bluespec

自動化されたフォーマルインタフェース規約を使用したベリフィケーションの削減

概要

大きく複雑なシステムを堅牢に(正しく)構築するためには、拡張可能で組み立て可能な規律が必要です。各モジュールのインタフェース - もしクライアントが正しい方法でモジュールを駆動し、そしてホストが規定どおりの方法で動作することを保障する、というような、ホストとクライアント間の規定を指定すべきです。これはソフトウェアエンジニアリングにおいては従来からあるアイデアで、現在PSLやSVAなどのアサーションを通してハードウェアデザインに導入する方法を模索しています。しかしながら、インプリメントが自動的に実現できて、自動的にクライアントで自動的に実行される場合だけ、そのようなメソドロジが “ベリフィケーションのボトルネック”を完全に削除します。すなわち、これは“correct by construction”のフレーズのまさに本質です。この資料ではBluespec SystemVerilogのルールとルールに基づくインタフェースがこのアイデアを実装するのかを記述します。

1. イントロダクション

ムーアの法則のおかげで、今日のチップは数千万ゲートをもつSoCを実現しました。チップ開発のコストは巨額で、そのため費用を回収するためには大きなマーケットを必要とします。しかしそれより小さなマーケットもあります。チップデザインの失敗によるコストは、ただ単にリスピンに掛かる業務と材料費のみでなく、マーケット規模の損失や縮小による潜在的な巨大な機会損失を意味します。

チップ開発の中心要因となるコストは“ベリフィケーションボトルネック”です。今日、おおよそ開発費の70%のリソースがベリフィケーションに掛かっています。これは明らかに複雑性の問題を解決するために対処しなければならない根本的な要素です。

一つのアプローチは、既存もしくは購入した“検証済み”IPブロックを使用して、一からシステムを設計することを回避することです。残念ながら、このようなIPを接続することは、それ自身が複雑で費用のかさむ問題を引き起こします。

上述のベリフィケーションとIPリユースの多くの困難は、モジュール構成に対するフォーマルなメソドロジが欠如していることに起因しているかもしれません。だれも大きな構造物を時代遅れの継ぎ手で構築することはできません。全体の構造がもろく、ちょっとした予期しない外因で倒壊するかもしれません。

私たちは、システマチックで強力、フォーマルな、例えばホストとクライアント間の規約のような、モジュールのインタフェースを必要としています。インタフェースセマンティックが単に構造（型と接続性）だけでなくビヘイビア（プロトコルの信号とスケジュール）を含むべきであることは即座に明らかです。さらにインタフェースのビヘイビアはモジュールと、そのクライアントのビヘイビアと密接に連携し、インタフェースセマンティックは個別に考慮されるべきではありません。それらはモジュールのビヘイビアのセマンティックと有機的に結合するべきです。

モジュールインタフェースがホストとクライアント間の規約を指定すべきであるというアイデアは、ソフトウェアエンジニアリングでは古くからあるアイデア[Design By Contract]で、今日、インタフェースに取り付けられたアサーション[Foster 2004, SynopsysABV 2003] や PSL[PSL 2005] 、 SVA もしくは SystemVerilogアサーション[SV 2005]などの言語という形でハードウェアデザインに導入する方法を模索しています。

しかしながら、このアイデアはそれが建設的に使用される場合にのみ“ベリフィケーションボトルネック”を完全に削除します。すなわち、この規定が実装時に自動的に作られ、生成されたコードが使用される場合には、構造上、規約違反を犯すことができません。この方法では、インタフェースアサーションを手書きで入力する際に起こりうる制限事項：アサーションの正確性、シミュレーション速度、何千ものモジュールインスタンスの繰り替しチェックの必要性、などが起こりません。

もちろん、誰もモジュールの完全なインタフェース規約を自動的に推測することはできません。これは制限のない問題であり、一般には解決が困難です。しかしながら、扱いやすく、高度に利用しやすいインタフェースセマンティックのサブセットであり、Bluespec 合成ツールが自動的に取り扱うBluespec SystemVerilogのルールとルールベースのインタフェース[BSV 2004]では、建設的な方法で記述可能です。第一に、BSVはRTL（そしてSystemCにおいても）見られる、ピンリストそのものの記述よりも高いレベルで考え、記述することのできるフレームワークと記述方法を提供します。第二に、BSVは合成ツールが取り扱い、明示的に表現できるフォーマルセマンティックをもっています。第三に、このフォーマルセマンティックは、正確に使用されることを保障するために、モジュールのコード生成時に使用されます。これがインタフェースセマンティックのサブセットであるため、ベリフィケーションの問題やインタフェースのアサーションのいくつかの型の値を削除することはできません。しかしトータルではこれらの特性がハードウェアデザインでモジュールを組み立てるときに見つかる多くの共通のエラー（ポート使用の違反、条件の競合、矛盾する状態、間違った信号など）を削除することによって、ベリフィケーションのボトルネックを削減します。より重要で、高度なベリフィケーションの問題にフォーカスさせるために設計者をこれらの問題から解放します。

この資料の構成

セクション2は現在のRTL（Verilog、SystemVerilog、VHDLそしてSystemCにおいても）によるメソドロジの分析で、組み立てやすさの欠如と、それがベリフィケーションのボトルネックに起因することを示します。

セクション3ではPSLもしくはSVAのアサーションを含むいくつかの部分的な解決策を議論します。

セクション4はBluespec SystemVerilogのルールとルールベースのインタフェースがいかにこの問題に真正面から取り組むかを議論します。

最後にセクション5でまとめと結論を行います。

2. なぜ RTL による現在のメソドロジがベリフィケーションのボトルネックとなるのか

用語：これからの議論では特定のモジュールを示すために“IP ブロック”という用語を使用し、IP ブロックに接続する他のモジュールを示すために“クライアント”という単語を使用します。
RTLのモジュールインタフェースは単なるポートリストです。モジュールが相互に接続されるとき、ツールはちょっとしたチェック、例えば全てのポートが接続されたか、接続される信号のデータ型、幅が正しいかどうかなどを行うに過ぎません。VHDLとSystemVerilogはより強力な型と型チェックの概念をもっており、モジュール定義から切り離してインタフェースを定義できる機能をもっています。それによってポートリストを書く退屈な作業を幾分やわらげてくれます。しかし最終的な解析では、ポートが信号に正しく接続されていることを確実にするためにセマンティックのチェックのみが実行されます。

問題の中心はインタフェースが特定の信号の低レベルなセマンティック以外にビヘイビアのセマンティックをもっていないことです。全てのモジュールに対して、設計者はインタフェースをまたがるビヘイビアの仕様を実現するために、一から始めます。例えば、FIFOブロックを設計するとき、設計者は以下のことを決めるでしょう：

- “エンキュー”操作には3つのポートが含まれる：インプットデータバス、FIFOがフルのときにエンキューを実行させないためのENQ_READY出力信号、実際にクライアントがデータをエンキューすることを合図するENQ_ENABLE入力信号。
- 同様の“デキュー”操作には3つのポートが含まれる：出力データバス、FIFOがエンプティのときにデキューを実行させないためのDEQ_READY出力信号、実際にクライアントがデータをデキューすることを合図するDEQ_ENABLE入力信号。
- FIFOをエンプティにする“クリア”操作のためのCLR_ENABLE入力信号

モジュールのポートを決めてしまった後、設計者は次のような単体の操作のルールを含むクライアントによる正しい動作のための規約に取り組むでしょう。

1. ENQ_ENABLE は ENQ_READY が false のときには決してアサートされない（下記のルールⅠとルールⅡを除いて）
2. DEQ_ENABLE は DEQ_READY が false のときには決してアサートされない
3. ENQ_ENABLE インプットデータバスで有効なデータがあるときには同時にアサートされなければならない

規約は以下のような複数のルールももっているでしょう。

Ⅰ ENQ_READY が false であっても同時に DEQ_ENABLE がアサートされるなら、ENQ_ENABLE がアサートされることができます（なぜなら、FIFO がフルであってもデキューされるデータの空き領域に新しいデータをエンキューすることができるからです）。

Ⅱ もし CLR_ENABLE がアサートされると、ENQ_ENABLE と DEQ_ENABLE が無視される（対応する READY 信号は false ですが、それらはアサートされることができます）

このような“規約”のルールは、通常はFIFOのデータシートにおいて文字や波形で非公式に指定されます。それらはしばしば不完全もしくはあいまいに記述されています。ときにはそれらは明らかに間違っている場合があります。例えば文字で書かれた仕様が実際のインプリメントに対応していないなど。少なくとも1つのこのような大手のIPベンダのFIFOのデータシートで、私たちはこれらのルールがいくつものページの文章にちりばめられているのを見つけました。上述のFIFOのような単純で分かりやすいブロックでさえ、ビヘイビアの規約や複雑になる場合があります。より大きなブロックやサブシステムでは本当に管理できなくなるでしょう。

上述の設計者の仕事は、各サブルーチンにおいて、どの引数がどのレジスタに渡されるのか、どの引数がメモリに渡されどのようにメモリ内に配置されるのか、などプログラマが慎重にプロトコルを設計していた古き日のコンピュータプログラミングを思い出させます。サブルーチンを使用する他のプログラマは（適切に文書化されていると仮定すると）、このプロトコルを注意深く理解し、それに応じてコードを書いていました。さらに悪いことに、レジスタの使用のサブルーチン設計者の選択は、すなわち、サブルーチン使用者自身のレジスタ割り当ての方向性に影響していました。言い換えると、もしサブルーチンが機能的に等価で、引数やレジスタ仕様の異なるサブルーチンに置き換えられると、サブルーチンユーザはそれに従って自分のコードを書き換える必要がありました。このメソドロジの欠如は、簡単に組み立てできず、簡単に拡張できませんでした。従って、もちろんこれがバグや壊れやすくリユースが制限されることの主要因でした。今日、高級言語とコンパイラによってこれらの全ての問題は、プログラマの心配事ではなくなり、決してバグにならず、そしてベリフィケーションを必要としません。コンパイラが構築することによって正しいコードをシステマチックに生成します。

RTLを使用するハードウェア設計者が、今日、同種の問題に直面しています。全てのIPブロックを使用するために、設計者は（完全に文書化されていると仮定すると）モジュール設計者によるポートプロトコルを理解する必要があります。モジュールのポートのビヘイビアはモジュールの設計者が一から作ったものであるため、IPユーザは全てのIP設計者ごとに異なるスタイルや慣習に合わせる必要があります。すなわち、各IPブロックのポートのビヘイビアを理解することは、完全に応用の効かない任務なのです。上記のⅠのようなルールはクライアントモジュールの制御回路に影響するため、同じポートリストや機能をもち規約の若干異なるIPブロックに変更した場合にはもちろんクライアントに変更を必要とします。

これがベリフィケーションにおける拡張性の欠如の主な要因です。IPモジュールAを使用したモジュールBの設計者は、Bの機能（設計者の注力すべきこと）のみを考えれば良いというわけではなく、BがAポートのビヘイビアの規約に違反する可能性についても考える必要があります。設計者はこのような違反が起こらないことを確実にするためのベリフィケーションテストを考える必要があります。これは、例えば何千ものFIFOが設計の中にある場合でも、全てのAのインスタンスについて起こりえます。そしてこれら数千の各インスタンスに対してのテストを個別に考え出し、実行する必要があります。さらに同じコーナーケースのために、各インスタンスは、このような状況を駆動するために個別に設計されたテストが必要です。

言うまでもなく、このような骨の折れるブロックやサブシステムのテストを採用する忍耐強さをもつエンジニアはほとんどおらず、彼らは最終的な“システムレベル”ベリフィケーションに問題を持ち越します。残念ながら、システムレベルまで来ると、深く組み込まれたIPブロックのコーナーケースを突き止めるのはより困難になります。

このベリフィケーションに対する責任の複雑さは、サブブロックからブロックを、サブシステムからシステムを構築する、それ自身の複合物で、すなわち、このメソドロジの欠如が正しく機能しません。そのため、今日の全体のシステムのベリフィケーションは悪夢で、全てのコーナーケースをカバーするのは不可能です。

要約すると、ベリフィケーションとIPリユースは難しく、大きなシステムへの困難さは増加しています。なぜならインタフェースビヘイビアのための良いフォーマルセマンティックモデルの欠如が以下のことを引き起こしています：

- リユースできるメソドロジがない：
  - IP設計者は、白紙の状態から始め、各モジュールのための専用のインタフェースプロトコルを作ります。IPユーザは、各IP特有のインタフェースプロトコルを一から理解することから始めます。
  - ビヘイビアを文書化する標準的な方法がなく、その結果、欠けている、不正確、不完全、あいまい、混乱させることになるか、ドキュメント化することを困難にします。
- あらゆるモジュールの全てのインスタンスにおいて繰り返される複雑なベリフィケーションの義務

より高いフォーマルセマンティックの世界では、代わりに、各IPブロックがそれ自身によって検証され、いかなる状況においても間違った使用方法が起きないことを保障します。重要性は、異なる状況におけるインスタンスでの多くのベリフィケーションの代わりに、ブロックを個別に徹底してベリフィケーションすることへとシフトしています。

※注意：
この分析は、科学計算のアルゴリズムからハードウェア作るプロセスを自動化する“ビヘイビア合成”が使用するアイデアとは全く異なります。このようなツールは通常、個々のブロックに制限され、個々のブロックを組み合わせて複雑なシステムを構成することには対応しません。ビヘイビア合成ツールが生成したブロックは、同様のインタフェースやリユースの問題をもっています。実際のところ、この資料で紹介しているルールベースのセマンティックを使用するソリューションは、ビヘイビア合成ツールによって精製されたハードウェアのターゲットとして適切化もしれません。

3. 関連する研究による部分的なソリューション

いくつかのツールや業界において数々の、部分的なソリューションがあります。

SystemVerilogでは設計者がインタフェースにタスクを定義することができます。このようなタスクは（FIFOのエンキューやデキューのような）インタフェース操作全体を表すことができ、全ての特定のポート信号のプロトコルをカプセル化することができます。クライアントモジュールは直接ポートの信号を発生する代わりにタスクを実行することができます。これによって設計者はモジュールの一部として、インタフェース自身[SV 2005]にトランザクションのビヘイビアを定義することができます。しかしながら、これらはポートリストの上部に被せるだけで、どこで呼び出ささされてもタスクを印ライン化することを記述します。すなわちいかなるインタフェースのビヘイビアの基礎的な、新しいフォーマルなモデルも導入していません。それらはまた、ポートが共有された場合の脆弱性ももっています。すなわち、タスクが複数の並列なプロセスから呼び出されたときに関して、必要なマルチプレックスやアービトレーションを取り扱う簡単な方法を提供しません。

SPIRITコンソーシアム[SPIRIT]はIPブロックのXMLベースの記述方法を使用することによって、IPリユースのための標準を定義することを試みています。しかし、再度、これらは単なる標準のRTLとRTLのポートリストの上のレイヤに過ぎません。根本的に改良するビヘイビアのフォーマルモデルは導入していません。

SystemCは、信号というよりトランザクションとしてインタフェースを記述するためにC++からメソッドのコンセプトを使用します。しかしながら、合成可能なSystemCは、未だにモジュール間の通信にRTLのような信号に頼っています。しかしモデリング目的でさえ、合成の可能性には興味がなくC++のメソッドは単に制限のないメカニズムを与えているだけでセマンティックやメソドロジではありません。この高位の概念をスタティックなベリフィケーションに有益に活用するツールを見出すことは困難です。

もっとも有望な部分的ソリューションは、ポートプロトコルの正確なコンディションを記述するためにインタフェースにPSL[PSL 2005]やSystemVerilogアサーション(SVA)[SVA 2005]もしくはOVL[AccelleraOVL 2005]のアサーションを使用することです[Foster 2004, SynopsysABV 2003]。このアイデアは、上記の高位の宣言的なロジックベースの言語をインタフェースに相互作用のフォーマルな正確性のプロパティを記述するために、インタフェースにイミディエートとテンポラルアサーションを付けます。

しかしながら、アサーションは以下の理由で、未だに部分的なソリューションに過ぎません。

- IPブロック設計者やベリフィケーションエンジニアが、ポートプロトコルの規約を正確に完全に特徴付けるために必要十分なアサーションを作る負荷が取り除かれていない。このレベルの正確性を達成するのは、単純なブロックでさえもとても困難です。設計者は、彼の意図を漏れのないフォーマルなロジックの記述に翻訳することが困難であると思うことがあります。
- 現在は、アサーションはシミュレーションによってチェックされます。全体のシミュレーション速度の影響を加味すると、完全なカバレッジを確実にするために内部のIPブロックのコーナーケースを駆動するテストベンチを記述するのは、未だに困難です。すなわち、全てのコーナーケースを捕らえるために十分にアサーションが活躍するのでしょうか？
- 将来、アサーションは定理立証を使用し、ますますスタティックにチェックされるようになり、カバレッジが完全になりシミュレーション速度の影響を受けなくなるでしょう。しかしながら、アサーションはとても強力な言語であるため、このゴールにたどり着くのは遅く、何年も要するでしょう。このテクノロジが利用可能であると仮定しても、このアサーションを追加する（手作業の）仕事は常に時間を浪費し、エラーが起こりやすく、不完全である傾向があるでしょう。
  質問：一般的なアサーションに比べ、これらの仕事に適切でより扱いやすい、スタイルと範囲が限定されたアサーションがあるでしょうか？以下に提唱するルールベースのセマンティックがその答えです。

まとめ：
多くの部分的なステップがベリフィケーションのボトルネックを軽減する方向に向かっている中、適切な高位の、組織だった、フォーマルなベリフィケーションセマンティックの欠如が、進歩を妨げています。

4. ルールとルールベースのインタフェースが正確性の問題に自動的に柔軟に対処します

ルールベースのシステムは、コンピュータサイエンスにおいては、セマンティック、パラレル性、コンカレンス性などの研究において長い歴史をもっています。ルールの元になった項書き換え系の豊かな理論体系があります[Baader & Nipkow 1998, TRS 2003]。ルールベースのシステムは人工知能の研究[Winston 1992]や複雑な並列プログラミングの研究[Chandy & Misra 1998]においての中心となります。これらの全ての仕事は、ベリフィケーションの中心的な目的である正確性に焦点を置いています。

正確性に関するルールの力は、アトミック性の特性によるものです[Lynch at.el. 1993]。ルールは複雑な動的コンディションにおける、複雑で動的に決定されるステートの遷移を記述することができます。

それにも関わらず、アトミック性によって設計者は一度に1つのルールの正確性を判断することができます。各ルールにおいて、システムは一貫したステートで始まることを仮定すると、ルールのステート遷移をしてもシステムに矛盾がないかどうかをチェックすることができます。- アトミック性によって並列に動作する他のルールを気にする必要がなくなります。個々のルールの正確性が確立されたら、それらのルールを組み合わせたものは自動的に正確です。これによって大きく、複雑なシステムへの拡張性をもたらします。

アトミック性のコンセプトは、ソフトウェア分野（OS、データベース、分散システム）において複雑な並列性を管理することにおける基礎的なブレークスルーとなりました。Bluespec SystemVerilog [BSV 2004] (BSV)は同様のパワーをHDL（ハードウェア記述言語）にもたらします。これは、並列なビヘイビアを単純に、正確にそしてフォーマルに指定できるフォーマルセマンティックモデルで、コンストラクションによって複雑なハードウェアシステムを正確に作り上げる拡張可能なメソドロジです（すなわち、ベリフィケーション作業を桁違いに削減します）。

セクション2のFIFOの例では、ポートリストの代わりにBSVは下記のようなメソッドを使用したインタフェースを定義します。

```
interface FIFOBuf#(x_t);

    method Action enq (x_t x);
    method ActionValue#(x_t) deq ();
    methoc Action clear ();
endinterface
```

enqメソッドはセクション2で述べた全てのポート、入力データバス（データ幅はx_tでインスタンス化された型に依存します）、ENQ_READY出力信号、ENQ_ENABLE入力信号をカプセル化しています。同様にdeqメソッドは出力データバス、DEQ_READY出力信号、DEQ_ENABLE入力信号をカプセル化します。

一般に、メソッドの引数は、モジュールの入力データバスポートになります。メソッドの結果（deqによる戻り値のような）は出力データバスポートになります。戻り値の型は複数のフィールドやベクトルをもつ構造体で良いため、メソッドは複数のデータバスポートをもつことができます。全てのメソッドは、原理上、READY出力信号をもちます。ActionとActionValueメソッドは、原理上、ENABLE入力信号をもちます。ActionとActionValueメソッドはシーケンシャル、すなわちモジュール内部のステートの変化の原因となります。これらの種類の3つ目のメソッドvalueメソッドは、純粋な組み合わせ回路で、その結果は、その引数と内部モジュールのステートの組み合わせ関数です。もしそのモジュールが常にアサートされるなら、コンピュータはREADYとENABLE信号を取り去って最適化するでしょう。

FIFOを使用するクライアントモジュールは、以下のようなenqとdeqメソッドを操作するルールを含みます。

```
module mkClient (...);

    ... instantiate fifo ...
    rule upstream (... cond1 ...);
        ... other actions ...
        fifo.enq (expr1);
    endrule
    rule downstream (... cond2 ...);
        x <- fifo.deq ();
        ... other actions ...

    endrule
```

各ルールは、上記でcond1やcond2として表されるような明示的なコンディションをもっています。これは純粋な組み合わせブーリアンの表現です。各ルールはルールコンディションがTrueになったときだけに自動的に実行される1つかそれ以上のアクションを含みます。例えば、upstreamルールはexpr1で表される値をFIFOにエンキューするアクションを含み、downstreamルールがFIFOからアイテムxをデキューするアクションを含みます。

ルールによって操作される全てのメソッドのコンディションは、ルールの全体にわたるコンディションに組み込まれます。例えばENQ_READY信号はupstreamルールの全体を決定するcond1と論理和されます。DEQ_READY信号はdownstreamルール全体を決定するcond2と論理和されます。

ルールは、その全てのコンディションが許可されるときのみにファイア（実行）されます。それがファイアするとき、全てのアクションや、アクションが操作する全てのメソッドを含み、合成されたアトミックなアクションとして同時に実行されます。したがってupstreamルールはENQ_READYがtrueのときだけファイアすることができ、エンキューはルールの全体的なアトミックなアクションの一部となります。ルールがファイアされるとき、エンキューされるデータが駆動され、ENQ_ENABLEがアサートされます。

メソッドやルールのコンディションを満たす必要がありますが、ルールがファイアされるにはそれだけでは十分ではありません。具体的には、ルールはリソース（上記のFIFOなど）をシェアできるので、アトミック性を維持するためには同時にファイアすることが可能ではないかもしれません。すなわち、同時のファイアは矛盾するステートを引き起こします。Bluespecコンパイラは、アトミック性を維持する場合にのみ同時にファイアすることを確実にするスケジューリングロジックを生成します。

FIFOのインプリメントをコンパイルしているとき、コンパイラはどのようなコンディションであればenqとdeqメソッドが安全に同時に実行できるかどうか、を推測するシステマチックな解析を実行します。異なるFIFO設計ではこのような同時の操作が許可されるか、許可されないか注意してください。

このインタフェースの情報はFIFOのインプリメントと共にコンパイラによって記録されます。そして、mkClientがコンパイルさ

れるとき、mkClientに、upstreamとdownstreamルールが安全に実行できるコンディションのときだけ同時にファイアされることを保障する適切な制御ロジックを導入するためにこの情報を使用します。

これらの理由で、私たちはインタフェースメソッドがルールベースであると言っています。すなわち、メソッドは単純にルールの一部であり、ルールはメソッドを操作するものの合成としてみることができます。特に、コンパイラはセクション2で述べた（1）（2）（3）（I）（II）規約要件を自動的に推測します。さらに、コンパイラはコンストラクションによってFIFOがmkClientとしていかにインスタンス化されても全ての規約要件を満たすことを確実にします。それゆえmkClientがFIFOを矛盾のあるステートで駆動するこおとは不可能です。同様にまた、数百ものFIFOモジュールをインスタンス化するmkClientを設計しても、コンパイラはいかなるケースでもFIFOの規約を成立させることを確実にします。

さらに、この確実さは本当にmkClientをブラックボックスとみなすことができるくらい推移的です。このインタフェースの規約はまた、コンパイラによって自動的に推測されます。もしmkClientが自分自身を繰り返しインスタンス化するなら、コンパイラは各ケースで規約を満たすことを確実にします。したがって、どのような環境においてもFIFOが矛盾のあるステートに決して駆動されないことを保障します。言い換えると、mkClientのインタフェース規約は、FIFOの規約も満たすことを確実にします。

それゆえ、ルールとルールベースのインタフェースはベリフィケーションの問題を劇的に単純化します。RTL設計におけるごく一般的な機能的なエラーの全ての原因は、信号エラー、信号サンプリングエラー、競合するコンディションなどです。これらのほとんどはアトミック性の失敗としてトレースでき、ビヘイビアがルールとルールベースのインタフェースの項で記述されていれば、コンストラクションによって削除することができます。まとめると、利点は以下のとおりです。

- ルールベースのインタフェースメソッドは、生のポートに比べ抽象度が高い
- Action、ActionValue、valueメソッドは、モジュールの相互作用と信号のために簡単で共通する語彙を提供する。各モジュールのためにそのモジュール用の信号のプロトコルを作る代わりに、接続先が既成のメカニズムの内の一つを利用する。同様に、もしインタフェースがこれらの項として記述されていれば、IPブロックのユーザは即座に信号プロトコルを理解することができる。Bluespecを使用すると、設計者は一回きりのタイミングダイアグラムを作ったり、解釈するような無駄な作業から開放される。
- ルールとルールベースのインタフェースであるため、コンパイラはIPブロックのインタフェースで指定された信号やスケジューリング規約を自動的に推測する。そしてコンパイラは、取り囲むコードを生成するときに全てのIPブロックのインスタンスで規約を満たすことを自動的に確実なものにする。この確実性が全ての階層のモジュールに適用されるため、全てのシステムはより強く安定した基礎の上に立つ、強固な大きなモジュールとなる。

このアプローチによって、設計者は大きなシステムを複雑な並列性をもったまま迅速に組み合わせることが可能になります。それによって、確実にインスタンス化されたことを確認するために行うコーナーケースへの注力を削減します。代わりに、より大きく、重要な問題であるシステム全体にわたるアーキテクチャや機能に焦点を当てます。

経験と実証：
この資料で示したこのアイデアはBluespec SystemVerilogのシステムやコンパイラで実現されており、何年にもわたって使用されてきました。Bluespecの顧客や大学のパートナーにおいて、ASICとFPGAの両方で、数千ゲートから2千万ゲートほどのサイズにわたる百件以上の実績があります。現在、劇的に少ないベリフィケーション作業、高い設計生産性、IPリユースに関する私たちの主張の裏づけとなる多数の実証済みの証拠があります。また、ルールとルールベースのインタフェースを使用してデザインを記述すると、RTLを直接手書きするのに比べて、チップ面積や速度などのパフォーマンスの犠牲がないことに関しても多数の証拠があります。

まとめと結論

RTLの生のポートリストでは、インタフェースセマンティックはとても脆弱です（実際のところインタフェースセマンティックが存在しない）。セマンティックの合成の続く弱点が“ベリフィケーションボトルネック”の根底にあります。より大きく複雑なシステムを構築すると、システムの中の弱い“ジョイント”があるため、益々きしみやすくなります。

RTLにおいては、IPブロックレベルに注力してベリフィケーションを行っても、適切な使用の元での正確性が確立されるに過ぎません。しかし、この適切な使用は将来のインスタンス化を仮定しません。IPブロックの適切なビヘイビアは、内部ロジックと同様に外部ロジックにも依存しています。ブロックがインスタンス化されるときに多くのエラーが発生するため、IPブロックレベルのベリフィケーションへの注力に対する利益は小さくなり、システムレベルのベリフィケーションではなく潜在的なコーナーケースのベリフィケーションにばかり注力しています。

強力なフォーマルなインタフェースセマンティックは、したがってボトルネックを排除するために必要です。アサーションはよい一歩ですが、制限があります。－ アサーションを追加するのは時間を要し、エラーが発生しやすく、不完全になりやすい傾向があります。また、アサーションのチェックはシミュレーションのオーバヘッドを追加し、カバレッジを確実にするのは困難です。そしてそれらの一般的なアサーションのフォーマルなベリフィケーションは困難です。

ルールとルールベースのインタフェースは、ビヘイビアと相互作用の強力で高位のセマンティックモデルを提供します。それらは、モジュールのインタフェースを設計し、テストするための高位の語彙とシンプルな知的モデルを提供します。コンパイラは自動的にモジュールのインタフェースのための規約を自動的に生成し、モジュールのインスタンスを正しい使用方法で生成するために、これらの情報を使用します。これらの全ては、直接RTLコードを書くのに比べてパフォーマンスの劣化がありません。その結果、大きなシステム設計におけるエラーとバグの主因がコンストラクションによって削減されます。

リファレンス

[AccelleraOVL 2005] Open Verification Library (OVL) 1.0, Accellera, http://www.accellera.org/activities/ovl/, 2005.

[Baader&Nipkow1998] Term Rewriting and All That, F. Baader and T. Nipkow, Cambridge Univ. Press, 1998, 300pp.

[BSV 2004] Bluespec System Verilog Reference Guide, www.bluespec.com, 2004-2006

[Chandy&Misra1998] Parallel Program Design: a Foundation, K. Mani Chandy and J. Misra, Addison Wesley, 1998, 516pp.

[Design By Contract] Design by Contract, Wikipedia, http://en.wikipedia.org/wiki/Design_by_contract. (See also Eiffel Software, http://www.eiffel.com)

[Foster 2004] Assertion-Based Design, 2nd Ed., H. D. Foster, A. C. Krolnik and D. J. Lacey, Springer, 2004, 414 pp.

[Lynch et. al. 1993] Atomic Transactions: In Concurrent and Distributed Systems, N. A.Lynch, M. Merritt, W. E. Weihl and A. Fekete, Morgan Kaufman Series in Data Management Systems, 1993, 476 pp.

[PSL 2005] PSL—Property Specification Language, IEEE Std 1850, http://standards.ieee.org, September 2005

[SPIRIT] The SPIRIT Consortium for industry level cooperation in developing standards for IP description, http://www.spiritconsortium.org

[SV 2005] SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800- 2005, http://standards.ieee.org, November, 2005

[SynopsysABV 2003] Assertion-Based Verification, Synopsys, Inc., http://www.synopsys.com/products/simulation/assertion_based_wp.pdf, March 2003, 14 pp.

[TRS2003] Term Rewriting Systems, Terese, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2003, 884 pp.

[Winston1992] Artificial Intelligence, Third Edition, P. H. Winston, Addison Wesley, 1992, 691pp.

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/