイン変更は以下のような原因で必要となります。

- 終盤になってからの仕様変更
- バグフィックスのためのマイクロアーキテクチャの変更
- 目標性能に合わせるためのマイクロアーキテクチャの変更
- タイミングクロージャ達成のためのマイクロアーキテクチャの変更
- 将来、デザインを再利用するときや、派生製品のデザインを行うとき

どのケースにおいてもHDLが脆弱であるため、ちょっとした変更がとてつもないベリフィケーション負荷を生むことになります。しかしながら、BSVのハイレベルのルールやインタフェースセマンティック、制御ロジックの再生成、型チェック、クロックチェック、異なるパラメータで再インスタンス化、などの機能が新たなバグを生むことなく顕著な変更を実現することに貢献します。

まとめ

もちろん、抽象度を上げることが、デザインの全てのバグを完全に排除するわけではなく、動作記述の基本的な複雑性が残っており、そのためベリフィケーションは必要です。抽象度を上げて達成できることは、ローレベルコーディングに起因する純粋に人為的な全ての種類のバグを排除することです。BSVの抽象化メカニズムは、全ての合成可能なHDLの中で最も強力で、セマンティック的に健全です（多くの合成のできないモデリング言語に比べても）。そして、構造的に正しいデザインを作り出す能力を飛躍的に伸ばし、その結果ベリフィケーション時間を著しく改善します。

BSV DUTとテストベンチのための実行オプション

最初に、全体がBSVで記述されたDUTとテストベンチの実行オプションについて記述します（BSVと非BSVの混合は後述します）。図1はBSVツールとツールフローを表しています。

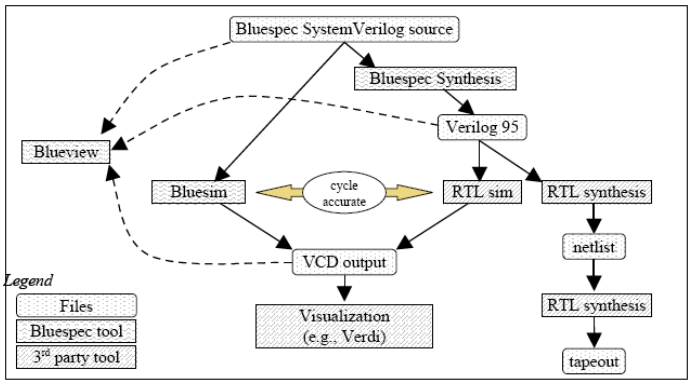

図 1. Bluespec ツールとツールフロー

BSVソースファイルは2通りのうち1つを実行可能です：

- Bluespec合成ツール（bsc）を使用してVerilogにコンパイルし、RTLシミュレータで実行する
- bscでコンパイルしてBluespecのシミュレータBluesimを使用して実行する

最初の方法（Verilogにコンパイル）は、以下の理由で有益です：

- 現在使用しているシミュレーション環境で実行できる
- 現在使用している他のシミュレーション環境（Verilog、VHDL、SystemVerilog、SystemC、e）と自動的に並列動作できる
- 生成されたVerilogは、現在サードパーティの論理合成ツールなどを使用して最終的にテープアウトするためのVerilogと同じであるため、ハードウェア化するVerilogを検証することになる（したがって、もしデバッグや早期のベリフィケーションのためにBluesimを使用したとしても、Verilogシミュレーションを使用して最終チェックやリグレッションも行いたくなるでしょう）

2つ目の方法（Bluesim）は、以下の理由で有益です：

- （現時点）
  - BSVデザインのシングルクロックドメインを扱うことができる
  - シミュレーション速度を2-3倍高速化する
- （今後の予定）
  - より高速化
  - より良いソースレベルデバッグ
  - マルチクロックドメインの取り扱い
  - RTL、SystemCとのコ・シミュレーション

混合DUTとTb（テストベンチ）の実行オプション（BSVと非BSV）

現時点では、混合デザイン（テストベンチもしくはDUT）の主な実行方法は、Verilogに続くルートです。すなわちBluespec合成ツールを使用してBSVソースコードをVerilogにコンパイルし、RTLシミュレータで実行します。その方法では、そのRTLシミュレータがサポートする例えばVerilog、VHDL、SystemVerilog、e、SystemCなどと並列に実行可能です。もしOSCI SystemCシミュレータを使用している場合、RTLシミュレータの標準的な方法でSystemCをVerilogやVHDLと一緒に実行します。将来、Bluesim上でBSVデザインを実行でき、以下のものと共に実行（コ・シミュレーション）できるようになります。

- 標準的なRTLシミュレータで実行しているVerilog/SystemVerilog/VHDL/e
- 標準的なSystemCシミュレータ上（OSCIもしくはサードパーティ）で実行しているSystemC

BSVモジュールに非BSVをつなぐインタフェース（Tb-DUT、Tb内もしくはDUT内）

設計者が最初にBSVを使用する際、一般的には、既存のRTLで書かれた大きなデザイン（新規もしくは既存のIP）に組み込まれるブロックを設計します。また、BSVモジュールをテストするときも既存のRTLやe、SystemCで書かれたテストベンチに組み込まれるでしょう。これらの全てのシチュエーションでBSVモジュールは非BSV内に組み込まれます。

BSVモジュールのインタフェースはダイレクトかつ単純に通常のVerilogポートにマッピングできるため、この種のインタフェースは容易に準備できます。例えば、下記のBSVインタフェース定義を考察します。

```
typedef Bit#(24) Addr;

typedef Bit#(32) Data;
interface IfcDUT;
    method Action request (Addr a);
    method Data response ();
endinterface
```

このインタフェースを提供するDUTがBluespec合成ツールで合成されると、Verilogモジュールは以下の様なポートリストで生成されます：

```
module mkDut(CLK,
             RST_N,
             request_a,
             EN_request,
             RDY_request,
             response,
             RDY_response);
  input CLK;
  input RST_N;

  // action method request
  input [23 : 0] request_a;
  input EN_request;
  output RDY_request;

  // value method response
  output [31 : 0] response;
  output RDY_response;
  ...
endmodule
```

Methodの引数（‘a’のような）はインプットポート（’request_a’）になります。Methodの結果は出力ポート（’response’）になります。Actionメソッド（’request’）は入力”enable”ワイヤ（’EN_request’）をもちます。全てのメソッドは”ready”出力ワイヤ（’RDY_request’、’RDY_response’）をもちます。これらの信号を具体化する単純なプロトコルがあります。RDYワイヤがTrueなるまでメソッドは使用されません。そのときメソッド引数が外部からアサートされ、Actionメソッドが外部からのENAワイヤによって使用され、値メソッドの結果が有効になります。

どのようにBSVインタフェースがワイヤにマップされるかということについての詳細はBluespecドキュメントに記載されており、上記の例はいかに簡単にBSVモジュールが外部のVerilogにインタフェースするかということを示しただけです。Verilog、SystemVerilog、VHDL、eもしくはSystemCのいずれでも、BSVモジュールへのインタフェースは単にRTLモジュールへのインタフェースのようなものです。

将来、BSVモジュールがBluesimとVerilog、SystemVerilo、e、SystemCへのインタフェースと共に実行できるようになっても、基本的な原則は同じで、それは単にRTLモジュールへのインタフェースのようなものでしょう。

テストベンチでBSVの使用（と再利用）

［このセクションはBSVの機能を活用してテストベンチの生産性を高めることに興味のあるベリフィケーションエンジニアに関連し、それ以外はスキップしても構いません］

BSVは迅速に正確なテストベンチを作る数多くの機能があります。

複雑な並列性の表現

今日のテストベンチは複雑な並列性を記述できなければなりません。例えば、

- 同時に生成されたパケットが2のDUTインプットポートに入力される。しかし両方のポートがフローコントロールされていなかい場合
- パイプライン化されたリクエストがDUTインプットポートに入り、パイプライン化されてDUTのアウトプットポートから出力される
- DUTからのアウトオブオーダ命令を扱う
- DUTからの“割り込み”出力に対応する
- DUT環境の並列性をモデル化する。例えば、DUTがプロセッサの場合、テストベンチは並列性や、アウトオブオーダ対応をモデル化する必要がある

これはまさにBSVの強みです。BSVのRuleとRuleベースのInterface Methodはこれらの動作を簡潔に、正確に記述する強力な能力を備えています。

トランザクションインタフェースとインタフェースの抽象化

全てのBSVモジュールはトランザクションインタフェースを持っています。すなわち、ハイパフォーマンスのハードウェアを設計しているとき、BSVインタフェースはInterface Methodにより記述され、テスト環境とモジュール間でトランザクションのやり取りを行います。

さらに、インタフェースがネストされ、型のパラメータ化（多型化）されていることによって、インタフェースがとてもハイレベルに抽象化されているにも関わらず、完全に合成可能です。例えば、BSVライブラリは、以下のインタフェースを提供しています（これはBSVで書かれ、ビルトインモジュールではありません）。

```
interface Put #(type t1);

    method Action put (t1 x);
endinterface

interface Get #(type t2);
    method ActionValue #(t2) get ();
endinterface
```

Put#()インタフェースはひとつのメソッドput()を含み、t1型のxアイテムをモジュールに格納します。Get#()インタフェースはひとつのメソッドget()を含み、t2型のアイテムをモジュールから取得します。ある意味、Get#()とPut#()は互いに多面的です。

これらのインタフェースから、ハイレベルの抽象度をもつ多面的なインタフェースを定義することは容易です。

```
interface Client #(type req_t, type resp_t);

    interface Get#(req_t) request;
    interface Put#(resp_t) response;
endinterface
interface Server #(type req_t, type resp_t);
    method Put #(req_t) request;
    method Get #(resp_t) response;
endinterface
```

Client#()インタフェースはリクエストを生成するget()メソッドと、レスポンスを受け付けるput()メソッドを含みます。Server#()インタフェースはリクエストを受け付けるput()メソッドと、レスポンスを生成するget()メソッドを含みます。

busのDMAブロックのインタフェースを考えてみます。

```
interface DMAIfc #(type bus_req_t, type bus_resp_t);

    interface Server#(bus_req_t, bus_resp_t) config;
    interface Client#(bus_req_t, bus_resp_t) read_stream;
    interface Client#(bus_req_t, bus_resp_t) write_stream
endinterface
```

DMAブロックはサーバとクライアント両方です。CPUで設定されると、サーバ（CPUからリクエストを受け付けてレスポンスをCPUに返す）として動作し、次にクライアントとして設定されるとDMAエンジンはクライアントとして動作します。

これらの例は、どれだけ簡単にBSVによって、システマチックに複雑なインタフェースが構築できるかということを表しています。そして、これらのインタフェースは完全に合成可能です。

接続の抽象化

BSVでは全てのロジックを含む一般的な接続の抽象化と、このような“多面的な”インタフェース型を接続するために必要な制御を書くことが可能です。特に、オーバーロードされたモジュールmkConnectionは型ifc1_tとifc2_tの2つのインタフェースをインスタンス化することができ、それらを接続するために必要なロジックを記述します。例えば、

```
module mkTop (…);

    Get#(int) ifc_g();                  // Line 1
    mkModuleA modA (ifcA);              // Line 2
    Put#(int) ifc_p();                  // Line 3
    mkModuleB modB (ifcB);              // Line 4
    mkConnection (ifcA, ifcB);          // Line 5
endmodule
```

Line1と2はモジュールmodAをインスタンス化し、それらはint型の値を取得するGet#()インタフェースをもっています。Line3と4はモジュールmodBをインスタンス化し、int型の値を受け入れるPut#()インタフェースをもっています。Line5では両方のインタフェースを互いに接続するためのモジュールをインスタンス化するためにmkConnection()を使用します。つまり、modAからmodBの整数値の転送を完全にカプセル化しています。

同様に、Client#(t1,t2)とServer#(t1,t2)型の2つのインタフェースにmkConnectionを適用することも可能です。インスタンス化されたモジュールでは、mkConnectionがクライアントのGet#(t1)インタフェースとサーバのPut#()インタフェースに再帰的に適用され、クライアントのPut#(t2)とサーバのGet#(t2)インタフェースに再帰的に適用されます。

mkConnectionは、それがインスタンス化される特定のインタフェースモジュールを、それが接続される2つのインタフェースの型に依存しているという意味でオーバーロードされます。

BSVライブラリはmkConnectionをオーバーロードして事前定義（Get/Put、Client/Serverのように）されています。しかしオーバーロードはシステマチックにユーザが拡張できます。例えばユーザはmkConnectionを要求されるどんなインタフェース型の対としてでも拡張できます。

つまり、複雑な設定がたった数行の簡潔なコードで記述できるのです（しかも完全に合成可能で！）。例えば、

```
module mkSoC (…);

    Client#(bus_req_t, bus_resp_t) cpu … instantiate …
    Server#(bus_req_t, bus_resp_t) mem … instantiate …
    DMAIfc#(bus_req_t, bus_resp_t) dma … instantiate …
    BusIfc#(3, 2, bus_req_t, bus_resp_t) bus … instantiate …
    mkConnection (cpu, bus.initiators[0]);
    mkConnection (dma.read_stream, bus.initiators[1]);
    mkConnection (dma.write_stream, bus.initiators[2]);
    mkConnection (mem, bus.targets[0]);
    mkConnection (dma.config, bus.targets[1]);
endmodule
```

抽象レベルを超えた接続のリファイン

BSVモジュールがトランザクションなので、ハイレベルであろうと詳細な信号レベルであろうと、抽象レベルを超えたインタフェースのリファインをサポートします。例えばイーサネットパケットを通信する2つのモジュールから始めます。

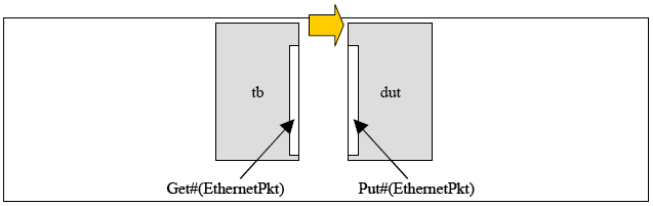

図 2. 抽象化されたインタフェースのテストベンチと DUT

コードは以下のようなものです。

```
module mkTop (…);

    Get#(EthernetPkt) tb … instantiate …
    Put#(EthernetPkt) dut … instantiate …
    mkConnection (tb, dut);
endmodule
```

後ほど、これをバス信号レベルの通信にリファインします。

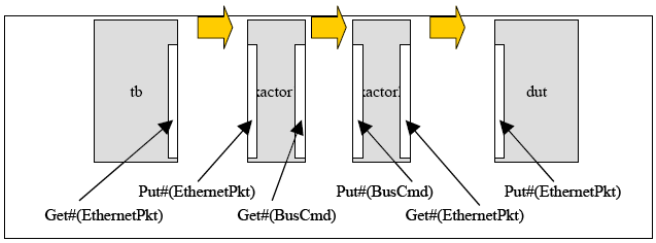

図 3. バス信号インタフェースのトランザクタを もつテストベンチと DUT

これは、以下のようなコードです。

```
module mkTop (…);

    Get#(EthernetPkt) tb … instantiate …
    Put#(EthernetPkt) dut … instantiate …
    Server#(EthernetPkt, BusCmd) xactor1 … instantiate …
    Server#(BusCmd, EthernetPkt) xactor2 … instantiate …
    mkConnection (tb, xactor1.request);
    mkConnection (xactor1.response, xactor2.request);
    mkConnection (xactor2.response, dut);
endmodule
```

テストベンチとDUTは同じままです。単純にトランザクタを透過的に挿入し実際のハードウェアのバス信号と通信する抽象レベルまで落としています。このようなトランザクタはしばしばバス機能モデル（Bus Functional Model：BFM）と呼ばれます。BSVで記述する際、全体の構造は自動的に、テストベンチからトランザクタ１、トランザクタ２を通ってDUTに流れるフロー制御されたパイプラインを形成します。

Ruleベースのインタフェース動作による再利用

BSVインタフェースはRTLのポートリスト以上のものです。BSVインタフェースメソッドはRuleの断片であるため、Ruleセマンティックに従います。Bluespec合成ツールでBSVのDUTをコンパイルするとき、それはトランザクション間スケジュール制約（inter-transaction scheduling constraints）として知られています。

例えば、2つのインタフェースメソッドm1とm2があり、その内部ロジックが共有リソースにアクセスし、m1とm2が同時に実行できないという環境の制約があると仮定します。RTLでは、これらの制約は非公式のドキュメントに記載され、ベリフィケーションエンジニアが簡単に見逃し、そうしてテストベンチは正しくインタフェースプロトコルに従わなくなるでしょう。BSVでは、これらの制約はBluespec合成ツールが正式に解析し、コンパイルされたコードに記録されます。テストベンチがコンパイルされるとき、これらの制約が明らかになりプロトコルを遵守するために必要な制御ロジックが自動生成されます。

したがってRTLとは異なりBSVのテストベンチとDUTはロバストで、壊れやすくはありません。正確な相互接続にはインタフェースプロトコルの正しく遵守することを含み、ただ単にポートリストへの正しい接続だけを意味したものではありません。インタフェースプロトコルが変更された場合には、その変更がポートリスト全体に及んでいるとしても、プロトコルの制御ロジックは再度合成可能であるため、インタフェースはより再利用可能です。

パラメータ化による再利用

BSVは強力なパラメータ化能力をもっています。上記の例が型のパラメータ化を示しています。加えて、BSVの関数とモジュールは他の関数やモジュール、インタフェース、Ruleなどでパラメータ化することができます。これによってテストベンチとDUTが高度に再利用可能になっています。

複雑なデータ型

RTLでは第一にビット表現で動作するのに対し、BSVはユーザ定義のシンボリックな列挙型、構造体、Tagged Unions(型セーフユニオン)、配列を定義できるメカニズムをもっています。これらの全ては多様体（型のパラメータ化）することができます。強力な型チェックによって、型t1（IPアドレスなど）の値が型t2（プログラムカウンタの値など）の値として間違って使用されるようなバグの可能性を削減します。RTLでは単に同じビット数(例えば32ビットなど)であるというだけで判断して間違うことが考えられます。

スタティックエラボレーションにおいて、BSVプログラマはリスト、ツリーその他の再帰的データ構造も使用できます。

データ表現の柔軟性

他の次元の抽象度は、ビット表現からデータの型の論理的な視点を分離することです。例えば、'int'データ型はリトルエンディアンもしくはビッグエンディアンフォーマットで表すことができます。プロセッサ命令は命令、ソースレジスタなどのフィールドを含む構造として論理的に見ることができます。しかし実際の32ビット命令のこれらのフィールドのパッキングは重要であるかもしれません。

BSVはこれをシステマチックな方法で対処します。データ型は常にこれらの論理的な構造によって見られます。一方、プログラマはpack()とunpack()関数を定義して論理的な視点からビット表現に変換することができます。変換はアービトラリになり得ます。Bluespecコンパイラは必要な箇所にこれらの関数を自動的に使用し、プログラマのコードは表現の方法で散らかった状態にする必要ありません。

ランダム数生成

BSVライブラリには、自由な幅のランダム数発生器を含んでおり、簡単にランダムテストベクタを生成できます。

“期待値”の保持とスコアボード

BSVライブラリにはいくつかの“レジスタファイル”や、一時的なデータを保持しテストベンチとして使用されるモデルがあります。

BSVライブラリにはFIFOファミリがあり、深さや中に入れるデータの型を完全にパラメータ化できます。それらはテストベンチ内で“保留中のレスポンス”を保持するために使用されます。すなわち、DUTに入れるリクエストを生成した後、予想値をFIFOに入れて、多少の遅れの後に返ってくるDUTのレスポンスをチェックします。

リクエストに対して不適切なレスポンスを発生するDUTのために、BSVライブラリは“完全なバッファ”を含んでいます。その概要は

- 各リクエストにユニークな、順序付けられたタグを付ける
- どのような順番でもタグと共にレスポンスを受付け、それを保持する
- タグの順序によってレスポンスを生成する（“期待値”のFIFOに対してチェックされます）

ステートマシン生成

テストベンチは頻繁にシーケンスアクションを慎重に編成する必要があります。トランザクタは特に、各ロジックのトランザクションのために低レベル信号のシーケンスを生成したり受け入れたりするステートマシンを含んでいます。

BSVはステートマシンを表現するための強力な表記方法と構造

メカニズムをもっています。

```
Stmt test_seq =

   seq
        for (i <= 0; i < NI; i <= i + 1)
             for (j <= 0; j < NJ; j <= j + 1) begin
                let pkt <gen_packet ();
                send_packet (i, j, pkt);
             end
        par
           send_packet (0, 1, pkt0);
           send_packet (1, 1, pkt1);
        endpar
   endseq
mkAutoFSM (test_seq);
```

最初の記述はステートマシンを指定し、2つ目はFSMをインスタンス化しています（生成する全てのロジックを明示的に指定する必要あり）。seq/endseqブロックは2つのコンポーネントを含み、par/endparコンポーネントに従うforループでシーケンシャルに動作します。forループは単にシーケンスアクションを表しています。ここで、ループはsend_packet()コールのシーケンスを実行し書くインプットiからDUTの出力jに送信します。

FSMがルールセマンティックに従うため、FSMは自動的に引き止められます。例えばテストベンチがsend_packet()を準備し終わってもDUTがパケットを受けられないときなど。

par/endparセクションはoutput1に向かうinput0とinput1の2つのパケットを正確に並列に送付します。再度述べますが、もしどちらか一方のパケットが受信できないときには、Ruleセマンティックによって、どちらのパケットも送信されません。

まとめると、BSVは複雑なFSMを表現する強力な方法をもっており、このFSMはRuleセマンティックの利点を享受できます。

クロックの規則

BSVは、クロックの規則の厳密なスタティックチェックをし、同期器などのあるマルチクロックドメインとゲーテッドクロックをサポートする強力な言語です。これらの機能は

- DUTのためのテストベンチでクロックを生成する正確な表現
- テストベンチのDUTへの正確な接続

合成可能なテストベンチ

BSVは”合成可能なサブセット”をもちません。全てのBSVは合成可能です。したがって、BSVテストベンチは例えば、FPGAベースのテストプラットフォーム上の実行のためなどにRTLに合成可能です。

実行の観測とBSVブロックのデバッグ

ブロックを検証するとき、入力信号を供給することと出力の収集（これは前のセクションで記述）することに加え、内部信号を観測しなければなりません。これは特にデバッグ、すなわちテストベンチからレポートされるエラーの原因究明をするときに当てはまります。

Bluespecにより生成されたVerilogの構造

Bluespecが生成したコードをVerilogシミュレータで観測したりデバッグするとき、Verilogコードの構造を理解することは有益です。この構造を頭に入れることで、Verilogコードとその波形をBSVソースに関連付けすることが容易になります。

ステートエレメント、モジュール構造、インタフェース

スポートリスト

生成されたVerilog内の全てのステートエレメントは、厳密にBSVソースで指定されたものと同一です。BSVコンパイルでは、ステートエレメントの”推定”はありません・・・あなたが指定したものが、あなたが手に入れたものです。

モジュールとモジュールのインスタンスは、Verilogと同様の方法でBSVソースに指定します。生成されたVerilog内の各モジュールは、BSVソース内のいくつかのモジュールと厳密に対応します。しかしながら、逆は成り立ちません。全てのBSVソースのモジュールがVerilogのモジュールになるわけではありません。BSVプログラマには(* synthesize *)属性をBSVモジュールの上に記述するというオプションがあります。この各モジュールがVerilogのモジュールになります。他のモジュールはBluespecコンパイラによってインライン化され、Verilogで分かれたモジュールとしては認識できません。

BSVモジュールに非BSVをつなぐインタフェース（Tb-DUT、Tb内もしくはDUT内）”のセクションで記述したように、BSVモジュールのインタフェースと、Verilogモジュールのポートリストの間は、とても直接的にマッピングされています。

BSVコンパイラはステートエレメントとモジュールの名前を、生成されるVerilogにできる限りそのまま残すように設計されています。一般的には、複数のモジュールやステートエレメントのインスタンスが存在するときや、“生成された”構造がある場合に、名前を厳密に残すことはできません。複数のインスタンスが一意の名前を必要としますが、最終的な名前はオリジナルのソースの名前を含みます。

内部の組み合わせロジック、CAN_FIREとWILL_FIRE信号

ステートエレメント、モジュール階層、ポート信号以外に残っているのは、各モジュールの内部の組み合わせロジックです。Verilogのこのコードを理解するためには、BSV Ruleを導入するために必要なロジックの構造を理解することが重要です。以下の例を使用することにします。このルールは2つの数値のGCD（最大公約数：Greatest Common Devisor）の計算を行う、ユークリッドのアルゴリズムを表しています。レジスタxとyが2つのインプットの数値によって初期化されると、 y が0になるまで実行され、そのときにxは、最初の2つの数値のGCDになります。

```
rule decr ((x <= y) && (y != 0));

     y <= y – x;
endrule

rule swap ((x > y) && (y != 0));
     x <= y;
     y <= x;
endrule
```

最初に、ルールdecrを実装したVerilogの構造を取り出して見てみましょう（図 4）。

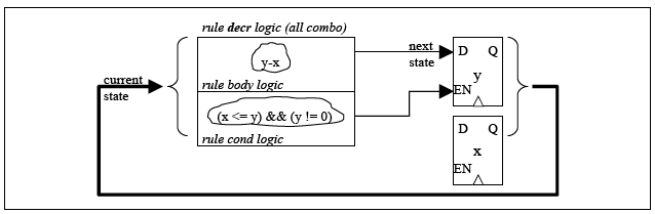

図 4 単体のルールのみを取り出したロジックの実装

ルールの条件は常に、どのルールが開始されるかという条件であ

る1つの真偽値を計算する純粋な組み合わせ回路です。ルール本体は、常に単なる組み合わせ回路で、ルールが開始されたあとに続く次の状態を計算します。次に図 5は両方のルールを実装したときの生成されたVerilogの構造を示しており、両方にreadとwriteといくつかの共通のステートがあります（このケースでは、両方のルールがレジスタyを更新します）。

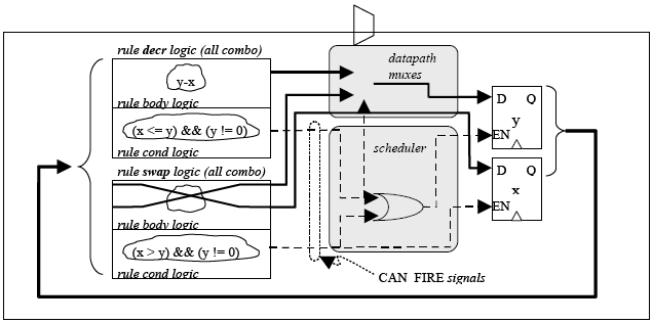

図 5 両方のルールを実装したロジック（スケジューリングとデータパスのマルチプレクス）

ステートエレメントとルールの状態、本体のロジックが追加されたことが分かります。BSVコンパイラがスケジューラロジックとデータパスマルチプレクサを挿入しました（全ては完全に組み合わせ回路です）。データパスマルチプレクサは、同じステートエレメントがアップデートされるルールの出力を結合します。スケジューラロジックはルールの状態（CAN_FIRE信号として知られている）の出力をとり、データパスマルチプレクサと、ルールのサブセットの開始に影響するステートEN信号を制御します。

（y!=0が繰り返されているように見えるロジックについて危惧しないでください。この図は表示しているだけです。事実、BSVコンパイラはとても強力な共通式の最適化を発揮し、ロジックの共有を確実なものにします。）

この単純な例では、ルールの条件は相互に排他的です（ルールdecrはx<=yのときだけ開始され、ルールswapはx>yのときだけ開始されます）。しかし、一般的にBSVのルールは、もしそれらが同じステートをアップデートするとしても相互に排他的条件である必要はありません。このケースではスケジューラロジックは、矛盾したステートになるルールの開始を引き止めておくアービトレーションロジックを含みます。CAN_FIRE信号がTrueになってもルールが引き止められるかもしれないので、スケジュールされたバージョンであるWILL_FIRE信号を参照します。

図 6はアービトラリBSVプログラムの一般的なケースを表しています。

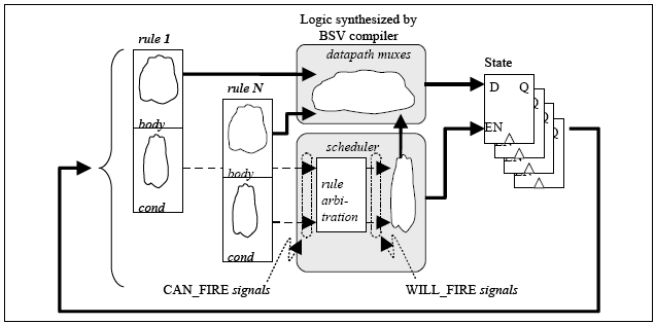

図 6 BSV コンパイラが生成した Verilog の一般的な構造

まとめると、生成されたVerilogを調査したり、波形を見るときに意味するところは以下のとおりです。

- モジュール階層はBSVソースから直接認識できる
- モジュールのインタフェースポートは、BSVソースから直接認識できる
- 全てのステートエレメントは、BSVソースから直接認識できる
- 各モジュールの中で、条件と本体のロジックはBSVルールのソースから直接導かれた（もちろん、真偽値とロジックの最適化は行っている）
- “新規”に合成されたロジックは、スケジューラとデータパスマルチプレクサのみ。新規とはいっても、CAN_FIREとWILL_FIRE信号はソースのルールの名前に準じているため、すぐに認識できます。

したがって、生成されたVerilogをソースレベルでデバッグすることはとても簡単です。

実行時の値の観測

BSVプログラマは、Verilogプログラマと同様の方法で実行時の値を観測することができます。最初に、BSVプログラムに$displayと$writeを従来どおり挿入することができます。次に、VCDをダンプして、通常通り波形を観測することができます。一般的なVerilogの$dumpvars、$dumpon、$dumpoff システムタスクをBSVコードに使用することができます。

最適化でも特定の信号が削除されないようにする

Bluespecコンパイラはコードの品質を高めるために積極的に最適化を実施します。このような最適化によって、ソースコード内にある信号全体が、生成されたVerilogから削除されるかもしれません。これはもちろんデバッグを悪化させます。

コンパイラ用のフラグ-keep-firesは、全てのルールのCAN_FIREとWILL_FIRE信号を取り去らないように、コンパイラに指示します。後述するように、これらの信号を観測することは、どのルールがなぜ開始されないのかということの原因を突き止めるのに有益です。ユーザーズガイドにはこのフラグについてより詳細を掲載しています。

“Probe”機能は、出力のVerilogのために選択して、名前付けされた信号を作成することを可能にします。例えば、

```
module …

  Probe #(int) foo <mkProbe;
  …
  rule r (… rule condition …)
    …
    foo <= … some expression of type int …;
    …
  endrule
  …
endmodule
```

これによって、生成されたVerilogでソースコード内のfooを含む信号が存在することを確実にします。したがってVCD波形はいかなるアービトラリ中間値を作ることができるようになります。

Blueviewの使用

BluespecのBlueview ™は、GUIベースの環境で、BSVソースコードと生成されたVerilogおよび、NovasのVerdi波形ビューワの波形をクロスリファレンスできます。図 7はBlueviewのスクリーンショットを示しています。上段の2つの表示は、Blueviewウインドウのもので、左側がBSVソースコードを示し、右側が生成されたVerilogコードを示しています。どちらの表示でも左側のペインが標準モジュールの階層表示で、そこでアイテムが選択されると、右側のペインで対応するコードの箇所が表示されます。下段の図は、NovasのVerdiツールのスクリーンショットで、BSVコー

ドの実行からの波形を表示しています。この3つの表示はクロスリファレンスされます。それぞれのコード表示で、様々なアイテム（ステートエレメント、インタフェース、ルールなど）がハイライト化されています。このようなハイライト化されているアイテムをクリックすることで、他方の対応したコード（例えば、BSVソースから、生成されたVerilogを見たり、逆もできます）もしくは、Novasツールの対応した波形を見ることができます。Novasの特定の波形を選択して、コード表示で対応するコードを見つけ出すこともできます。

Blueviewの操作方法は、Blueviewのユーザーズガイドに記載されています。

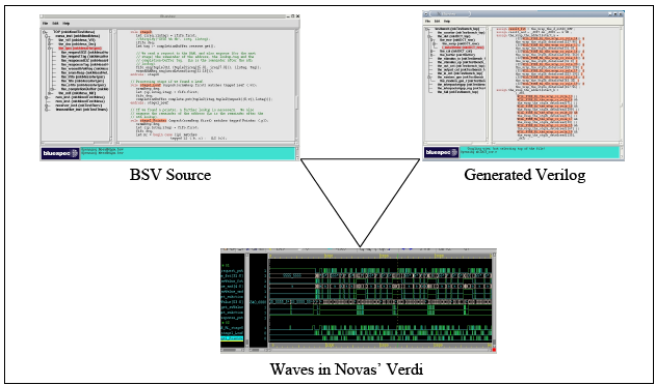

図 7 Blueview の GUI コンポーネント

BSVの実行のデバッグ

BSVの実行を観測することは単なる第一段階です。もし問題を発見したら、次の段階ではなぜ問題が観測されたのかという原因を突き止める、デバッグを行い、そして問題を修正します。

BSVの実行を観測するときには、基本的な2種類の問題があります。

- 間違った値が、どこかで計算された
- アップデートされるはずなのに値がアップデートされない

これらは相互に関連しています。つまり、ある時間T1の間違った値が、時間T2における別の場所の値をアップデートしなくなる、その逆もしかりです。言い換えると、間違った値がステートに格納されることによって、将来行われるべきアップデートを妨げ、それが順々につながって他の間違った値の結果となるのです。両方のケースでも、観測された問題から遡って、問題の原因を追跡したいと思います。時間的に遡るだけでなく、空間的（すなわち、観測された問題の原因となるロジックコーン）にも遡って追跡したいと思います。

間違った値のデバッグ：ロジックコーンの標準的な追跡

図 6を参照すると、真偽値の最適化や共通のサブ表記の共有などの最適化を除いて、デザインの全ての実際の機能ロジックがソースコードで明示的に示されていることがわかります。言い換えると、全てのステートエレメントは明示的で、機能的な組み合わせロジックはRuleとInterface Methodの条件と本体において明示的です。

そのため、例えば、もし－の代わりに＋と書いたり、値を増加させることを忘れたとき、このバグの場所を見つける方法は標準的なものです。問題の値からそれを作り出している演算子、演算子の入力、等まで、問題の原因を突き止めるまでロジックコーンを遡ります。このような機能的なエラーは、BSVソースコード内のバグ（おそらくタイプミス）に直接関係しているでしょう。

もしレジスタに書き込まれる値が間違っている、より一般的に言うと、もしインタフェースメソッドの引数として送られる値が間違っていると、BSVソース内でその値を発生させていると思われる全てのルールを特定します。この起こりうる書き込みは常にスタティックに判明します。例えば単純なソースのテキスト検索でそれらを特定したり、波形ビューワで遡って追跡することによって迅速に特定します。例えば、レジスタxに書き込む3つのルールがあり、各ルールはxの新しい値を計算するロジックを含んでいるとし、これら3つの値はxの入力であるDにマルチプレクスされると仮定します。同じ法則が、共有されたサブモジュールで同じインタフェースメソッドを使用する複数のルールについて当てはまります。マルチプレクサを残すことで、間違った値の原因を探し出すために次のレベルに戻して追跡することが可能になります。

ルール開始のデバッグ

BSVプログラムのその他の種類のバグは、RuleもしくはInterface Methodが予期されるときに実行されず、予期されるとおりに値がアップデートされないことです。これは様々な方法で明らかになり、例えば次のような方法です：

- FIFOに値を格納するRule/Methodが開始されないことが原因でレジスタやFIFOが間違った値を保持していることが観測される
- FIFOに値を格納するRule/Methodが一度も開始されないことが原因でレジスタやFIFOが空の値を保持していることが観測される
- 全てのルールが開始されず、システム全体が止まった状態になっていることが観測される

これらはつまり、Ruleが予期されるときに開始されない、ということです。

再度図 6を参照すると、CAN_FIRE信号がFalseかWILL_FIRE信号がFalseのどちらでもRuleは開始されません。全てのRuleのCAN_FIREとWILL_FIRE信号は一般的に、生成されたVerilogで明確に見ることができます。コンパイラフラグ-keep-firesは、この信号が最適化によって削除されないことを確実にします。

最初のステップは、そのCAN_FIRE信号が意図せずFalseになっていないかどうか観測することです。このケースは標準的なロジックコーントレースすることを要約しています。すなわち、CAN_FIRE信号がRuleの明示的な条件の単なる組み合わせで、Ruleで使用されているInterface Methodの条件と共に、全てのInterface MethodはこれらのMethodなどで使用されます。これらの全ての条件はソースコードにおいて明示されています。標準的なロジックコーントレースは、なぜ条件が意図せずFalseなのかを認識させることができます。ソースコードの間違った演算子もしくはいくつかのステートエレメントの値が間違っていることの両方において。

もしCAN_FIRE信号が意図どおりにTrueで、WILL_FIRE信号が意図せずFalseの場合、ルールが共有リソースを使用する他のルールが競合しているために、Bluespecが作成したスケジューラロジック（再度図 6を参照）が開始されることを阻止しています。そしてスケジューラが他のルールに優先度を与えています。このケースでは、スケジューラロジックをトレースしようとするよりも、コンパイラによってスケジュールを作成させることを試行することが簡単です。ソースをコンパイルするとき、フラグ-show-scheduleはコンパイラにスケジューリングレポートを書かせます。このレポートは共有リソースのルールの間の全ての競合を記述し、選択の優先度を記述します。この情報は、CAN_FIRE信号がTrueであるのになぜルールが開始されないのかを語ります。

もしルールが開始されないスケジュール上の理由を突き止めたら、BSVプログラムで衝突したリソース本体が明らかになります。リソースの衝突を解決することは、標準的な方法で実行されます。例えば、

- 優先度を変更してスケジュールを変更する。例えば、descending_urgency属性を使用して共有リソースが競合しているルールの優先度を変更する。
- タイミング変更でスケジュールを変更する。例えばルールを含んでいるステートマシンを修正し、異なる時間が異なる条件下でリソースへアクセスするようにする。
- 複製。例えば、レジスタを複製し、それぞれのルールが異なるレジスタにアクセスするようにしたり、モジュールのポートを数を増やしてそれぞれのルールが異なるポートにアクセスするようにする。
- マイクロアーキテクチャの変更。例えばカウンタを増加/減少が同時にできるように修正したり、FIFOのenq/deqを同時にできるように修正する。

ルールやメソッドのスケジュールに相当するソースコードでスケジュールアサーションも配置することができます（詳細はリファレンスガイドを参照してください）。

- インタフェースメソッド属性のalways_readyとalways_enabledはメソッドの条件が毎クロックでTrueにすることや、ActionもしくはActionValueメソッドを毎クロックで有効にすることをアサーションします。
- Ruleの属性fire_when_enabledとno_implicit_conditionsはルールのWILL_FIREがCAN_FIREと等しいこと（つまり、他のルールから引き止められない）や、ルールのCAN_FIREがルールの明示的な条件に等しいこと（つまり使用しているメソッドの全てが常にReadyであること）をアサーションします。
- Rule属性descending_urgencyは共有リソースの競合のときに、コンパイラに対して優先度を知らせます。

これらの全てのスケジューリングアサーションはコンパイラによってスタティックにチェックされ、もし不可能なルール/メソッドのスケジュールがアサーションされたら、エラーを返します。

その他の一般的な落とし穴は、RWire（そのwgetメソッドを使用）から有効ビットをテストせずに盲目的に値を読み出すことです。すなわち、RWireのwgetメソッドが、他のいくつかのルールから同時に呼び出されることを仮定します。もし有効ビットがFalseであれば、RWireに運ばれる値はごみデータでしょう。もしこの現象を観測したら、以下のことができます。

- 有効ビットをテストし、無効なケースを正しく扱えるようにする
- ルールのスケジュールをより早く、wsetが予期したときに実行されるように修正する

カバレッジ

再度図 6と最初のほうで述べたモジュール階層を参照すると、カバレッジの大きさは、ステートエレメントのアクセス、モジュールポートの活動など、RTLのときと同じであることが分かります。

動作のカバレッジは以下のように見積もられます。

- RuleのWILL_FIRE信号が、実行の際にどれくらいの頻度で開始されるかを示すことができる
- MethodのENABLE信号が、モジュールのメソッドがどれくらいの頻度で実行されたかを示すことができる

BSVのアサーションベースのベリフィケーション

アサーションベースのベリフィケーションの人気は最近増し、その傾向は続いていくようです。アサーションはロジックベースの形式主義を表現する宣言ですで、デザインが持つべきいくつかの属性を表します。アサーションは即時的かも知れません。つまりずっと保持しているはずです。もしくは、アサーションは一時的であるかもしれません。つまりある一定期間の属性の値を表現します。即時的なアサーションの例は、”レジスタxの値は0から5の間である”。一時的なアサーションの例は”ステートがBUSYになったら、それからそのステートは50サイクル以内にIDLEになる”。一時的なアサーションは特にプロトコルの属性を表現するのに便利です。

原則では、興味のあるどんな特性も、チェックするためにRTLもしくはBSVソースコードを書くことができます。しかしアサーションが単体で記述されるため、ロジックベースの形式主義で、それらは宣言型で、とてもハイレベルで、そして属性の正確さの独立した仕様を表現します（つまり、実装とは独立しています）。アサーションはスタティックに使用（例えば、定理証明）することで属性をチェックすることができる部分において将来のフォーマルベリフィケーションの基本でもあります。これらの理由のために、アサーションは最近、ますます際立ったベリフィケーションのテクニックになりました。Accellera標準の本体で31のパッケージ化されたVerilogで書かれたアサーションのセットを提供するOVL（Open Verification Library）を提供します。各アサーションはパラメータ化されたVerilogモジュールです。SystemVerilogはSystemVerilog Assertions（SVA）と呼ばれる、そのような属性を表現するための下位言語をもっています。

BSVプログラマにはアサーションを使用する2つの方法があります。1つ目にBluespecは、BSVモジュールとしてBSV内で使用できるようにしたOVLモジュール、”ラップされた”OVLライブラリをもっています。したがって、BSVプログラマは標準のOVLメソドロジをBSV環境内で使用可能です。

2つ目に、BSVソースコード内にSVAアサーションを直接書くことができます。これは現時点ではベータ版です。

まとめ

BSVの抽象化メカニズムは、全ての合成可能なHDLの中で最も強力で、セマンティック的に健全です（多くの合成のできないモデリング言語に比べても）。そして、構造的に正しいデザインを作り出す能力を飛躍的に伸ばし、その結果ベリフィケーション時間を著しく改善します。このドキュメントはこれらのこれらの考え方を探索し、BSVデザインをデバッグすることと、テストベンチを構築することの両方に対してベリフィケーションにBSVを使用することに関するいくつかの詳細を記述しました。

お問い合わせ先：

CYBERNET

サイバネットシステム株式会社
新事業統括部
〒101-0022 東京都千代田区神田練塀町３富士ソフトビル
Tel: 03-5297-3295　Fax: 03-5297-3637

e-mail: bluespec@cybernet.co.jp
http://www.cybernet.co.jp/bluespec/