2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

Portable Test and Stimulus Standard 標準の概要および標準化最新情報

Canon USA - 細川博司

accellera SYSTEMS INITIATIVE

PSS標準化とその背景にある課題・動機

PSS : Accellera 標準

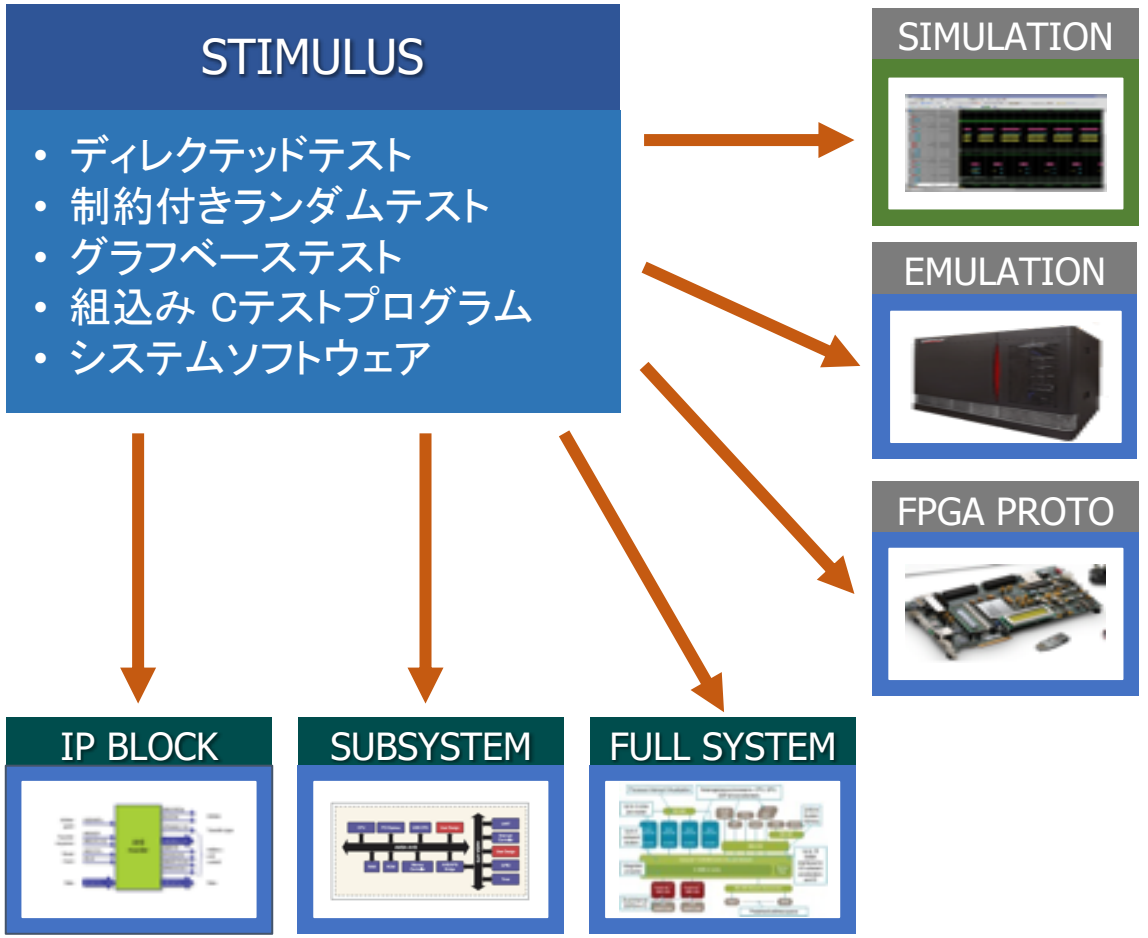
- PSS : Portable Test and Stimulus Standard
- 2014年12月にAccelleraで承認されたWG
  - Chair: Faris Khundakjie, Intel
    Vice Chair: Tom Fitzpatrick, Mentor Graphics
    Secretary: Tom Anderson, AMIQ EDA
- リリースされた言語仕様書
  - 2018年6月26日 PSS Language Reference Manual 1.0
  - 2019年2月25日 PSS Language Reference Manual 1.0a
  - 2021年4月14日 PSS Language Reference Manual 2.0
  - 2.1に向けて仕様策定作業が進んでいる

PSS WGのスコープ

- Currently there is no single standard way to specify intent and behaviors that is reusable across target platforms (e.g., emulation, silicon, simulation, etc.). With this proposed standard, user companies will be able to specify the behaviors once, from which multiple implementations may be derived.
  With a single specification, user companies will be able to select the best tool(s) from competing vendors to achieve the best results for their desired target platform. Initial scope for the WG will be to define a portable test and stimulus specification language that can be used to generate stimulus for multiple target implementations.
- 現在、ターゲットプラットフォーム（エミュレータ、シリコン、シミュレータなど）間で再利用可能な、テストインテントと動作を指定する単一の標準的な方法は存在しません。
  この標準規格を使うと、ユーザー企業は動作を一度指定すれば、そこから複数のテスト実装を派生させることができます。また標準仕様をサポートする複数の競合するベンダーの中から最適なツールを選択し、希望するターゲットプラットフォームで最良の結果を得ることができるようになります。このWGの最初の目的は、複数のターゲット実装のためのスティミュラス生成に使用でき、移植性のあるテストとスティミュラス仕様の言語を定義することです。

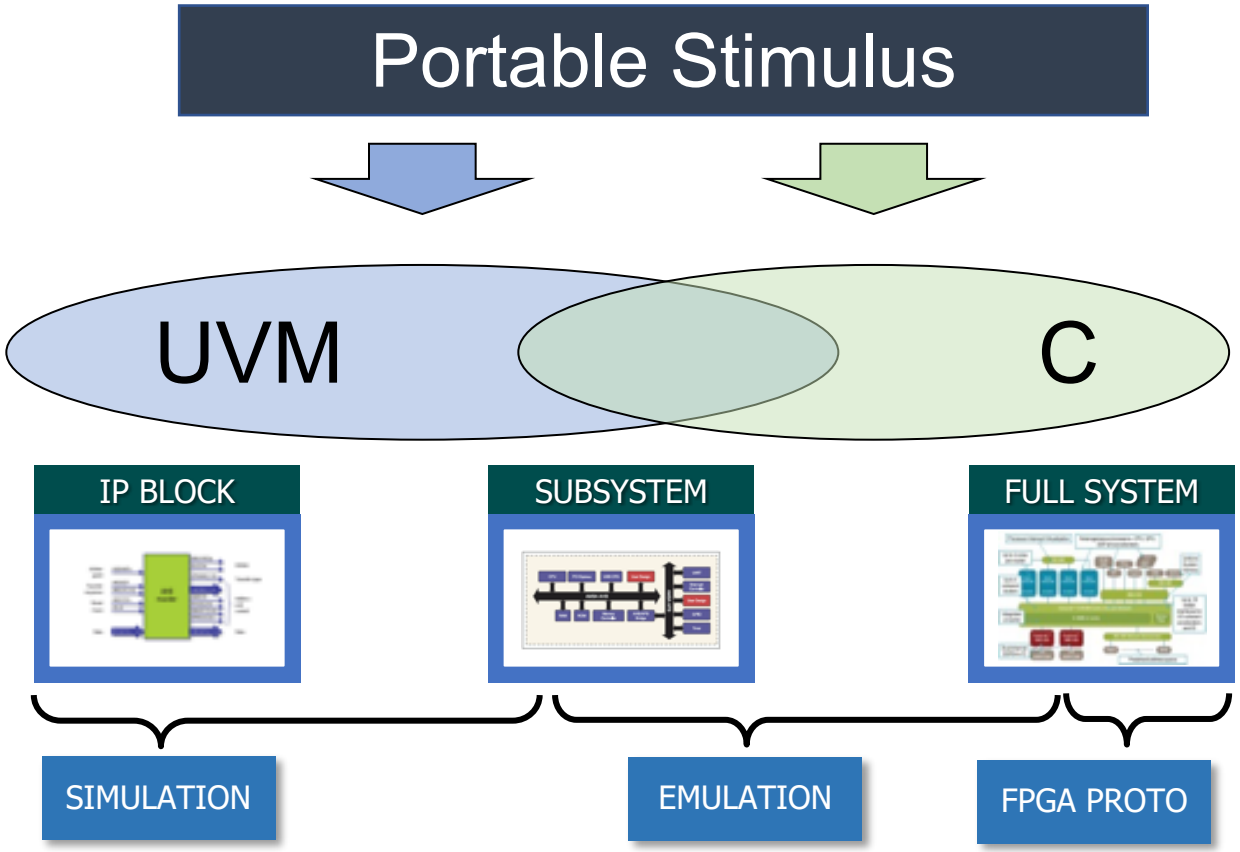
プロジェクトで行われている検証の実態

- 異なる複数の検証環境に対応
  - プロジェクトをとおして異なる複数のテストが使用される
  - UVMの制約付きランダム検証はSoC検証では限定的
  - Cテストはディレクテッドのみ
- 求められるソリューション
  - ブロックからシステムまでの再利用
  - 異なるプラットフォームへの移植性
  - より高いテスト抽象度
    - スティミュラス→シナリオ→インテント

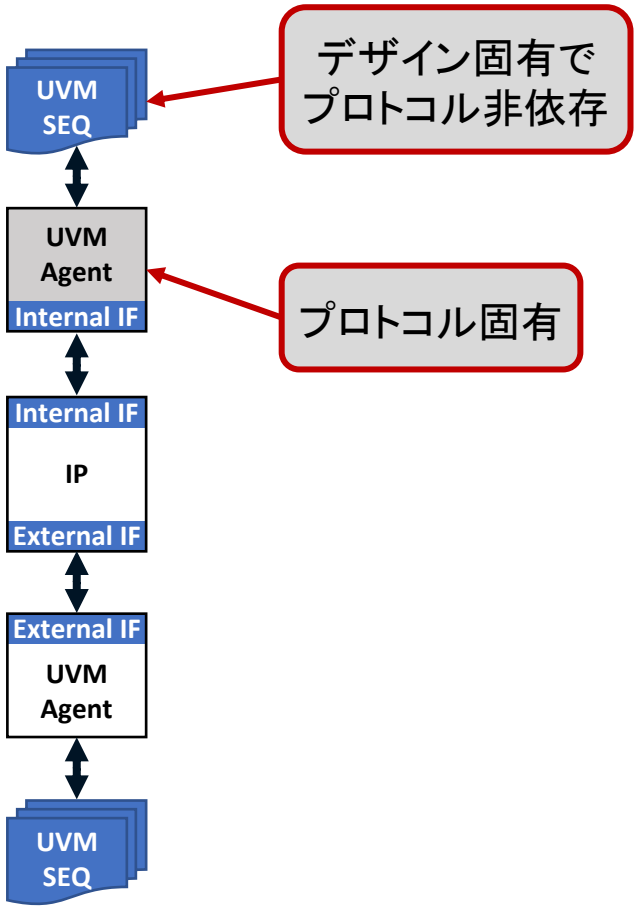


環境やユーザを超えたテストインテント

- テストインテントを一度だけ指定すれば済むという環境が理想
- シナリオ空間を定義する
  - 機能間の相互的な作用
  - 依存性
  - リソース競合
- 高い抽象度モデルの指定によりツールによる自動生成を可能に
  - 複数の異なるターゲット
  - ターゲット固有のカスタマイズ

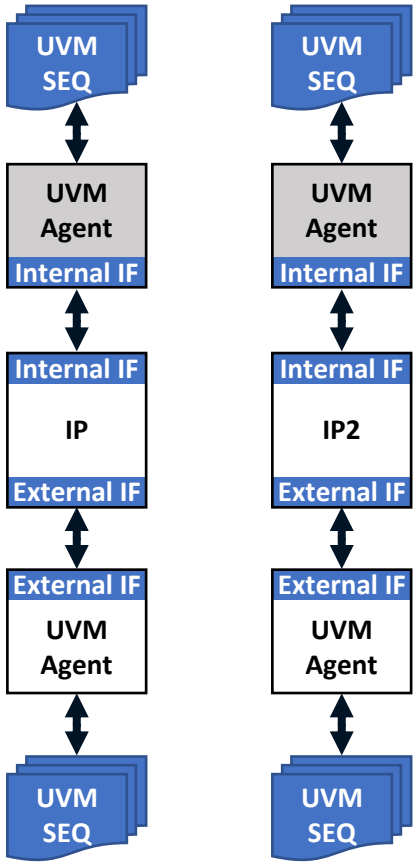


UVMは“How”－どう検証するかが焦点



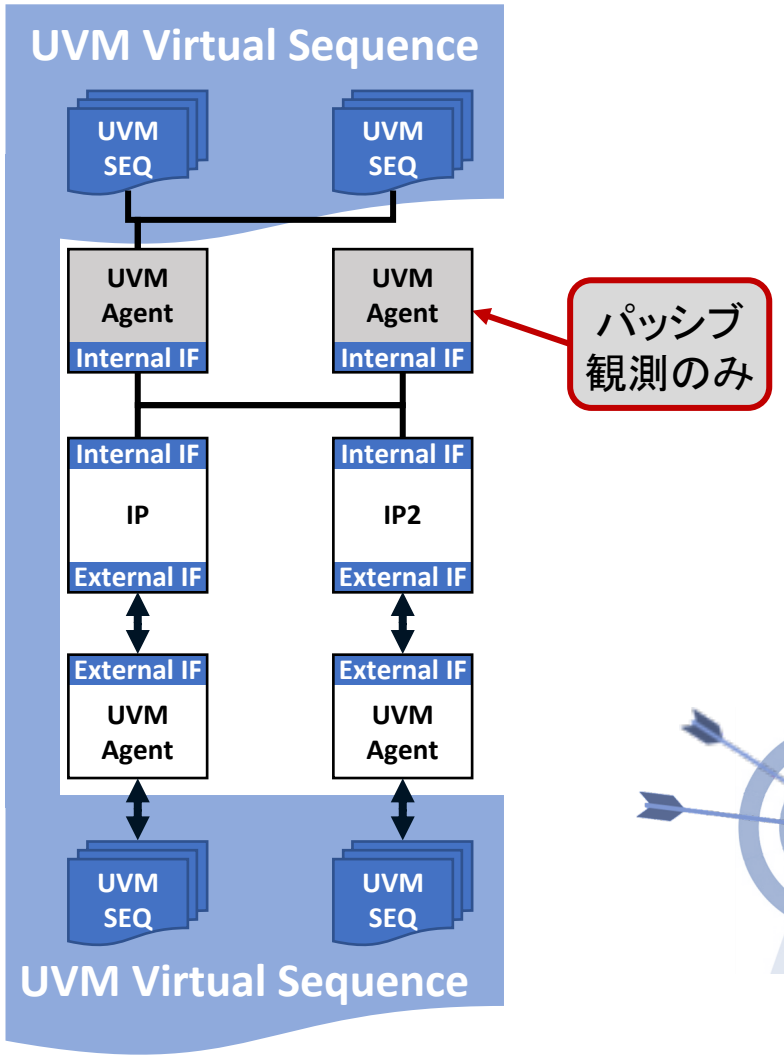
- ブロックレベル検証に最適
- モジュラー化され再利用可能な検証コンポーネント
- What と How の分離
  - 構成可能なトランザクションレベルのシーケンス
  - ドライバ／モニタがトランザクションと信号間を変換
- 抽象的なシーケンスを異なるエージェントに適用
  - シーケンスは“What”
  - エージェントはプロトコル固有の“How”
- IPのインタフェースは同じアプローチを用いる

UVMは“How”－どう検証するかが焦点

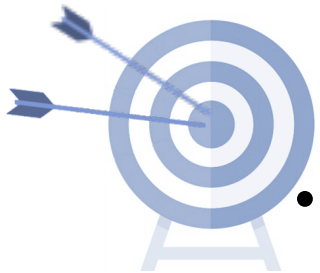


- すべてのテストやテストベンチは共通の基本構造を持つ
- 水平方向の再利用
  - 同じインタフェースを持つ異なるブロックで共通のエージェントが使用できる

UVMは“How”－どう検証するかが焦点

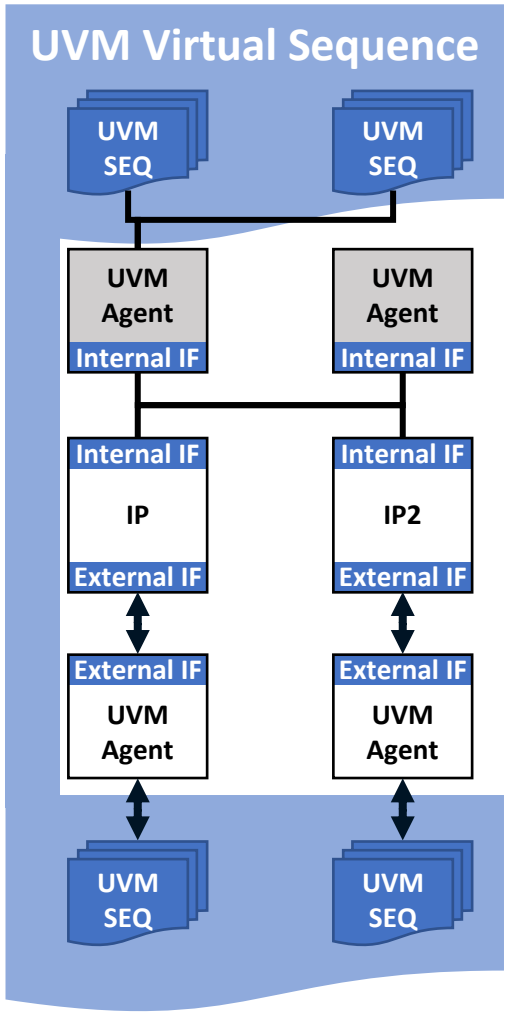


- すべてのテストやテストベンチは共通の基本構造を持つ
- 水平方向の再利用
  - 同じインタフェースを持つ異なるブロックで共通のエージェントが使用できる
  - ブロックレベルの環境をコンフィギュレーションし、コンポーネントを再利用する
  - ブロック用のテストはVirtual Sequenceから呼出される

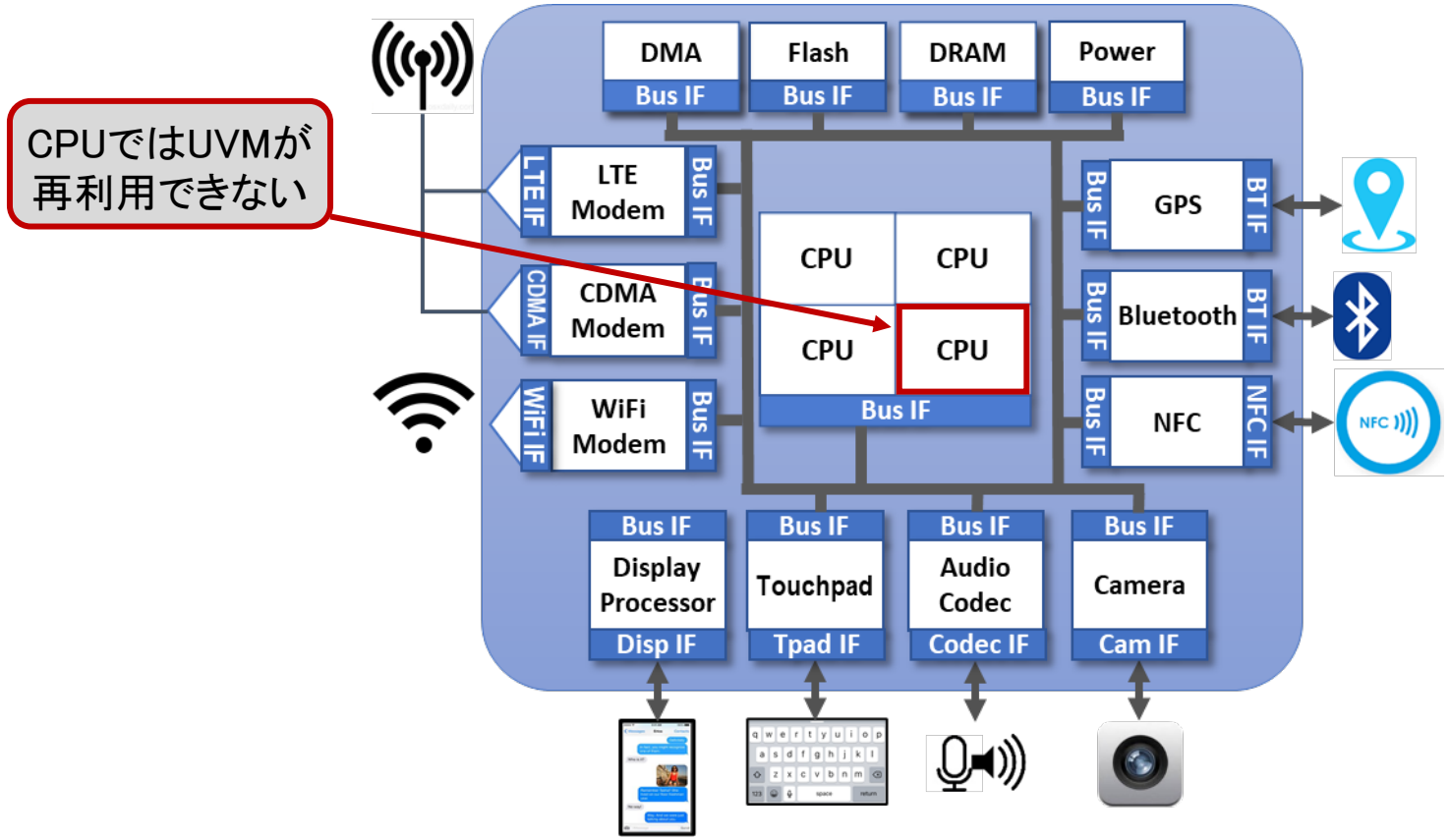


- 10年前はこれが正しいターゲットだった

ターゲットが変わるのが検証



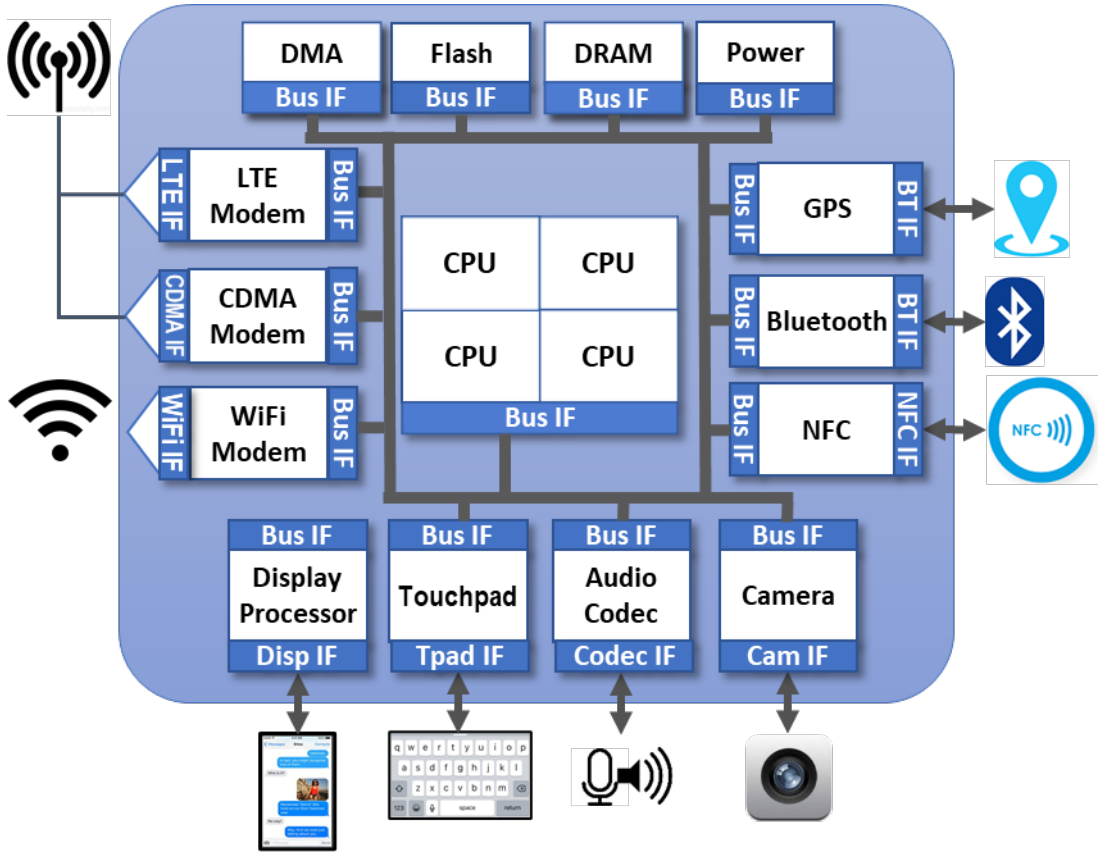
- ブロックレベルのテストはSoCにスケールしない



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

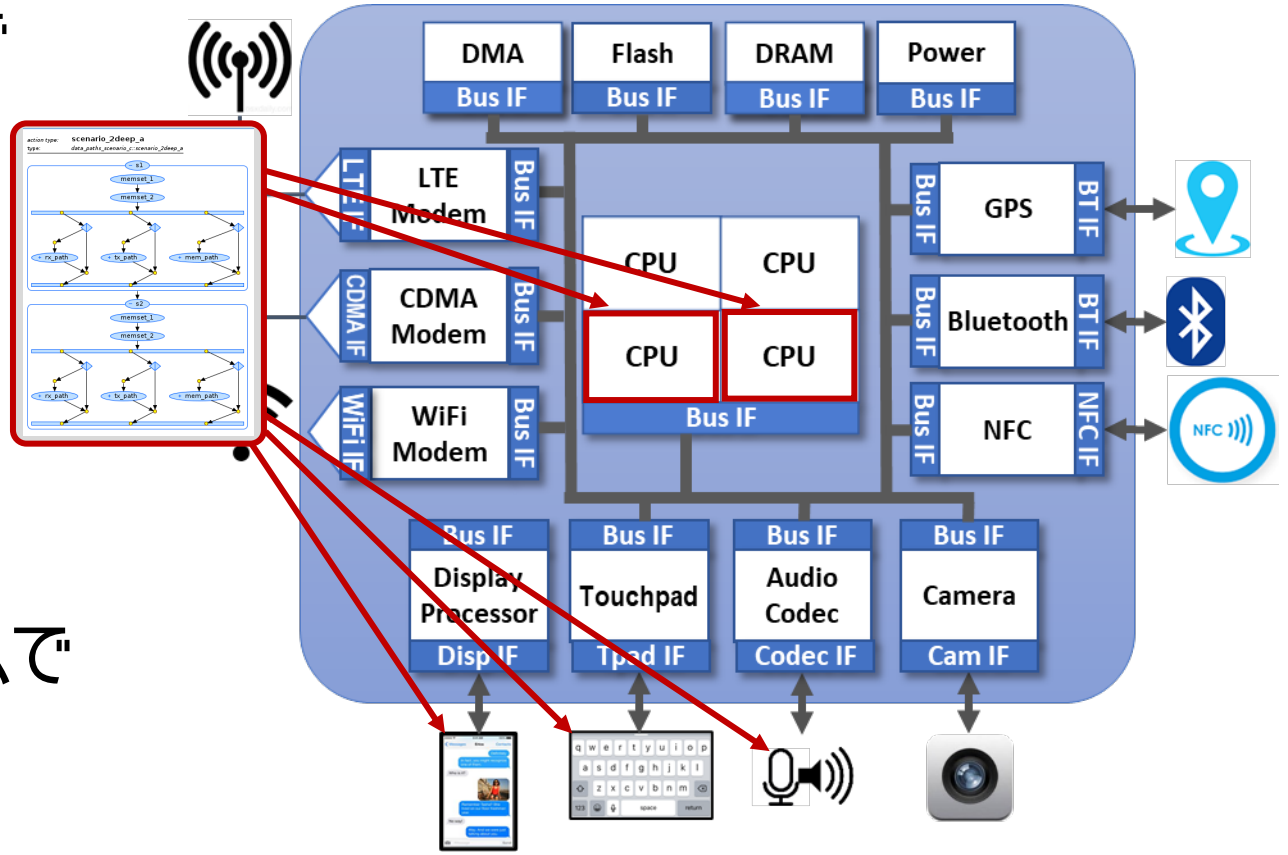
組込みプロセッサを含むSoCの検証

- 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



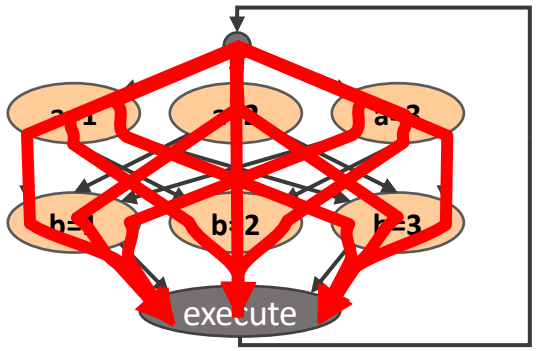
SoC検証に対して自動化が適用できたら

- 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

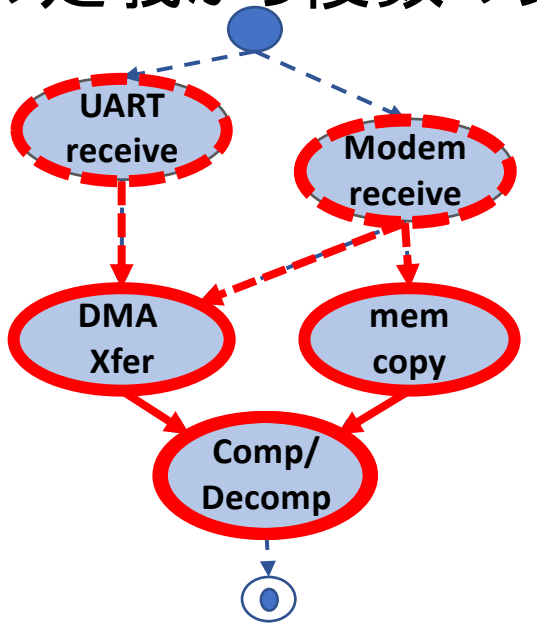


より高位におけるスティミュラス

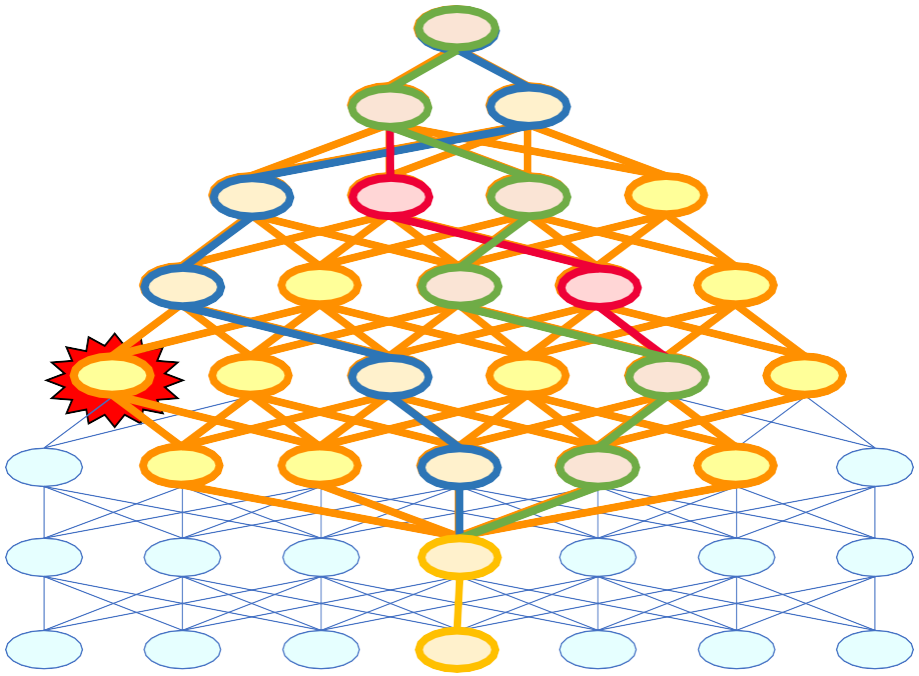
- UVM Sequenceはトランザクションのセットを定義
  - トランザクション内容をランダム化
  - トランザクションのフローは明示的にランダム化できない
- シーケンス間におけるランダム化が難しい



- 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

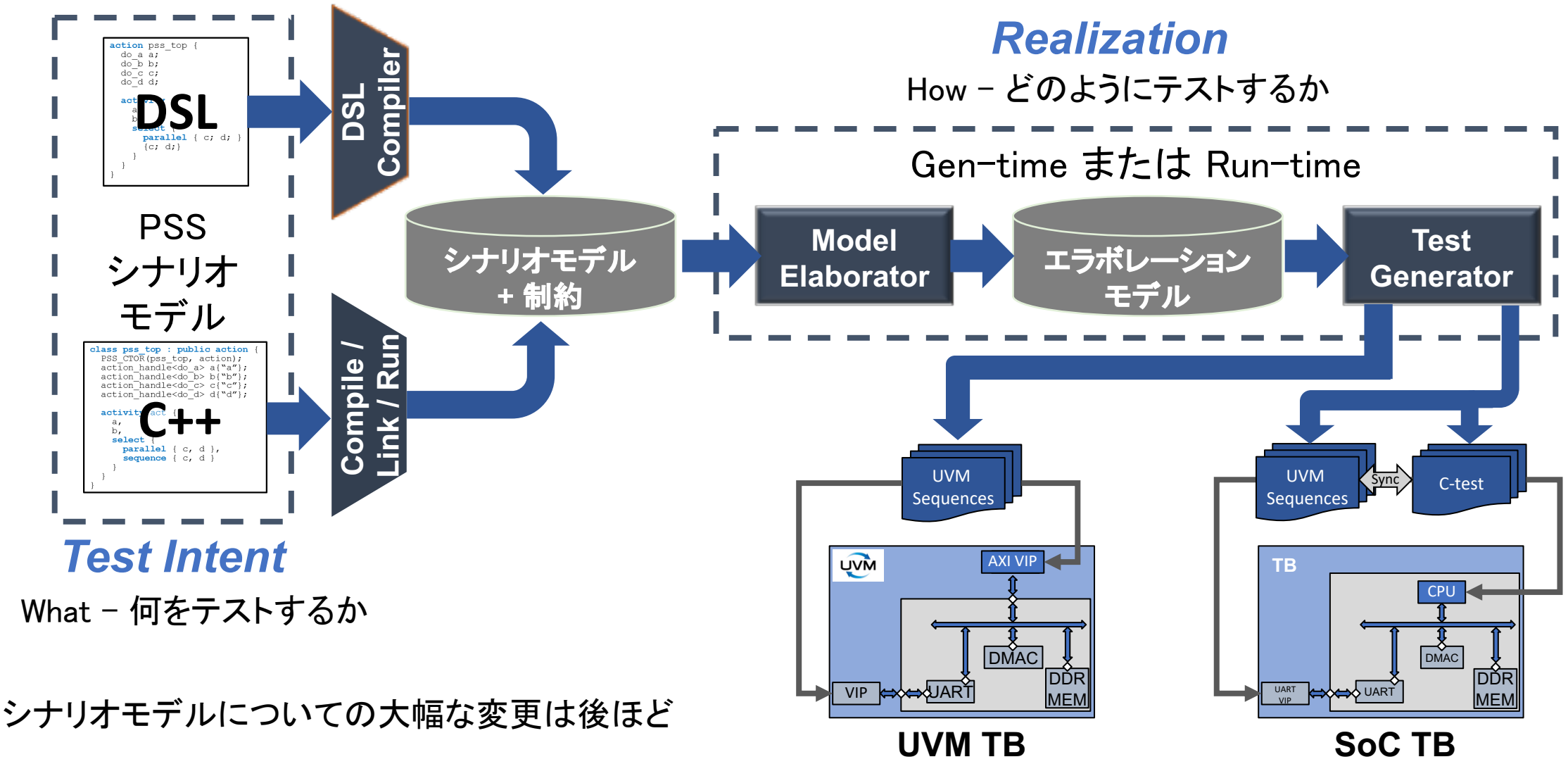
複数の実装を
ターゲットに

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

PSS標準が想定しているツールフロー



※シナリオモデルについての大幅な変更は後ほど

PSSモデルの主要要素とテストインテント

PSS モデル

PSSモデルは 動作を表現する要素 と パッシブなオブジェクト から構成される

action ***dataflow, resource, state, etc***

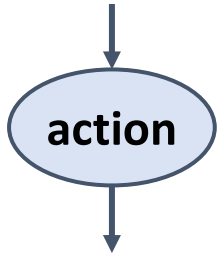
クラス定義

正しいテスト空間を構成する PSSモデルの インスタンス = ***scenario***

Test Realization として PSSから呼ばれる実行部分 = ***target platform***

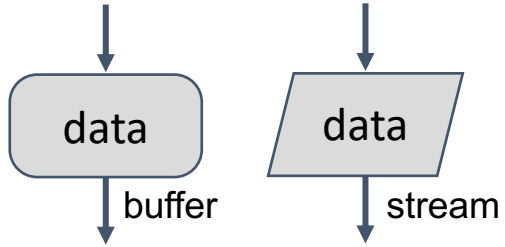
PSSモデルを構成する主要要素

- action - 動作をエンキャプスレーションしたもの
  複合 action でインテントを表現する
  - DMA転送
  - パケットの送信／受信
  - ビデオのデコード
  - ・・・

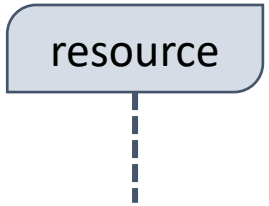



```
action dma_m2m_a {
  input  mem_buf src_data;
  output mem_buf dst_data;
  lock   chan_r  chan;
  ...
}
```

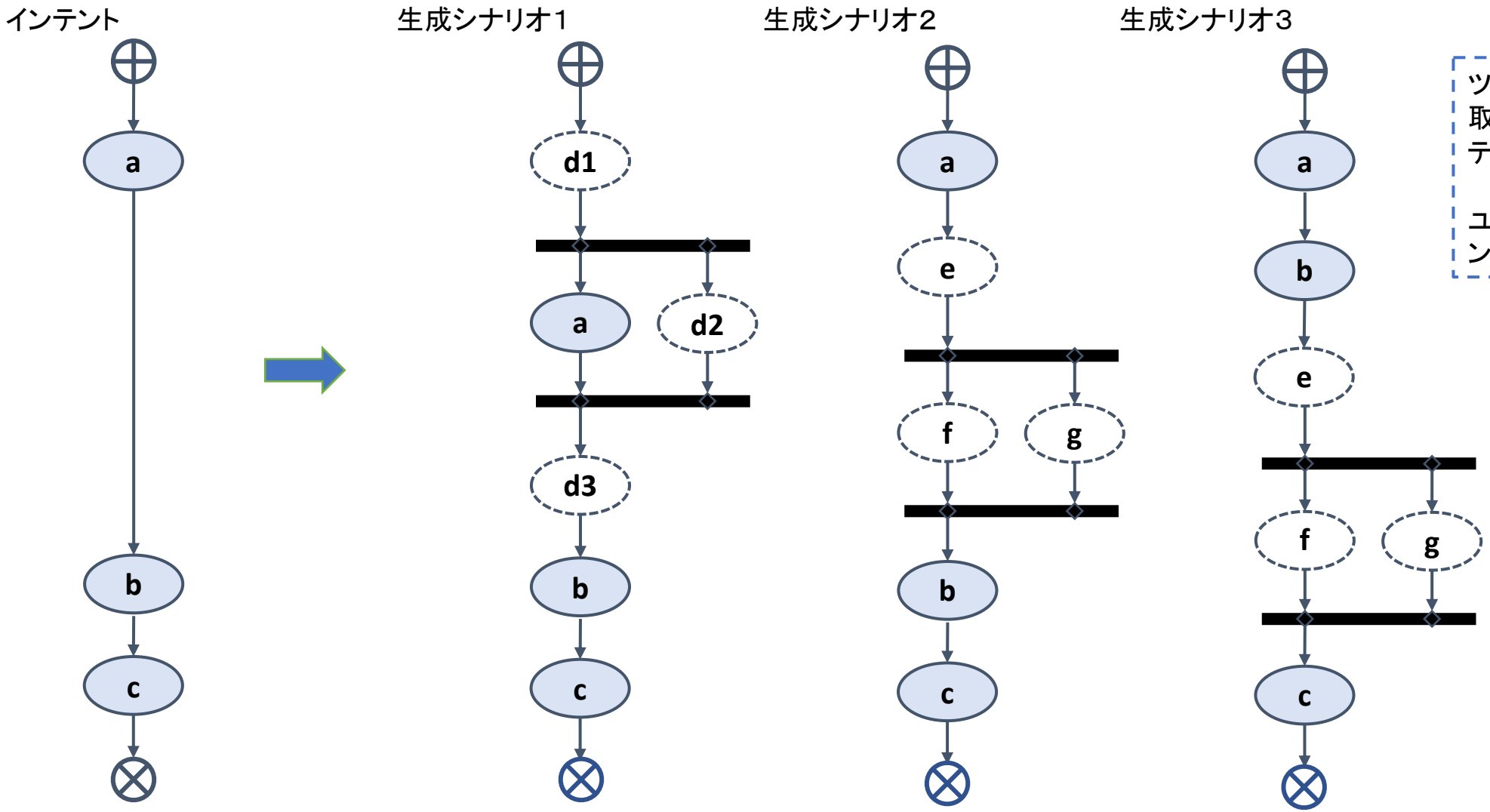
- data flowオブジェクト - actionが扱うデータフロー
  - バッファされるデータ
  - ストリーミングデータ
  - ・・・



- resource object - actionを実行するリソース
  - DMAチャンネル
  - CPU
  - ・・・



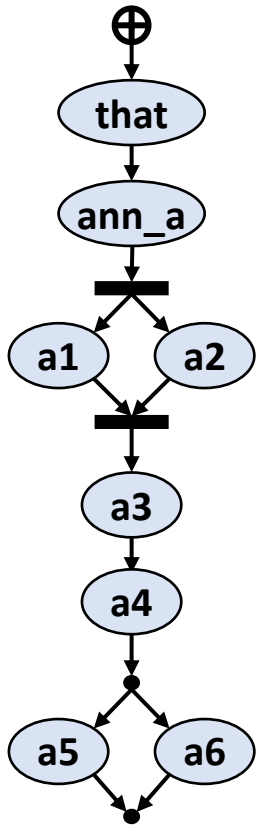
テストインテントの部分的仕様



ツールは制約やルールを元に
取り得るアクションを補完し
テストシナリオを生成する
↓
ユーザは重要なテストインテ
ントのみを指定すれば良い

activity ステートメント

```
action activity_example_a {
  activity {
    that;
    do ann_a;
    parallel {a1, a2};
    sequence {a3, a4};
    select {a5, a6};
    schedule {a7, a8};
    if (i == 0) {a9;}
      else {a10;}
    repeat (2) {a11, a12};
    foreach (arr[j]) {
      a13 with {a13.val == arr[j];};
  }
}
```



do

parallel

sequence

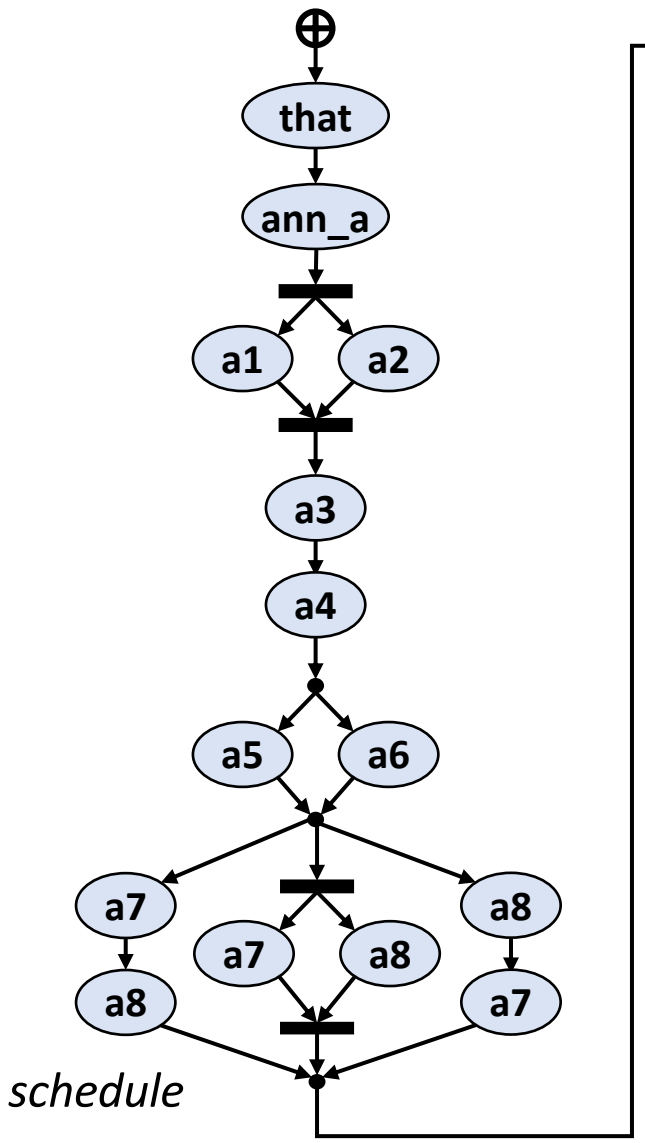
select

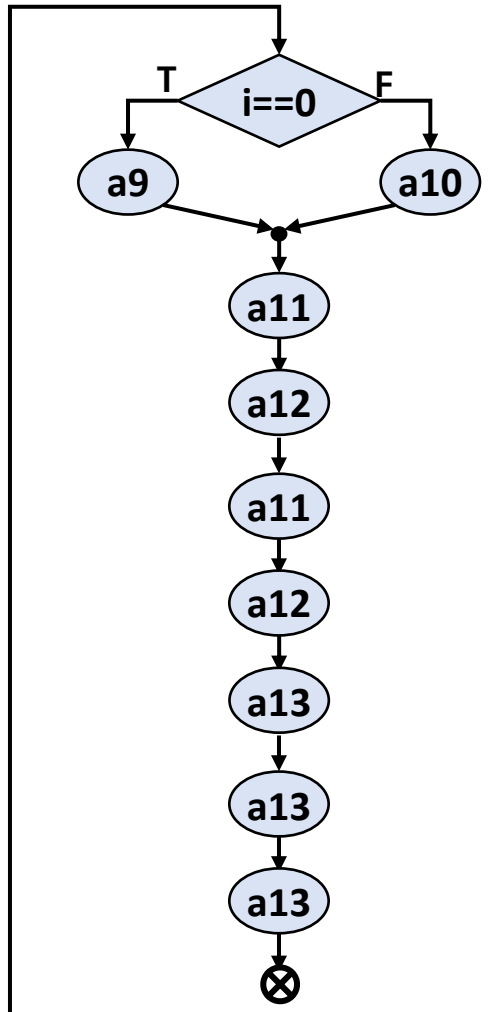
- that は action のインスタンスを呼び出している
- do は action type – ann_a で指定されるアノニマスな action がトラバースされる

activity ステートメント

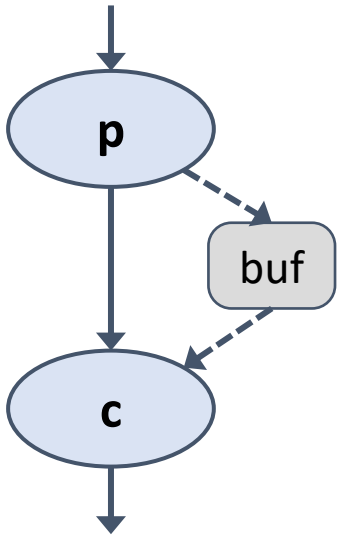
```
action activity_example_a {
  activity {
    that;
    do ann_a;
    parallel {a1, a2};
    sequence {a3, a4};
    select {a5, a6};
    schedule {a7, a8};
    if (i == 0) {a9;}
      else {a10;}
    repeat (2) {a11, a12};
    foreach (arr[j]) {
      a13 with {a13.val == arr[j];};
  }
}
```

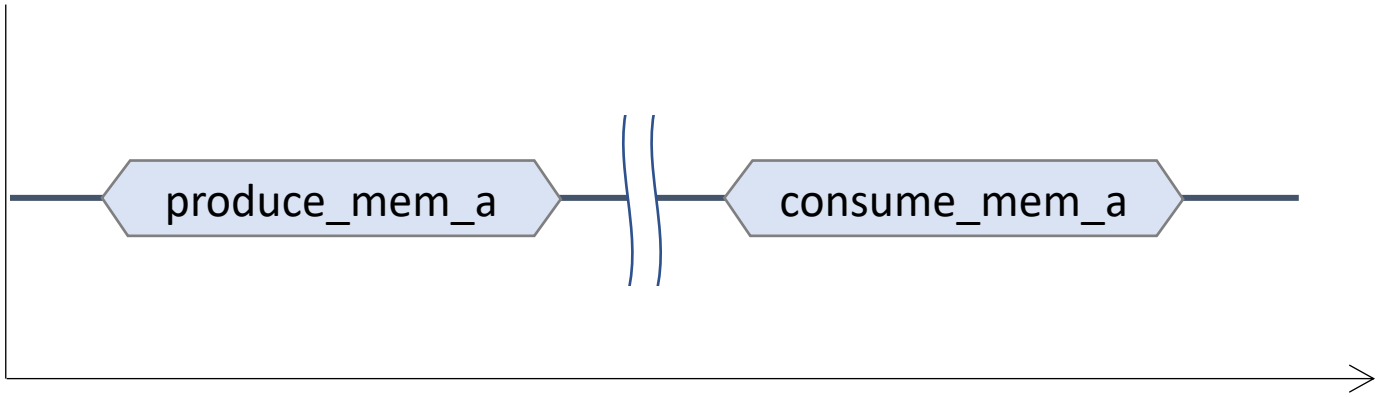
- schedule は resource / data flow などの制約を満たすようにスケジューリングする





data flow オブジェクト - buffer

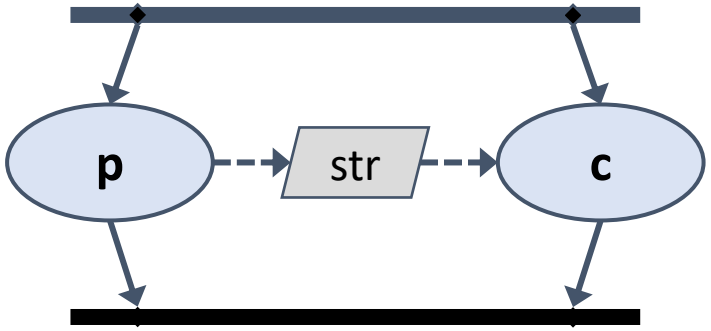


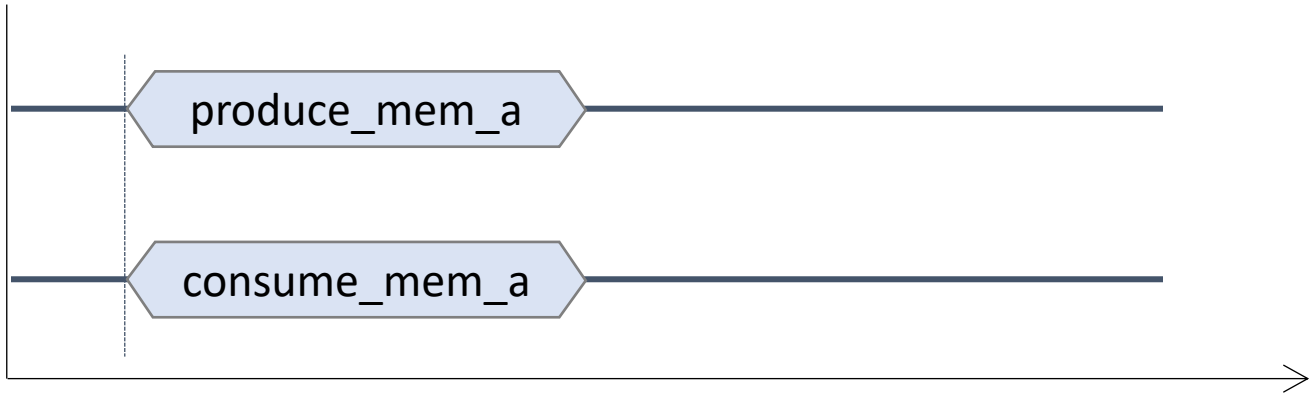


p : producer　　：データ生成側
c : consumer　　：データ消費側

- 記憶要素に一度蓄えられるて引き渡されるためP➔Cへとシリアル処理になる

data flow オブジェクト - stream

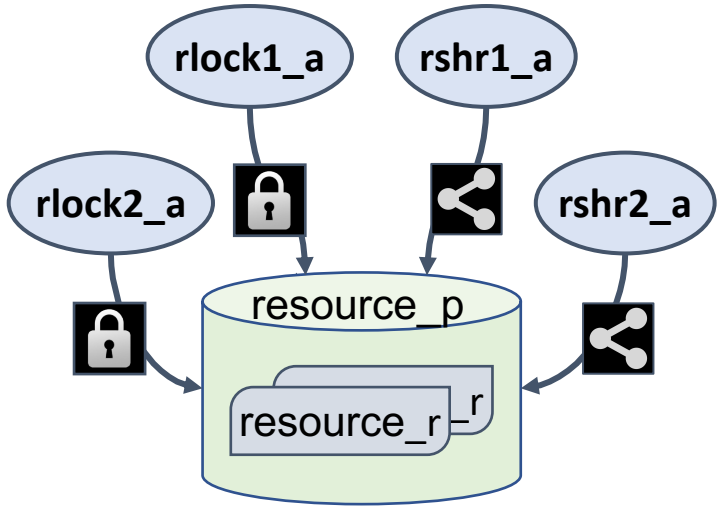




p : producer　　：データ生成側
c : consumer　　：データ消費側

- Pで生成するに連れてCで消費されるためパラレル処理になる

resource オブジェクト



action の中には指定された動作を実現するためのリソースを必要とするものがある

- action / resource 間は明示的に bind できる
- 使用できる resource はセットとして pool にグループ化される
- 複数の同一 resource には instance_id が付与され個別に扱える
- resource は lock または share することができる
- resource 情報は PSSツールのスケジューリングへの制約情報となる

lock

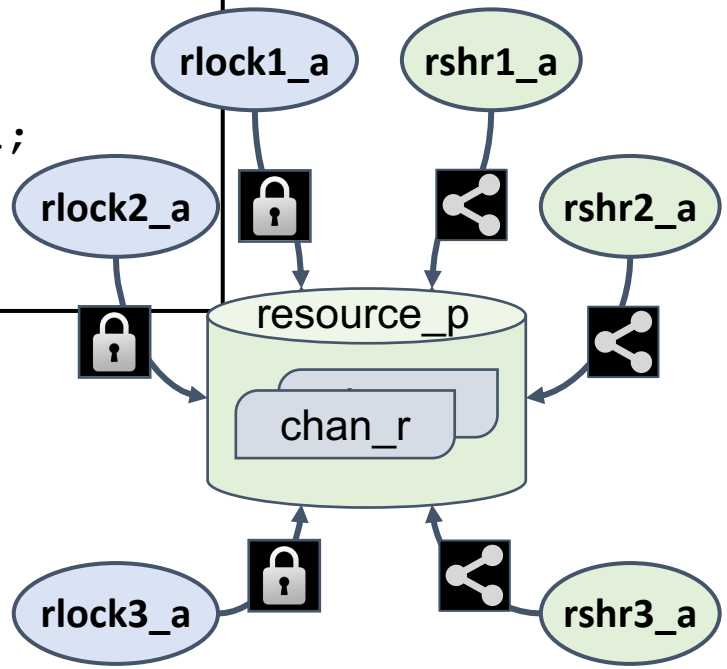
- action が resource への排他的（独占的）なアクセスを要する場合、resource を lock できる
- lock すると action が完了するまで、その resource を使用する他の action の実行を阻止できる

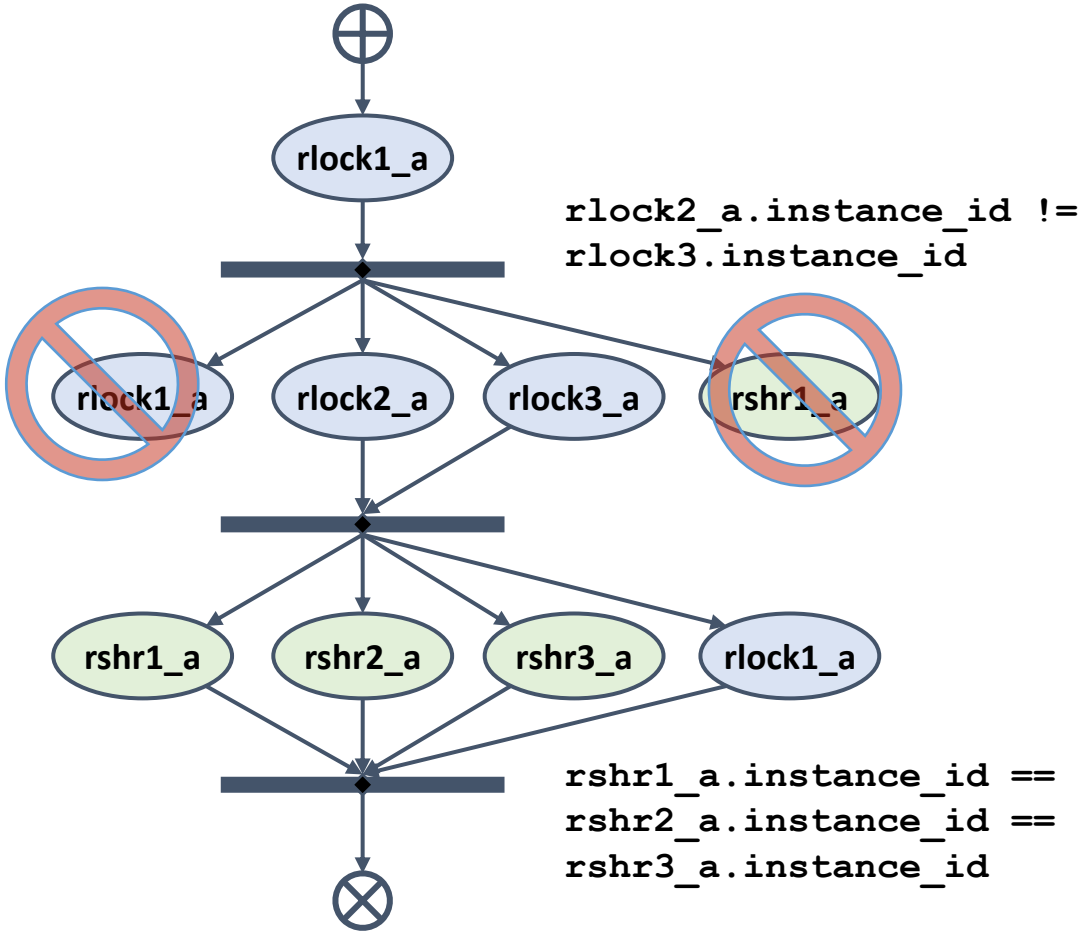
share

- action が resource への非排他的（非独占的）なアクセスを要する場合、resource を share できる
- 他の action に lock された resource インスタンスを共有することはできない

シナリオ生成における resource 制約

```
resource  chan_r {...};
pool [2]  chan_r chan_p;
bind      chan_p {*};

action  rlock_a {
  lock  chan_r chan;
        ...};
action  rshare_a {
  share chan_r chan;
        ...};
}
```

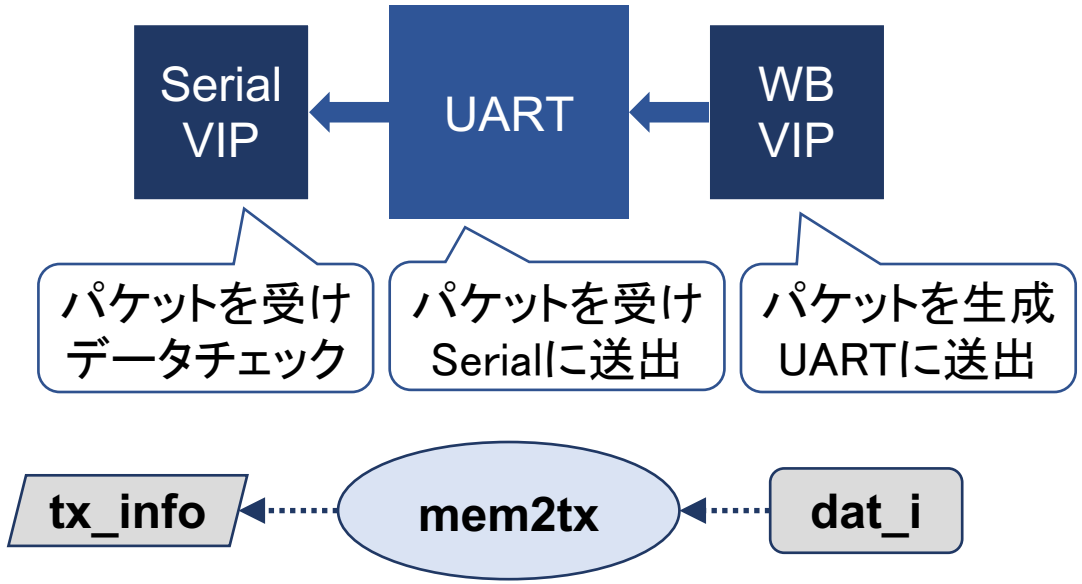




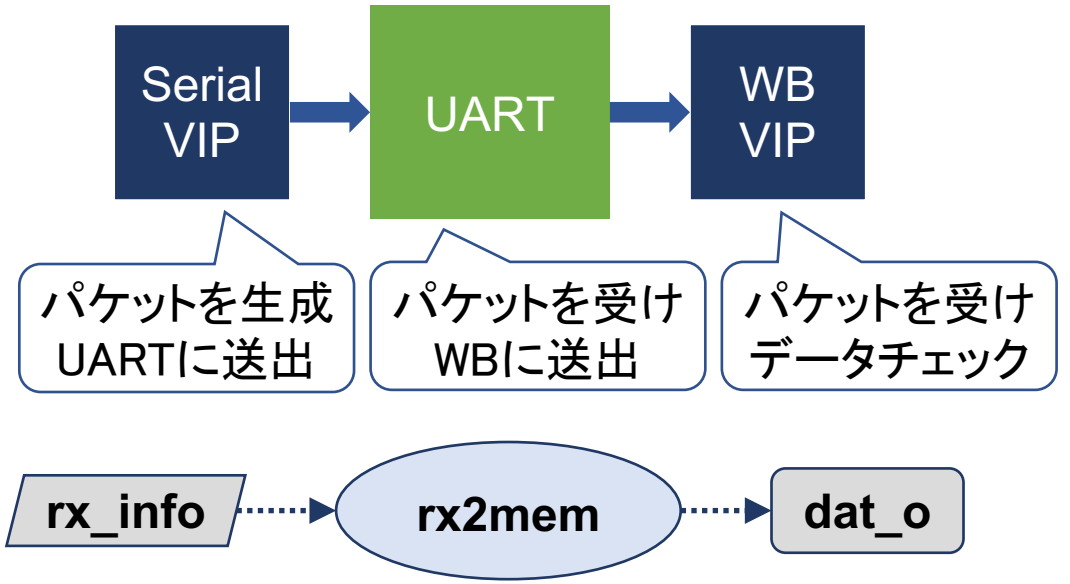
PSSモデリングの一連の流れ

動作を action に落とす

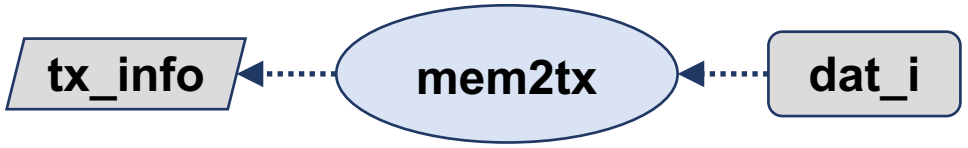
✓ シリアルポートからパケットを送出



✓ シリアルポートからパケットを受信



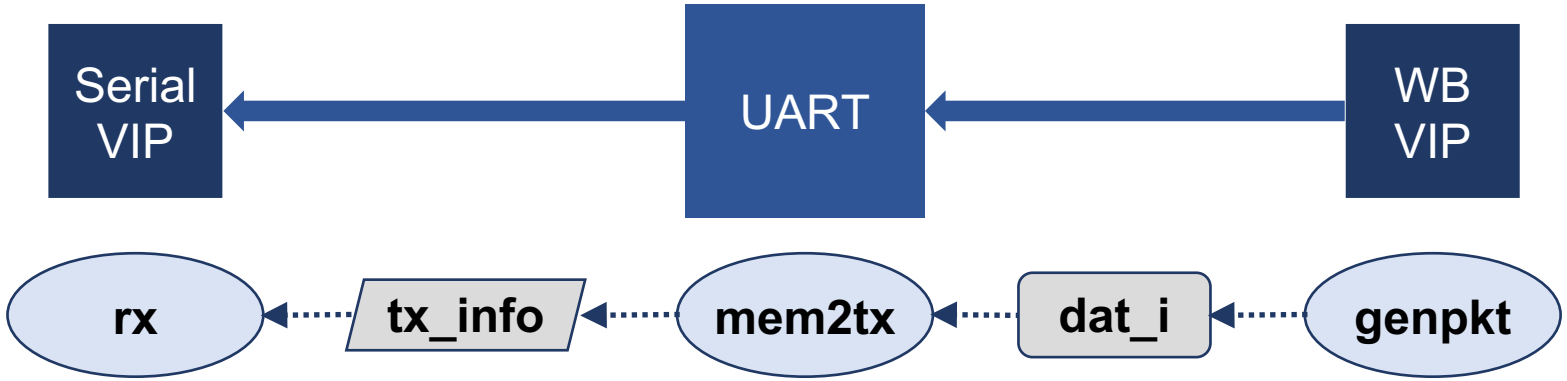
action / dataflow の記述

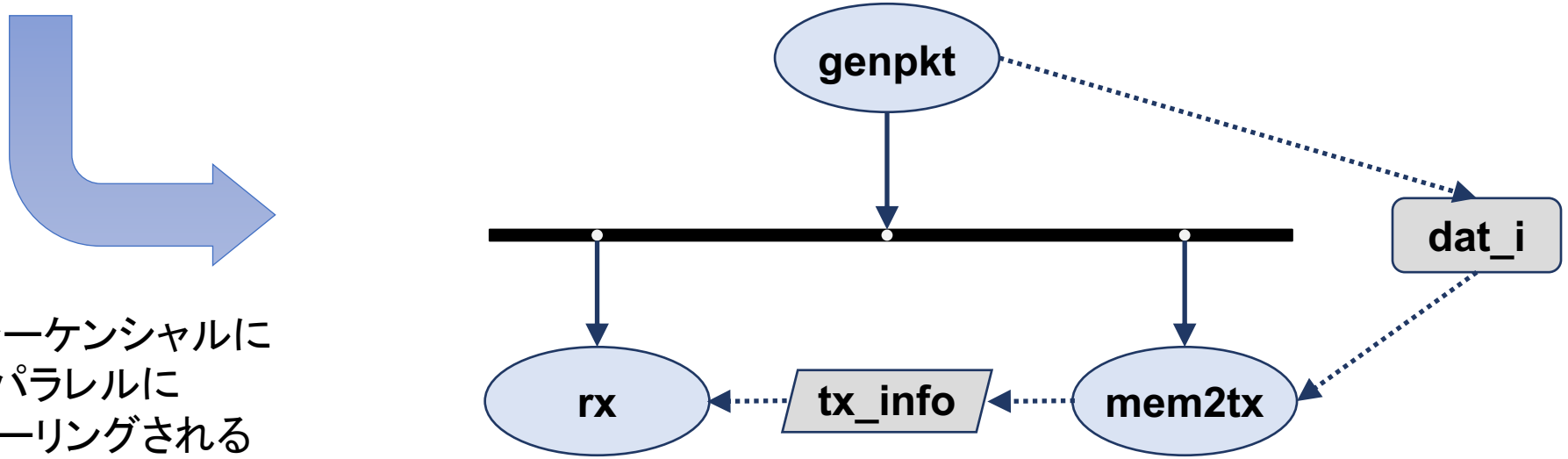



```
action   mem2tx_a : uart_dev_a {

  input  data_ref_b  dat_i;

  output data_ref_s  tx_info;
}

action   uart_dev_a {
  rand   bit[31:0] devid;
}
```

```
struct   data_mem_t {
  rand   bit[31:0]  addr;
  rand   bit[31:0]  sz;
}

         dat_i
buffer   data_ref_b : data_mem_t {
  rand   bit[31:0]  ref;
}

         tx_info
stream   data_ref_s {
  rand   bit[31:0]  sz;
  rand   bit[31:0]  ref;
}
```

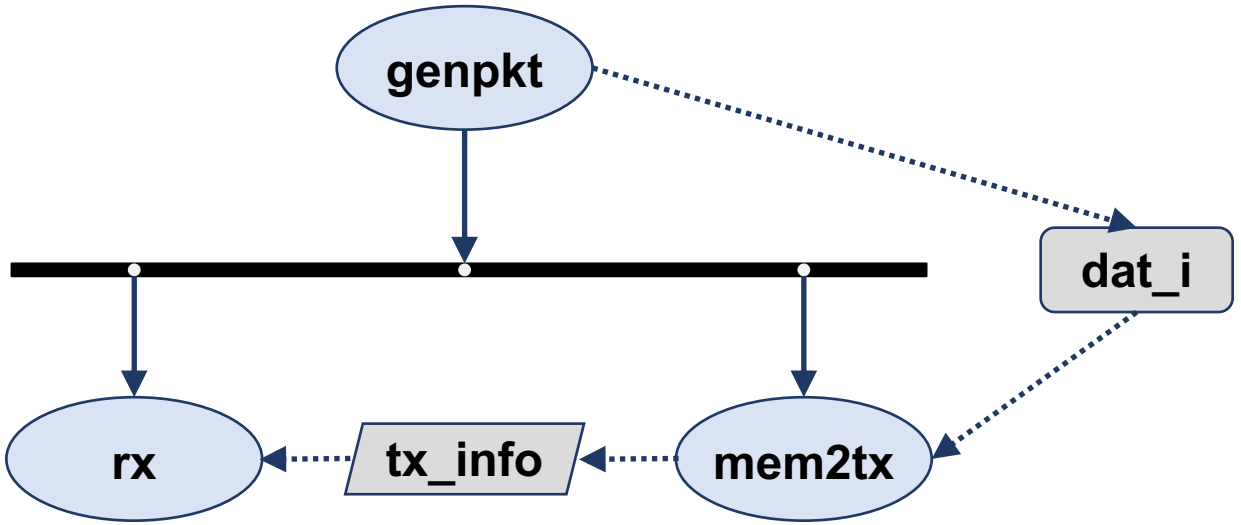
dataflow はスケジューリングに影響する





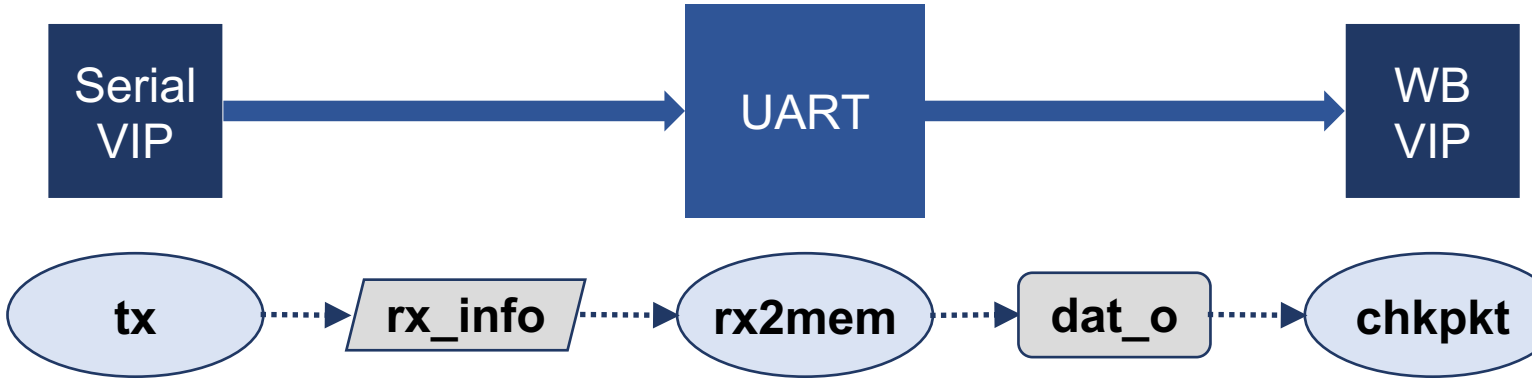
buffer タイプはシーケンシャルに
stream タイプはパラレルに
テストがスケジューリングされる

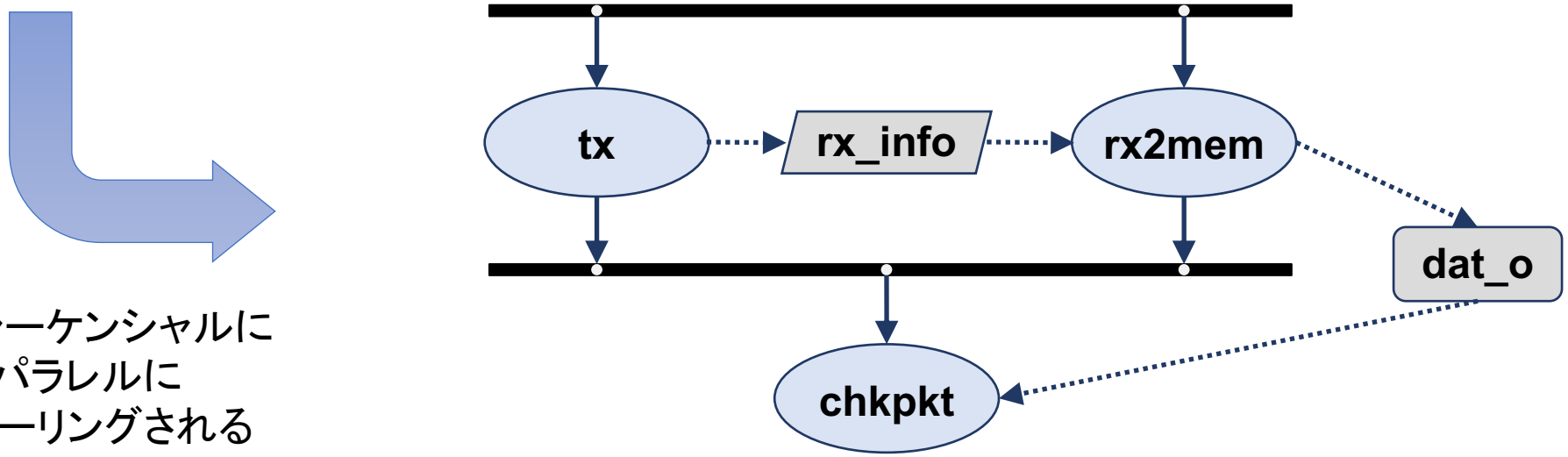
重要なシナリオは明示的に




```
action test_tx_a {
  gendata_a          genpkt;
  mem2tx_a           mem2tx;
  uart_agent_rx_a    rx;

  activity {
    genpkt;
    parallel {
      mem2tx;
      rx;
    }
    bind genpkt.dat_o mem2tx.dat_i;
    bind mem2tx.tx_info rx.rx_info;
  }
}
```

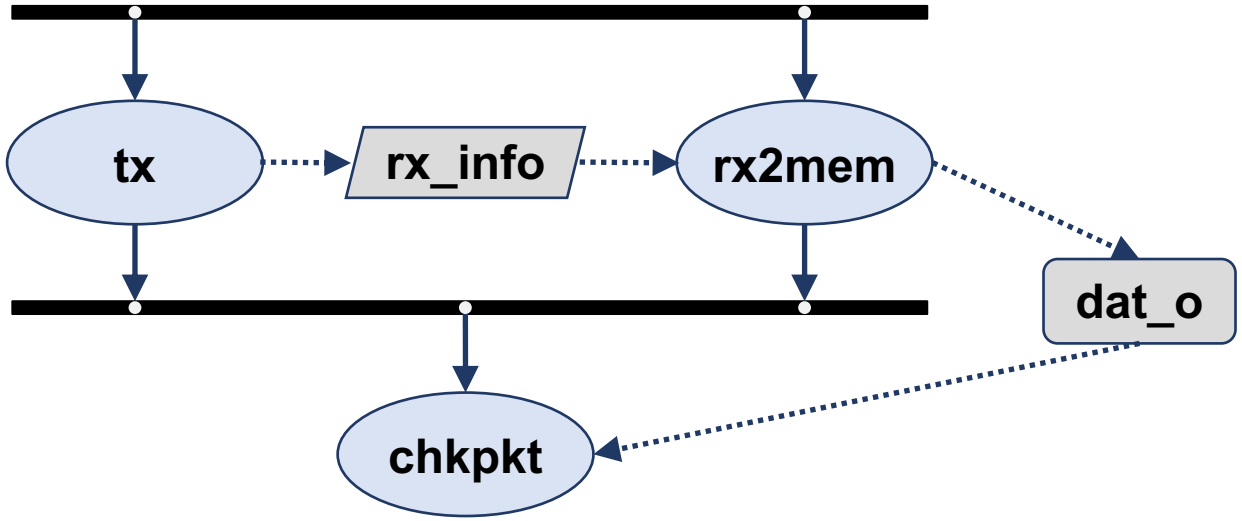
dataflow はスケジューリングに影響する





buffer タイプはシーケンシャルに
stream タイプはパラレルに
テストがスケジューリングされる

重要なシナリオは明示的に

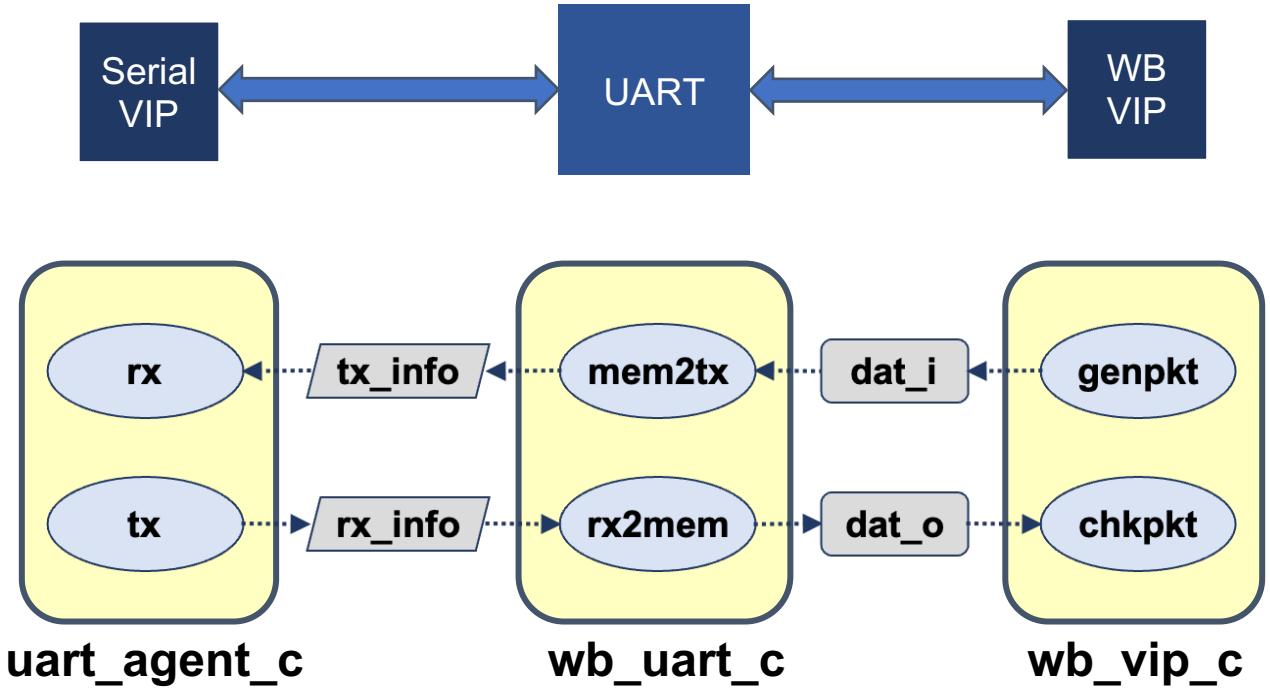



```
action test_rx_a {
  chkdata_a          chkpkt;
  rx2mem_a           rx2mem;
  uart_agent_tx_a    tx;

  activity {
    parallel {
      tx;
      rx2mem;
    }
    chkpkt;
    bind rx2mem.rx_info tx.tx_info;
    bind chkpkt.dat_i rx2mem.dat_o;
  }
}
```

component へのグルーピング

```
component uart_agent_c {
  action rx_a : uart_dev_a {
    input data_ref_s  rx_info;
  }
  action tx_a : uart_dev_a {
    output data_ref_s tx_info;
  }
}
```

```
component uart_top_c {
  action test_tx_a {
    wb_vip_c::gendata_a genpkt;
    wb_uart_c::mem2tx_a mem2tx;
    uart_agent_c::rx_a rx;
  action test_rx_a {
    uart_agent_c::tx_a tx;
    wb_uart_c::rx2mem_a rx2mem;
    wb_vip_c::chkdata_a chkpkt;
...
}
```



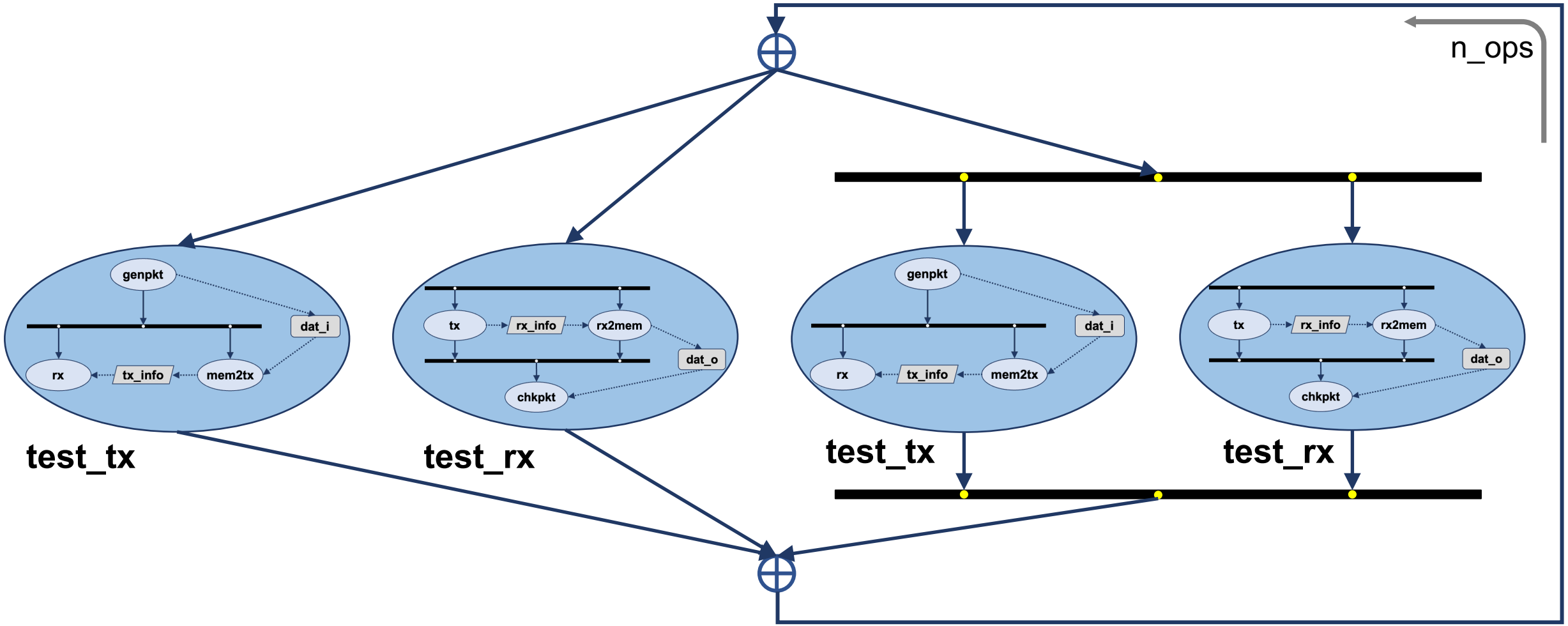
- component にまとめることで再利用性が高まる

テストのトップコンポーネント

```
component uart_top_c {
  action test_tx_a {...}
  action test_rx_a {...}
  action transfer_a {
    test_tx_a test_tx;
    test_rx_a test_rx;
    action bit[7:0] in [3..5] n_ops;

    activity {
      n_ops;
      repeat(n_ops) {
        select {
          test_tx;
          test_rx;
          parallel { test_tx; test_rx;}
        }
      }
    }
  }
}
```

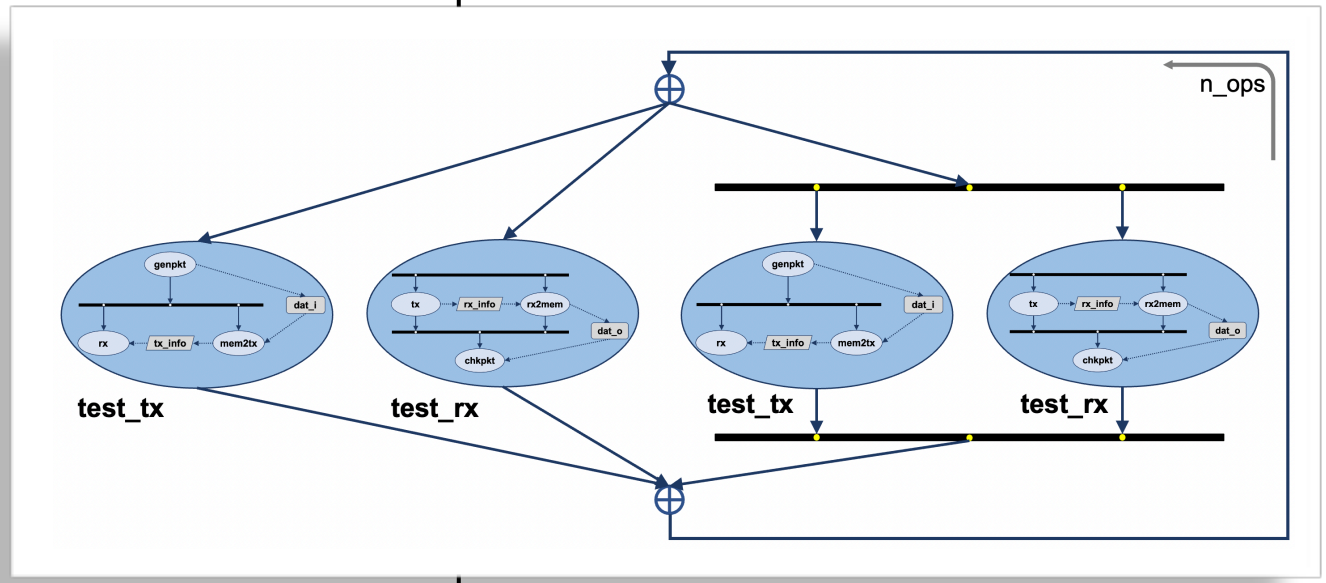
- 宣言時に範囲を指定
- ランダムな action のデータフィールド
- n_ops をランダム化
- 実行するテストをランダムに選択

生成されるテストシナリオ



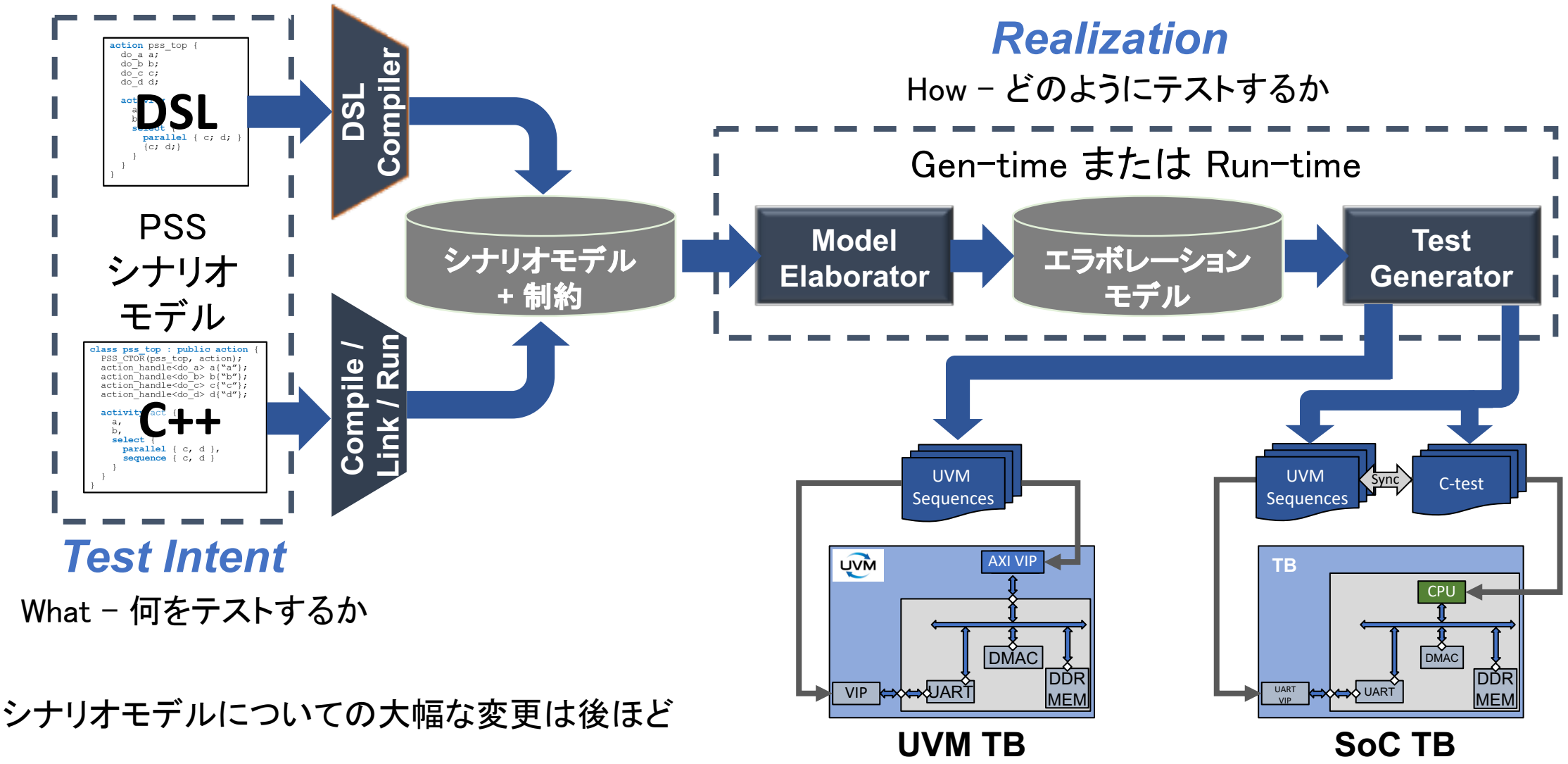
PSSコードへのカバレッジの追加

```
component uart_top_c {
  ...
  enum snr_e {snr_tx,snr_rx,snr_txrx }
  action transfer_a {
    action snr_e snr;
    covergroup {
      snr_cp : coverpoint snr;
    } mode_cg;
    activity {
      n_ops;
      repeat(n_ops) {
        snr;
        select {
          (snr==snr_tx):test_tx;
          (snr==snr_rx):test_rx;
          (snr==snr_txrx):parallel {test_tx; test_rx;}
        }
      }
    }
  }
}
```



PSS最新バージョンのアップデート情報

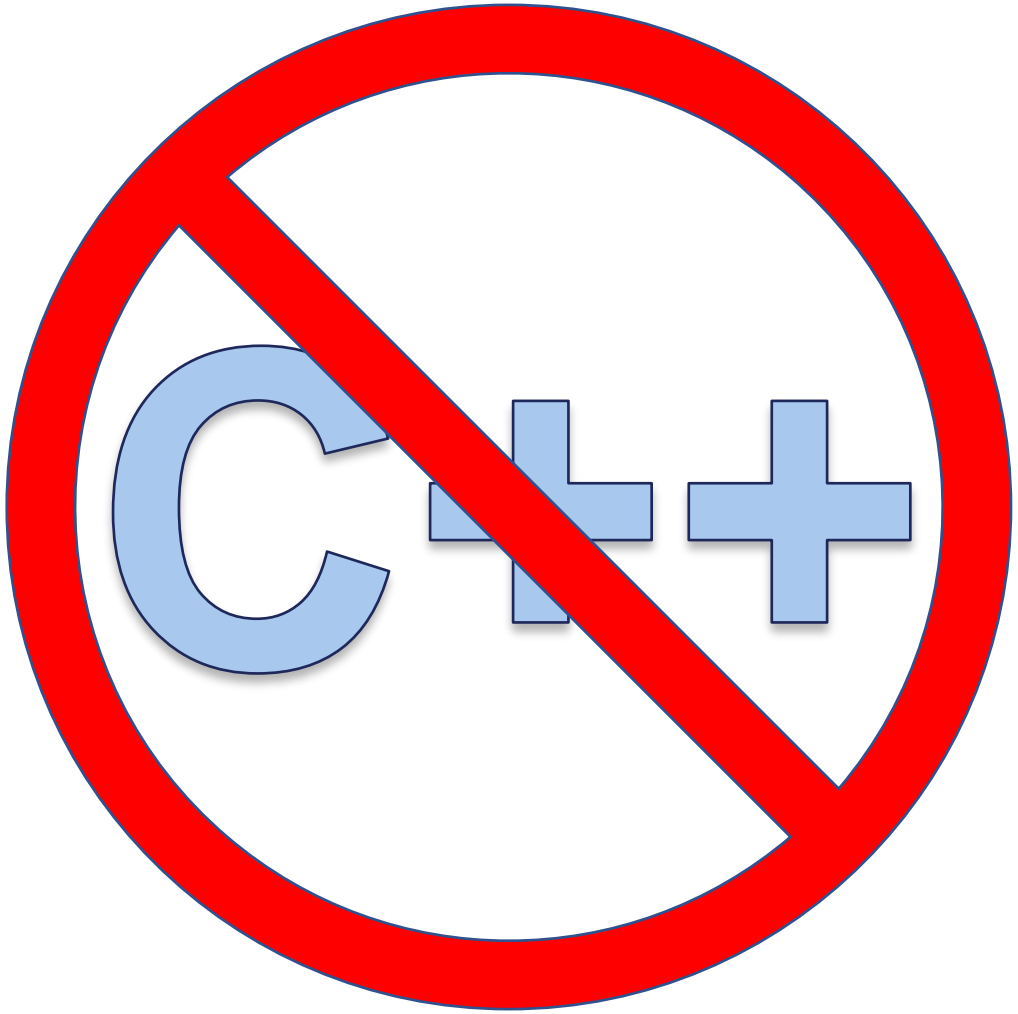
PSS標準が想定しているツールフロー



※シナリオモデルについての大幅な変更は後ほど

最大の変更点

DSL

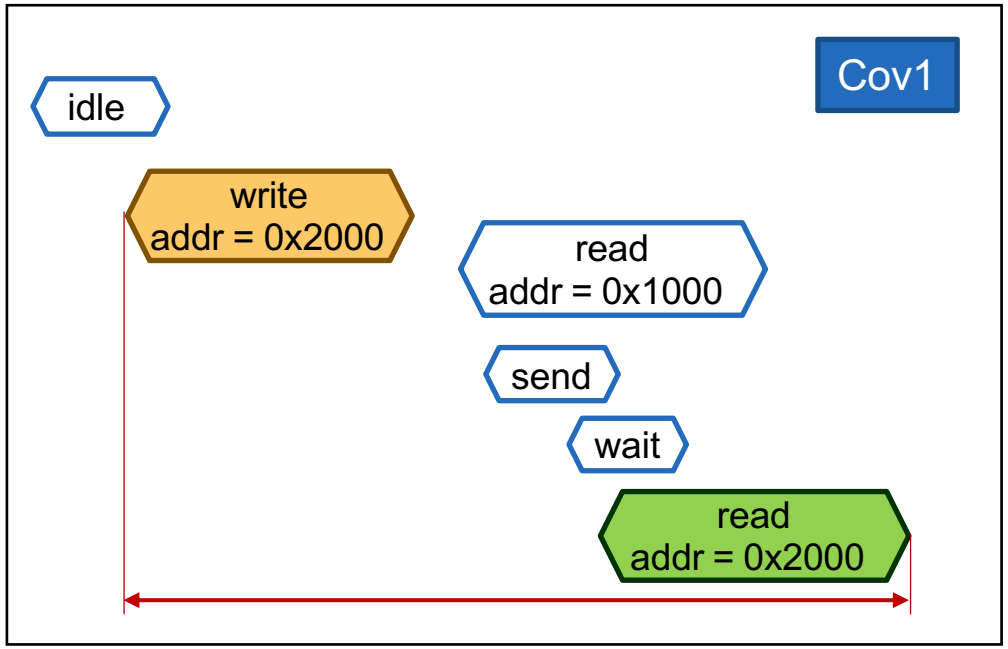


最大の変更点

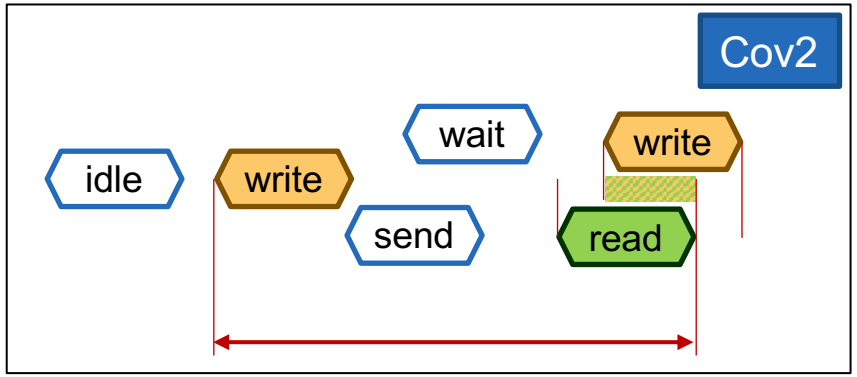
PSS

ビヘイビアル・カバレッジの追加（策定中）

- 与えられたactionのストリーム実行に対して、そのストリームに指定したテンポラルシナリオ(query)が発生したかどうかを知る
- coverステートメントでは、興味ある注目シナリオを指定
- monitorはカバーすべきビヘイビアをカプセル化する
  - coverステートメントにより暗示的、もしくは明示的

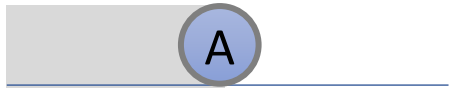



```
action write { rand bit [32] addr; }
action read { rand bit [32] addr; }

monitor read_after_write { write w; read r; activity { w; r; }}

cover Cov1 { read_after_write w_r; activity { w_r with r.addr == w.addr; }}
cover Cov2 { write w1, w2; read r; activity { w1; r overlaps (w2) ;}
```



ビヘイビアル・カバレッジのトラッキング

- action traversal (ex. do A with x ==5)
  - 与えられた制約に合う最も近いaction実行にマッチ



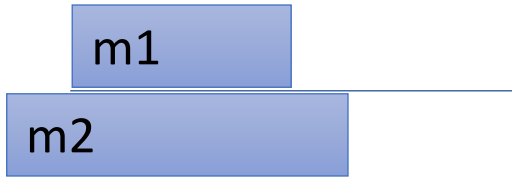
- Concatination： concat {m1; m2}
  - m1後の最初のm2にマッチ
    - 現在のポイントからm1にマッチし、そのマッチポイントから任意のポイントでm2にマッチ

- Eventually: eventually m
  - 将来のどこかの時点で発生
    - 現在のポイントから任意ポイントで m にマッチ

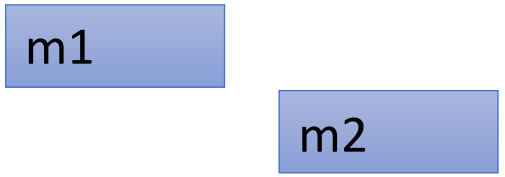
ビヘイビアル・カバレッジのトラッキング

- Sequence: sequence {m1; m2} または単に {m1; m2}
  - m1発生後にm2にマッチ、連続である必要はない
    - 現在のポイントからm1がマッチし、そのマッチポイントから任意ポイントで m2にマッチ
    - concat {m1; eventually m2} と等価

- Overlap: {m1 overlaps m2}
  - m1実行中の任意ポイントでm2が実行すればマッチ



- Select: select {m1; m2}
  - m1もしくはm2にマッチ



ソルブ時 / ランタイムのメッセージング

- Core Libraryへの新規追加
  - ソルブ時
    - フォーマット
    - プリント
  - ランタイム
    - メッセージ
  - ソルブ／ランタイム
    - error / fatal

```
solve function void print_foo(my_struct s) {
  print("The context of the struct is:¥n");
  print("value = %d¥nname = '%s'¥n", s.value, s.name);
}
solve function string get_foo_context_string(my_struct s) {
  return format("value = %d¥nname = '%s'¥n", s.value, s.name);
}

exec body {
  y = my_func();
  message(FULL, "The values of the variables x and y are: ");
  message(LOW, "%d, %d", x, y);
}

package io_pkg { // may change to std_pkg or...
  function void error(string format, type... args);
  function void fatal( int status, string format, type... args);
}
```

呼出環境への戻り値

ソルブ時 / ランタイムのメッセージング

- ソルブ時のファイルI/O
  - ファイルハンドル経由

```
package io_pkg {
  typedef chandle file_handle_t;
  static const file_handle_t nullhandle = /* implementation-specific */;
  enum file_option_e {TRUNCATE, APPEND, READ};
  function file_handle_t file_open(string filename, file_option_e opt = TRUNCATE);
  function void file_close(file_handle_t file_handle);
  function bool file_exists(string filename);
  function void file_write(file_handle_t file_handle, string format, type... args);
  function string file_read(file_handle_t file_handle, int size = -1);
```

  - 単純な関数呼出し

```
function void file_write_lines(string filename, list<string> lines,
                               file_option_e opt = TRUNCATE);
function list<string> file_read_lines(string filename);}
```

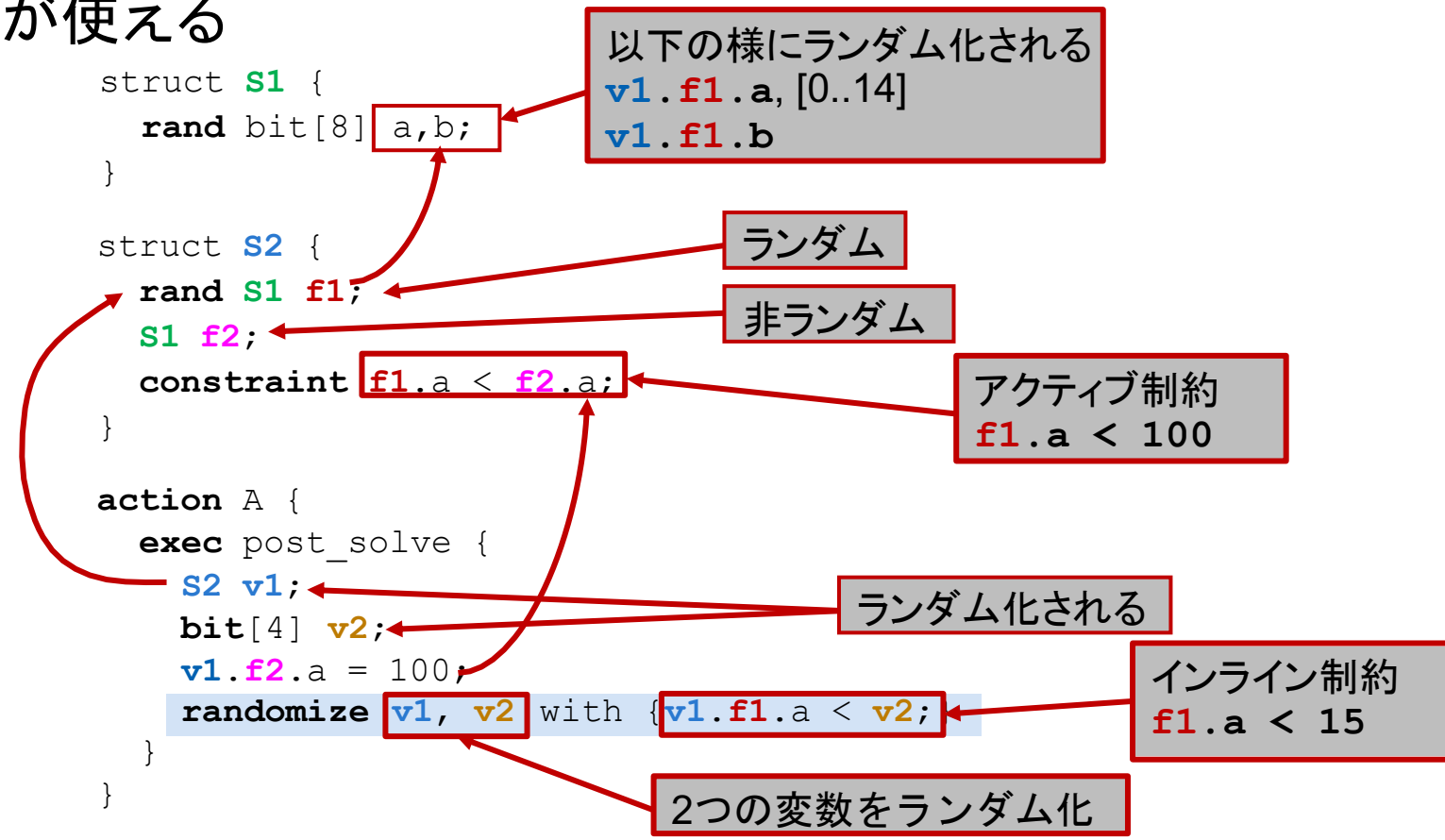
list のランダム化

- list は rand として宣言可能
  - 他のrandフィールドと同様にコンテイナーをランダム化する
- size はステート変数
  - sizeに対して直接制約を与えることはできない

```
struct S {
  rand list<bit[8]> lst;
  exec pre_solve { // Initialize the list
    repeat (100) {
      lst.push_back(0);
    }
  }
  constraint {lst.size() in [4..100]; // Error: illegal constraint on size
    foreach (lst[i]) {
      lst[i] == i+lst.size(); // OK: size is a state variable in foreach
    }
  }
}
```

ランダム化の手続き的なステートメント

- solveのexecブロック内で使用可能
- ターゲットのexecでサブセットが使える
  - ただしstruct ランダム化以外
- 以下もサポート
  - urandom
  - urandom_range



ランダム化の制約 - dist

- SystemVerilogに類似
  - := は個別の重み付けの割り当て
  - :/ はリストに渡り重み付けを配分
- 他の制約も受ける
  - -dist はリーガルな範囲内の値に対する偏り

```
struct s {
  rand bit[32]  x;
  bit           y;
  constraint dist x in [100..102 :=1, 200 :=2, 300 :=5];
  constraint dist x in [100..102 :/1, 200 :=2, 300 :=5];

  constraint dist x in [100..102 :=1, 200 :=2, 300 :=5];
  constraint (y==1) -> x > 300;
```

100:1, 101:1, 102:1, 200:2, 300:5

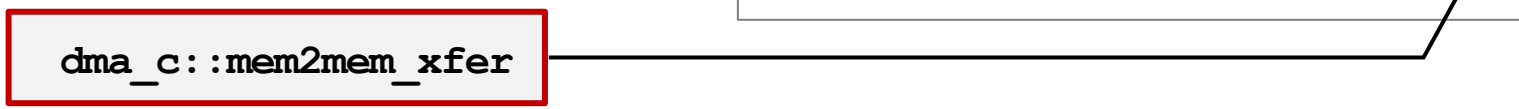
100:1/3,
101:1/3, 200:2, 300:5
102:1/3

制約により dist は無効

action のハンドルとしてのラベル

- actionのハンドルとしてラベル付け
  - アノニマスのactionトラバースに対するハンドルを作成
  - ラベルを用いて参照可能

```
action mem2mem_chain {
  activity {
    do mem_c::load_buff;
    repeat (10) {
      select {
                xfer:do dma_c::mem2mem_xfer;
                cpy: do cpu_c::memcpy;
      }
    }
  }
}
action my_test {
  activity {
    do mem2mem_chain with { xfer.size > 10; };
  }
}
```

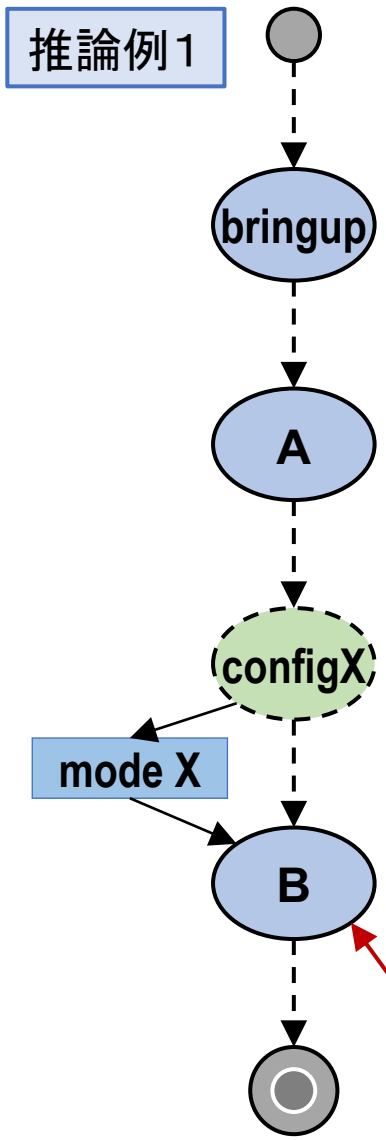


推論の問題

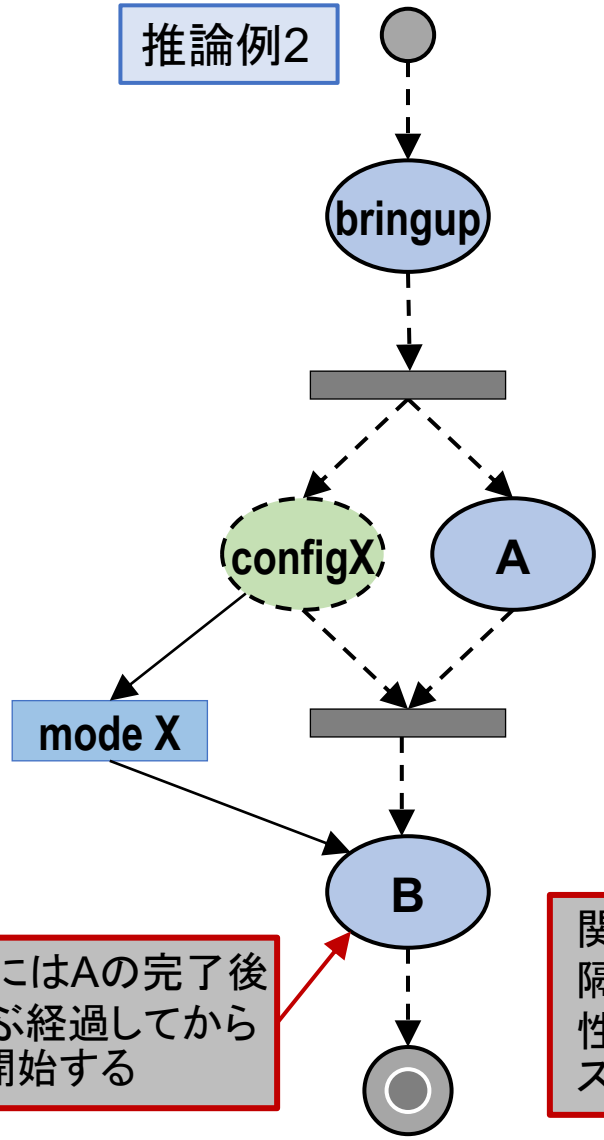
```
state config_s {
  rand mode_e mode;
}
action configX {
  output config_s out_cfg;
  constraint out_cfg == X;
}
action B {
  input config_s cfg;
  constraint cfg.mode == X;
}
```

```
state my_stress_seq {
  activity {
    do bringup;
    do A;
    do B;
  }
}
```

Aの完了直後にBを開始するシナリオが重要な場合



実際にはAの完了後だいぶ経過してからBが開始する



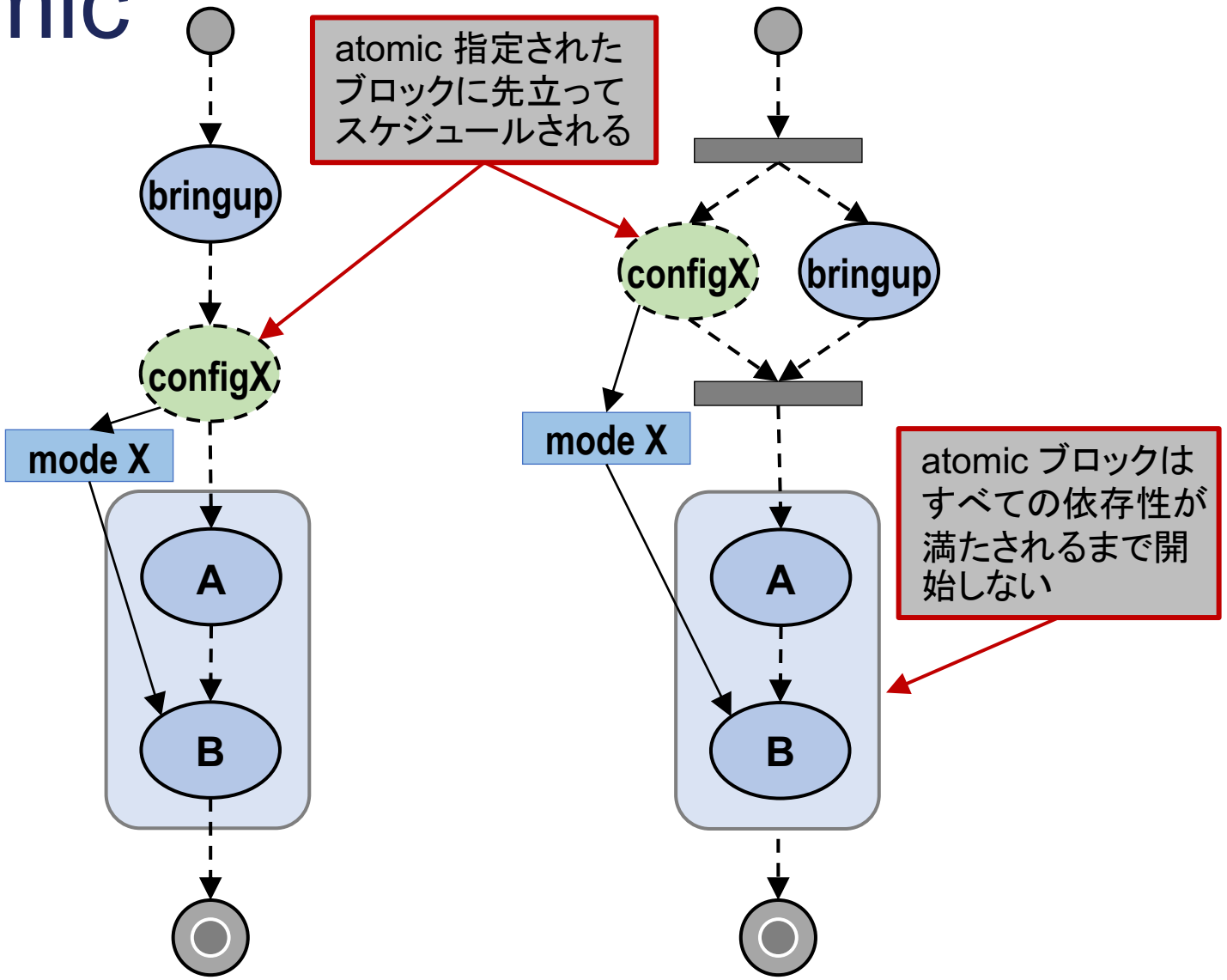
関連するアクションの間隔があいたり、その重要性が指定できない場合、ストレス試験にならない

推論に向けた”atomic”

atomic 指定されたアクティビティブロックでは各アクションが、シーケンスの先行アクション以外いかなるアクションも待たずに開始することが保証される

```
state my_stress_seq {
  activity {
    do bringup;
    atomic {
      do A;
      do B;
    }
  }
}
```

Bのいかなる依存関係でもブロック全体の依存関係となる



簡素化された read-modify-write

1つのオペレーションで操作可能

```
struct CR : packed_s<> {
  bit      en;
  bit[11]  pad;
  bit[4]   mode;
  bit[16]  coeff;
}

component dut_c {
  dut_regs_c    regs;
  action cfg_a {
    rand bit[4]   mode;
    rand bit[16]  coeff;
    exec body {
      comp.regs.cr.write_masked({.mode=~0, .coeff=~0}, {.mode=mode, .coeff=coeff});
      comp.regs.cr.write_val_masked(0xFFFFF000, (coeff << 16) | (mode <<12));
      comp.regs.cr.write_fields({"mode", "coeff"}, {mode, coeff});
    }
  }
}
```

```
pure component reg_c < type R,
  reg_access ACC = READWRITE,
  int SZ = (8*sizeof_s<R>::nbytes)> {
    function R read();
    function void write(R r);
    function bit[SZ] read_val();
    function void write_val(bit[SZ] r);

    function void write_masked(R mask, R val);
    function void write_val_masked(bit[SZ] mask, bit[SZ] val);
    function void write_field(string name, bit[SZ] val);
    function void write_fields(list<string> names,list<bit[SZ]> vals);
}
```

追加された新機能

- コンポーネント内のスタティック・ファンクション
  - コンポーネントのタイプに関連付けられたファンクション宣言（インスタンス単位ではない）
  - スコープ演算子 “::” によって呼び出すことが可能
- Tag/Addr_value
- Enumベースタイプ
- 浮動小数点の計算とストレージタイプ

Questions