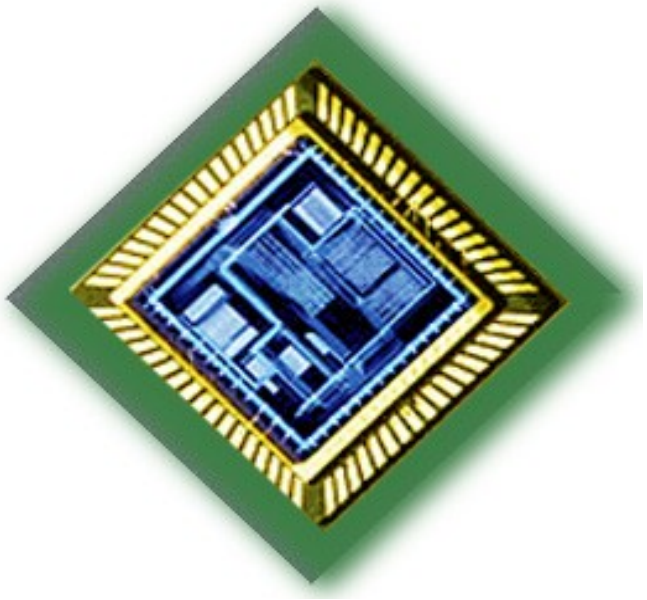


Verificação funcional

Curso do Programa Brazil-IP

Elmar Melcher
UFCG
elmar@dsc.ufcg.edu.br
http://lad.dsc.ufcg.edu.br

Roteiro

1) Introdução
2) Motivação
3) Fluxo de projeto
4) Verificação
5) Tipos de verificação
6) Verificação funcional
7) Abordagens de verificação
8) Plano de verificação

Roteiro

9) Metodologia BVM
10) Testbench
11) Elementos básicos
12) Regras de projeto
13) Implementação
14) Tipos de estímulos
15) Cobertura
16) Biblioteca OVM

Roteiro

17) Exemplos

- Decodificador MPEG4
- Lacre Eletrônico
- SPVR
- Dummy DPCM

Introdução

Ninguém usa um IP core no qual não confia.

IP core: “Lógica ou os dados necessários para construir um projeto de hardware.

Idealmente ele é “reusável” e pode ser adaptado a vários tipos de dispositivos de hardware. Pode-se entender o *IP core* como a implementação de um dado projeto de hardware em uma linguagem específica para esse objetivo.”

Motivação

Caso famoso: NASA

Exemplo de problema causado por má verificação na NASA:

Mars Climate Orbiter de 1999

- Erro de verificação de sistema fez com que o Orbiter voasse muito próximo à atmosfera marciana e queimasse.
- Problema era uma mistura de unidades métricas e unidades inglesas que causou um erro no cálculo da trajetória.

Caso famoso: INTEL

- Problema descoberto no Pentium em 1994 por um professor de matemática durante uma pesquisa matemática.
- A FPU produzia resultados de divisão errôneos no oitavo digito significativo para certos argumentos.
- Embora no máximo 1 usuário em 1 milhão jamais foi afetado, a confiança na INTEL caiu.
- Abriu o caminho para crescimento maior da AMD.
- Prejuízo de 500 MUS$ (?)

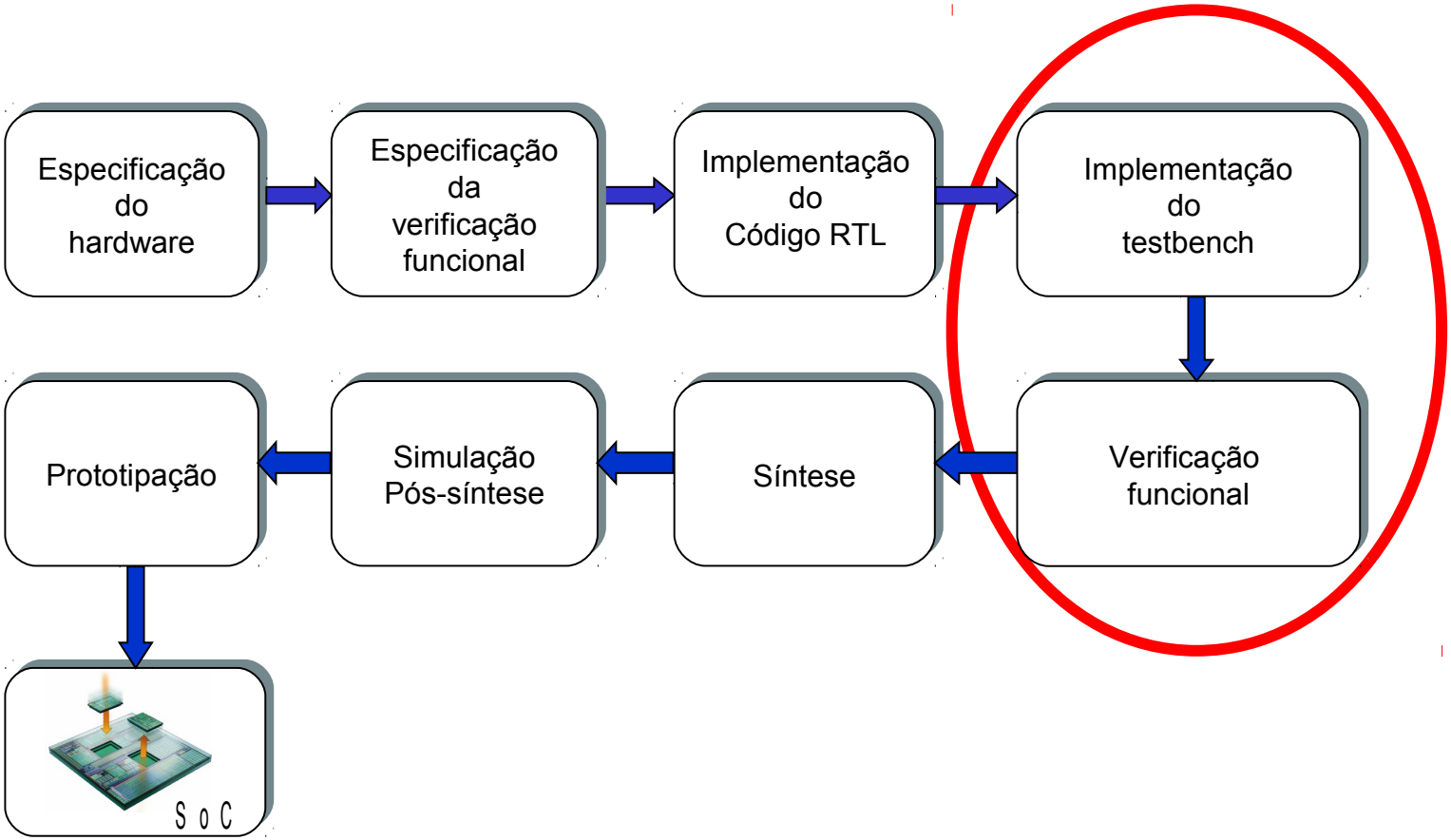
Verificação de IP cores

- **Como alcançar confiança em um projeto?**

Processo de verificação bem executado e documentado.

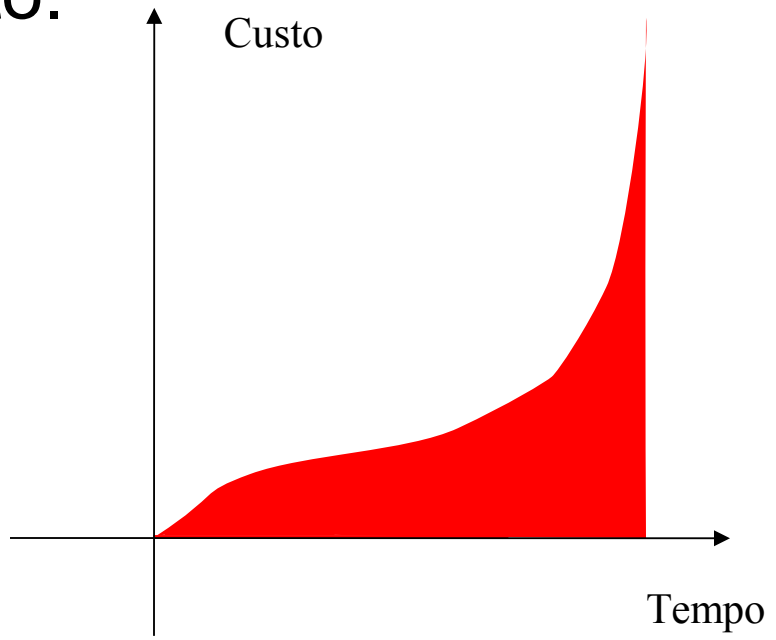
- IP cores precisam ser verificados mais amplamente;
- Todas propriedades e utilizações possíveis devem ser verificadas;
- Não somente um ambiente específico.

Fluxo tradicional de um projeto de hardware



Custo versus Tempo

- Entregar produtos e ganhar dinheiro.
- **Custo** e tempo de corrigir um defeito cresce quando descoberto mais tarde no ciclo de vida do produto.
  - na especificação
  - na simulação
  - na prototipagem
  - na fabricação em volume
  - no uso pelo cliente



Verificação

Tipos de verificação

Verificação:

- Dinâmica ou Funcional.
- Estática ou Formal.
- Híbrida.

Verificação estática

Analisadores de código HDL, *Lint* ou *linting tools*

Tipos de problemas detectáveis:

- **case** incompleto
- atribuições em **if...else** inconsistentes
- falta de sinais na lista de sensibilidade
- reset síncrono / assíncrono
- falta de sinais a serem resetados
- clocks ativos todos na borda de subida ou descida
- resets ativos todos em nível alto ou nível baixo

Verificação estática

Exemplos de ferramentas de *Lint*:

- Leda da Synopsys
- HAL da Cadence
- DesignAnalyst da Mentor

Verificação funcional

“Verificação funcional é um processo usado para demonstrar que o objetivo do projeto é preservado em sua implementação” [bergeron2003]

Verificação funcional x Teste

Verificação Funcional: confrontar um modelo a ser verificado a outro modelo padrão, comparando a funcionalidade por simulação.

X

Teste: verificar se um CI está sem erro de fabricação.

Verificação funcional

- Mais da metade do esforço de projeto está na verificação.
- Um testbench muitas vezes contém duas vezes mais linhas que a própria descrição do projeto.
- A equipe de engenheiros de verificação é maior do que a equipe de projetistas.

Verificação é difícil.

Verificação funcional

- Engenheiros da Motorola no SBCCI 2002:
"Estamos procurando trabalhos sobre verificação."

- Novas metodologias em 2006:
VMM, AVM, OVM, …

- Procura por "bons" engenheiros de verificação em todas as empresas.

Custo da verificação

Um mal necessário

Sempre leva tempo demais e custa caro demais

Mas indispensável

Porque afeta diretamente os três requisitos:

- cronograma
- custo
- qualidade

Isso funciona mesmo?

A verificação funcional deve responder a esta pergunta.

- “isso” é uma descrição RTL de um projeto.
- “funciona” se refere a simulação.

O funcionário de uma empresa deve poder **dormir tranquilo** se a verificação responde “sim”.

Como saber fazer?

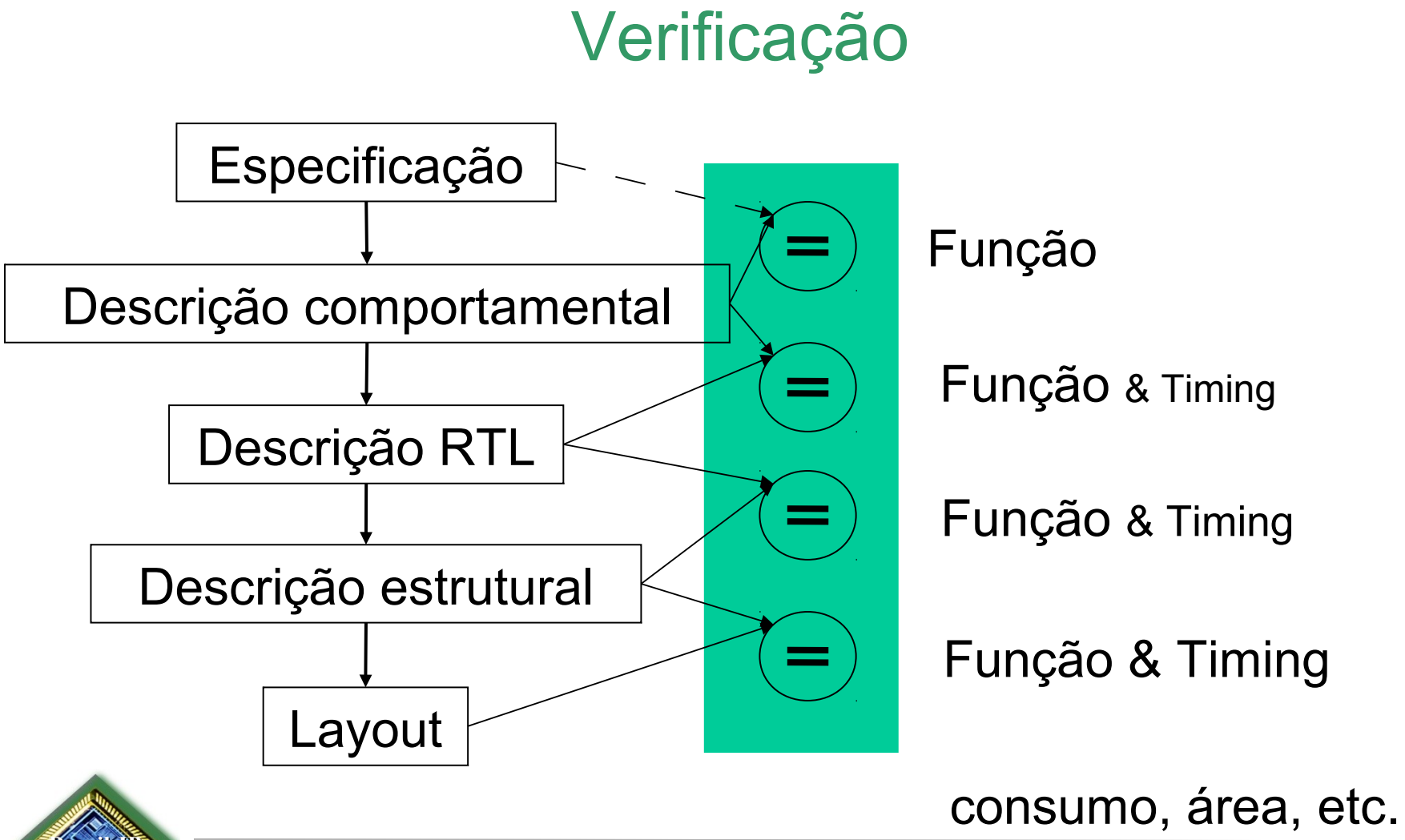
- Muitos livros falam sobre implementação.
- Nem tantos falam sobre verificação:

- *Writing Testbenches: Functional Verification of HDL Models* by Janick Bergeron, 3rd edition, KluwerAcademic Publishers, 2006.

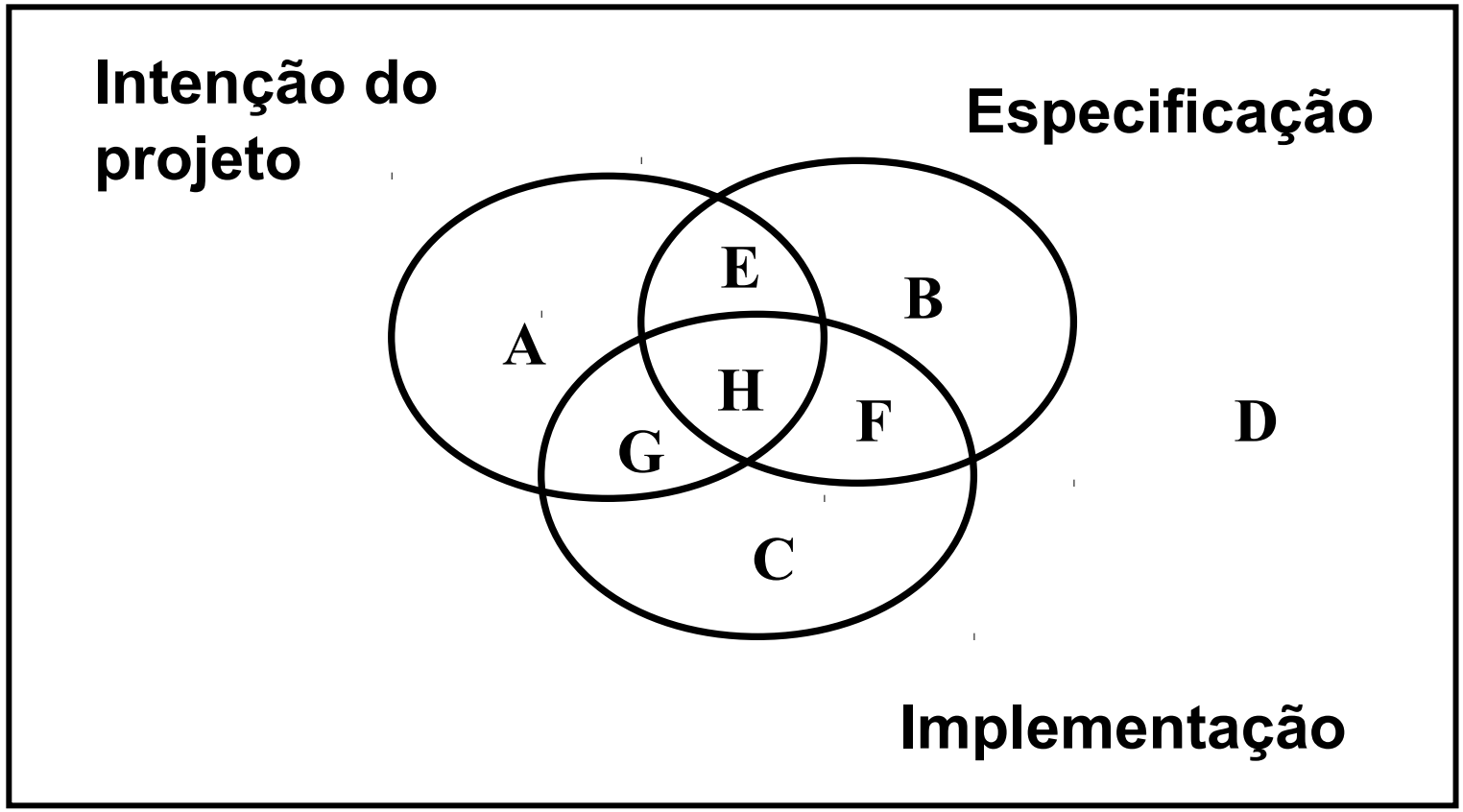
- Piziali*, Functional verification Coverage Measurement and Analysis,* Kluwer 2004*.*

- *Writing Testbenches Using SystemVerilog* by Janick Bergeron, 1st edition, Springer, 2006.

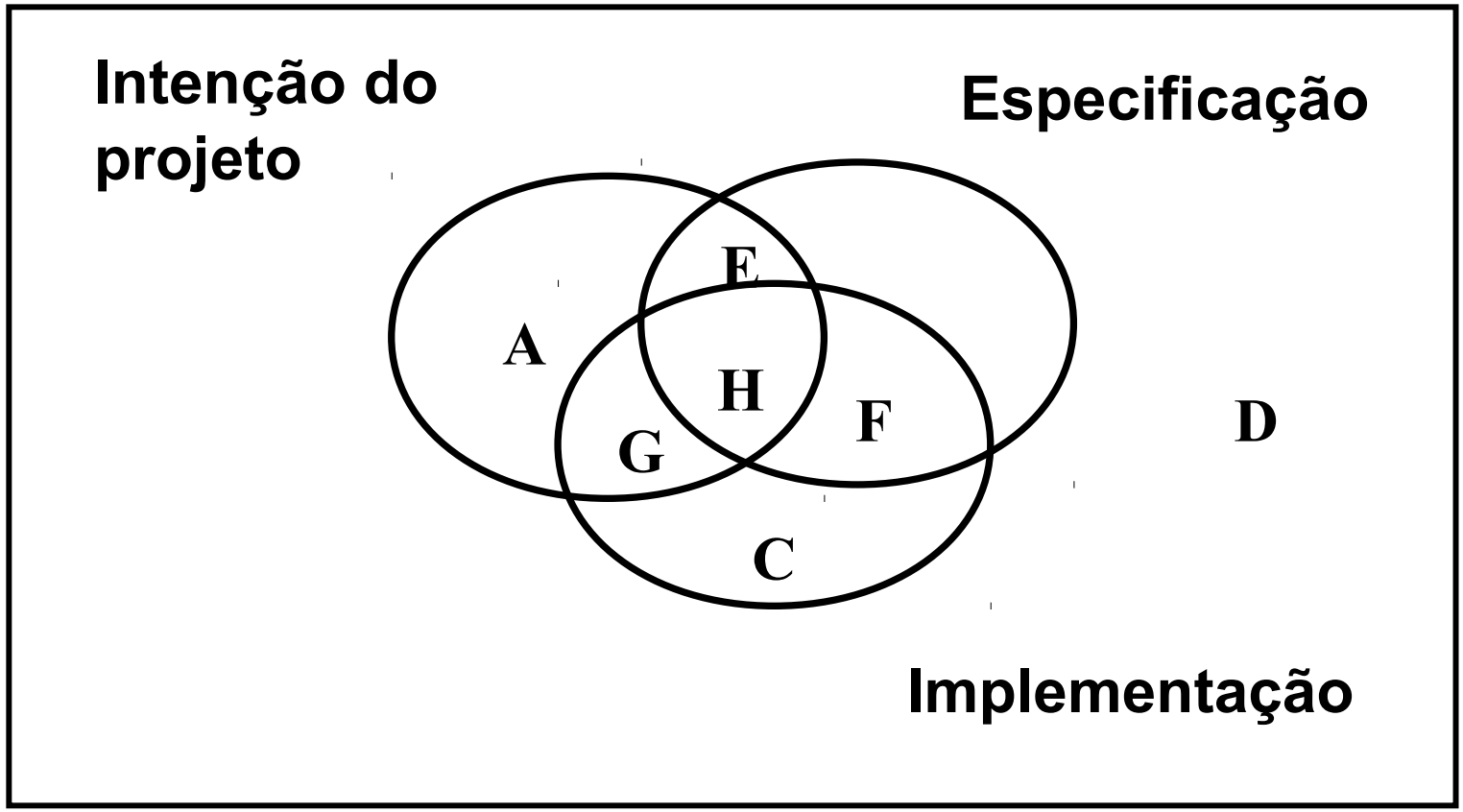
Fluxo de projeto



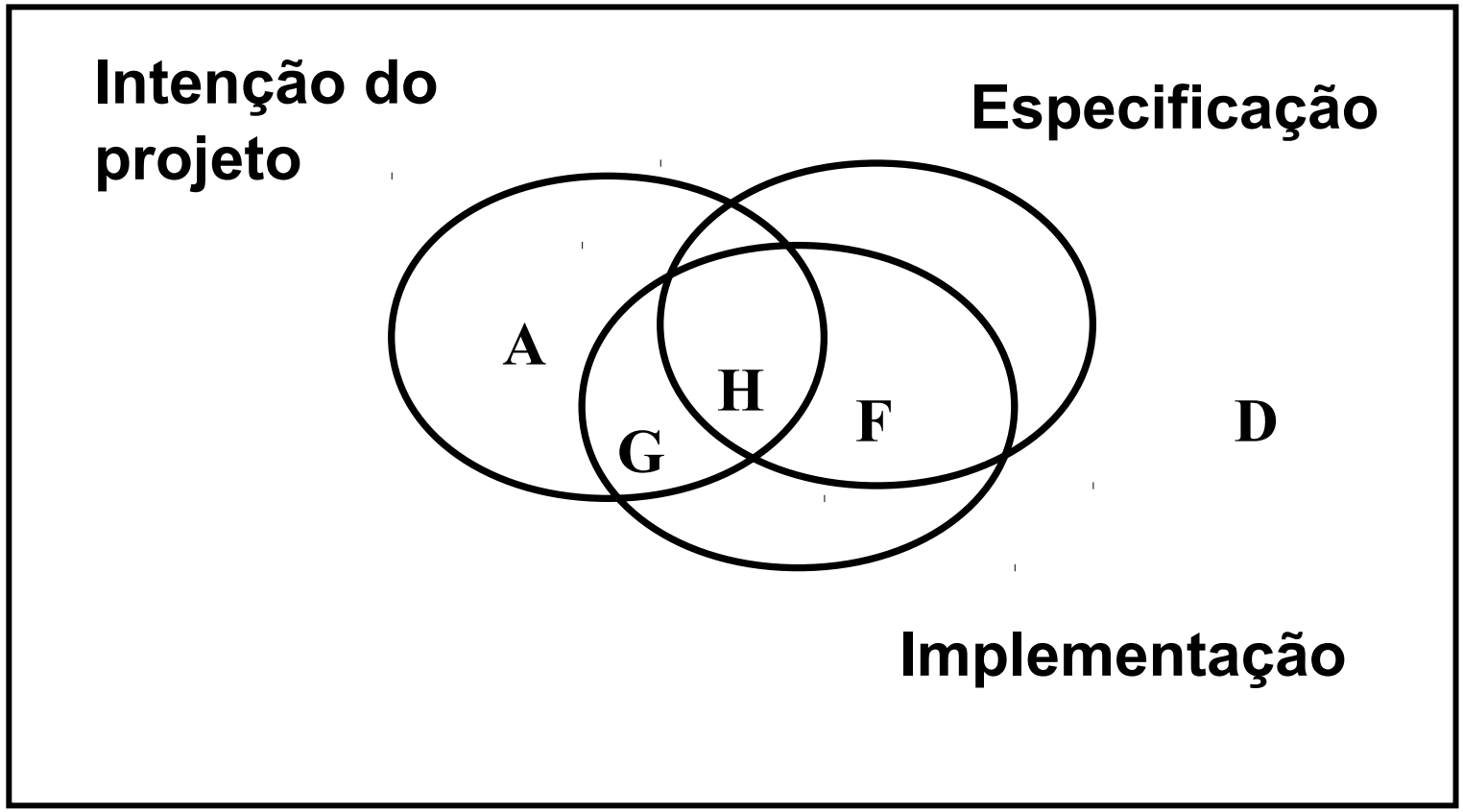
Projeto x Verificação



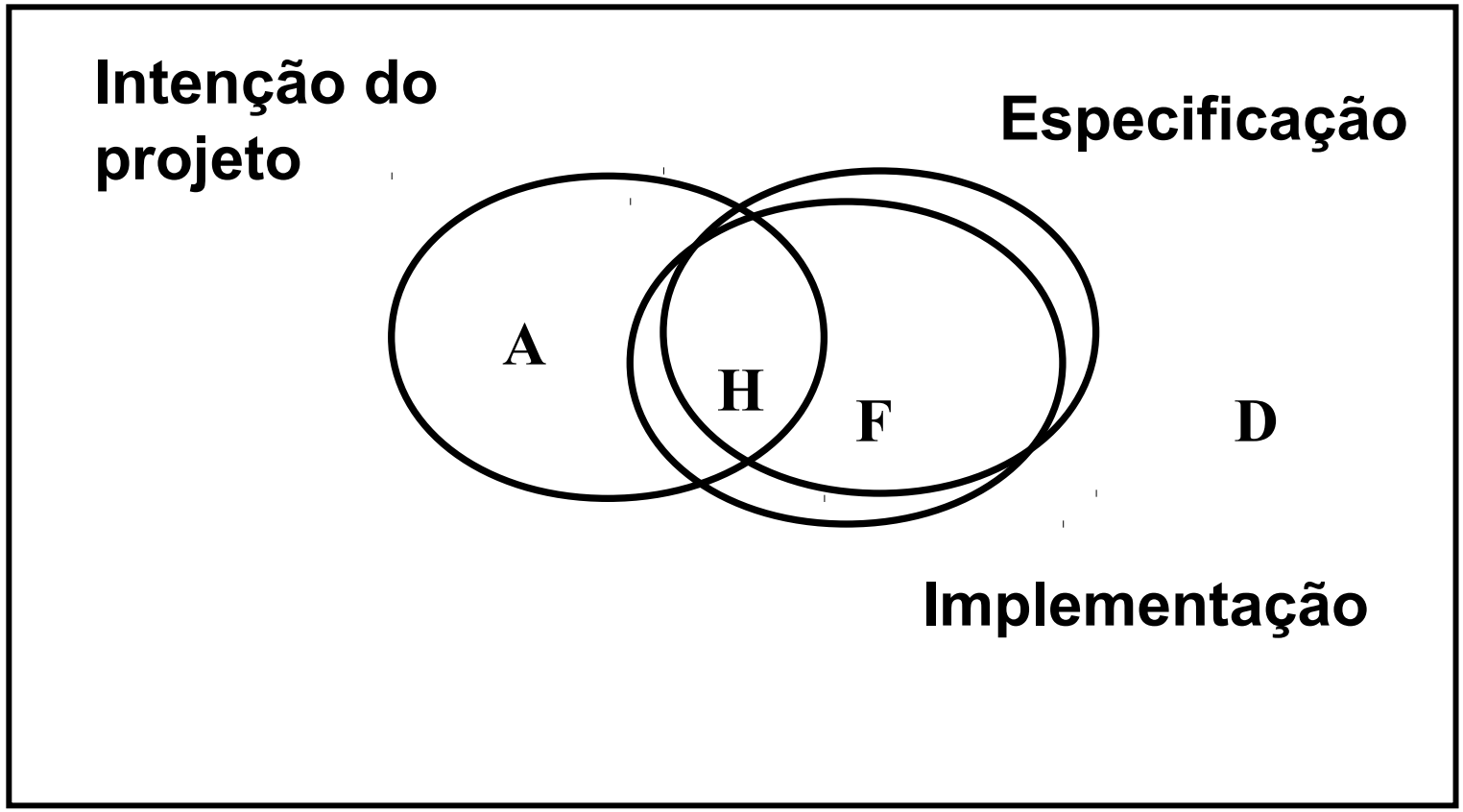
Projeto x Verificação



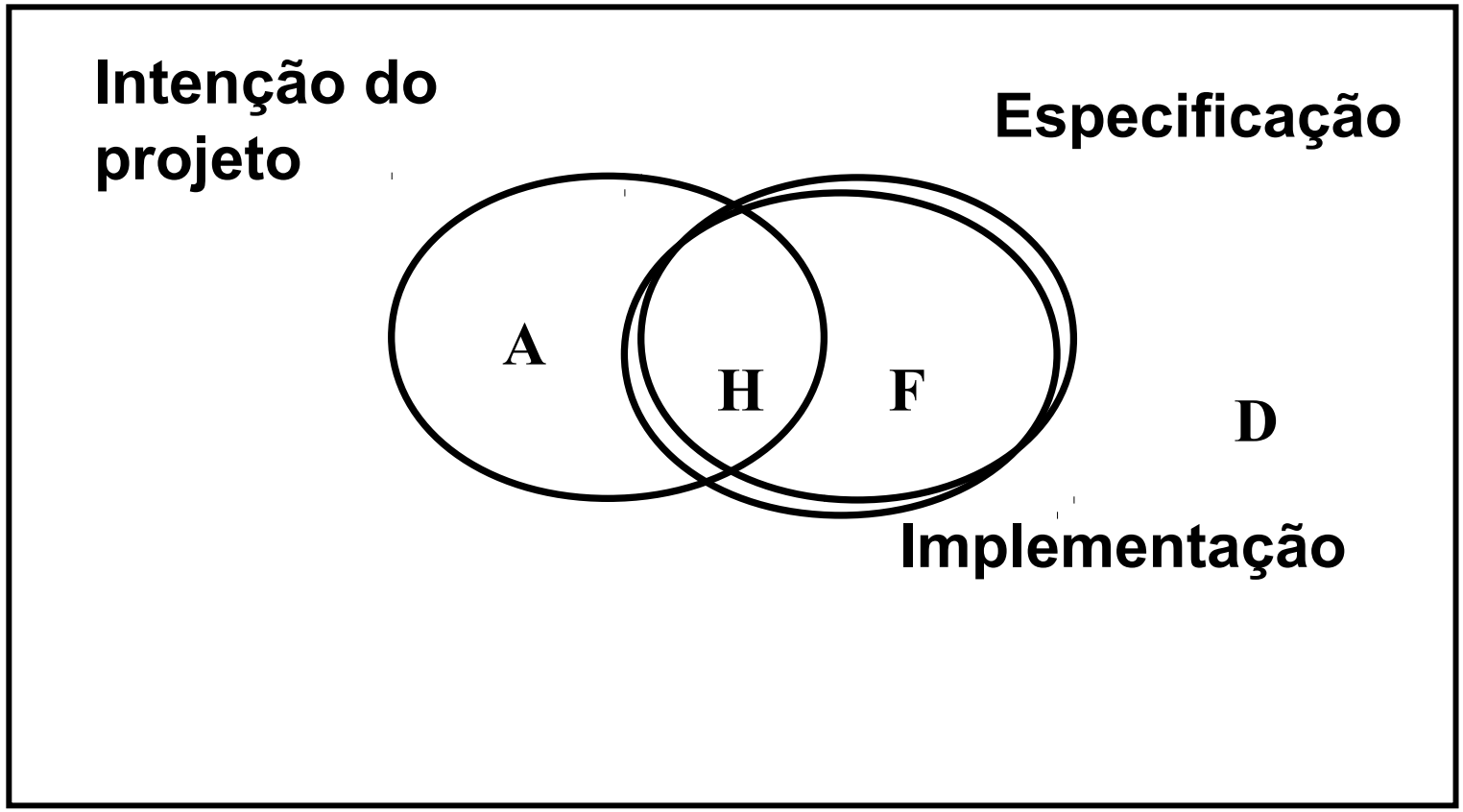
Projeto x Verificação



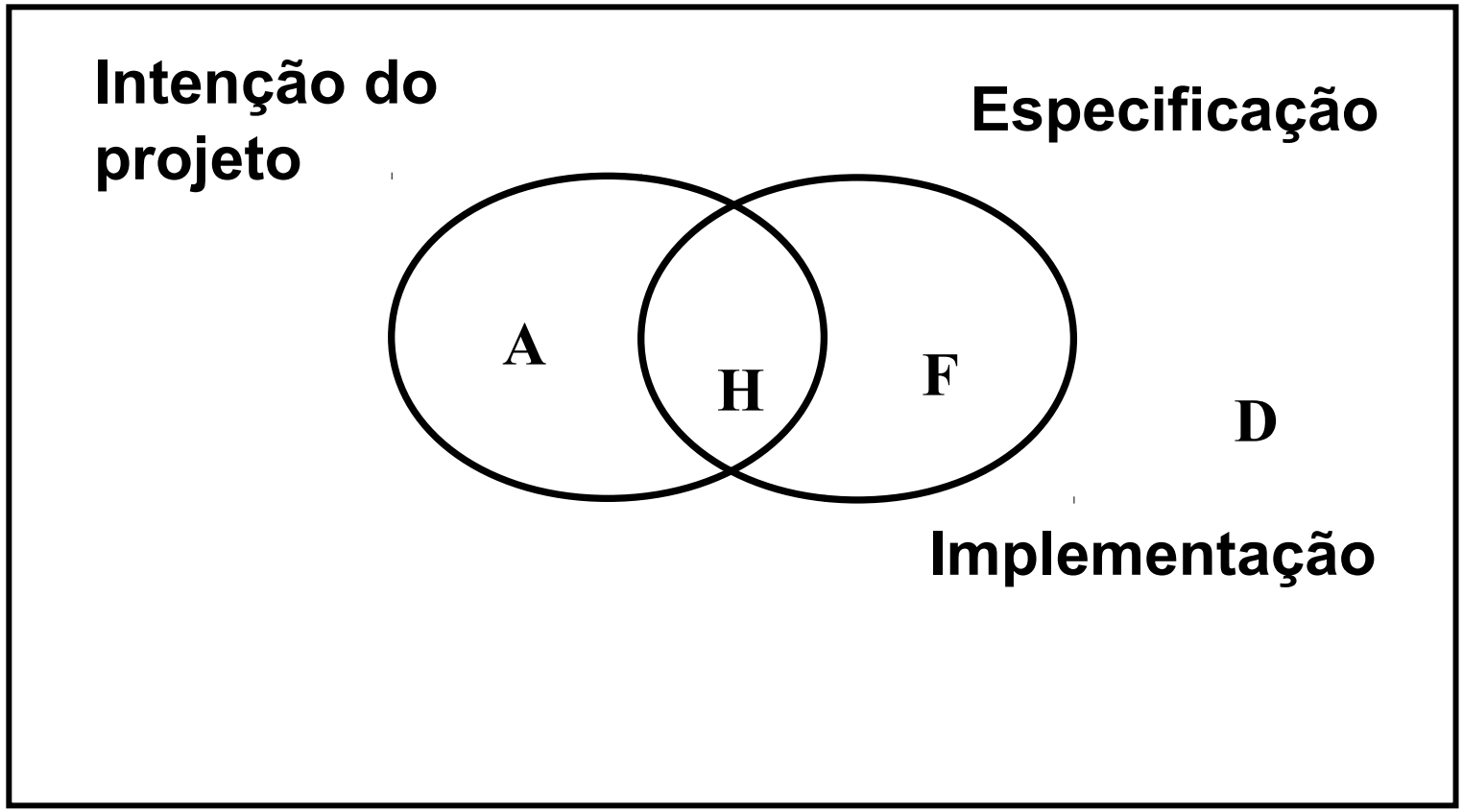
Projeto x Verificação



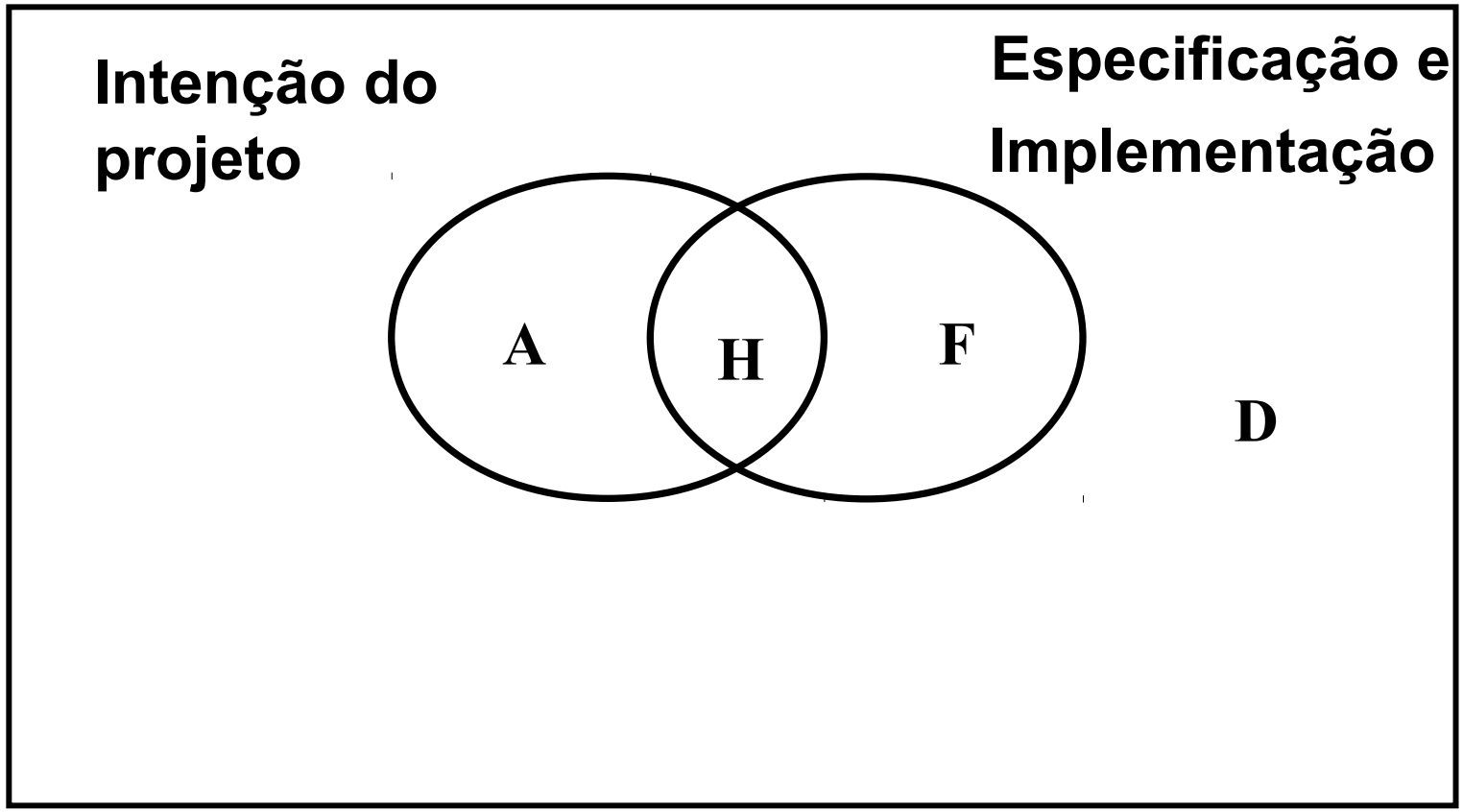
Projeto x Verificação



Projeto x Verificação



Projeto x Verificação



Nível hierárquico adequado

Verificação funcional pode ser realizada a vários níveis:

- Componente/unidade/sub-unidade, ...
- ASIC/FPGA/IP
- Sistema/SoC
- Placa

Nível hierárquico adequado

Como decidir qual nível?

- Nível mais baixo fornece mais observabilidade mas exige mais esforço,
- A nível mais alto os elementos menores são verificados implicitamente com menos esforço desde que os cenários e vetores de entrada sejam completos.

Decisão depende do projeto.

Faz parte do plano de verificação

- para cada elemento existe uma seção nele.

Verificação em vários níveis

- Na verificação no nível de sistema é suficiente verificar a interação dos componentes se uma verificação correta a nível de componentes foi feita.
- Verificação entre dois níveis adjacentes, do topo até a base.

Elementos da Verificação funcional

Para realizar a verificação funcional necessita-se de:

- Um **DUV**

O projeto a ser verificado: Design Under Verification.

- Um **Modelo de Referência**

Que implementa as funcionalidades do DUV de forma ideal.

- Um ambiente de verificação (**testbench**).

Responsável por gerar estímulos para o DUV e comparar as respostas do DUV com as respostas do Modelo de Referência.

Elementos da Verificação funcional

Tudo a mesma coisa:

- UUV (Unit Under Test).
- MUT (Model Under Test).
- DUT (Device Under Test, Design Under Test).
- EUV (Entity Under Verification).
- **DUV** (Design Under Verification).

Elementos da Verificação funcional

O **Modelo de Referência** pode ser escrito em qualquer linguagem de alto nível que possa se comunicar com SystemVerilog.

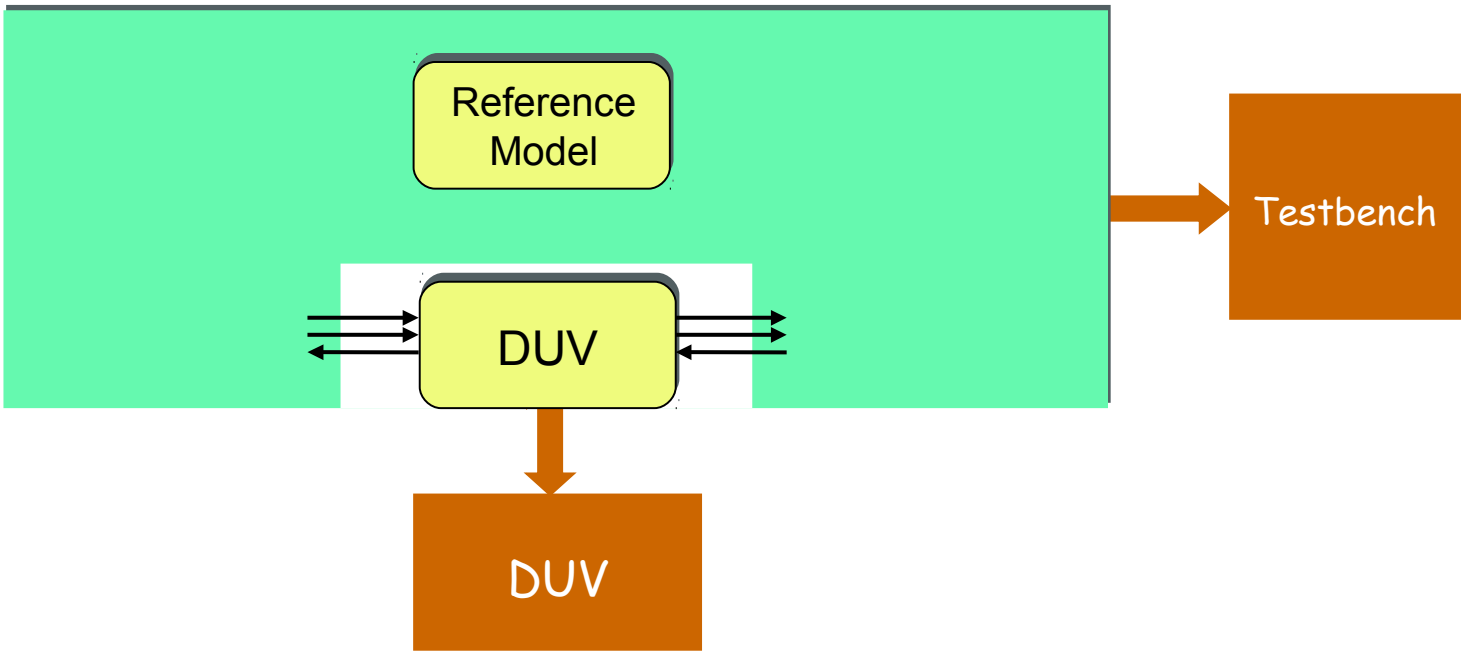
Normalmente escrito em C, C++, podendo ser escrito em outras linguagens.

Elementos da Verificação funcional

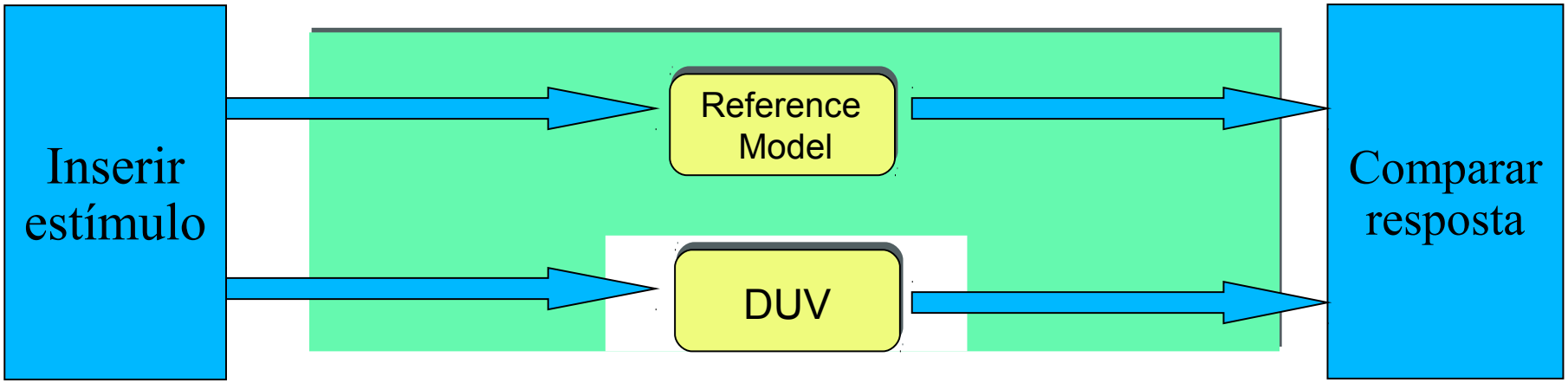
Testbench

- Montagem de verificação para simulação.
- Código escrito em SystemVerilog.
- Cria estímulos e verifica a resposta.
- Não tem entrada nem saída.
- Um modelo do universo em volta do projeto.
- Imprime mensagens quando o DUV apresenta comportamento inesperado.
- Caso tudo está ok imprime uma única mensagem no final.

Elementos da Verificação funcional



Função básica do Testbench



Requisitos da Verificação funcional

Verificação funcional deve:

Ser dirigida pela cobertura (Coverage-driven)

Quando parar de simular?

- **Cobertura** - processo que mostra que as funcionalidades especificadas estão sendo exercitadas.
- Auxilia na análise de redirecionamento de estímulo.
- Mostra o progresso da verificação.
- Determina quando terminar a verificação.

Deve possuir estímulos:

- Direcionados.
- Corner-cases.
- Reais.

Requisitos da Verificação funcional

Verificação funcional deve:

Apresentar estímulos com aleatoriedade direcionada (Random-constrained)

- Estímulos são gerados baseados em uma distribuição de probabilidade direcionada para o DUV.
- Visa alcançar os critérios de cobertura.
- Encontra erros não previstos.

Requisitos da Verificação funcional

Verificação funcional deve:

Ser auto verificável (Self-checking)

- O ambiente de verificação deve comparar as respostas do DUV com as respostas do Modelo de Referência sem intervenção humana.

Requisitos da Verificação funcional

Verificação funcional deve:

Ter o testbench implementado no nível de transação (Transaction Level Modeling (TL ou TLM))

Transação: uma estrutura de instruções e/ou dados que tem início e fim no tempo.

Exemplos:

- Pacote Ethernet
- Frame

Palavras-chave da Verificação

Automação

Eliminar intervenção humana no processo.

Realidade mostra que não é possível:

- processos mal definidos,
- precisando de invenção e criatividade humana.

Redundância

Usar três engenheiros (ou grupos) para um verificar o outro.

- Engenheiro de especificação
- Engenheiro de verificação
- Engenheiro de projeto

Quem pode errar?

Um projetista pode implementar uma funcionalidade de forma errada ?

Sim, o erro será descoberto pela verificação.

Um engenheiro de verificação pode testar uma funcionalidade de forma errada ?

Sim, um erro falso aparecerá na verificação.

O projetista e o engenheiro de verificação podem cometer o mesmo erro ?

Não, o erro será aceito na verificação.

Quem pode errar?

Um projetista pode esquecer de implementar alguma funcionalidade ?

Sim, a falha será descoberta pela verificação.

Um engenheiro de verificação pode esquecer de testar alguma funcionalidade ?

Não, um possível erro do projetista passará despercebido.

Verificação funcional

Pode provar a presença de erros, mas não pode provar a ausência de erros.

É preciso saber quando pode-se terminar o processo de verificação.

medição de ***cobertura***

Abordagens de Verificação

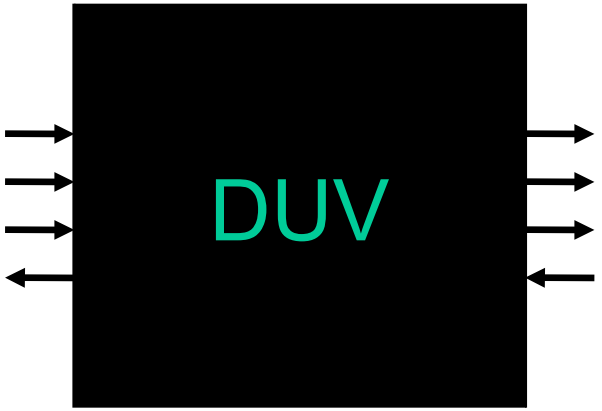
Ver a descrição RTL como:

Black Box – boa para verificação!

Grey Box – pode ser usada, mas cuidado!

White Box – ruim para verificação!

Black Box

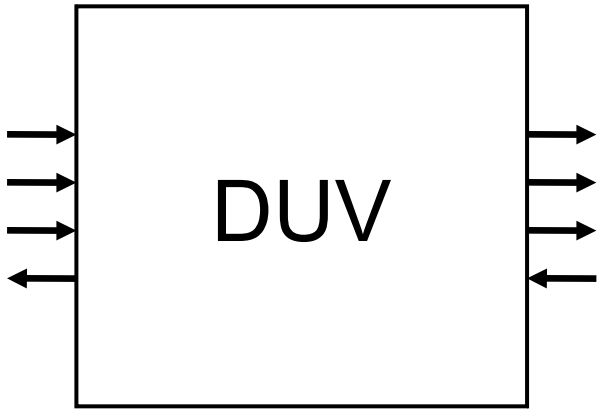


Entradas, saídas, função

Função bem documentada (ou não...)

Para verificar, é preciso entender a função e prever as saídas sabendo as entradas.

White Box

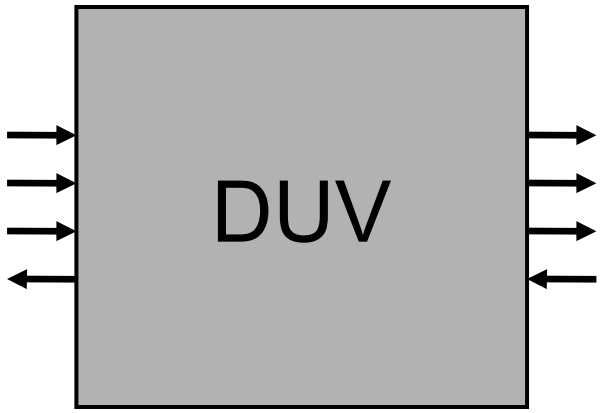


Todas as variáveis internas visíveis.

Podem ser acessadas para verificação.

Para verificação de unidades pequenas nas folhas da hierarquia

Grey Box



Uma seleção restrita de variáveis internas pode ser usado para verificação.

Exemplo: registradores de um processador

Plano de Verificação

Especificação do processo de verificação;

Define o quê será verificado e como.

- Abordagem tradicional:

faça como quiser

- Abordagem nova:

uso de métricas para saber quando a verificação estiver completa: **Cobertura da Verificação**

idealmente a definição de sucesso “de primeira”.

Plano de Verificação

Feito a partir da especificação do DUV.

Define os cenários de verificação (testbenches a serem escritos):

define a complexidade deles,

as dependências entre eles.

A partir daí é feito um cronograma:

recursos (humanos, máquinas, etc.) necessários,

recursos disponíveis

Plano de Verificação

Pertence à equipe:

todo mundo envolvido é responsável,

tudo mundo deve contribuir.

Plano de Verificação não é algo novo, já é usado por:

NASA

FAA

Software

Conteúdo do Plano de Verificação

Resumo do sistema

Níveis de abstração

Tecnologias de Verificação

Modelos de referência a serem usados

Fluxograma da verificação

Definição dos estímulos

Testes de regressão

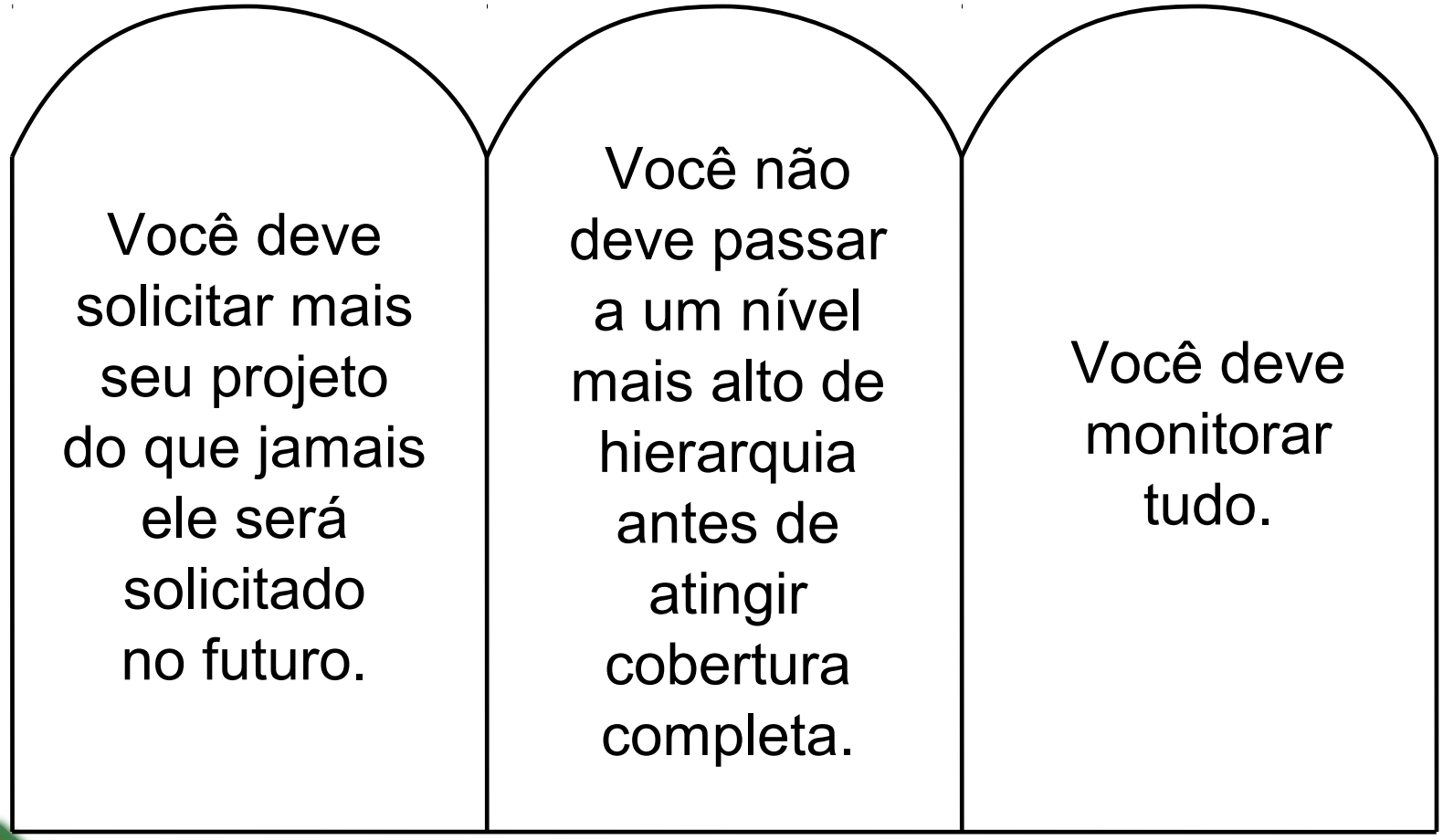
Conteúdo do Plano de Verificação

Gerência de falhas

Plano de recursos

Cronograma

Os três mandamentos da verificação funcional



Metodologia BVM

Brazil-IP

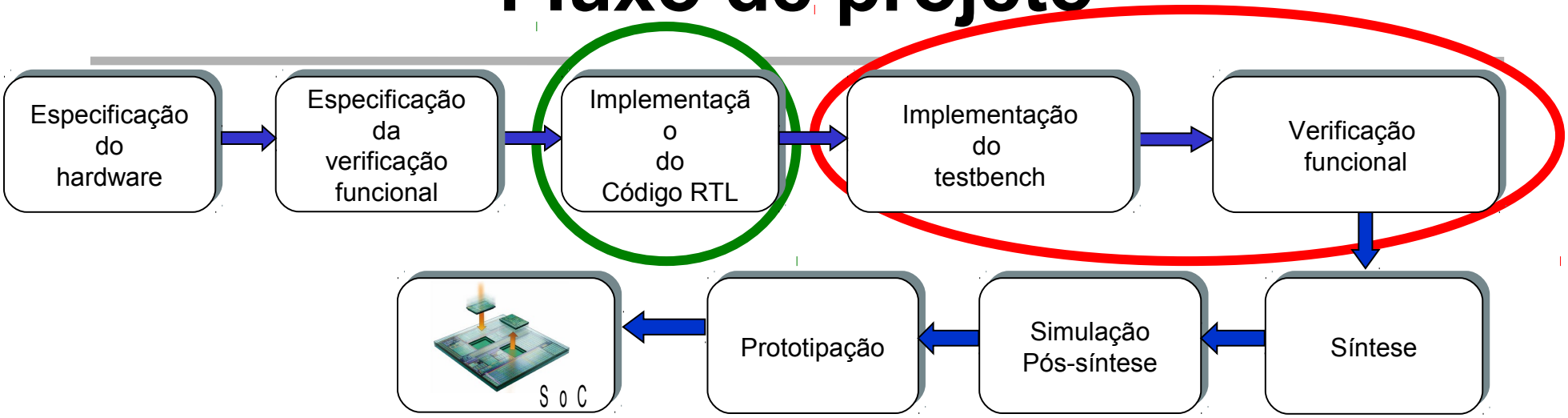
Metodologia BVM é VeriSC usando OVM.

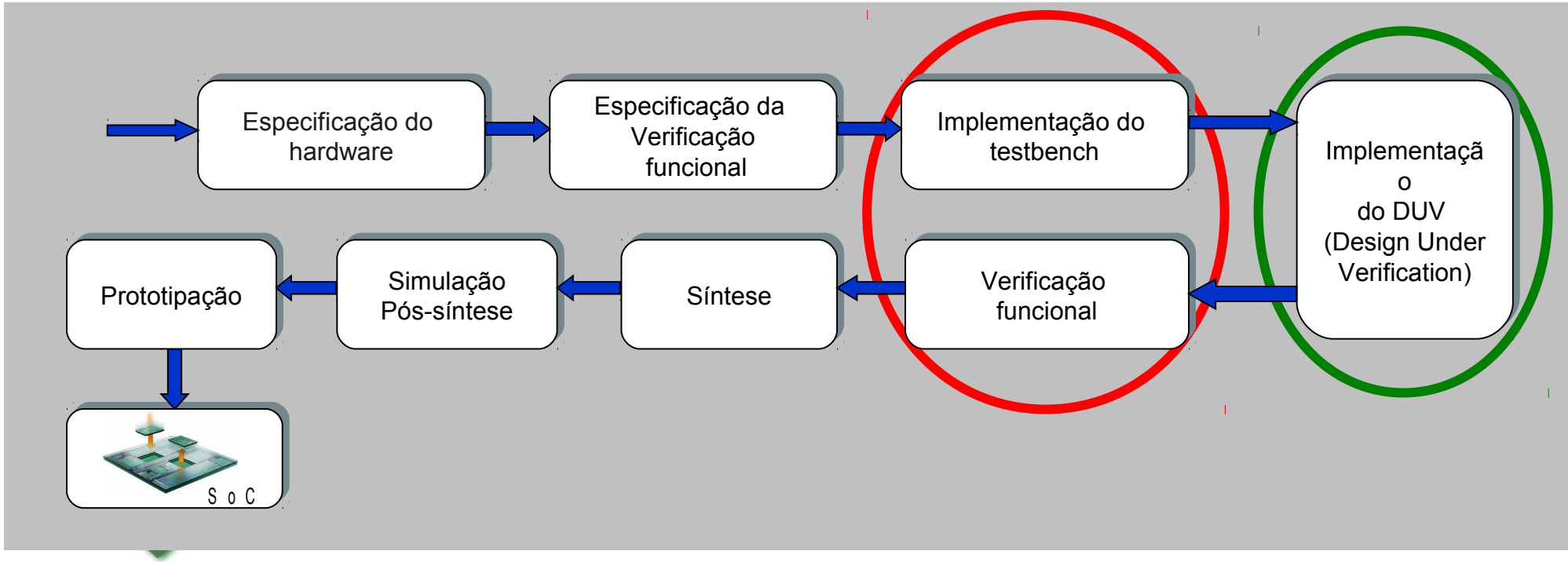
VeriSC surgiu no contexto do projeto Brazil-IP.

Brazil-IP é um esforço de universidades brasileiras para a capacitação de pessoas para a produção de IPs.

VeriSC foi defendida como tese de doutorado.

Fluxo de projeto





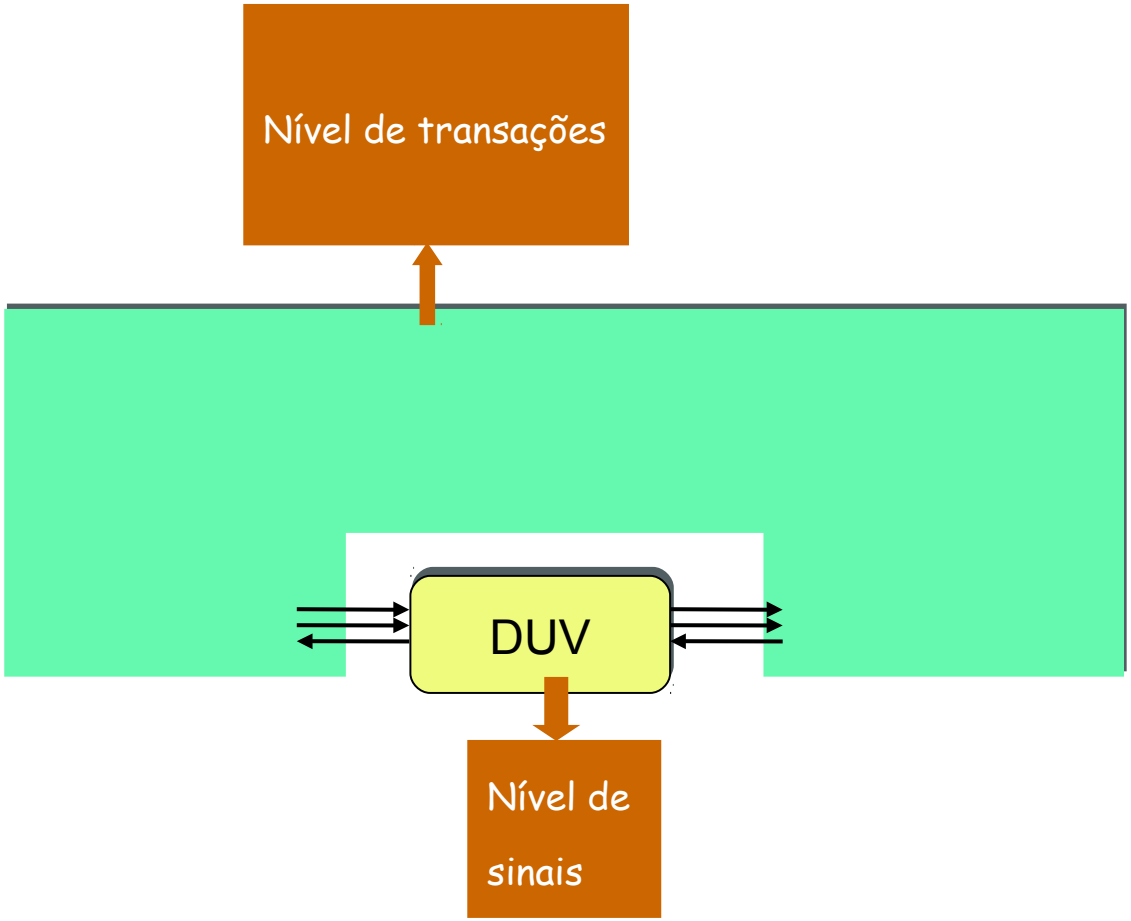
Metodologia VeriSC

Inicia pela implementação do testbench.

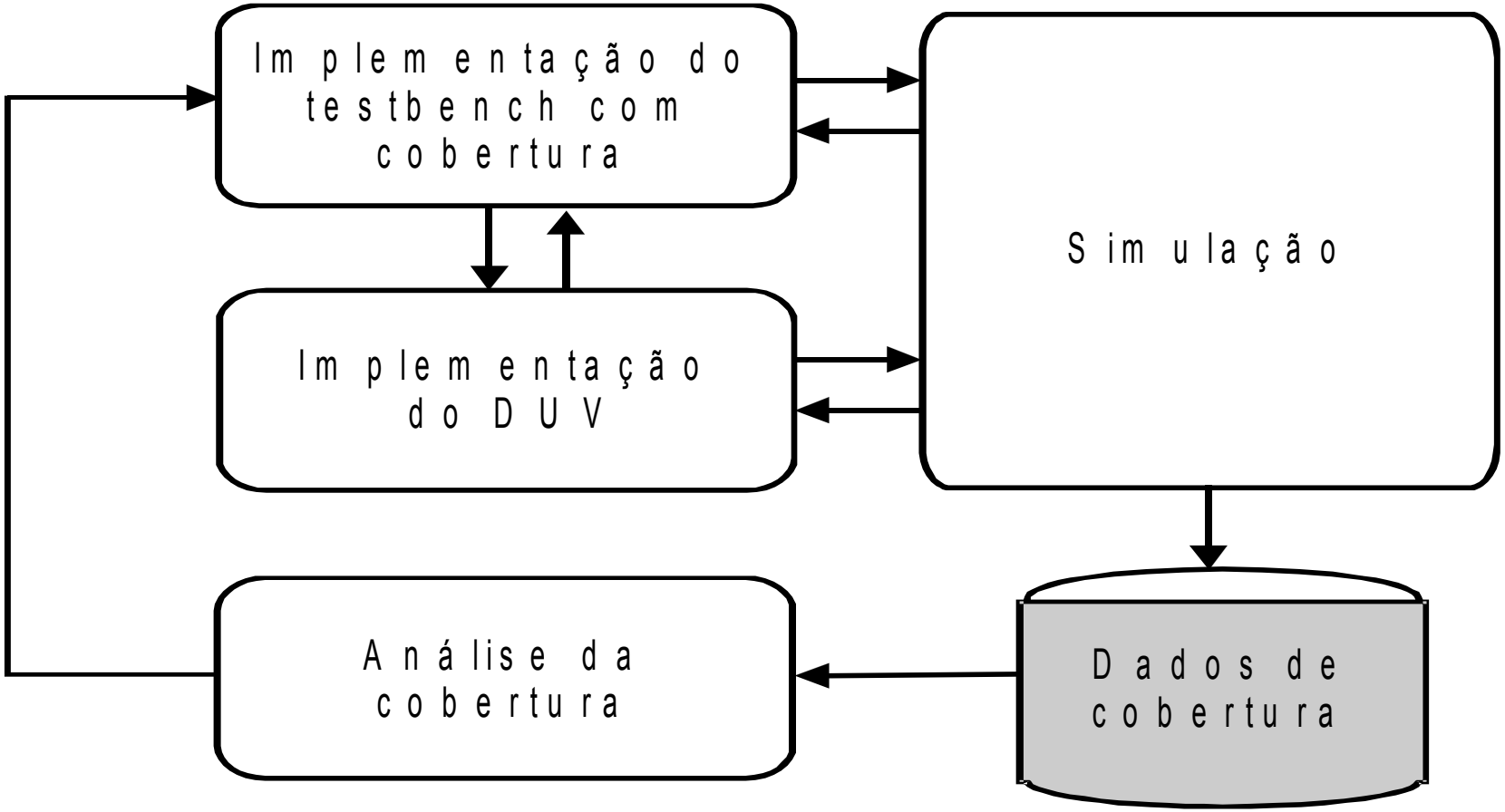
Implementa um mecanismo para simular a presença do DUV unicamente com os elementos do testbench, sem usar nenhum artifício extra.

Todas as partes do testbench podem estar prontas e simuladas antes do início do desenvolvimento do DUV.

Testbench da metodologia VeriSC



Resumo


Implementação do testbench com cobertura
Simulação
Implementação do DUV
Análise da cobertura
Dados de cobertura

Testbench

Definição:

Montagem em volta do *Design Under Verification*

Transação

Definição:

Uma operação que inicia num determinado momento no tempo e termina em outro.

É caracterizada pelo conjunto de instruções e dados necessárias para realizar a operação.

Exemplos:

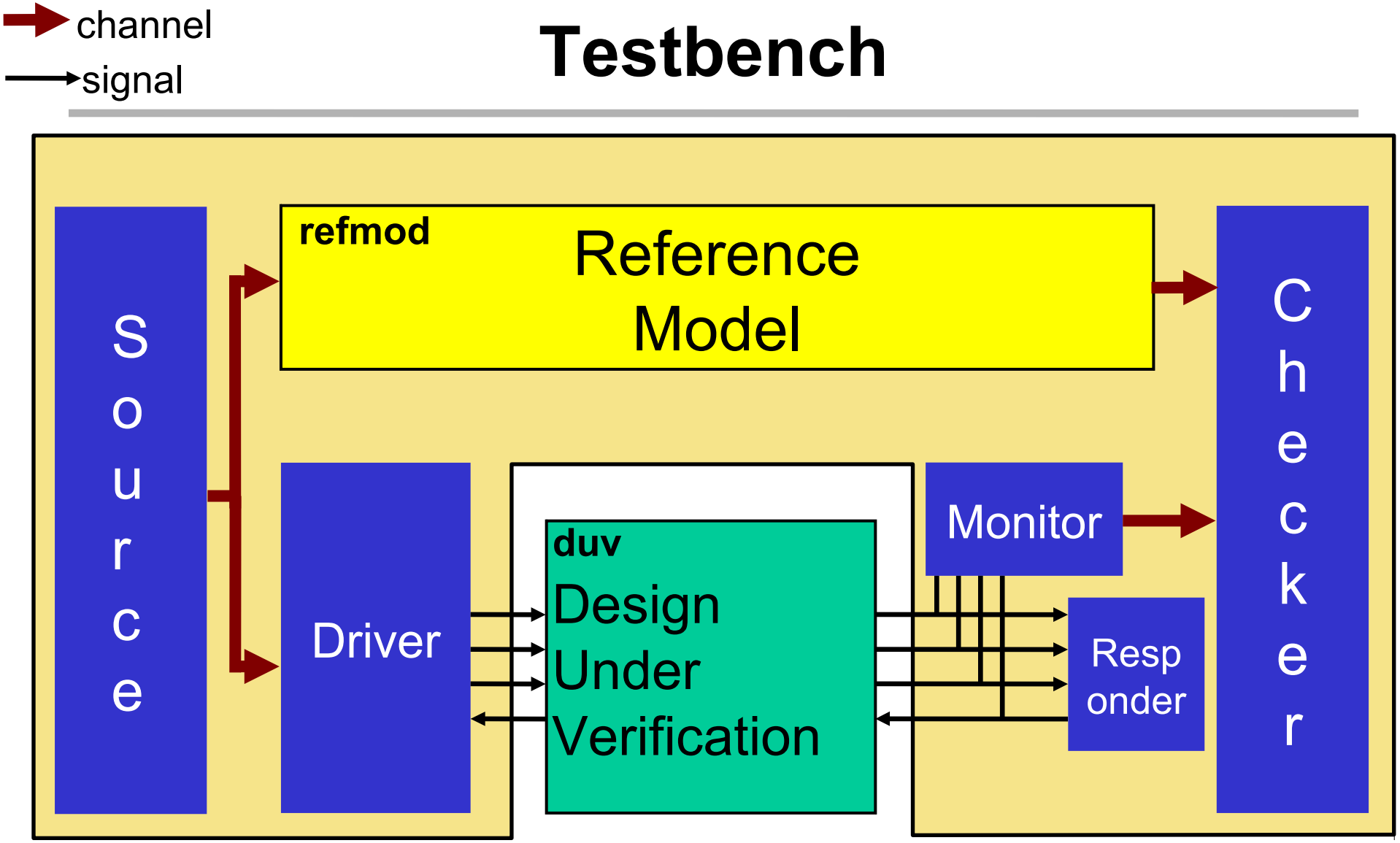
transmissão de um pacote ethernet

recepção de uma imagem

uma escrita num barramento

execução de uma instrução de máquina de um processador

Testbench


channel
signal
refmod
Reference Model
Source
Driver
duv
Design Under Verification
Monitor
Resp onder
Checker

Elementos de um testbench

Source

Envia transações de entrada para o driver e para o reference model.

Checker

Compara as transações de saída recebidas do monitor com as de um reference model.

É bom ser reutilizável, ou seja, depender pouco do DUV.

Reference Model

Tipicamente *timeless*

Elementos de um testbench

Driver

Recebe transações de entrada e as converte em transições de sinais da interface de entrada do DUV.

Monitor

Observa sinais da interface de saída do DUV e gera transações de saída que ele repassa para o checker.

Responder

Observa sinais da interface de saída do DUV e implementa o protocolo de comunicação.

Regras de projeto

Driver

Acesso ao DUV somente pela interface do mesmo;

Transação flui do source para o driver mas nunca na direção oposta.

Monitor

Acesso somente pela interface do DUV;

Envia transações ao checker;

Independente de driver e checker.

Responder

Faz verificação de protocolo (baixo nível);

Regras de projeto

Source

Não envia sinais diretamente para o DUV

Checker

Nunca escreve (força sinal) dentro do DUV;

Pode eventualmente ler informação do DUV (por exemplo registradores internos);

Reference Model

Modela a funcionalidade, mas não a interface

Ferramentas usadas na Metodologia BVM

Linguagem SystemVerilog.

OVM (Open Verification Methodology).

Ferramentas usadas na Metodologia BVM

Linguagem SystemVerilog.

OVM (Open Verification Methodology).

Ferramenta eTBc (Easy Testbench Creator) de geração semi-automática de testbenches.

Passos da metodologia BVM

Passo 1: Testbench Conception

1.1 Single Refmod

1.2 Double Refmod

1.3 DUV Emulation

Passo 2: Hierarchical Refmod Decomposition

2.1 Refmod Decomposition

2.2 Single Hierarchical Refmods

2.3 Hierarchical Refmods Verifications

Passos da metodologia BVM

Passo 3: Hierarchical Testbench

3.1 Double Hierarchical Refmods

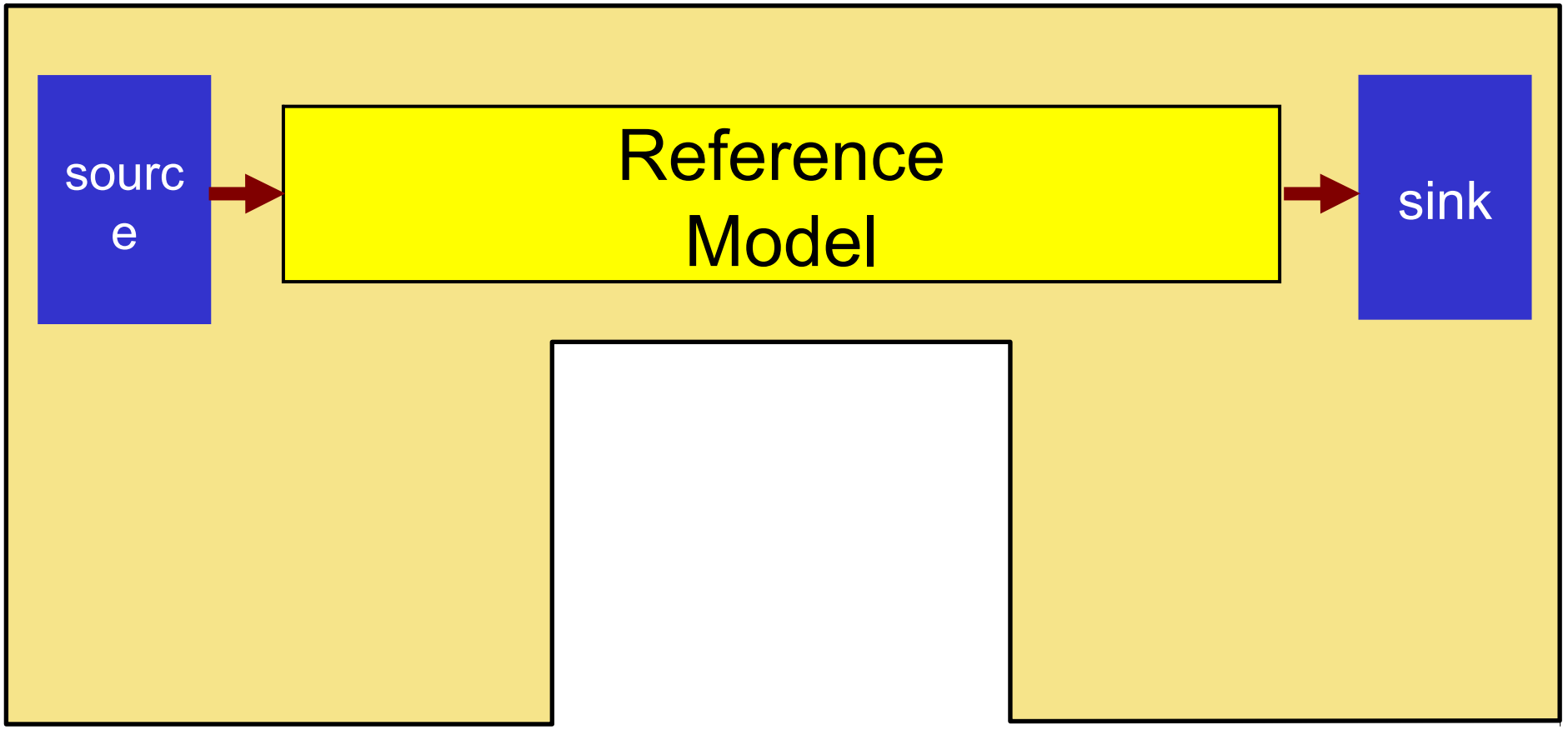
3.2 Hierarchical DUV Emulation

3.3 Hierarchical DUV

Passo 4: Full Testbench

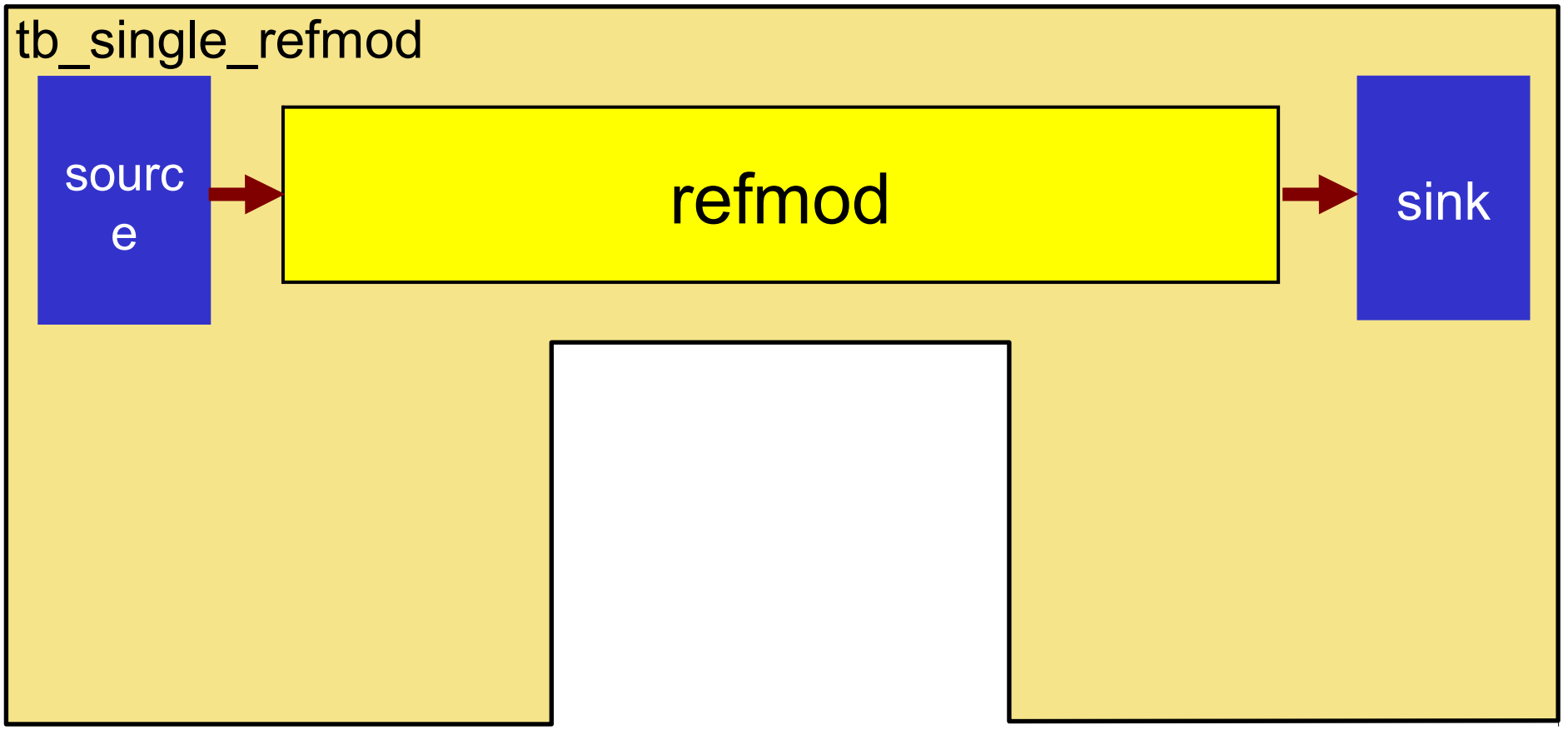
Passo 1: Testbench Conception

1.1 Single Refmod



Passo 1: Testbench Conception

1.1 Single Refmod – **Templates (eTBc)**



Passo 1: Testbench Conception

1.1 Single Refmod – **Templates (eTBc)**

- **source**
- **refmod**
- **sink**
- **tb_single_refmod**
- **trans**
- **tb_tcl**
- **Makefile_single_refmod**

Passo 1: Testbench Conception

1.1 Single Refmod

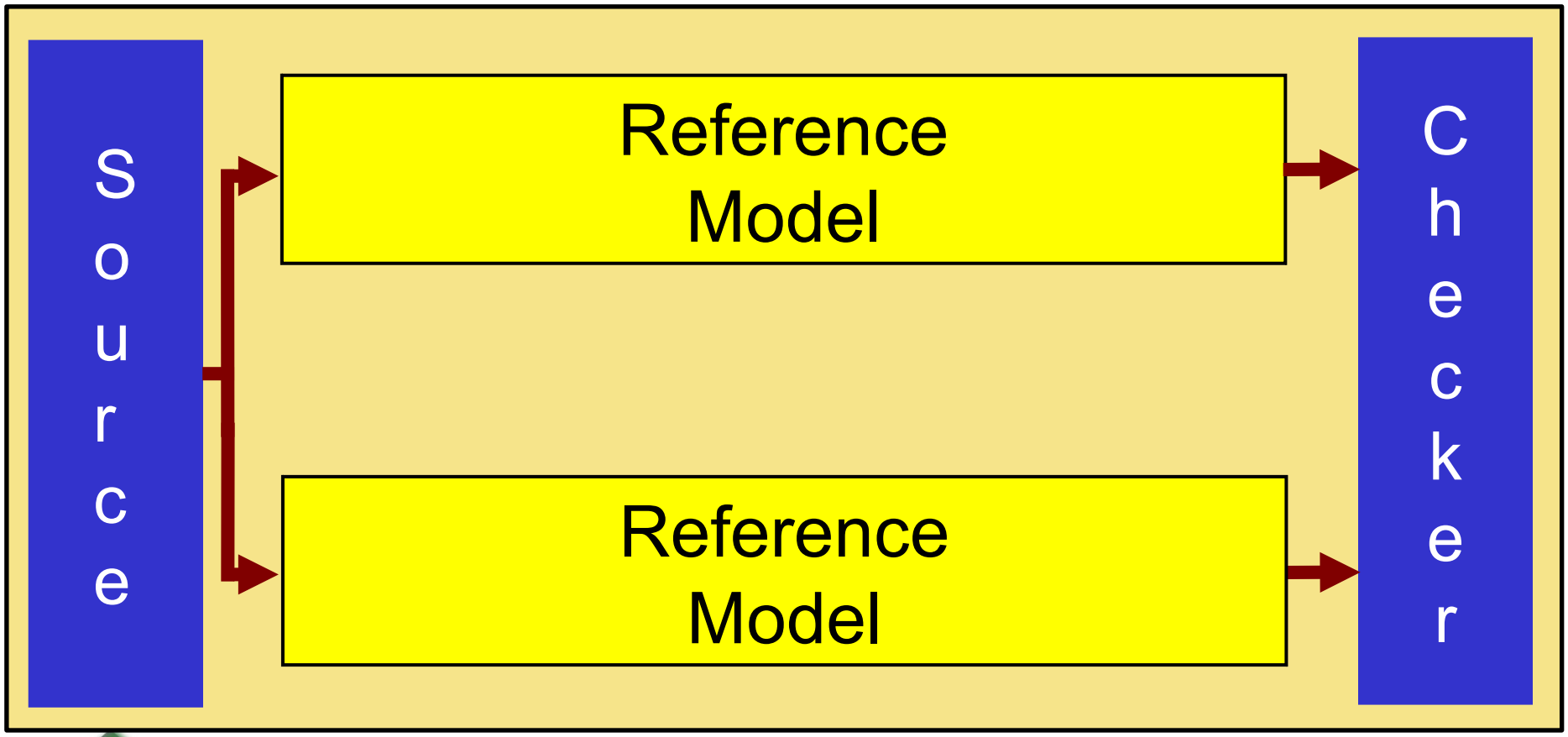
Reference model é testado em sua capacidade de interagir com o testbench.

Pré-Source é um subconjunto do Source com quase as mesmas funcionalidades.

Sink possui um subconjunto das funcionalidades do Checker.

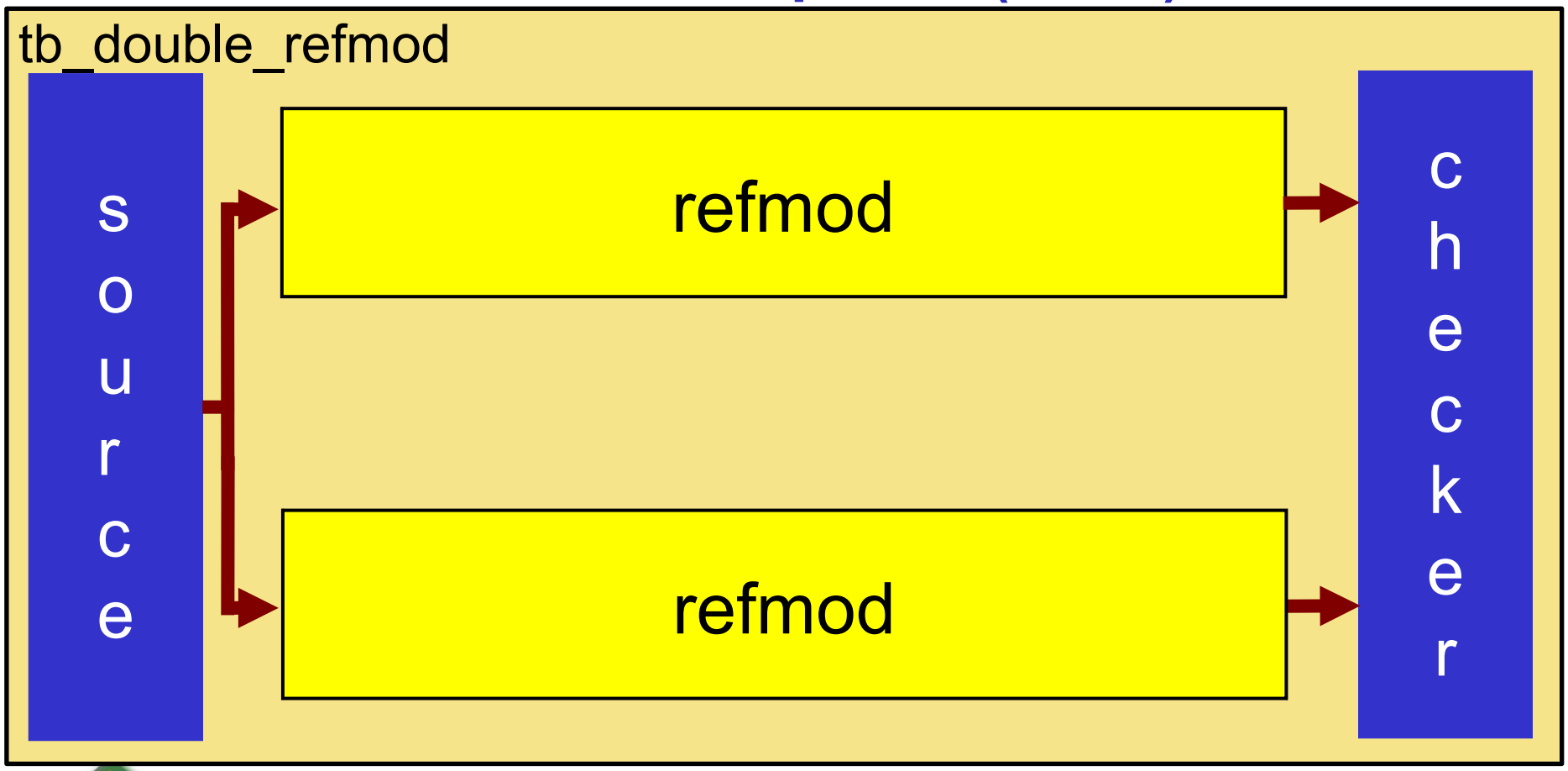
Passo 1: Testbench Conception

1.2 Double Refmod



Passo 1: Testbench Conception

1.2 Double Refmod – **Templates (eTBc)**



Passo 1: Testbench Conception

1.2 Double Refmod – **Templates (eTBc)**

- **source**
- **refmod**
- **checker**
- **tb_double_refmod**
- **trans**
- **tb_tcl**
- **Makefile_double_refmod**

Passo 1: Testbench Conception

1.2 Double Refmod

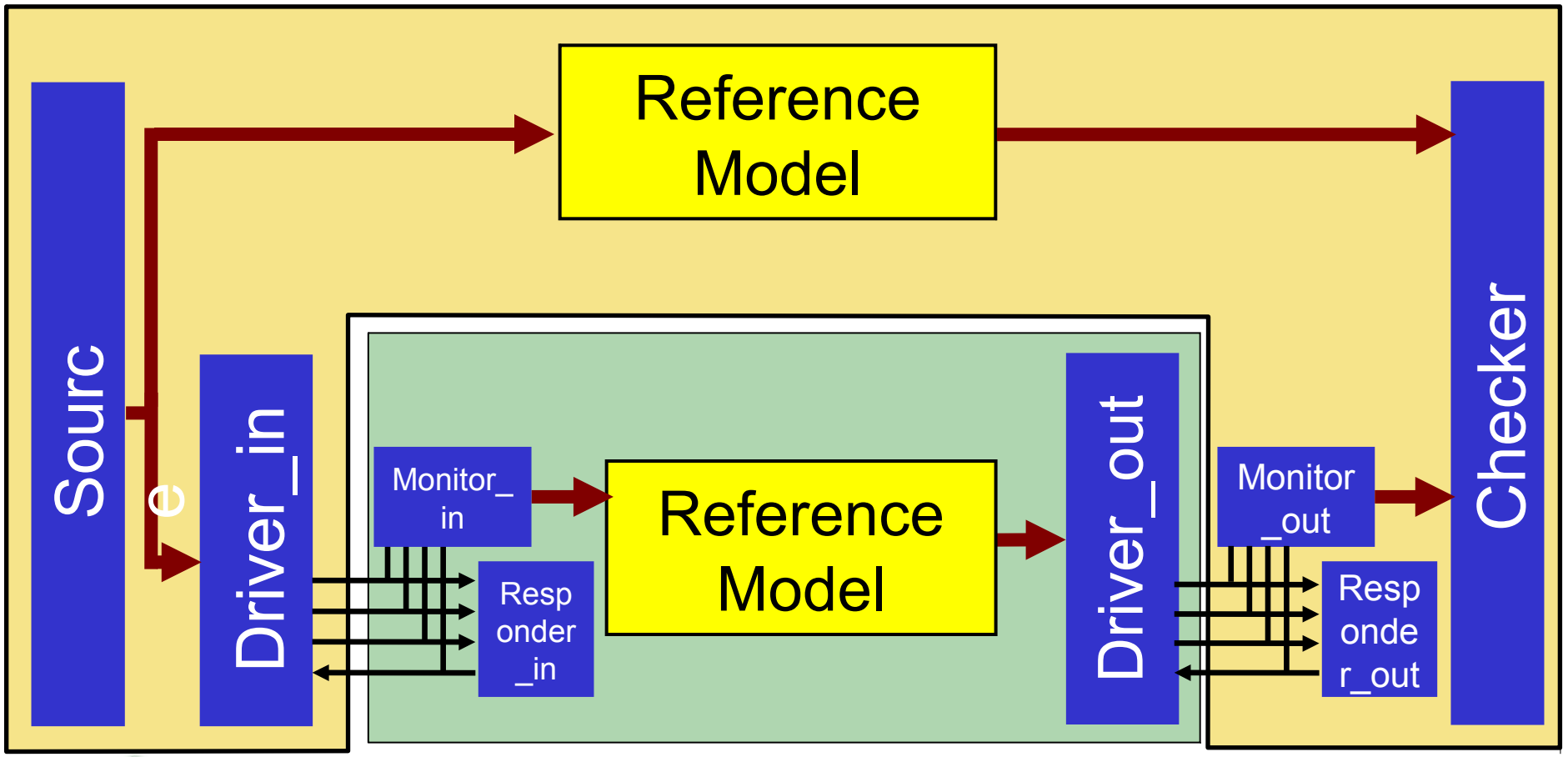
Testar Source e Checker.

Pode-se inserir erros em uma das instâncias do reference model para testar a capacidade do Checker de detectá-los.

Checker emite mensagem somente em caso de ERRO.

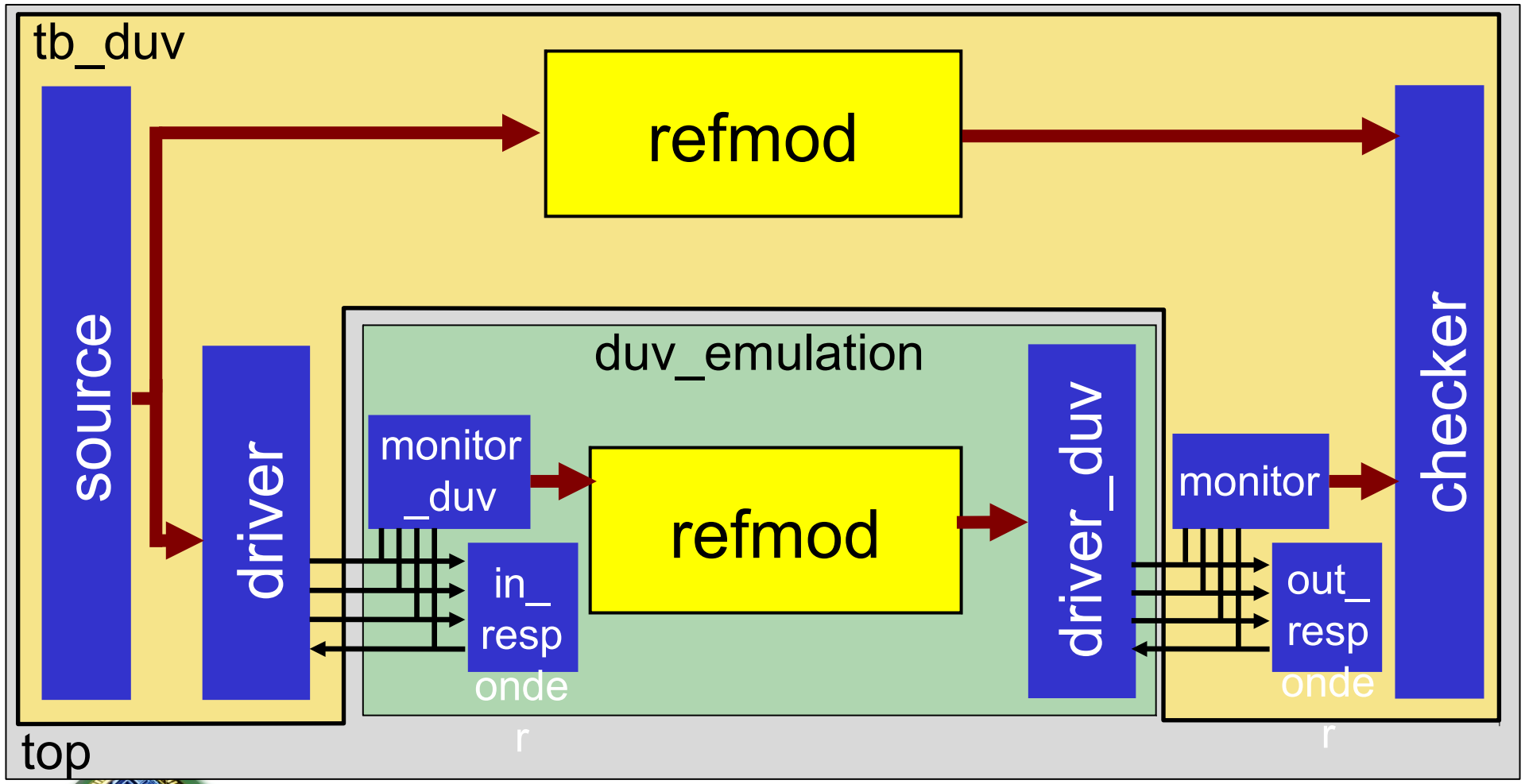
Passo 1: Testbench Conception

1.3 DUV Emulation



Passo 1: Testbench Conception

1.3 DUV Emulation – **Templates (eTBc)**



Passo 1: Testbench Conception

1.3 DUV Emulation – Templates (eTBc)

- source
- refmod
- checker
- driver
- monitor
- driver_duv
- monitor_duv
- in_responder
- out_responder
- duv_emulation
- tb_duv
- top
- trans
- top_tcl
- gene_clock
- axi_cover
- Makefile_duv_emulation

Passo 1: Testbench Conception

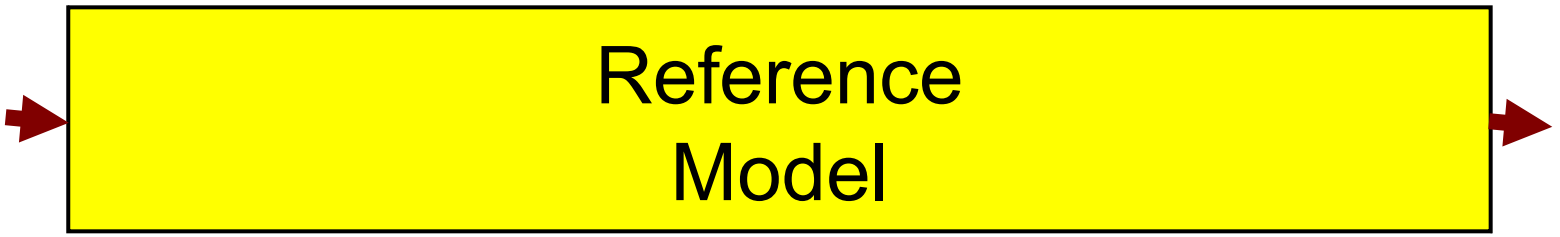
1.3 DUV Emulation

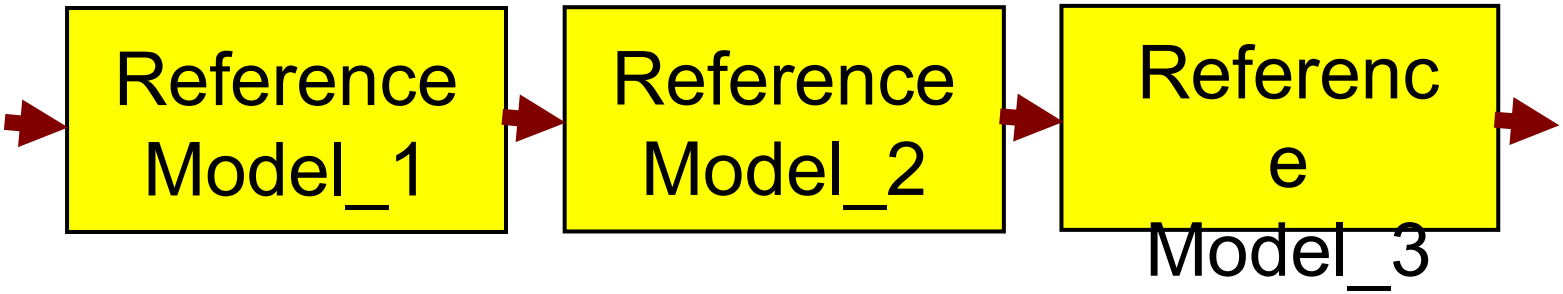
Testar Driver(s), Responder(s) e Monitor(es)

Driver_in-Monitor_in e Driver_out-Monitor_out precisam ser simétricos.

Passo 2: Hierarchical Refmod Decomposition

2.1 Refmod Decomposition





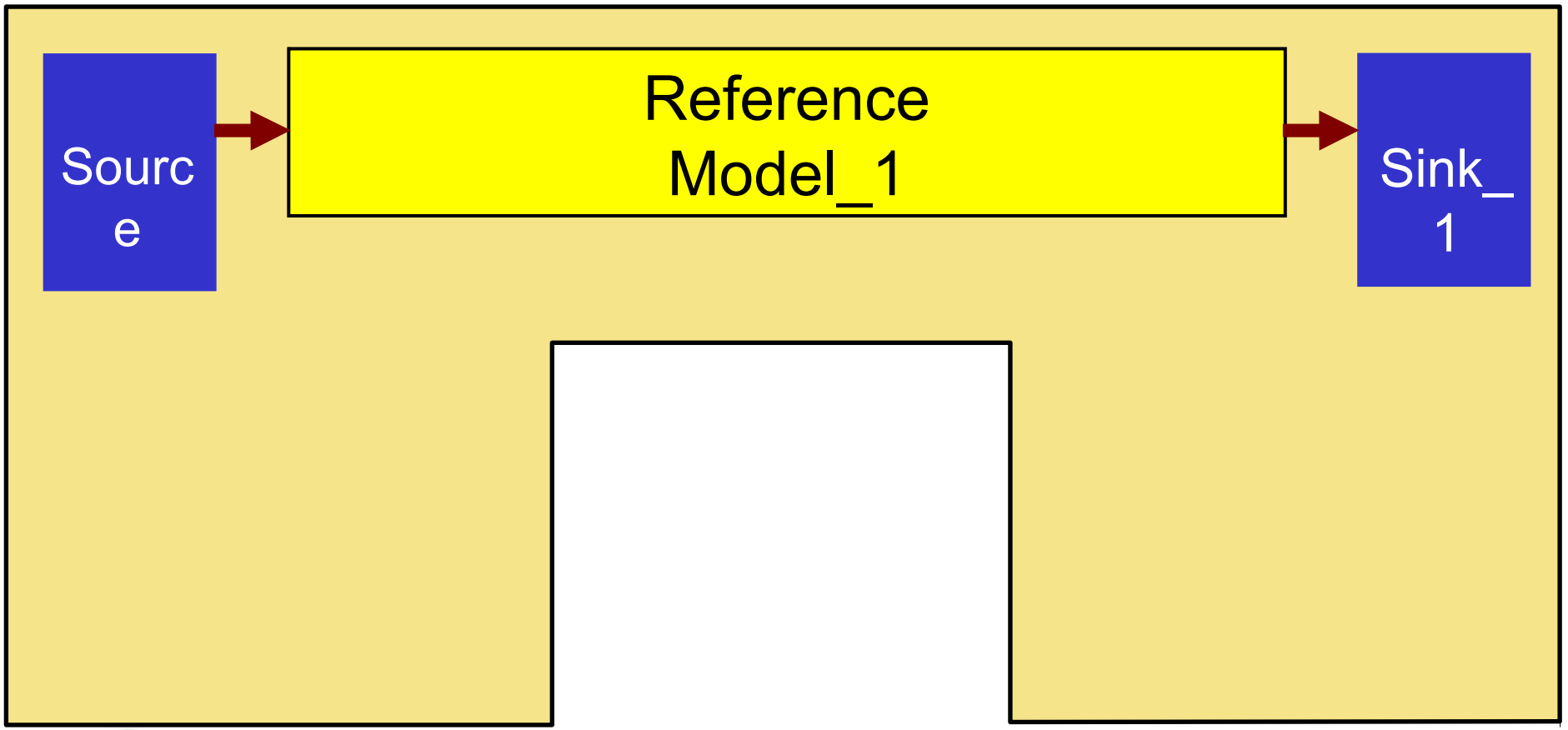
Passo 2: Hierarchical Refmod Decomposition

2.1 Refmod Decomposition

Reference Model é dividido hierarquicamente como o DUV.

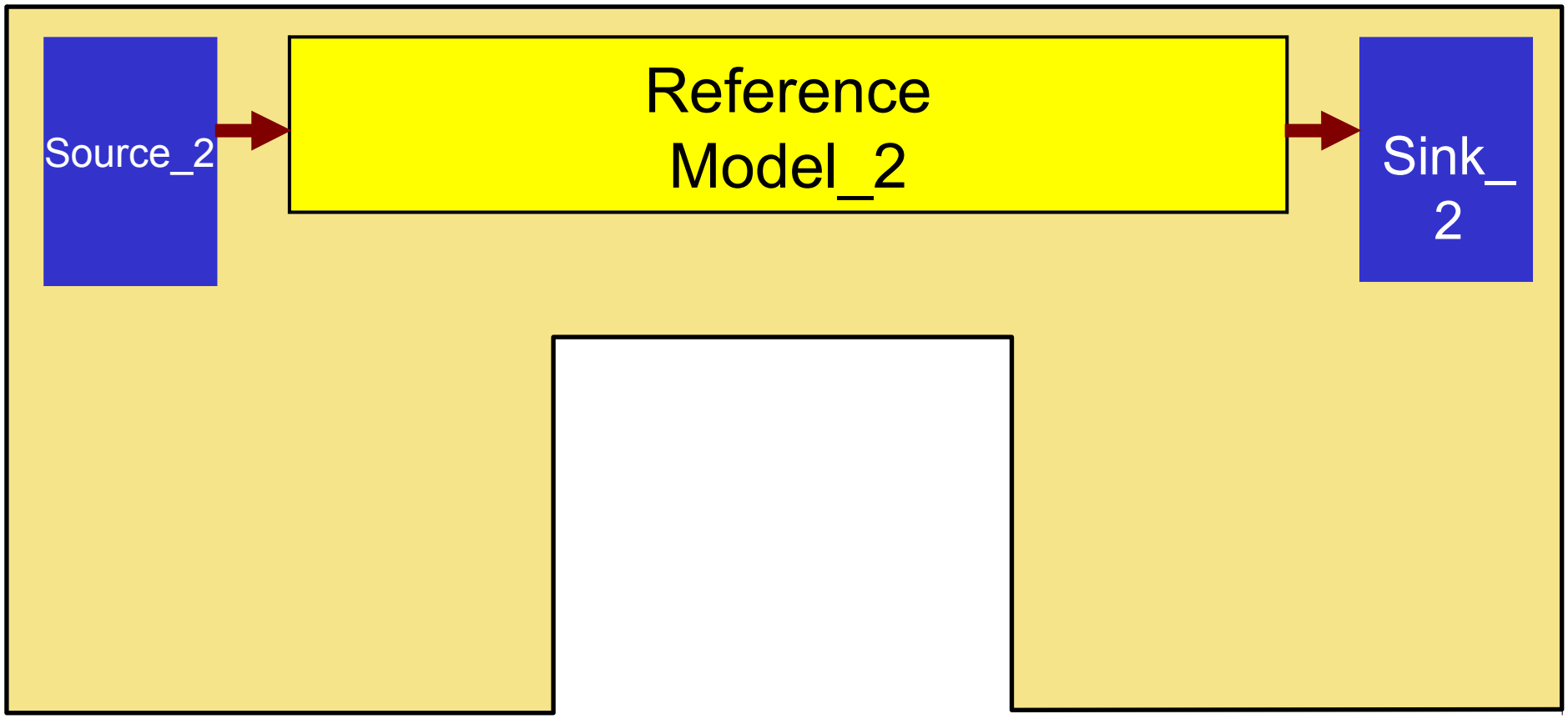
Passo 2: Hierarchical Refmod Decomposition

2.2 Single Hierarchical Refmods



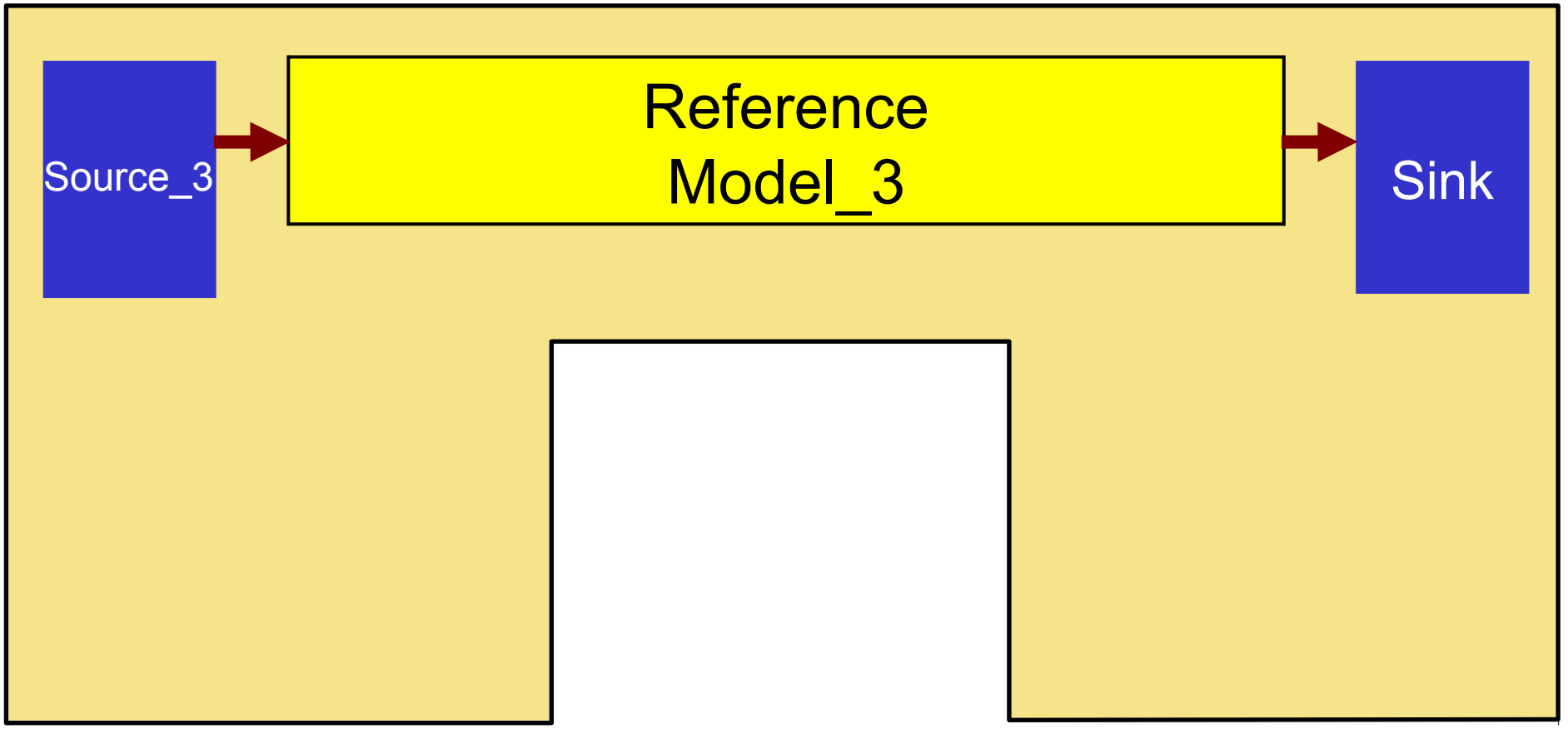
Passo 2: Hierarchical Refmod Decomposition

2.2 Single Hierarchical Refmods



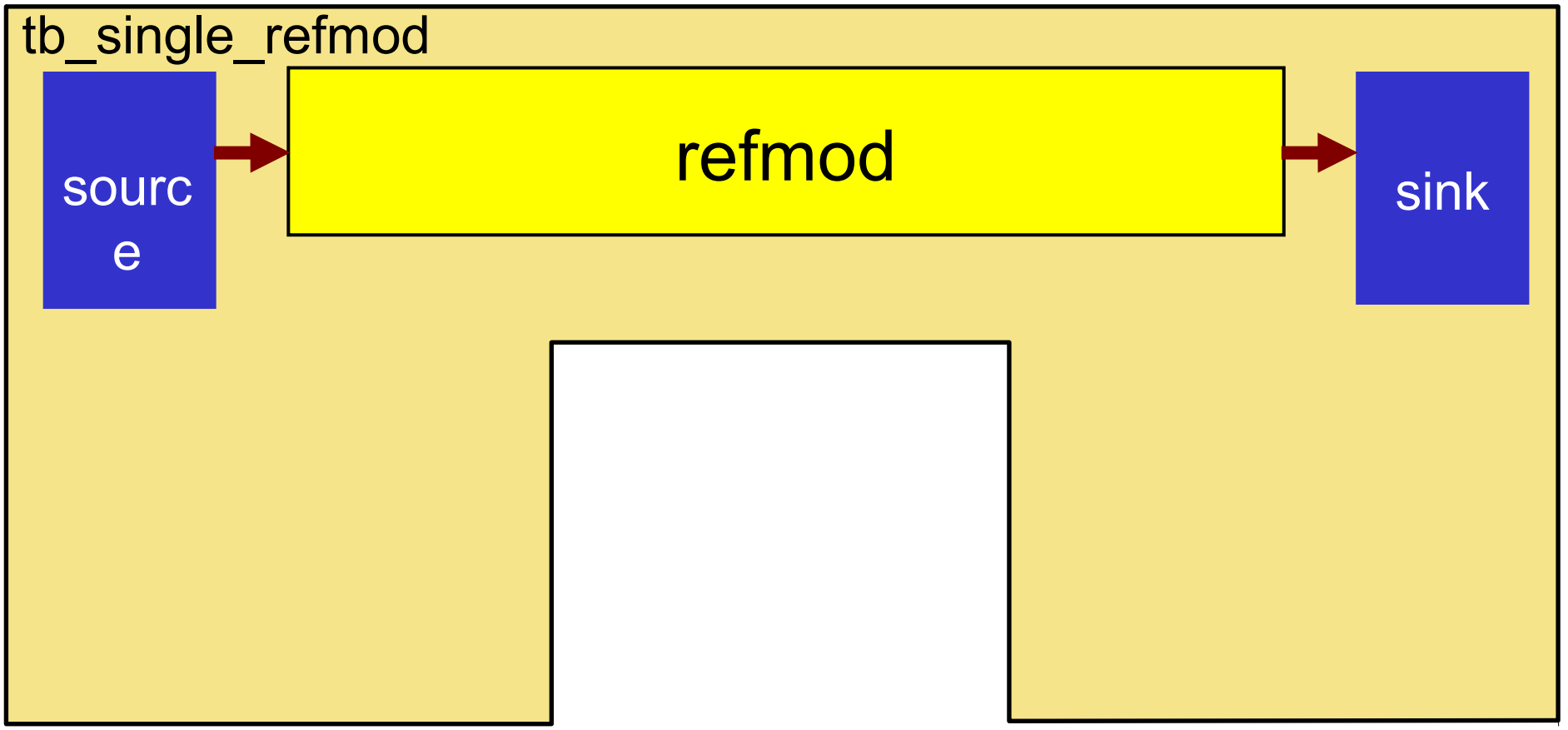
Passo 2: Hierarchical Refmod Decomposition

2.2 Single Hierarchical Refmods



Passo 2: Hierarchical Refmod Decomposition

2.2 Single Hierarchical Refmods – **Templates (eTBc)**



Passo 2: Hierarchical Refmod Decomposition

2.2 Single Hierarchical Refmods – **Templates (eTBc)**

- **source** *
- **refmod**
- **sink** *
- **tb_single_refmod**
- **trans**
- **tb_tcl**
- **Makefile_single_refmod**

* **Um source e um sink podem ser reusados.**

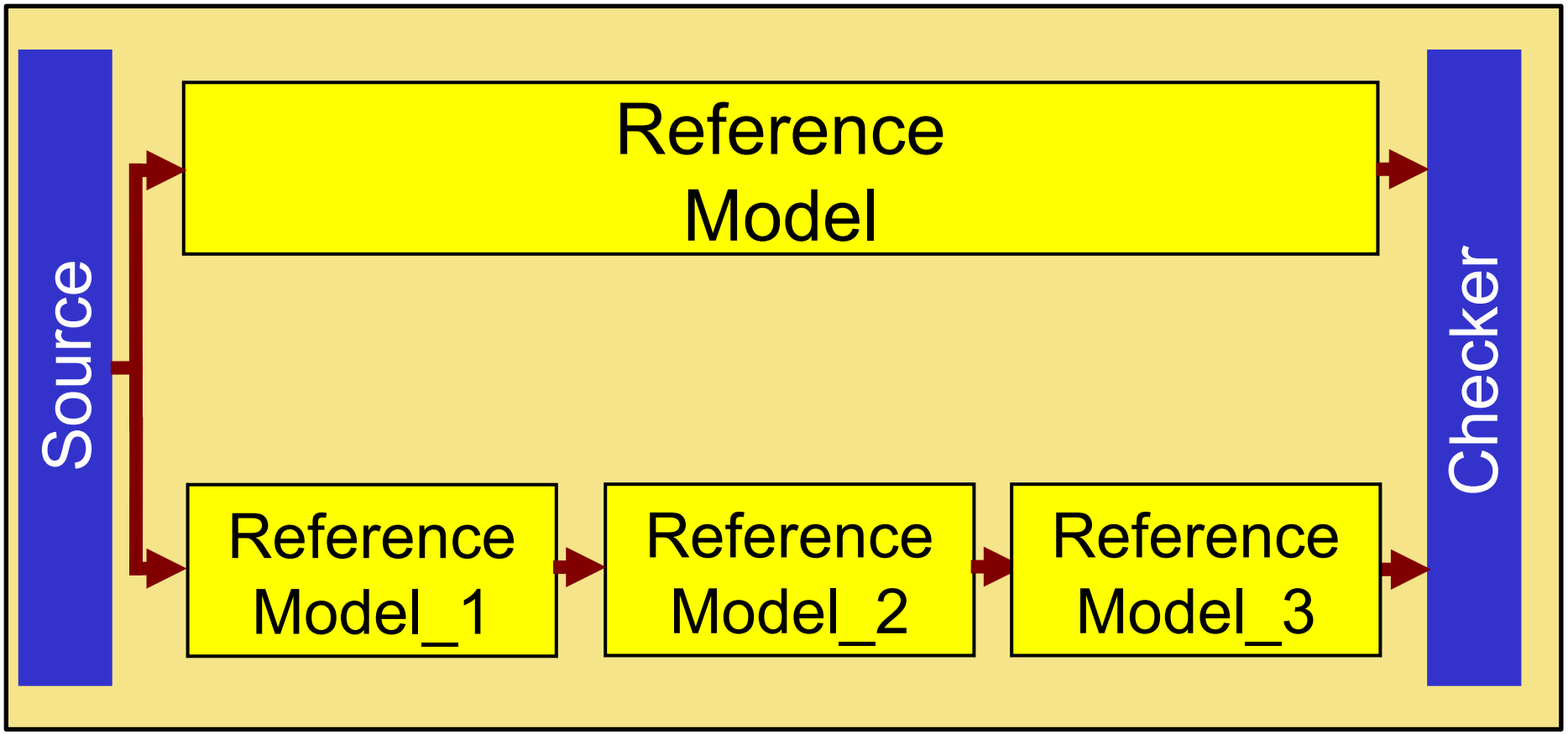
Passo 2: Hierarchical Refmod Decomposition

2.2 Single Hierarchical Refmods

Cada bloco do Reference Model deve ter suas entradas e saída testadas

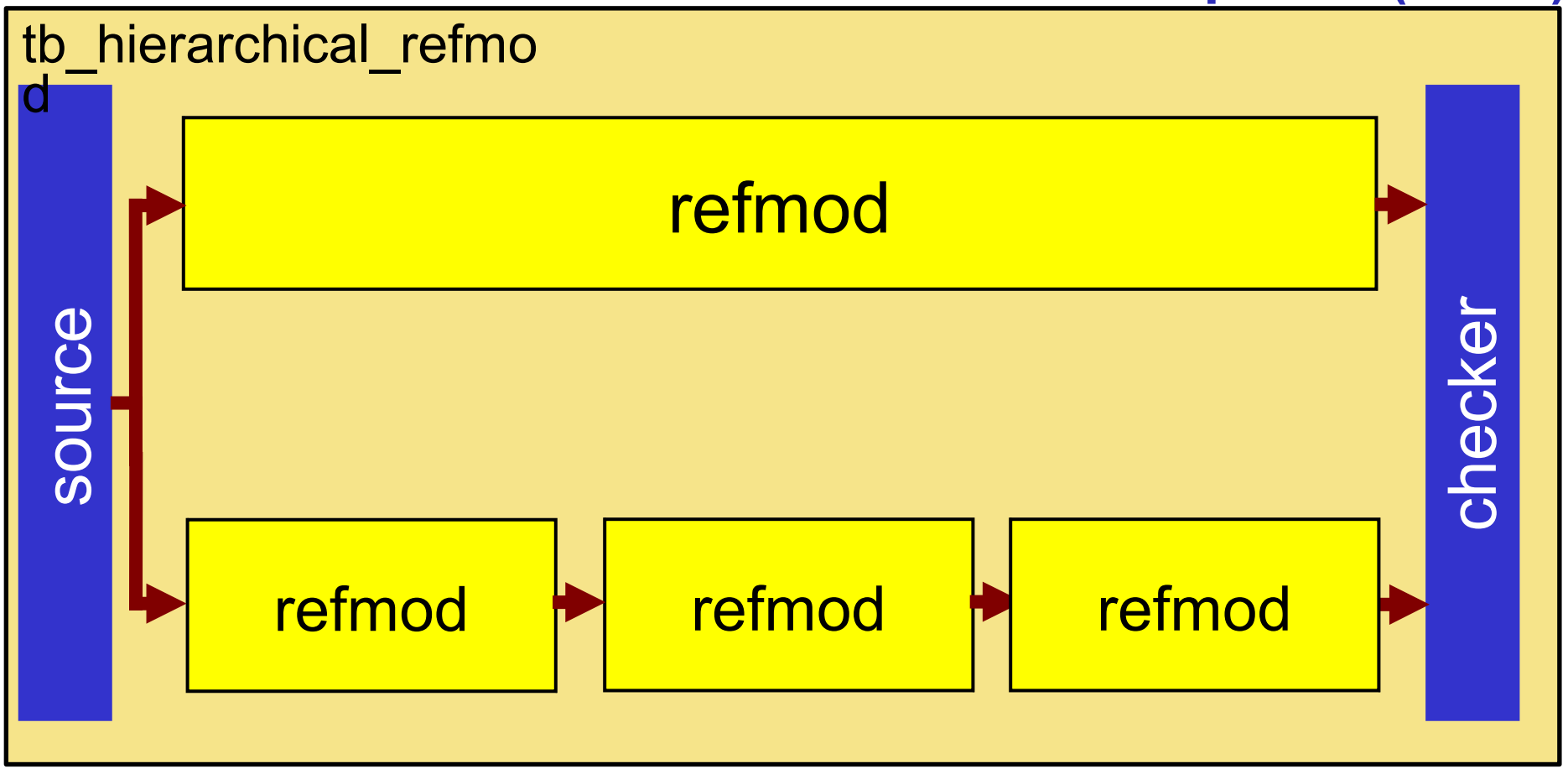
Passo 2: Hierarchical Refmod Decomposition

2.3 Hierarchical Refmods Verification



Passo 2: Hierarchical Refmod Decomposition

2.3 Hierarchical Refmods Verification – **Templates (eTBc)**



Passo 2: Hierarchical Refmod Decomposition

2.3 Hierarchical Refmods Verification – **Templates (eTBc)**

- **source**
- **refmod ***
- **checker**
- **tb_hierarchical_refmod**
- **trans**
- **tb_tcl**
- **Makefile_hierarchical_refmod**

*** Todos os refmods usados nesse passo são reusados.**

Passo 2: Hierarchical Refmod Decomposition

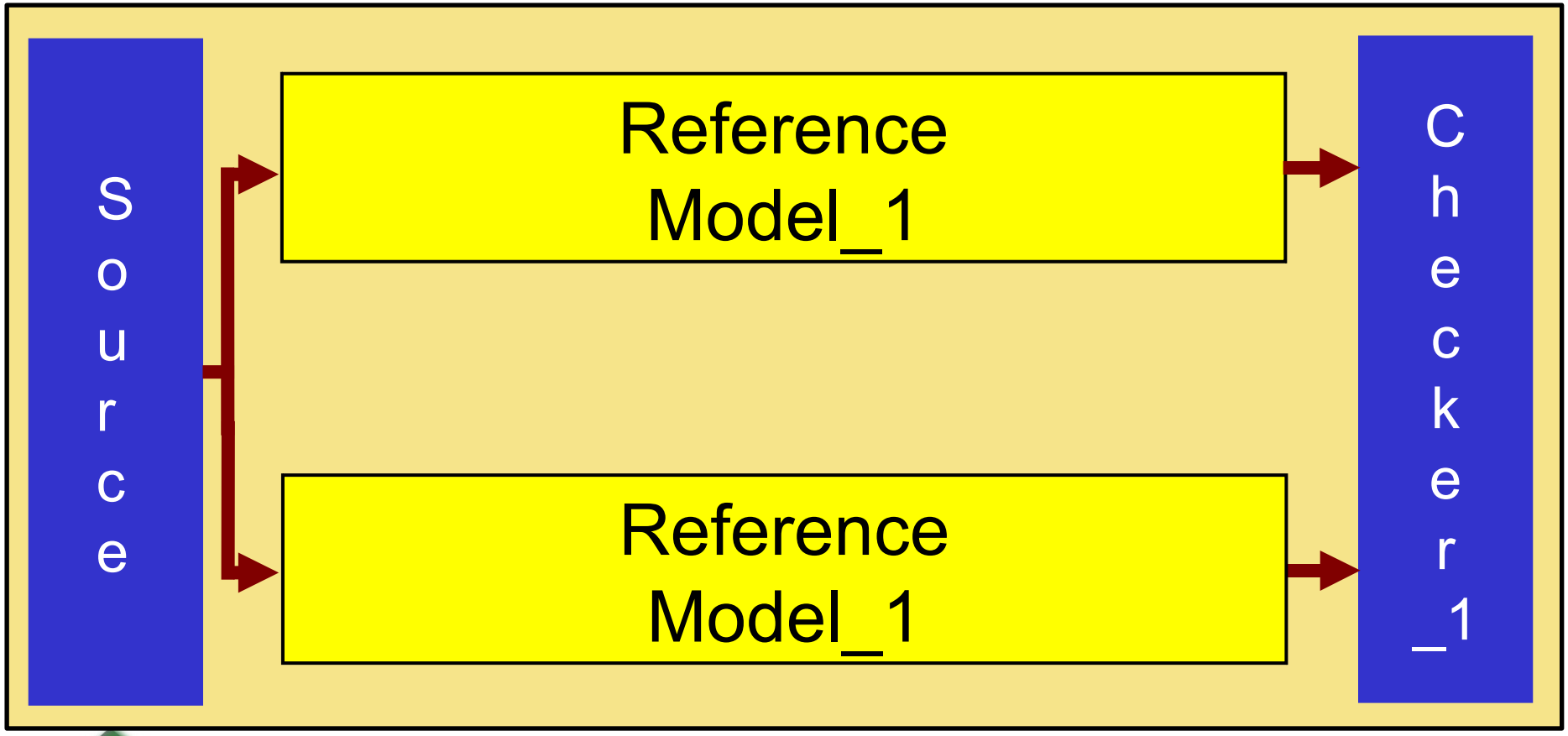
2.3 Hierarchical Refmods Verification

A junção dos reference models hierárquicos deve ser comparada com o Modelo original para ver se ainda é equivalente.

É importante realizar esse passo depois do 2.2 e não inverter a ordem.

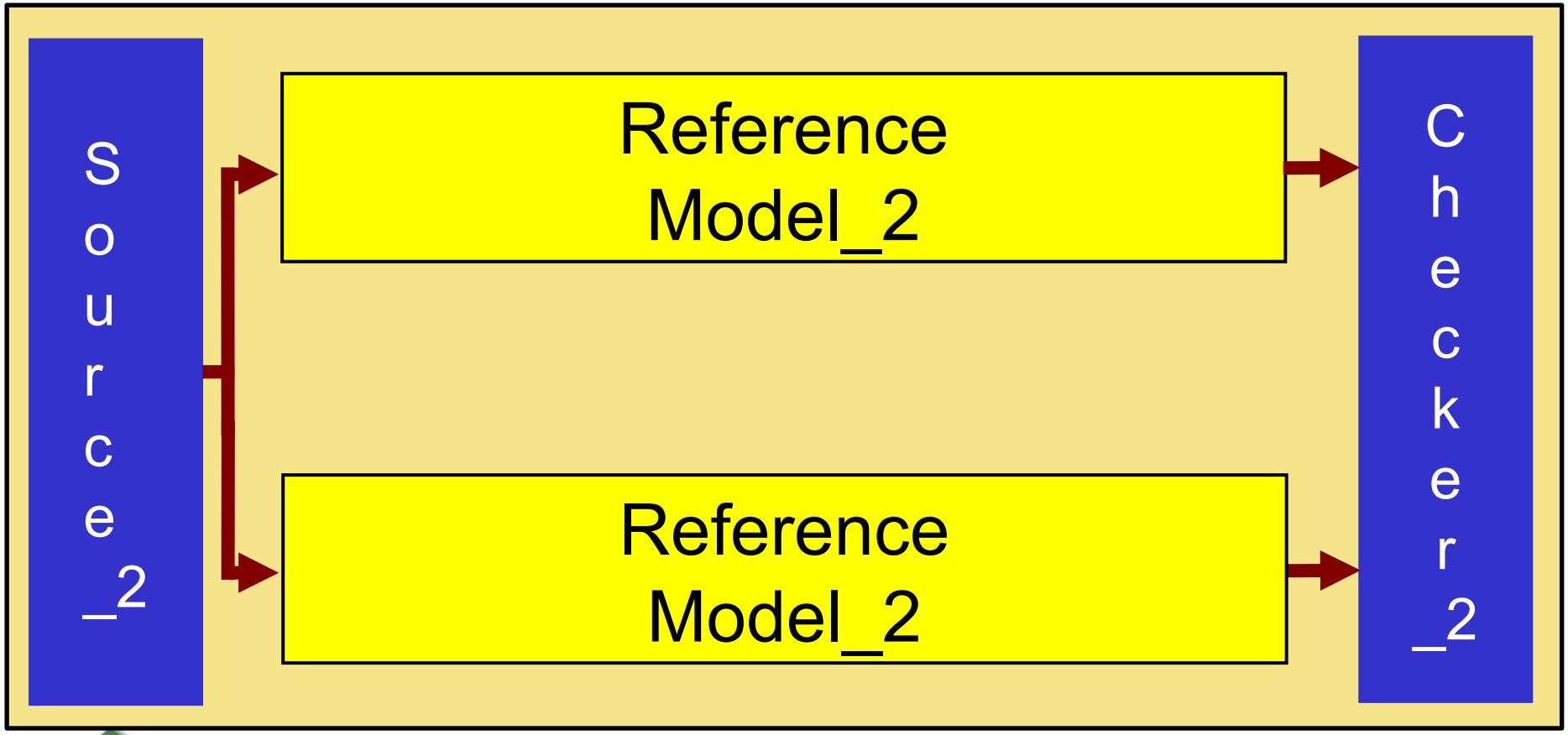
Passo 3: Hierarchical Testbench

3.1 Double Hierarchical Refmods



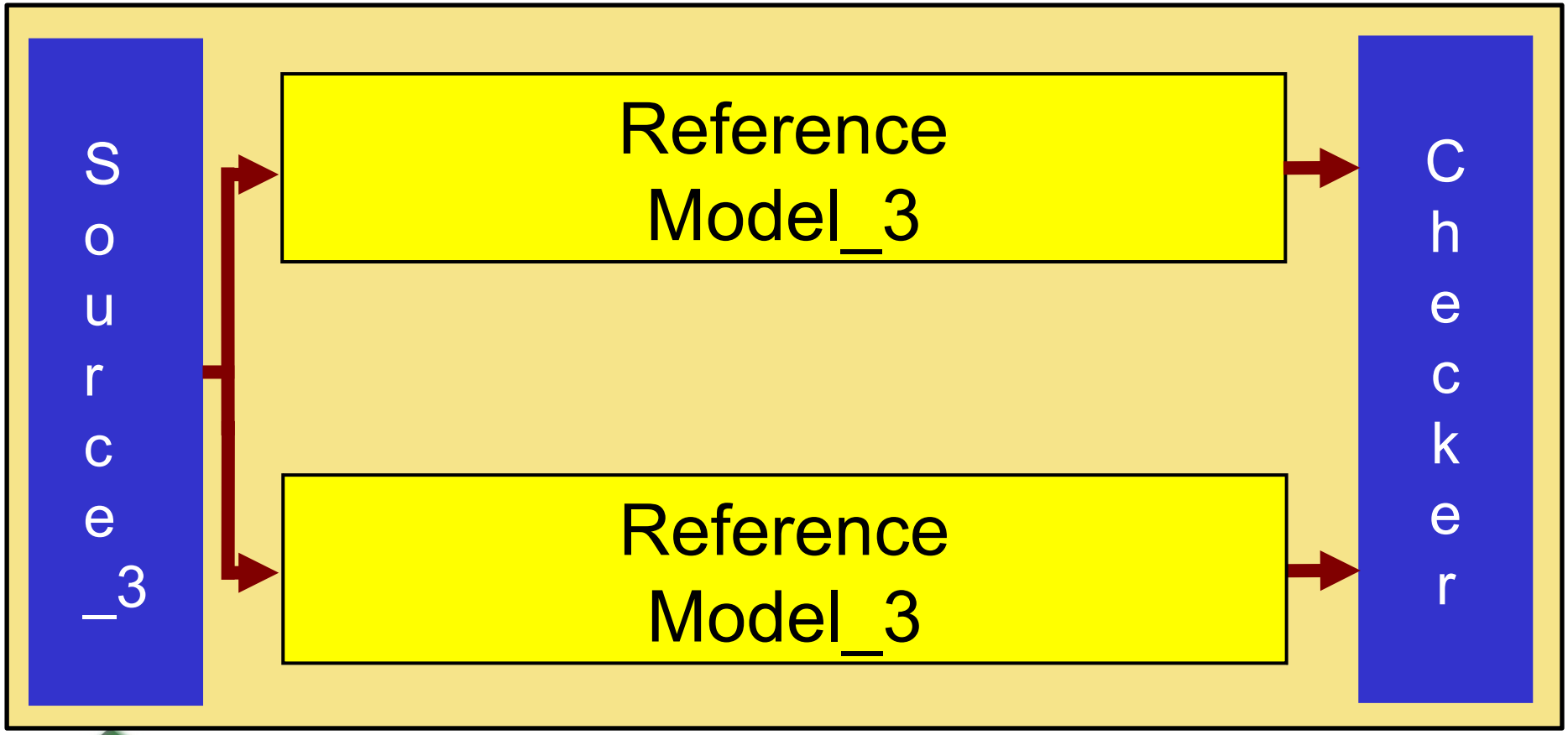
Passo 3: Hierarchical Testbench

3.1 Double Hierarchical Refmods



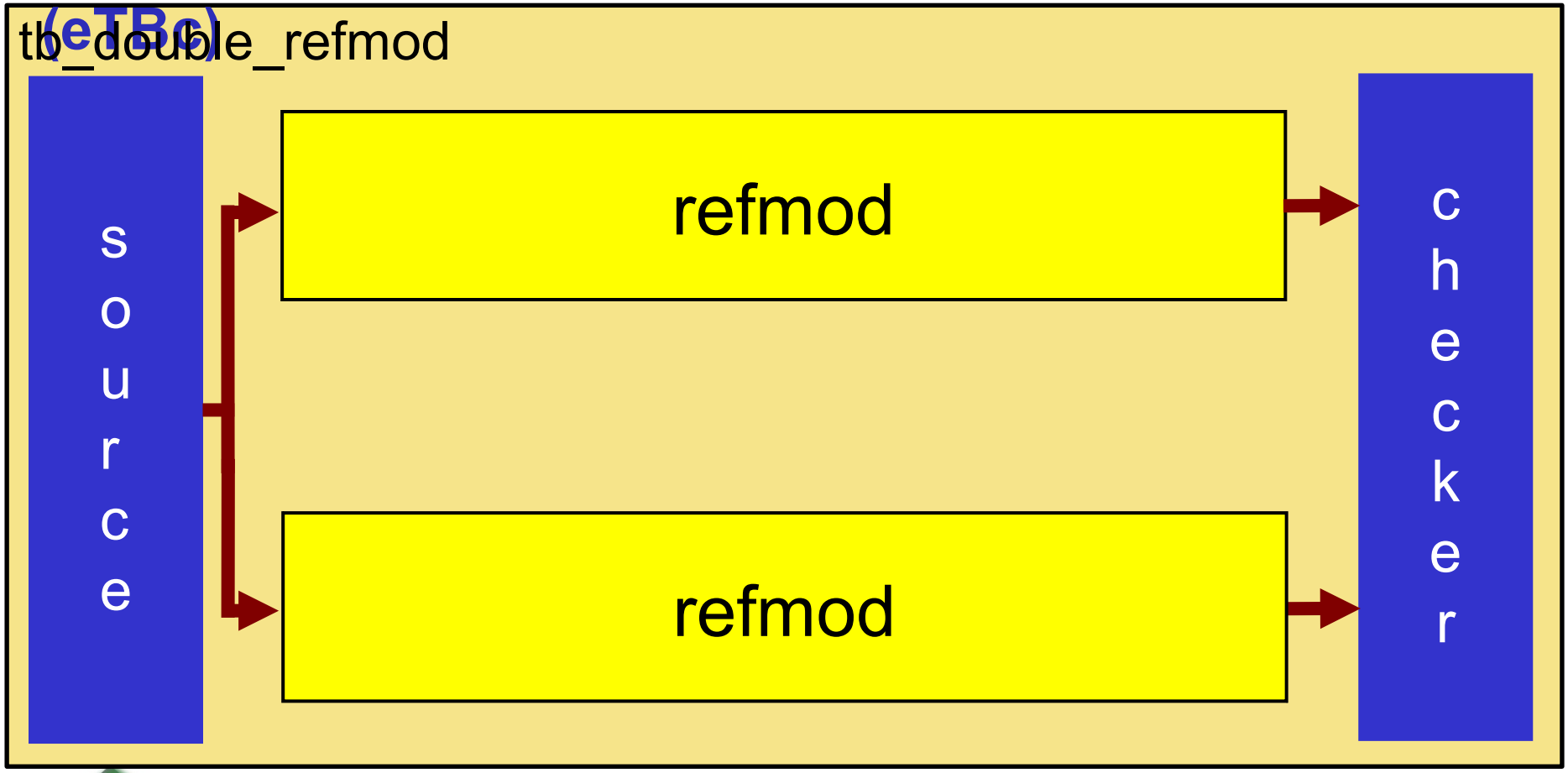
Passo 3: Hierarchical Testbench

3.1 Double Hierarchical Refmods



Passo 3: Hierarchical Testbench

3.1 Double Hierarchical Refmods – **Templates (eTRc)**



Passo 3: Hierarchical Testbench

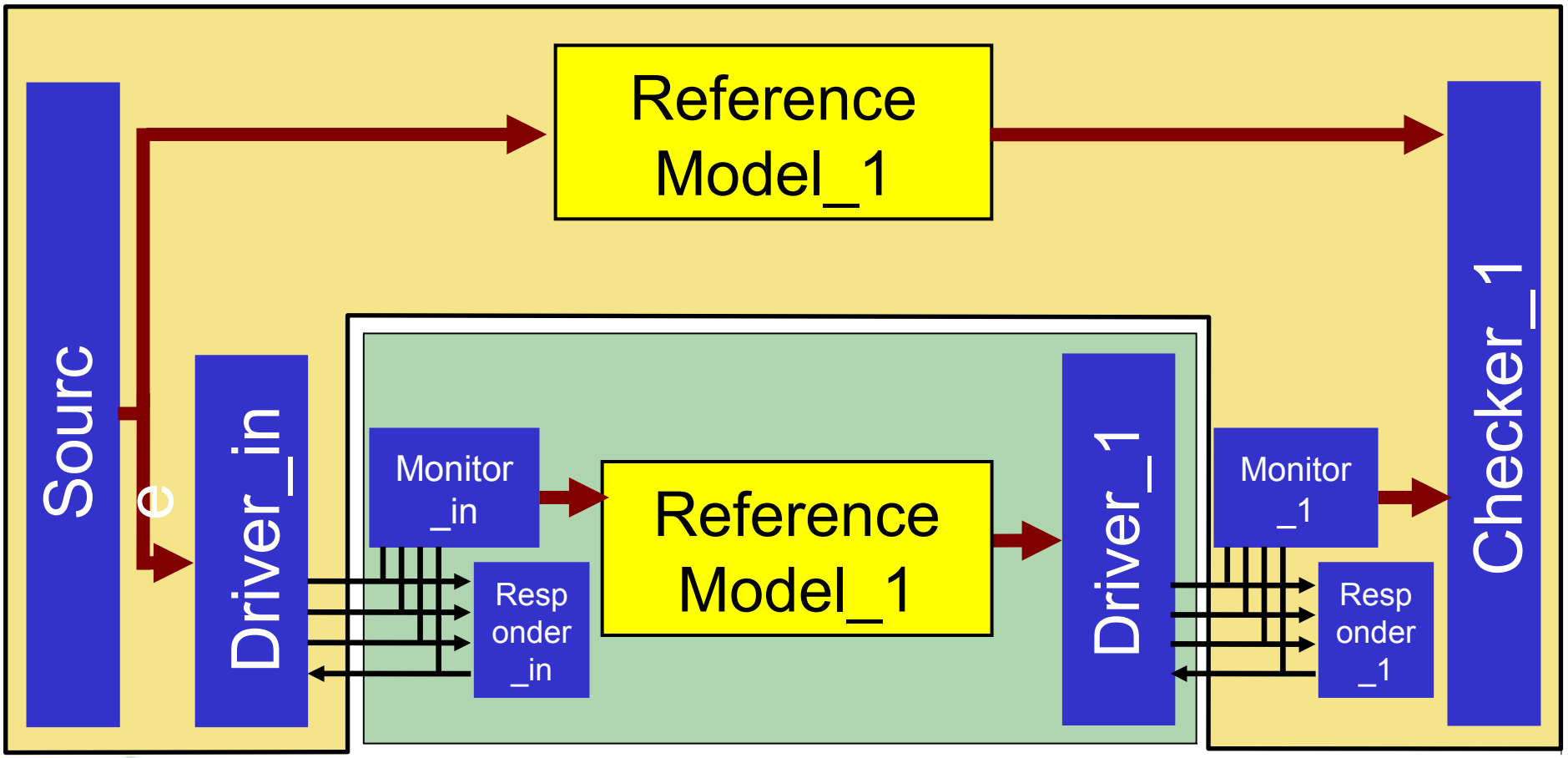
3.1 Double Hierarchical Refmods – **Templates (eTBc)**

- **source** *
- **refmod**
- **checker** *
- **tb_double_refmod**
- **trans**
- **tb_tcl**
- **Makefile_double_refmod**

* **Um source e um checker podem ser reusados.**

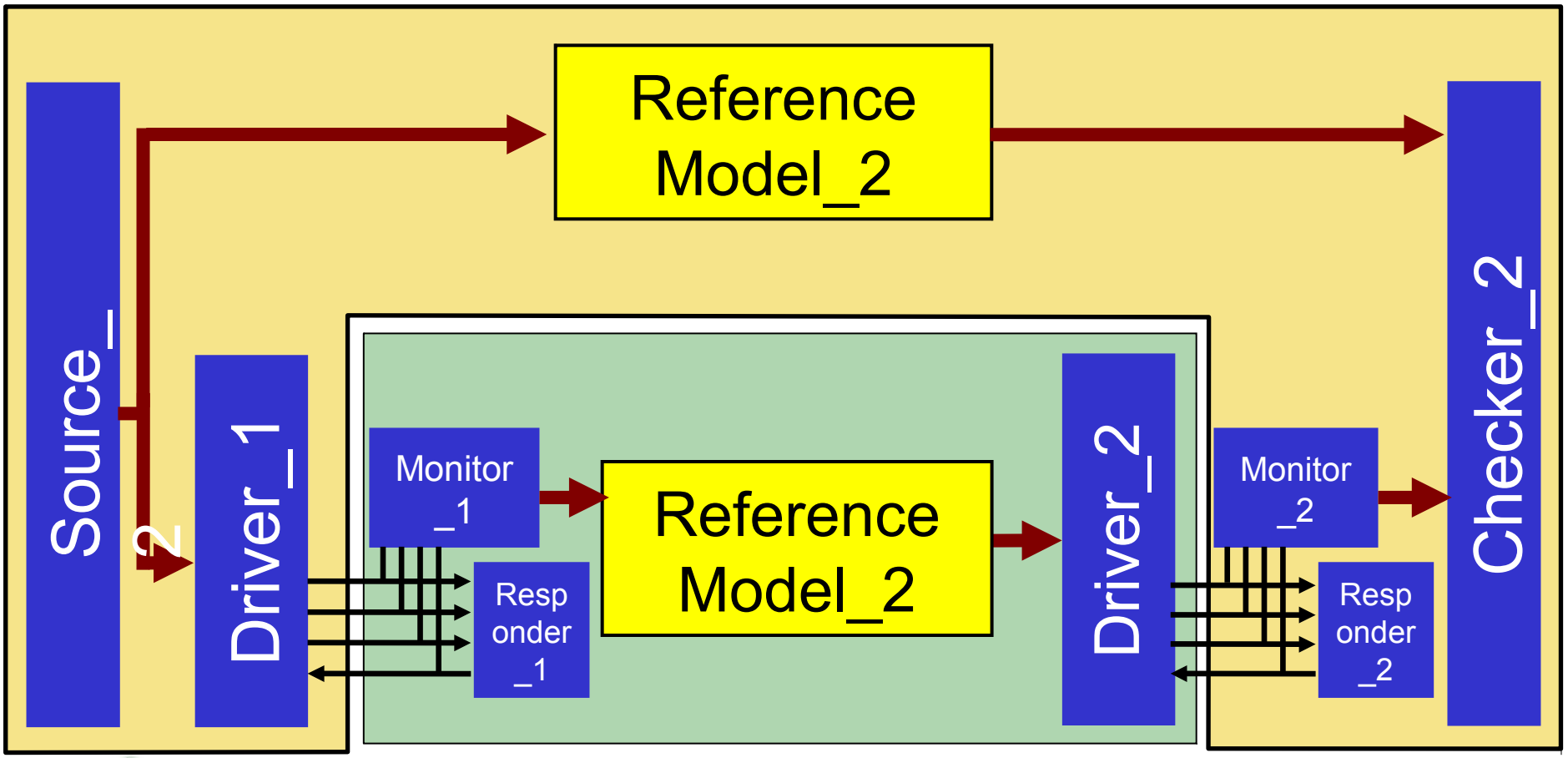
Passo 3: Hierarchical Testbench

3.2 Hierarchical DUV Emulation



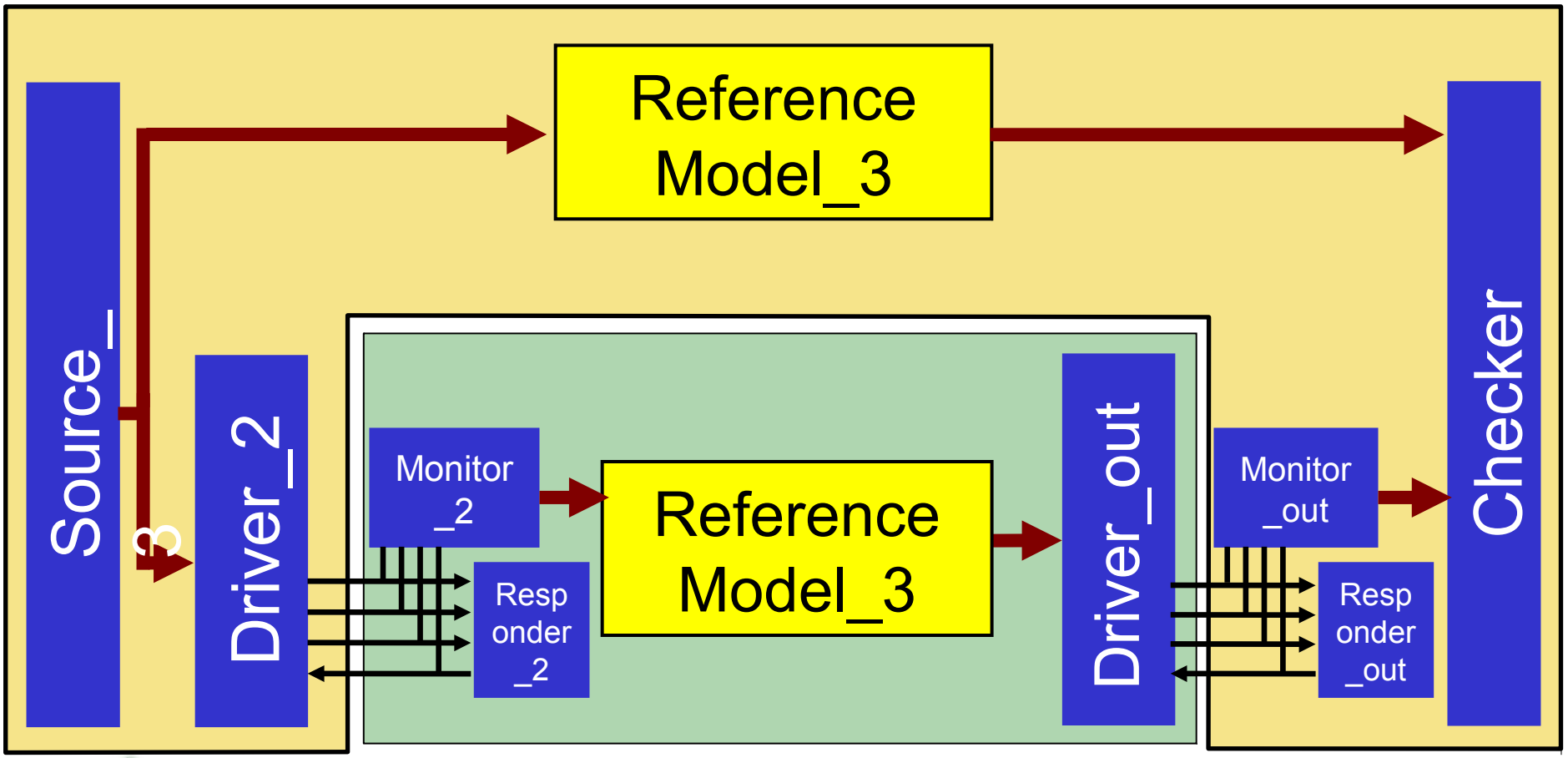
Passo 3: Hierarchical Testbench

3.2 Hierarchical DUV Emulation



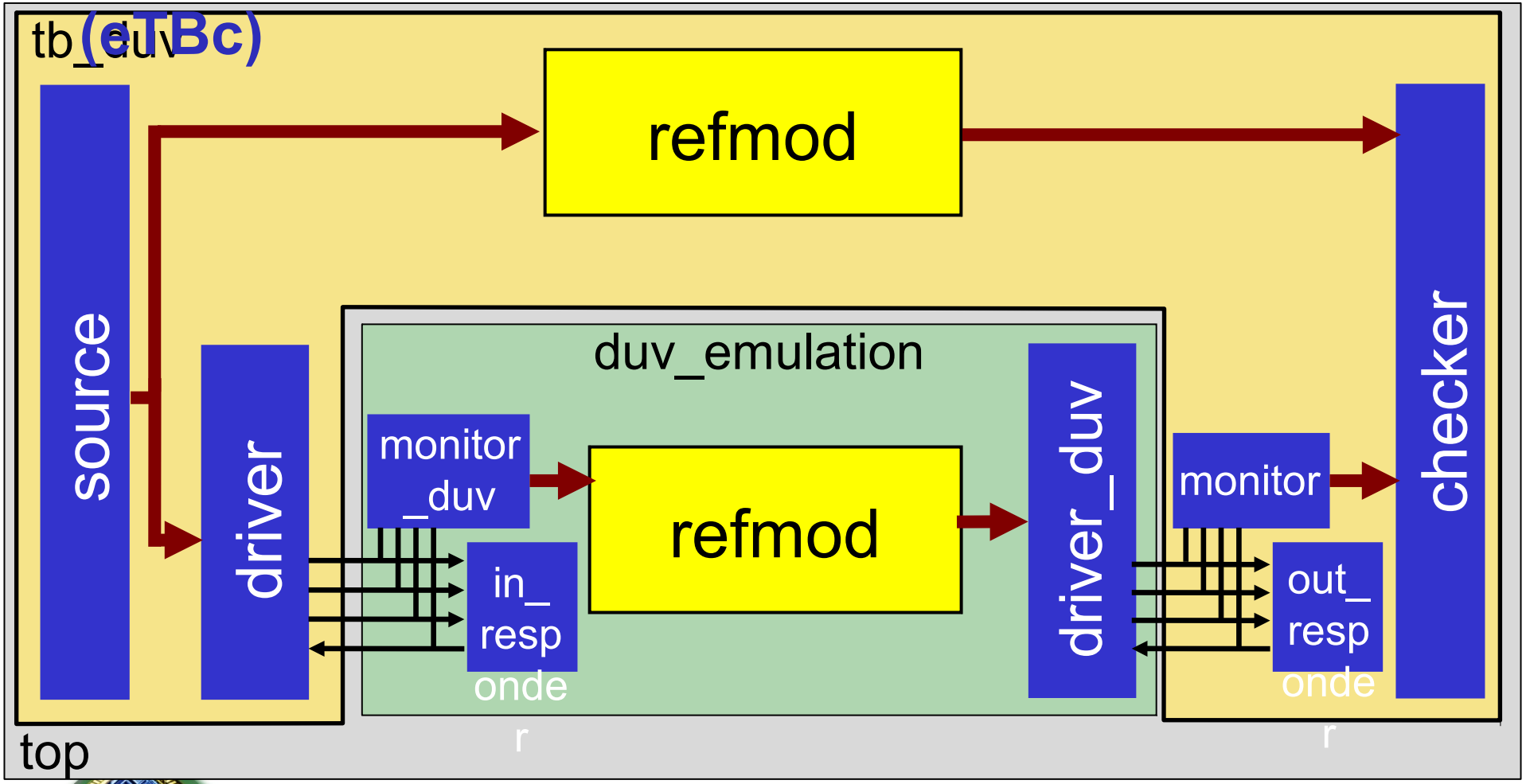
Passo 3: Hierarchical Testbench

3.2 Hierarchical DUV Emulation



Passo 3: Hierarchical Testbench

3.2 Hierarchical DUV Emulation – **Templates**



Passo 3: Hierarchical Testbench

3.2 Hierarchical DUV Emulation – Templates (eTBc)

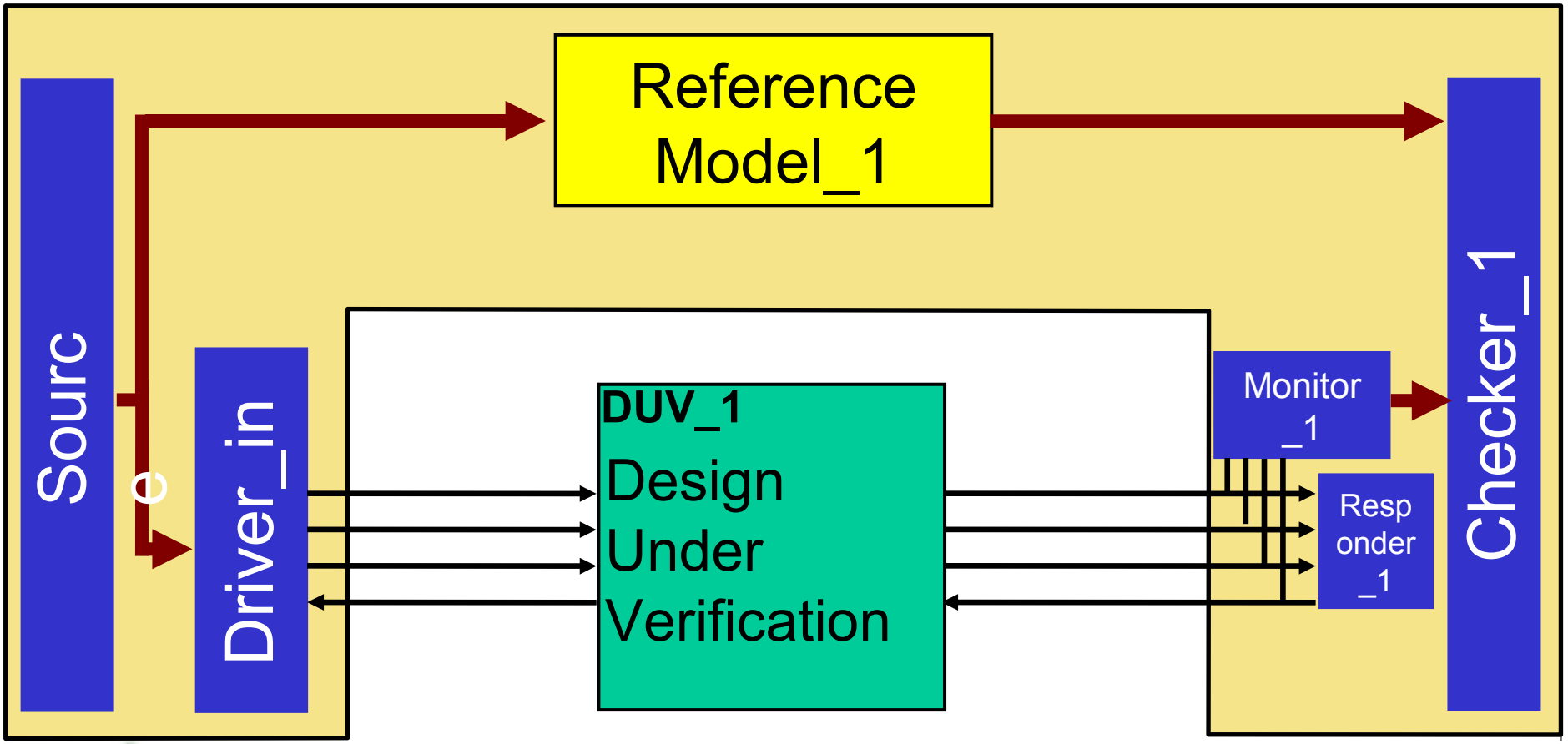
- source
- refmod
- checker
- driver *
- monitor *
- driver_duv *
- monitor_duv *
- in_responder *
- out_responder *

- duv_emulation
- tb_duv
- top
- trans
- top_tcl
- gene_clock
- axi_cover
- Makefile_duv_emulation

channel
sinal

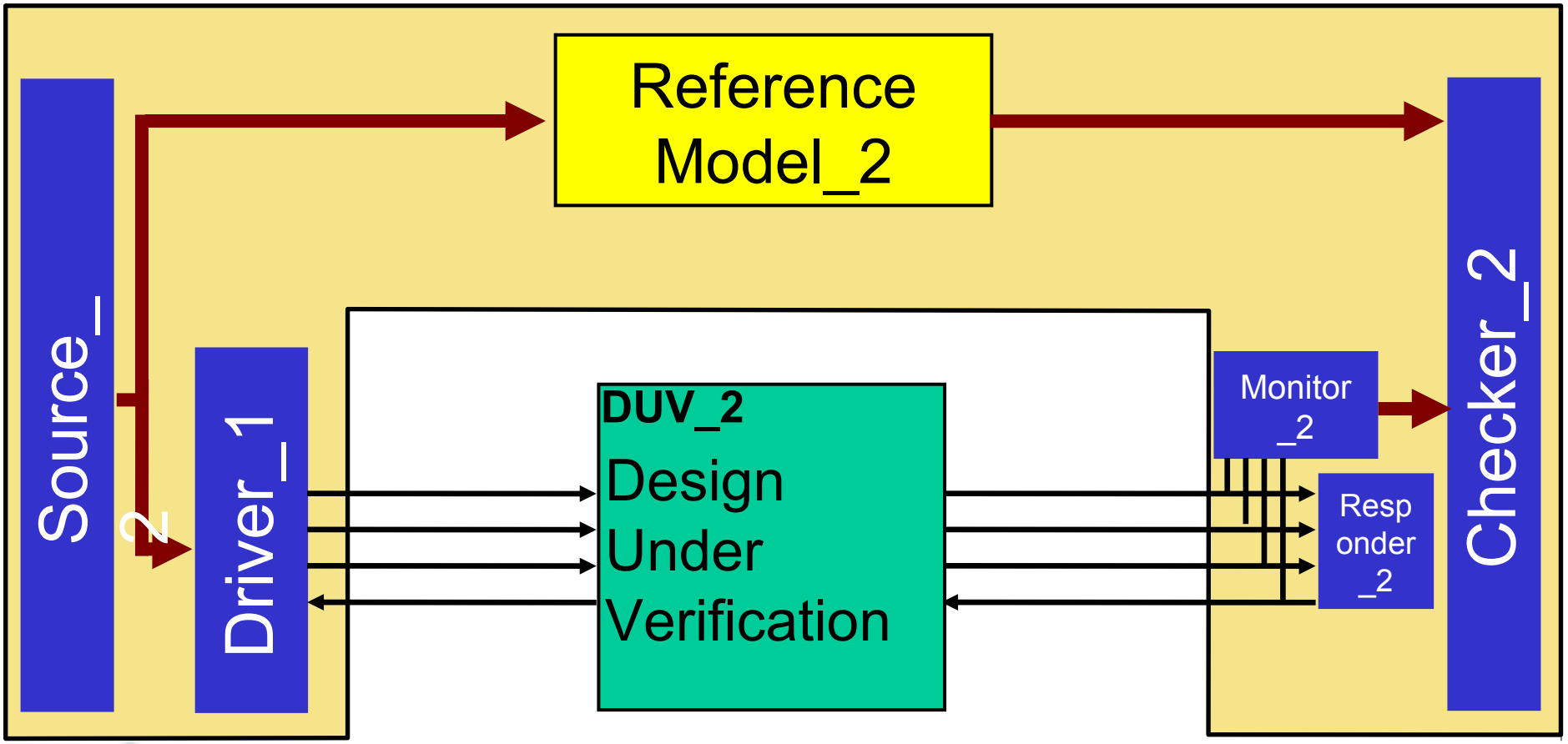
Passo 3: Hierarchical Testbench

3.3 Hierarchical DUV



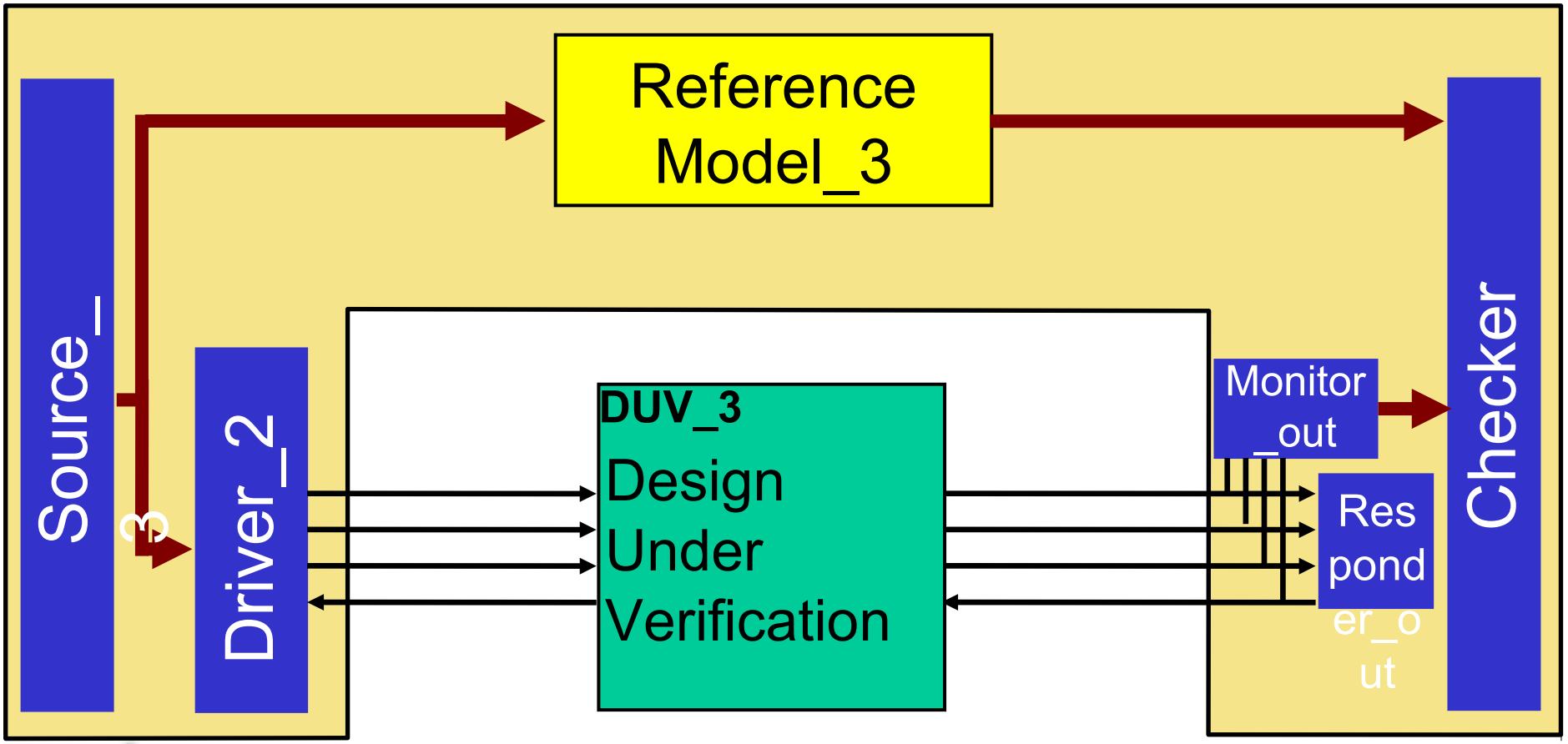
Passo 3: Hierarchical Testbench

3.3 Hierarchical DUV



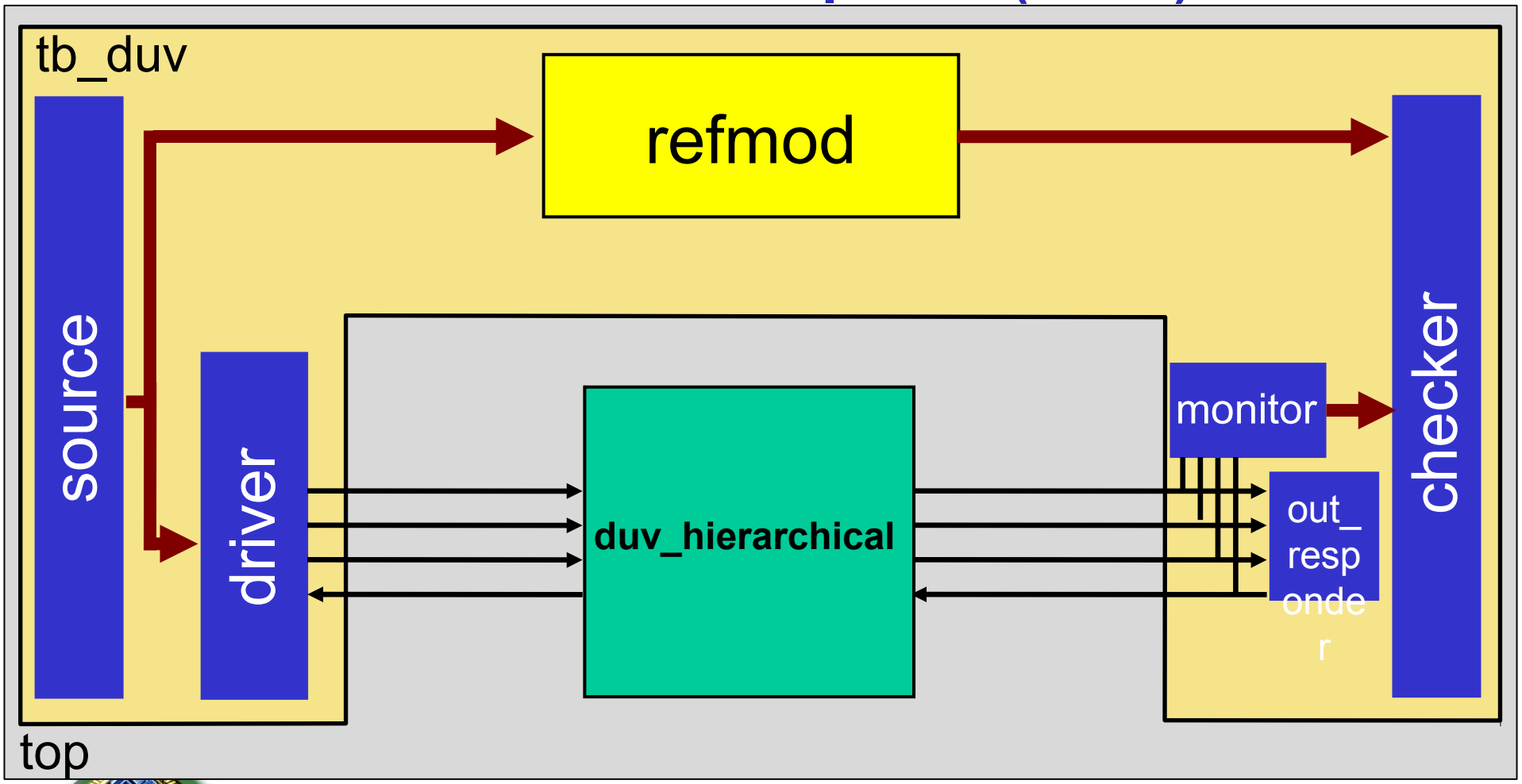
Passo 3: Hierarchical Testbench

3.3 Hierarchical DUV



Passo 3: Hierarchical Testbench

3.3 Hierarchical DUV – **Templates (eTBc)**

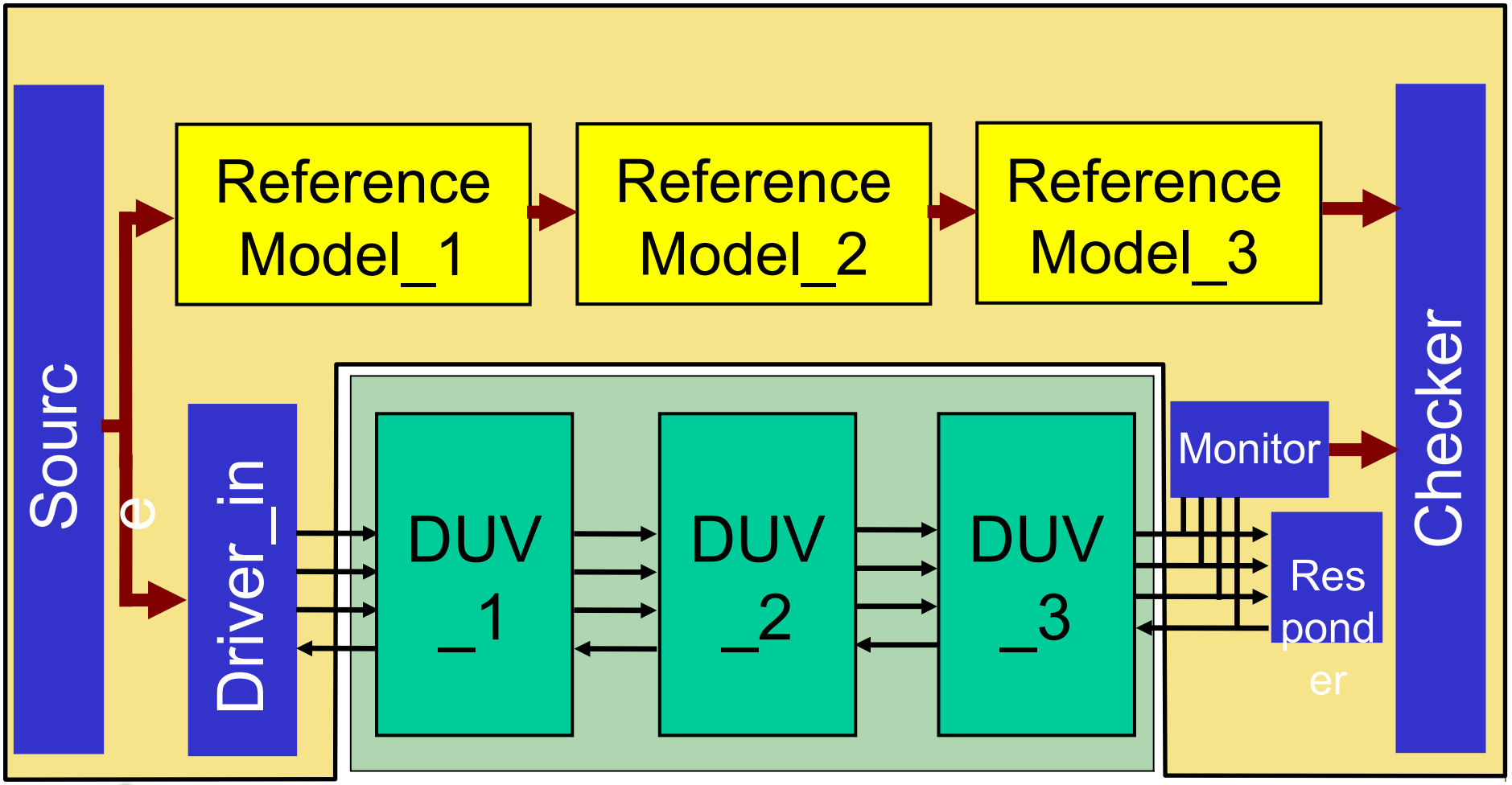


Passo 3: Hierarchical Testbench

3.3 Hierarchical DUV – **Templates (eTBc)**

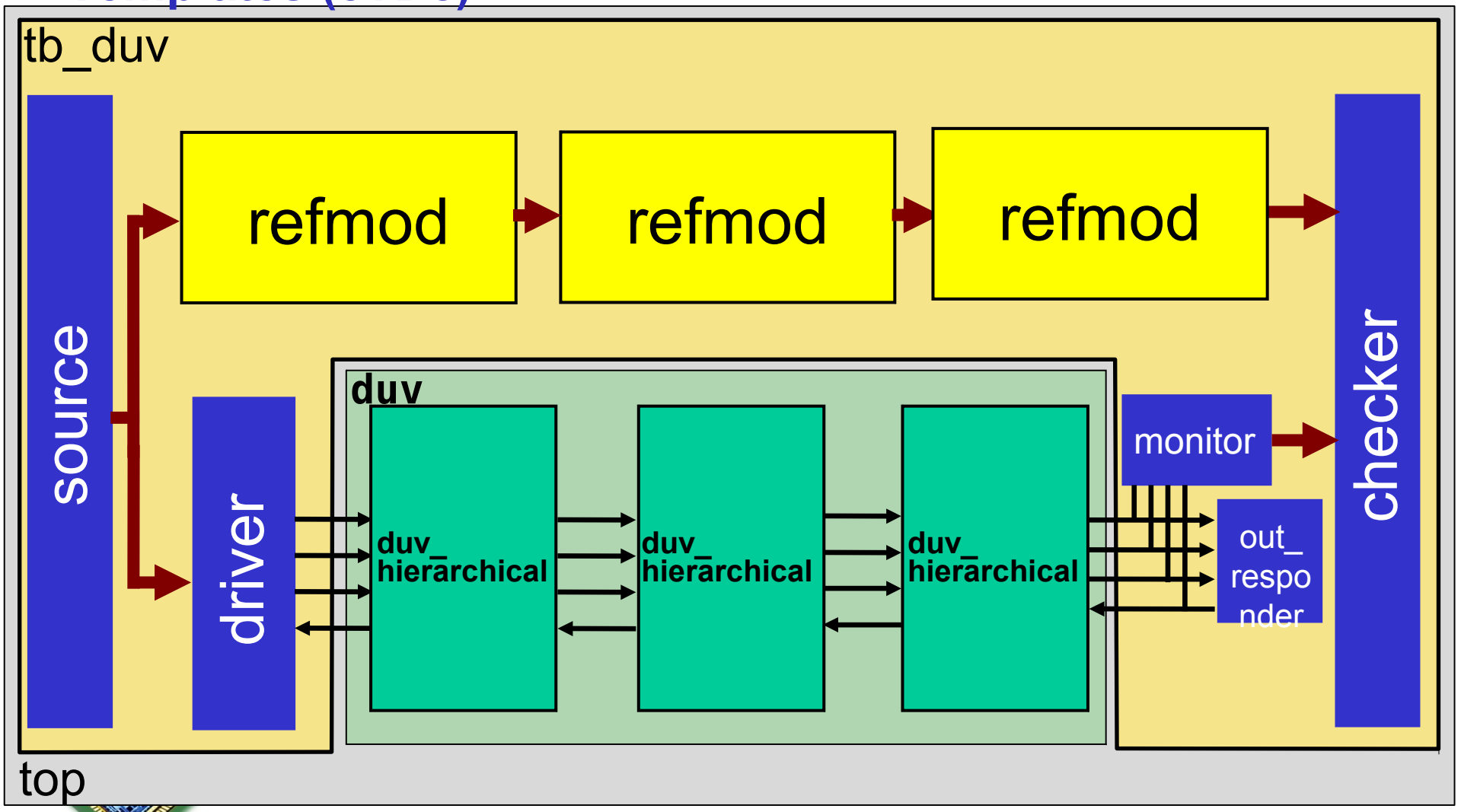
- **source**
- **refmod**
- **checker**
- **driver**
- **monitor**
- **out_responder**
- **duv_hierarchical**
- **tb_duv**
- **top**
- **trans**
- **top_tcl**
- **gene_clock**
  - **axi_cover**
- **Makefile_duv**

Passo 4: Full Testbench



Passo 4: Full Testbench

Templates (eTBc)

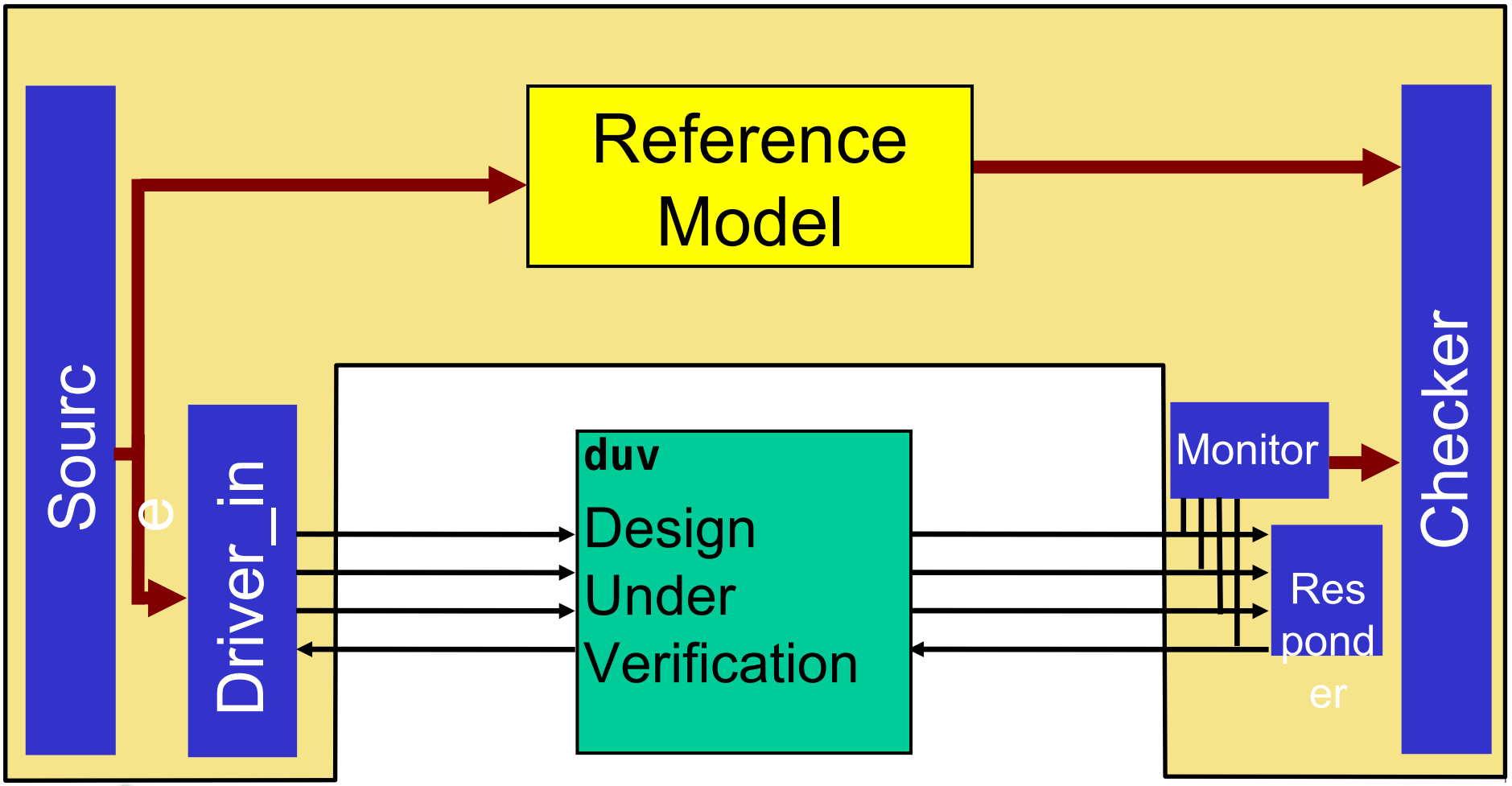


Passo 4: Full Testbench

4 Full Testbench – **Templates (eTBc)**

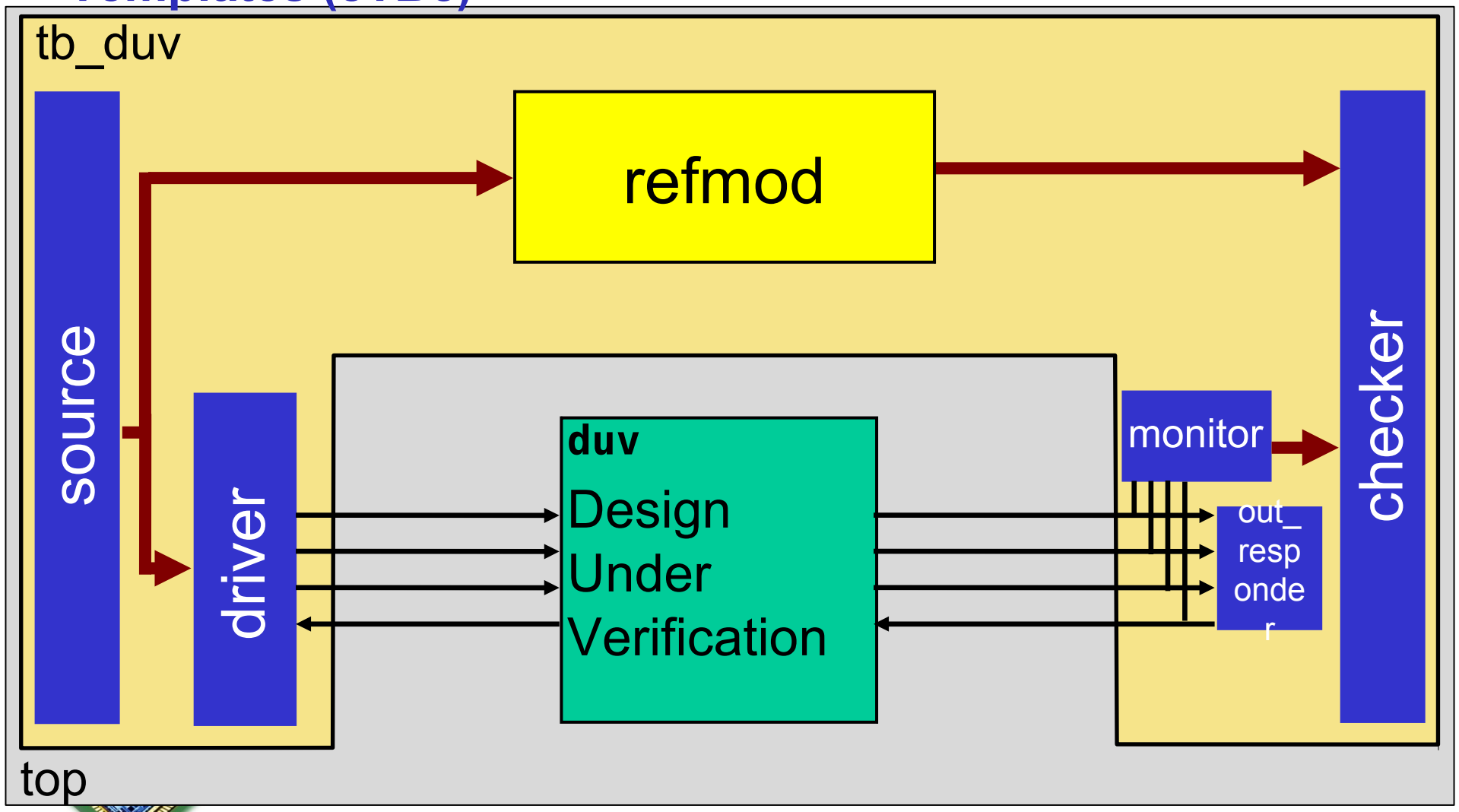
- source
- refmod
- checker
- driver
- monitor
- out_responder
- duv_hierarchical
- duv
- tb_duv
- top
- trans
- top_tcl
- gene_clock
  - axi_cover
- Makefile_duv

Passo 4: Full Testbench



Passo 4: Full Testbench

Templates (eTBc)



Metodologia BVM

Testbench passa pela fase de depuração composta de passos, reduzindo a quantidade de erros.

Muito reuso de código, diminuindo tempo de implementação do testbench.

Resumo

Resumo

Source em SystemVerilog

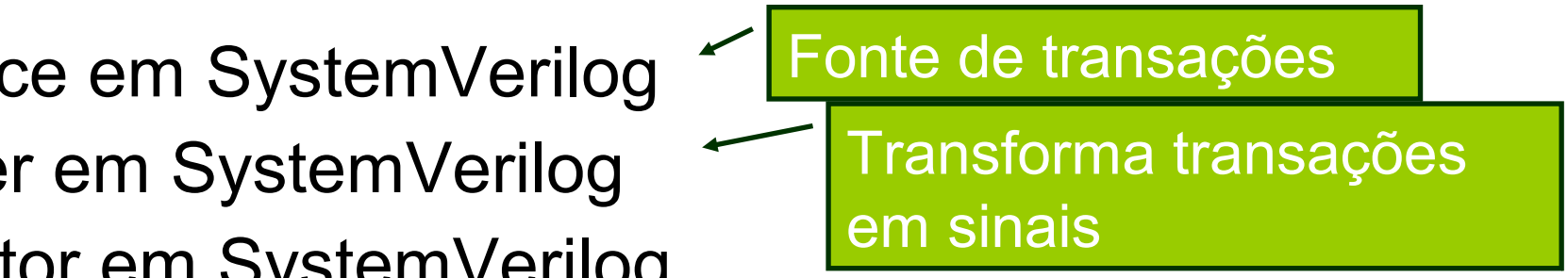
Driver em SystemVerilog

Monitor em SystemVerilog

Modelo de referência em SystemVerilog

Checker em SystemVerilog

Testbench em SystemVerilog



Resumo

Source em SystemVerilog

Driver em SystemVerilog

Monitor em SystemVerilog

Modelo de referência em SystemVerilog

Checker em SystemV

Testbench em SystemVerilog

Resumo

Transações em SystemVerilog

Source em SystemVerilog

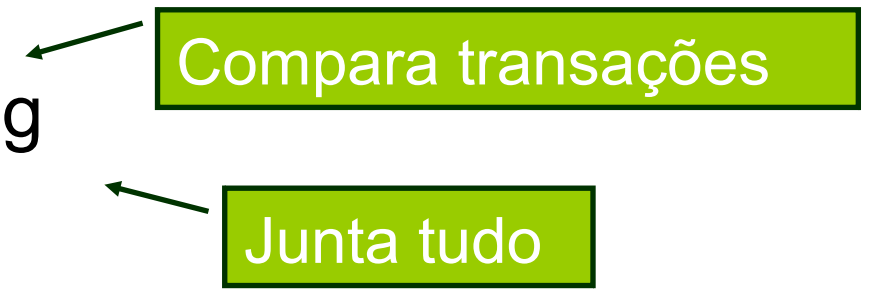
Driver em SystemVerilog

Monitor em SystemVerilog

Modelo de referência em SystemVerilog

Checker em SystemVerilog

Testbench em SystemVerilog



A Arte da verificação

Estou exercitando todos os possíveis cenários de entrada?

Como vou saber se ocorreu um erro?

Ideal x Real

Gerar todas as possíveis combinações de entrada tem custo inviável para unidades maiores.

Testar muitas unidades pequenas também tem custo inviável.

É necessário fazer uma escolha.

Tipos de Estímulos

Compliance Testing

Verificar situações mencionadas na especificação

Corner Cases

Verificar situações críticas (extremas) do projeto

Real Code

Utilizar estímulos reais da aplicação

Random

Cria situações “inusitadas”

Cobertura tipicamente melhor do que os outros tipos porque pode gerar cenários que seriam esquecidos.

A Arte da verificação

Forma mais óbvia: inspeção visual de formas de onda:

muito usado,

impossível de reaproveitar,

passível a mal interpretação (erro humano),

impraticável para muitas transações.

Use a visualização de formas de onda somente para depuração.

A Arte da verificação

Sniffers

Possuem interface dedicada para monitorar e extrair estatísticas dos sinais de um módulo.

Muito usado na depuração de sinais.

A Arte da verificação

Self Checking Testbench

Driver, Monitor e Responder implementados em código do simulador (SystemVerilog);

Checker implementado em SystemVerilog compara dados recebidos do Monitor com dados esperados;

Implementação do checker

- A simulação deve rodar quieta enquanto tudo estiver o.k. para não afogar mensagens de erro no meio de outras mensagens.

- Quando aparece um erro, devem ser fornecidas todas as informações disponíveis sobre o problema para facilitar a depuração.

- Inserir erro em um dos refmods, nos passos 1.2 e 3.1, para ver se o Checker funciona.

Implementação do Reference Model

Refmod implementado em código

SystemVerilog, SystemC, C++, C, Pascal, Java etc.

Checker compara diretamente saída do Monitor com a saída do refmod.

Refmod fácil de escrever.

Basic, Matlab, etc.

Problema de comunicação com SystemC, pode usar IPC ou arquivos.

Refmod ainda mais fácil de escrever.

Implementação do Reference Model

Refmod implementado em código (continuação)

possibilita verificação pseudo-aleatório com número grande de amostras (muitos vetores de entrada),

deixar rodar a noite,

dormir tranqüilo !

Cobertura

Ao acordar de manhã ainda está rodando...

Quando vou parar ?

Quando atingir a cobertura indicada no plano de verificação

Tipos de cobertura:

de código (block, expression, toggle, ...)

Funcional

FSM

Cobertura de código

Cobertura de linhas (block)

Mostra quantas vezes uma linhas de código (um comando) foi executada.

Também chamada cobertura de blocos ou cobertura de segmentos.

Cobertura de chaveamento (toggle)

Conta quantas vezes cada sinal mudou de nível lógico.

Cobertura de estados de FSM

Conta quantas vezes se chegou em cada estado

quais transições entre estados foram simuladas.

Cobertura de código

Cobertura de eventos

Verifica se todos os eventos na lista de sensibilidade de todos processos foram exercitadas.

Cobertura de desvios

resultado semelhante ao cobertura de blocos

Cobertura de expressões (expression)

verifica se todas as combinações de expressões lógicas foram exercitadas

Cobertura de caminhos

verifica passagem variada por seqüência de comandos if...else

Cobertura funcional

É mais importante do que a cobertura de código.

Observa sinais e transações durante a simulação.

Conta ocorrências de determinadas situações, por exemplo: “quantas vezes o valor da saída fica entre 0 e 8”

Cobertura funcional

Cobertura desejada é especificada no Plano de Verificação (“a tal situação deve ser verificada 100 vezes”)

Quando a cobertura desejada é atingida a simulação é encerrada

Cobertura funcional

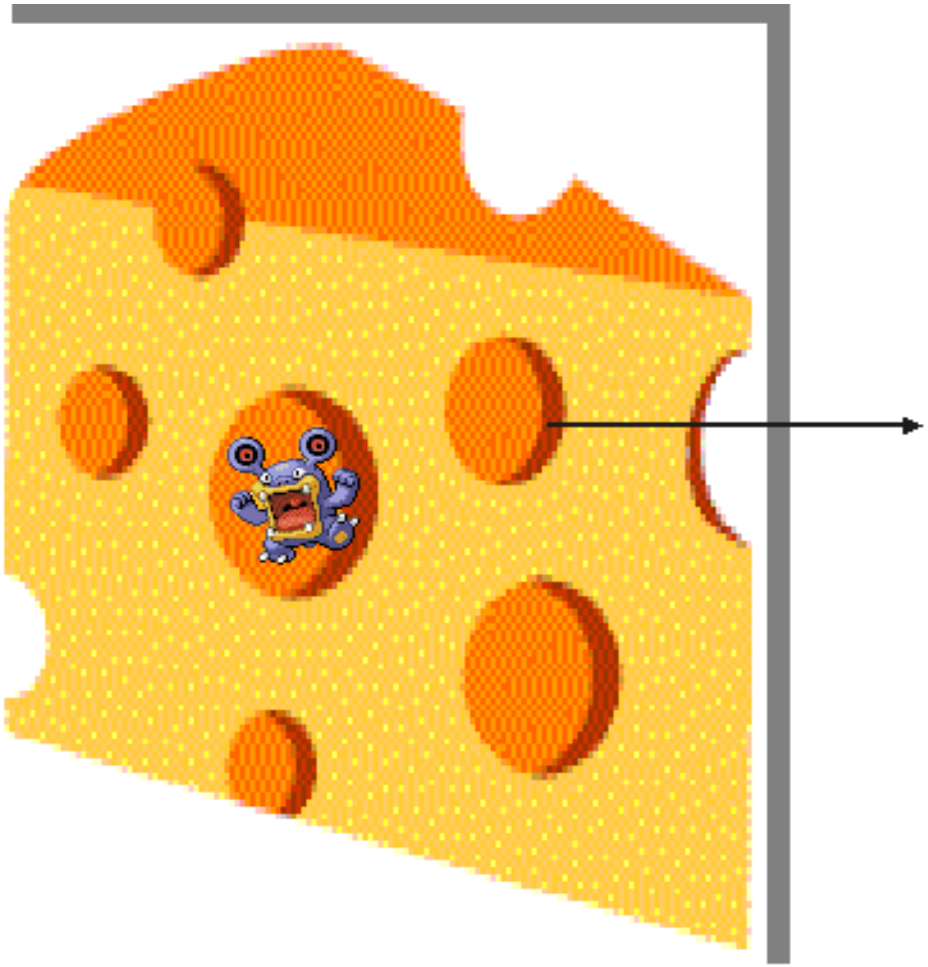
“Cobertura é responsável por medir o progresso da verificação através de várias métricas pré-estabelecidas e ajudar o engenheiro a se localizar com relação ao término da verificação” [piziali2004].

Cobertura funcional

Mede o progresso da simulação e reporta quais funcionalidades não foram exercitadas ou foram exercitadas mais de uma vez.

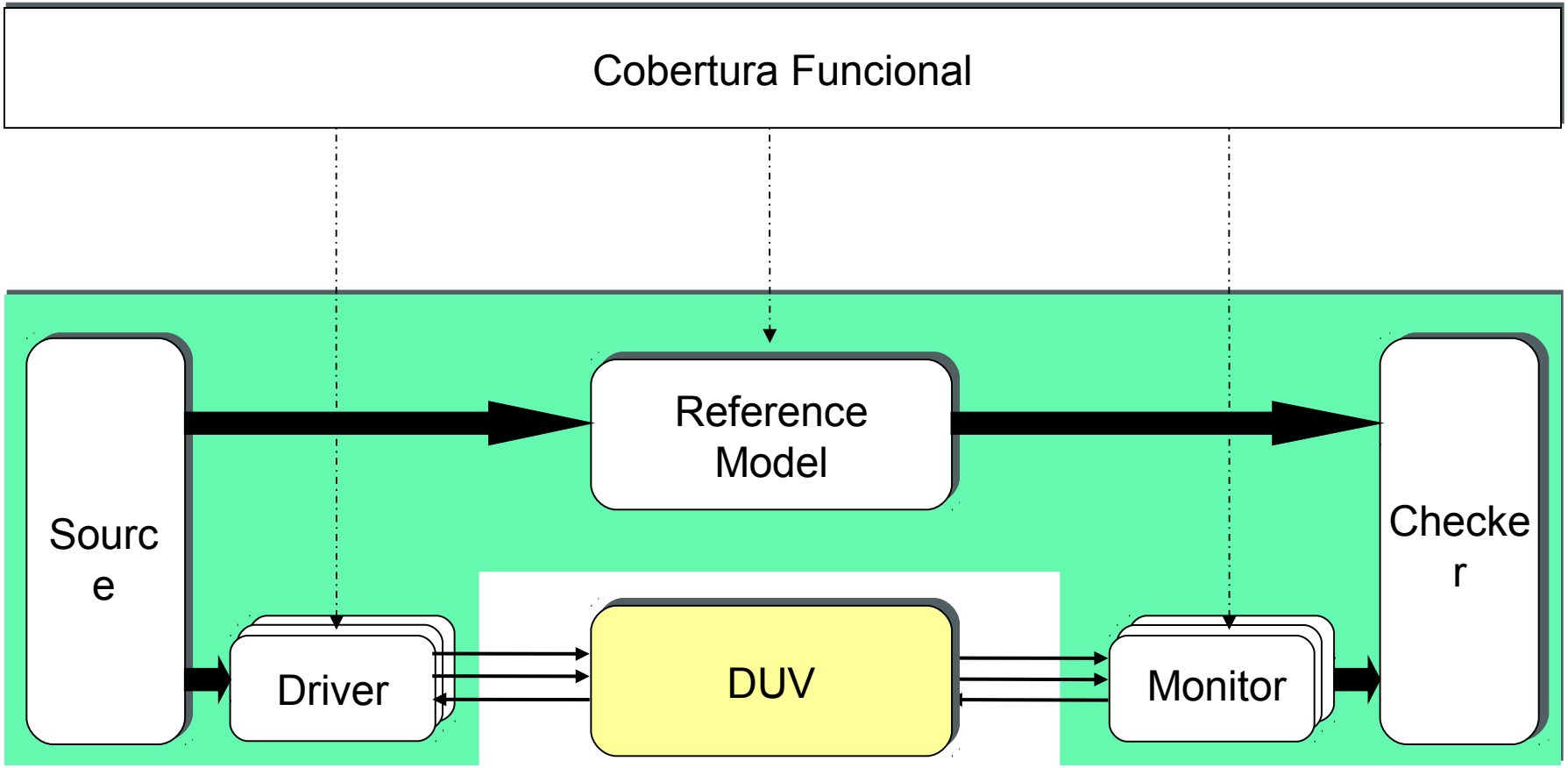
Pode ajudar a inspecionar a qualidade da verificação e direcionar os estímulos de forma a alcançar as funcionalidades não cobertas (**buracos de cobertura**).

Cobertura funcional



Buracos de Cobertura

Onde Fazer Cobertura Funcional?



Cobertura Funcional

Consiste em medir o número de vezes que uma variável recebe um determinado valor durante uma simulação:

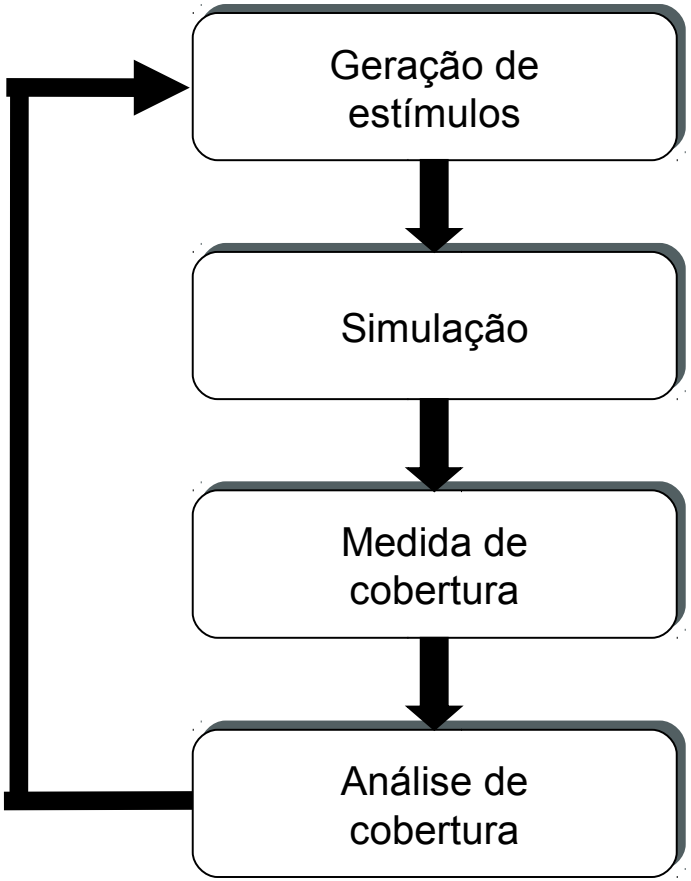
É fácil de interpretar;

Ajuda a identificar valores ou intervalos não testados;

Quando a cobertura desejada é atingida a simulação é encerrada.

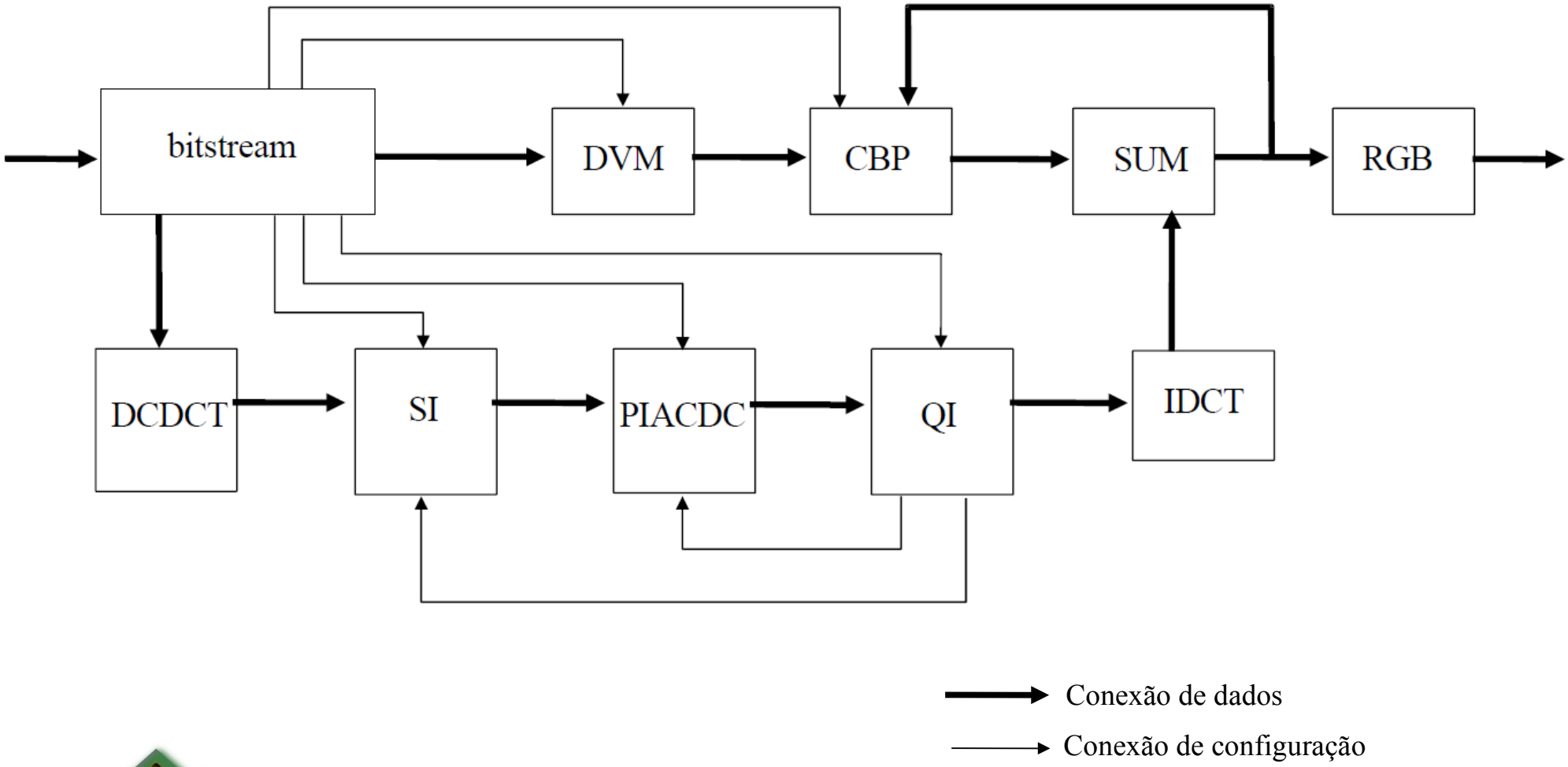
Porém, a complexidade aumenta com o aumento do intervalo de valores, e com a inclusão de cruzamentos, de combinações e de valores inválidos (ilegais).

Cobertura Funcional



Exemplos

Exemplo MPEG4



Exemplo MPEG4

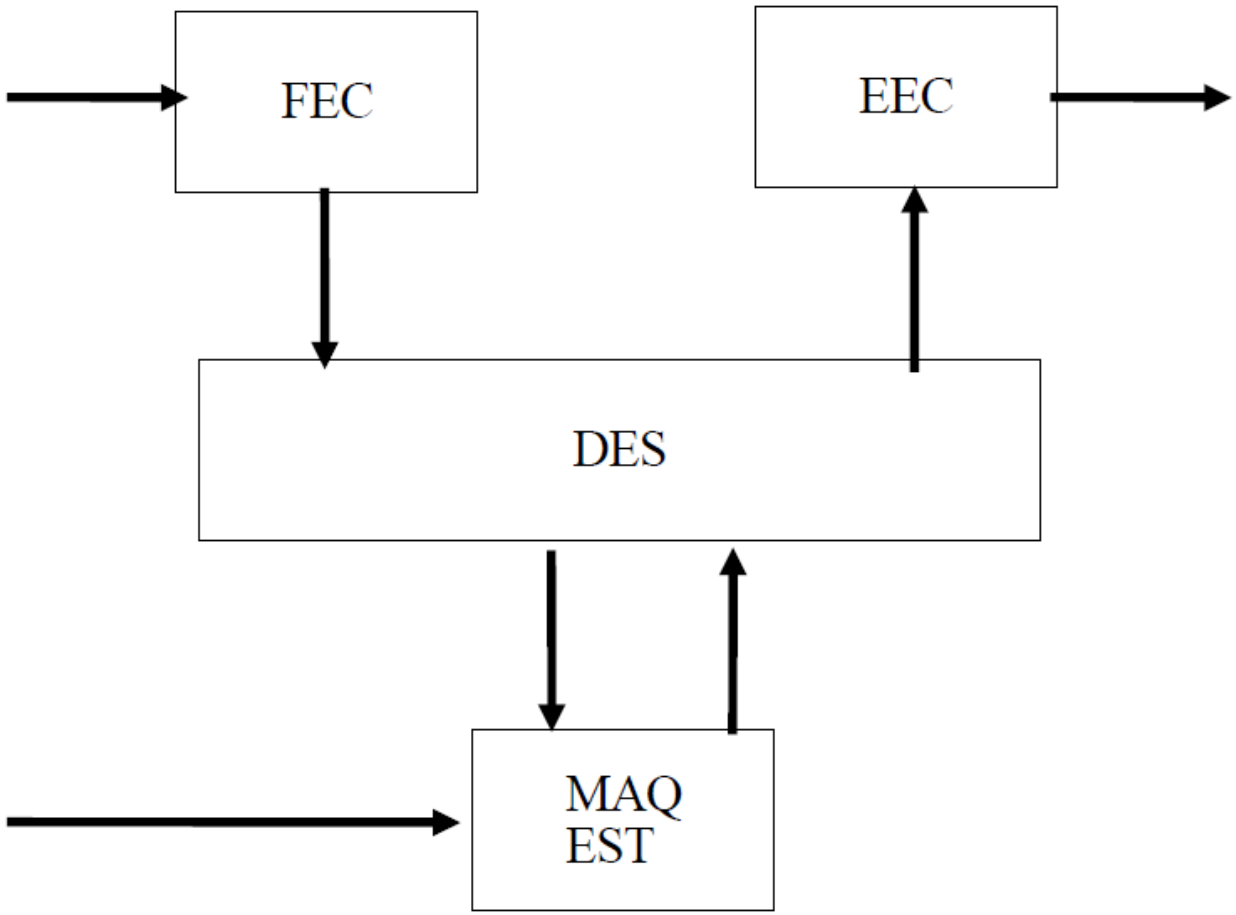
Testbench depois do RTL, com ferramenta para protótipos.

Pior problema: erramos em fazer um só testbench para PIACDC e QI

Tempo entre Verificação dos sub-blocos o.k. e verificação do conjunto dos DUVs o.k.: **3 semanas**

Tempo entre Verificação o.k. e FPGA rodando: **3 dias**

Exemplo Lacre Eletrônico



Exemplo Lacre Eletrônico

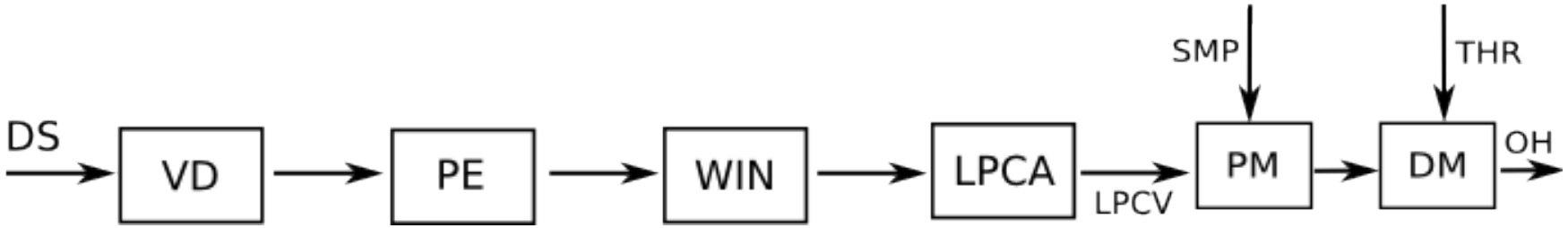
Testbench antes do RTL, sem ferramenta para protótipos

Pior problema: Verificação da junção dos sub-blocos feita já depois de ter iniciado verificação de 2 blocos.

Tempo entre Verificação dos sub-blocos o.k. e verificação do conjunto dos DUVs o.k.: 1 dia

Tempo entre Verificação o.k. e FPGA rodando: 0

Exemplo SPVR



Exemplo SPVR

Testbench antes do RTL, com ferramenta para protótipos de arquivo de código

Pior problema: Execução de passos posteriores sem passos anteriores estarem bem executados.

Tempo entre Verificação dos sub-blocos o.k. e verificação do conjunto dos DUVs o.k.: ??? dias

Tempo entre Verificação o.k. e FPGA rodando: ??? meses

Obs.: Do momento em que a verificação foi dada como concluída, até o momento em que o IP rodou na FPGA, nenhuma alteração foi feita no RTL.

Exemplo SPVR

Estatísticas:

Testbench: ~7450 linhas
RefMod: 523 linhas
RTL: 1367 linhas
Total: ~9340 linhas

FPGA: 30.156 elementos lógicos (Cyclone II 70k)
ASIC: ~1.200.000 transistores (NAND equivalent)