2022
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
JAPAN

プロジェクトの現場で使われ始めた Accellera標準のPSS

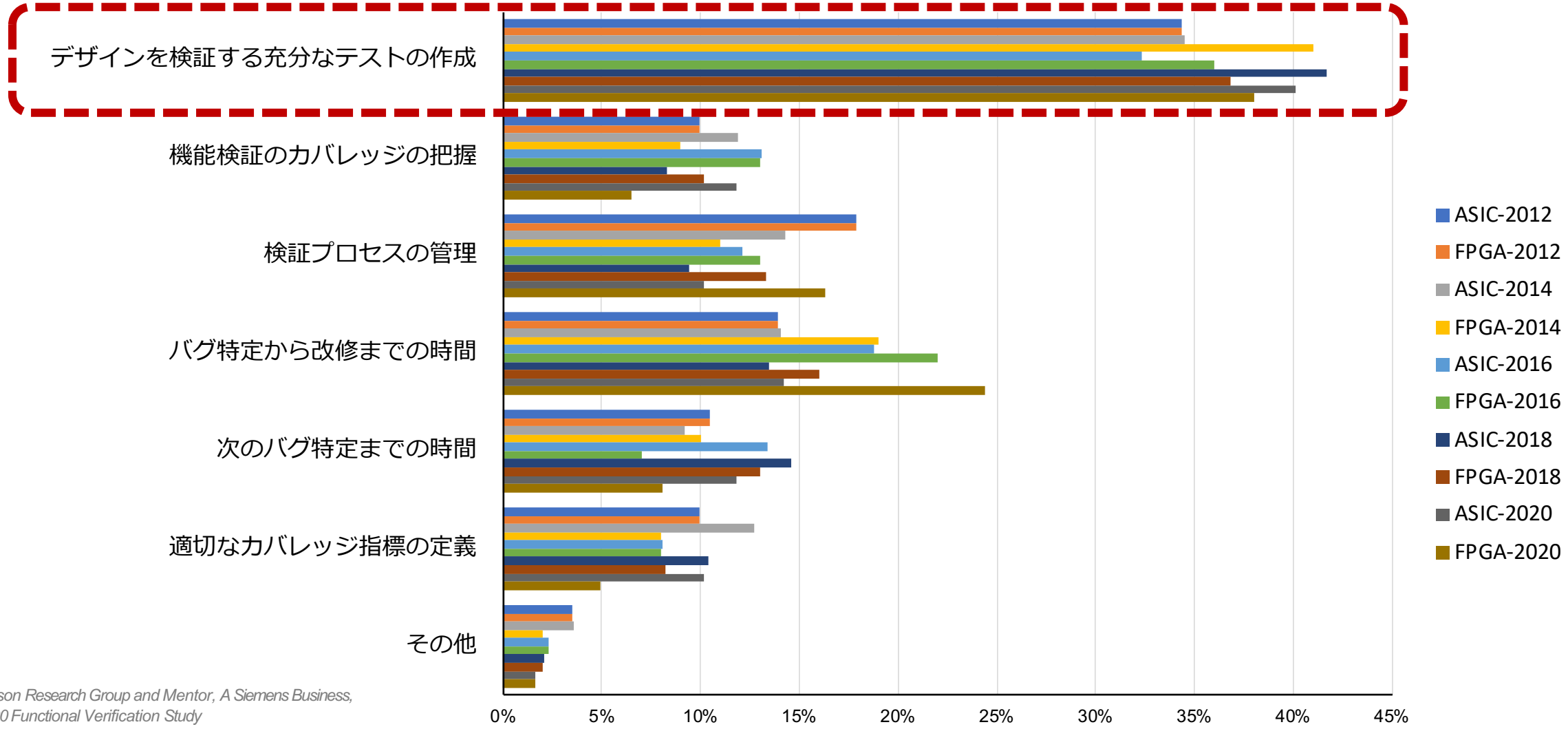
DVCon Japan 実行委員会
細川博司・三橋明城男

accellera
SYSTEMS INITIATIVE

アジェンダ

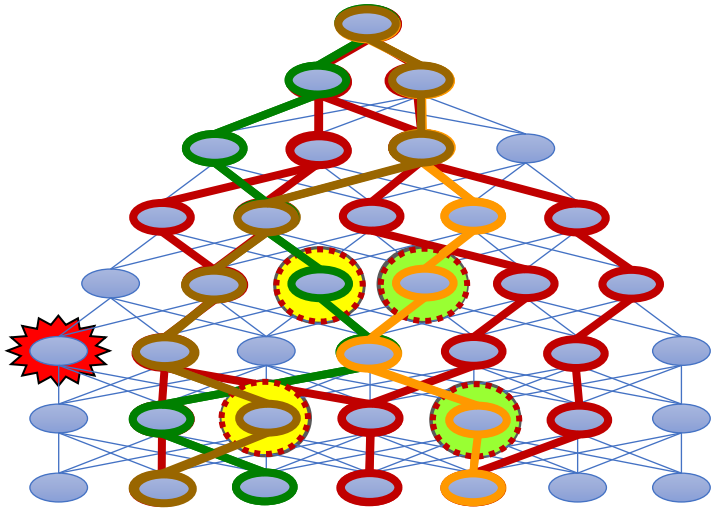
- PSSの概要と現在の状況
- ディスプレイコントローラの適用例
- メモリ＆キャッシュの適用例
- SoCレベルの適用例
- まとめ

機能検証における最大の課題

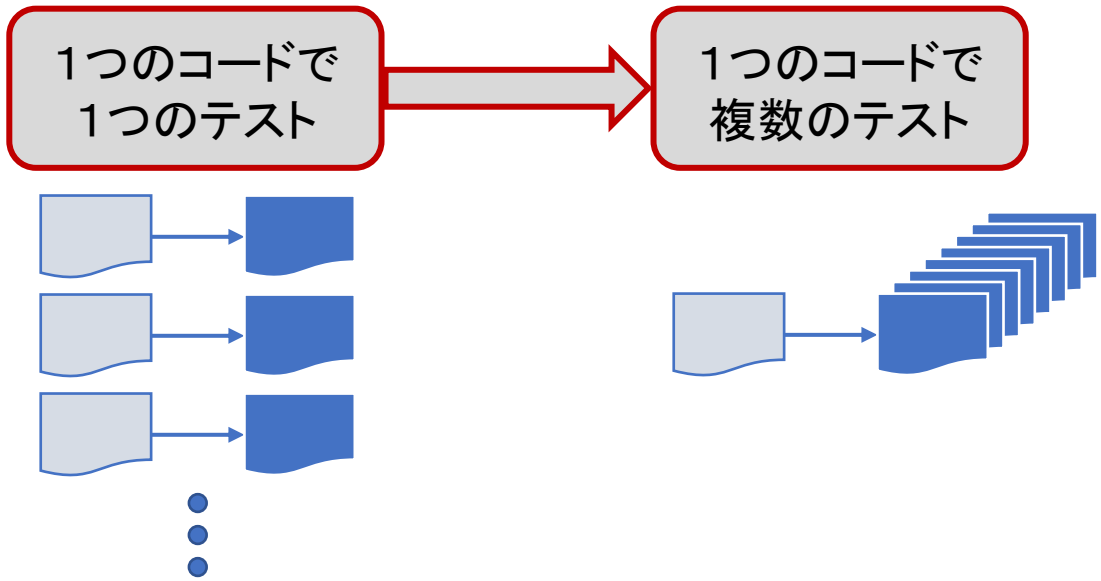


Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study

メソドロジのシフトには新たな発想が必要

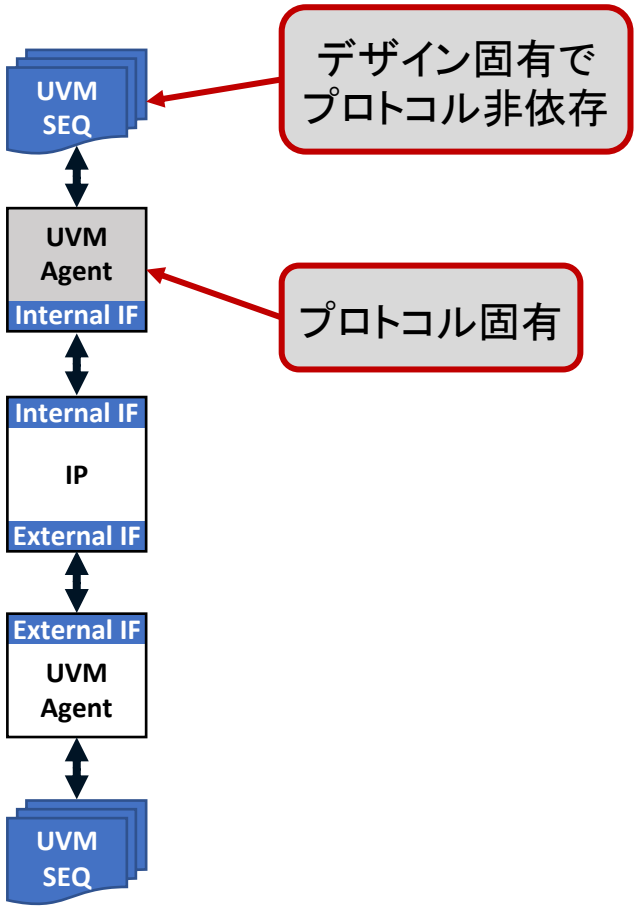


- SystemVerilogは検証に新たなアプローチをもたらした
  - 他の独自言語からの機能を標準言語で実現
  - ディレクテッドテスト → 制約付きランダムテスト



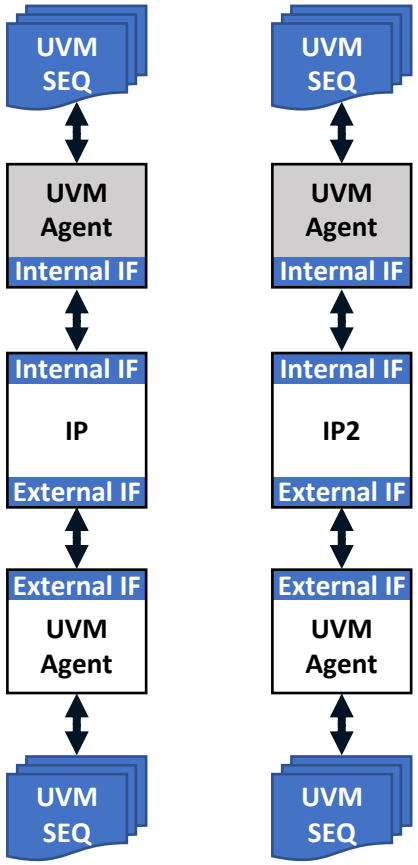
- 制約付きランダムには何が起きたかを知る機能カバレッジが必要
- エコシステムの醸成には詳細な仕様が必要

UVMは “How” - どう検証するかが焦点



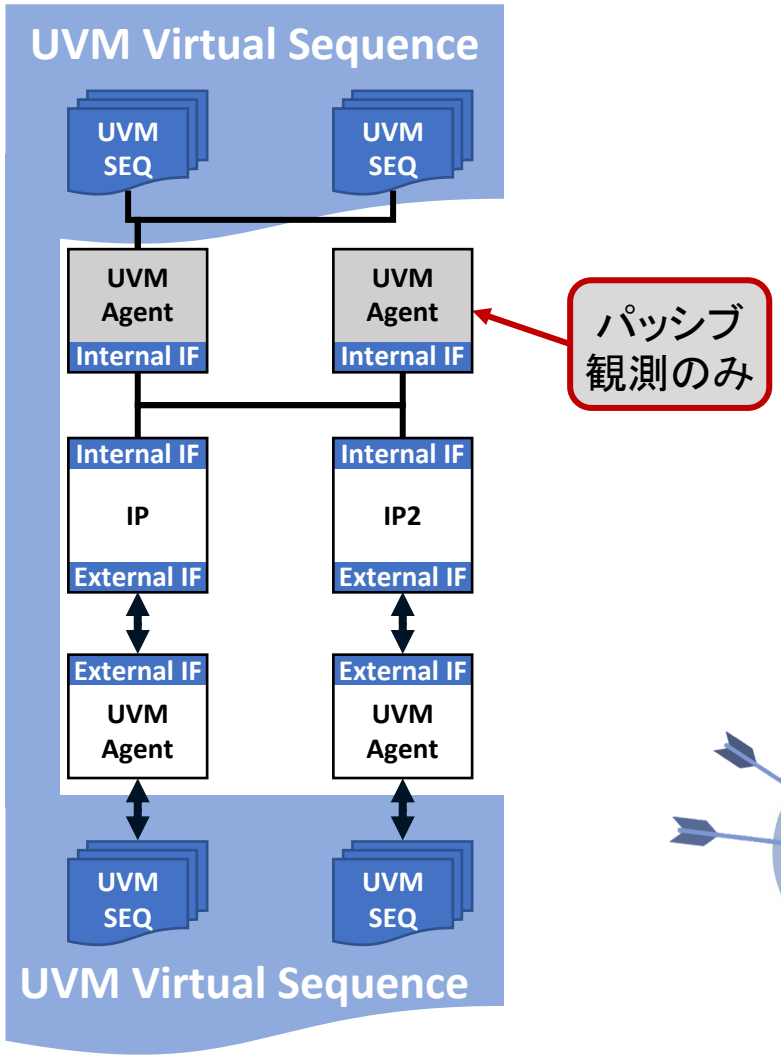
- ブロックレベル検証に最適
- モジュラー化され再利用可能な検証コンポーネント
- What と How の分離
  - 構成可能なトランザクションレベルのシーケンス
  - ドライバ／モニタがトランザクションと信号間を変換
- 抽象的なシーケンスを異なるエージェントに適用
  - シーケンスは “What”
  - エージェントはプロトコル固有の “How”
- IPのインタフェースは同じアプローチを用いる

UVMは “How” - どう検証するかが焦点

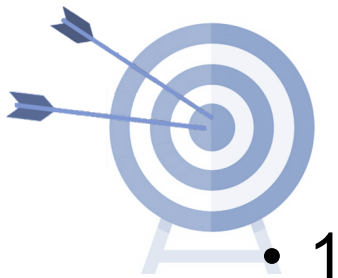


- すべてのテストやテストベンチは共通の基本構造を持つ
- 水平方向の再利用
  - 同じインタフェースを持つ異なるブロックで共通のエージェントが使用できる

UVMは “How” - どう検証するかが焦点

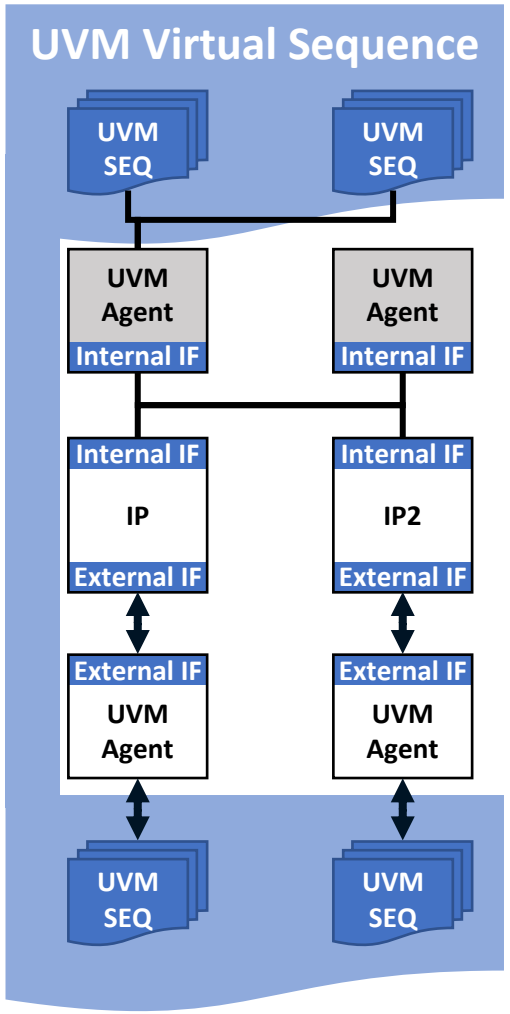


- すべてのテストやテストベンチは共通の基本構造を持つ
- 水平方向の再利用
  - 同じインタフェースを持つ異なるブロックで共通のエージェントが使用できる
  - ブロックレベルの環境をコンフィギュレーションし、コンポーネントを再利用する
  - ブロック用のテストはVirtual Sequenceから呼出される

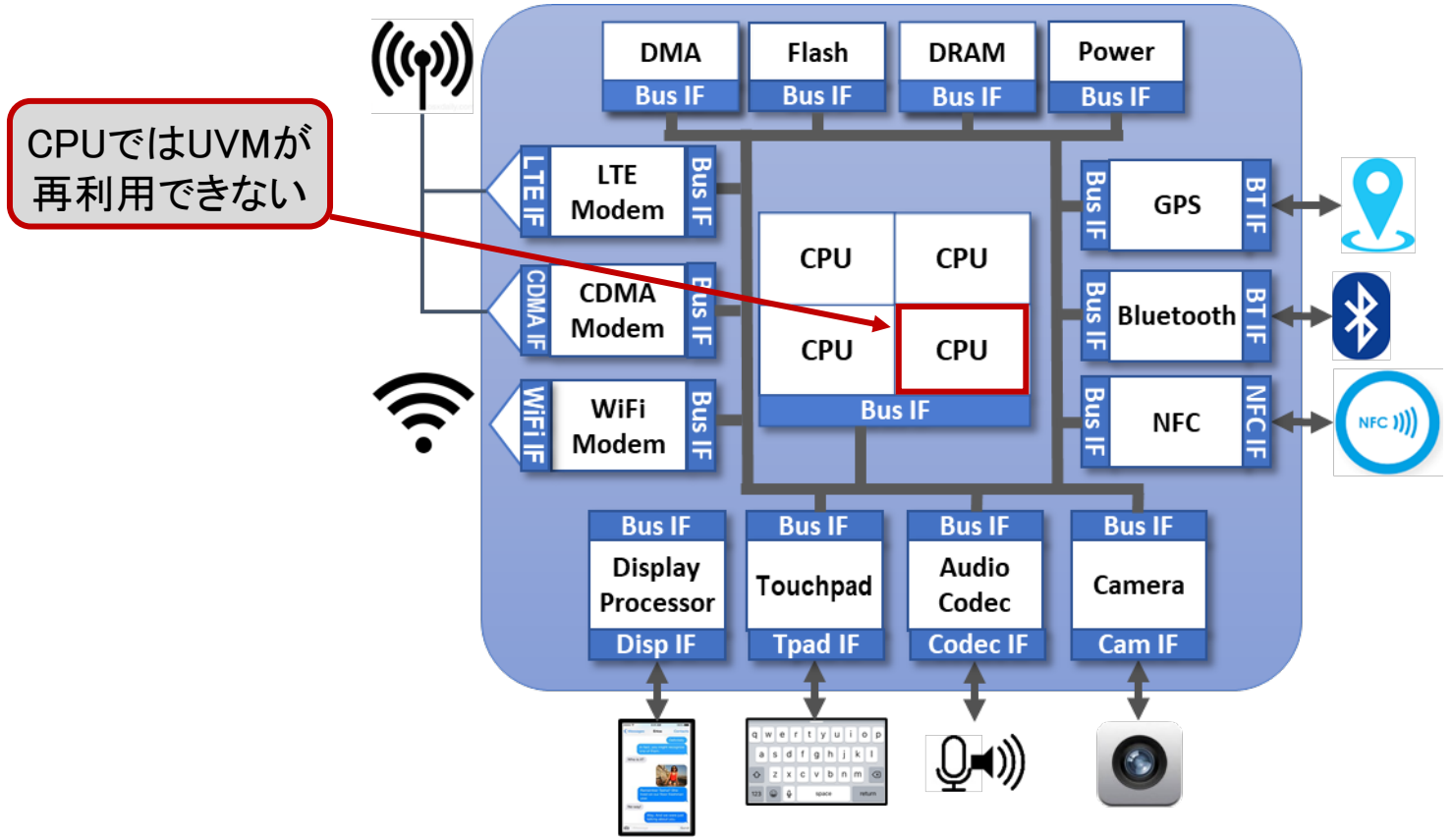


- 10年前はこれが正しいターゲットだった

ターゲットが変わるのが検証



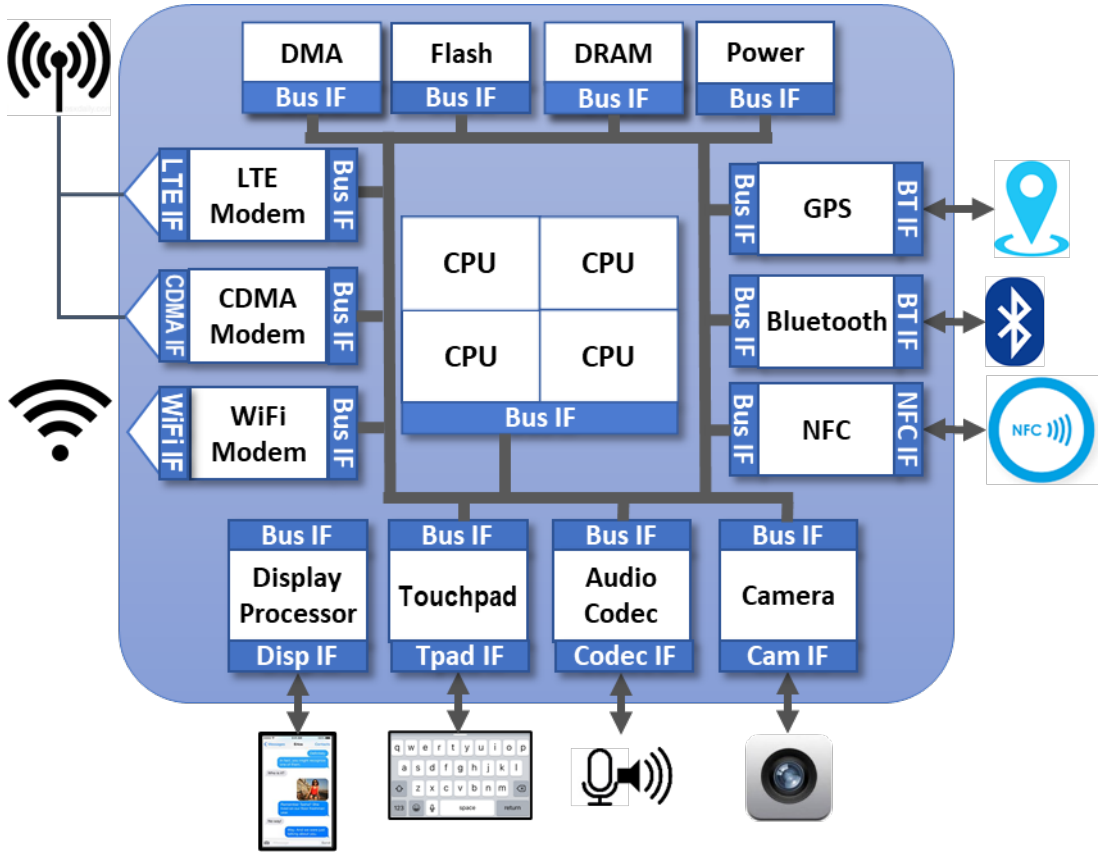
- ブロックレベルのテストはSoCにスケールしない



- ASM:C == Gate:RTL == UVM:?

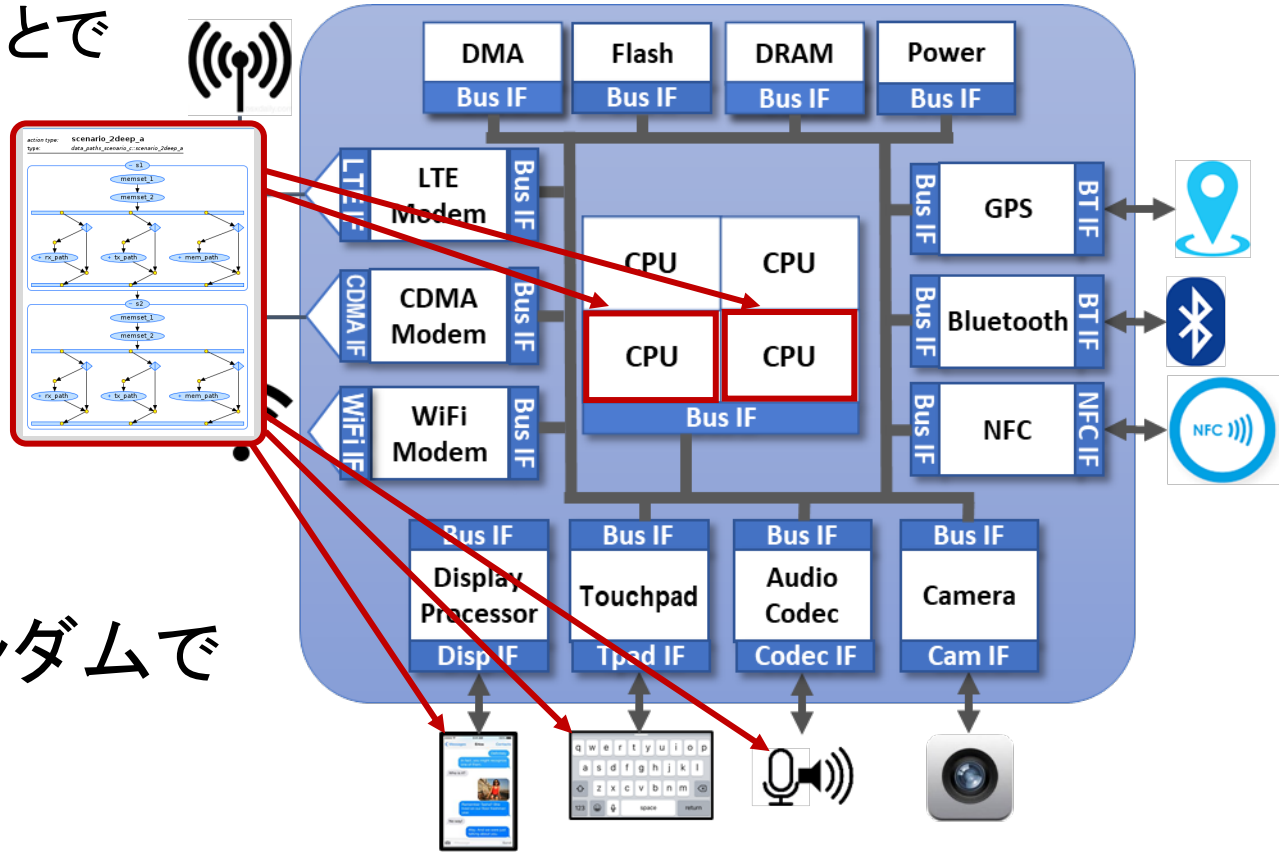
組込みプロセッサを含むSoCの検証

- 通常はマニュアルでCコードを記述
- ランダムによるカバレッジ改善は不可
- マニュアルコードのIPライブラリ
  - IPの再利用は困難
- テスト空間を手続的にモデル化する必要がある
  - ビデオデータは複数ソースから来る
  - DMAが Nチャネルある
  - グラフィクスは使用前にパワーアップ
  - パワーマネジメントのシーケンス



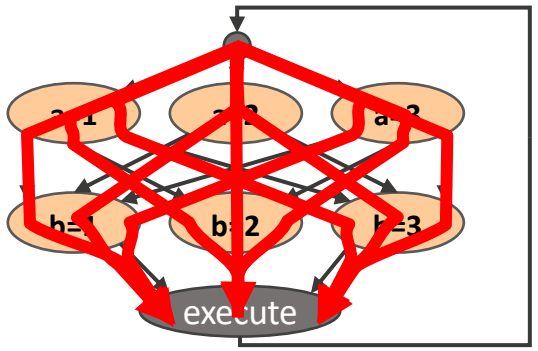
SoC検証に対して自動化が適用できたら

- 1つの定義指定 – 複数のテスト
  - テスト空間をフォーマル定義することで ツールがシステム制約を満たし ながらテストを生成可能にする
- 1つの定義指定からテスト分散
  - プロセッサでインテントを再利用
- パーティショニングと調整を 自動化
- シナリオレベルでの制約付きランダムで バグを特定

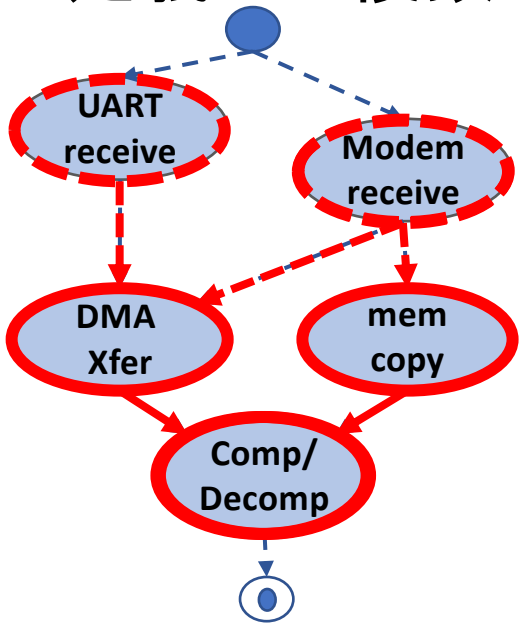


より高位におけるスティミュラス

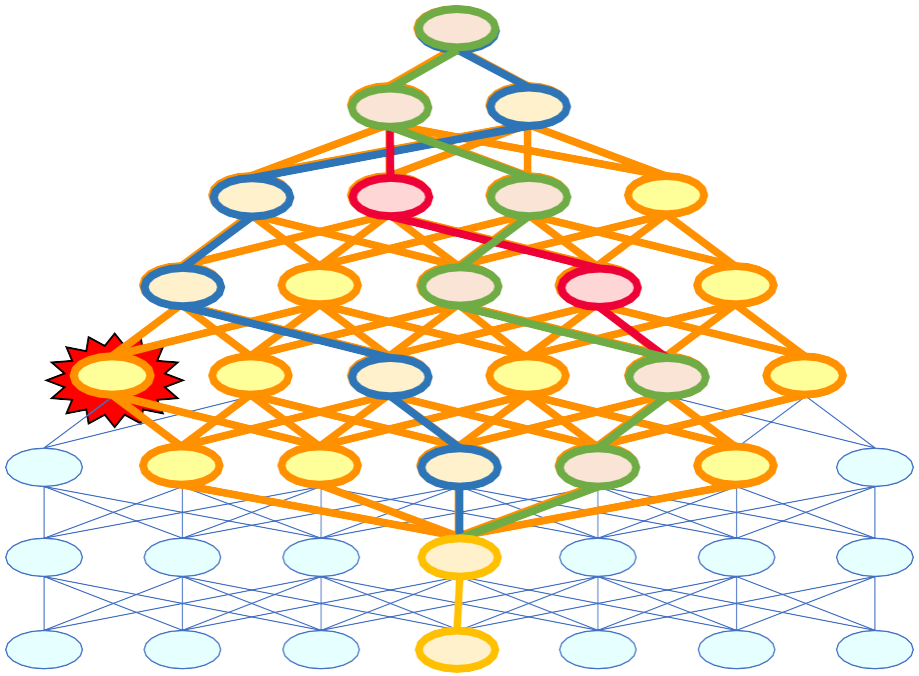
- UVM Sequenceはトランザクションのセットを定義
  - トランザクション内容をランダム化
  - トランザクションのフローは明示的にランダム化できない
- シーケンス間におけるランダム化が難しい



- シナリオは動作のセットを定義
  - 重要な検証インテントを定義
  - 重要なインテントをサポートするルールを定義
- 単純なPSSの定義から複数のシナリオが生成できる



PSSは制約付きランダムをシナリオ生成に適用



- PSSの部分的な指定で重要な検証インテントを定義
  - コーディングして期待する必要がない
  - SVの機能カバレッジより直感的かつ直接的
- ルールによりツールが追加アクションを挿入可能
- シナリオをランダム生成を可能に
  - 各シナリオはリーガルであることが保証される
  - 特定のアクションを制約
  - アクション間のスケジュール関係を制約

Portable Stimulusのキーポイント

テストインテントを
キャプチャ

部分シナリオを
定義

構成可能な
シナリオ

テスト空間の
フォーマルな表現

テストを
自動生成

複数の実装を
ターゲットに

分離されたテストインテント

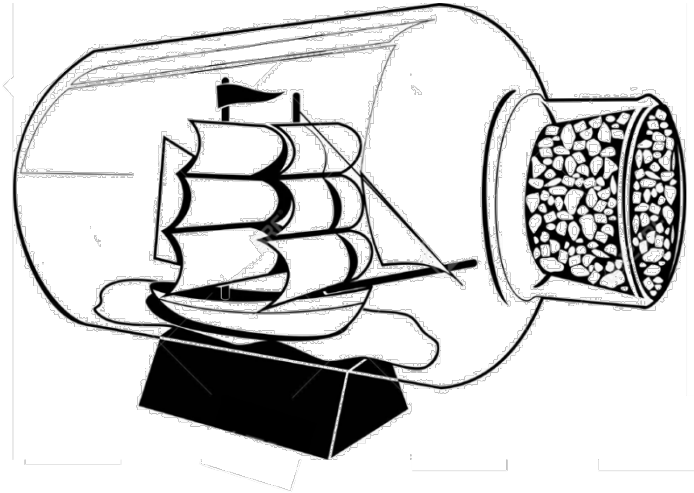
独立したテスト実装

検証プロセスにわたって
高いカバレッジを実現するテストを
より少ない労力で生成

Portable Stimulus Model とは？

抽象化
モデル

What =
何を行うか



リアライ
ゼーション
レイヤー

How =
行う内容を
どう実現するか

伝えたいこと

重要な動作をアクティビティで定義する

- 仕様の一部のみを定義することが可能
- ツールはシナリオを完成させるために他の要素を追加

モデルのその他の部分では動作がどのように相互作用するかを定義する

- インスタンス化されたコンポーネントが利用可能なアクションを定義
- フロー・オブジェクトのバインディングにより推論の選択肢が制約される
- 利用可能なリソースによりスケジューリングの選択肢が制約を受ける

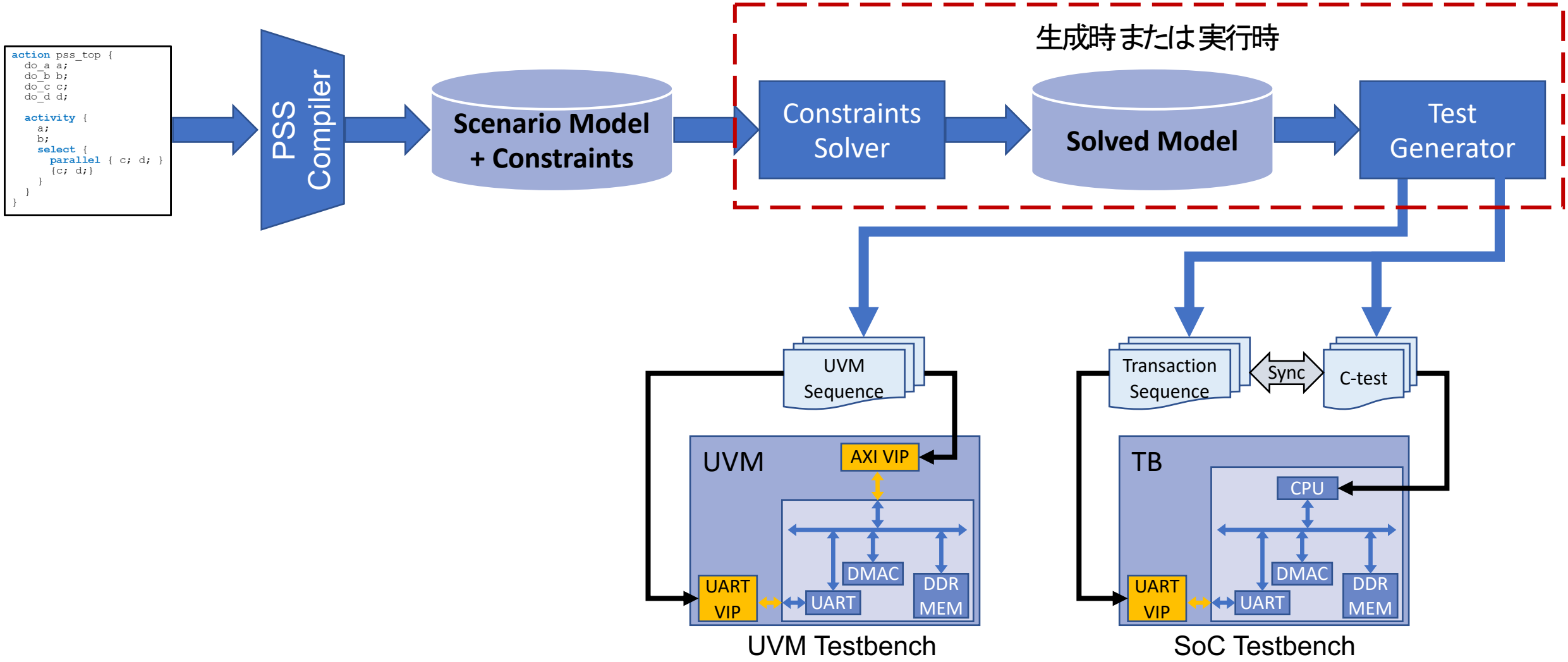
タイヤが道路に出会うように・・・

抽象化モデルは異なるターゲット上で実装できなくてはならない

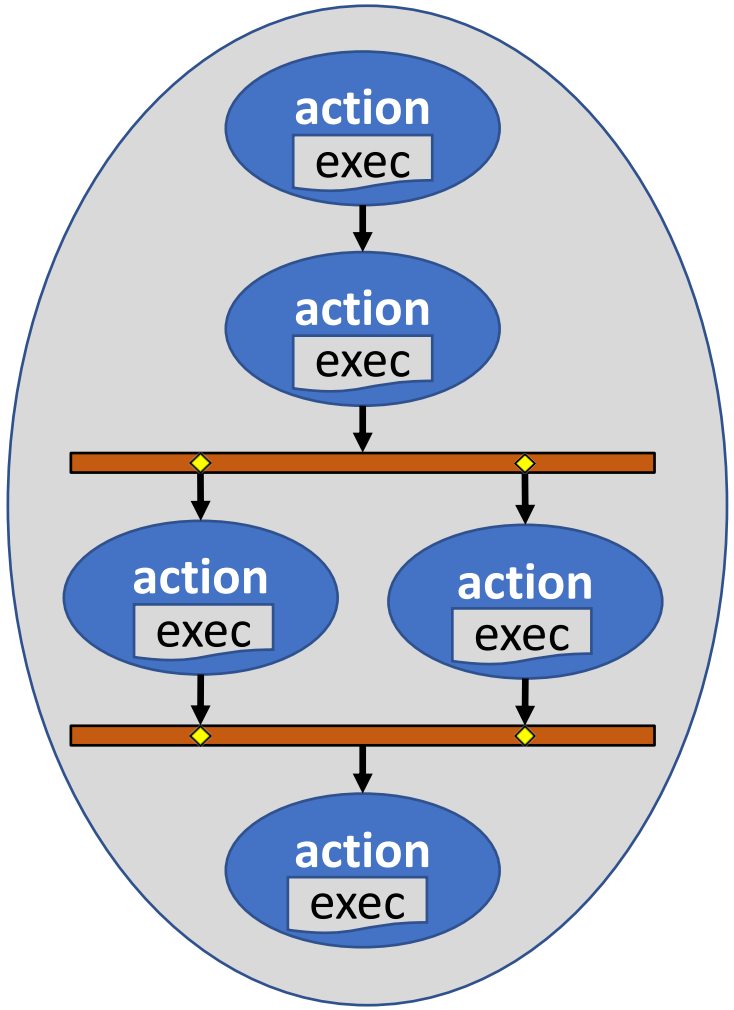
不可分なアクションをターゲット上のコードとしてexecブロックにモデル化

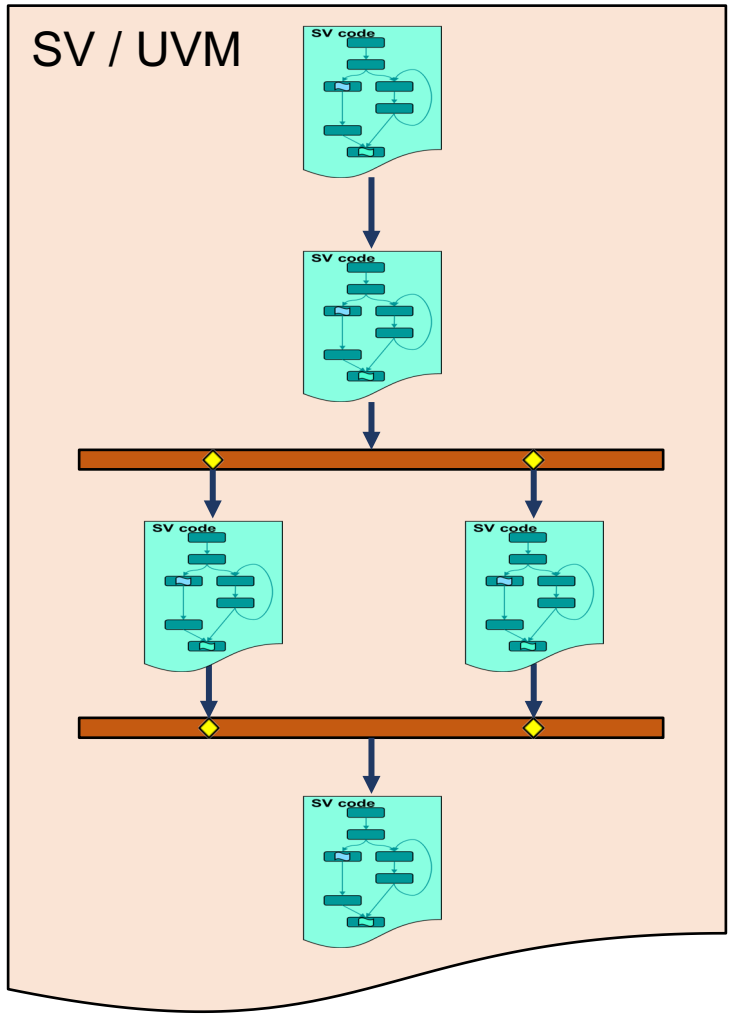
ツールはアクティビティのスケジュールに従いターゲットコードを構築する

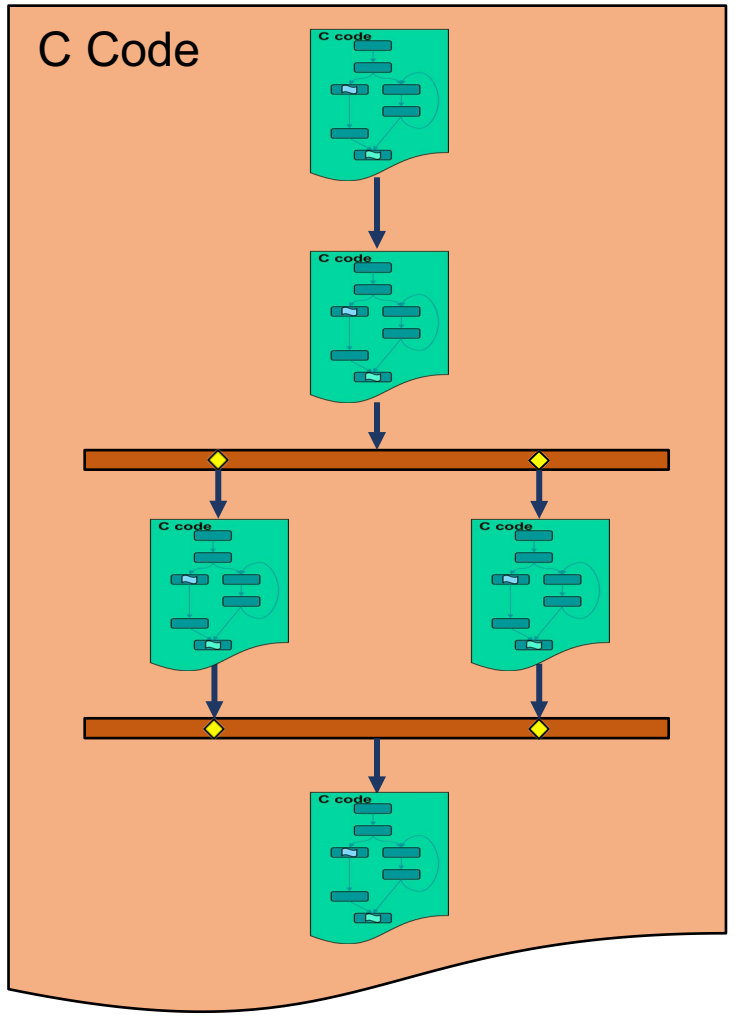
一般的なPSSフロー


action pss_top {
do_a a;
do_b b;
do_c c;
do_d d;
activity {
a;
b;
select {
parallel { c; d; }
{c; d;}
}
}
}
PSS Compiler
Scenario Model + Constraints
生成時または実行時
Constraints Solver
Solved Model
Test Generator
UVM Sequence
UVM
AXI VIP
DMAC
UART VIP
UART
DDR MEM
UVM Testbench
Transaction Sequence
Sync
C-test
TB
CPU
DMAC
UART VIP
UART
DDR MEM
SoC Testbench

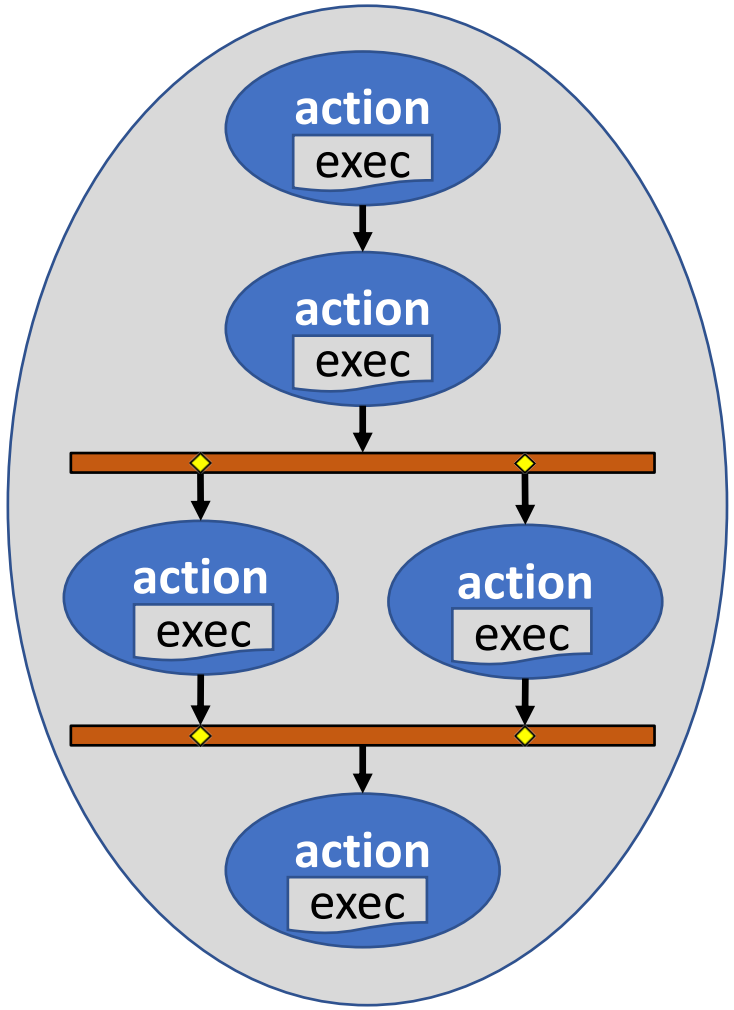
コード生成はアクティビティのスケジュールに従う

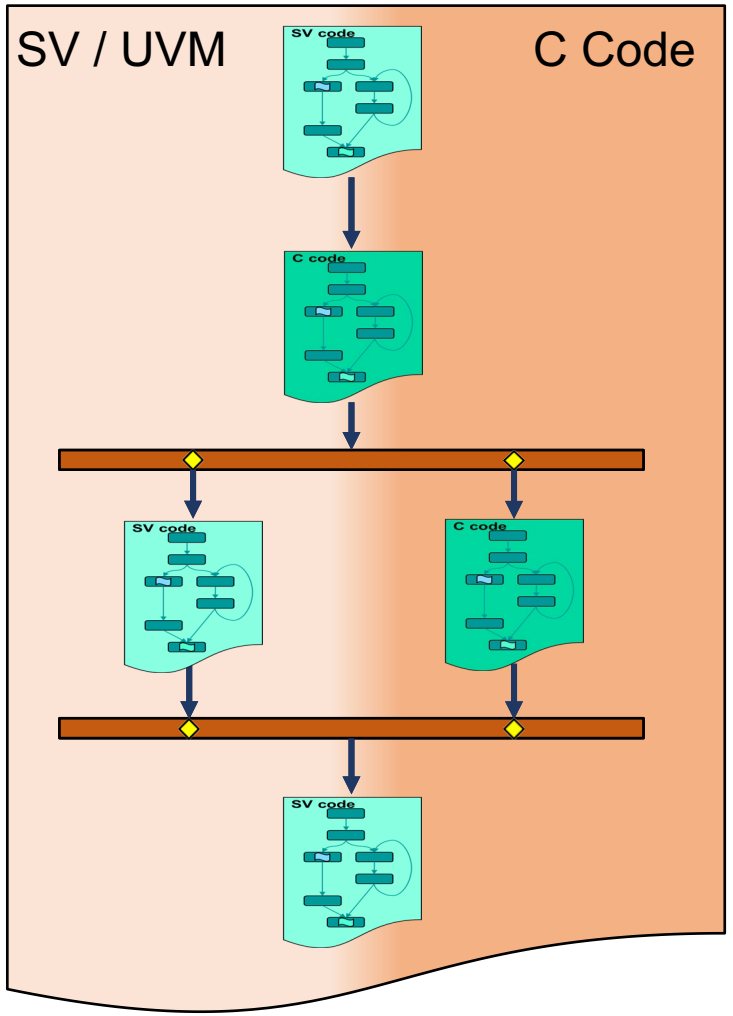






生成コードはアクティビティのスケジュールに従う





アジェンダ

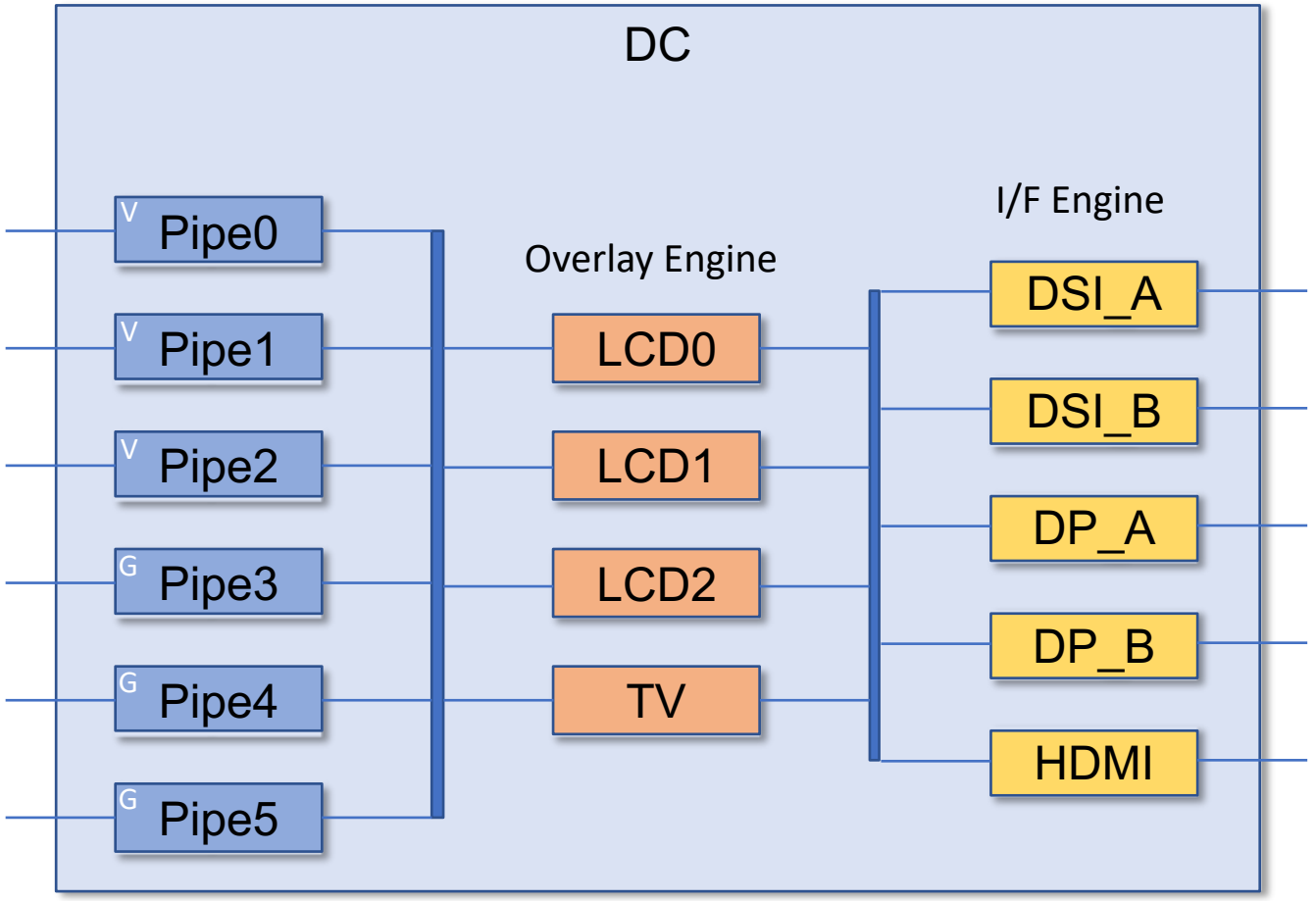
- PSSの概要と現在の状況
- ディスプレイコントローラの適用例
- メモリ＆キャッシュの適用例
- SoCレベルの適用例
- まとめ

概要

- このセクションの目的
  - 典型的なIP検証問題に対するPSSの適用を紹介
  - PSSのリソースプールとクレームの意味、使い方を解説
  - PSSによるシナリオのモデリングの主なメリットをSystemVerilogとの対比において解説
- このセッションの構成
  - 検証の課題 - ディスプレイコントローラにおけるデータパスの活性化
  - SystemVerilogによるシナリオ空間のモデリングとその限界
  - PSSによるシナリオ空間のモデリング – 一般的な解決策

ディスプレイコントローラの解説

- 処理用パイプx6
  - Pipe 0..2 はVideo用
  - Pipe 3..5 はGraphics用
- オーバーレイエンジンx4
  - LCD0, LCD1, LCD2, TV
- インタフェースエンジンx5
  - DSI_A, DSI_B, DP_A, DP_B, HDMI

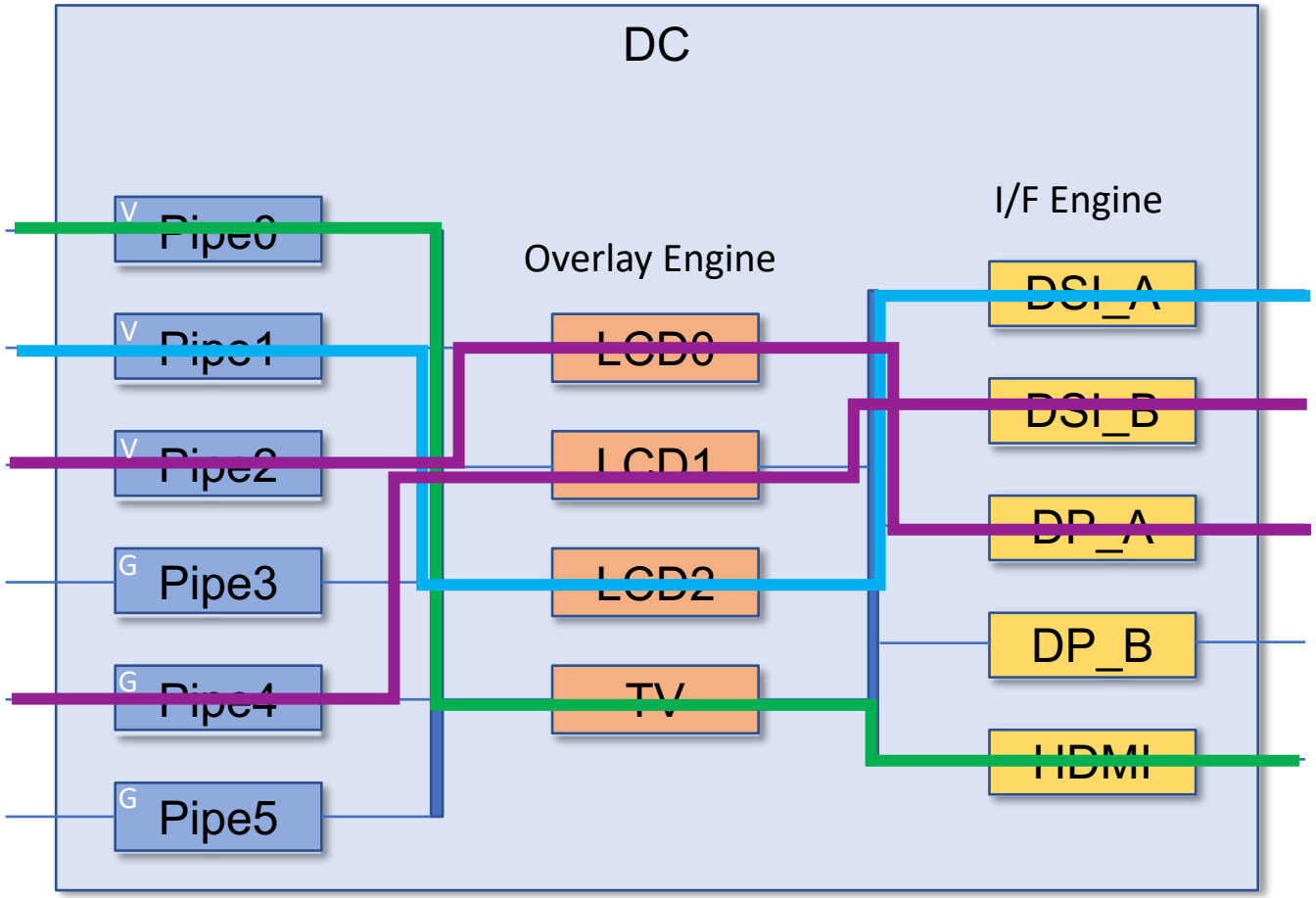


ディスプレイコントローラの解説（続き）

- 各エンジンは同時には1つのデータストリームのみを処理
  - 1..4のストリームを同時処理

- データパスのルール
  - Video Pipeはオーバーレイエンジン LCD1にデータ渡し不可
  - Graphics Pipeはオーバーレイエンジン LCD0にデータ渡し不可

  - LCD0はDP_Aのみにデータ渡し可
  - LCD1はDSI_A / DSI_Bにデータ渡し可
  - LCD2はDSI_A、DSI_B / DP_Bにデータ渡し可
  - TVはDP_AまたはHDMIにデータ渡し可

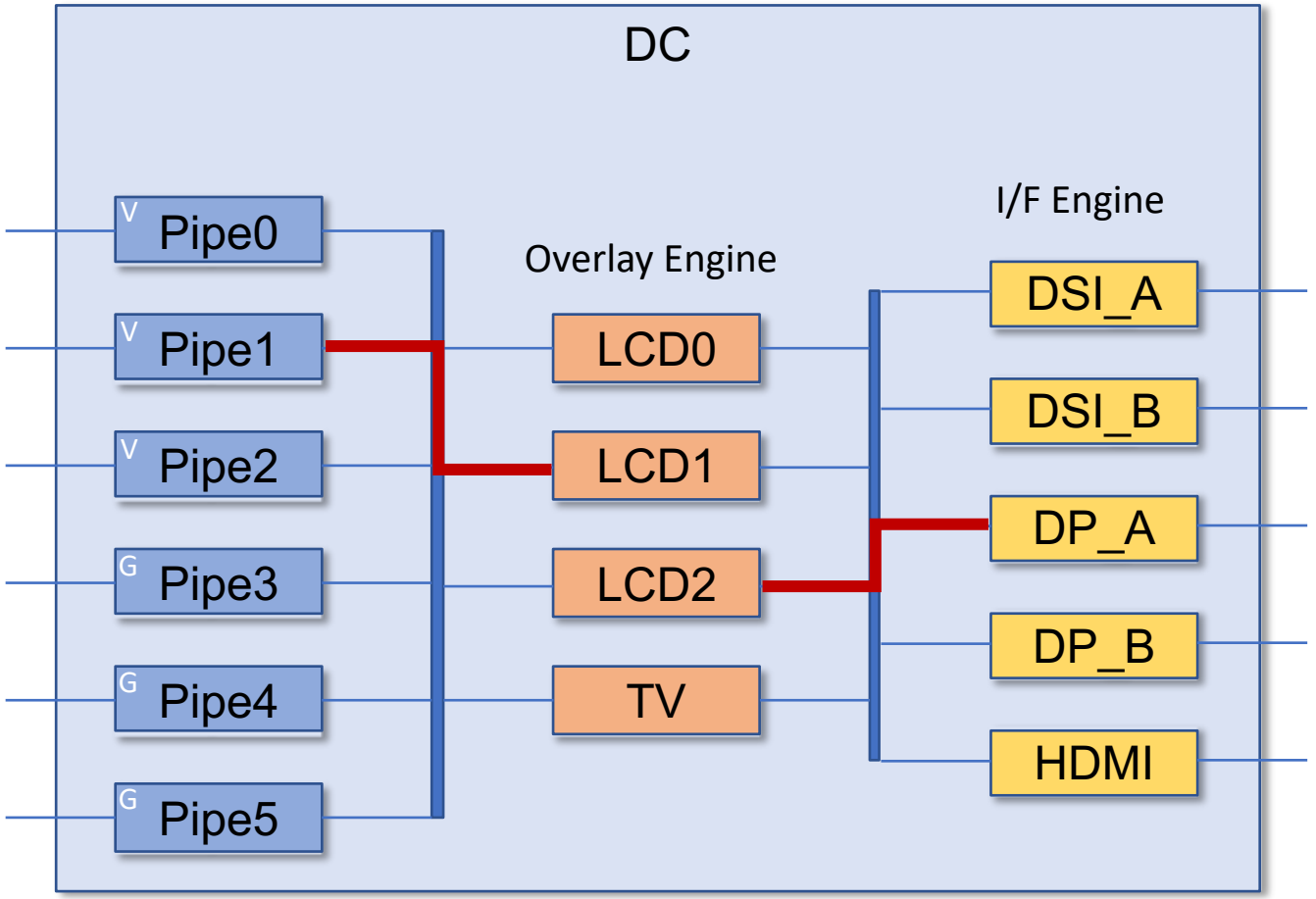


ディスプレイコントローラの解説（続き）

- 各エンジンは同時には1つのデータストリームのみを処理
  - 1..4のストリームを同時処理

- データパスのルール
  - Video Pipeはオーバーレイエンジン LCD1にデータ渡し不可
  - Graphics Pipeはオーバーレイエンジン LCD0にデータ渡し不可

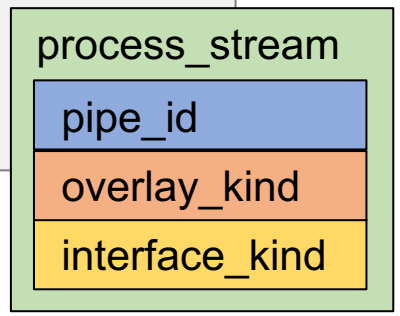
  - LCD0はDP_Aのみにデータ渡し可
  - LCD1はDSI_A / DSI_Bにデータ渡し可
  - LCD2はDSI_A、DSI_B / DP_Bにデータ渡し可
  - TVはDP_AまたはHDMIにデータ渡し可

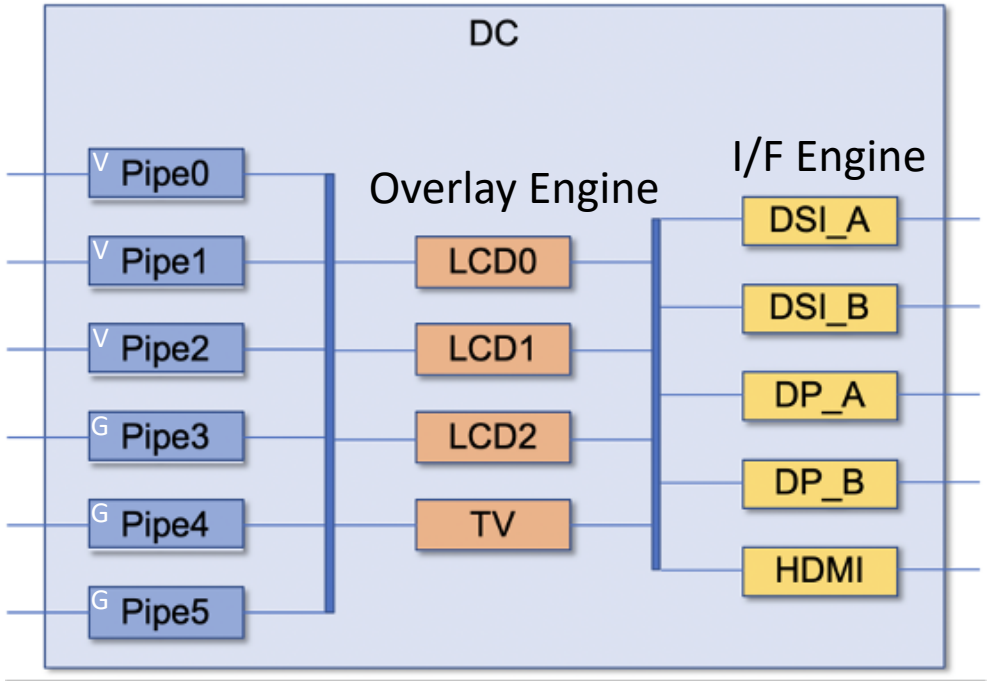


SVによるデータストリームのモデリング

```
typedef enum {VID, GFX} pipe_kind_e;
typedef enum {LCD0, LCD1, LCD2, TV} overlay_kind_e;
typedef enum {DSI_A, DSI_B, DP_A, DP_B, HDMI}
interface_kind_e;
class process_stream extends uvm_object;
  rand int pipe_id;
  rand overlay_kind_e overlay_kind;
  rand interface_kind_e interface_kind;

  rand pipe_kind_e pipe_kind;
  constraint pipe_kind_c {
    pipe_kind == VID -> pipe_id inside {0,1,2};
    pipe_kind == GFX -> pipe_id inside {3,4,5};
  }
  . . .
endclass;
```

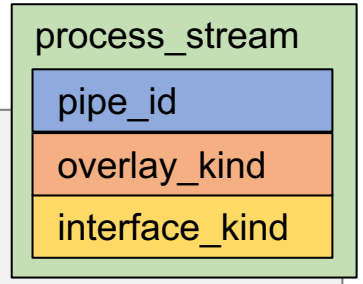
データパスに割り上てたストリームのフィールドをランダム宣言している





SVによるデータパスのルール設定

```
class process_stream extends uvm_object;
  . . .
  constraint pipe2overlay_c {

    pipe_kind == GFX -> overlay_kind != LCD0;
    pipe_kind == VID -> overlay_kind != LCD1;
  }
  constraint overlay2interface_c {
    overlay_kind == LCD0 -> interface_kind == DP_A;
    overlay_kind == LCD1 -> interface_kind inside {DSI_A, DSI_B};
    overlay_kind == LCD2 -> interface_kind inside {DSI_A, DSI_B, DP_B};
    overlay_kind == TV   -> interface_kind inside {DP_A, HDMI};
  }
  . . .
endclass;
```



- Graphics Pipeはオーバーレイエンジン LCD0にデータ渡し不可
- Video Pipeはオーバーレイエンジン LCD1にデータ渡し不可

- LCD0はDP_Aのみにデータ渡し可
- LCD1はDSI_AまたはDSI_Bにデータ渡し可
- LCD2はDSI_A、DSI_BまたはDP_Bにデータ渡し可
- TVはDP_AまたはHDMIにデータ渡し可

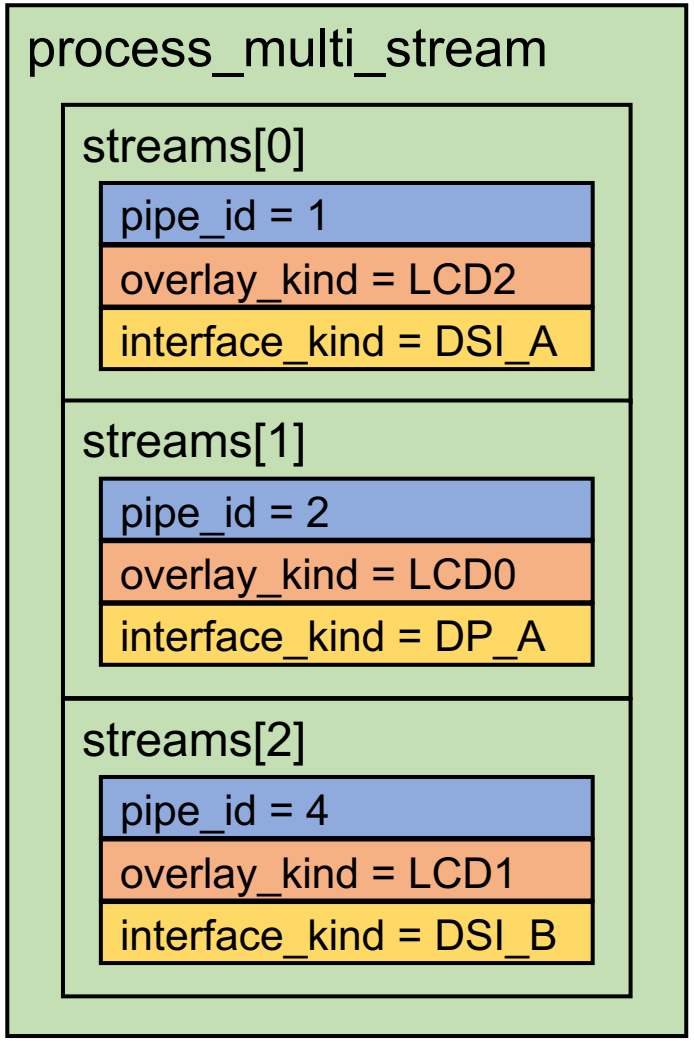
SVによる並列ストリームのモデリング

```
class process_multi_stream extends uvm_object;
  rand process_stream streams[];
  constraint num_streams {
    stream.size inside {[1:4]};
  }
  constraint resource_unique_c {
    foreach (streams[i]) {
      foreach (streams[j]) {
        if (i != j) {
          streams[i].pipe_id != streams[j].pipe_id;
          streams[i].overlay_kind != streams[j].overlay_kind;
          streams[i].interface_kind != streams[j].interface_kind;
        }
      }
    }
  }
. . .
endclass;
```

1～4の並列ストリーム数

pipe、overlay、interfaceはストリームごとに異なる種類でなくてはならない

SV/UVMコードのオーバーヘッド
-> コンストラクタ、配列アロケート、uvm_object_utils .....



どのように一般化するか？

- これまでに達成したこと
  - 複数の並列ストリーム間における制約関係のモデリング
  - しかしこの例に特化している
- まだ足りていないことは？
  - 他のストリーム処理が進む間に新たなストリームを開始／停止できること
  - 異なる機能／処理時間のタスクを行うエンジン間での負荷分散
  - ストリームのランダムな、または複雑なスケジュールの生成
  - . . .
- このような問題の本質はSystemVerilogでモデリングできない ---
  問題の本質：限られたリソースを奪い合うようなタスクの表現 .....

リソースをPSSでモデリング

```
component display_c {
  enum pipe_kind_e {VID, GFX};
  resource pipe_r {
    rand pipe_kind_e kind;
    constraint {
      kind == VID -> instance_id in [0..2];
      kind == GFX -> instance_id in [3..5];
    }
  }
  pool [6] pipe_r pipe_pool;
  enum overlay_kind_e {LCD0, LCD1, LCD2, TV};
  resource overlay_r {
    rand overlay_kind_e kind;
    constraint {instance_id == (int)knd;
  }
  pool [4] overlay_r overlay_pool;
  enum interface_kind_e {DSI_A, DSI_B, DP_A, DP_B, HDMI};
  resource interface_r {
    rand interface_kind_e kind;
    constraint instance_id == (int)kind;
  }
  pool [5] interface_r interface_pool;

}
```

resourceオブジェクトの
タイプで、アトリビュートや
制約を含むことが可能

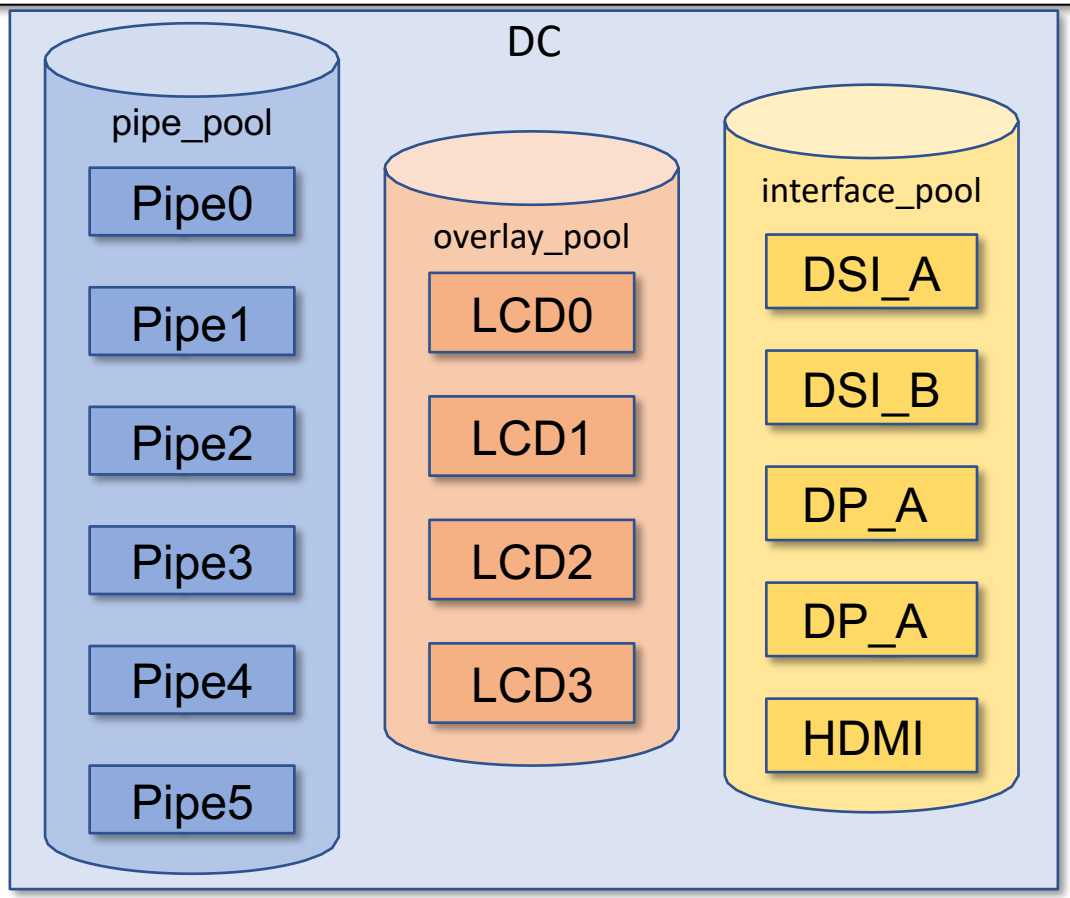
ビルトインアトリビュート
である instance_id はPool内
のインデックスを示す

Poolにはリソースタイプを N
インスタンス含むむことが可能

ユーザ定義の列挙型属性によ
りインスタンスに意味ある名
称が関連付けられる

15. Resource objects

Resource objects represent computational resources available in the execution environment that may be assigned to actions for the duration of their execution.



1つのデータストリームのモデリング

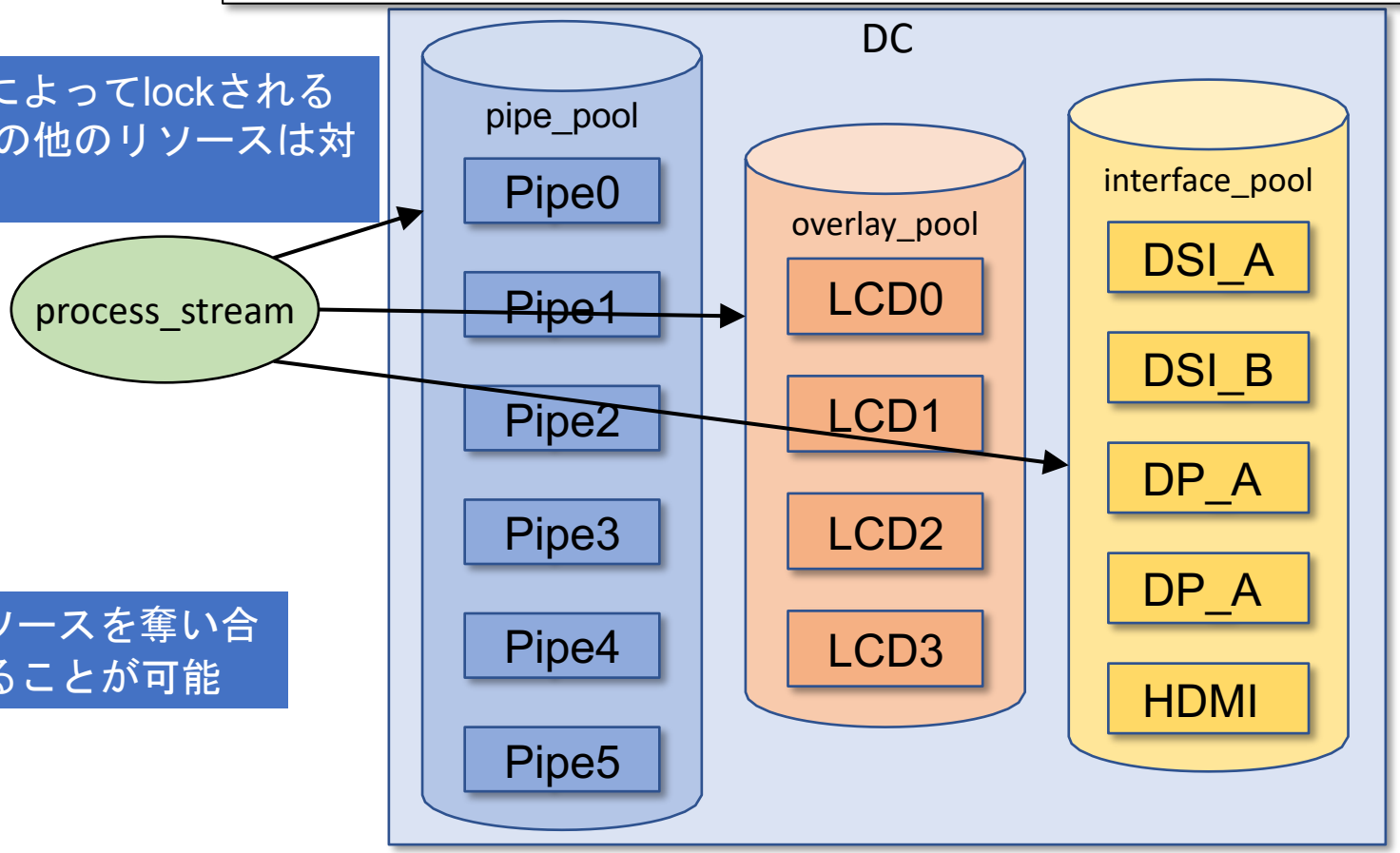
15. Resource objects

Resource objects represent computational resources available in the execution environment that may be assigned to actions for the duration of their execution.

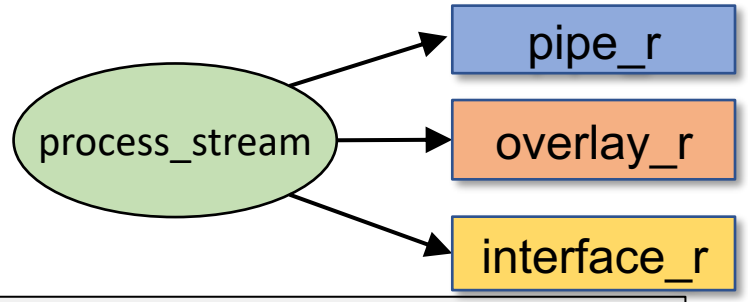
```
component display_c {
  action process_stream {
    lock pipe_r pipe;
    lock overlay_r overlay;
    lock interface_r interface;

    exec body {
       drive_stream(pipe.instance_id,
                    overlay.instance_id,
                    interface.instance_id);
    }
  }
}
```

resourceはactionによってlockされる – ただし並行使用の他のリソースは対象外

問題の本質 – 限られたリソースを奪い合うような振舞いを表現することが可能



データパスのルールを指定



- Graphics Pipeはオーバーレイエンジン LCD0にデータ渡し不可
- Video Pipeはオーバーレイエンジン LCD1にデータ渡し不可
- LCD0はDP_Aのみにデータ渡し可
- LCD1はDSI_AまたはDSI_Bにデータ渡し可
- LCD2はDSI_A、DSI_BまたはDP_Bにデータ渡し可
- TVはDP_AまたはHDMIにデータ渡し可

```
action process_stream {
  . . .
  constraint pipe2overlay_c {
    pipe.kind == GFX -> overlay.kind != LCD0;
    pipe.kind == VID -> overlay.kind != LCD1;
  }
  constraint overlay2interface_c {
    overlay.kind == LCD0 -> interface.kind == DP_A;
    overlay.kind == LCD1 -> interface.kind in [DSI_A, DSI_B];
    overlay.kind == LCD2 -> interface.kind in [DSI_A, DSI_B, DP_B];
    overlay.kind == TV   -> interface.kind in [DP_A, HDMI];
  }
}
```

SystemVerilog

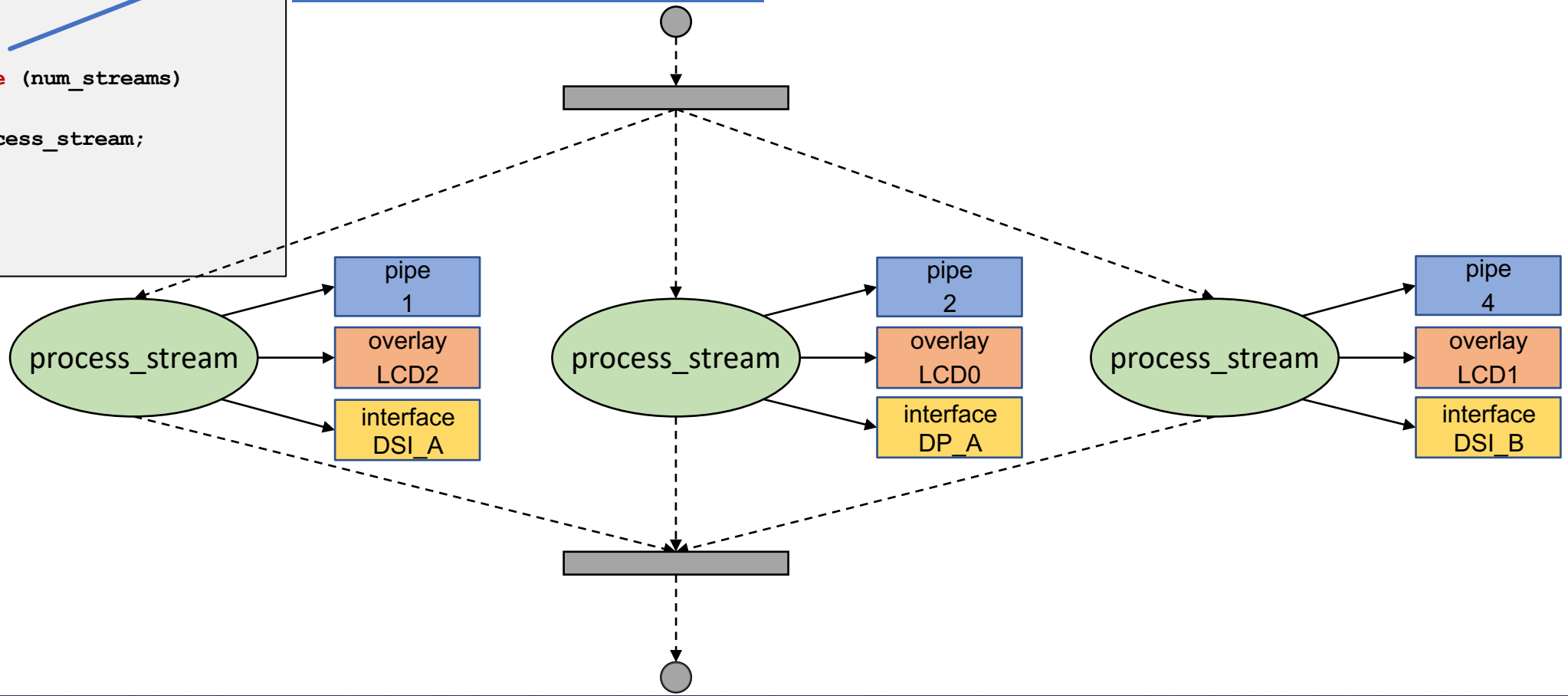
```
class process_stream extends uvm_object;
  . . .
  constraint pipe2overlay_c {
    pipe.kind == GFX -> overlay.kind != LCD0;
    pipe.kind == VID -> overlay.kind != LCD1;
  }
  constraint overlay2interface_c {
    overlay_kind == LCD0 -> interface_kind == DP_A;
    overlay_kind == LCD1 -> interface_kind inside {DSI_A, DSI_B};
    overlay_kind == LCD2 -> interface_kind inside {DSI_A, DSI_B, DP_B};
    overlay_kind == TV   -> interface_kind inside {DP_A, HDMI};
  }
  ...
endclass
```

データパス指定におけるPSSの制約指定は
SystemVerilogの制約とそれほど変わらない

並列にストリームをドライブ

```
action process_multi_stream {
  rand int in [1..4]
  num_streams;

  activity {
    parallel {
      replicate (num_streams)
  {
        do process_stream;
      }
    }
  }
}
```

ロックされたリソースは相互排他的にアサインされ並列実行されることが保証される



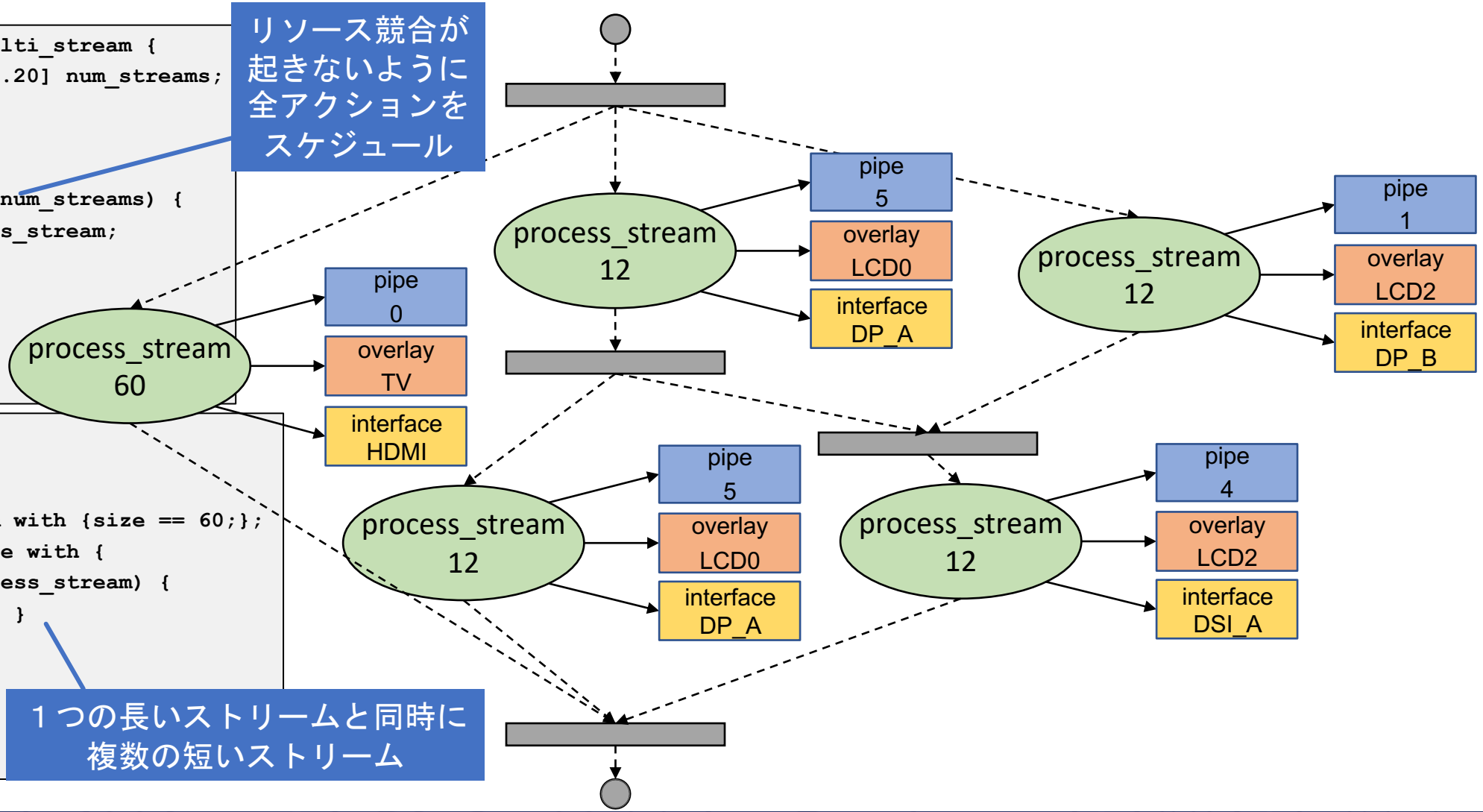
任意のスケジューリングを生成する

```
action process_multi_stream {
  rand int in [2..20] num_streams;

  activity {
    parallel {
      replicate (num_streams) {
        do process_stream;
      }
    }
  }
}
```

リソース競合が起きないように全アクションをスケジュール

```
action my_scenario_26 {
  activity {
    parallel {
      do process_stream with {size == 60;};
      do random_schedule with {
        forall (p: process_stream) {
          p.size == 12; }
      };
    }
  }
}
```

1つの長いストリームと同時に複数の短いストリーム



このセクションのまとめ

- PSSはリソースプールやクレームなどハイレベルなモデリングのコンストラクトをサポート
- コンストラクトでは設計／テストの動作間の依存関係を自然に表現
- 単純ではないフローやスケジューリングを容易に記述したりランダム化することが可能
- SystemVerilogを始めとする検証言語ではデータの制約付きランダムをサポートし、単純ではないフローやスケジューリングは難しい

アジェンダ

- PSSの概要と現在の状況
- ディスプレイコントローラの適用例
- メモリ＆キャッシュの適用例
- SoCレベルの適用例
- まとめ

このセクションの流れ

- Problem #1: DDR メモリコントローラ・ページマネジメント
- Problem #2: キャッシュ・コヒーレンシの状態遷移の探索
- Problem #3: Problem #1とProblem #2の同一シナリオでの組合せ
- SystemVerilogによるシナリオ・モデリングの課題
- Executor、メモリのモデリングと基本的な read/write テスト
- Solution #1: DDRのページマネジメントをPSSでモデル化
- Solution #2: キャッシュ・コヒーレンシの状態探索をPSSでモデル化
- Solution #3: DDRページマネジメントとキャッシュ・コヒーレンシを組合せたシナリオ

問題を解決する多くのソリューション

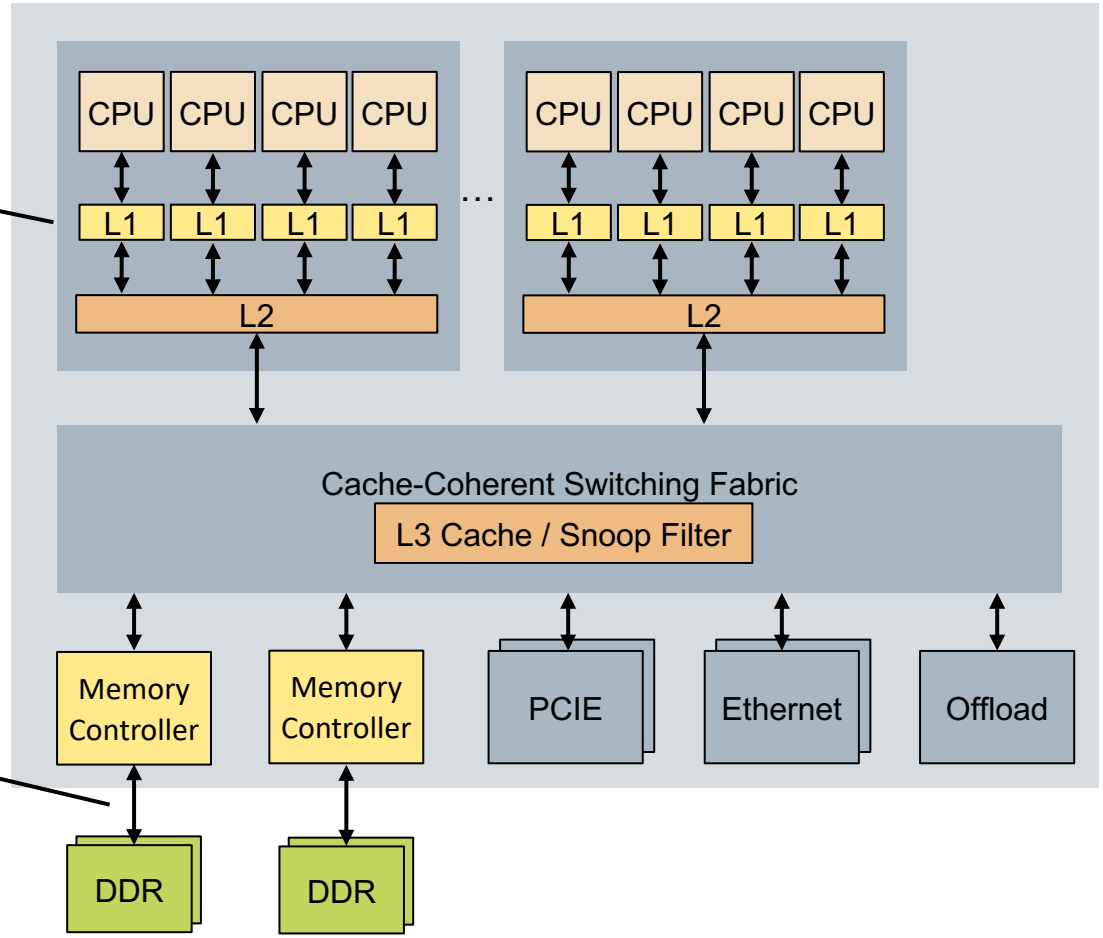
- Problem #1とProblem #2はそれぞれ独立していれば、より単純に解決可能

- このセクションではオペレーションのシーケンスを作成する2つの異なるアプローチと、それをどのように組合せてクロスカバレッジを取るかについて解説

解決すべき問題

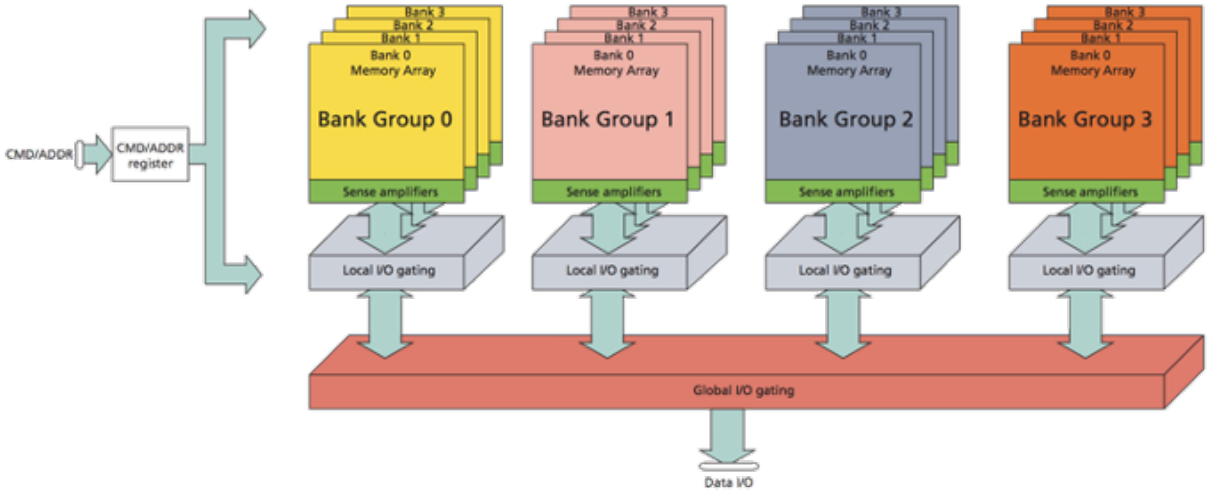
Problem #2:
キャッシュ・コヒーレンシ
状態探索

Problem #1:
DDRメモリコントローラ
ページマネジメント



Problem #1: DDRページマネジメント

- DDRはページで構成される
  - group、bank、row、column で特定
- 同一ページ内は高速アクセス
  - メモリコントローラはリフレッシュのインタリーブが必要
- 異なるページへのアクセスは遅い



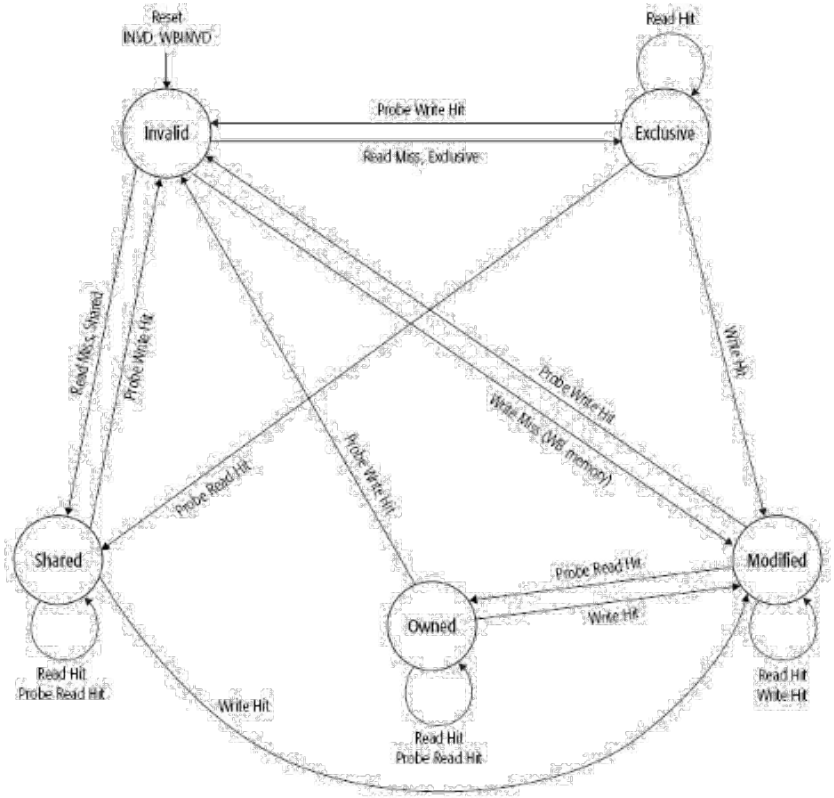
- 目的：メモリコントローラのページマネジメントをストレス検証するための read/write アドレスのシーケンスを生成する

4 Gb Addressing Table

Configuration		1 Gb x4
Bank Address	# of Bank Groups	4
	BG Address	BG0~BG1
	Bank Address in a BG	BA0~BA1
Row Address		A0~A15
Column Address		A0~A9
Page size		512B

Problem #2: キャッシュ・コヒーレンシ状態探索

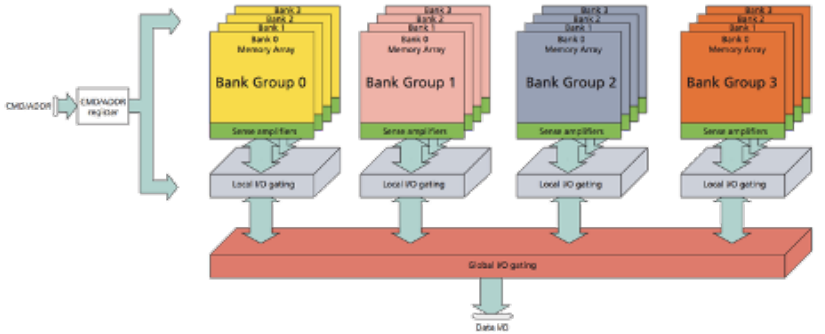
- SoCには複数段のキャッシュが存在
- キャッシュレベルごとに異なるコヒーレンシ・プロトコルが存在し得る
- 目的：適切なコア上でスケジューリングされるすべてのコヒーレンシの遷移を探索するオペレーションのシーケンスを生成する

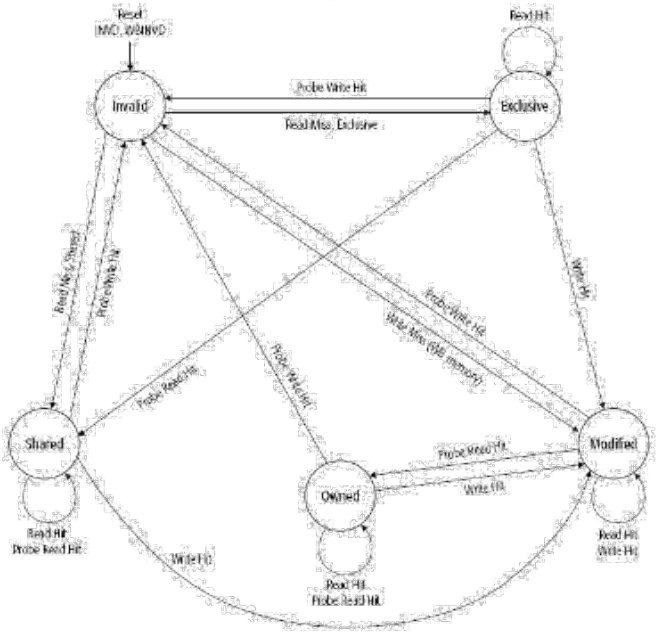


Problem #3: Problem #1 と #2の組合せ

- キャッシュの状態探索においてさまざまなタイミングでDDRメモリにアクセスしなくてはならない
- DDRのread/writeオペレーションはアドレスのページマネジメントのパターンに基づいて異なるタイミングになる

- 目的：DDRページマネジメントにストレスを与えながら、すべてのコヒーレンシの遷移を探索するように、適切なコア上にスケジュールされたオペレーションシのーケンスを生成する





SystemVerilogによるシナリオモデリングの課題

- Problem #1（DDRページマネジメント）もProblem #2（キャッシュ・コヒーレンシの状態遷移）も、異なるCPUコアを超えてスケジュールされる関連オペレーションのシーケンスが必要

- UVM/SystemVerilogでネイティブにモデリングすることが困難

- このような問題に対してPSSは簡潔なソリューションを提供する

基本となる Read/Write テスト

大切なポイント : ランダムな read/write テストが簡単に作れること

- システム内にExecutorのコアがいくつ存在するかを定義
- 使用可能なシステムメモリを定義
- 基本的な read/write オペレーションを定義
- ランダムなアドレスジェネレータを定義
- これらすべてを集約する

Executorとシステムメモリのコンフィギュレーション

システムでいくつのcoreが使えるかを定義

各executorにはそのidとともに”trait”がある

“trait” を持つcoreの配列

各core_idを初期化

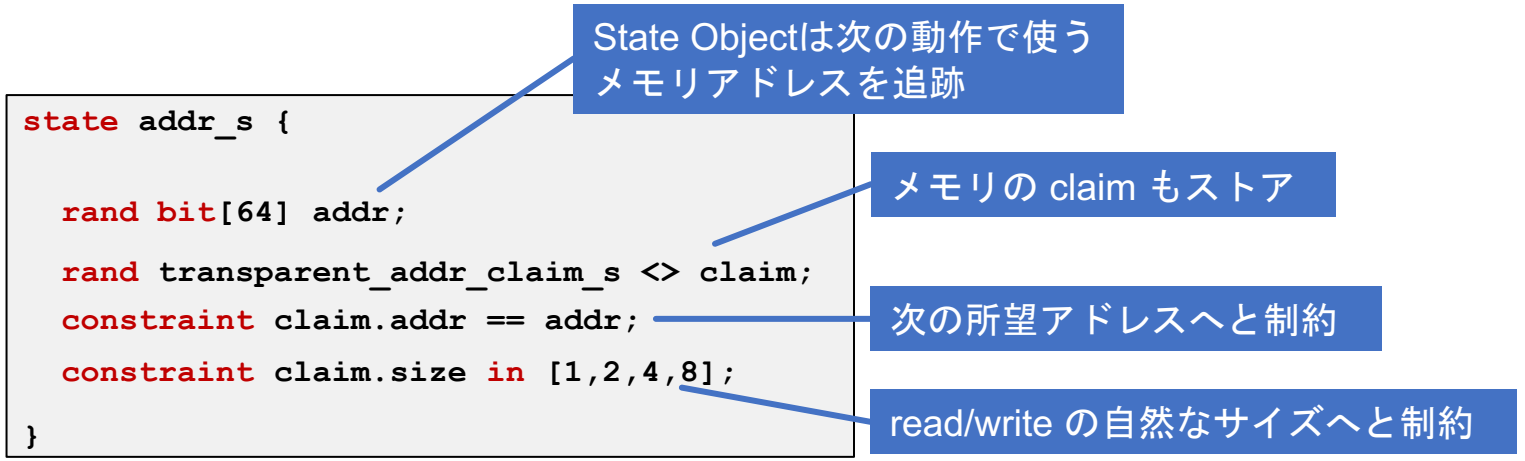
```
package config_pkg {
  const int NUM_CORES = 4;
}

struct core_traits_s : executor_trait_s {
  rand int in [0 .. config_pkg::NUM_CORES -
  1] core_id;
}
component cores_c : executor_group_c<core_traits_s> {
  executor_c<core_traits_s> cores
  [config_pkg::NUM_CORES];

  exec init_down {
    foreach (c : cores[i]) {
      c.trait.core_id = i;
      add_executor(c);
    }
  }
}
```

使用可能なシステムメモリの宣言

```
component pss_top_rand_addrs_c {

  transparent_addr_space_c<> sys_mem;

  exec init_up {
    transparent_addr_region_s<> region;
    region.size = ( 1 << 40 );
    region.addr = 0x80000000;
    (void)sys_mem.add_region(region);
  }
```

※ “trait” は特質という意味、PSS仕様書では Executor などのコアライブラリに定義されているアトリビュートのため英語とした

基本的な read/write 動作：State Object




```
state addr_s {

  rand bit[64] addr;

  rand transparent_addr_claim_s <> claim;
  constraint claim.addr == addr;
  constraint claim.size in [1,2,4,8];

}
```

基本的な read/write 動作

State Objectの入力

ランダムなCPU coreの claim

claimからハンドルをゲット

writeの実行

```
component mem_ops_c {

  action write_a {
    input addr_s inp;

    rand executor_claim_s<core_traits_s> core;
    rand bit [64] data;
    exec body {
      addr_handle_t h = make_handle_from_claim(inp.claim);
      match (imp.claim.size) {
        [1]:  write8(h,  data[7:0]);
        [2]: write16(h, data[15:0]);
        [4]: write32(h, data[31:0]);
        [8]: write64(h, data[63:0]);
      }
    }
  }
```

```
  action write_a {
    input addr_s inp;

    rand executor_claim_s<core_traits_s> core;
    bit [64] data;
    exec body {
      addr_handle_t h = make_handle_from_claim(inp.claim);

      match (imp.claim.size) {
        [1]:  data[ 7:0] =  read8(h);
        [2]:  data[15:0] = read16(h);
        [4]:  data[31:0] = read32(h);
        [8]:  data[63:0] = read64(h);
      }
    }
  }
} // mem_ops_c
```

ランダムアドレス生成とインスタンス化

```
component rand_addrs_c {

  action rand_addrs_a {
    output addr_s out;
  }

}
```

制約を受けない
ランダムなアドレスを出力

```
component rand_addr_test_c {
  rand_addr_c rand_addrs;
  mem_ops_c mem_ops;
  pool addr_s addr_p;
  bind addr_p *;
  action rand_addr_test_a {
    activity {
      do rand_addrs_c::rand_addrs_a;

      select {
        do mem_ops_c::write_a;
        do mem_ops_c::read_a;
    } }
  }
}
```

インスタンス化して bind

次のアドレスを生成

write か read を実行

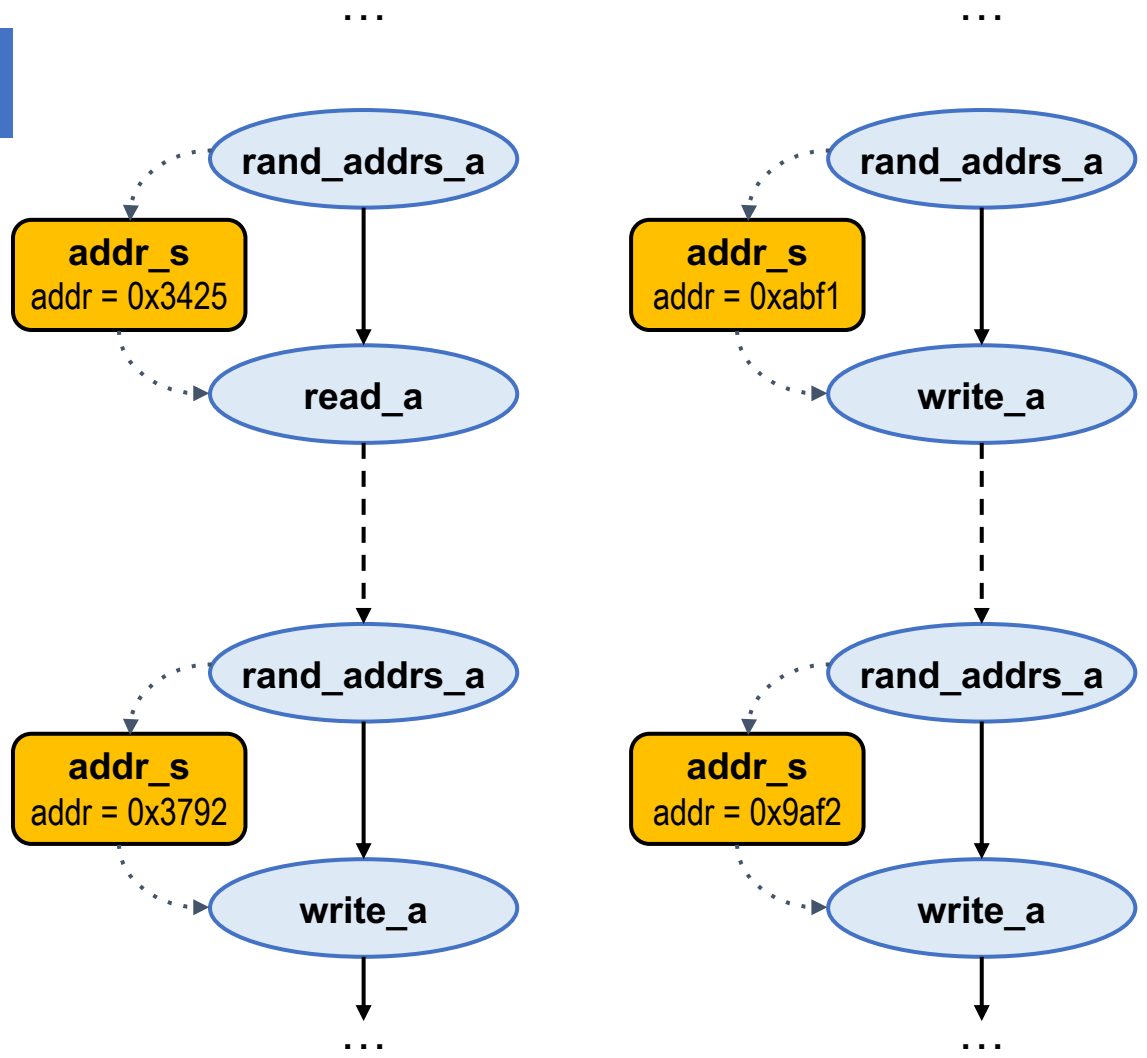
基本的なメモリテスト

```
component pss_top_rand_addrs_c {

  transparent_addr_space_c<> sys_mem;
  exec init_up {...}
  cores_c cores;

  rand_addr_test_c rand_addr_test[10];

  action entry_a {
    activity {
      schedule {
        replicate (100) {
          do rand_addr_test_c::rand_addr_test_a;
        }
      }
    }
  }
}
```

システムで使用可能なメモリとコアを宣言

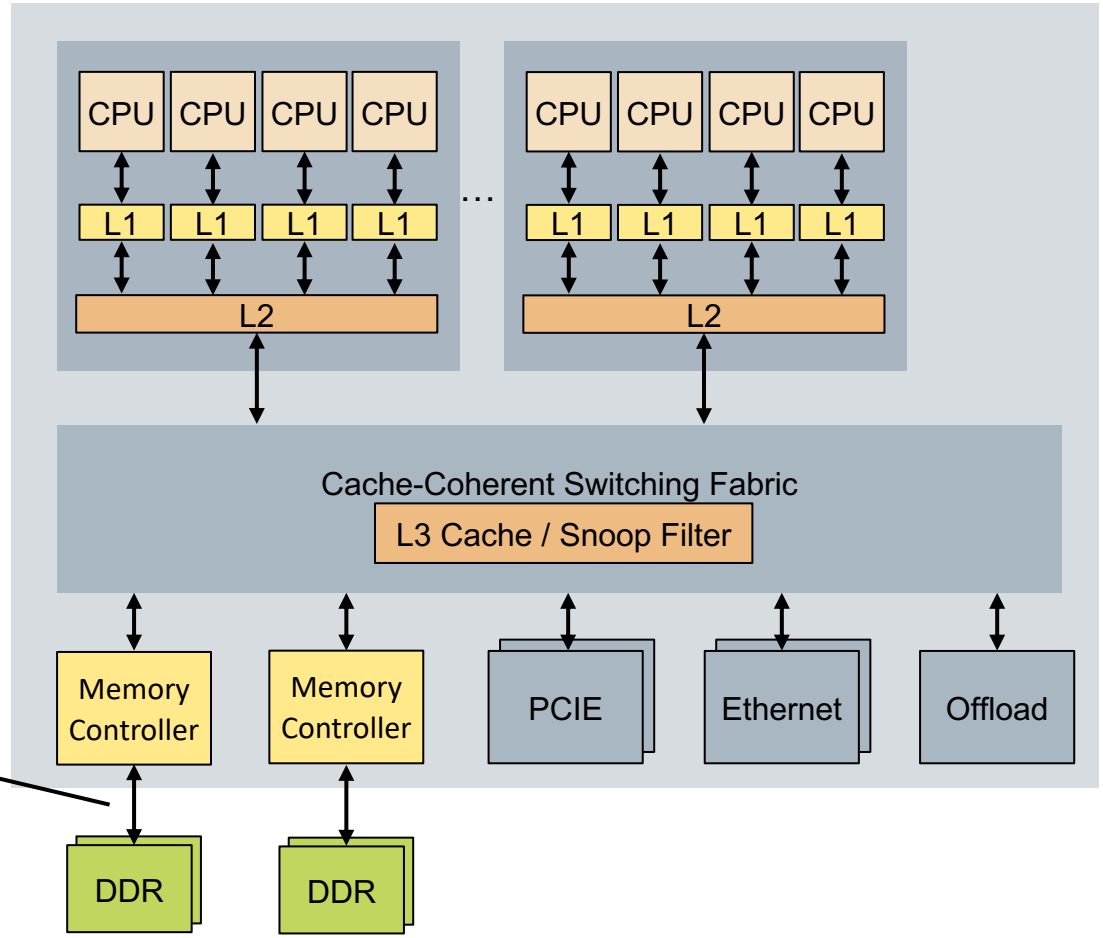
インスタンス化

アクションをスケジュール



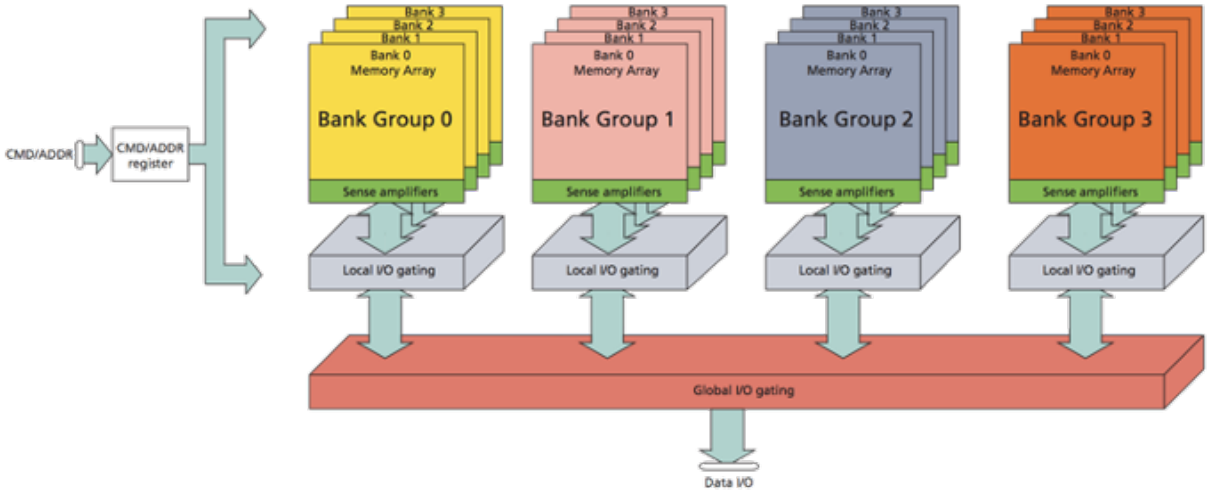
Solution #1: DDRページマネジメント

Problem #1:
DDRメモリコントローラ
ページマネジメント



Solution #1: DDRページマネジメント

- DDRはページで構成される
  - group、bank、row、column で特定
- 同一ページ内は高速アクセス
  - メモリコントローラはリフレッシュのインタリーブが必要
- 異なるページへのアクセスは遅い



- 目的：メモリコントローラのページマネジメントをストレス検証するための read/write アドレスのシーケンスを生成する

4 Gb Addressing Table

Configuration		1 Gb x4
Bank Address	# of Bank Groups	4
	BG Address	BG0~BG1
	Bank Address in a BG	BA0~BA1
Row Address		A0~A15
Column Address		A0~A9
Page size		512B

DDRメモリバンク・アドレスシーケンスのモデリング

大切なポイント：アドレスのシーケンスを State Object を用いてモデリングすること

```
component ddr_page_addrs_c {
  state ddr_page_s {
    rand bit [ 2] group;
    rand bit [ 2] bank;
    rand bit [16] row;
    rand bit [10] column;
    rand bit [64] addr;

    constraint addr[18: 9] == column;
    constraint addr[34:19] == row;
    constraint addr[36:35] == bank;
    constraint addr[38:37] == group;
  }

  pool ddr_page_s ddr_page_p;
  bind ddr_page_p *;
```

State Object はビルトイン変数 prev によりシーケンス生成を制約できる

DDRのページ識別

解決されたアドレス

メモリタイプ固有の制約

pool を作成して bind する

DDRメモリバンク・アドレスシーケンスのモデリング

```
action ddr_same_page_a {
  output ddr_page_s ddr_page;

  constraint c {
    ddr_page.group  == ddr_page.prev.group  ;
    ddr_page.bank   == ddr_page.prev.bank   ;
    ddr_page.row    == ddr_page.prev.row    ;
    ddr_page.column == ddr_page.prev.column ;
   };
}

action ddr_same_bank_a {
  output ddr_page_s ddr_page;

  constraint c {
    ddr_page.group    == ddr_page.prev.group   ;
    ddr_page.bank     == ddr_page.prev.bank    ;
    ( ddr_page.row    != ddr_page.prev.row  ||
      ddr_page.column != ddr_page.prev.column )  ;
   };
}
```

同じページのアクセスヒット
高速だがリフレッシュが必要

同一バンクの異なるページ
へのアクセスヒット
ページスイッチングが遅い

他のアドレス指定方法の
戦略を複数準備しておく

```
action ddr_next_col_a {
  output ddr_page_s ddr_page;

  constraint c {
    ddr_page.group   ==   ddr_page.prev.group  ;
    ddr_page.bani    ==   ddr_page.prev.bank   ;
    ddr_page.row     ==   ddr_page.prev.row    ;
    ddr_page.column  ==
           (ddr_page.prev.column + 1) & 0x3ff  ;
  };
}
```

DDRメモリバンク・アドレスシーケンスのモデリング

```
  action select_strategy_a {
    activity {
      select {
        [8]: do ddr_same_page_a;
        [1]: do ddr_next_col_a;
        [1]: do ddr_same_bank_a;
          ...
      }
    }
  }
  action constrain_addr_a {
    input ddr_page_s ddr_page;
     output addr_s out;

  constraint out.addr == ddr_page.addr;
  }
} // component ddr_page_addrs_c
```

次のアドレスを選択 –
80%の確率で同一ページ

状態からaddrを出力

インスタンス化と bind

アドレスを選択して
read か write を実行

```
component ddr_test_c {
  ddr_page_addrs_c ddr_page_addrs;
   mem_ops_c mem_ops;
  pool addr_s addr_p;
  bind addr_p *;
  action ddr_test_a {
    activity {
      do
  ddr_page_addrs_c::select_strategy_a;
      do ddr_page_addrs_c::constrain_addr_a;
      select {
        do mem_ops_c::write_a;
        do mem_ops_c::read_a;
      }
    }
} }
```

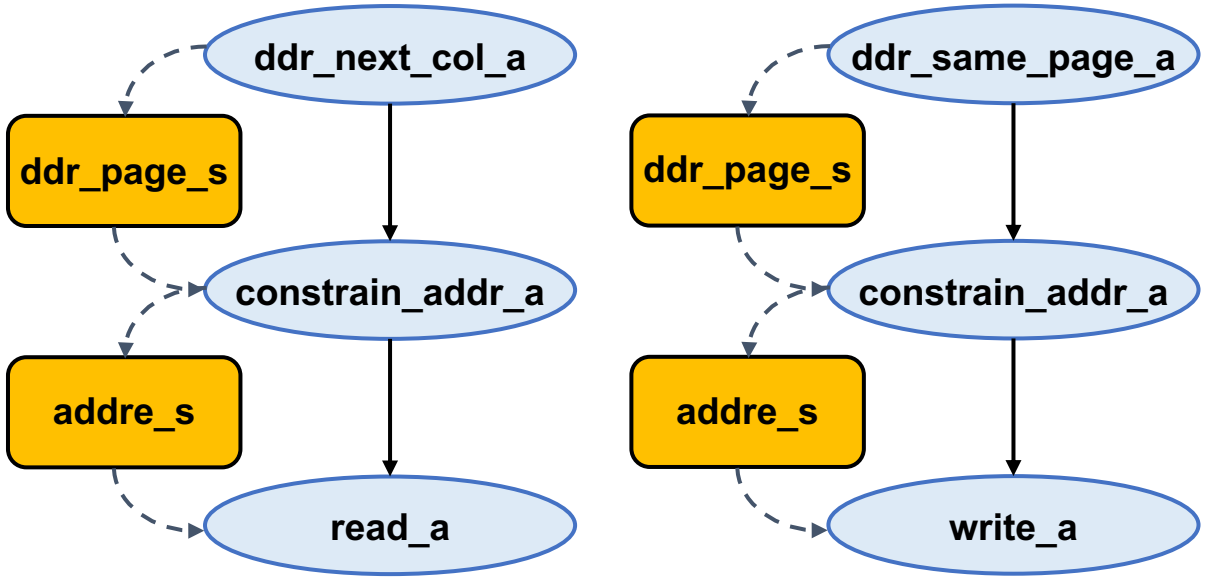
DDRメモリバンク・アドレスシーケンスのモデリング

```
component pss_top_ddr_addrs_c
  {
  cores_c cores;
  ddr_test_c ddr_test[10];
  transparent_addr_space_c<> sys_mem;
  exec init_up {...}
  action entry_a {
    activity {
      schedule {
        replicate (100){
          do ddr_test_c::ddr_test_a;
        }
      }
    }
  }
}
```

インスタンス化

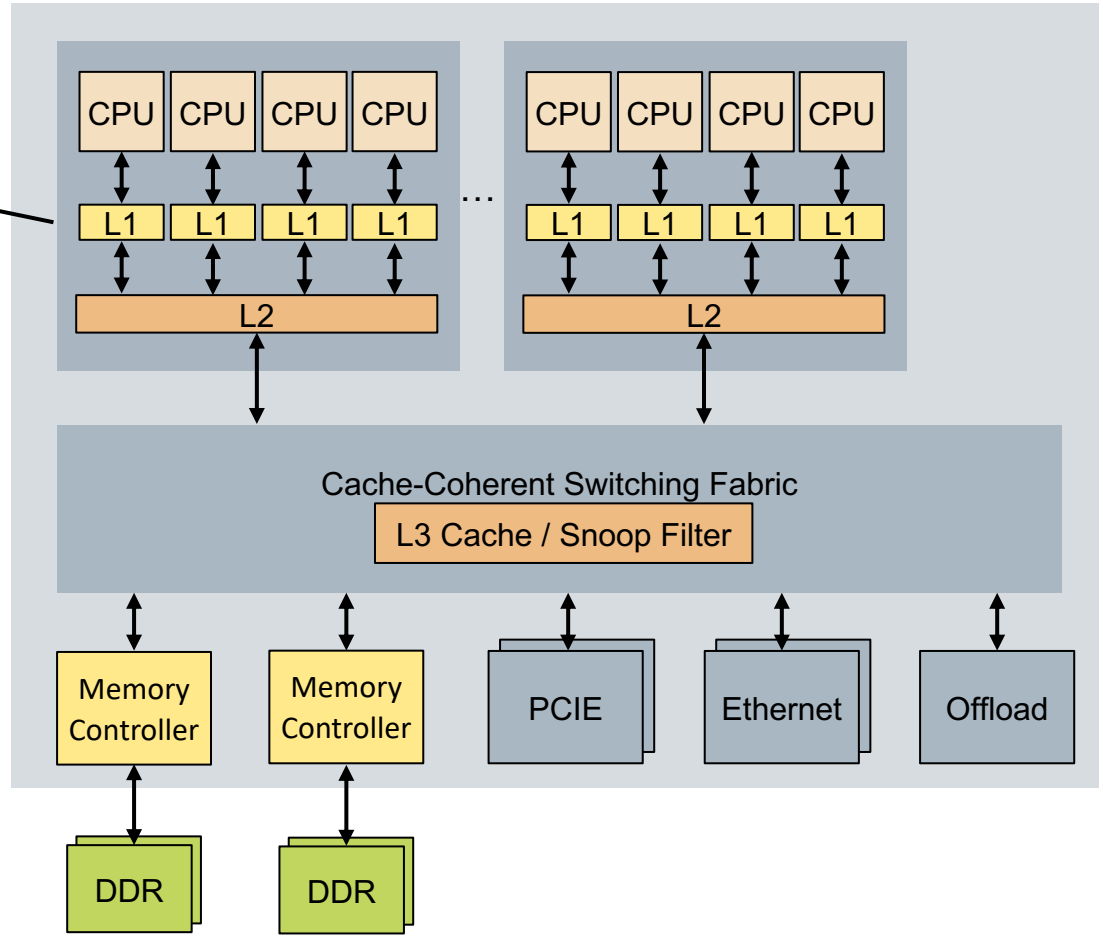
使用可能な
システムメモリの宣言

アクションの
スケジュール



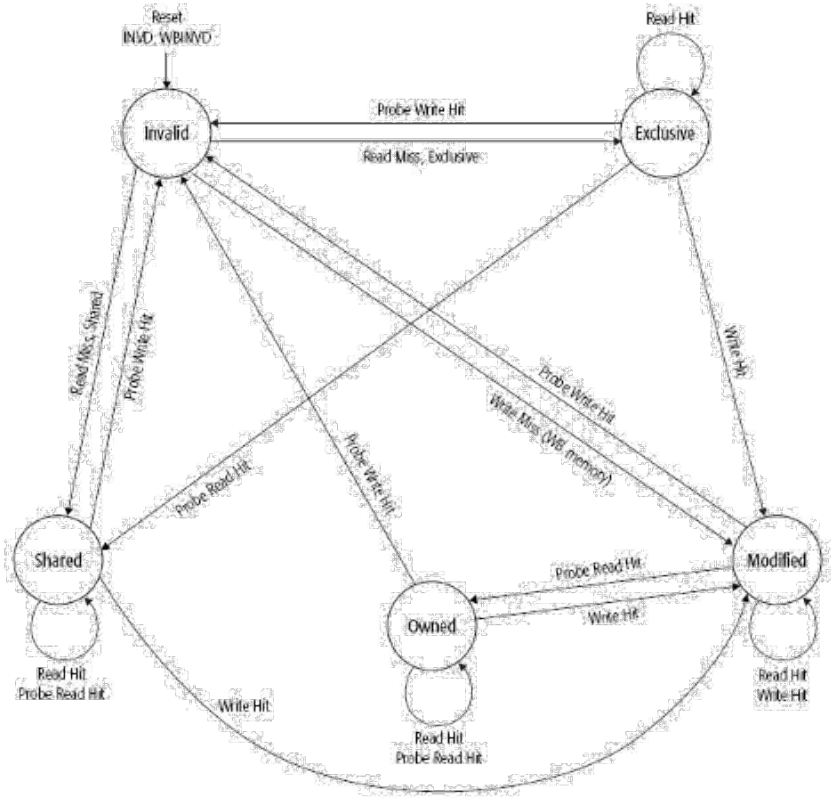
Solution #2: キャッシュ・コヒーレンシ状態探索

Problem #2:
キャッシュ・コヒーレンシ
状態探索



キャッシュ・コヒーレンシ状態遷移

- SoCには複数段のキャッシュが存在
- キャッシュレベルごとに異なる
  コヒーレンシ・プロトコルが
  存在し得る
- 目的：適切なコア上でスケジューリング
  されるすべてのコヒーレンシの遷移を
  探索するオペレーションのシーケンスを
  生成する



キャッシュ・コヒーレンシ状態遷移

大切なポイント：コヒーレンシのシーケンスを action 推定を用いてモデリングすること

```
component coherency_c {

  enum cl_state_e {INVALID, EXCLUSIVE, MODIFIED, OWNED, SHARED};
  state cl_state_s {
    rand cl_state_e cl_state;
    constraint initial -> cl_state == INVALID;

    rand int home_core_id;

    rand int count;
    constraint initial -> count == 0;
  }

 pool cl_state_s cl_state_p;
 bind cl_state_p *;
```

ターゲットのキャッシュステートを追跡

シナリオの ”Home” となる core_id

テスト長のカウンタ

pool を作成して bind する

```
action reset_counter_a {
  output cl_state_s out;
  constraint out.count == 0;
}
```

カウントをリセットする action

キャッシュ・コヒーレンシ状態遷移のモデリング

```
abstract action transition_base_a {
  input cl_state_s  inp;
  output cl_state_s out;

  constraint out.cout == inp.count +1;

  constraint out.home_core_id == inp.home_core_id;

}
```

ステートの入力＆出力

シナリオ長を管理

“Home” core id　を管理

```
action invalid_to_exclusive_a : transition_base_a {
  constraint transition {
    inplcl_state == INVALID;
    out.cl_state == EXCLUSIVE;
  }

  activity {
    do mem_ops_c::read_a_with {
      core.trait.core_id == this.inp.home_core_id;
};
} }
```

Prev/Next ステート

“Home” core から読出す

```
action exclusive_to_invalid_a : transition_base_a {
  constraint transition {
    inp.cl_state == EXCLUSIVE;
    out.cl_state == INVALID;
  }

  activity {
    do mem_ops_c::write_a with {
      core.trait.core_id != this.inp.home_core_id; };
    }
  }
} // coherency_c
```

他の core から書出す

キャッシュ・コヒーレンシ状態遷移のモデリング

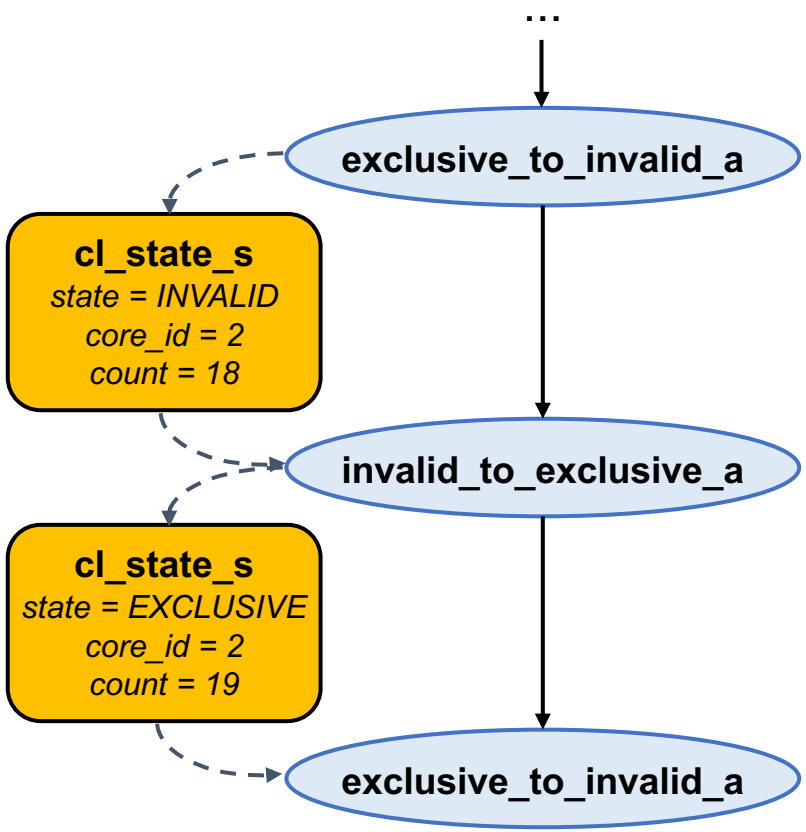
```
component ipss_top_coherency_simple_c {
  cores_c cores;
  rand_addrs_c rand_addrs;
  mem_ops_c mem_ops;
  coherency_c coherency;
  pool addr_s addr_p;
  bind addr_p *;

  transparent_addr_space_c<> sys_mem;
  exec init_up {...}


  action entry_a {
    activity {
      do rand_addrs_c::rand_addrs_a;

      do coherency_c::exclusive_to_invalid_a
           with { out.count == 20; };
    }
  }
}
```

インスタンス化と bind

使用可能な
システムメモリを宣言

次のランダムな
アドレスを生成

20の状態遷移の
シーケンスを推定



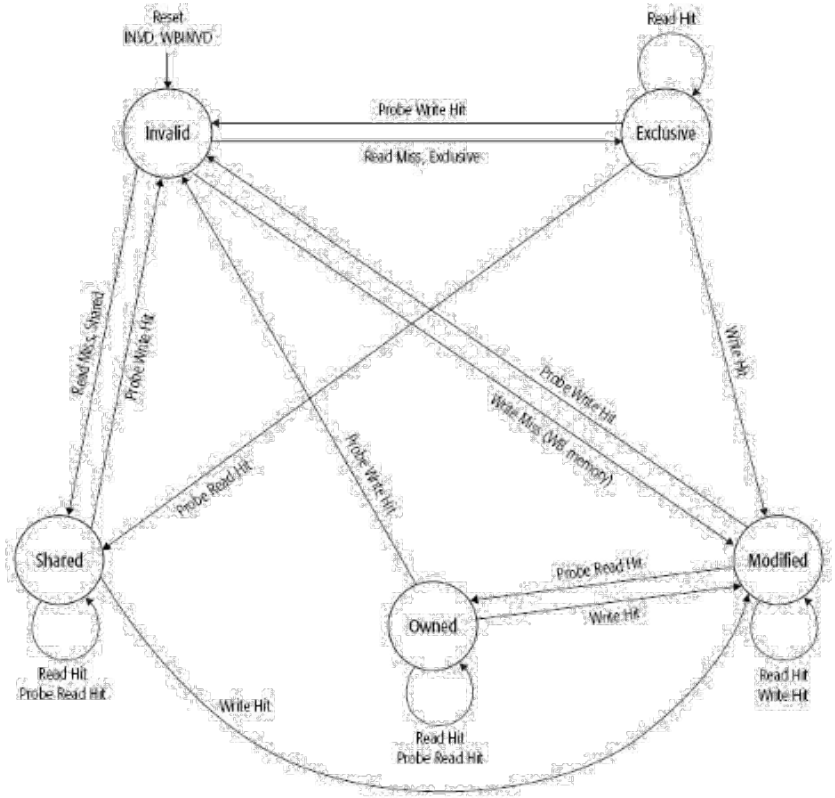
キャッシュ・コヒーレンシ状態遷移のモデリング

```
component coherency_c {
  ...
  action invalid_to_modified_a    : transition_basd_a { ...}
  action invalid_to_owned_a       : transition_basd_a { ...}
  action invalid_to_shared_a      : transition_basd_a { ...}
  action exclusive_to_modified_a  : transition_basd_a { ...}
  action exclusive_to_shared_a    : transition_basd_a { ...}
  ...

  action state_selector_a {
    rand int count;
    activity {
      select {
        do exclusive_to_invalid_a with {out.count == this.count;};
        do modified_to_invalid_a  with {out.count == this.count;};
        do owned_to_invalid_a     with {out.count == this.count;};
        do shared_to_invalid_a    with {out.count == this.count;};
      }
    }
  }
}
```

すべての遷移を定義

”count” 変数は上記から制約を受ける

常に invalid で帰結させることで 続くシナリオはアドレスを再利用できる



キャッシュ・コヒーレンシ状態遷移のモデリング

```
component coherency_test_c {
  rand_addrs_c rand_addrs;
  mem_ops_c mem_ops;
  coherency_c coherency;
  pool addr_s addr_p;
  bind addr_p *;    :

  action coherency_test_a {
    rand int count;

    activity {

      do rand_addrs_c::rand_addrs_a;
      do coherency_c::reset_counter_a;

      do coherency_c::state_selector_a
           with { count == this.count;};

    }
  }
}
```

インスタンス化と bind

count 変数は上記から制約を受ける

ランダムアドレスを選び
シナリオ長のカウンタをリセット

コヒーレンシ状態遷移の
ランダムシーケンスを推定

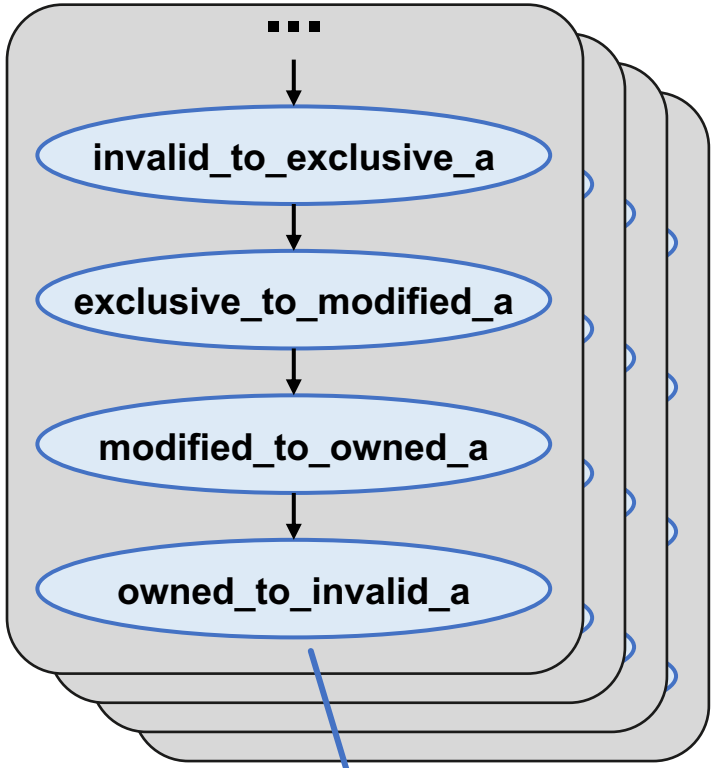
キャッシュ・コヒーレンシ状態遷移のモデリング

```
component pss_top_coherency_c {
  cores_c cores;
  coherency_test_c coherency_test[10];

  transparent_addr_space_c<> sys_mem;
  exec init_up { ... }

  action _entry_a {
    activity {
      schedule {
        replicate (100) {
          do coherency_test_c::coherency_test_a
            with {count == 20;};
        }
      }
    }
  }
}
```

インスタンス化

使用可能な
システムメモリの宣言

一連の20のコヒーレンシ
状態遷移を指定する

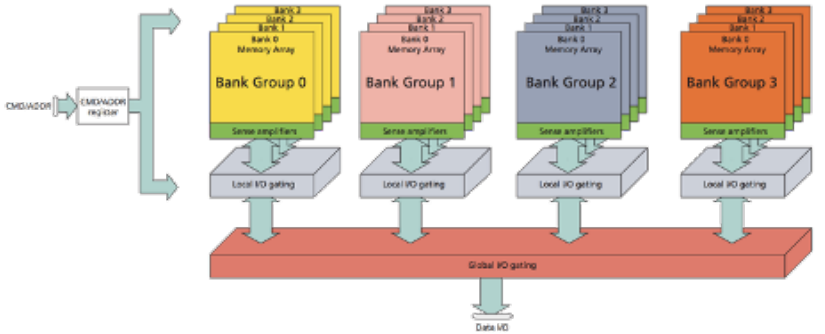


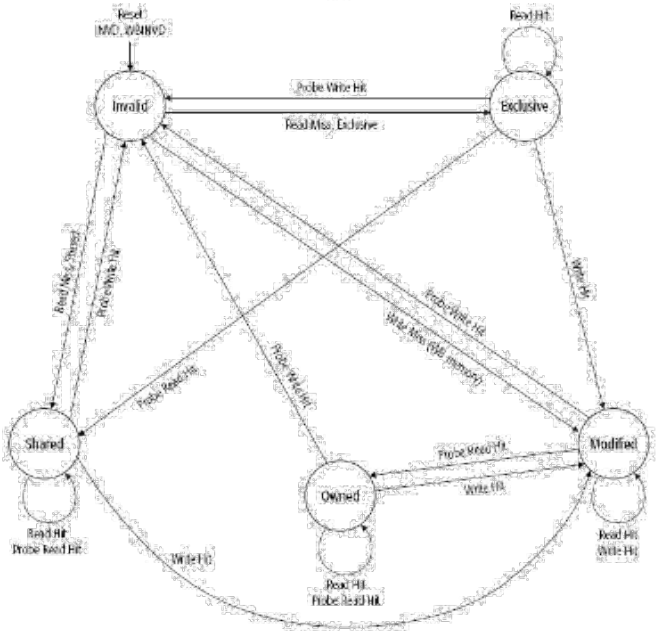
ランダム遷移のそれぞれの推定では
１つのランダムアドレスを再利用する

Solution #3: Solution #1 と #2の組合せ

- キャッシュの状態探索においてさまざまなタイミングでDDRメモリにアクセスしなくてはならない
- DDRのread/writeオペレーションはアドレスのページマネジメントのパターンに基づいて異なるタイミングになる

- 目的：DDRページマネジメントにストレスを与えながら、すべてのコヒーレンシの遷移を探索するように、適切なコア上にスケジュールされたオペレーションシのーケンスを生成する





Problem #1 と #2 の組合せ

大切なポイント：複数シナリオによる組立て

```
component coherency_ddr_test_c {
  ddr_page_addrs_c ddr_page_addrs;
  mem_ops_c mem_ops;
  coherency_c coherency;
  pool addr_s addr_p;
  bind addr_p *;    :

  action coherency_ddr_test_a {
    rand int count;

    activity {
      do ddr_page_addrs_c::select_stragety_a;
      do ddr_page_addrs_c::constraing_addr_a;

      do coherency_c::reset_counter_a;

      do coherency_c::state_selector_a
           with {count ==
this}count;};
  }
}
```

インスタンス化と bind

count 変数は上記から制約を受ける

DDRバンクアドレスを選択

シナリオ長カウンタをリセット

コヒーレンシ遷移のランダムシーケンスを推定

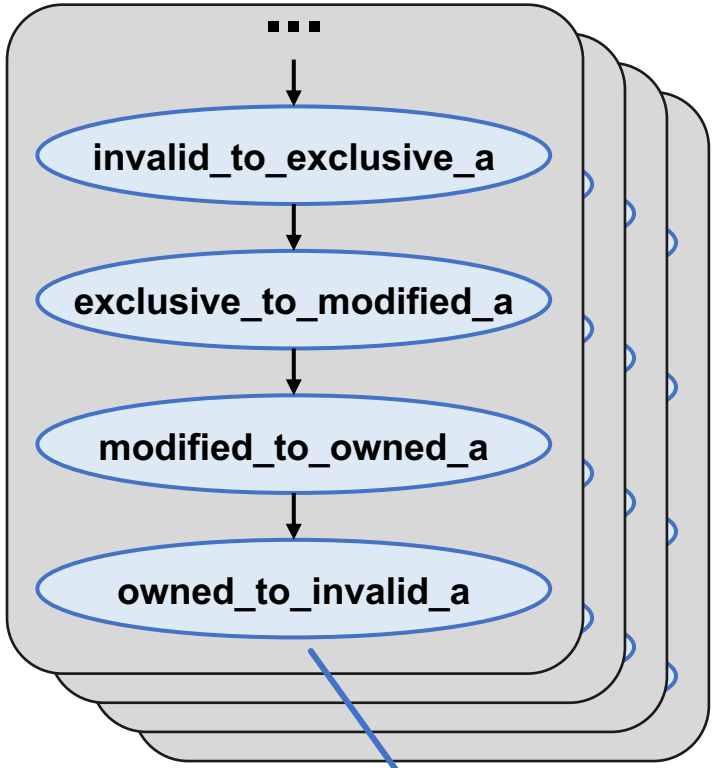
Problem #1 と #2 の組合せ

```
component pss_top_coherency_ddr_c {
  cores_c cores;
  coherency_ddr_test_c coherency_ddr_test[10];

  transparent_addr_space_c<> sys_mem;
  exec init_up { ... }

  action entry_a {
    activity {
      schedule {
        replicate(100) {
          do coherency_ddr_test_c::coherency_ddr_test_a
              with {count == this.count;};
        }
      }
    }
  }
}
```

インスタンス化

使用可能な
システムメモリの宣言

20の一連の
コヒーレンシ状態遷移を指定



ランダム遷移のそれぞれの推定では
DDRのアドレスシーケンスからの
アイテムを再利用する

まとめ

- Problem #1: ステート変数を用いてDDRのページアドレスシーケンスをモデル化した
- Problem #2: アクションの推定を用いてコヒーレンシ状態遷移をモデル化した
- Problem #3: 複数モデルを組立て Problem #1 と #2 に対応した
- SystemVerilogやC/C++ではこれらのモデル化は困難かつ膨大な時間を要するだろう
- PSSでは洗練されたソリューションになる

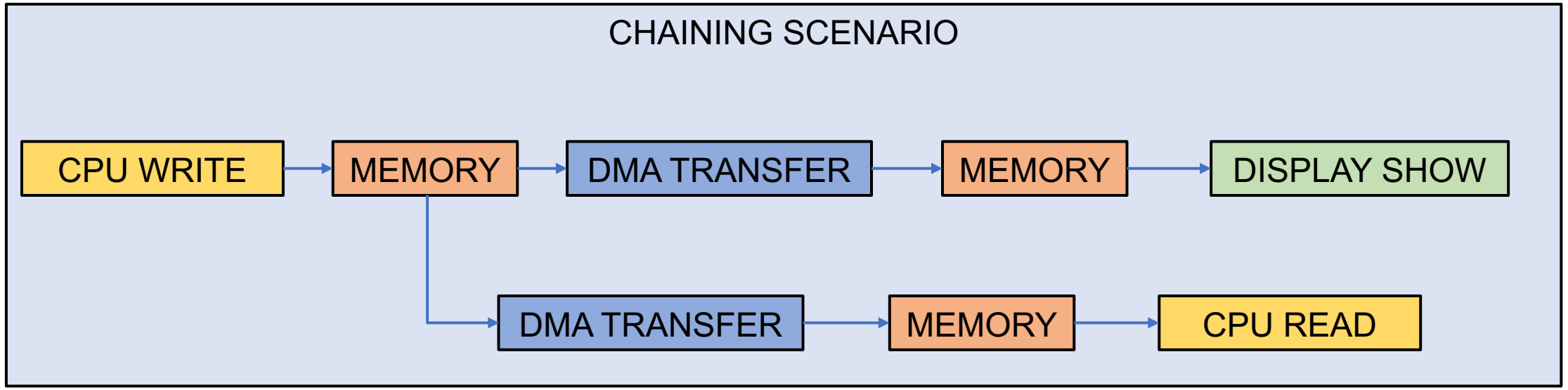
アジェンダ

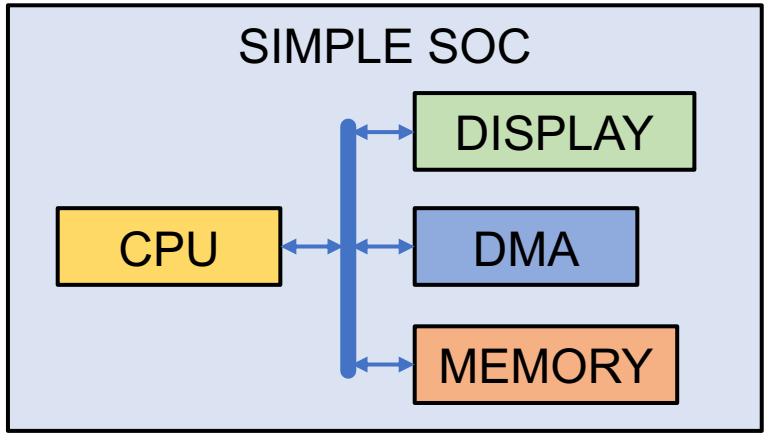
- PSSの概要と現在の状況
- ディスプレイコントローラの適用例
- メモリ＆キャッシュの適用例
- SoCレベルの適用例
- まとめ

このセクションの流れ

- SoCにおけるシナリオのチェイニング – DVCon US 2020から
- DMAパイプライニング・シナリオのモデリング
- 一般的なIPパイプライニング・シナリオのモデリング
- SoCシナリオのチェイニング／パイプライニングのモデリング

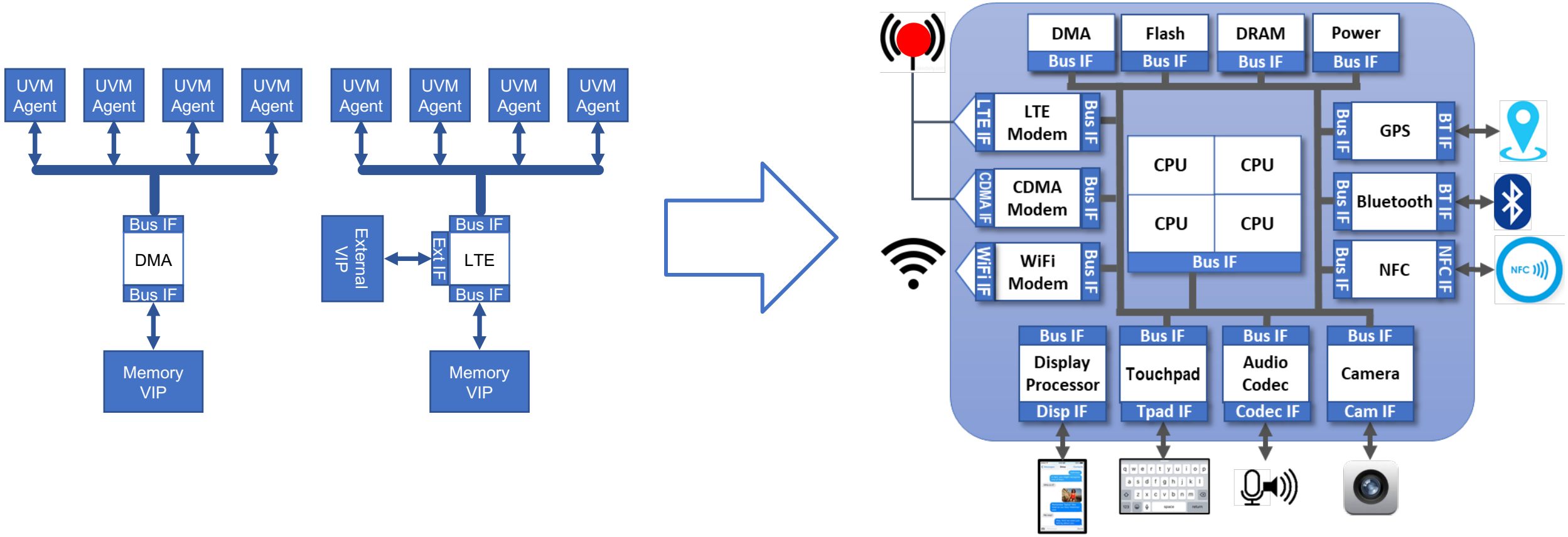
複数IPのシナリオインテント





訳者注釈：
Source = 供給源
Sink = 吸収源

ブロックからシステムへの可搬性と生産性 (DVCon US 2020より)



システムレベルの定義

```
extend component pss_top {
  // RTL Agents
  dma_c dma;
  lte_c lte;
  display_c display;
  ...
  // Execution agents
  pool [4] execution_agent_r cpu;
  pool [1] lte_cip_r lte_vip;
  ...
}
```

```
extend component pss_top {
  // Address Space
  contiguous_addr_space_c<mem_trait_s> mem_addr_space;
  addr_region_s<mem_trait_s> dram_region;
  addr_region_s<mem_trait_s> flash_region;
  exec int {
    dram_region.trait.kind = DRAM;
    mem_addr_space.add_region(dram_region);
    mem_addr_space.add_region(flash_region);
  }
}
```

メモリバッファを介した複数IPからのチェイニング・スティミュラス

Flow Object Typeを共通化して出力バッファ宣言

チェイニング構成;
シーケンシャル、パラレル、グラフ

データ保全のためのストレージアロケーション割当ての活用

チェイニングのカバレッジ

Flow Object Typeを共通化し出力バッファを宣言

```
package common_target {
  buffer data_buffer {
    rand addr_space_pkg::addr_claim_s<mem_trait_s> mem_seg;
  }

  abstract action mem_copy_a {
    input data_buffer buf_in;
    output data_buffer buf_out;
  }
}
```

IP オーナーのスティミュラス

LTE Modem

CDMA Modem

WiFi Modem

ソース：
外部 VIPから Memoryへの侵入
Memory2Memory：
(a) Memoryから外部VIPへの退出
(b) 外部VIPループバックからMemoryへの侵入
シンク：
Memoryから外部VIPへ

GPS

Bluetooth

NFC

シンク：
Memoryから外部VIPへ

Audio Codec

Display Processor

ソース：
外部VIPから Memoryへ

Touchpad

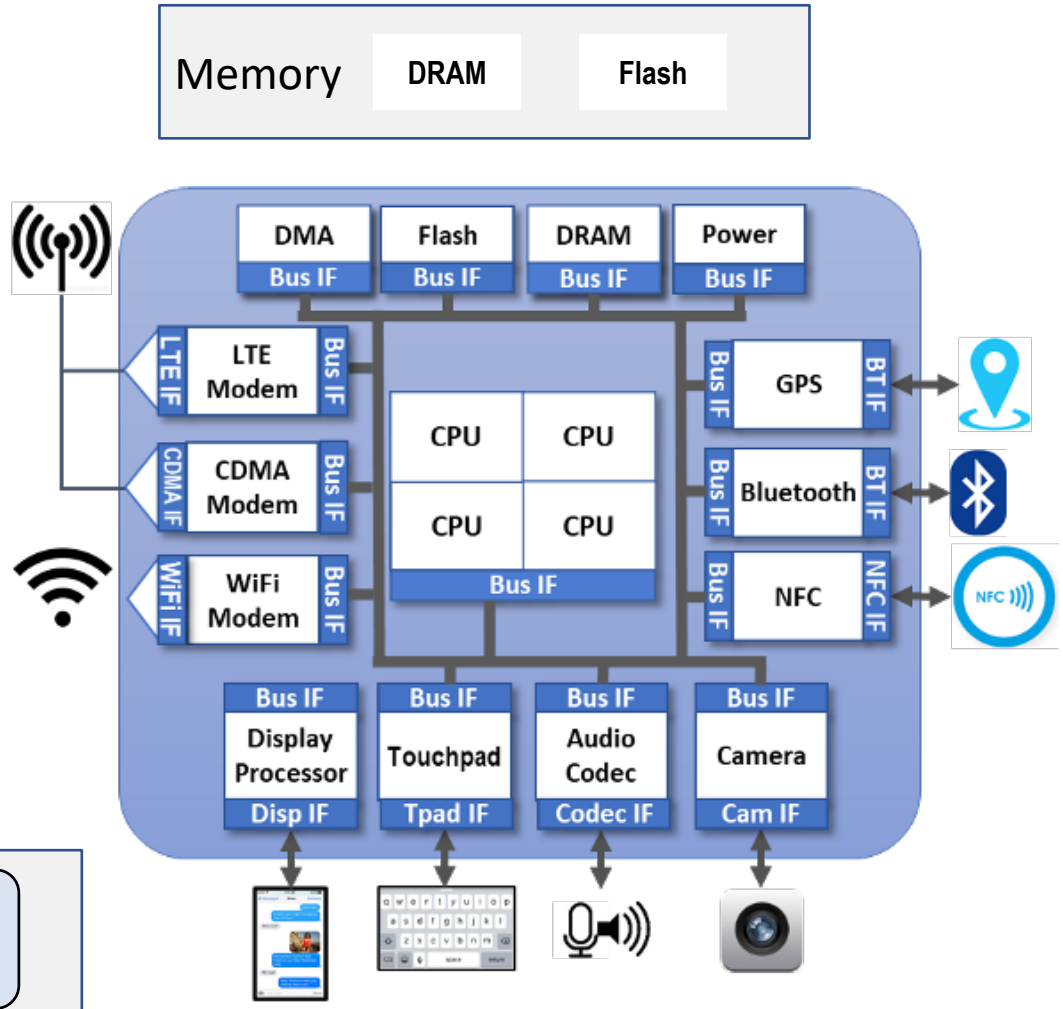
Camera

DMA

Memory2Memory：
DMA転送

CPU

Memory2Memory：
(a) Memory 読出し
(b) Memory 書込み



IP オーナーのアクション

LTE Modem

CDMA Modem

WiFi Modem

```
action <device-name>_source_a {
  output data_buffer ctb_out;
  …
}
action <device-name>_mem2mem_a :
  mem_copy_a {…}
```

GPS

Bluetooth

NFC

```
action <device-name>_sink_a
 { input data_buffer ctb_in;
   ...
}
```

Audio Codec

Display Processor

```
action <device-name>_source_a {
 { output data_buffer ctb_out;
   ...
}
```

Touchpad

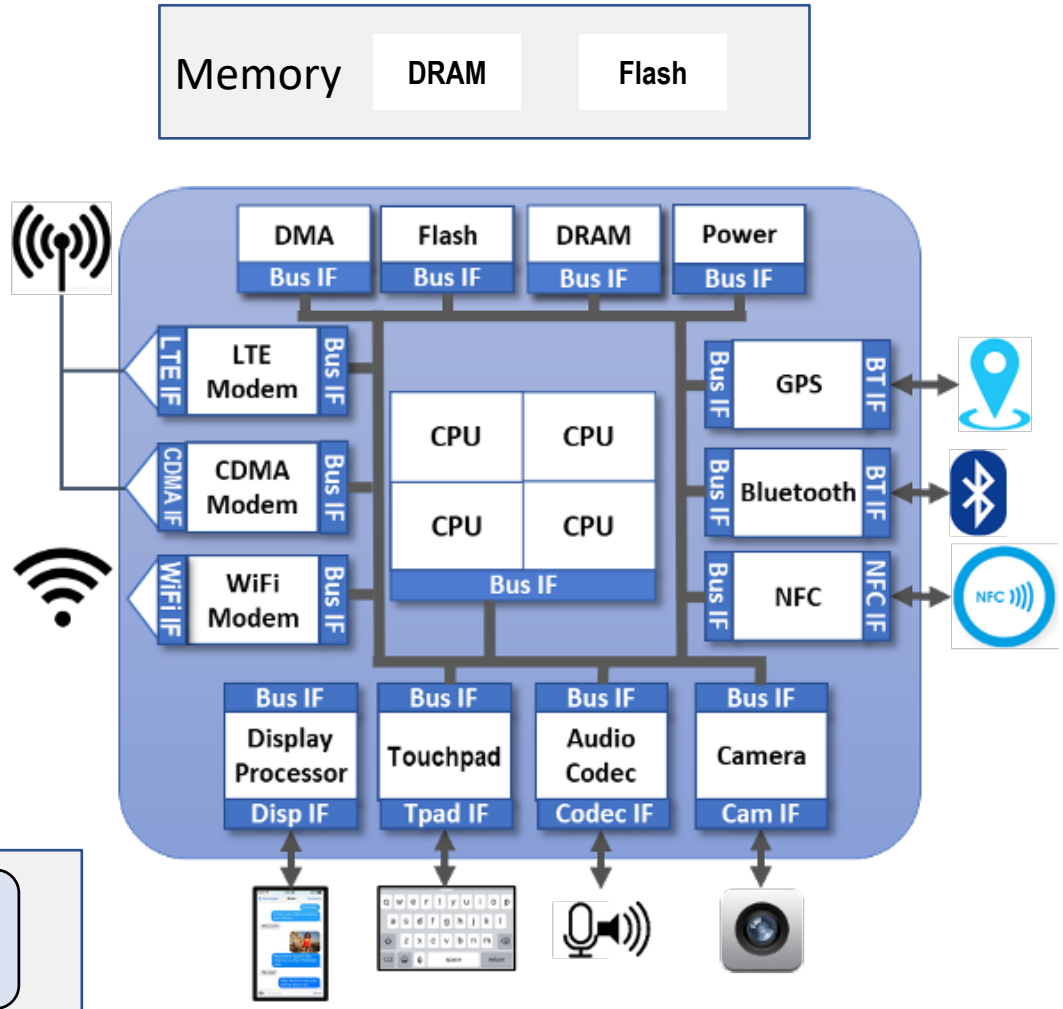
Camera

DMA

```
action core_mem2mem_a :
  mem_copy_a {...}
```

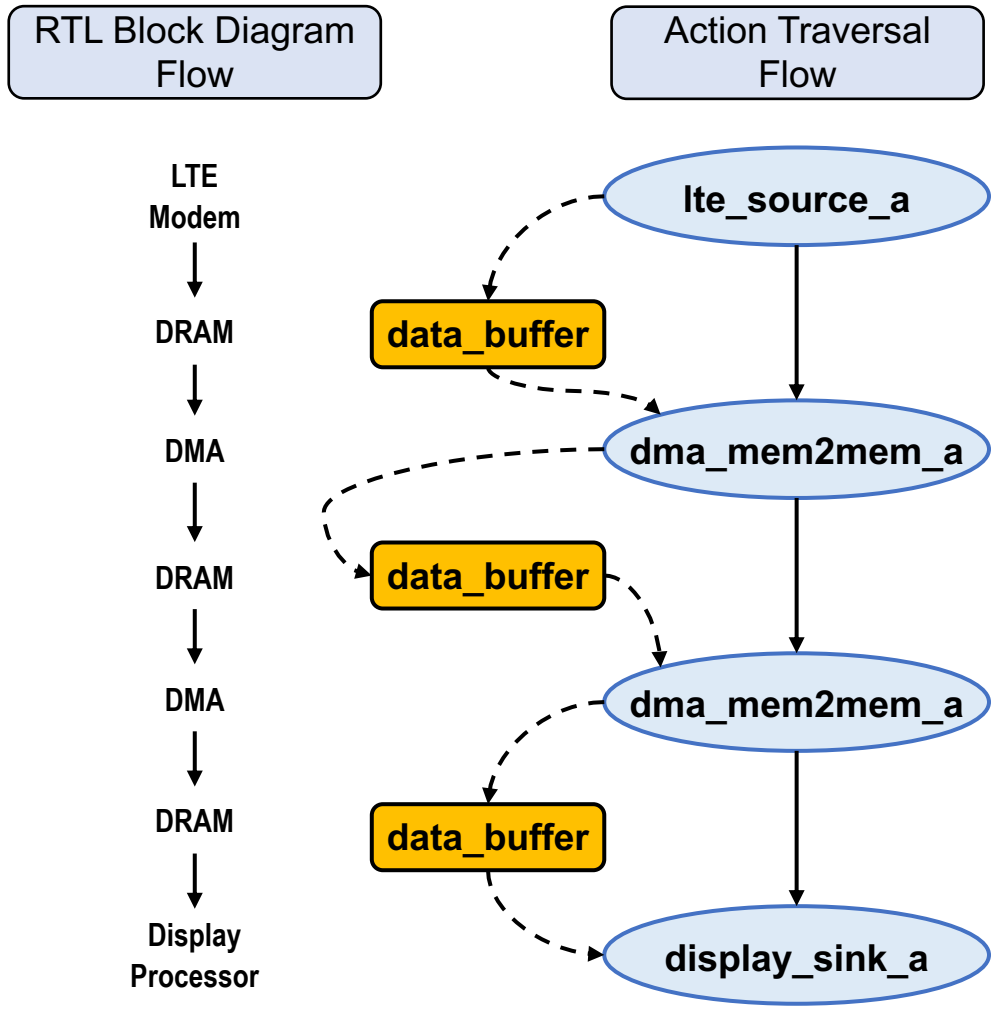
CPU

```
action <core_mem2mem_a :
  mem_copy_a {...}
```



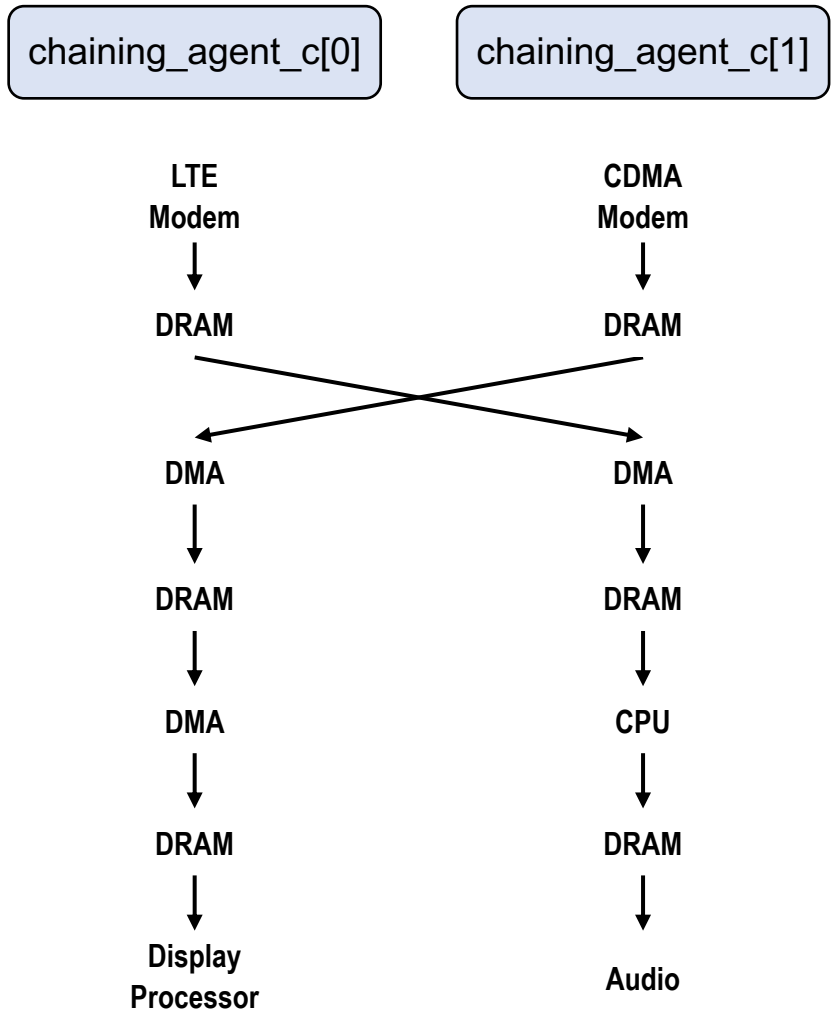
シーケンシャルなチェイニング

```
action sequential_chaining_a {
  activity {
    // Source
    select {
      [10] : do lte_source_a;
      [20] : do cdma_source_a;
      [10] : do camera_source_a;
    }
    // Memory2Memory
    replicate (2) {
      select {
        do core_mem2mem_a;
        do dma_mem2mem_a;
        do bluetooth_mem2mem_a;
      }
    }
    // Sink
    do display_sink_a;
  }
}
```



パラレルのチェイニング

```
extend component pss_top {
  action entry {
    schedule {
      replicate (5) {
        do sequential_chaining_a;
      }
    }
  }
}
```



チェイニングの組合せのカバレッジ

```
covergroup chaining_cg {
  source: coverpoint source_id;
  mem2mem_0: coverpoint mem2mem_id0;
  mem2mem_1: coverpoint mem2mem_id1;
  sink: coverpoint sink_id;
  cross chain: source,
               mem2mem_0,
               mem2mem_1,
               sink_id,
               size;
}
```

```
action sequential_chaining_a {
  rand source_e source_id;
  rand mem2mem_e mem2mem_id0, mem2mem_id1;
  rand sink_e sink_id;
  rand int size;
  activity {
    // Source
    select {
      [10] : do let_source_a with {
                 size == this.size;
                 this.source_id == LET_SOURCE; }
      [20] : do cdma_source_a with {
                 size == this.size;
                 this.source_id == CDMA_SOURCE; }
      [10] : do camera_source_a with {
                 size == this.size;
                 this.source_id == CAMERA_SOURCE; }
      }
      // Memory2Memory
      ...
      // Sink
      ...
  }
}
```

ストリーミングシナリオのインターリーブ制御と網羅

単純なストリームのインターリーブ例

DMA Pipeline (2ステップ)

1. move (Mx – xはステップが属する転送番号)
2. wait (Wx – サイズなどの属性をもとに時間を消費)

このようなシナリオをモジュール化し、ポータブルなフレームワークによって制御、網羅することが、多くの検証やバリデーションで求められている

シナリオ0	シナリオ1	シナリオ2	シナリオ3
M0	M0	M0	M0
M1	W0	M1	M1
M2	M1	M2	W1
W0	W1	W2	M2
W1	M2	W1	W0
W2	W2	W0	W2

コントローラ（例：組込コア、AXIバス）が１つの場合、N個のDMAストリーム転送を管理する必要がある。
例えば左の図では３つのDMA転送に対する4つの可能なシナリオを示している。

シナリオ0：最も多くのDMAチャネルを使用し、コーナーケースのバグを発見することができる

シナリオ3：DMAチャンネルが2つしかない場合に使用可能

シナリオ2：DMAの特定のトランザクションタイプにおいて、最良のパフォーマンス結果を得られる可能性がある

DMAストリームのインターリーブシナリオ - SV

```
while (j < NOF_SCENARIOS) begin
  // Randomly pick a new step at a time, accounting for transfer steps that
  // have already been picked. There is no obvious set of constraints that
  // can guarantee: (1) Exactly 2 steps will be picked for each transfer
  // (2) first step will appear before second step for all transfers.
   while (i < NOF_TRANSFERS*2) begin
    if (available(move) and available(wait))
      step = $random % 2; // 2 available steps
    else if(available(move))
       step = 0;
    else
      step = 1;
    scenario_step[i].type=step
    scenario_step[i].transfer_id=get_available_transfer_id(step);
     i++;
  end
  if (!exists(scenario_step) begin  add_scenario(scenario_step);
    j++;
  end
end
```

PSSではシナリオの制御、調整、混在化、網羅、移植を宣言的に表現できる

アルゴリズムは、様々なシナリオを提供する上で効率的ではない

シナリオを制御するためには、疑似コードで使用している関数を変更する必要がある

ステップ数が増えたら、別の異なる実装が必要になる

同じリソースの一部を使用する他のスティミュラスと混在させることができない

組込みコア用のC言語実装コードに移植ができないため、書き直す必要がある。

DMAストリームのインターリーブをモデリング

```
package dma_pkg {
  const int NOF_TRANSFERS = 4;
  state state_s {
    rand bit move [NOF_TRANSFERS] ;
    rand bit wait [NOF_TRANSFERS] ;
    constraint initial -> {
      foreach (move[i]) {
        move[i] == 0;
        wait[i] == 0;
      }
    }
  }
}
```

state オブジェクトの
フィールドを初期化

```
action move_a {
  input state_s in_s;
  output state_s out_s;
  rand int transfer_num;
  constraint out_s.move[transfer_num] == 1'b1;
  constraint {
    foreach(out_s.move[i]) {
      out_s.wait[i] == in_s.wait[i];
      if (i != transfer_num) {
        out_s.move[i] == in_s.move[i];
      }
    }
  }
}
action wait_a {
  input state_s in_s;
  output state_s out_s;
  rand int transfer_num;
  constraint out_s.wait[transfer_num] == 1'b1;
  constraint {
    foreach(out_s.wait[i]) {
      out_s.move[i] == in_s.move[i];
      if (i != transfer_num) {
        out_s.wait[i] == in_s.wait[i];
      }
    }
  }
}
```

MOVEの転送が
終わった事の印

他のstateの値が
変化しない
という制約を設定

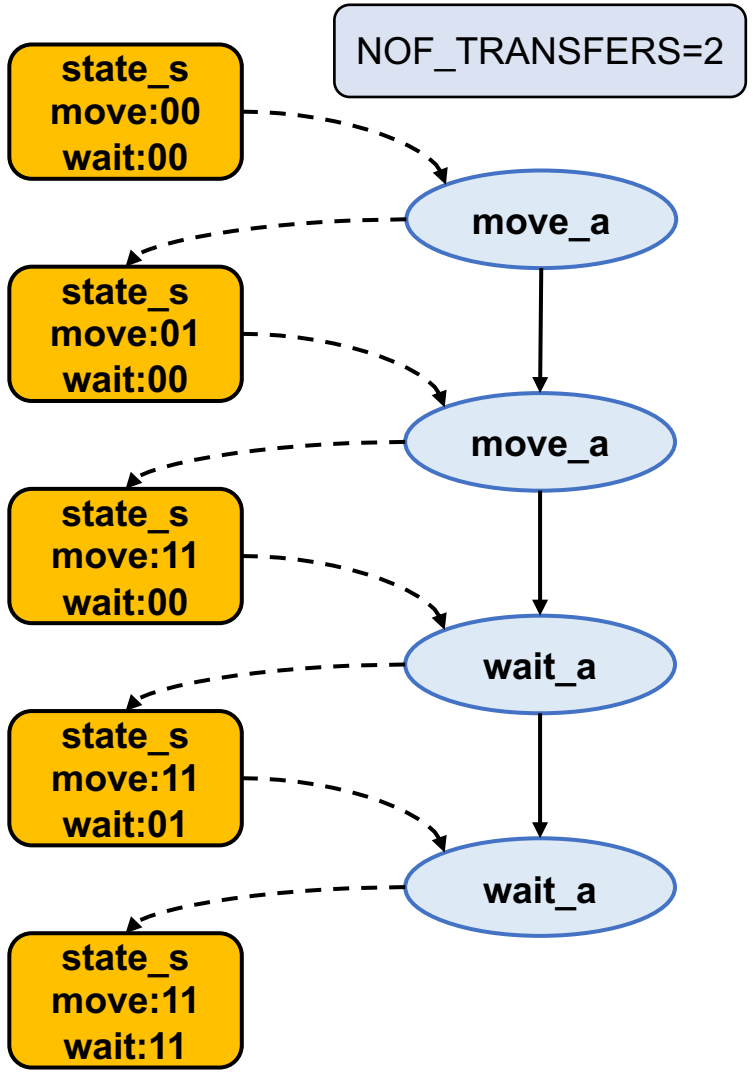
WAITの転送が
終わった事の印

他のstateの値が
変化しない
という制約を設定

インターリーブ時のDMAステップのスケジュール

scheduleオペレータが
複数回の転送の間に
MとWをインターリーブ

```
component dma_c {
  import ip0_pkg::*;
  pool state_s state_p;
  bind state_p *;
  action dma_transfer_a {
    rand int transfer_num;
    move_a M;
    wait_a W;
    activity {
      M with {transfer_num == this.transfer_num; }
      W with {transfer_num == this.transfer_num; };
    }
  }
  action all_dma_transfers_a {
    dma_transfer_a DT[NOF_TRANSFERS];
    activity {
      schedule {
        replicate (i:NOF_TRANSFERS) {
          DT[i] with {transfer_num == i;} ;
        }
      }
    }
  }
}
```



制約の上書きも簡単

```
extend action wait_a {
  constraint countones(in_s.move) == NOF_TRANSFERS;
}
```

すべての転送 (move) が終わってから待機 (wait) する

```
action move_a {
  input state_s in_s;
  output state_s out_s;
  rand int transfer_num;
  constraint out_s.move[transfer_num] == 1'b1;
  constraint {
    foreach(out_s.move[i]) {
      out_s.wait[i] == in_s.wait[i];
      if (i != transfer_num) {
        out_s.move[i] == in_s.move[i];
      }
    }
  }
}
action wait_a {
  input state_s in_s;
  output state_s out_s;
  rand int transfer_num;
  constraint out_s.wait[transfer_num] == 1'b1;
  constraint {
    foreach(out_s.wait[i]) {
      out_s.move[i] == in_s.move[i];
      if (i != transfer_num) {
        out_s.wait[i] == in_s.wait[i];
      }
    }
  }
}
```

インターリーブするストリームのリソースパイプライン

タスクを複数のサブタスクにパイプライン化し、スティミュラスとのやり取りを要するIP用のスティミュラスをどのようにモデルリングするか

異なるサブタスクのインターリーブを可能にするIP用のマルチステップ・スティミュラスのモデリング

リソースがパイプラインの異なるステージで解放された際に異なるストリームで共有されるようにモデリング

異なる複数のタスクが存在する際の意味深いインターリーブシナリオのカバレッジ

具体例：

- 複数のチャネルリソースを持ち、MOVEと転送完了を待つWAITをサブタスクとして持っているDMA IPの例。DMA転送ではMOVE、WAITともに同じチャネルを使用する必要がある。
- PIPE、OVERLAY、INTERFACEのリソースを持っているDISPLAY IPの例。それぞれのリソースを処理するサブタスクが存在する。

パイプライン化スティミュラス – PSSモデリング

State Flowオブジェクトと共にScheduleオペレータを使うことで、パイプライン化シナリオの生成とキャラクタライズが可能なモデルとなる

actionの命名則:
stream_step_<i>_a

ここで <i>は0からストリームで必要なサブタスク数-1を取る
（例：DMAでは２つ、
DISPLAYでは3つ）

各サブタスクに必要となる resource オブジェクトの使い方

入力の state オブジェクトに追加の制約を加えてパイプライニングを制御する

covergroup をモデリングしサンプリングすることで、生成されたテストのインターリーブをキャラクタライズする

3-Step Display パイプラインのモデリング

```
package display_pkg {
  resource display_engine_step0_r {}
  resource display_engine_step1_r {}
  resource display_engine_step2_r {}
   state state_s {
     rand bit step0 [NOF_TRANSFERS] ;
     rand bit step1 [NOF_TRANSFERS] ;
     rand bit step2 [NOF_TRANSFERS] ;
     rand bit [NOF_TRANSFERS] step0_b;
     rand bit [NOF_TRANSFERS] step1_b;
     rand bit [NOF_TRANSFERS] step1_b;
     constraint {foreach([step0[i]) {
        step0_b[i] == step0[i];
        step1_b[i] == step1[i];
        step2_b[i] == step2[i];
     }}
     constraint initial -> {
       foreach (step0[i]) {
         step0[i] == 0;
         step1[i] == 0;
         step2[i] == 0;
     }
   }
}
```

```
component display_c {
  action stream_step0_a {
    input state_s in_s;
    output state_s out_s;
    rand int transfer_num;
    lock display_engine_step0_r engine_1;
    constraint out_s.step0[transfer_num] == 1'b1;
    constraint {
      foreach(out_s.step0[i]) {
        out_s.step1[i] == in_s.step1[i];
        out_s.step2[i] == in_s.step2[i];
        if (i != transfer_num) {
          out_s.step0[i] == in_s.step0[i];
        }
      }
    }
    covergroup {
      step0: coverpoint in_s.step0_b;
      step1: coverpoint in_s.step1_b;
      step2: coverpoint in_s.step2_b;
      all: cross step0, step1, step2 {
        ignore_bins interleaving = all with (
          ((step0 | step1) == step0) &&
          ((step1 | step2) == step1)        );
      }
    } cg;
  }
}
```

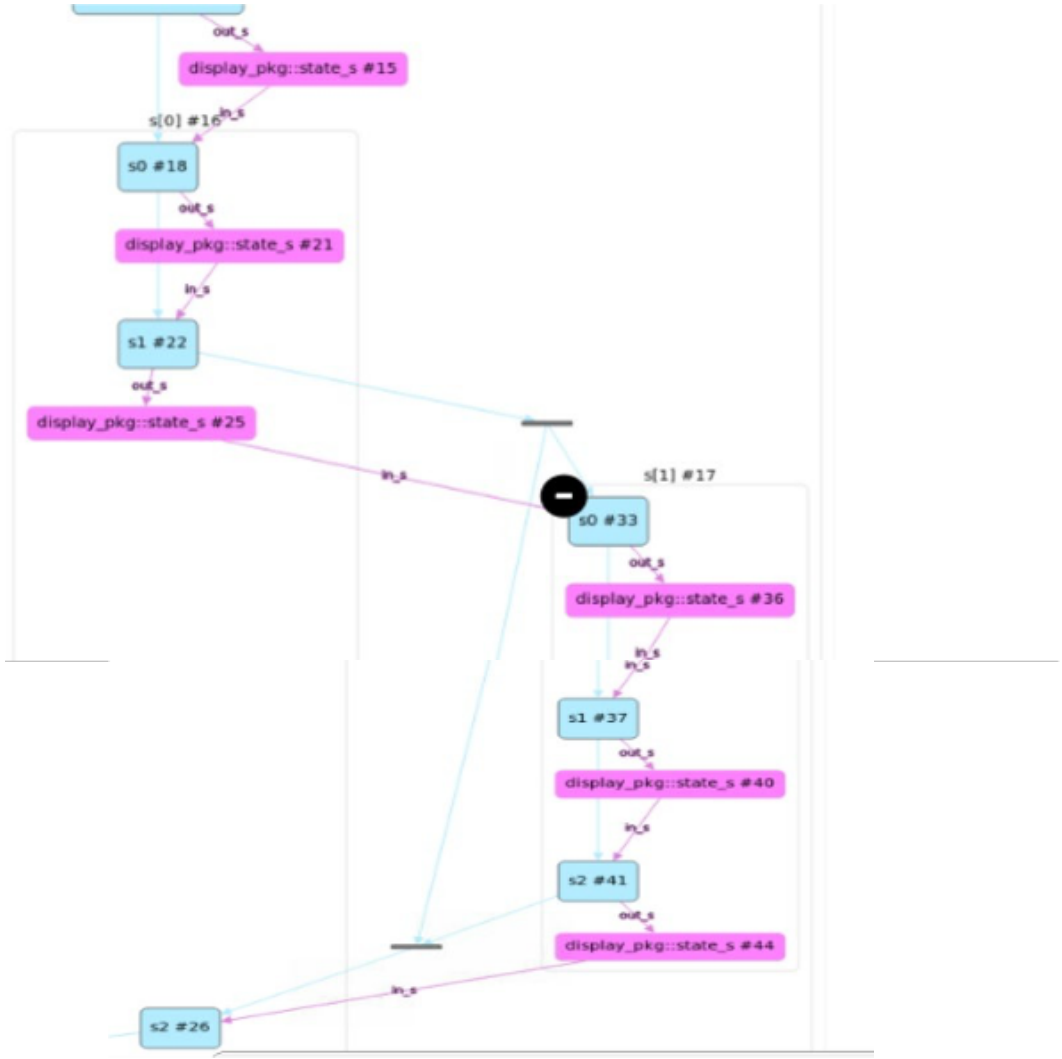
Stepに応じてリソースをロック
1つ目のリソースはPIPE、
2つ目はOVERLAY、
3つ目はINTERFACE

Stepごとのカバレッジを
サンプリングし
そのクロスで特徴づけられる
全インターリーブを
実現することがゴール

詳細は次スライド・・・

インタリーブされるステップのスケジューリング

```
component display_c {
  import display_pkg::*;
  pool state_s state_p;
  bind sample_state_p *;
  action stream_a {
    rand int transfer_num;
    stream_step0_a s0;
    stream_step1_a s1;
    stream_step2_a s2;
    activity {
      s0 with {transfer_num == this.transfer_num;}
      s1 with {transfer_num == this.transfer_num;}
      s2 with {transfer_num == this.transfer_num;}
    }
  }
  action all_stream_a {
    stream_a s[NOF_TRANSFERS];
    activity {
      schedule {
        replicate (i:NOF_TRANSFERS) {
          s[i] with {transfer_num == i;} ;
        }
      }
    }
  }
}
```



パイプラインでインターリーブするストリームのモデリング（リソースがステップから解放される際に共有される）

```
extend component soc_c {
  import display_pkg::*;
  pool [6] display_engine_step0_r display_engine_step0_p;
  bind display_engine_step0_p *;
  pool [4] display_engine_step1_r display_engine_step1_p;
  bind display_engine_step1_p *;
  pool [5] display_engine_step2_r display_engine_step2_p;
  bind display_engine_step2_p *;
  action all_ip_all_tasks_a {
    activity {
      schedule {
        replicate (i:DISPLAY_STREAMS) {
          do display_c::all_stream_a
                  with {comp == pss_top.soc.display[i];};
        }
      }
    }
  }
}
```

DISPLAYリソースはプールされ
複数のコントローラで共有可能

あるコントローラが
リソースを解放すると
別のコントローラが
リソースを取得できるため
リソースを持っていた
ストリームの終了を
待つ必要がない

DISPLAY例からリソース制約を汎用ストリームのインターリーブ・パイプライン・パターンに適用

```
package display_pkg {
  enum pipe_kind_e {VID, GFX};
  extend resource display_engine_step0_r {
    rand pipe_kind_e kind;
    constraint {
      kind == VID -> instance_id in [0..2];
      kind == GFX -> instance_id in [3..5];
      }
    }
  enum overlay_kind_e {LCD0, LCD1, LCD2, TV};
  extend resource display_engine_step1_r {
    rand overlay_kind_e kind;
    constraint instance_id == int(kind);
  }
  enum interface_kind_e {DSI_A, DSI_B, DP_A, DP_B, HDMI};
  extend resource display_engine_step2_r {
    rand interface_kind_e kind;
    constraint instance_id == int(kind);
  }
}
```

```
extend action display_c::stream_a {
  constraint pipe2overlay_c {
    s0.engine_1.kind == GFX -> s1.engine_1.kind != LCD0;
    s0.engine_1.kind == VID -> s1.engine_1.kind != LCD1;
  }
  constraint overlay2interface_c {
    s1.engine_1.kind == LCD0 -> s2.engine_1.kind == DP_A;
    s1.engine_1.kind == LCD1 -> s2.engine_1.kind in [DSI_A, DSI_B];
    s1.engine_1.kind == LCD2 -> s2.engine_1.kind in [DSI_A, DSI_B, DP_B];
    s1.engine_1.kind == TV -> s2.engine_1.kind in [DP_A, HDMI];
   }
}
```

インターリーブ・シナリオのカバレッジ

```
covergroup {
  step0: coverpoint in_s.step0_b;
  step1: coverpoint in_s.step1_b;
  all: cross step0, step1 {
         ignore_bins interleaving = all with (((step0 | step1) != step0));
       }
} cg;
```

	move	wait
Pt #1	1111	0000
Pt #2	1111	0011
Pt #3	0011	0001

転送前に始まった待機は無視する

複数IPのパイプラインタスクを組合わせたSOCシナリオ

```
extend component soc_c {
  import dma_pkg::*;
  pool [8] dma_engine_step0_r dma_engine_step0_p;
  bind dma_engine_step0_p *;
  pool [8] dma_engine_step1_r dma_engine_step1_p;
  bind dma_engine_step1_p *;

  import display_pkg::*;
  pool [6] display_engine_step0_r display_engine_step0_p;
  bind display_engine_step0_p *;
  pool [5] display_engine_step1_r display_engine_step1_p;
  bind display_engine_step1_p *;
  pool [4] display_engine_step2_r display_engine_step2_p;
  bind display_engine_step2_p *;

  action all_ip_all_tasks_a {
    activity {
      parallel {
        schedule {
          replicate (i:DISPLAY_STREAMS) {
            do display_c::all_stream_a with {comp == pss_top.soc.display[i];};
          }
        }
        schedule {
          replicate (i:DMA_STREAMS) {
            do dma_c::all_stream_a with {comp == pss_top.soc.dma [i];};
          }
        }
      }
    }
  }
}
```

異なるIPからなる
インターリーブシナリオは
互いに独立して
実行される

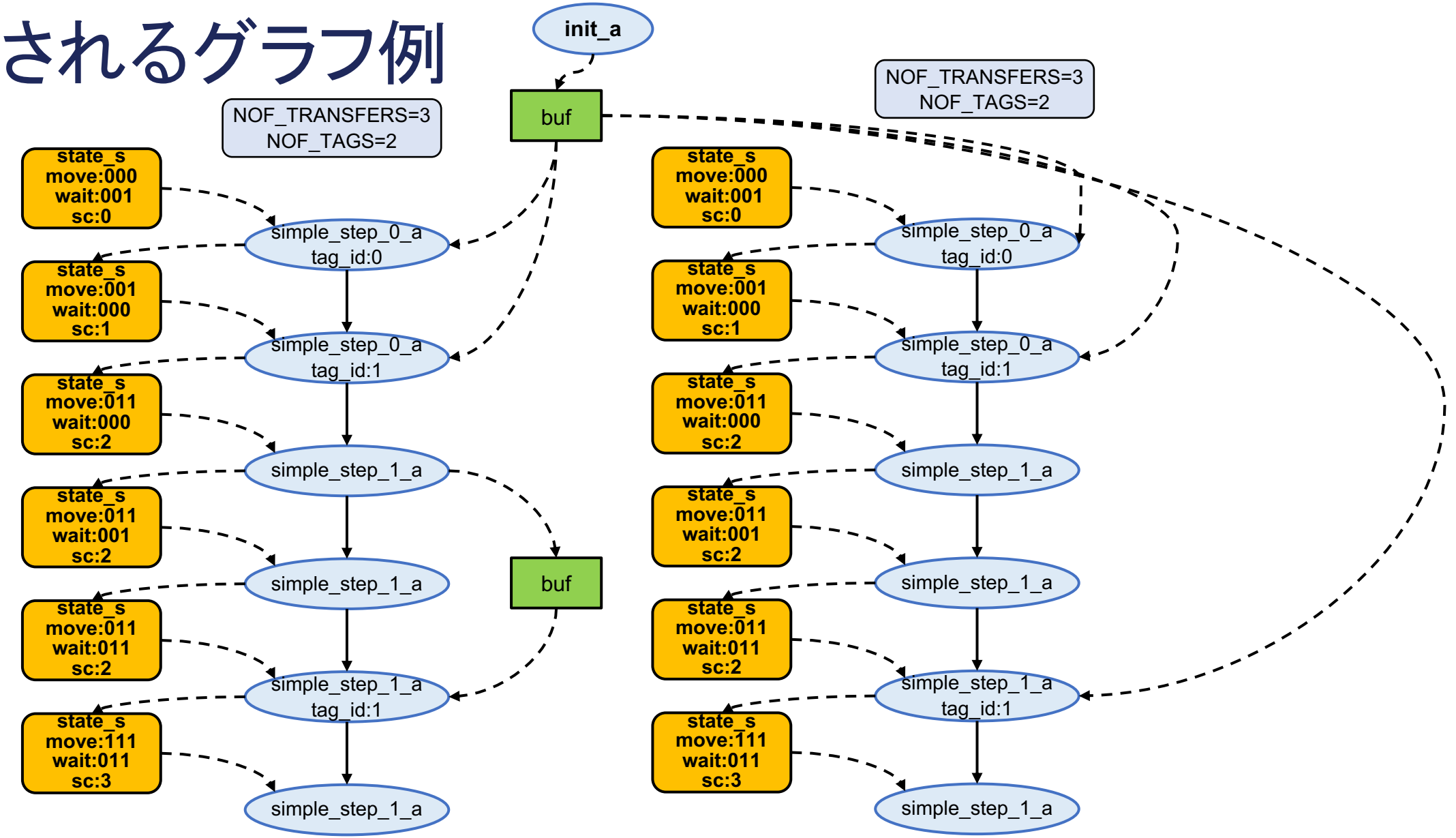
すべてのパイプラインをチェイニング

```
action simple_a {
  input buf in_buf;
  output buf out_buf;
  rand int transfer_num;
  simple_step_0_a s0;
  simple_step_1_a s1;
  simple_step_2_a s2;
  activity {
    s0 with {transfer_num == this.transfer_num; };
    s1 with {transfer_num == this.transfer_num; };
    s2 with {transfer_num == this.transfer_num; };
  }
}
```

パイプラインのストリームは
入力バッファからのデータを処理して
その結果を出力バッファに出力し
その後に異なるIPに消費される

これでチェイニングパターンと
インターリーブパターンを
組合わせることができる

生成されるグラフ例


init_a
buf
NOF_TRANSFERS=3
NOF_TAGS=2
state_s
move:000
wait:001
sc:0
simple_step_0_a
tag_id:0
state_s
move:001
wait:000
sc:1
simple_step_0_a
tag_id:1
state_s
move:011
wait:000
sc:2
simple_step_1_a
state_s
move:011
wait:001
sc:2
simple_step_1_a
buf
state_s
move:011
wait:011
sc:2
simple_step_1_a
tag_id:1
state_s
move:111
wait:011
sc:3
simple_step_1_a
NOF_TRANSFERS=3
NOF_TAGS=2
state_s
move:000
wait:001
sc:0
simple_step_0_a
tag_id:0
state_s
move:001
wait:000
sc:1
simple_step_0_a
tag_id:1
state_s
move:011
wait:000
sc:2
simple_step_1_a
state_s
move:011
wait:001
sc:2
simple_step_1_a
state_s
move:011
wait:011
sc:2
simple_step_1_a
tag_id:1
state_s
move:111
wait:011
sc:3
simple_step_1_a

シナリオのチェイニング／インターリーブ　まとめ

- チェイニング例を用いてIPのモデルを純粋にSOCに適用する手法を紹介した
- ストリームのインターリーブ／パイプライニングのシナリオに対して schedule と state flow オブジェクトを併用することで微調整する手法を紹介した
- ストリームのインターリーブ／パイプライニング例に対してPSSの宣言型構文の使用方法を示し、SystemVerilogでは困難または不可能であることを示した
- ストリームのチェイニングおよびインターリーブ／パイプライニングの組合せなど、異なるタイプのSoCシナリオを容易に生成できることを示した

アジェンダ

- PSSの概要と現在の状況
- ディスプレイコントローラの適用例
- メモリ＆キャッシュの適用例
- SoCレベルの適用例
- まとめ

PSSのまとめ

- IPテスト空間のフォーマルな指定
  - IPライフサイクルの早期からテスト空間のドキュメンテーションとして
- システム・シナリオのフォーマルなドキュメンテーション
  - SoCアーキテクチャを定義する際に作成可能
- 複数のIPのテストシナリオを簡単に構成
  - テストにおける state、resource、schedule の依存関係を自動ハンドリング
- 部分的なテストシナリオ
  - SoCの完全に深く理解しなくてもテスト生成が可能
- テスト生成時およびテスト実行時のカバレッジレポート
  - パワーステートの遷移、機能モードなどのカバレッジ
- テストそのものがポータブル
  - IP (UVM/SystemC) からSoCの組込みプロセッサ、ポストシリコンまで

まとめ

PSSは宣言型のフォーマルかつポータブルなテスト指定ができ
プラットフォーム固有のテストを自動的に生成することができる

PSS 2.0は今日現在、正式プロダクションリリースの状態

PSS 2.1およびその先に向けた優れたアイデアがある

PSSの未来を一緒にかたちにしていきましょう

2022
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
JAPAN

ご聴講
ありがとうございました

Any Questions ?

accellera
SYSTEMS INITIATIVE