「形式手法と仕様記述」学習チーム報告

有賀　一輝　　伊澤　崇史　　小田部　健
羽田　裕　　平野　純子　　水野　徹平
山村　繁行　　和田　圭司

発表目次

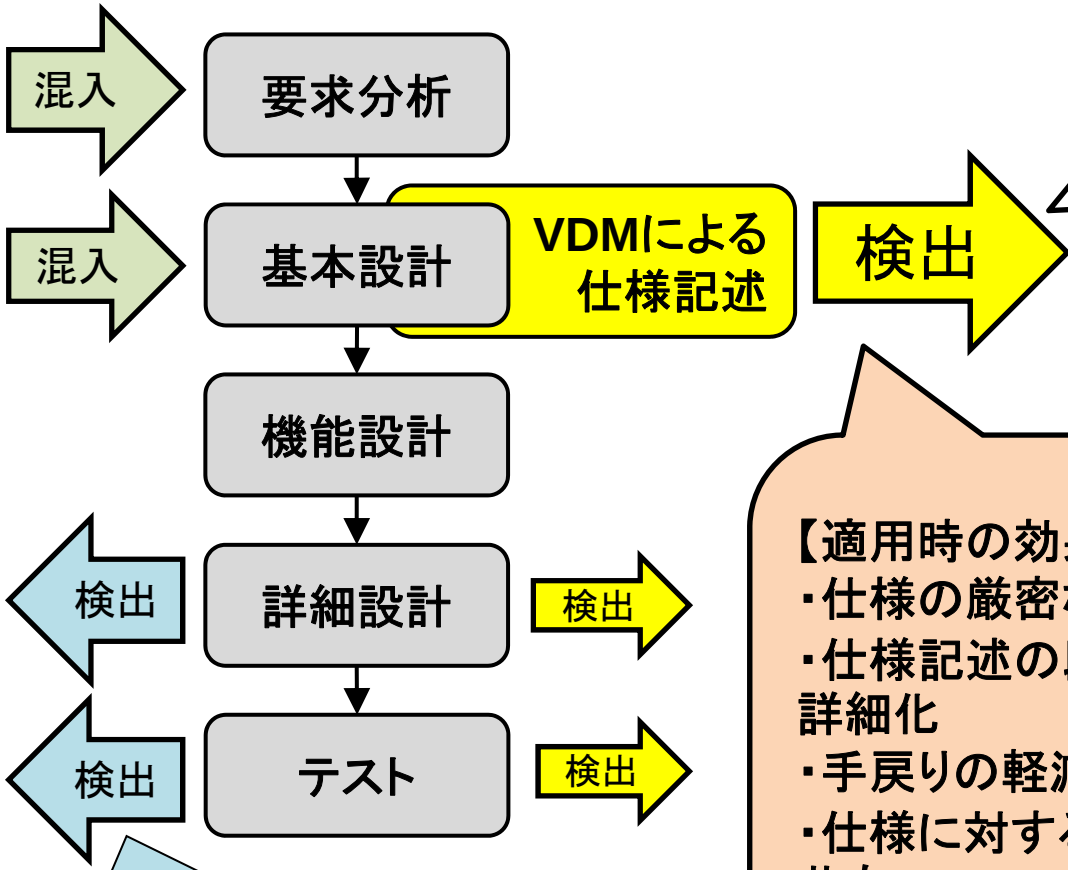
「学習」組

形式仕様基礎（VDM） 「動かしてみる」～共通基礎学習	小田部
モデル検査（NuSMV） 「網羅的に探索，検証する」～教育コース作成	有賀・羽田・和田
実時間モデル検査（UPPAAL） 「時間制約を・・・（同上）」～入門例題集作成	平野
モデル発見（Alloy） 「例を洗い出す」～テスト生成問題に試用	水野

VDMの概要

- 形式手法の一つで、Vienna Development Method（ウィーン開発手法）の略称
- 自然言語に比べて厳密な記述が可能
- 利用例としては仕様を厳密に記述することによる、上流工程からのソフトウエア品質向上が挙げられる
- 仕様（データや操作など）を記述する際、仕様の抽象度に応じた記述が可能である
- 記述ルールがプログラミング言語に近く、ソフトウェア開発者にとって比較的馴染みやすい
- ツールや関連書籍が揃っている

ソフトウェア開発への適用例



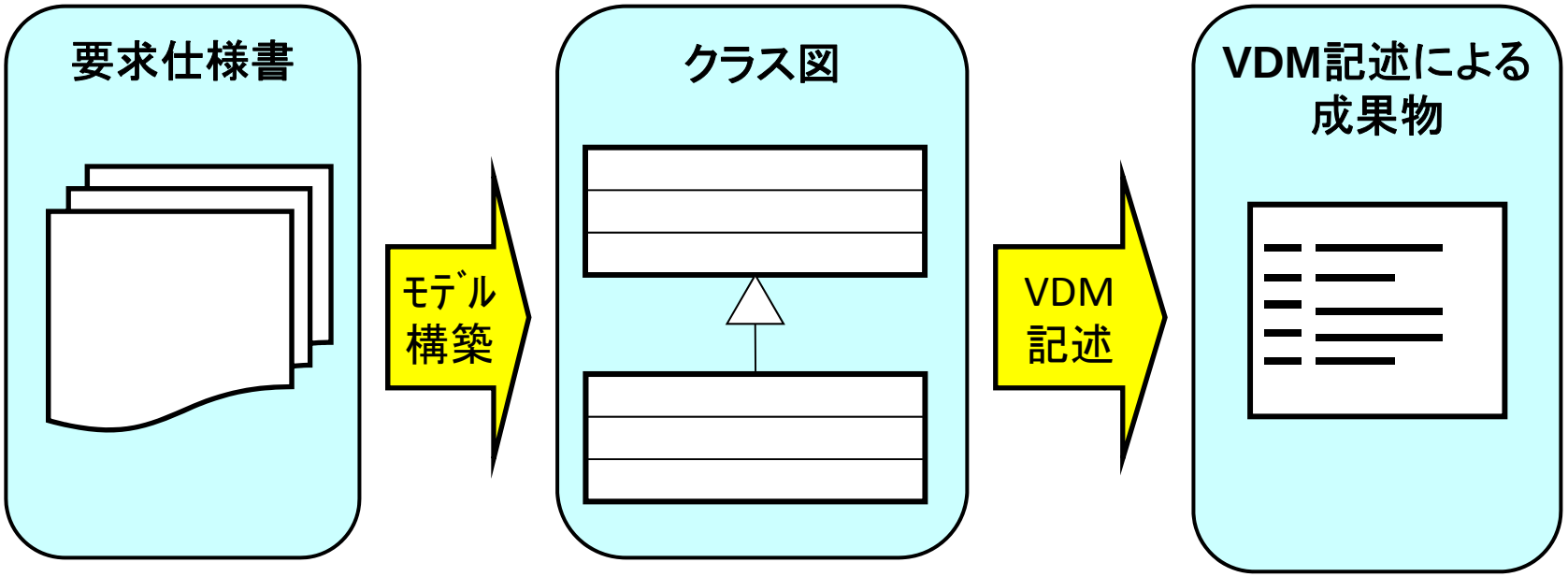
仕様をVDMにより厳密な形式で記述することで、早期の検証が可能になる

【適用時の効果】
・仕様の厳密な記述
・仕様記述の段階的な詳細化
・手戻りの軽減
・仕様に対する理解の共有
・暗黙知の見える化
・テストケースの生成

【必要な知識】
◇VDM本体
・数学的知識
・記述ルール
◇モデル構築
・モジュール設計
・クラス設計
・抽象化
◇モデル対象
・ドメイン知識
・利用環境

【非適用時の課題】
・後工程での不具合検出による手戻りの発生

VDM記述のプロセス例



- 要求仕様書などの成果物をVDMで記述する場合、一般的な開発と同様の作業（**モデル構築**など）が必要になる
- モデル構築には**機能的振る舞いの分析**や記述対象への理解，**抽象化**の能力などが必要になる
- 事前条件、事後条件など、VDMの記述に必要な**情報の抽出**も必要になる

VDMで記述し直す過程で、仕様間の**不整合の発見**、**暗黙知の明確化**、関係者間の**認識齟齬の解消**などの効果も期待できる

【勘違い】
成果物から直接VDMによる記述が可能だと思っていた

発表目次

「学習」組

形式仕様基礎（VDM） 「動かしてみる」～共通基礎学習	小田部
モデル検査（NuSMV） 「網羅的に探索，検証する」～教育コース作成	有賀・羽田・和田
実時間モデル検査（UPPAAL） 「時間制約を・・・（同上）」～入門例題集作成	平野
モデル発見（Alloy） 「例を洗い出す」～テスト生成問題に試用	水野

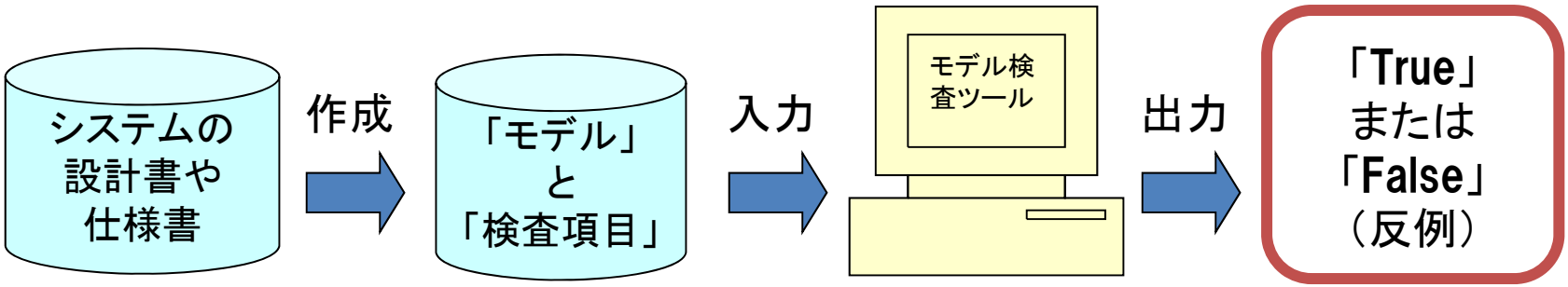
デルモチーム　有賀・羽田・和田

モデル検査とは？

システムのモデルを作成し，モデルが検査項目を満たすかどうかを，システムが取り得る全ての状態に対して検査を実施することによって不具合を発見する技術です．検査は，モデル検査ツールにモデルと検査項目を与えることにより自動で行われます．モデルが検査項目を満たさない場合は，反例（どのような状況で満たされないか）が出力されます．

モデルとは？

対象システムの振る舞い（状態遷移表など）を，モデル検査ツール専用の言語で記述したものです．モデルを作ることで，検査対象の振る舞いについて，システムが取り得る全ての状態に対して網羅的に検査が行えます．このため限定されたテストパターンに依存することなく，状態の組み合わせの不具合などを見つけることが可能です．



モデル検査

- モデル検査ツールには，どのようなものがあるか？

  代表的なものとして，「SMV」，「SPIN」，「UPPAAL」があります．これらには，モデルを記述する言語や，検査項目を記述する際の論理式の体系，などの違いがあります．ここでは，SMVコードと呼ばれる専用言語でモデルを記述し，LTL（線形時相論理）やCTL（計算木論理）などの論理体系で検査項目を記述するモデル検査ツール SMVについて紹介します．

- モデル検査ツール SMVの種類は？

  SMV（Symblic Model Verifier）は，シンボリックな手法を最初に用いたモデル検査ツールとして1992年に登場しました．今では以下の3つがあります．

  - CMU SMV

    90年代初めにCarnegie Mellon大学で開発．状態遷移機械によるモデルとCTLで書かれた検査項目を入力とする．

  - NuSMV

    イタリアの大学や研究機関により，CMU SMVを再実装し，拡張したもの．民間企業での営利目的の製品設計への使用が許可されている．

  - Cadence SMV

    Cadence社で開発されたツール．教育などの非営利利用が許可されている．

モデル検査の学習

著者：産業技術総合研究所
モデル検査を早くから研究し普及を図っていた
独立行政法人

本の構成
解説と多くの例題と課題、演習問題で構成

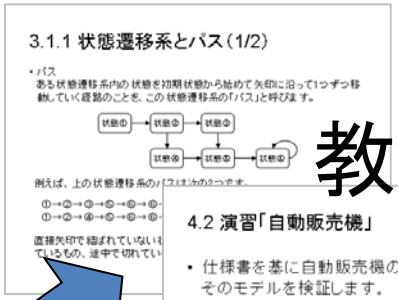
大きく4部構成になっていて、1部を1日でこなして、全部で4日で終了するコース

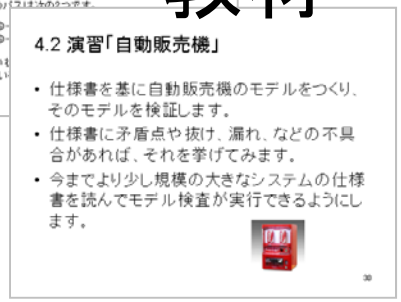
４日かかるところ
この本がなくてもを半日で学べるようにしました！

教育コースの作成

ARCS動機づけモデル

教材



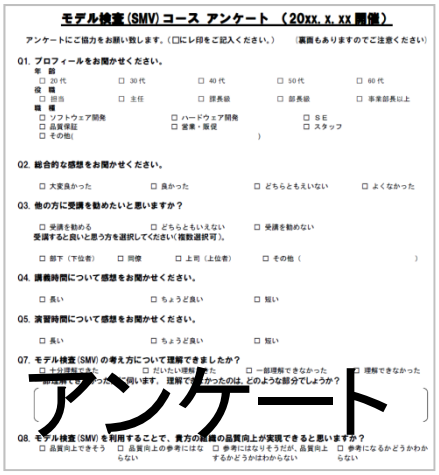


学習教材

教育コース設計書

ガニエの9つの教授事象

ガニエの9つの教授事象		
導入	G1	Gain Attention:学習者の注意を喚起する/情報の受け入れ態勢を作る
	G2	Inform Learners of the Objectives:学習者に目標を知らせる/頭を活性化し，重要な情報に集中させる
情報提示	G3	Stimulate Recall of Prior Learning:前提知識を思い出させる/関連事項を思い出す
	G4	Present the Stimulus:新しい事項を提示する/[illegible]学ぶ内容を頭に知らせる
	G5	Provide Learner Guidance:学習の指針を与える/意味のある形で頭にいれる
学習活動	G6	Elicit Performance:練習の機会をつくる/頭から取り出す練習をする
	G7	Provide Feedback:フィードバックを与える/学習状況をつかみ，弱点を克服する
まとめ	G8	Assess Performance:学習の成果を評価する/成果を確かめ，学習結果を味わう
	G9	Enhance Retention and Transfer:保持と転移を高める/長持ちさせ，応用がきくようにする



アンケート

発表目次

「学習」組

形式仕様基礎（VDM） 「動かしてみる」～共通基礎学習	小田部
モデル検査（NuSMV） 「網羅的に探索，検証する」～教育コース作成	有賀・羽田・和田
実時間モデル検査（UPPAAL） 「時間制約を・・・（同上）」～入門例題集作成	平野
モデル発見（Alloy） 「例を洗い出す」～テスト生成問題に試用	水野

UPPAALとは

時間に関する要件の検証

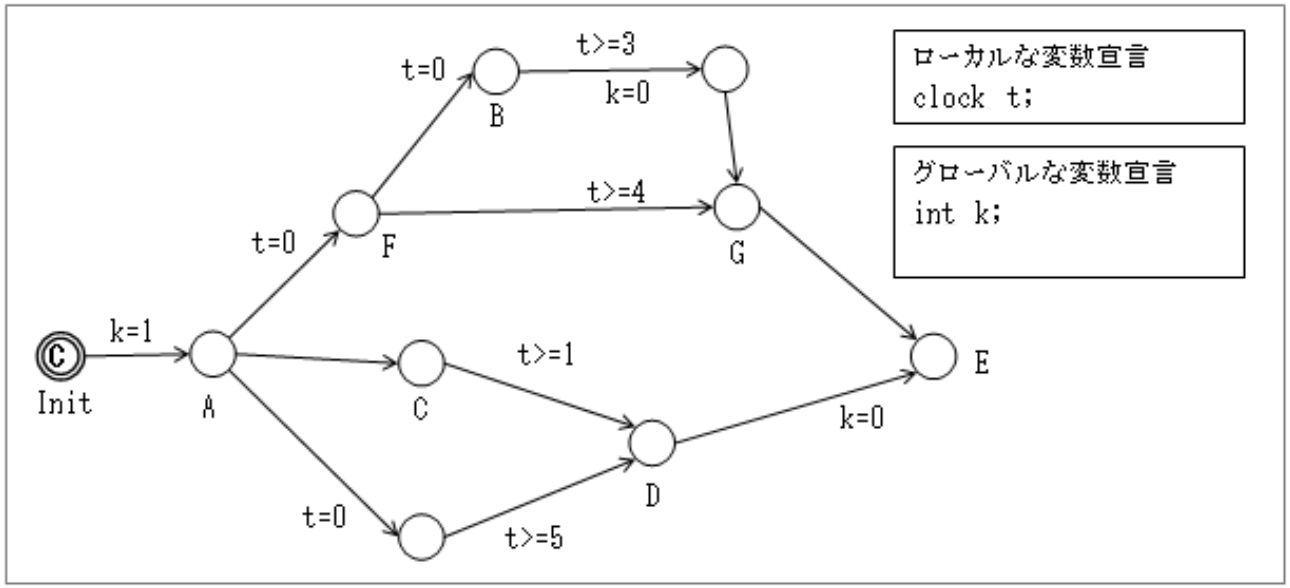
- デッドロック等のタイミングにまつわる様々な問題
- 要求された処理が所定の時間内に終わるかどうか

➡ 問題をレビューやテストで検出することは非常に難しい

UPPAALとは

- Uppsala UniversityとAalborg Universityが共同で開発
- リアルタイムシステムのモデリング，シミュレーションおよび検証を行うためのツール
- 時間の可能性を網羅的に検査できる
  - 性質が必ず成り立つことを理論的に保証する
  - 成り立たない場合は、反例を提示する
- 検査するモデルは、時間オートマトンでモデル化
  - GUI操作で簡単に記述できる
- 検証したい性質は時相論理CTL(Computation Tree Logic)の式として記述

UPPAALのシステムモデルと検証式



問　上のモデルで次の検証式は成り立つか？

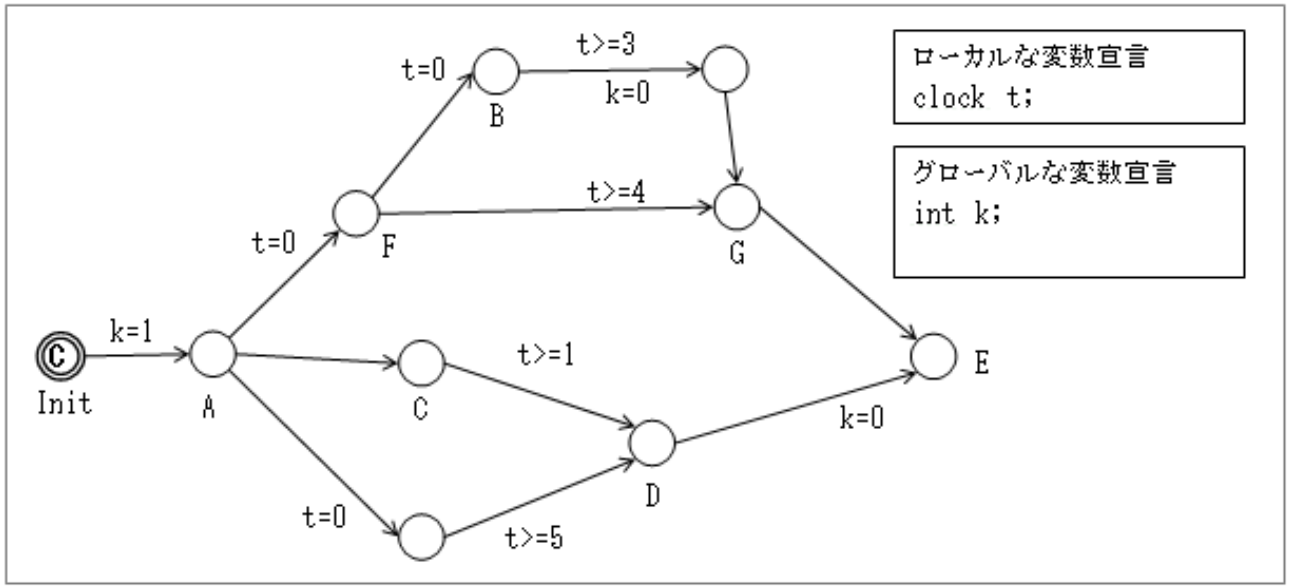
① E<> P.B （あるパスでいつかはBに到達する）

② E<> P.C （あるパスでいつかはCに到達する）

③ A<> P.E （全てのパスでいつかはEに到達する）

④ A<> k==0 （全てのパスでいつかはkの値が0になる）

UPPAALのシステムモデルと検証式



問　上のモデルで次の検証式は成り立つか？

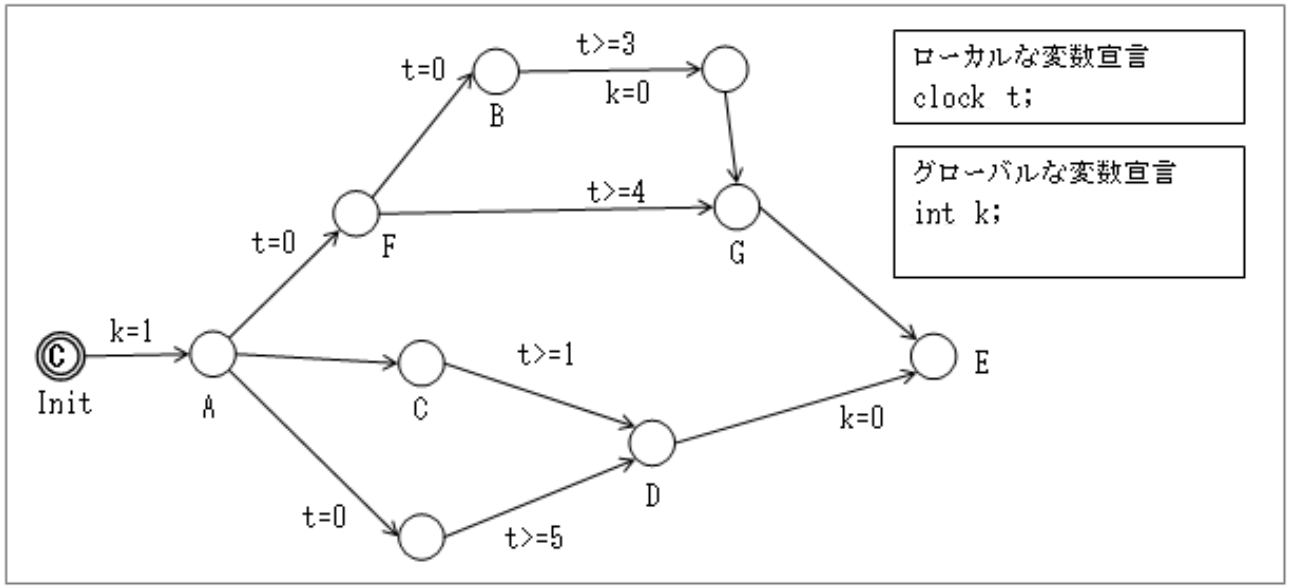
① E<> P.B （あるパスでいつかはBに到達する）　：真?

② E<> P.C （あるパスでいつかはCに到達する）　：真?

③ A<> P.E （全てのパスでいつかはEに到達する）　：真?

④ A<> k==0 （全てのパスでいつかはkの値が0になる）　：偽?

UPPAALのシステムモデルと検証式



問　上のモデルで次の検証式は成り立つか？

① E<> P.B （あるパスでいつかはBに到達する）　：真

② E<> P.C （あるパスでいつかはCに到達する）　：真

③ A<> P.E （全てのパスでいつかはEに到達する）　：偽

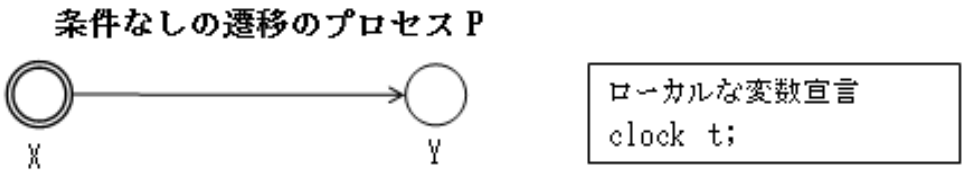
④ A<> k==0 （全てのパスでいつかはkの値が0になる）　：真

オートマトンと時相論理式の意味論を正しく理解していないと，勘違いや間違いによって正しい検証が行えない

時間オートマトンと時相論理式の理解

極小規模なモデルを作成し，その性質を検証式で表現し、作成したモデルが，意図通りのものとなっているかを検証することにより，勘違いや間違いの要素を排除する

例



検証式	結果
A[] P.X	:偽
A<> P.Y	:偽
E[] not P.Y	:真
E<> P.Y	:真
E<> P.X and P.Y	:偽
A[] P.X or P.Y	:真
A[] P.t==0 imply P.X	:偽
P.t==0 --> P.X	:偽

発表目次

「学習」組

形式仕様基礎（VDM） 「動かしてみる」～共通基礎学習	小田部
モデル検査（NuSMV） 「網羅的に探索，検証する」～教育コース作成	有賀・羽田・和田
実時間モデル検査（UPPAAL） 「時間制約を・・・（同上）」～入門例題集作成	平野
モデル発見（Alloy） 「例を洗い出す」～テスト生成問題に試用	水野

「研究」組

「日常」（USDM）との融合を模索する	日下部・宮本

1．Alloyとは

一般的なモデル検査ツールであるSPINなどが，仕様の正しさに対する網羅的な検査をするのに対し，Alloyは特定の範囲に絞ってとりうるパターンを網羅的に列挙するする．そのため正しさの証明はできないが，テストで行うよりはるかに膨大なパターンを検証できる．

例題：木構造を表す仕様

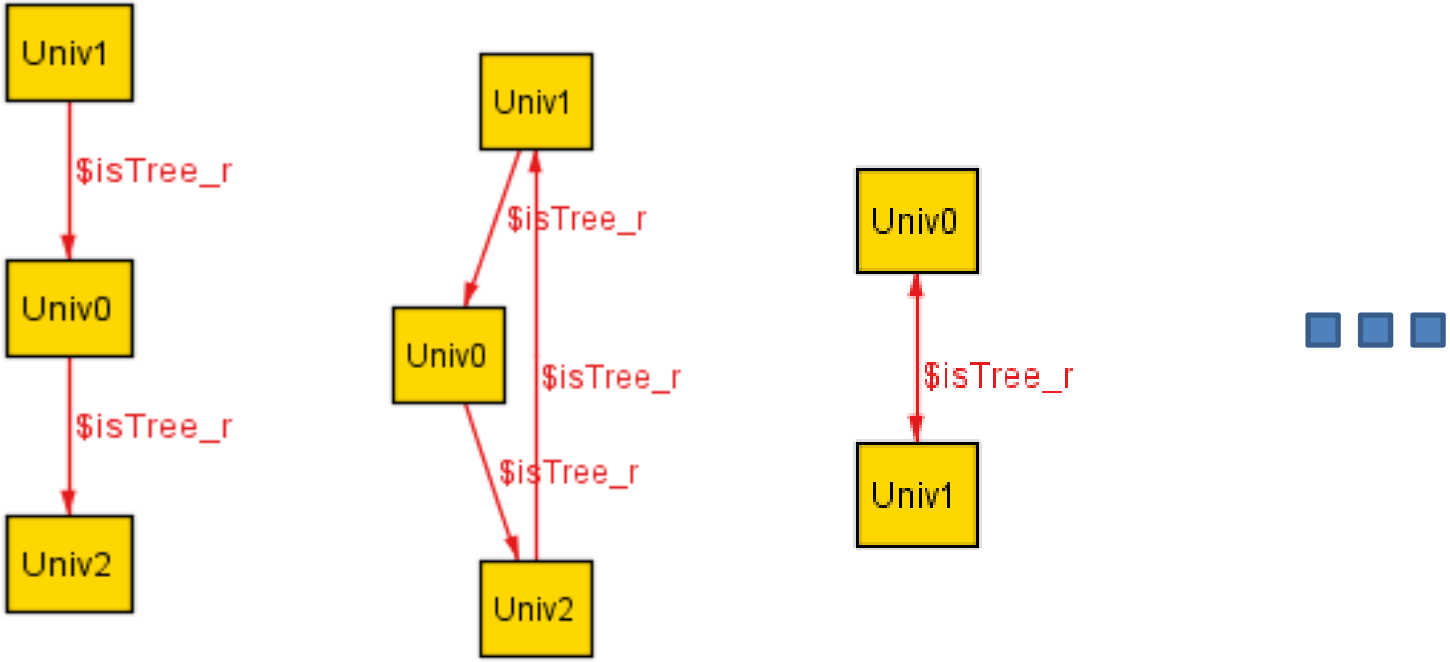
```
pred isTree [r:univ->univ] {}
run isTree for 3
```

自己ループが出てくるので、木の仕様として正しくないことが分かる

1. Alloyとは

自己ループを禁止する仕様を追加する

```
pred isTree [r:univ->univ] {
        no(r & iden)
}
run isTree for 3
```

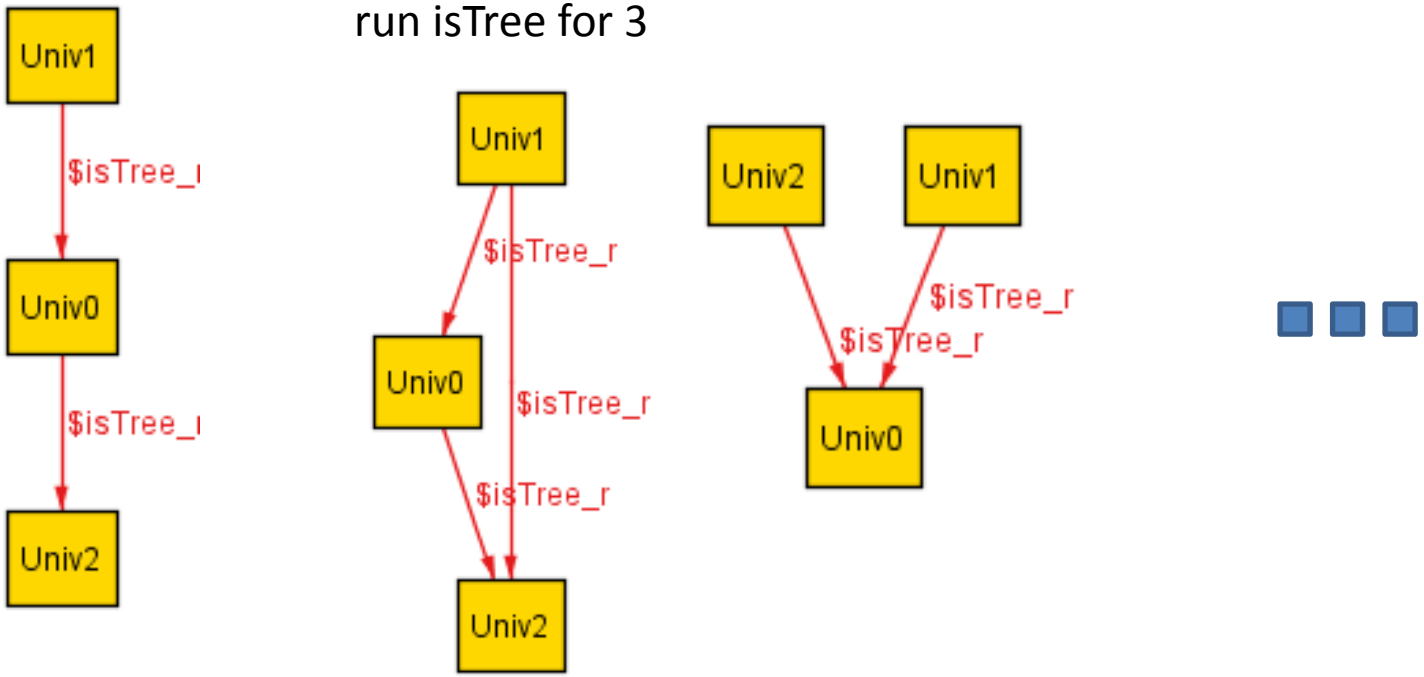


自己ループは無くなったが、サイクルのパターンがでてきてしまう

1. Alloyとは

サイクルを禁止する仕様を追加する

```
pred isTree [r:univ->univ] {
        no(^r & iden)
}
run isTree for 3
```



サイクルは無くなったが、親が複数のノードがまだ出てきてしまう。。。（以下略）

仕様が正しく、十分でないと、期待しない結果がでる。

2. 目的と結果

- 正しい仕様かどうか検証したい

  このような状態にはなりえないの？
  なっても回避する方法がかならずあるの？
  といった質問に、自信を持って大丈夫です！とはなかなかいえない
  しかし、“証明”することは難しい。
  そこで、alloyを試してみた。

- 具体的には

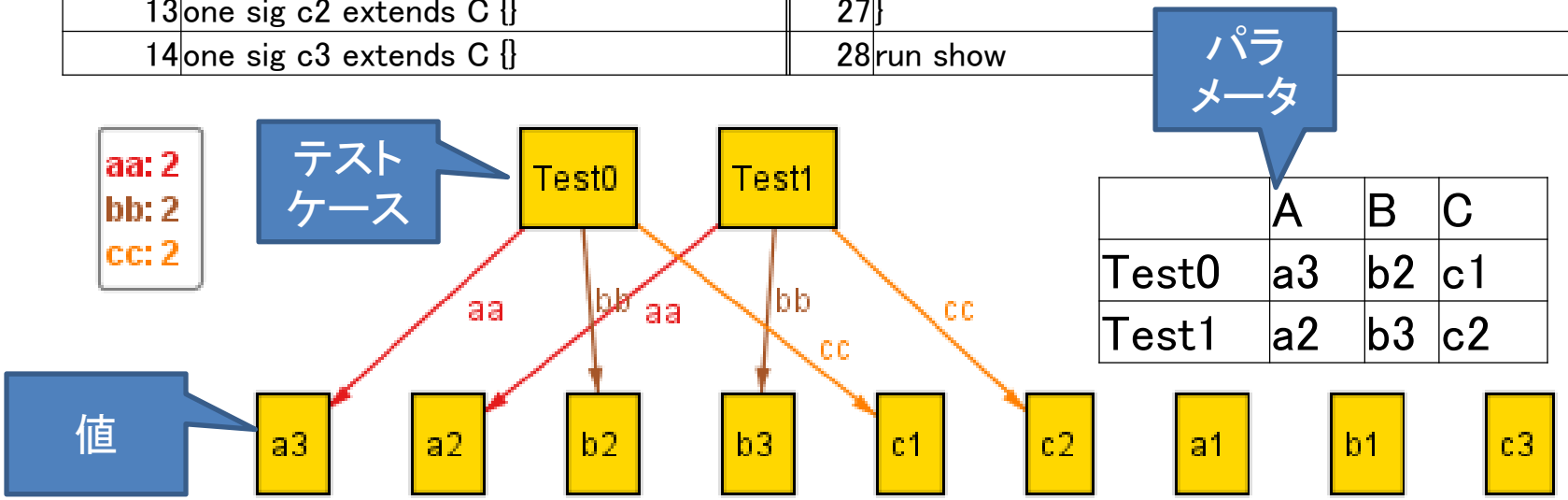
  テストケースを作成するアプリケーションにおいて、ユーザーの意図しない結果がでることが無いよう。Alloyで現在の仕様を記述し、さまざまな出力パターンを確認する。

- 結果

  仕様のすべてを記述はできなかったが、
  仕様の一部をAlloyで記述し、出力パターンの確認を行った。
  残りの仕様も学習が進めば記述できると考えられる。

3. 形式記述と実行結果

```
1  abstract sig A{}
2  one sig a1 extends A {}
3  one sig a2 extends A {}
4  one sig a3 extends A {}
5
6  abstract sig B {}
7  one sig b1 extends B {}
8  one sig b2 extends B {}
9  one sig b3 extends B {}
10
11 abstract sig C{}
12 one sig c1 extends C {}
13 one sig c2 extends C {}
14 one sig c3 extends C {}
15
16 sig Test{
17 aa: one A,
18 bb: one B,
19 cc:  one C
20 }
21 {
22 not (aa = a1 and bb = b2)
23 not (aa = a1 and bb = b2)
24 not (aa = a1 and bb = b3)
25 }
26 pred show{
27 }
28 run show
```



パラメータ

	A	B	C
Test0	a3	b2	c1
Test1	a2	b3	c2

4. Alloyの感想

1. 仕様を部分的に表現した段階で検査し，不都合な結果があれば仕様を追加していくといったやり方を反復する書き方が可能である．この試行錯誤の方法は一般的な開発者にはなじみやすく，また仕様の不具合を見つける手法を，記述しながら練習することができる．

2. 仕様を部分的に記述した中途半端な状態でもパターンを列挙できるので，少しずつ不足する仕様を検証しながら追加していく作業には向いていると考えられる．

3. 言語など手続き型と違い，すべてを関係として宣言的に記述するなっているため，発想の転換が必要．