

# 研究・標準化動向分析にもとづく システムレベル設計手法の提案

## - JEITA SLD研究会の活動から-

JEITA EDA技術専門委員会 SLD研究会

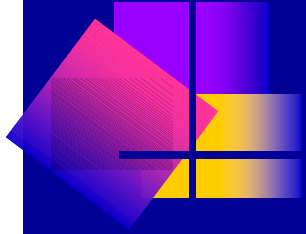
黒坂 均 (NECエレクトロニクス) 温 兆祺 (メンター・グラフィックス・ジャパン)

荒木 大 (インターデザイン・テクノロジー) 吉永 和弘 (沖電気工業)

齊藤 博文 (三洋電機) 野々垣 直浩 (東芝) 大塚 正人 (富士通)

竹村 和祥 (松下電器) 塚本 泰隆 (リコー)

<http://eda.ics.es.osaka-u.ac.jp/jeita/eda/english/project/sld/index.html>



# 発表の構成

- 黒坂 ← 1 .はじめに (SLD研究会紹介)
- 黒坂 ← 2 .研究・標準化動向分析
- 野々垣 ← 3 .設計手法に関する調査と提案
  - 3.1 ハードウェア/ソフトウェア協調検証
  - 3.2 動作合成
  - 3.3 ハードウェア検証
- 大塚 ← 4 .モデリングに関する調査と提案
- 荒木 ← 5 .消費電力見積りに関する調査と提案
- 黒坂 ← 6 .むすび

# 1.はじめに

SLD研究会は、JEITA EDA技術専門委員会の下部組織に属しており、1998年11月から2003年7月まで活動した。

(社)電子情報技術産業協会(JEITA) 電子デバイス部会

半導体事業委員会

EDA技術専門委員会 23社 (2002年度)

System Level Design (SLD)研究会 16社 + 3大学 (2002年度)

Physical Design Methodology (PDM)研究会

標準化小委員会

EDS Fair 実行委員会

# SLD研究会のメンバ

半導体メーカー、システムメーカー、EDAベンダの委員と  
大学などの研究機関の客員から構成される (延べ107名)

## 1998年度

主査 安田 光宏  
副主査 山口 雅之  
委員 嶋崎 等  
岡田 茂之  
古知屋 正樹  
橘 昌良  
中島 克彦  
栗原 郁夫  
中村 敦資  
浅野 哲也  
大塚 正人  
堺 宏明  
相沢 真  
客員 今井 正治

## 1999年度

主査 安田 光宏  
副主査 山口 雅之  
委員 岡田 茂之  
古知屋 正樹  
橘 昌良  
山田 明宏  
栗原 郁夫  
黒坂 均  
浅野 哲也  
大塚 正人  
堺 宏明  
相沢 真  
川原 常盛  
吉田 憲司  
客員 今井 正治  
小澤 時典

## 2000年度

主査 安田 光宏  
副主査 山口 雅之  
副主査 黒坂 均  
委員 宇部 努  
松村 浩二  
古知屋 正樹  
荒木 大  
茂木 浩介  
栗原 郁夫  
小林 和彦  
山下 智規  
大塚 正人  
竹村 和祥  
奥内 康議  
堺 宏明  
温 兆祺  
客員 吉田 憲司  
今井 正治  
橘 昌良  
吉田 紀彦  
小澤 時典

## 2001年度

主査 黒坂 均  
副主査 山口 雅之  
副主査 温 兆祺  
委員 宇部 努  
齊藤 博文  
百瀬 茂  
古知屋 正樹  
荒木 大  
葉久 尋之  
杉山 陽一  
小林 和彦  
山下 智規  
大塚 正人  
菰田 浩  
竹村 和祥  
山元 浩幸  
木村 仁  
客員 吉田 紀彦  
橘 昌良  
今井 正治  
特別委員 安田 光宏

## 2002年度

主査 黒坂 均  
副主査 山口 雅之  
副主査 温 兆祺  
委員 荒木 大  
吉永 和弘  
齊藤 博文  
牧野 真  
橋本 毅久  
野々垣 直浩  
入江 将裕  
杉山 陽一  
小林 和彦  
大塚 正人  
菰田 浩  
李 建道  
竹村 和祥  
山元 浩幸  
木村 仁  
塚本 泰隆  
客員 今井 正治  
吉田 紀彦  
橘 昌良  
特別委員 安田 光宏

	1998	1999	2000	2001	2002	合計
委員	13	14	16	18	20	81
客員	1	2	5	3	3	14
合計	14	16	21	21	23	95



# SLD研究会の目的

- \* 設計現場の声を集める
- \* 設計現場に設計手法や技術を紹介、システムレベル設計普及のきっかけを作る

## Phase1 (1998.11-2001.3)

- 1)システムレベル設計に対するニーズ、シーズの調査
- 2)システムレベル設計フローの提案
- 3)システムレベル設計言語の調査
- 4)システムレベル設計フローへの適用化検討

## Phase2 (2001.4-2003.7)

- 1)標準化団体と研究機関の調査
- 2)設計フロー、モデリング、消費電力見積り手法の調査と提案

# 活動の足跡

主な活動内容	1998年度	1999年度	2000年度	2001年度	2002年度
標準化動向 / 研究動向調査		SLDI (P)	システムレベル設計言語調査		
ニーズ調査		システムレベル設計に関するニーズ調査 (14社153件の回答)		標準化動向・研究動向調査	
シーズ調査		システムレベル設計に関するシーズ調査			
設計フロー検討		システム提案・改善提案			
モデリング検討		システムレベル設計フローの提案			
見積り手法検討					

キーツール調査  
(動作合成、HW-SW協調検証、HW検証)

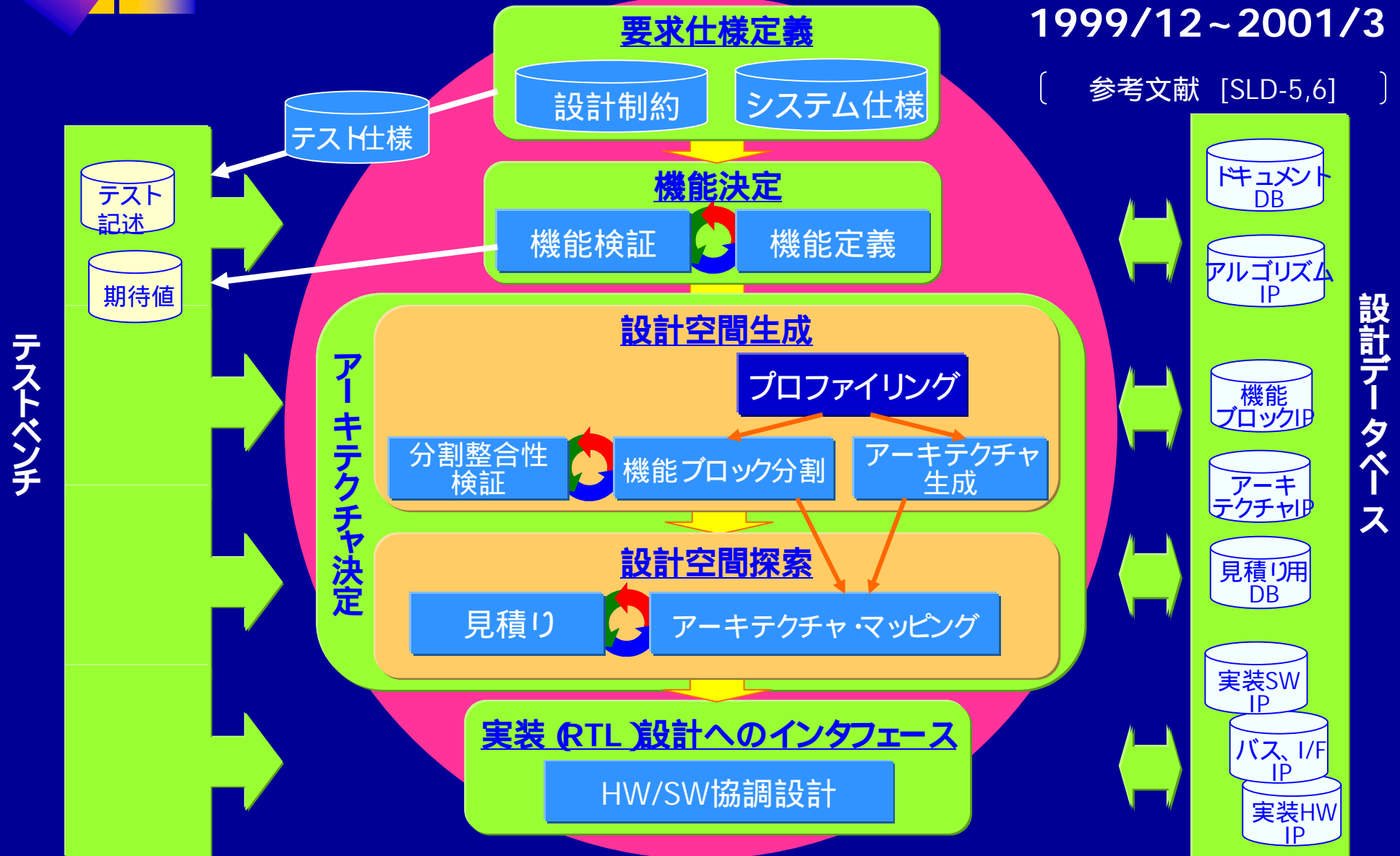
計算モデルと適用性評価

消費電力見積り技術調査

# システムレベル設計フロー

1999/12 ~ 2001/3

[ 参考文献 [SLD-5,6] ]



# 設計フロー実現のための課題

琵琶湖WS発表時点での課題 (2000/11)

1. 高速/高精度見積り技術
2. 機能ブロック自動分割技術
3. プロファイリング技術
4. IP設計データベースインフラ整備
5. アーキテクチャ決定ツールの充実
6. 仕様からテストベンチ生成ツールの実現
7. システム仕様、設計制約の記述標準化
8. システムモデルの標準化
9. 合成セマンティクスの定義
10. 設計フローの実証検証

研究機関の活動  
を調査、分析

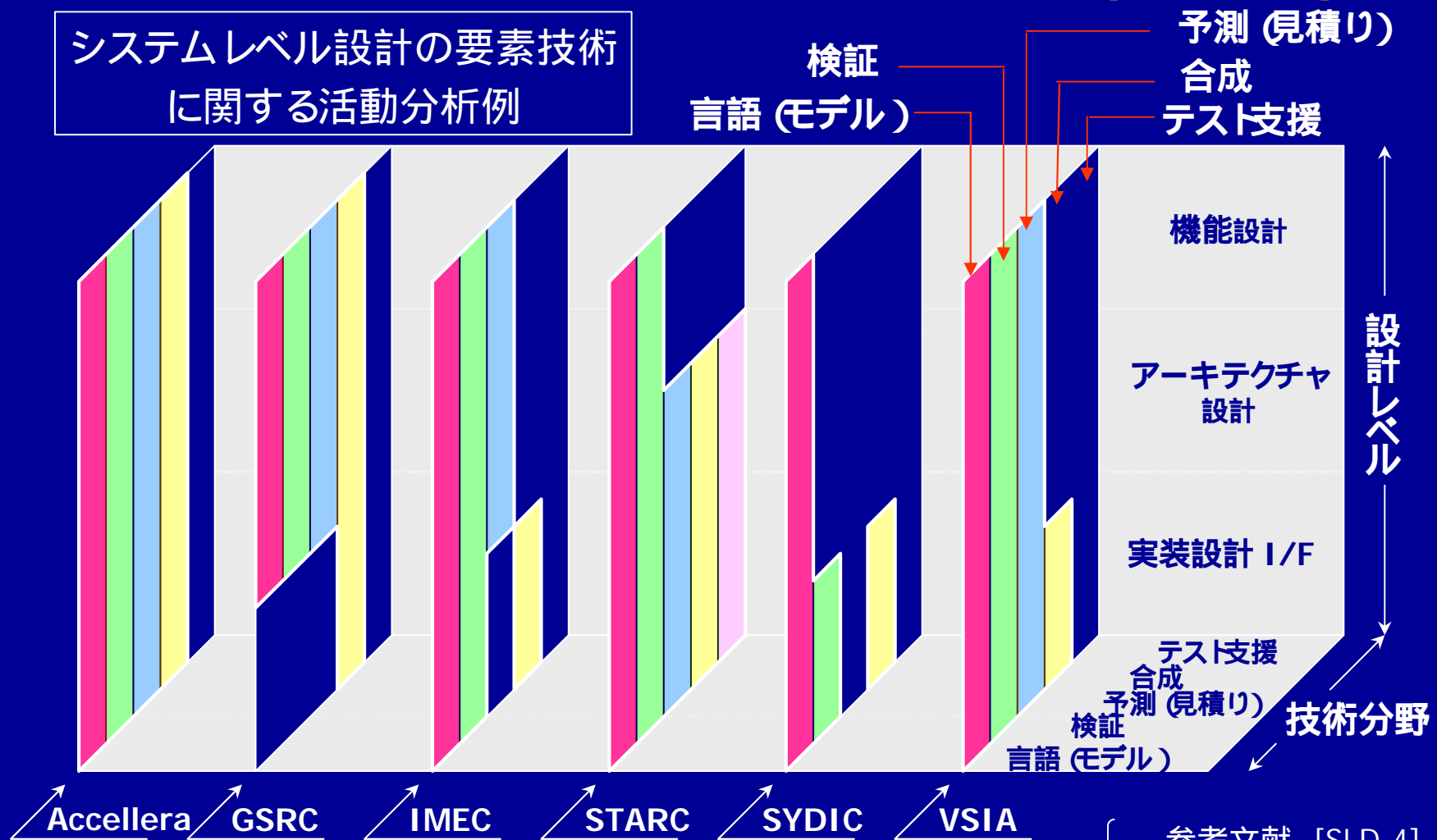
標準化団体の活動  
を調査、分析



# 2. 研究・標準化団体動向分析

3つの研究機関と3つの標準化団体にフォーカスして  
システムレベル設計に関する活動調査を実施（～2002/3）

システムレベル設計の要素技術  
に関する活動分析例



標準化団体/研究機関

# 設計フロー実現のための課題と現状

琵琶湖WS発表時点での課題 (2000/11)

現時点(2003/7)での状況

1. 高速/高精度見積り技術
2. 機能ブロック自動分割技術
3. **プロファイリング技術**
4. **IP設計データベースインフラ整備**
5. アーキテクチャ決定ツールの充実
6. **仕様からテストベンチ生成ツールの実現**
7. **システム仕様、設計制約の記述標準化**
8. システムモデルの標準化
9. **合成セマンティクスの定義**
10. 設計フローの実証検証

EDAツールの提供

PSL (Accellera)などの標準化、EDAツールの提供

SystemC (OSCI), SystemVerilog (Accellera)の標準化、EDAツールの提供

SystemC合成サブセット定義の活動開始 (OSCIのSynthesisWG)

# 設計フロー実現のための課題と現状

琵琶湖WS発表時点での課題 (2000/11)

1. 高速/高精度見積り技術

2. 機能ブロック自動分割技術

5. アーキテクチャ決定ツールの充実

8. システムモデルの標準化

10. 設計フローの実証検証

見積りTG

モデリングTG

設計手法TG



## 3. 設計手法に関する調査と提案

### 章構成

- 背景と目的
- ハードウェア/ソフトウェア協調シミュレーション
  - マルチレイヤHW/SW協調シミュレーション提案
- 動作合成
  - 動作合成ツールの評価
- ハードウェア検証
  - アサーションベース検証
- 提案と課題のまとめ



# 背景と目的

## 背景

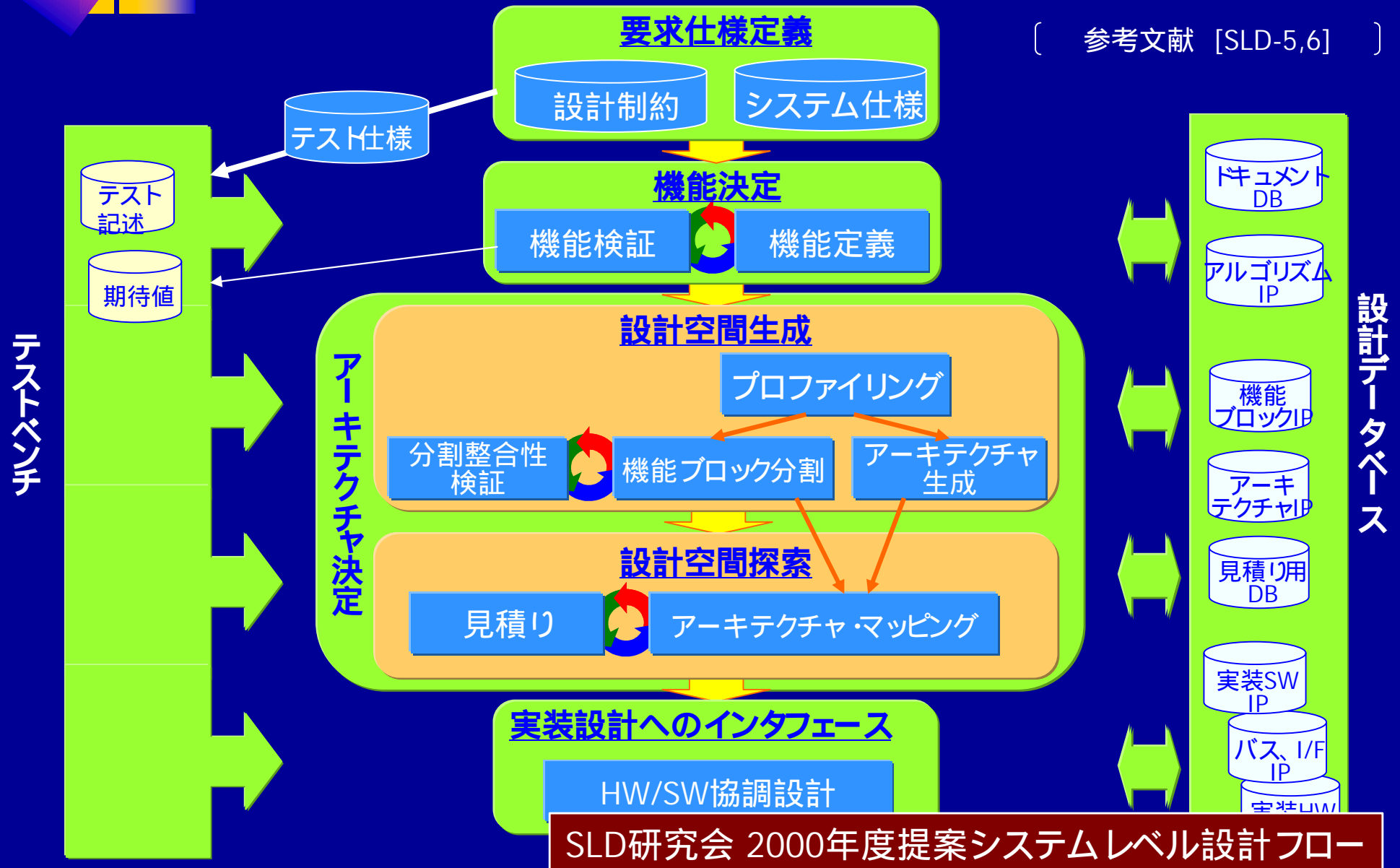
- 多くの製品設計は流用設計が主流である
  - プラットフォームベース設計手法が普及しつつある
- システムレベル設計領域での利用可能技術が少ない
  - C言語ベースの設計手法が注目されつつある
- ツール利用技術が未成熟である
  - ツールの実力評価、適切な適用範囲の設定、有効利用に関するガイドラインの不足など。

## 目的

- システムレベル設計フロー普及のための技術/手法/施策を提案する

# システムレベル設計フロー

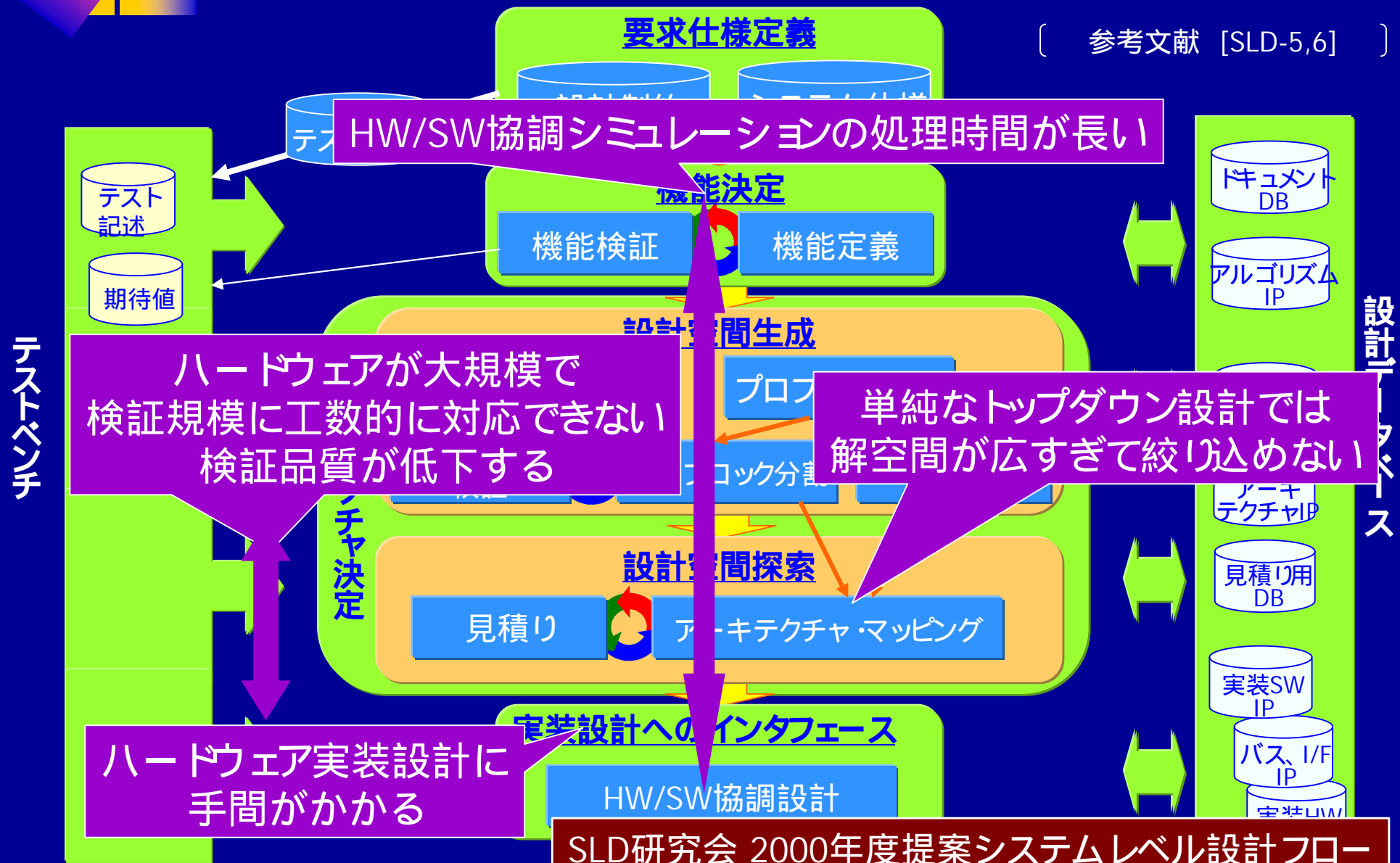
[ 参考文献 [SLD-5,6] ]



SLD研究会 2000年度提案システムレベル設計フロー

# システムレベル設計フロー上の課題

[ 参考文献 [SLD-5,6] ]





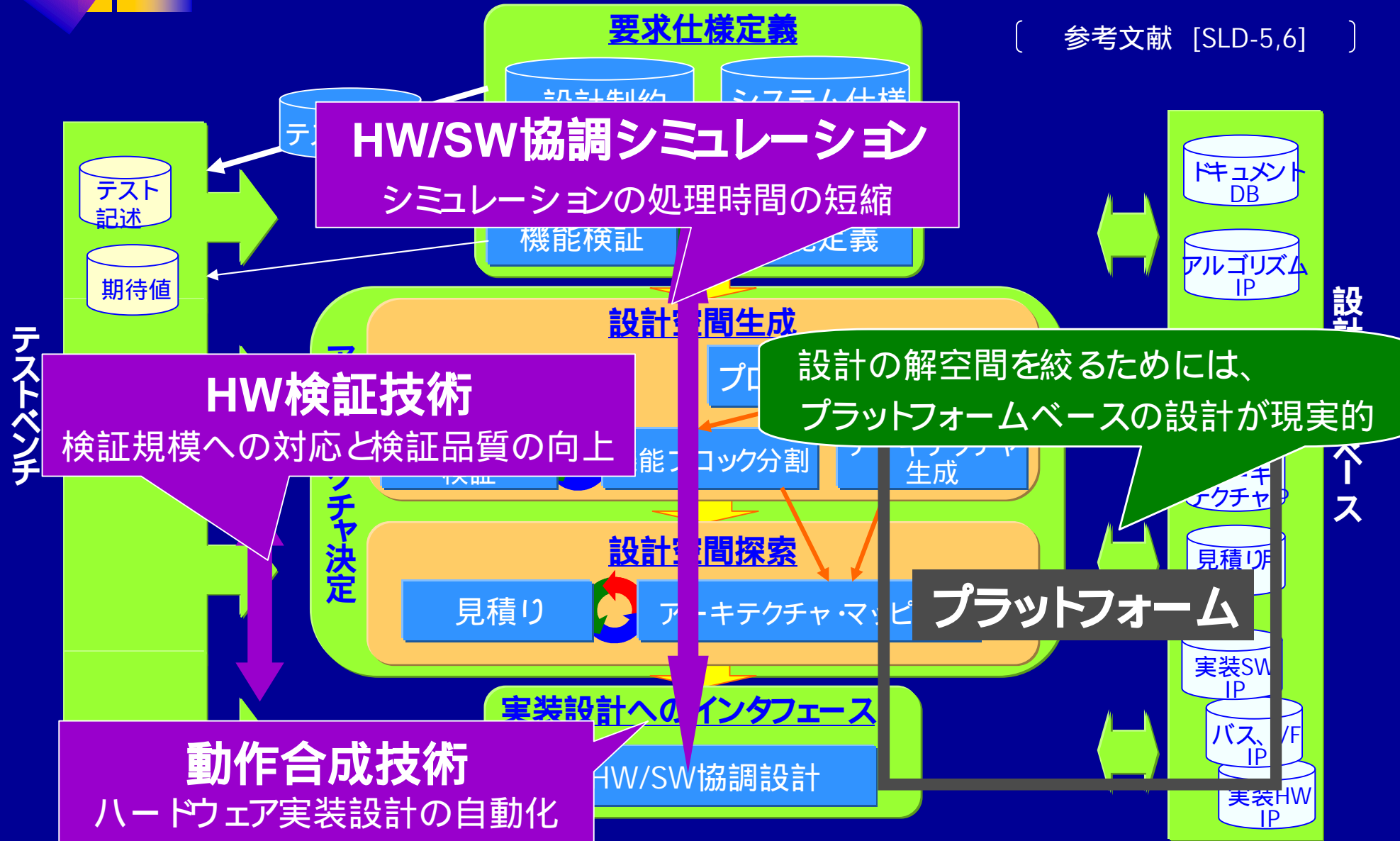
# 普及のための重点技術

- 設計の解空間が広すぎて絞り込めない
  - プラットフォームベースでの設計が現状では現実的
- HW / SW協調シミュレーションの処理時間が長い
  - 使用目的に適したツール機能、精度、パフォーマンスの調査
  - レイヤを意識したシミュレーションモデルの利用を提案する
- ハードウェアの実装設計に手間がかかる
  - 動作合成ツールの現状について調査
  - 問題点の指摘と改善要望項目を提案する
- ハードウェア設計の検証品質が低下する
  - アサーションベース検証の技術と使用環境を調査
  - ハードウェア検証への積極的使用を提案する



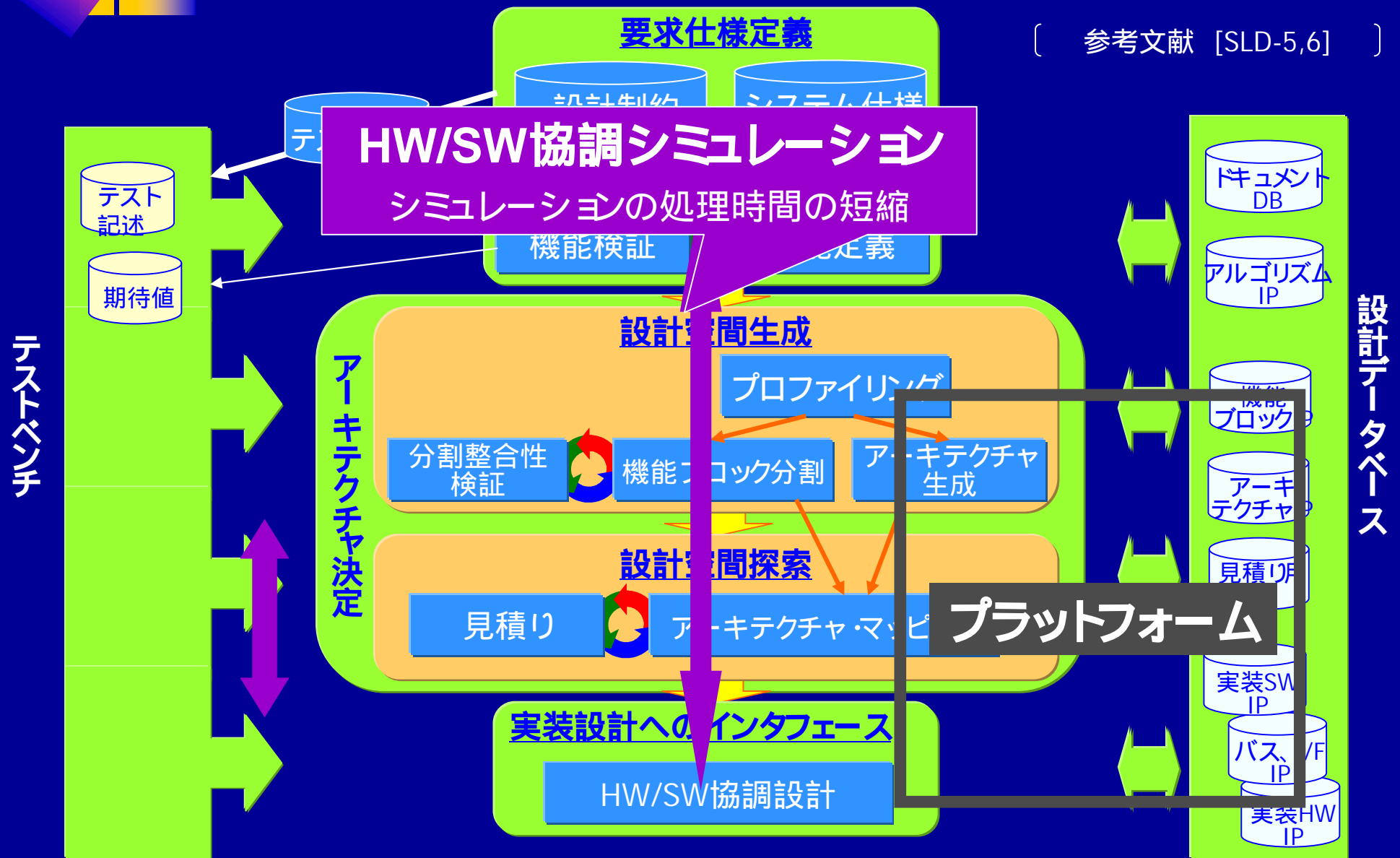
# 普及のための重点技術

[ 参考文献 [SLD-5,6] ]



# 3.1 HW/SW協調シミュレーション

[ 参考文献 [SLD-5,6] ]





## 3.1 HW/SW協調シミュレーション

### 背景

- HW/SW間インタフェースの早期検証は必須であると考えられている
  - ハードウェア開発完了を待たずに、ソフトウェア開発においてHW/SW間のインタフェース検証を効率よく実施できることが望まれる
- HW/SW双方の設計規模が増加している。
  - 十分に短い時間でHW/SW協調シミュレーションを完了できない
    - OSのブート検証に数時間～数日かかる
- 各種のHW/SW協調シミュレーションツールは出揃いつつある
  - ツールによって検証できる項目が不明確である
    - タイミングについてどこまで信頼できるのか?
    - キャッシュやバス等のハードウェアリソース競合の影響は評価できるのか?



# HW/SW協調シミュレーション

## 課題

- HW/SW協調シミュレーションの処理時間がかかる
- HW/SW協調シミュレーションによって検証できる項目が不明確

## 目標

- 現在のHW/SW協調シミュレーションツールの動向を踏まえた、効率的なHW/SW協調シミュレーションを実施する方法論を提案する

## 提案

- マルチレイヤでHW/SW協調シミュレーションを実施をする
- HW/SWインタフェースの階層に合わせて複数の適切なレイヤを定義し、それぞれのレイヤに特化したシミュレーションモデルおよびシミュレータを使用する

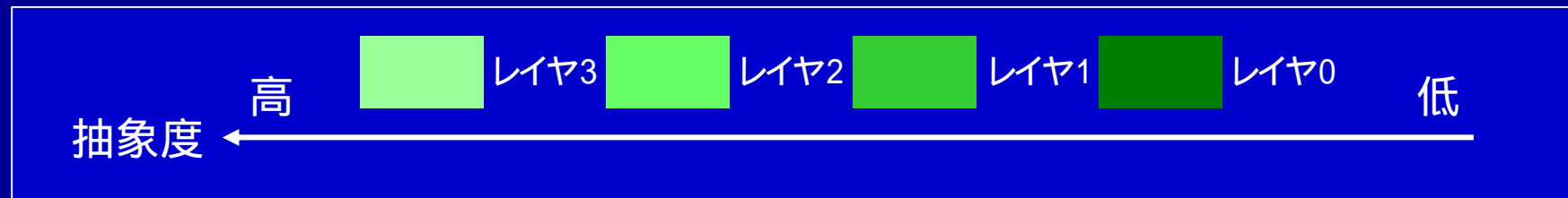
## 効果

- シミュレーション目的にあわせて精度とシミュレーション速度性能のトレードオフを最適化できる
  - 効率的に協調シミュレーションが実施できる

# HW/SW協調シミュレーション手法の提案

HW/SW協調シミュレーション用に検証モデルの  
抽象度 (レイヤ) を定義する

- “レイヤ”の定義
  - レイヤとはシミュレーション時に保証される精度および保証しない精度を決定することによって定義した抽象度
- シミュレーション・レイヤ例
  - 開発するシステムの特長や、開発体制によって、適切なレイヤの定義は異なると考えられる
  - 参考文献[Co-sim 1-7]等でいくつかのレイヤリングが提案されている



# HW/SW協調シミュレーション手法の提案

## シミュレーション・レイヤ例について説明する

レイヤ	SWモデル	HWモデル	目的
L3	Host Native	Abstract Function Layer Model	<ul style="list-style-type: none"><li>- アルゴリズム、処理の流れの検証</li><li>- HW/SW分割の見積り</li></ul>
L2	Target Code + ISS	Transaction Layer Model	<ul style="list-style-type: none"><li>- リソース配分後の HW動作および動作レベルでのタイミング検証</li><li>- オブジェクトコードレベルでの HW/SWインタフェース検証</li></ul>
L1	Target Code + Cycle Accurate ISS	Transfer Layer Model	<ul style="list-style-type: none"><li>- サイクル精度での HWタイミング検証</li><li>- サイクル精度での HW/SWインタフェース検証</li></ul>
L0	Target Code + Processor RTL Model	RTL Model	<ul style="list-style-type: none"><li>- ピンレベル動作とサイクル精度での HWタイミング検証</li><li>- サイクル精度での HW/SWインタフェース検証</li></ul>

# HW/SW協調シミュレーション手法の提案

## 各レイヤで期待するシミュレーション速度について述べる

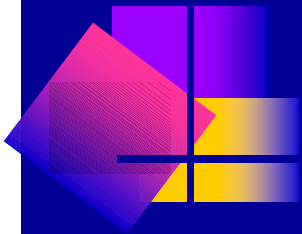
レイヤ	レイヤ間速度比の目標	速度低下の要因
L3	$10^0$	
L2	$10^{-1} \sim -3$	<ul style="list-style-type: none"><li>- ISSモデルおよびHWモデルを駆動するために速度低下が生じる。</li><li>- HW/SWそれぞれのモデルの記述および構造が処理速度に影響する。</li></ul>
L1	$10^{-3} \sim -5$	<ul style="list-style-type: none"><li>- HW/SW双方でクロックサイクル単位での精度を保证するため速度低下が生じる。</li></ul>
L0	$10^{-5} \sim$	<ul style="list-style-type: none"><li>- クロック駆動/ピン動作/イベント等の詳細なシミュレーション、F/Fの正確なシミュレーションのため速度低下が生じる。</li></ul>



# シミュレーションツールの調査

- 現在入手可能なHW/SW協調シミュレーションツールについて調査を実施した
  - 調査対象:9社 9 ツール
    - ツール名は A~Iとする
  - 調査方法:カタログレベルで調査
- 例示した3~0の4レイヤでシミュレータの各機能を分類
- マルチレイヤHW/SW協調シミュレーションを行うには
  - 同一レイヤに属する機能のみを備えた特化したシミュレータが望ましい
  - 複数レイヤを“混在してシミュレーションできる”ことをうたった環境の場合、各レイヤの定義(シミュレーション精度)とシミュレーション処理速度の関係が明示されていることが望ましい





# シミュレーションツールの調査結果

評価項目	A	B	C	D	E	F	G	H	I
ソフトウェアのホストネイティブ実行	レイヤ3	レイヤ2	レイヤ1	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
接続可能なソフトウェア開発向けシミュレータ	命令セットシミュレータ(ISS)	レイヤ2	レイヤ1	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
	サイクルアキュレートプロセッサシミュレータ	レイヤ2	レイヤ1	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
ソフトウェアデバuggとの接続性	ターゲットプロセッサ用デバuggが接続可能	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
	専用デバuggが利用可能	レイヤ3	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
ハードウェアモデルのインポート	C言語等で記述されたハードウェアモデル	レイヤ2	レイヤ1	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
	HDLで記述されたハードウェアモデル	レイヤ2	レイヤ1	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
ハードウェアシミュレータの接続インタフェース	FLI	レイヤ2	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
	PLI	レイヤ0	レイヤ0	レイヤ1	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
接続可能なハードウェアシミュレータ	サイクルベースシミュレータ	レイヤ0	レイヤ1	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
	イベントドリブンRTLシミュレータ	レイヤ0	レイヤ1	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0
	ハードウェアアクセラレータ(エミュレータ)	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0	レイヤ0

レイヤ3
  レイヤ2
  レイヤ1
  レイヤ0 (A~Iは各ツールに対応する)



# HW/SW協調シミュレーションのまとめ

## ■ HW/SW協調シミュレータの課題

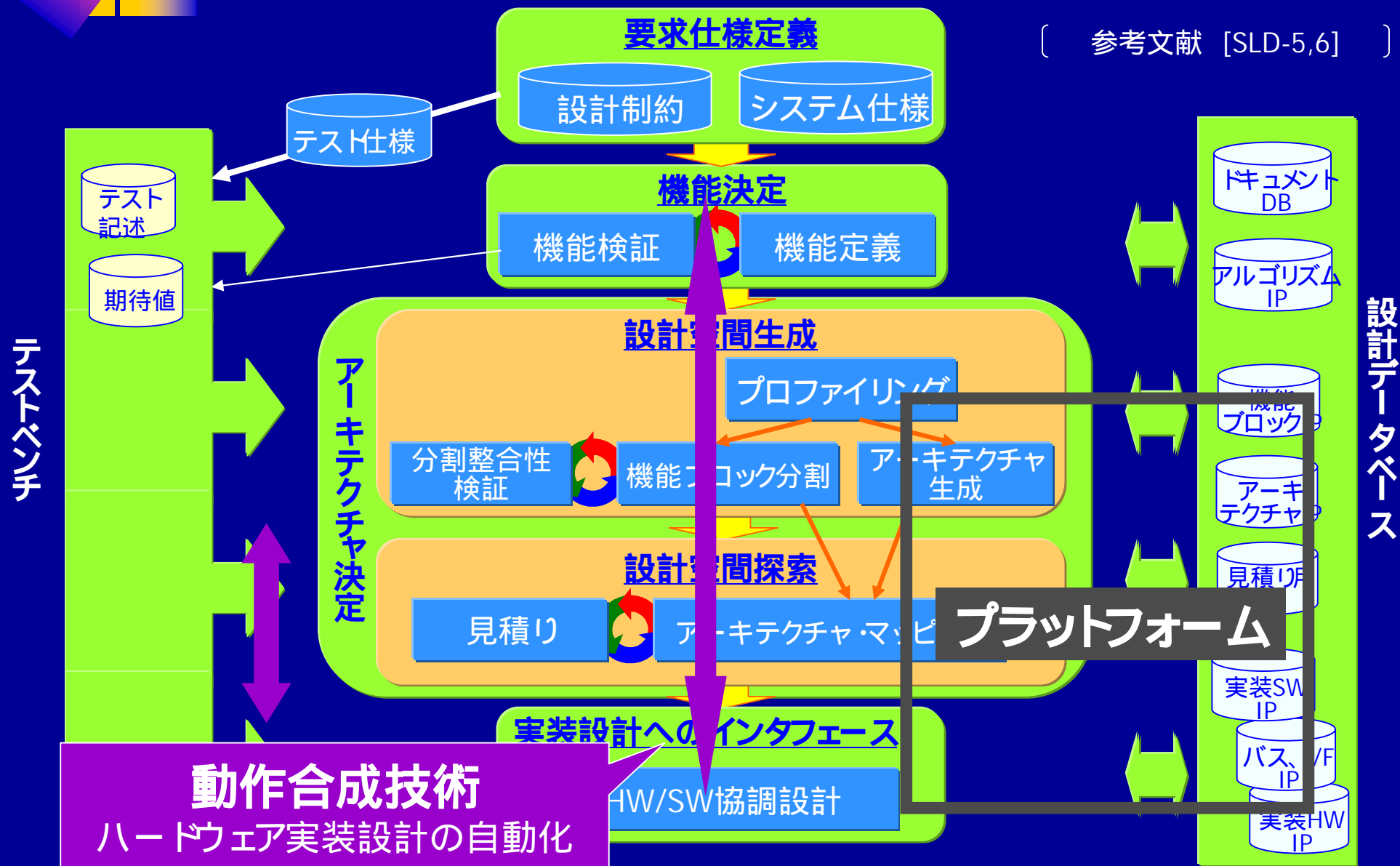
- 検証用途ごとに必要なシミュレーション精度は異なる。
- シミュレーション精度と速度の間にはトレードオフが発生する。
- すべてを詳細な精度でシミュレーションするのは効率的ではない。

## ■ マルチレイヤHW/SW協調シミュレーションの提案

- 検証用途にあわせたマルチレイヤ・シミュレーションモデルを用意し、必要な精度を維持してシミュレーション処理時間を最小限に抑える。
- シミュレーションモデルは保証されるシミュレーション精度とともに**保証されないシミュレーション精度**の明確な定義を行い、適切なシミュレータを選択することが重要である。

# 3.2 動作合成

[ 参考文献 [SLD-5,6] ]





## 3.2 動作合成

### 背景

- C言語ベースの開発によるシステム記述と検証高速化が期待されている
  - 動作レベルのC記述から、RTL記述の自動生成の要求が高い
  - プラットフォーム設計ではシステムの大部分を再利用、新規開発部分が数十万ゲート規模で動作合成の適用可能性の期待あり

### 目的

- 現状の動作合成ツールの実用性を評価

### 方法

- EDAベンダー (6社) の動作合成ツールについて、機能に関する評価項目表を作成
- 実用的なツールがあるかどうかを評価項目表から検討

# 動作レベルとRTレベルの違い

動作レベルとRTレベルの違いは **レジスタ記述の有無** とする

記述レベル	クロック記述	レジスタ記述
動作レベル (Untimed Function)	なし	なし
動作レベル (Cycle Accurate)	あり	なし
RTLレベル	あり	あり

UnTimed Function 動作レベルの記述例

```
for( i=0; i<7; i++){  
    a = b * c;  
    d = e * f;  
    g = a + d;  
}
```

Cycle Accurate 動作レベルの記述例

```
for( i=0; i<7; i++){  
    a = b * c;  
    clock();  
    d = e * f;  
    clock();  
    g = a + d;  
    clock();  
}
```

RTレベルの記述例(擬似コード)

```
always @(posedge clk)  
    current_state <= next_state;  
  
always @(current_state or ...)  
case( current_state)  
s1 : i=0;  
    a = b * c;  
    i=i+1;  
    flag=(i <7);  
    next_state = s2;  
s2 : if( flag==1){  
    d = e * f;  
    next_state = s3;  
    }  
    else{  
    next_state = s1;  
    } .....  
}
```



# 動作合成ツールの評価

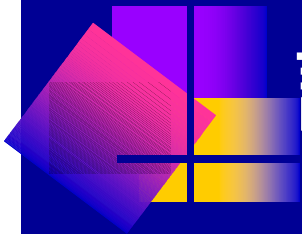
- 動作合成ツールとは
  - 動作合成ツールとは論理合成可能なRTL記述を、動作レベル記述から自動生成するツールである。
    - 少なくともレジスタの割当を自動的に決定する。
    - 各クロックサイクルごとにどのような演算を実施するかを決定する。
- 動作合成ツールの評価方法
  - カタログスペックでの評価
    - 要素技術項目をピックアップし、対応、未対応の表を各動作合成ツールに対し作成
  - C言語ベース言語に対応した合成ツールを中心に評価

# 評価結果

## ツール

	A	B	C	D	E	F
1 Cベースでの配列をレジスタ(レジスタファイルでは無い。フリップ・フロップ)にマッピングできるか？						
2 並列記述 / 合成は可能か？					×	
3 パイプライン記述 / 合成は可能か？						
4 自動スケジューリング機能はあるか？			×			
5 人手によるスケジューリング機能はあるか？				×	×	
6 入力可能な設計制約条件は？						
処理サイクル数			×			
使用可能なリソースの数 (例、乗算器の数)			×	×	×	×
動作周波数			×			
面積	×	×	×		×	×
7 ループの全展開 / 部分展開機能はあるか？						
8 等価性チェック(RTL vs Gates)が容易にかかるRTLか？						
9 生成されるRTLの可読性は高いか？						
10 2ポートRAMへのマッピングは可能か？						
11 サイクル・アキュレートなコードを生成できるか？						
12 自動パイプライン化機能はあるか？			×			
13 <b>ブロック単位での自動並列化機能はあるか？</b>	×	×	×	×	×	
14 浮動小数点から固定小数点への変換手段はあるか？	×	×		×	×	×
15 論理合成や等価性チェックなどを考慮して、階層構造を持ったRTLを生成できるか？	×				×	×
16 ポインタについては扱えないケースが明らかにされているか？						
17 構造体は扱えるか？メンバーとしてポインタはOK？配列は？						
18 ステートマシンの自動分割機能はあるか？	×	×	×	×	×	×
19 関数の再帰呼び出しは？	×	×		×	×	×
20 malloc、freeは可能か？	×	×	×	×	×	×

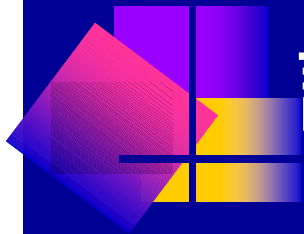
対応、 : 一応対応しているがユーザは苦労しそう あるいは不明、× 未対応



# 評価結果

		A	B	C	D	E	F
5	人手によるスケジューリング機能はあるか？				×	×	
7	ループの全展開 / 部分展開機能はあるか？						
8	等価性チェック(RTL vs Gates)が容易にかかるRTLか？						





# 評価結果

		A	B	C	D	E	F
2	並列記述 / 合成は可能か？					×	
3	パイプライン記述 / 合成は可能か？						
4	自動スケジューリング機能はあるか？			×			
13	ブロック単位での自動並列化機能はあるか？	×	×	×	×	×	

# 評価項目例

		A	B	C	D	E	F
2	並列記述 / 合成は可能か？					×	

```
func1();  
  
for( i =0; i < 10; i++ ){  
    a[i] = b[i] *c[i];  
}  
for( i =0; i < 10; i++ ){  
    d[i] = e[i] *f[i];  
}  
func2();
```

## ブロック単位

- 上記二つのforブロックを並列処理させるような記述手段がCベース記述にあるか？またRTLを合成可能か？
- func2()は、二つのforブロックの終了を待ってから開始。VerilogHDLのfork-joinに対応する動作。この動作を簡単に合成できないツールや言語がある。

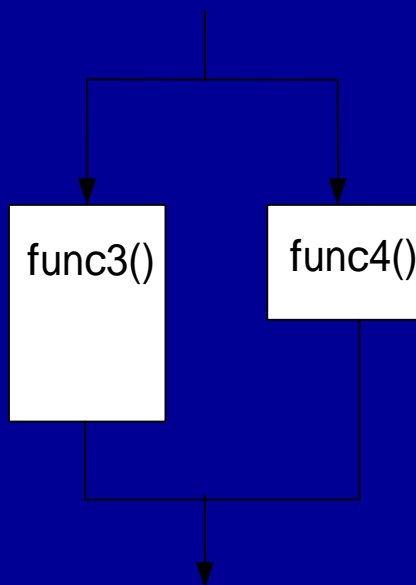
# 評価項目例

		A	B	C	D	E	F
2	並列記述 / 合成は可能か？					×	

## 関数単位

```
for( i=0; i < 10; i++){  
  en1 = func1(i);  
  en2 = func2(i);  
  func3(en1);  
  func4(en2);  
}
```

- func3() とfunc4() を並列に動作させるための記述手段はあるか？また、RTLを生成できるか？
- func3() とfunc4() の両方が終わるまで次のループに行ってはいけない。この動作を簡単には記述できない合成ツールや言語がある。





# 評価項目例

		A	B	C	D	E	F
3	パイプライン記述 / 合成は可能か？						

## ブロック単位

```
for( i=0; i <10; i++ ){  
    a = x * i;  
    b = y * i;  
    c[i] = b*2;  
}
```

## 関数単位

```
for( i=0; i < 10; i++ ){  
    func1();  
    func2();  
    func3();  
}
```

パイプライン動作を記述したり、合成したりできるか？左側のケースは自動パイプライン化できるツールも、右側のケースになるとダメな場合もある。

# 評価項目例

		A	B	C	D	E	F
2	並列記述 / 合成は可能か？					×	
4	自動スケジューリング機能はあるか？			×			
13	ブロック単位での自動並列化機能はあるか？	×	×	×	×	×	

## 関数単位

```
for( i=0; i < 10; i++){  
    en1 = func1(i);  
    en2 = func2(i);  
    func3(en1);  
    func4(en2);  
}
```

## ブロック単位

```
for( i =0; i < 10; i++ ){  
    a[i] = b[i] *c[i];  
}  
for( i =0; i < 10; i++ ){  
    d[i] = e[i] *f[i];  
}
```

並列性を自動抽出できるか？

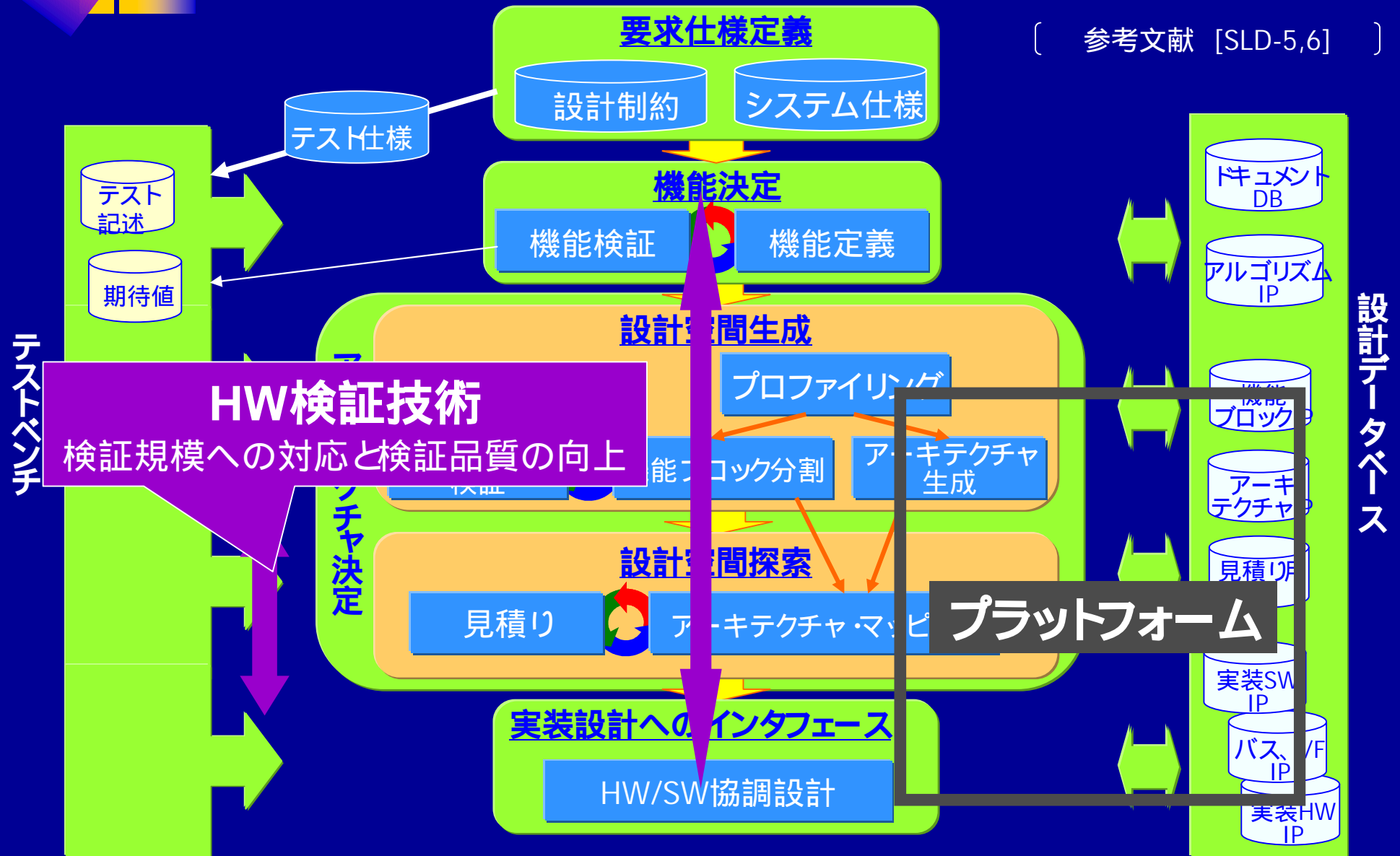


# 動作合成のまとめ

- ハードウェア設計のための合成機能がそろってきた
- 並列性の自動抽出、パイプライン動作の合成機能が弱い
  - オリジナルのC言語記述をかなり書き換える必要がある。
- 動作合成ツールベンダーへの要求
  - 並列性の自動抽出機能の向上
  - 並列性を容易に記述 (指定) できる必要あり
    - 並列性の自動抽出機能が弱い場合は必須
  - ポインタや構造体の記述に関するガイドライン作成

# 3.3 ハードウェア検証

[ 参考文献 [SLD-5,6] ]



テストベンチ

設計データベース

## HW検証技術

検証規模への対応と検証品質の向上

## プラットフォーム



## 3.3 ハードウェア検証

### 目的

- 現状の検証に要する多大な工数(設計工数の70%以上)の低減および検証品質を改善する手法の現実的な解として、アサーションベース検証が考えられる。アサーションベース検証の現状調査と適用推進策の提案を行う。
- 設計記述の抽象度がRTレベルから動作レベルまで上がっていることを踏まえ、動作レベルのアサーションベース検証についても検討する。





# アサーションベース検証

- アサーションベース検証[HV 2] [HV 3]とは、デザインのある属性 (プロパティ)の成立を仮定しチェックする検証手法。
  - デザインのHDL記述が、プロパティの仮定に反する条件を検出する。
  - 仕様または論理の欠陥と判断できる。
- アサーションベース検証で検証可能な項目
  - 時間的關係
  - システムの入力または実行結果として成立する/成立しない性質

# アサーション記述

- プロパティ記述言語 (PSL) による記述例
  - プロパティ
    - メモリインタフェースにおいて、wr\_nの立下りで、イネーブル信号ena\_nはHighでなければならない

```
//psl assert always (ena_n) @(negedge wr_n);
```

- "//psl"はPSL記述の開始キーワード
- ";"まで有効
- Vendorによりこの部分は異なる可能性あり

PSL: Property Specification Language (旧名称Sugar) [HV 5]



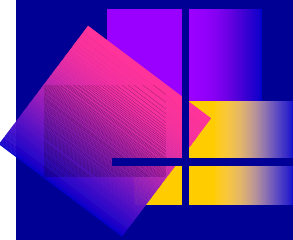
# アサーションベース検証の実現手法

## ■ 動的検証手法

- シミュレーションによって入出力等の仮定が守られているか検証する
- テストパターンを用いる
- 問題発生箇所の絞込みに効果

## ■ 静的検証手法

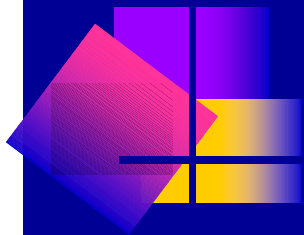
- 形式的に検証する(論理や状態の組合わせから静的に推論する)
- テストパターンを用いない
- シミュレーションではコーナーケースになりやすい項目の検証に効果
- ただし、適用可能な回路規模に限界 動的検証手法の補完が必要



# アサーションベース検証の用途

---

- ブロック内の検証
  - 設計者が期待したように、ブロックが動作するか検証する
- ブロック間の検証
  - ブロックの利用者が、設計者が想定した使い方をしているか検証する
    - チップレベルで、IPを正しく使用しているか？
    - プロトコルに則った使用方法をしているか？



# アサーションベース検証ツール

ベンダー	0-In	@HDL	Averant	Cadence	Synopsys	Verisity	Veritable	Verix	Verplex
ツール名	0-In Check	@Verifier	Solidify	ABV	Vera	Specman Elite	Verity Check	Real Intent	BlackTie
設計記述言語	Verilog	Verilog / VHDL	Verilog / VHDL	Verilog / VHDL	Verilog / VHDL / SystemC / C	Verilog / VHDL	Verilog	Verilog / VHDL	Verilog / VHDL
プロパティ記述言語	独自 / PSL	独自 / PSL	独自 / OVL	PSL	OVA	'e'	独自 / PSL / SVA3.0	独自 / SVA3.0 / OVL	OVL / PSL
検証エンジン (静的 / 動的)	動 / 静	静	静	動	動	動	静	静	静

OVA: OpenVera Assertion, OVL: Open Verification Library,

PSL: Property Specification Language, SVA: SystemVerilog Assertion

2003年3月末現在



# プロパティ記述言語の標準化動向

米国標準化団体Accelleraが主導的に活動している

- SystemVerilog (Accellera)

SystemVerilog3.1を最初のHDVL (Hardware Design and Verification Language) とすべく標準化を推進中でDAC'03で発表予定。[HV 5] [HV 8]

SystemVerilog3.1はpowerful assertion、testbench creation、direct C I/F、およびhigh level abstractionが可能である。Unified assertionsはSystemVerilog3.0、PSLサブセットおよびOVAを統合予定。

- PSL (Accellera)

2003/1にLanguage Reference Manual完成。PSLは言語非依存でSystemVerilog 3.1 Assertionのスーパーセットの位置づけ

- 'e' (IEEE P1647)

Verisity社のプライベート言語だったが、2003/6にIEEEに標準案として提案された(正式採択はまだ)。(ユーザ数が多いのが強み)

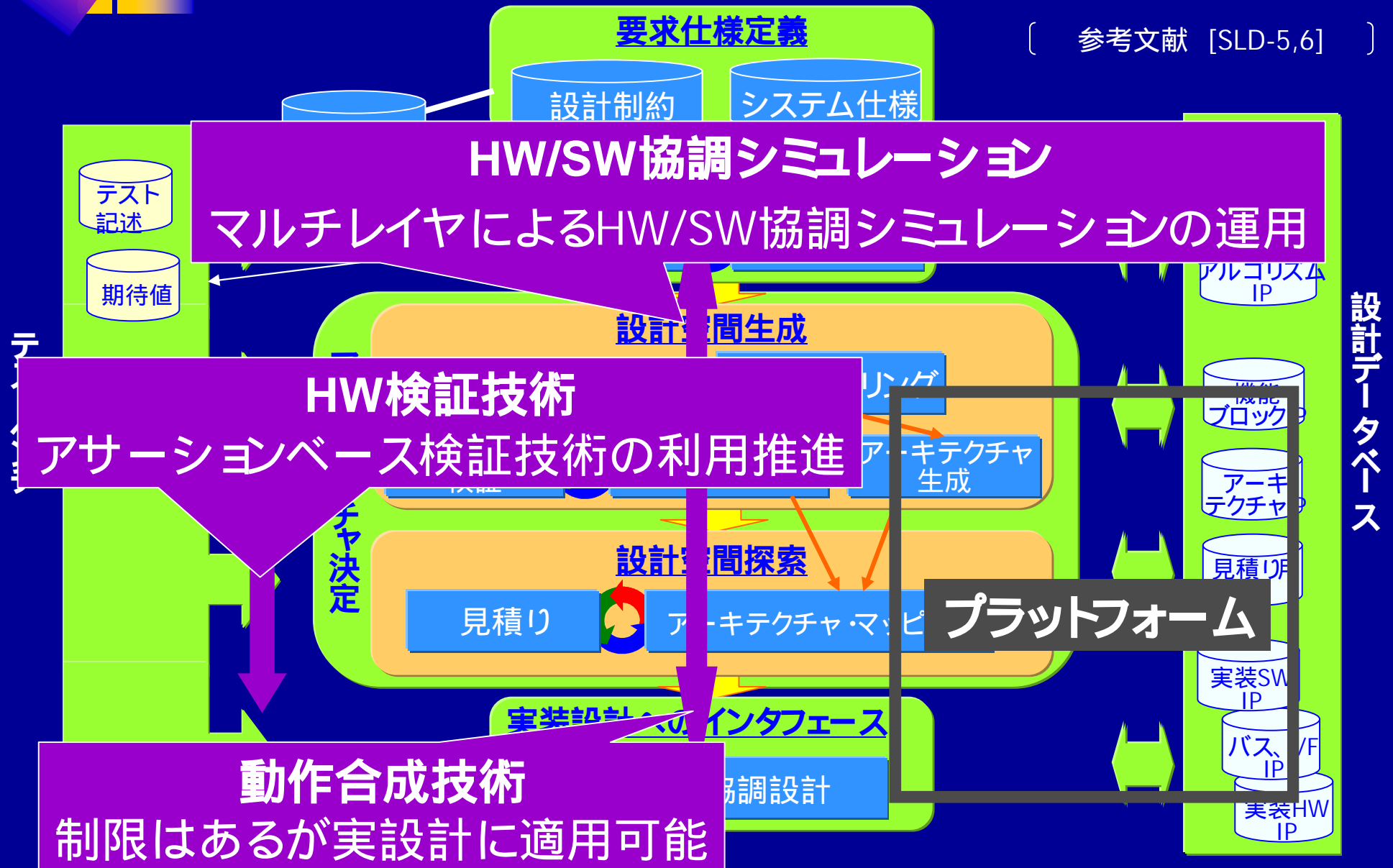


# ハードウェア検証のまとめ

- アサーションベース検証の効果  
検証危機を緩和する有力な手法
  - 静的検証手法は網羅性があるため検証品質が向上
  - 動的検証手法を静的検証手法で補完することで検証効率が向上
- アサーションベース検証のツール/言語
  - 各ベンダーからツール、検証用IPなどが出始めている
  - プロパティ記述言語はAccelleraのSystemVerilog Assertion 3.1とPSLに集約されつつある
- アサーションベース検証の課題
  - 動作レベルで使用したアサーション記述のRTL記述への自動生成
  - アサーションベース検証利用推進策(記法ガイドラインの整備、教育等)

# 課題と提案のまとめ

[ 参考文献 [SLD-5,6] ]







# 研究機関/EDAベンダ/標準化団体への期待

- HW/SW協調シミュレーション
  - 実設計における検証用途を強く意識した研究・ツール開発を期待する。
- 動作合成
  - 動作レベル記述のガイドラインの策定、不足機能の開発と合成品質の向上を期待する。
- HW検証
  - 動作レベルで使用したアサーション記述からRTL記述を自動生成ツールの研究・開発を期待する。



## 4.モデリングに関する調査と提案

---

### 章の構成

- 設計フローとモデリング
- 計算モデル概要
- 特徴比較
- 設計フロー適用可能性
- 提案と課題のまとめ

# SLD研究会の設計フローとモデリング

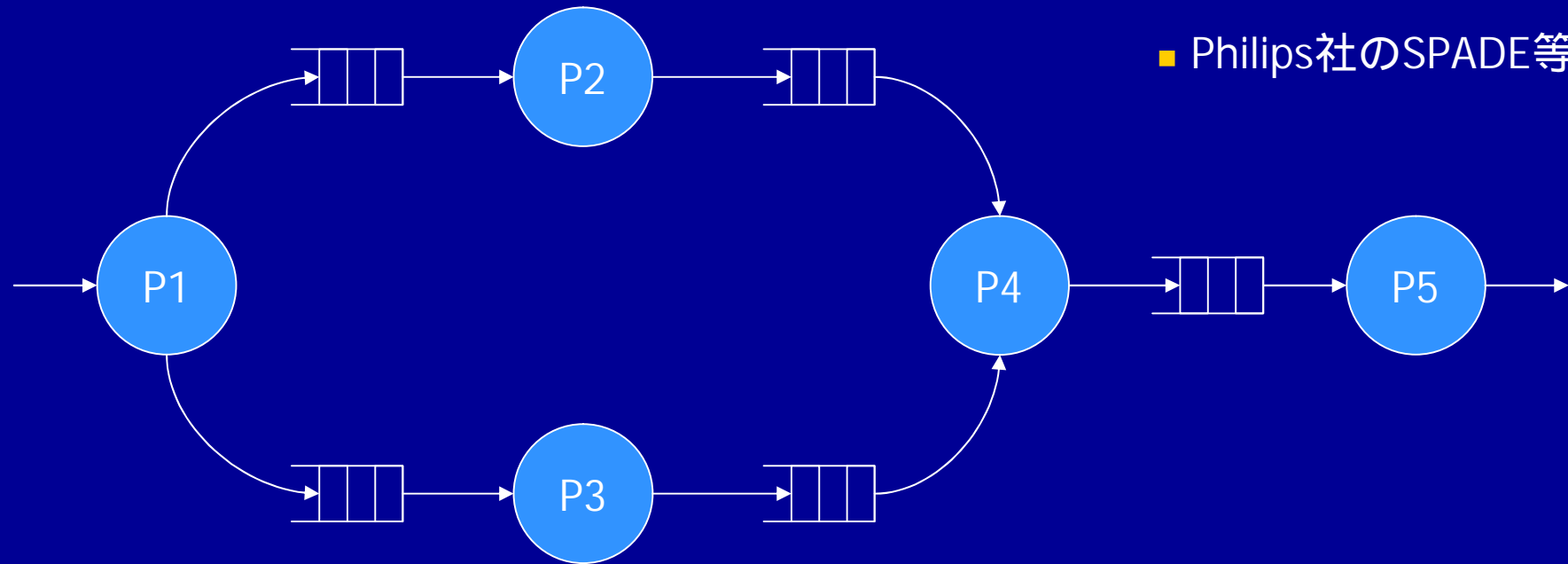




# 計算モデル概要

- 計算モデル(MoC: Model of Computation)
  - システムの動作仕様を形式的に定義し、詳細化するためのフレームワーク
  - システムは、複数のプロセスが互いに通信するプロセスネットワークやペトリネット、FSMベースのモデルを用いてモデル化
- MoCによるシステムモデル化
  - 対象システムに適合したMoCを用いることによって、モデル化の容易性、設計品質、設計生産性が向上

# プロセスネットワーク系MoCの例 :KPN



■ Philips社のSPADE等

● P1 プロセス  
中身はチューリングマシンに準拠

III サイズ無制限FIFO  
書き込みはノンブロッキング  
読み込みはブロッキング



# KPN利用時のメリット/ デメリット(一例)

## ■ メリット

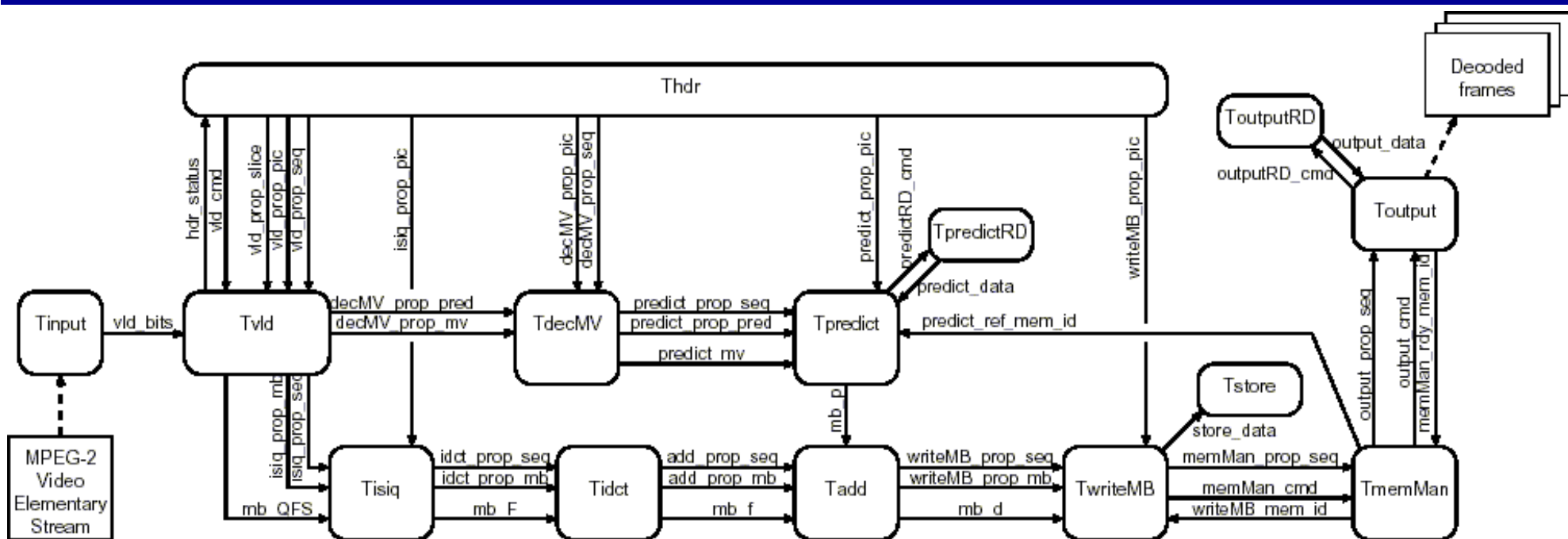
- 実装に縛られず抽象度の高い機能定義可能
  - untimed
  - サイズ無制限FIFO
- プロセスの実行順序に依存せず機能を検証可能
  - 決定性

## ■ デメリット

- 割込みの表現方法なし
- アーキテクチャモデルは別に必要

# MoCによるモデリング例 (KPN)

- MPEG2デコーダの機能モデル
  - プロセス内 / プロセス間の計算量の分析
    - プロセスの粒度と通信データの粒度を決定

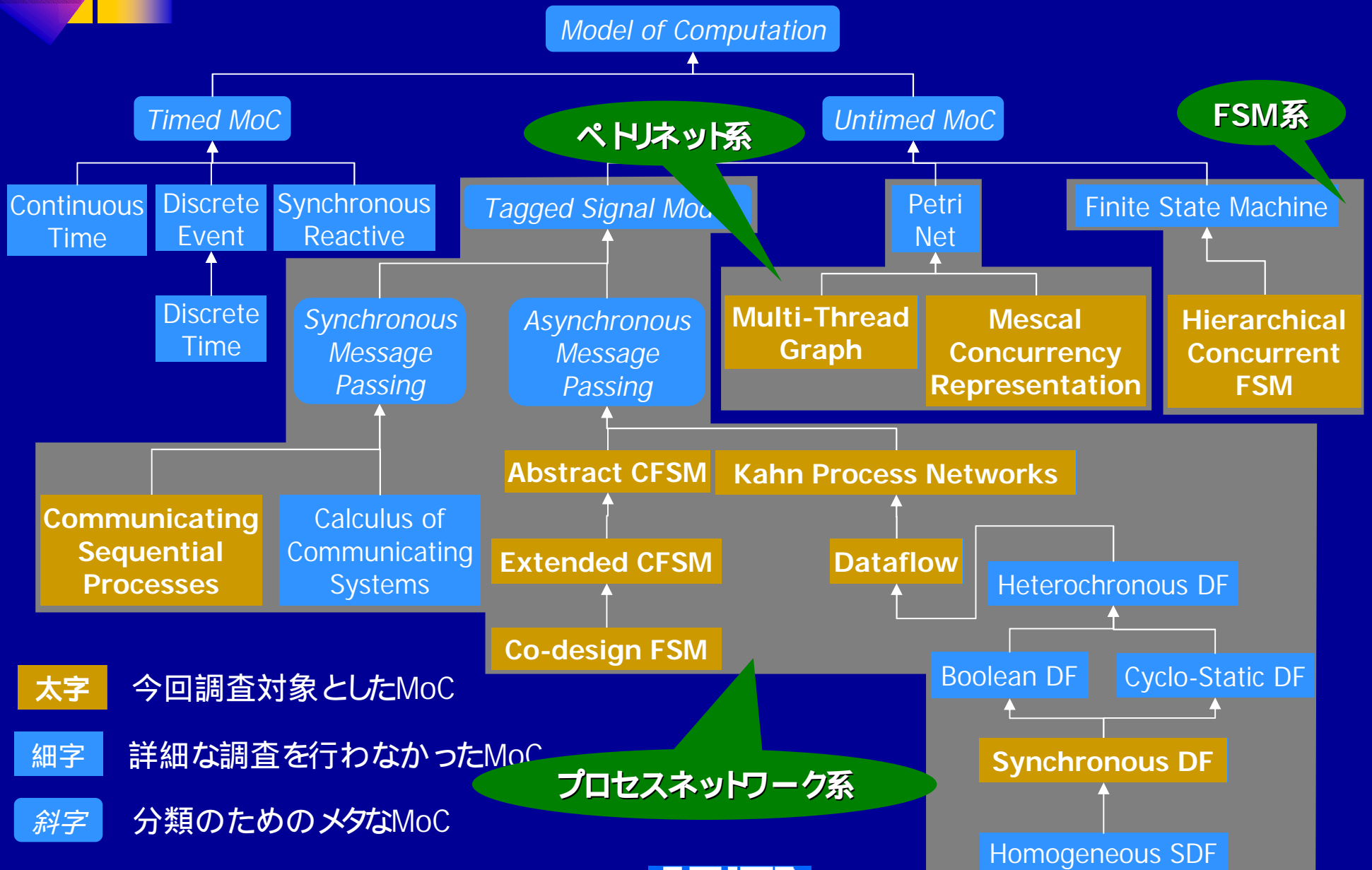


アーキテクチャ・マッピングへ

出展 Pieter van der Wolf他

“An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology”

# MoC関係系統図



- 太字** 今回調査対象としたMoC
- 細字** 詳細な調査を行わなかったMoC
- 斜字** 分類のためのメタなMoC





# 特徴比較

- 各MoCの向き・不向きを明らかにするため、以下の特徴を比較
  - 時間モデル
  - 同期 / 非同期
  - プロセス間通信
  - 割込み
  - 活性化ルール
  - 階層構造
  - プロセスの状態
  - 決定性
  - デッドロック
  - 静的スケジューリング
  - リソースシェアリング

# MoC特徴比較表 (2 / 5)

	プロセス間通信	割込み
KPN	FIFO、サイズ制約なし、ブロッキング・リード/ノン・ブロッキング・ライト	なし
DF	FIFO、サイズ制約なし、ブロッキング・リード/ノン・ブロッキング・ライト	なし
SDF	FIFO、サイズ制約はないがスケジューリングアルゴリズムにより常に固定サイズにすることが可能、ブロッキング・リード/ノン・ブロッキング・ライト	なし
CSP	FIFOなし(プロセスで再帰的にFIFO表現は可能) ブロッキング・リード/ブロッキング・ライトのランデブー通信	プロセス内割込みあり
CFSMs ACFSMs ECFSMs	FIFO、ブロッキング・リードまたはノン・ブロッキング・リード CFSMs: サイズ1、ブロッキング・ライトまたはノン・ブロッキング・ライト、上書きあり ACFSMs: サイズ無限大、ノン・ブロッキング・ライト、上書きなし ECFSMs: サイズ有限、ブロッキング・ライトまたはノン・ブロッキング・ライト、上書きあり	なし
MCR	色付きトークンによるプロセス間通信、FIFOプロセス有、FIFOサイズは無限、FIFO通信はブロッキング・リード/ノン・ブロッキング・ライト	不明
MTG	共有メモリ変数を用いたデータ通信 拡張してFIFO記述対応の予定あり	イベントノードにより表現可能
HCFSM	広域変数による通信	なし

# MoC特徴比較表 (4 / 5)

	決定性	デッドロック
KPN	決定性あり	デッドロックが起こりうるモデルを表現可能だが、一般にデッドロックの有無は判定不能
DF	決定性あり	デッドロックが起こりうるモデルを表現可能。制約を与えることによりデッドロックが起こりえないモデルを表現可能
SDF	決定性あり	デッドロックが起こりうるモデルを表現可能。デッドロックの有無を判定可能
CSP	決定性 / 非決定性のどちらも表現可能	デッドロックが起こりうるモデルを表現可能
CFSMs ACFSMs ECFSMs	アーキテクチャマッピング前：非決定的 アーキテクチャマッピング後：決定性あり	デッドロックが起こりうるモデルを表現可能。デッドロックが起きないことを検証可能
MCR	非決定的	デッドロックが起こりうるモデルを表現可能
MTG	非決定的	デッドロックが起こりうるモデルを表現可能
HCFSM	非決定的	デッドロックが起こりうるモデルを表現可能



# MoCの設計フロー適用可能性

SLD研究会が提案したシステムレベル設計フローに対して、各MoCの適用可能性を検討

- 機能決定
  - 機能定義
  - 機能検証
- 設計空間生成
  - 機能ブロック分割
  - アーキテクチャ生成
  - 分割整合性検証
- 設計空間探索
  - アーキテクチャマッピング
  - 見積り
- 実装設計へのインタフェース
  - 動作合成
  - インタフェース合成

# 適用可能性 (機能決定フェーズ)

	機能定義	機能検証
KPN	システムの機能を複数の並列プロセスに分解し、FIFOを介した非同期メッセージ送信によりプロセス間通信を行う。抽象度が高いモデルを用いてシステムを定義。プロセスはチューリングマシン相当であるため、非常に幅広い機能を定義可能	並列シミュレータによる検証が可能
DF	KPNと同様。ただしスケジューリングの効率化のため、プロセスはアクターと呼ばれるより粒度の小さい単位に分割して定義	firing ruleに基づくスケジューリングにより、KPNと比較してより効率的なシミュレーションによる検証が可能
SDF	DFと同様。ただし更なるスケジューリングの効率化のため、プロセスが消費・生成するトークンの数が常に一定であるようなモデルのみ定義可能	プロセスが消費・生成するトークンの数が常に一定であるため、静的スケジューリングを用いた効率的なシミュレーションによる検証が可能
CSP	複数プロセスがパラレルに動作するシステムの同期の問題を発見できる	CSPモデルの数学的証明により検証可能
CFSMs ACFSMs ECFSMs	ESTERELやVHDLのサブセット等の高級言語を使用	プロパティの中で、インプリメンテーションに非依存のものはこの段階でモデルチェック手法等を用いて検証可能
MCR	システムをMCRの並列ネットワークを用いて定義	MCRを用いて機能検証可能
MTG	システムレベル設計言語の内部モデルを詳細化するために用いられ、CoWareのモデルからMTGを抽出する試みもある。また、リアルタイム組込みシステムの仕様記述用にも開発されており、外部環境からの割り込み記述が可能である	MTGを用いて機能検証可能
HCFSM	ESTERELまたはVEMを使用	並列シミュレータによる検証が可能

# 適用可能性 (設計空間生成フェーズ)

	機能ブロック分割	アーキテクチャ生成	分割整合性検証
KPN	プロセスを分割することで対応	× 別のアーキテクチャモデルを要する	シミュレーションベースで検証
DF	プロセスを分割することで対応	× 別のアーキテクチャモデルを要する	シミュレーションベースで検証
SDF	プロセスを分割することで対応	× 別のアーキテクチャモデルを要する	シミュレーションベースで検証
CSP	機能ブロックをCSPの逐次処理プロセスとして定義できればシステムのモデル化は可能	× 別のアーキテクチャモデルを要する	ブロック分割後のCSPモデルの数学的証明をすることにより、分割整合性が保証される
CFSMs ACFSMs ECFSMs	× あらかじめ分割してからモデルを作成する必要あり	× 別のアーキテクチャモデル (各CFSMの遅延情報)を要する	シミュレーションベースで検証
MCR	MCRの分割が可能	× 別のアーキテクチャモデルを要する	× 可能性はあるが、研究例はなし
MTG	MTGモデルの再構成、異なる分割仕様の生成、有界性の解析、マルチレート等のMTGのモデルを分割する研究が行われている	× 別のアーキテクチャモデルを要する	分割後の処理量の正当性を確認する手段として、レイテンシ制約と応答時間制約の解析や、実行速度の解析、有界性の解析に関する研究が行われている
HCFSM	階層構造による機能の記述が可能のため、ブロックの分割はしやすい	× 別のアーキテクチャモデルを要する	シミュレーションベースで検証



# 設計フロー適用性に関する考察

- KPN/DF
  - 抽象度が高いモデルの記述に向いており、特に機能検証フェーズで有効。下流フェーズではドメインに特化した他のMoCを階層的に利用して実装につなげるためのツールが望まれる
- SDF
  - データフロー系アプリケーションに特化した研究・開発がなされており、DSP開発ツール等が実用化されている
- CSP
  - ソフトウェア分野で数学的証明手法の研究がなされており、設計につながる実用的なツールが期待される
- CFSMs/ACFSMs/ECFSMs
  - 研究機関やベンダーでツール開発がされており、設計適用例も報告されている
- MCR
  - 研究の初期段階にあるようだが、設計フローの多くをカバーする研究がなされているようであり、今後に期待
- MTG
  - 豊かな表現力により様々なシステムをモデル化可能である。一方MoCが複雑すぎると記述の目的に応じてガイドライン等が必要であろう
  - 消費電力見積り等の幅広い研究がなされているが、ツール開発はまだこれから
- HCFSM
  - 制御系アプリケーションの記述に有効



# 提案と課題のまとめ

- 調査内容
  - システムレベル設計における機能決定及び機能分割のための言語・ツールのベースとなる10種類のMoCに関するサーベイと特徴比較を実施
- 提案/主張
  - MoCの特徴を正しく理解し利用することで設計品質 設計生産性の向上が可能
  - MoC/システムレベル言語/ツールの相互補完が重要
- 課題
  - 要求仕様定義から機能決定につなげる技術
  - MoCとアーキテクチャ生成をうまくつなげる技術
- 研究機関/EDAベンダ/標準化団体への期待
  - MoCに関する研究 調査活動の活発化
  - MoCの利点を活かしたツール開発と実用化
  - 使用MoCの選定ガイドラインや記述ガイドラインの策定





# 5.消費電力見積りに関する調査と提案

## 章構成

- 背景
- 低消費電力設計手法の調査
- 消費電力見積り手法の分類
- ツールの調査と見積り技術の適用性検討
  - 「機能決定」における見積り技術の適用検討
  - 「アーキテクチャ決定」における見積り技術の適用検討
- 提案と課題のまとめ



# 背景（消費電力見積り技術へのニーズ）

- 携帯情報機器を中心としたSoCのアプリケーションが発展
    - 製品の小型化に伴い、電池駆動で長時間動作、低発熱が要求されるようになった
    - 一方で、製品の高機能・高性能化を実現するために、SoCの大規模化・高性能化が進み、消費電力が増える要因は増えている
- SoCの低消費電力化への厳しい要求  
低消費電力設計技術が重要

## 現状

- 低消費電力化は、レイアウト・回路・プロセスレベルでの技術が中心
  - システムレベル設計での消費電力削減効果が高いと期待されている
- システム ~ 1/100      RTL/論理 ~ 1/2、レイアウト ~ 2/3、回路 ~ 1/3、プロセス ~ 1/10  
(半導体技術動向に関する調査研究報告、平成12年3月、日本電子機械工業会より)

## 課題

- システムレベルでの低消費電力設計技術は未成熟で実用化はこれから



# 低消費電力化設計手法の調査

[ L. Benini & G. De Micheli, System-level power optimization: Techniques and Tools, ISLPED99 ]

## A) HW/SW partitioning

- 消費電力の観点で最適な分割ポイントを見つける  
Avalanche

## B) Instruction-level power optimization

- アプリケーションにとって最適な命令セットを見つける
- コンパイラの最適化
  - 命令毎の消費電力見積り値を使って最適化  
文献 ( DAC 2000 No.18.3 ) 文献 ( DAC 2000 No.21.1 )

## C) Control-data-flow transformation

- HWの消費電力低減を指向した動作合成技術によるアプローチ
  - 動作合成の前処理としてCDFGを変換する技術

## D) Memory optimization techniques

- メモリアクセスに伴う電力消費を低減させるアプローチ  
ATOMIUM

# 低消費電力化設計手法の調査 (つづき)

## E) Interface power optimization

- バス転送に伴う電力消費を低減させるアプローチ
  - バスエンコーダ
    - 文献 ( ICCAD 2000 6D.1 )
    - 文献 ( CODES 1999 )

## その他のアプローチ

## F) Variable-voltage techniques

- 動的電圧制御による低消費電力化

## G) Dynamic power management

- 非稼動状態で、低電力のスリープ状態にさせる

## H) Approximate signal processing

- 演算精度を下げて低消費電力化
- 特定のアプリケーションの性質に依存した手法

本活動では、  
A～Eのアプローチに  
フォーカスする

# 見積り手法の分類

## I. プロセッサでの消費電力

- 命令セットごとの呼び出し回数
- キャッシュヒット率

## II. カスタムHWでの消費電力

- CMOS回路における電力消費の基本算出式

$$P = \sum_i ( \text{スイッチング率} \times \text{キャパシタンス} \times V_{dd}^2 \times f_{clk} )$$

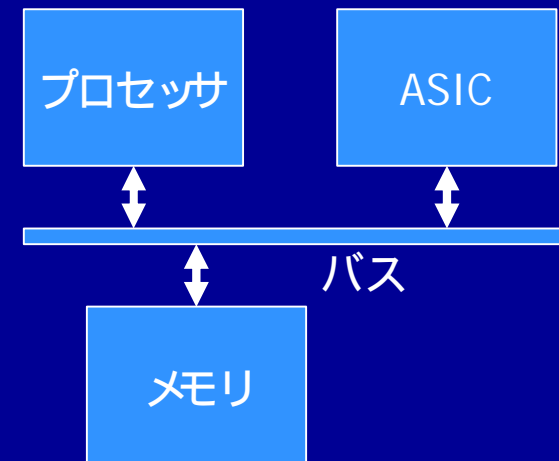
$i$  : 回路のすべてのノード

## III. バスでの消費電力

- データ転送量
- スwitchング量

## IV. メモリでの消費電力

- アクセス回数・ビット数
- 必要なメモリの大きさ



# ツール調査と見積り技術の適用性検討

要求仕様定義

機能決定

機能検証

機能定義

システムレベル設計フローと  
見積り技術の関係

プロファイリング

通信量

演算量

リソース利用(メモリ)

設計空間生成

分割整合性  
検証

機能ブロック分割

アーキテクチャ生成

アーキテクチャ決定

見積り

全体

カスタム HW

SW (CPU)

メモリ

バス

アーキテクチャ・マッピング

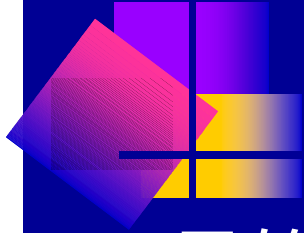
HW

CPU

メモリ

設計空間探索

実装設計へのインターフェース



# 機能決定」における適用検討

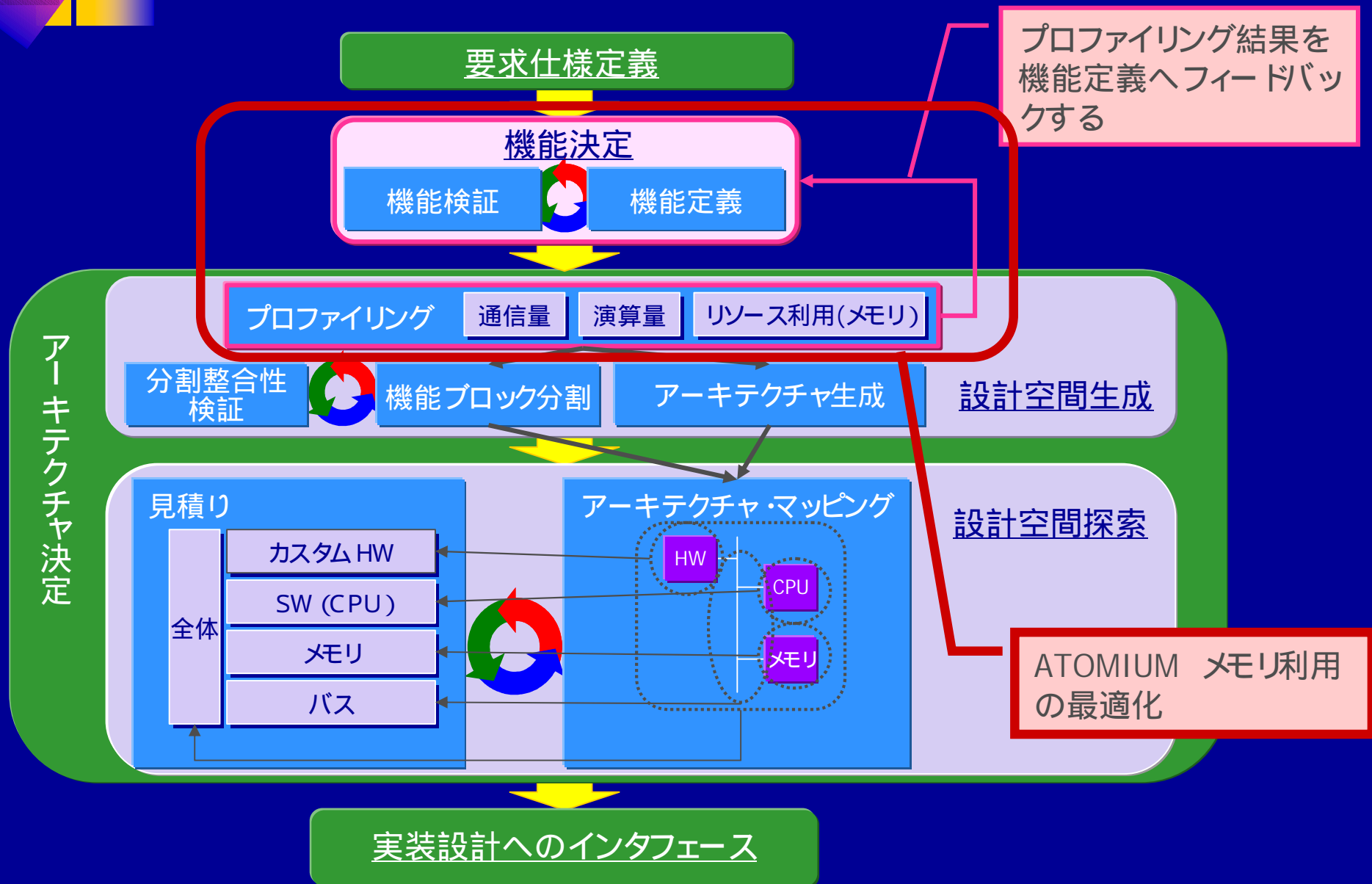
## ■ 目的

- 低消費電力設計の観点で「機能定義」を最適化する

## ■ 前提条件

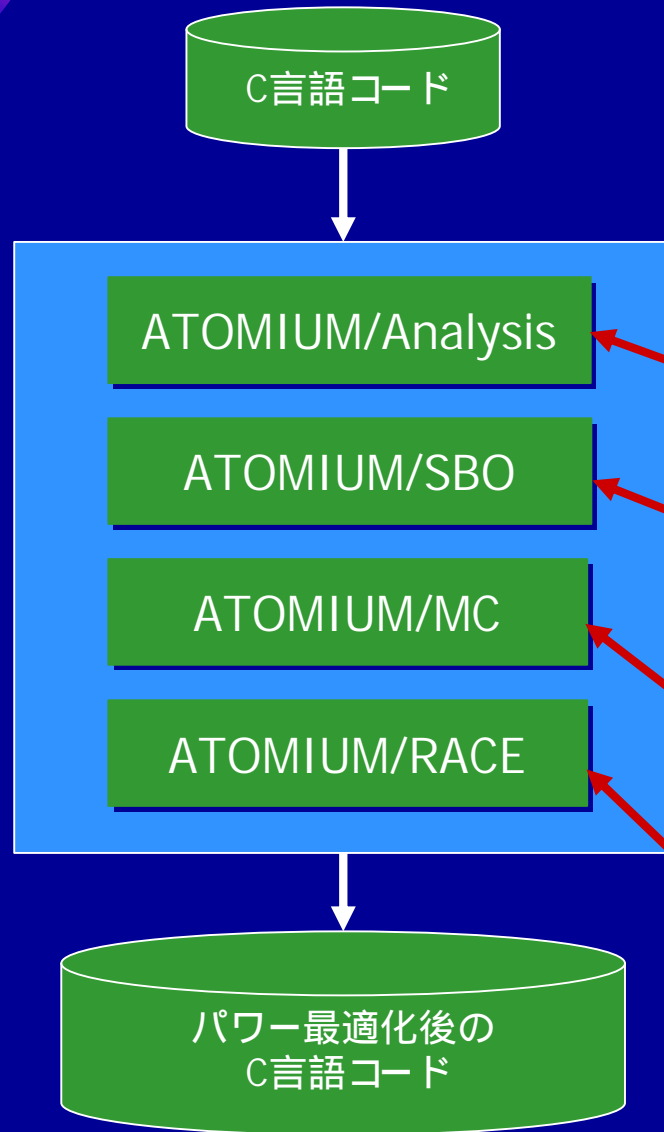
- 設計情報はシステムの機能記述である
  - HW・SWにまだ分割されていない機能レベルの記述
  - たとえば、C言語で書かれたアルゴリズム
- 必ずしもシステム全体の機能記述が存在するわけではない
  - システム全体ではなく、特定の機能だけを最適化したい場合がありえる。
- 実装アーキテクチャが部分的に決まっている場合がある
  - CPU、メモリ、バスなどの構成が決まっている場合は、対象アーキテクチャごとの見積り情報が存在しているという意味

# 検討する適用シナリオ





# ATOMIUMの紹介



## 概要：

- ATOMIUM toolsは、Analysis/SBO/MC/RACEで構成
- 最新バージョン: ATOMIUM 1.2.3(2003年4月現在)
- IMECで開発継続中
- ツールDEMOはAnalysisのみあり  
ただしツール出力結果イメージのみ

メモリアクセスのボトルネック解析

タイミング制約にあった最適なメモリアーキテクチャの探索

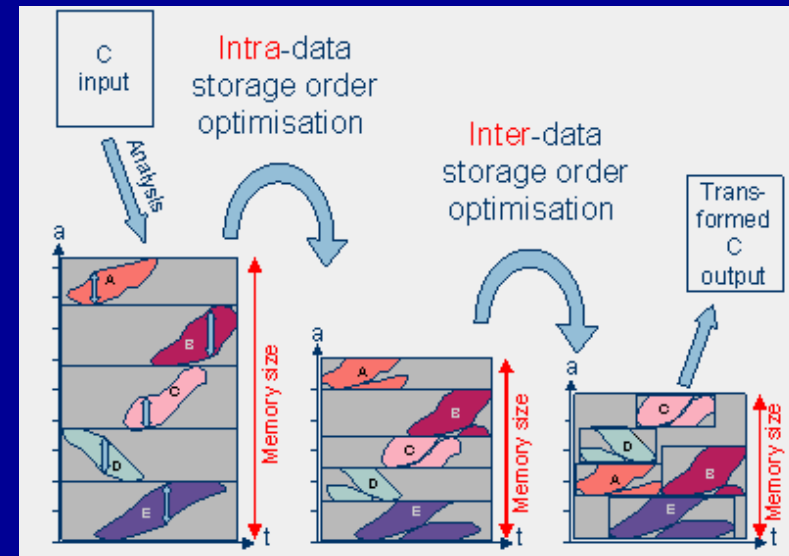
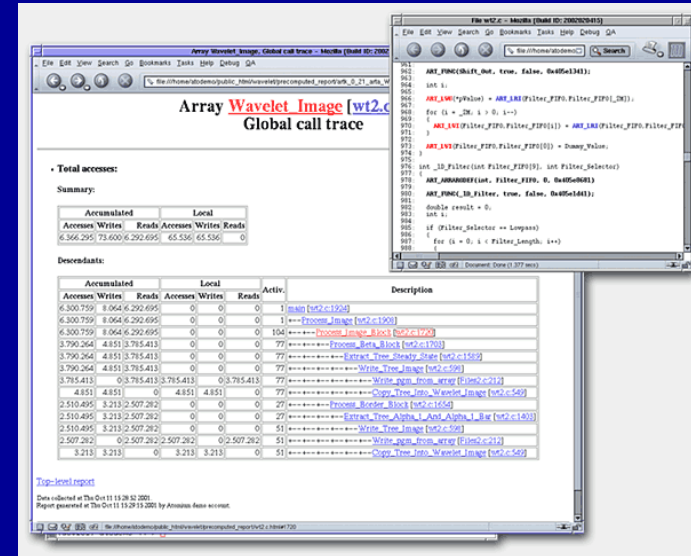
メモリの再利用検討

配列アドレス計算の最適化

[ <http://www.imec.be/design/multimedia/atomium> ]

# ATOMIUMの適用例

- MPEG4 (40k行のヘッダー+120k行のCソース)
  - ATOMIUM Analysis で30%のコードを削減
- GSM auto-correlationカーネル、Medical Imagingアプリカーネル、MPEG4 motion-compensationカー
  - ATOMIUM MCで5%から40%のメモリサイズ削減
- Voice-Coding (2k行のC)
  - ATOMIUM MCでメモリサイズを71KBから55KBに



<http://www.imec.be/design/multimedia/atomium>



# シナリオ適用の課題と解決策の提案

## ■ 課題1

- SWコンパイラの最適化がHWにそのまま適用できるか？

### 解決策1

HWの最適化 (動作合成技術) を考慮した見積りを行う

### 解決策2

HWの最適化はHW/SW分割後に行う

## ■ 課題2

- 見積り精度の問題

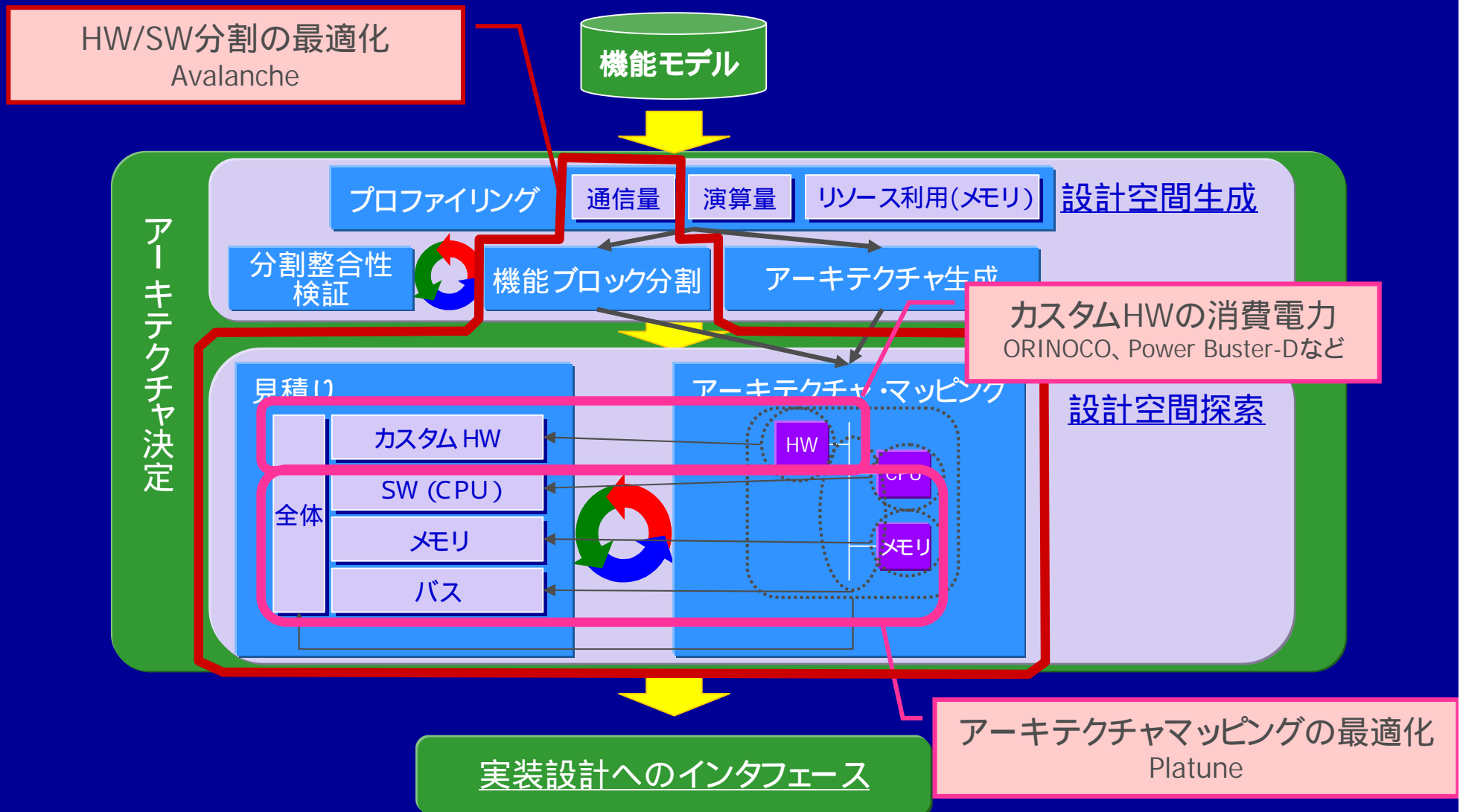
- 絶対評価ではなく相対評価に使うとしても、相対精度の正しさがどこまで保証されるのか？

### 解決策

流用設計など過去のデータを使って精度向上

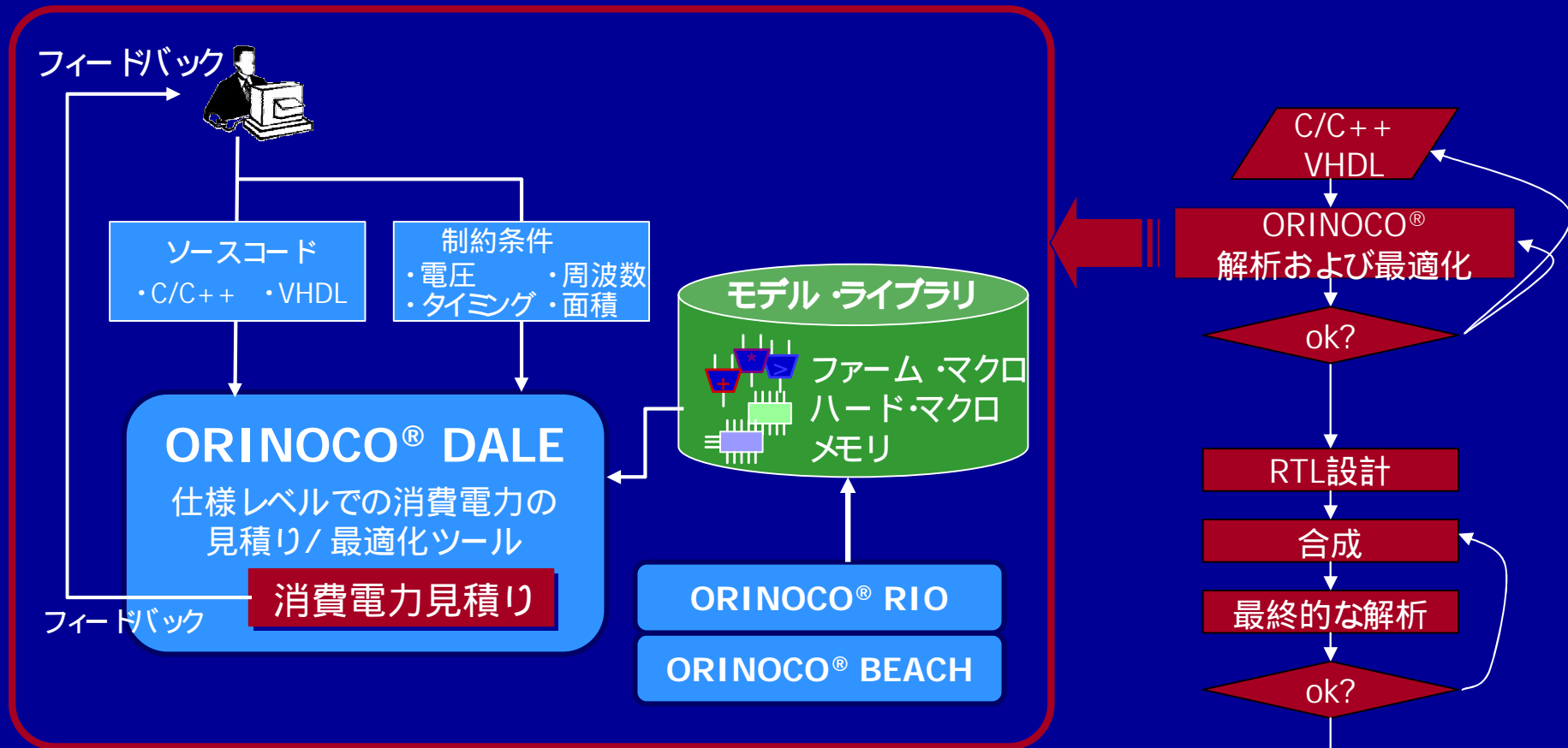
# 「アーキテクチャ決定」における適用検討

## 「アーキテクチャ決定」と見積り技術の関係



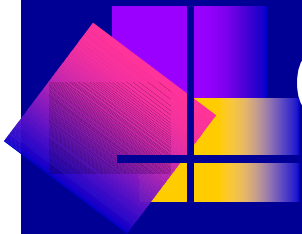
# ORINOCO® のデザインフロー

ChipVision社



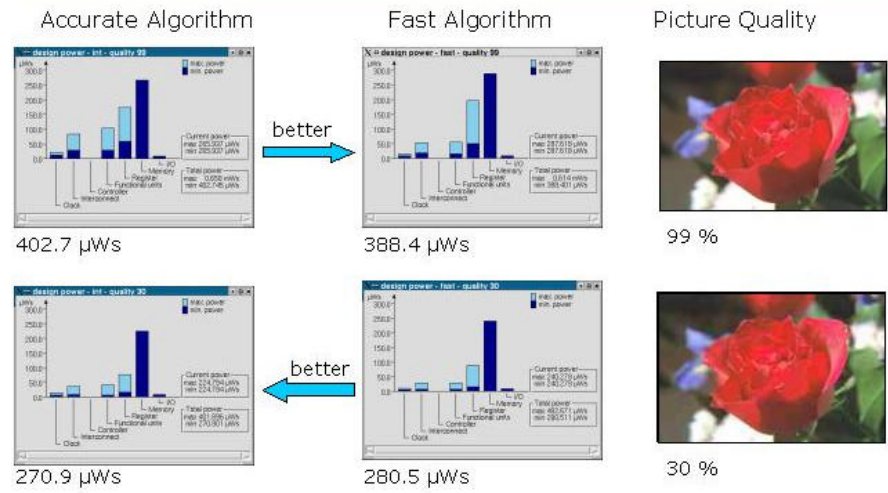
出典：以下のデータを基に作成

- <http://www.chipvision.com/press/index.php>  
(ダウンロード用Presskit中の「Press\_Presentation.ppt」)
- [http://www.lowpower.de/products/ORINOCO-Presentation\\_2\\_02.pdf](http://www.lowpower.de/products/ORINOCO-Presentation_2_02.pdf)



# ORINOCO<sup>®</sup>の適用例

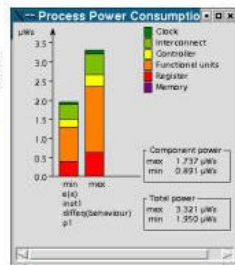
## Algorithm Selection JPEG Decompression



## Algorithm Transformation DIFFEQ Benchmark

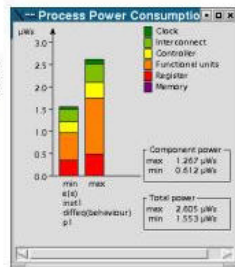
```
original
while ( c>1 )
{
  A = A * c;
  c--;
}
```

FU: 0.891 μWs  
Total: 1.950 μWs



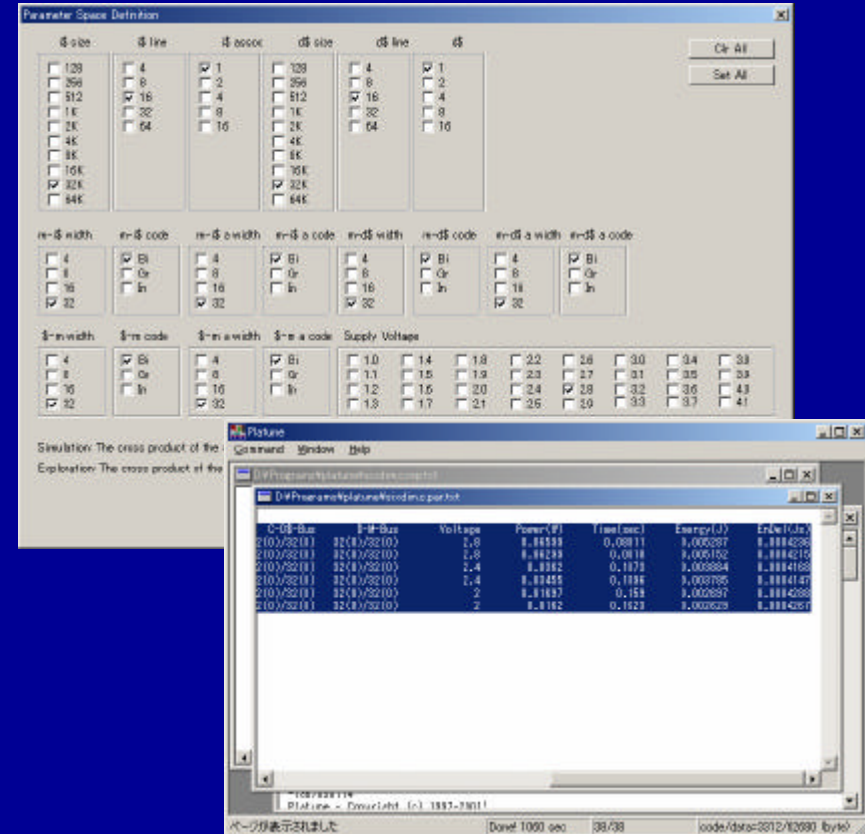
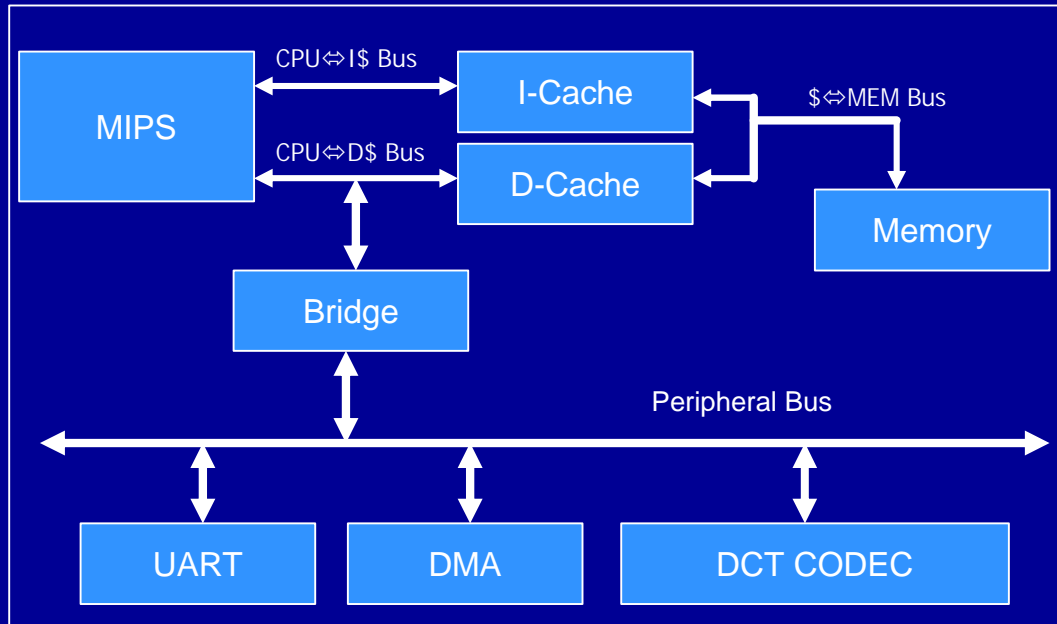
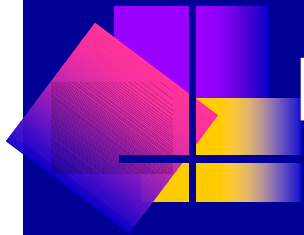
```
modified
if ( c==2 )
  A = A << 1;
else
  while ( c>1 )
  {
    A = A * c;
    c--;
  }
```

FU: 0.612 μWs  
Total: 1.553 μWs



Power savings:  
FU: 31 %  
Total: 20 %

<http://www.chipvision.com/orinoco/>



## ■ プラットフォームの最適構成を探索する

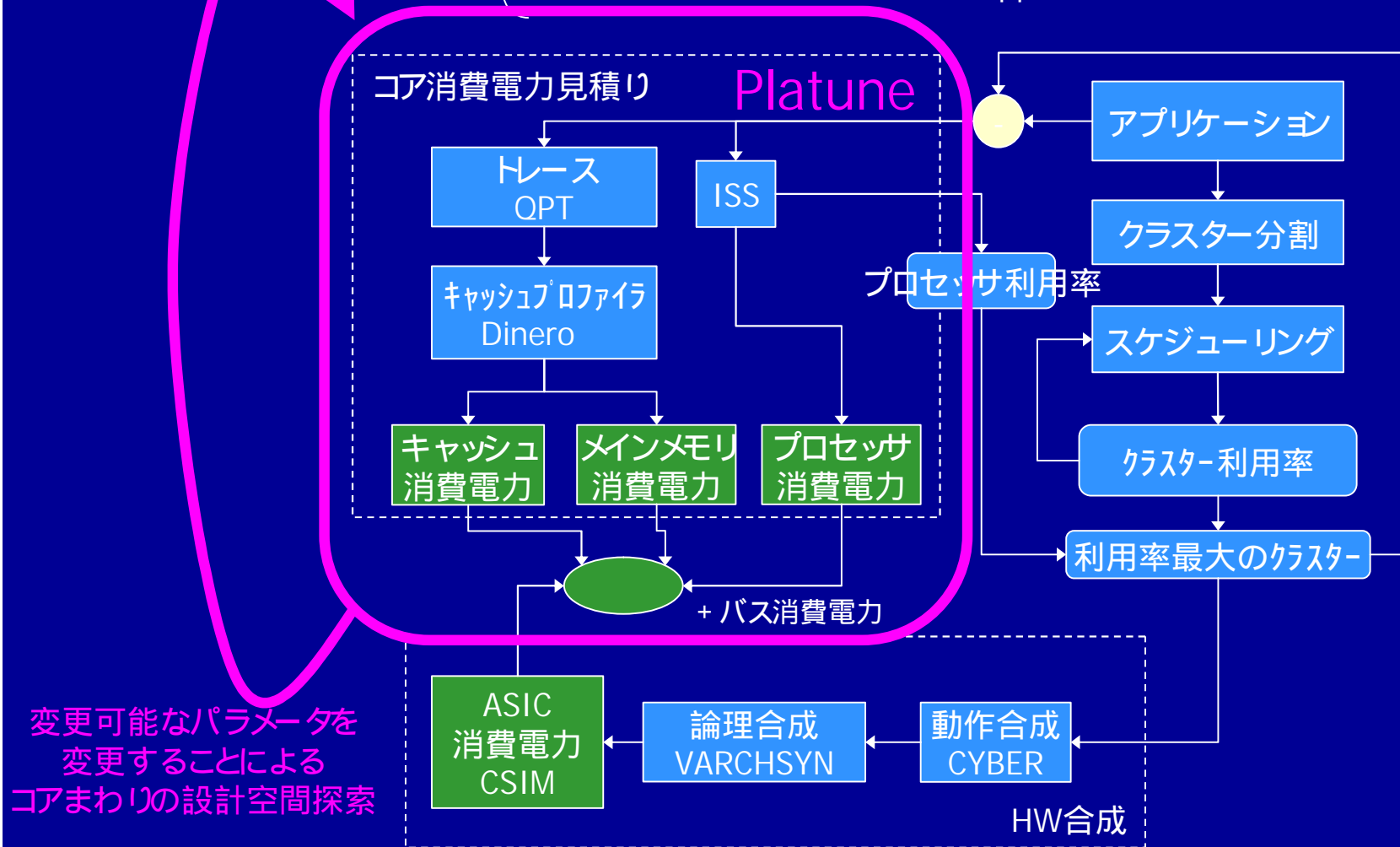
- 入力 : 構成を決定するパラメータの候補値 , アプリケーションSW
- 出力 : 構成毎のPower/Performance値 , 最適な構成 (パラメータ)

## ■ 見積りはPower Modelを利用したシミュレーションに基づく

- シミュレーション速度 : ~50-100K inst/sec, GateSim比 >2000x
- Power 精度 : GateSim比 < 10%

# AvalancheとPlatuneの設計フロー

J.Henkel, Yanbing Li. Avalanche: An Environment for Design Space Exploration and Optimization of Low-Power Embedded Systems. IEEE trans on VLSI Vol10. No.4, pp. 454- 468 AUGUST 2002







# 検討する三つの設計シナリオ

- 現行設計の部分最適化
  - 既存モデルを改良した低電力版モデルを設計する場合などを想定する
  - シナリオ1
    - CPUを変更しないで、既存機能のHW/SW分割もそのままの前提での最適化
  - シナリオ2
    - CPUを変更しないが、既存機能のHW/SW分割を変更する前提での最適化
- プラットフォームの変更設計
  - 現行プラットフォームをベースにして、次期プラットフォームを設計する場合などを想定する
  - シナリオ3
    - CPUの変更も考慮するが、既存機能のHW/SW分割は大幅には変更しない前提での設計



# シナリオ1の課題と検討

- 現行設計の部分最適化 (HW/SW分割は変更しない時)
  - CPU及びCPU周辺部の最適化・見積り
    - CPU動作周波数・キャッシュサイズ・内部バス幅最適化  
Platuneを使用する
      - CPU電圧 (MIPS)、キャッシュサイズ、内部バス最適化が可能で、プラットフォーム設計なので事前に見積り最適化環境を準備できる
  - HW/SW間のI/F (バス、共有メモリ)を含めたシステム全体の見積り
    - 今回調査したツールでは実現が難しい
    - 課題 :電力データベースを含んだRTLに近いモデルでシミュレーションする以外に良い方法はないか？  
解決策：  
「Platune」の拡張を考える。カスタムHWモデルにパワーの属性を加えたスタブを使用する。ただし本スタブを作成しておく必要はある



# シナリオ2の課題と検討

- 現行設計の部分最適化 (HW/SW分割も変更する場合)
  - 分割後の最適化・見積りに対しては、シナリオ1の方法を使用すれば良い
  - 課題 :カスタムHW部のスタブ (電力属性込み)を分割変更後すぐに準備できるか  
解決策：
    - HW部が減る場合 (HW部 SWとする場合)
      - カスタムHW部の電力属性の算出を機能単位で行っておき、単純な引き算で生成できるようにしておく
    - HW部が増える場合 (SW部 HWとする場合)
      - SW部からの電力属性を求める際は、ORINOCOを使用して算出する
  - 尚、シナリオ2は新規機能を追加した場合も同様である



# シナリオ3の課題と検討

## ■ プラットフォームの変更

### ■ CPUの選択・CPU周辺部アーキテクチャ決定

- 最適なCPUの選択 (XX命令が低電力で実行可能なものを選択) に対して、今回調査したツールでは対応はできない
- CPUが決まれば、Platuneを使用してパラメータの最適化ができる

### ■ バスアーキテクチャの選択

- 同一バスプロトコルでパラメタライズされた範囲内での比較は行えるが、異種バスアーキテクチャ間での比較検討に対しては、今回調査のツールでは対応できない

### ■ 課題

- 見積りモデルについてプロセッサベンダが対応できる部分とユーザも含めて対応すべき部分があり、事前に消費電力見積りモデルを準備しておくことが難しい

解決策：

簡易モデルを使ってまず選別する手法が考えられるが、具体策は難しい



# 提案と課題のまとめ

## 調査内容

- 論文調査 (DAC、ICCAD、CODESなど)
- 低消費電力設計技術、消費電力見積り手法の分類
  - 静的と動的 (シミュレーション) アプローチがあるが、システムレベルでは動的的手法による見積りが多い
  - 動作レベル記述から消費電力を見積る技術の研究例は多いが、実設計での成功例は少ない
    - 実用上の課題がどこにあるかが見えない

## 提案 / 主張 / 課題

- SLD研究会が提案するシステムレベル設計フローの上で、消費電力見積り技術の適用を想定して、課題と解決策を検討した

### (1) 機能決定」における見積り技術の適用

- ATOMIUMによる最適化を中心に検討した
  - 入力ソースコード(C言語)を、メモリ等のリソース利用を最適化したコードへ変換
- 相当な効果が期待できるが、HWとして実装した場合の最適性には注意が必要
- 低消費電力化が可能だが、消費電力の絶対値の見積りへの利用に課題



# 提案と課題のまとめ (つづき)

## (2) 「アーキテクチャ決定」における見積り技術の適用

- 設計変更のシナリオごとでの消費電力見積り技術の適用を検討した
- 既存ツールの活用 (AvalancheとPlatuneの併用など)により一定のレベルの最適化は期待できる
- CPU及びCPU周辺部 (動作周波数、キャッシュサイズ、内部バス幅) の最適化
  - Platuneが適用できる
  - HW/SWのインタフェースを含めた最適化は今後の課題
- HW/SW分割の変更も含めた最適化
  - カスタムHW部の見積り精度が重要になる
  - ORINOCOのようなHW部の消費電力見積り最適化と、PlatuneのようなCPU周りの見積もり最適化の併用が必要
- CPU、バス・アーキテクチャなどのプラットフォームの選択の最適化
  - 各プラットフォームごとに見積りモデルの準備が必要
  - モデルの開発を、プロセッサベンダとユーザのいずれが対応すべきか境界が不明

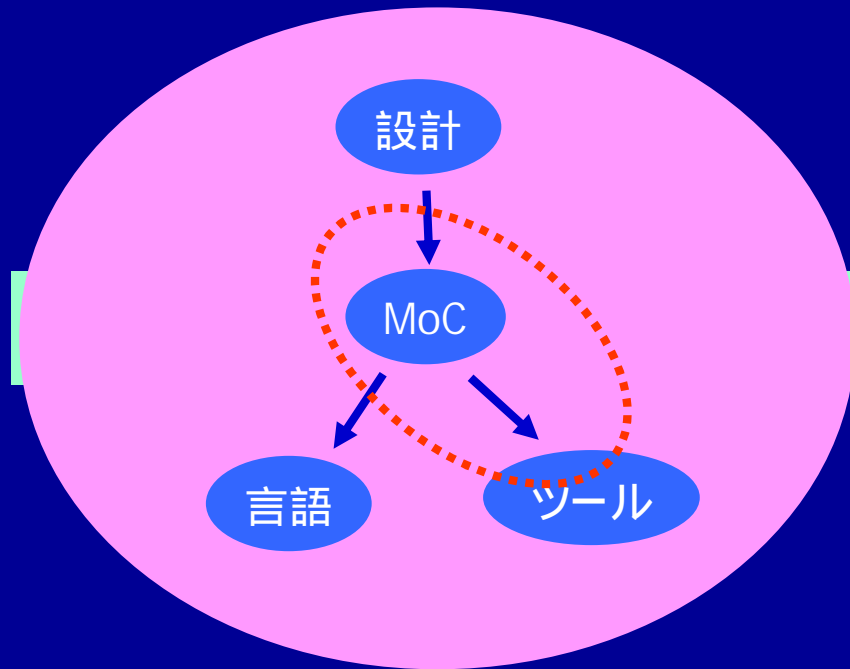


# 提案と課題のまとめ (つづき)

## 研究機関 / EDAベンダ / 標準化団体への期待

- システムレベルでの消費電力見積りの精度を高めることは難しいので、見積もり技術の応用手法の開発に注力して欲しい
  - たとえば、以下のアプローチ
    - 消費電力最適化の観点でアーキテクチャ候補間でのトレードオフを見る
    - 流用設計の場合はリファレンスデータを利用して見積りの精度を高める
    - 上流設計で得た見積り情報を下流設計でも活用する、あるいは、下流設計で行った見積り上流設計にフィードバックする
- 実設計との比較 評価を通して、見積り精度の向上にも注力して欲しい
- ベンダとユーザの関係においては、見積りモデルあるいはDBの整備・開発を、誰が対応すべきかの境界を明確にしてゆくことが期待される

# むすび



各MoCの設計フロー適用可能性を検討

~~モデリングガイドライン、  
設計適用検討には至らず~~



# むすび

要求仕様定義

~~要求仕様定義から機能決定の  
インタフェースは未検討~~

機能決定

計算モデル (MoC) の分析

アーキテクチャ決定

消費電力見積り技術の適用検討

実装設計 (RTL) への  
インタフェース

目的別の検証技術 (HW/SW協調検証) 提案  
設計の自動化 (動作合成) の調査  
HW検証品質 (アサーションベース検証) 調査

~~実証実験には至らず~~



# むすび (つづき)

## 今後実現を期待する技術についてまとめる

- システムレベル設計フロー実現のための基盤技術開発
  - (1) **要求仕様定義から機能決定に至る設計手法の開発**
  - (2) プロファイリング技術の中核とする**アーキテクチャ設計ツールの開発**
  - (3) **動作レベルのフォーマル検証技術の開発**
- モデリング技術開発
  - (4) 設計抽象度の定義とMoCをベースとした**モデリングガイドライン作成**
  - (5) インタフェースモデルの定義と**インタフェース生成技術の開発**
- 合成および見積り技術強化
  - (6) 動作合成用モデルの記述容易性のための改善と**並列性抽出などの自動化強化**
  - (7) **低消費電力設計用最適化ツールの開発**と消費電力見積り技術の精度問題対応