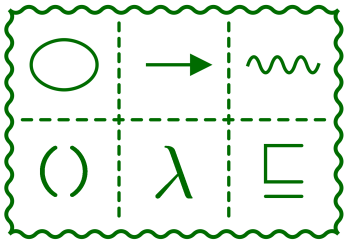


ソフトウェア開発を工学にする

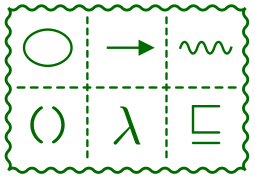
形式手法の紹介

Hisabumi HATSUGAI

hatsugai@principia-m.com

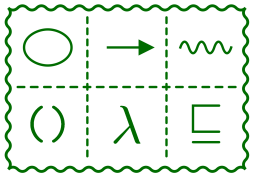
PRINCIPIA Limited

http://www.principia-m.com/

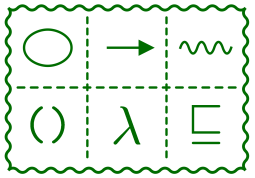


もくじ

- ソフトウェア開発は工学なのか
- 形式手法とは何か
- ソフトウェアの 2 つの側面：計算と相互作用
- 相互作用に力点を置くリアクティブシステム
- ソフトウェアの仕様を厳密に記述する
- ソフトウェアが仕様を満たす条件とは
- ソフトウェアの正しさを数学的に証明する
- リアクティブシステムの仕様と検証
- 並行・並列システム開発の難しさと対応



ソフトウェア開発は工学なのか？

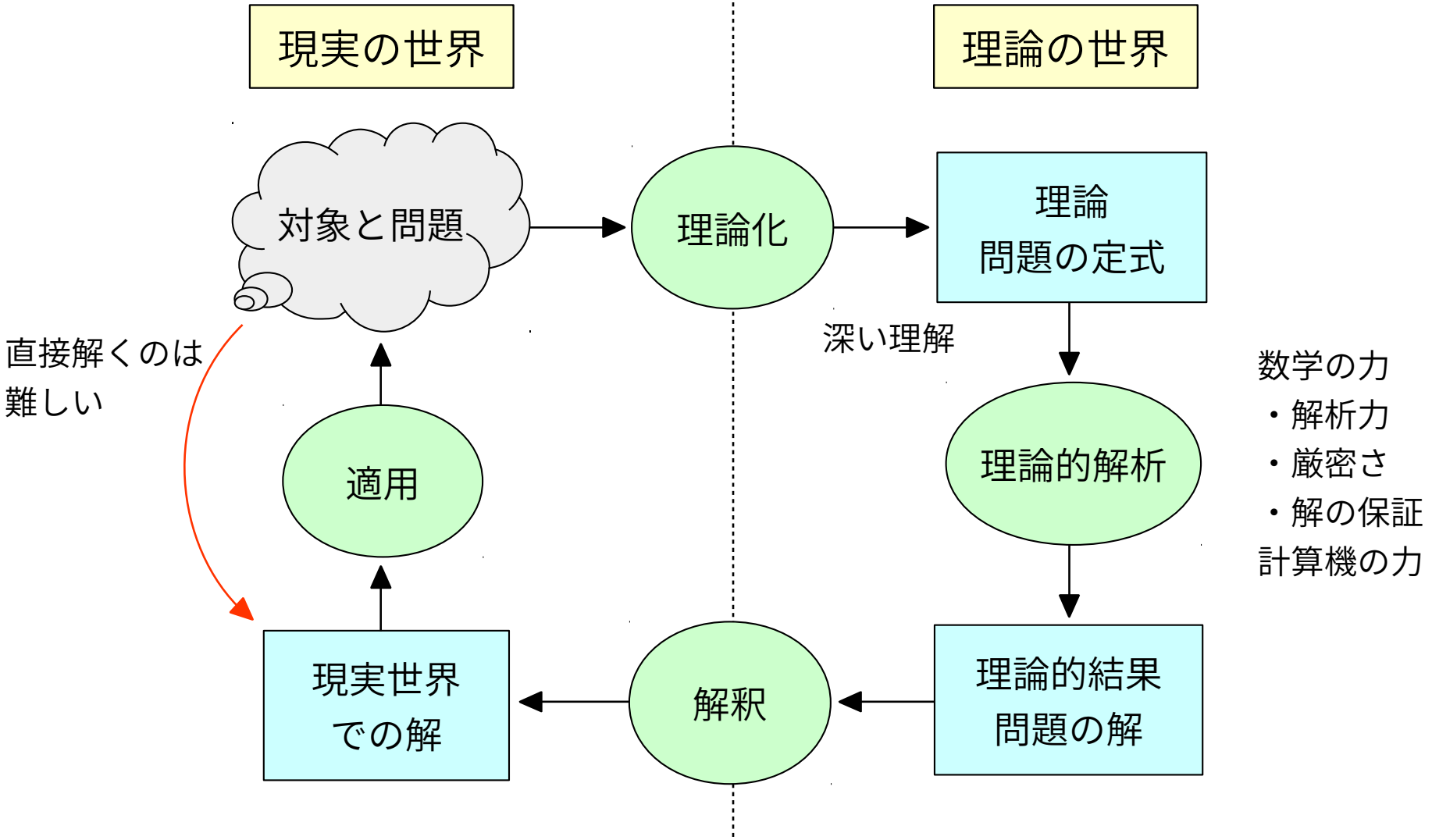


工学とは？

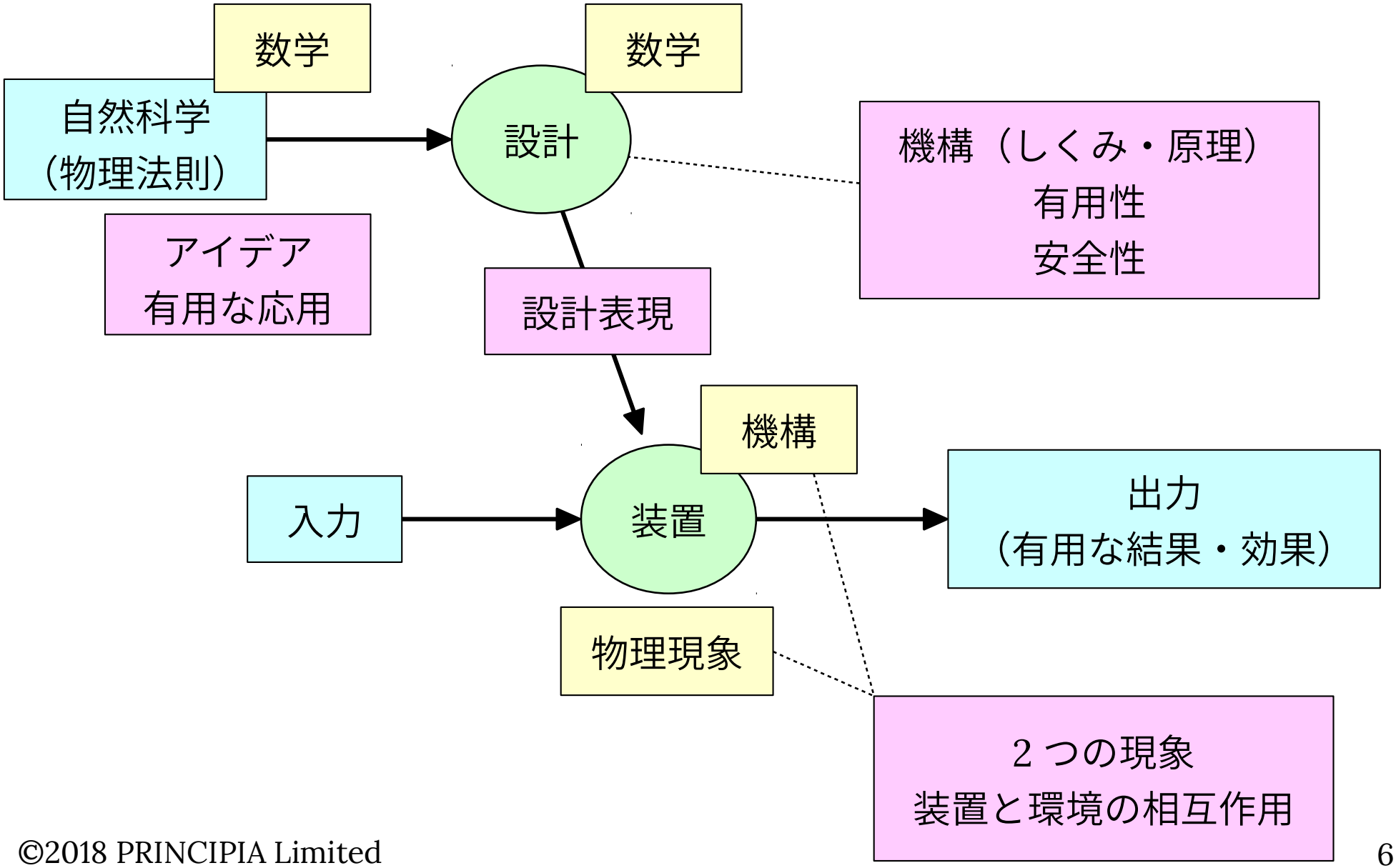
工学とは数学と自然科学を基礎とし、ときには人文社会科学の知見を用いて、公共の安全、健康、福祉のために有用な事物や快適な環境を構築することを目的とする学問である。

「８大学工学部を中心とした工学における教育プログラムに関する検討」, 工学における教育プログラムに関する検討委員会，平成 10 年 5 月 8 日

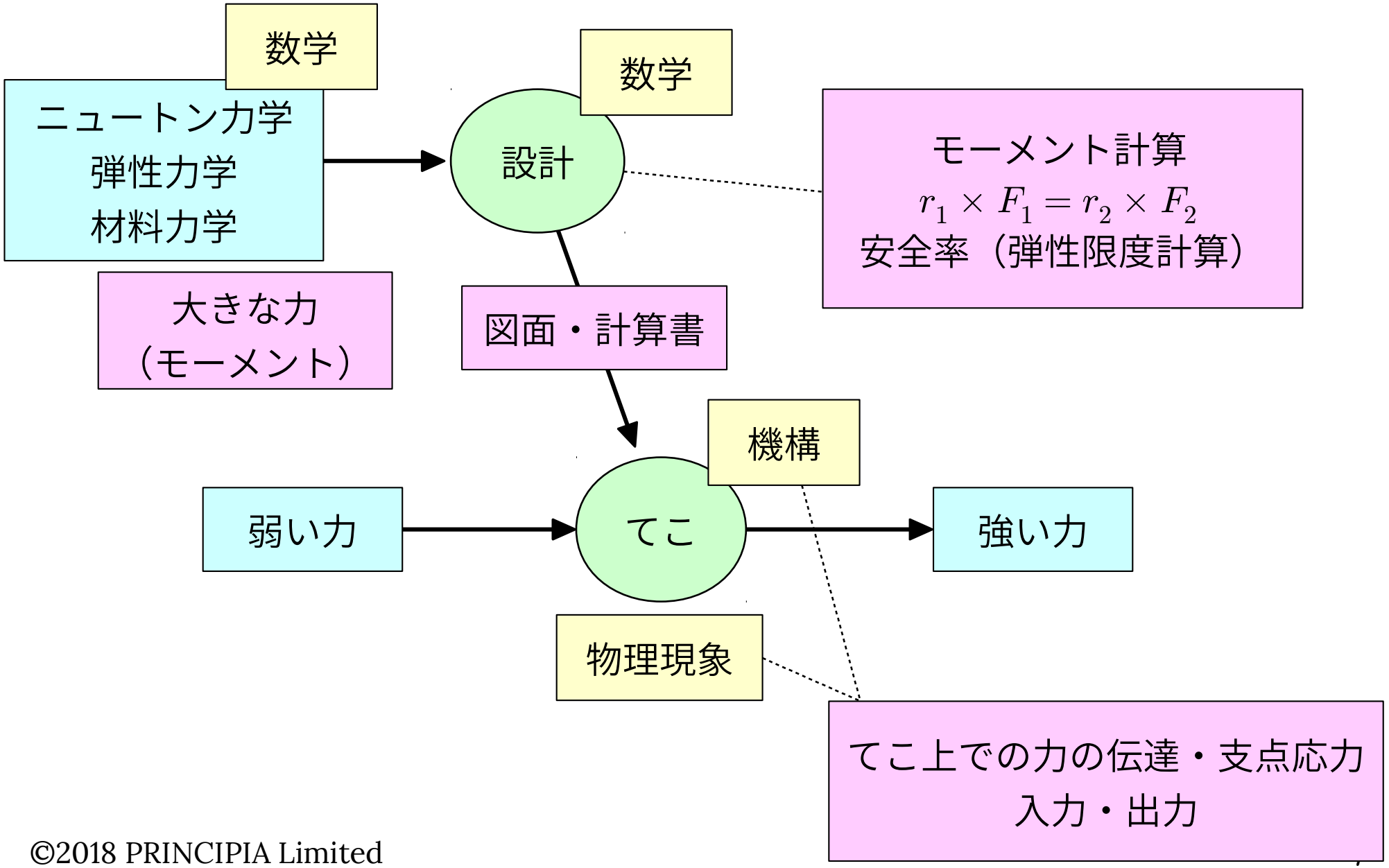
なぜ数学と理論を使うのか？



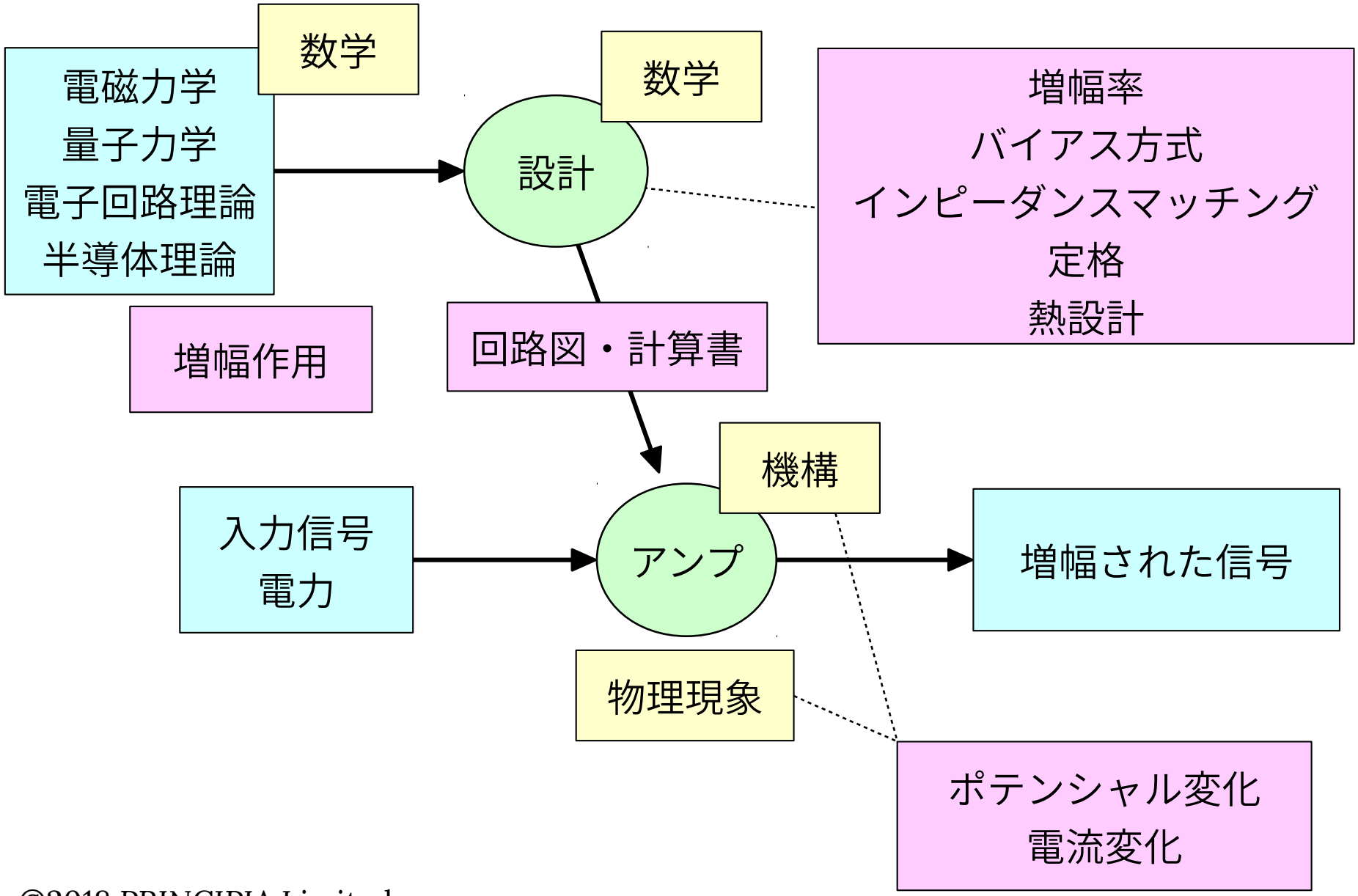
工学による開発と利用

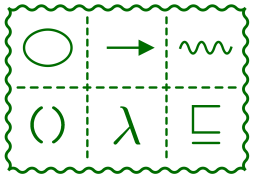


例：てこ



例：アンプ

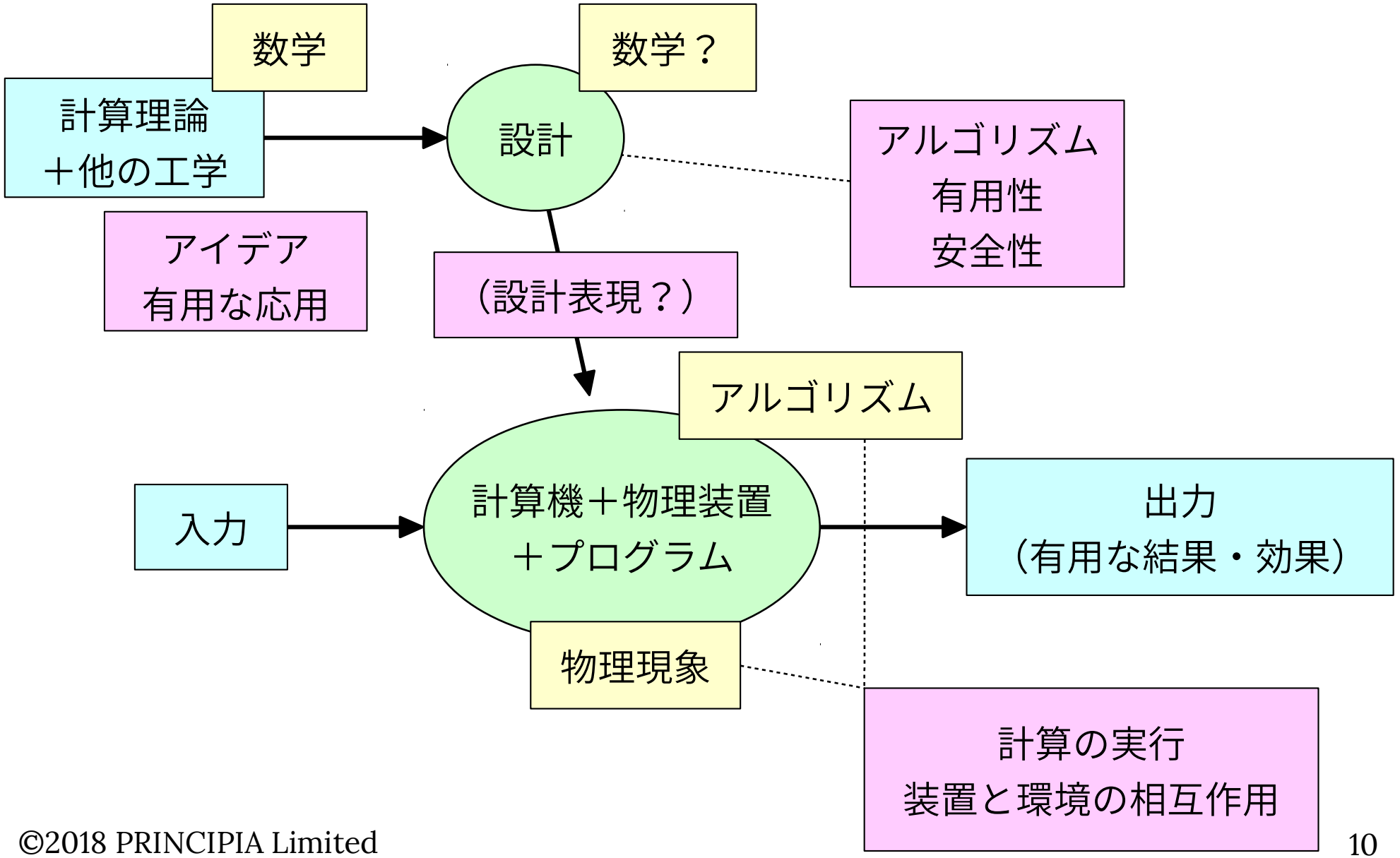




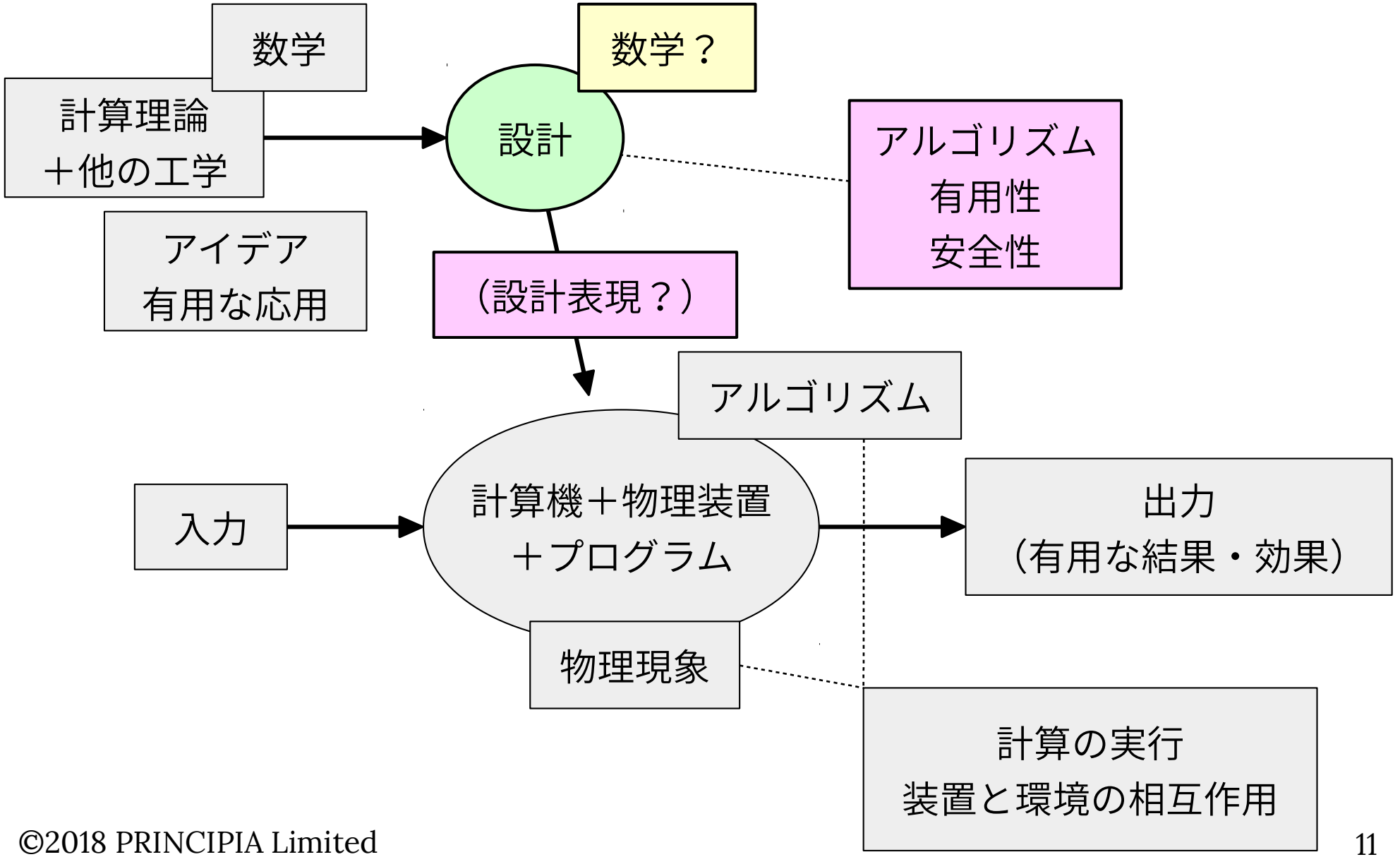
ソフトウェア開発は工学になれるか？

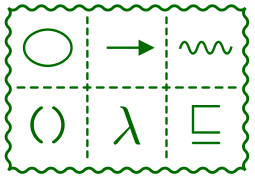
- ソフトウェア開発の基礎は計算機科学？
- 計算機科学は自然科学 (science) ではない
  - 物理法則・物理現象を相手にしているわけではない
- 計算理論に基づく有用な結果・効果がある
  - 広い意味での計算（数値計算，探索・検索）
  - 通信
  - 制御
- 物理装置としての計算機や物理装置が必要
  - 記憶装置，ネットワーク，制御装置と対象

ソフトウェア開発は工学になれるか？



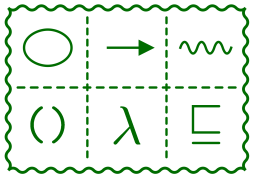
ソフトウェア開発は工学になれるか？





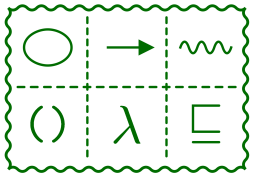
機械・電子工学とソフトウェアの比較

	機械工学	電子工学	ソフトウェア工学
数学	解析 , 代数 幾何 , 確率	解析 , 代数 確率 , 統計 , グラフ	論理学 , 集合論 代数 , グラフ
基礎理論	ニュートン力学 流体 , 弾性 熱 , 統計	電磁気学 , 量子力学 熱 , 統計 バンド理論，物性	計算理論 言語意味論
応用理論	機構 , 材料 , 加工 制御	素子 , 回路 , 通信 , 信号 情報 , 制御 , RF, 電波 デジタル	アルゴリズム データベース CG
設計表現	図面 , 計算書	回路図 , 計算書 , HDL タイミングチャート	ソースコード？ 形式仕様
設計と検証	実験 シミュレーション	実験 シミュレーション	テスト 形式検証 モデル検査

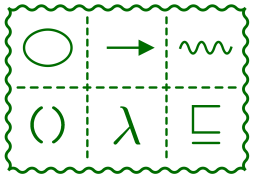


設計表現の課題

- ソースコードは設計表現か？
  - 最終成果物の完全な記述（実行ファイルを生成可能）
- 抽象度の高い設計表現の必要性
  - ソースコードは試行錯誤の中間表現として慣性が大きい
- 抽象度の高い設計表現の課題：不完全な情報
  - 意味内容が厳密ではない
  - 検証できない
  - 最終成果物を生成するのに必要な十分な情報を有していない



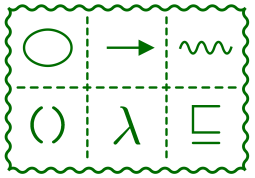
形式手法とは何か



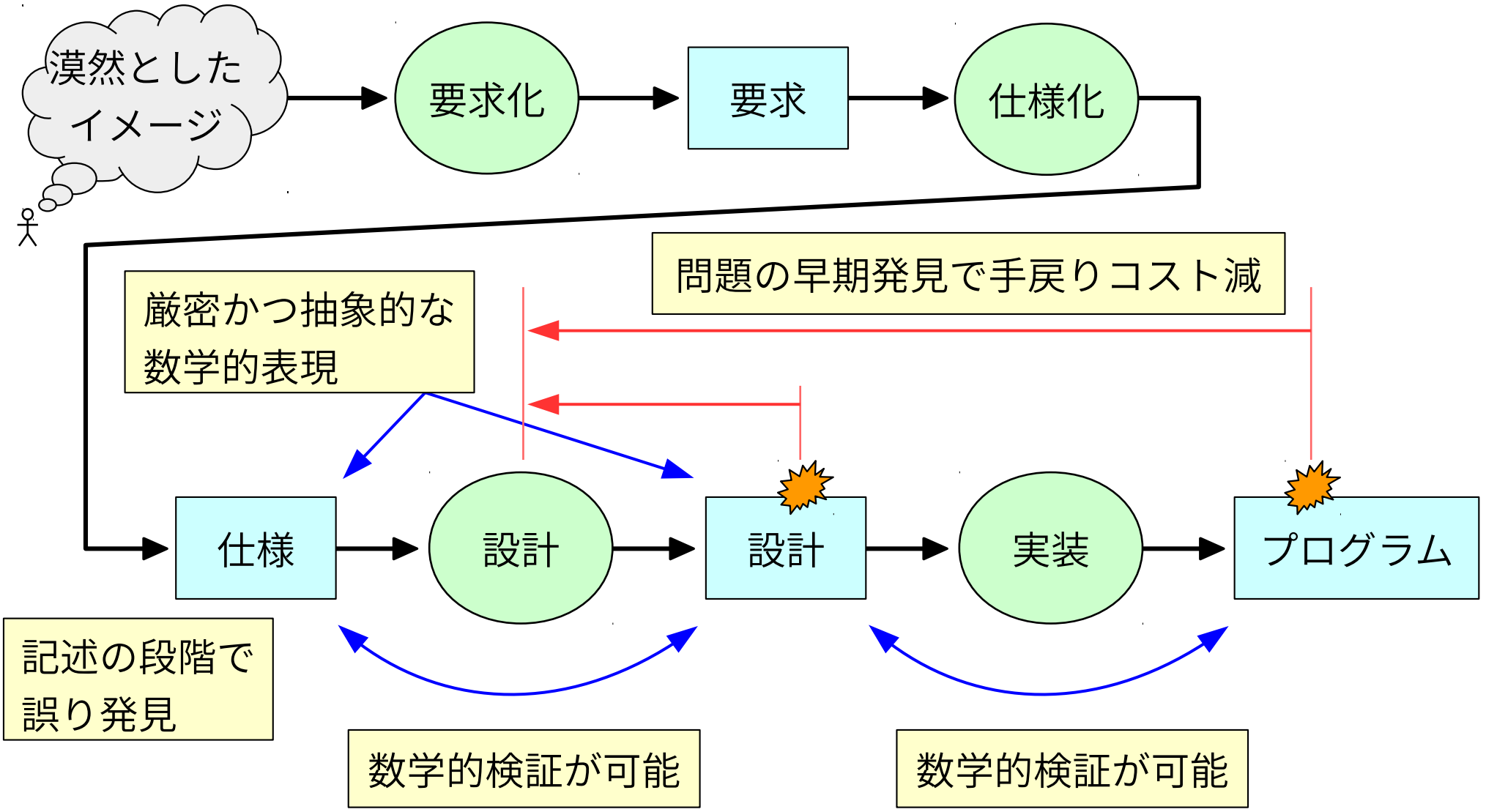
形式手法 (formal method) とは？

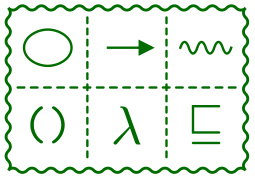
計算機科学を基礎とし，数学（数理論理学など）を使用して品質の高いソフトウェアを効率よく開発するための手法

- 数学的表現（論理式等）は抽象的であると同時に厳密な表現形式である．設計の各段階における成果物を段階に応じた抽象度で厳密に表現できる
- 成果物を厳密に表現することで問題点を発見できる．設計の早い段階での成果物作成はコストの高い手戻りを減らし，開発効率を向上させる
- 数学的表現は論証による検証が可能である．誤りを確実に発見することができるので品質を向上させることができる



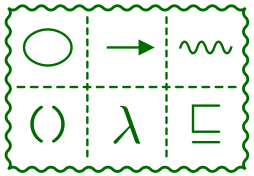
形式手法 (formal method) とは？



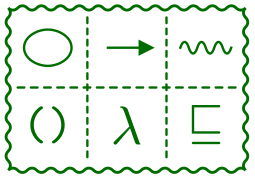


主な形式手法

- 形式仕様記述
  - 数理論理学・集合論等に基づく厳密な数学的言語を使って仕様・設計を厳密に記述する
- 形式検証（定理証明）
  - 数学的な論証によってソフトウェアを検証（証明）する
- モデル検査
  - ソフトウェアが持つすべての動作を網羅的かつ自動的に探索することで性質を検証する

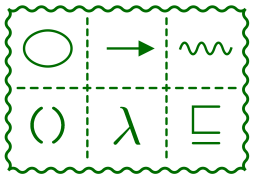


ソフトウェアの 2 つの側面：計算と相互作用

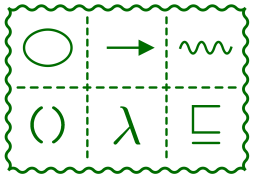


計算機は計算するだけの機械ではない

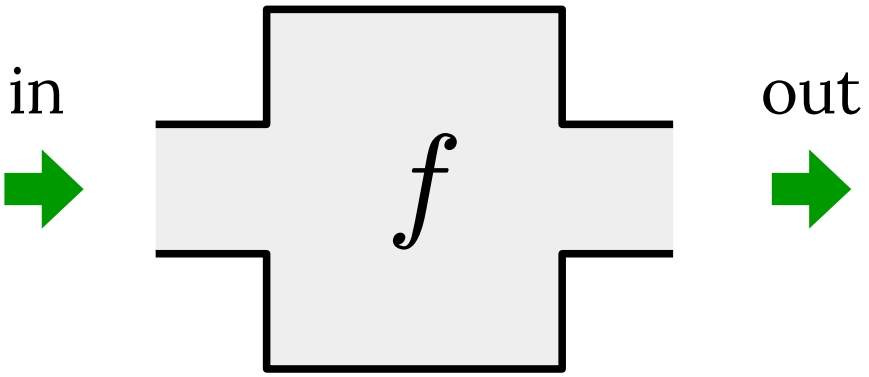
- 計算機は計算のために発明された
- コンピュータを応用したシステムの多くは相互作用あるいは通信・コミュニケーションを主体としたものになっている
  - 制御・組み込みシステム
    - コンピュータと制御対象との相互作用
  - ネットワークシステム
    - ノード間通信，クライアント・サーバ
  - 対話的システム：ユーザとの相互作用
    - スマートフォン，アプリケーション，ゲーム， ATM

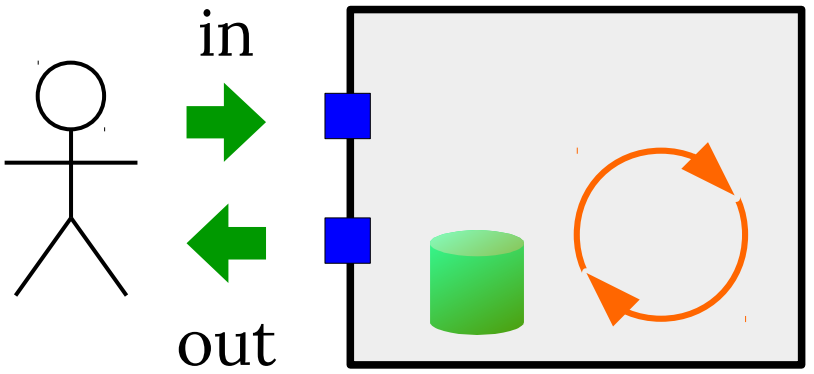


相互作用に力点を置く
リアクティブシステム



計算システムとリアクティブシステム





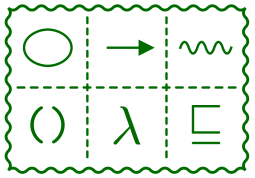
計算システム

リアクティブシステム

入力が与えられると計算を実行し
結果を出力して停止する

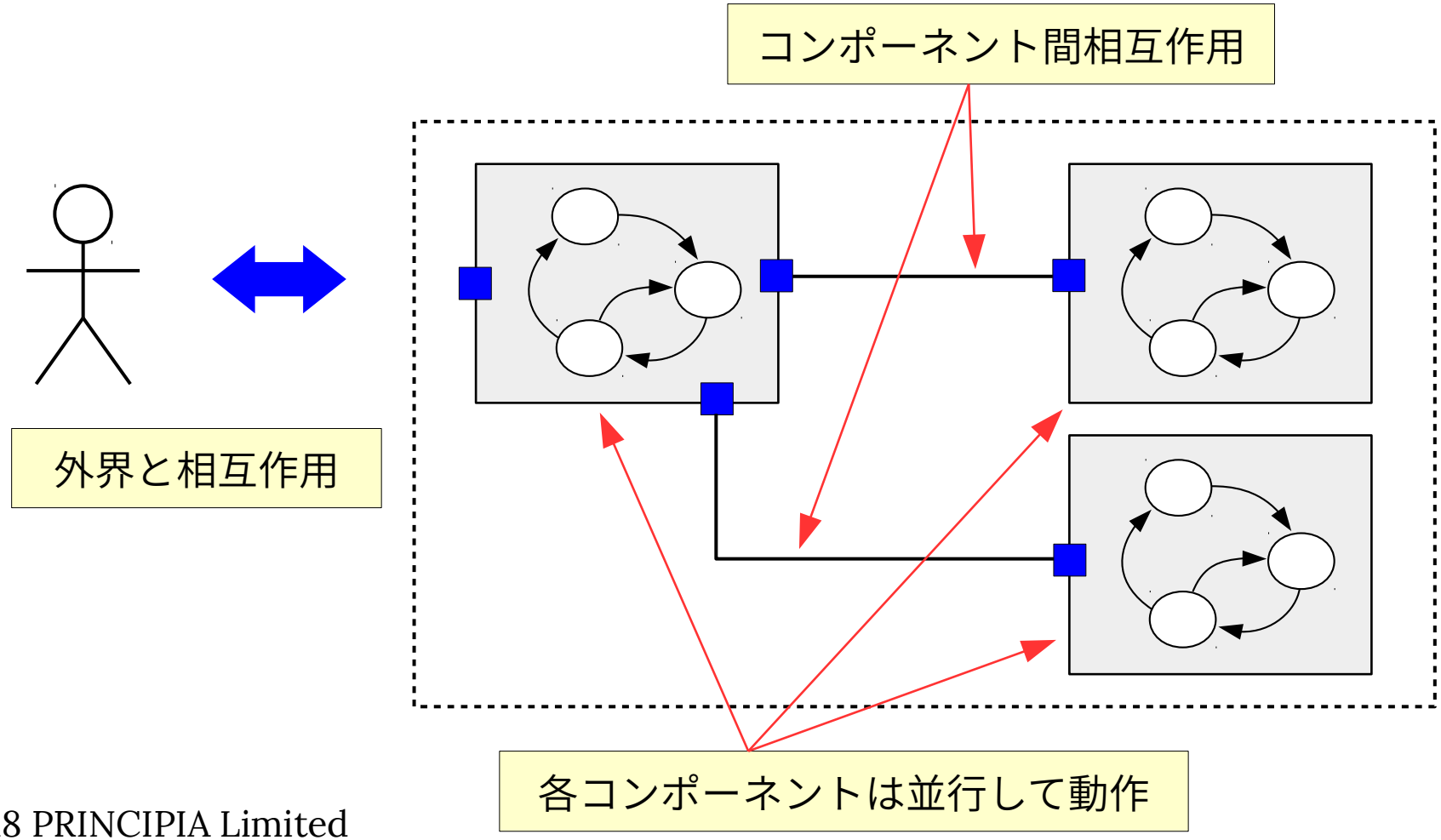
環境（ユーザや他のシステム）と
相互作用をしながら計算の実行や
サービスの提供を継続的に行う

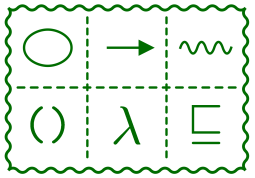
- 自律的に動作する
- 内部状態を持つ
- 特別な場合を除き停止しない



並行システム

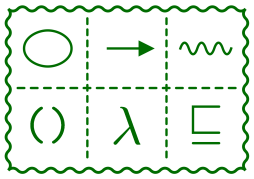
複数の構成要素からなるシステムで，各構成要素が自律的に動作し，互いに作用を及ぼしあいながら全体として目的の仕事をするシステムのこと



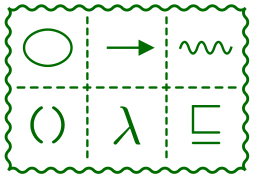


形式手法の主なツール

	計算を主とする ソフトウェア向けの手法	相互作用を主とする ソフトウェア向けの手法
形式仕様記述	Z, VDM, B, OBJ	CSP, π計算 , Event-B
形式検証	Isabelle, Coq, B, OBJ	CSP, π計算 , Event-B
モデル検査 （※広い意味での）	Alloy, SAT/SMT solver	SyncStitch, FDR, PAT, LTSA, SPIN, SMV

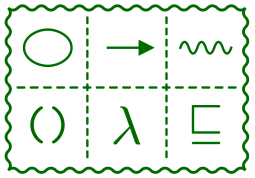


ソフトウェアの仕様を厳密に記述する



ソフトウェアにおける仕様とは何か？

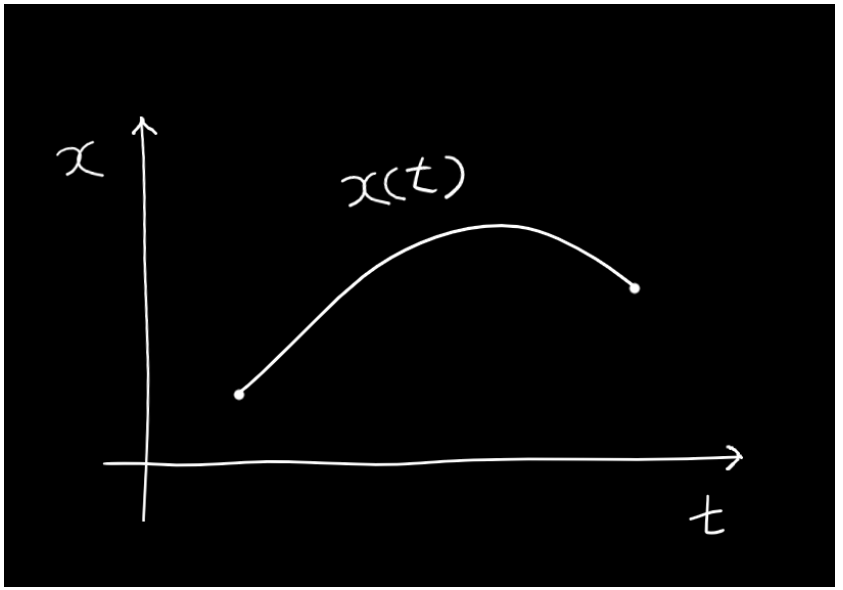
- 要求とは問題の記述である
  - コンピュータシステム（ハードウェア＋ソフトウェア）を使って解決したい問題は何かを記述したもの
- 仕様と設計は解の記述である
  - では仕様と設計の違いは何か？

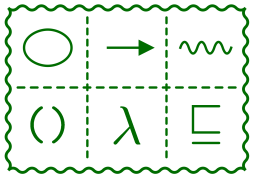


プログラムの“動き”を記述する

- 時間的に変化するものを，どのように静的に表現するか？
- ニュートン力学：
  「運動を理解するとは，運動を記述することである」
  - 系の配位（物体の位置）を時間の関数として表す

ひとつ高い次元の空間を用意し
時間軸を空間軸に置き換えることで
静的な幾何学的表現になる



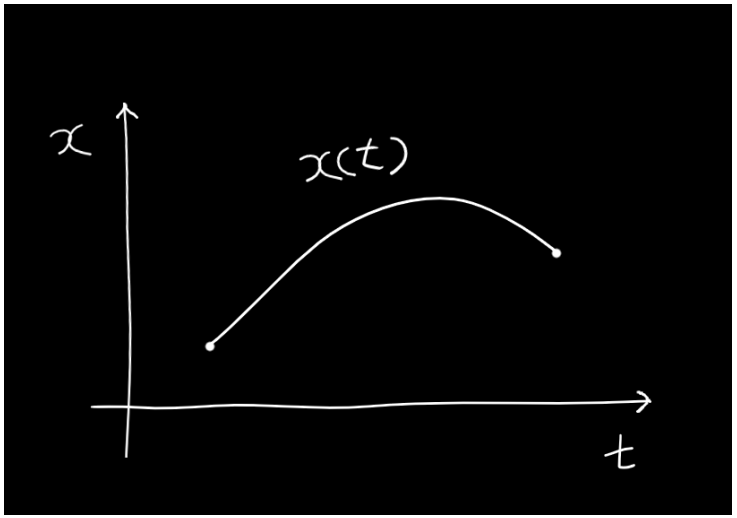


プログラムの“動き”を記述する

物理系

配位が系の状態を定義する
微分方程式が系の動きを決定する

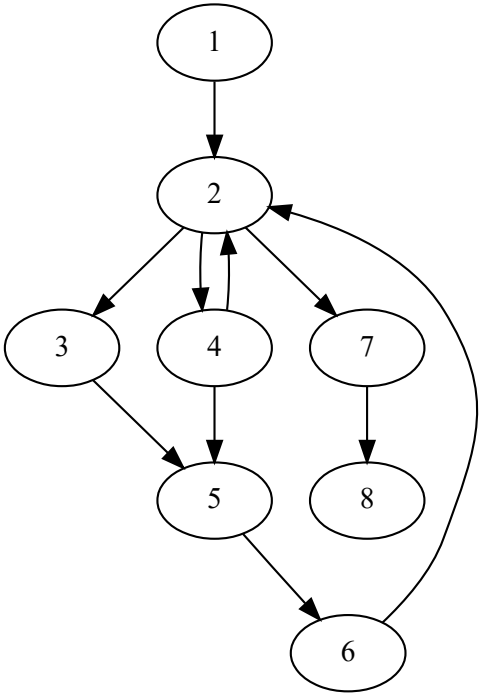
$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = F(\boldsymbol{x}, t)$$



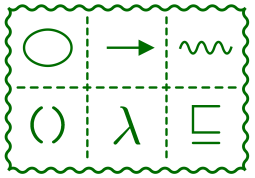
連続的な遷移

プログラム

変数と値によって状態を定義する
遷移が系の動きを決定する



離散的な遷移



遷移を記述する

物理系　無限小時間後の状態と現在の状態との間の関係を式で記述する

$$\boldsymbol{x}(t+\mathrm{d}t) = \boldsymbol{x}(t) + F(\boldsymbol{x}, t)\mathrm{d}t$$

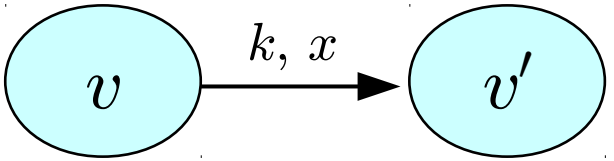
事後状態　　事前状態

プログラム　事前・事後の状態を定める変数間の関係を論理式で記述する

$$\phi(s, s')$$

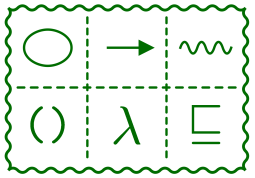
事後状態の変数にはプライム $'$ をつけて区別する※

例：配列 v のインデックス k の要素を x に更新する



$$0 \le k < n \Rightarrow v'[k] = x \wedge \forall i.\ 0 \le i < n \wedge i \neq k \Rightarrow v'[i] = v[i]$$

※ 流儀はいろいろあります



日本語による仕様は曖昧になりやすい

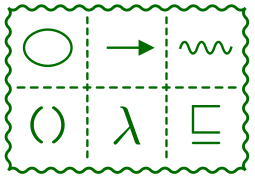
例題：

配列 v の中で値 x の後ろに y があるかどうか判定する
プログラムを作ってください

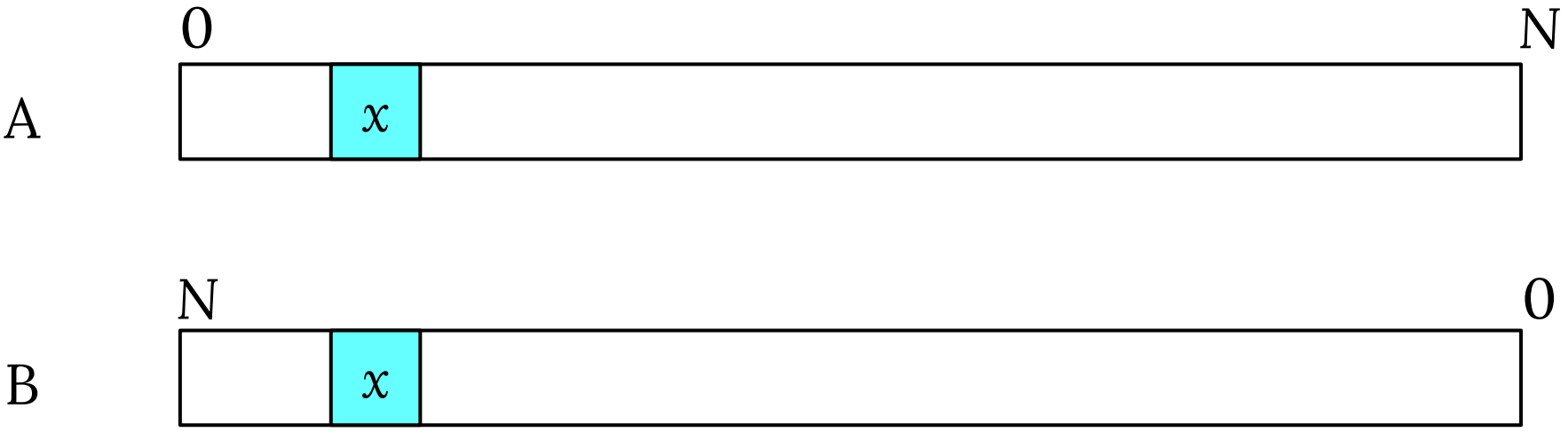
“後ろ”とは？

A

B



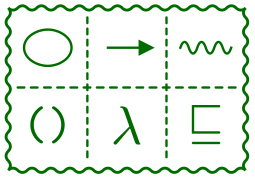
そもそもどちらが前？



ビットフィールドやメッセージパケットの図でよく誤解がある

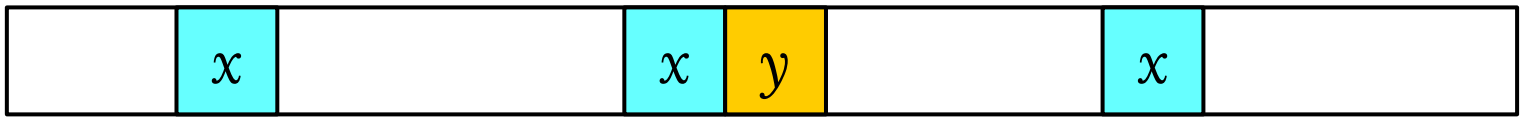
メモリマップを縦に書くときもどちらが上かで混乱する

機械部品や電子部品のデータシートで top view と bottom view を勘違いして作業を進めてしまったときの脱力感 ...

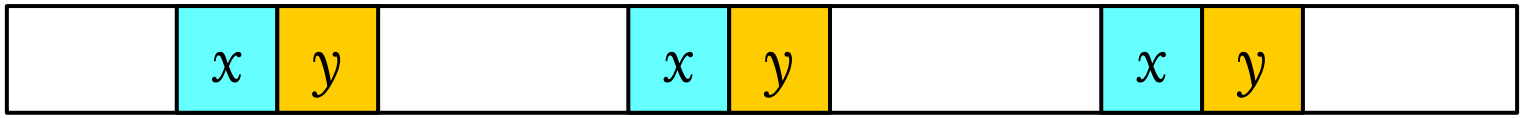


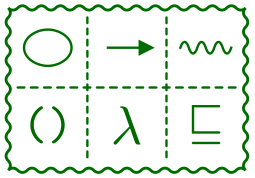
x が複数ある場合は？

A



B

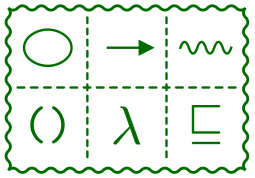




x が複数ある場合は？

A

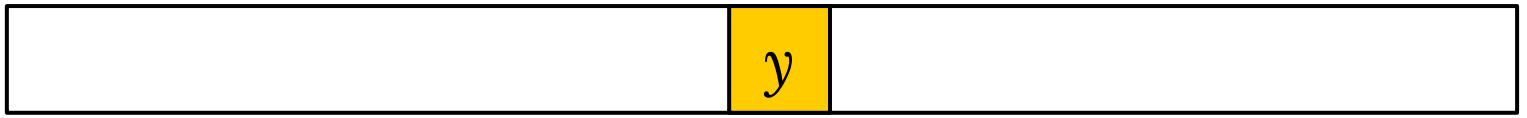
B



x がなかったら？

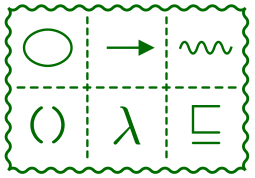
A

B



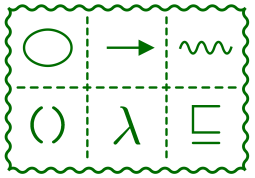
関係者が全員論理学の論理を使っているとは限らない

- 日常論理
- 法学界の論理 (!?)



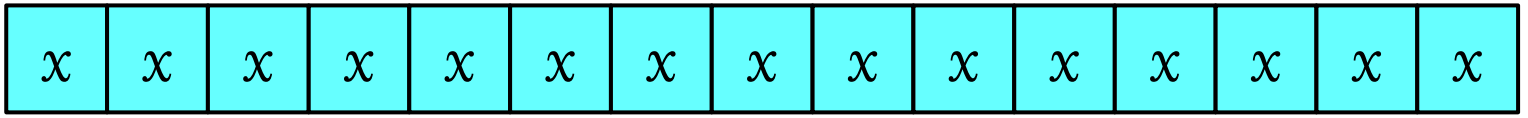
x が末尾にあったら？

A

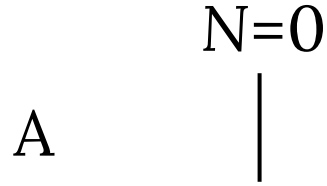


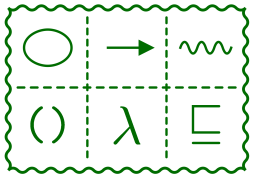
x と y が同じだったら？

A



配列の大きさが 0 だったら？





形式的に記述する

厳密な形式的定義の例

A. $\forall i.\, 0 \leq i < n-1 \wedge v[i] = x \Rightarrow v[i+1] = y$

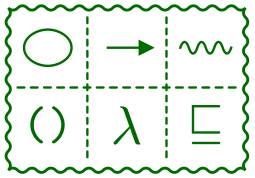
末尾に x があってもよい

直後にある

B. $\forall i.\, 0 \leq i < n \wedge v[i] = x \Rightarrow \exists k.\, i < k < n \wedge v[k] = y$

直後でなくてもよい

どちらも配列の大きさが 0 の場合は成りたつ



形式仕様は分析や性質の証明ができる

B. $\forall i.\, 0 \le i < n \wedge v[i] = x \Rightarrow \exists k.\, i < k < n \wedge v[k] = y$

形式仕様を書いたからといって必ずしも意図したものになっているとは限らない．しかし形式仕様は厳密な議論を可能とするので確認しやすい

例 $x = y$ のときで条件が成り立つことはあるか？

B で $x = y$ と置く

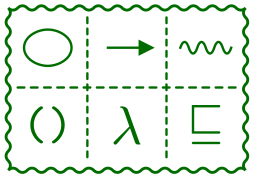
もし v のどこか i に x があれば $k > i$ にも x がなければならない

v は有限なので最後の x があることになり条件を満たさない

よって B が成り立つなら v は x を含まない

逆に v が x を含んでいなければ B は成り立つ

ツールによる分析・証明も可能



仕様記述言語 Z

スキーマと呼ばれる以下のような構造を使って状態遷移を記述する

遷移の名前

$Purchase_0$

$BoxOffice$

$BoxOffice'$

$s? : Seat$

$c? : Customer$

$s? \in seating \setminus \mathrm{dom}\, sold$

$sold' = sold \cup \{s? \mapsto c?\}$

$seating' = seating$

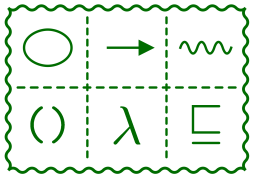
事前状態（スキーマ）

事後状態（プライムをつける）

この遷移での入力パラメータ

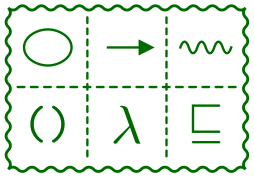
変数間の関係を記述

ISO/IEC 13568:2002 として標準化されている
IBM CICS, Transputer FPU などの実績

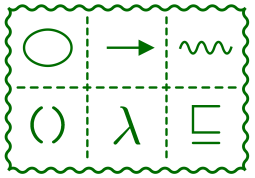


形式仕様記述の事例

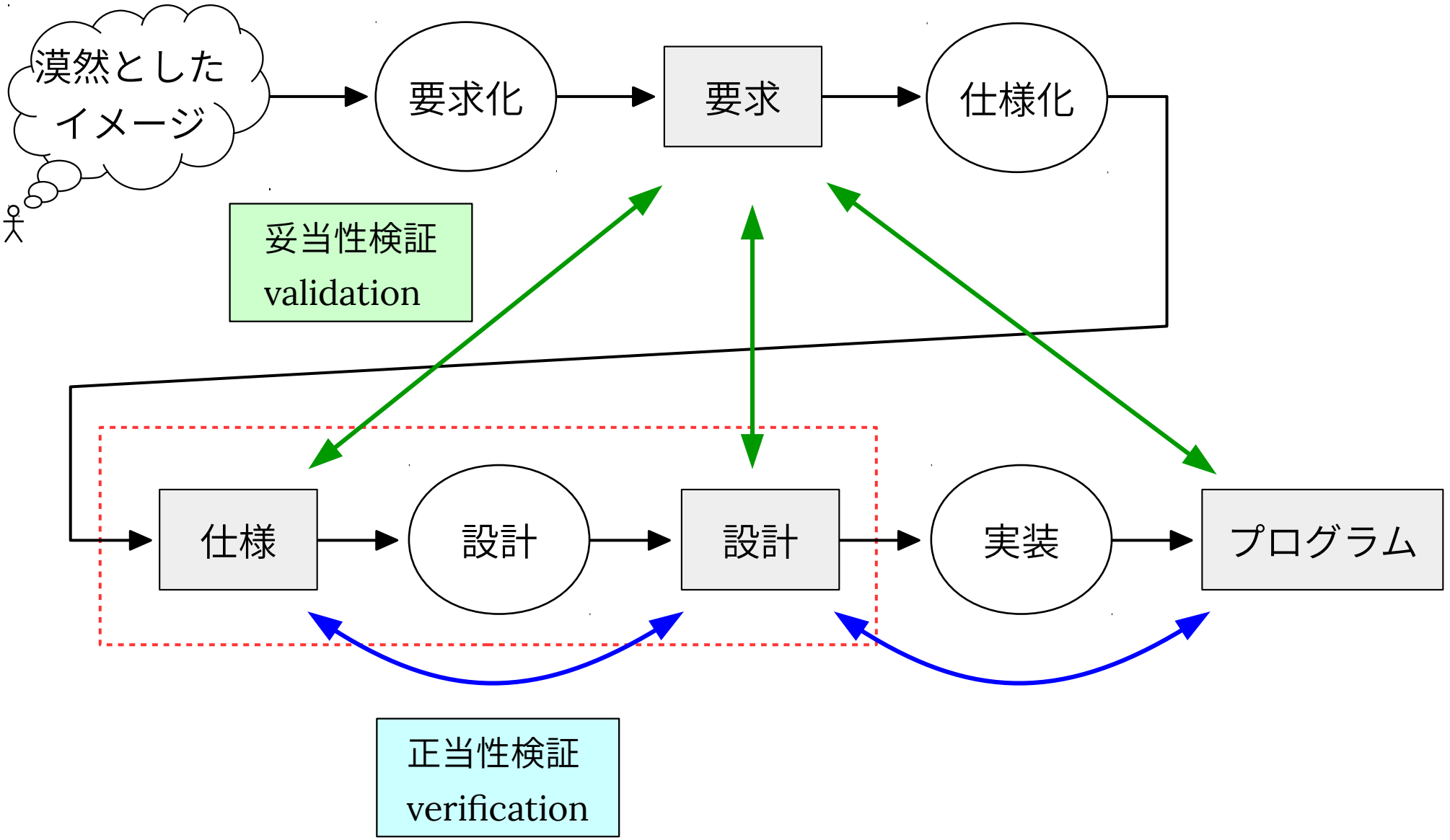
- フェリカネットワークス (VDM)
  - https://ipa.go.jp/files/000044283.pdf
- ライフロボティクス（自社開発の仕様記述言語）
  - http://jasst.jp/symposium/jasst17tokyo/pdf/D3.pdf

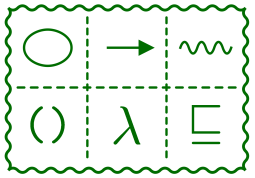


ソフトウェアが
仕様を満たす条件とは



Verification & Validation (V&V)





正当性関係

プログラムが仕様に照らして正しいという性質を正当性関係という

正当性関係は仕様とプログラムの間になりたつ関係である

仕様 $\sqsubseteq$ プログラム

正当性を表す記号
不等号 $\leq$ や部分集合 $\subseteq$ に似ている

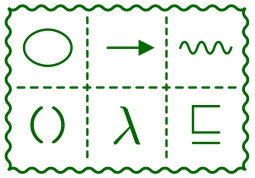
正当性関係は仕様と設計，設計とプログラムの間の関係にもなりうる

正当性関係は推移律をみたす

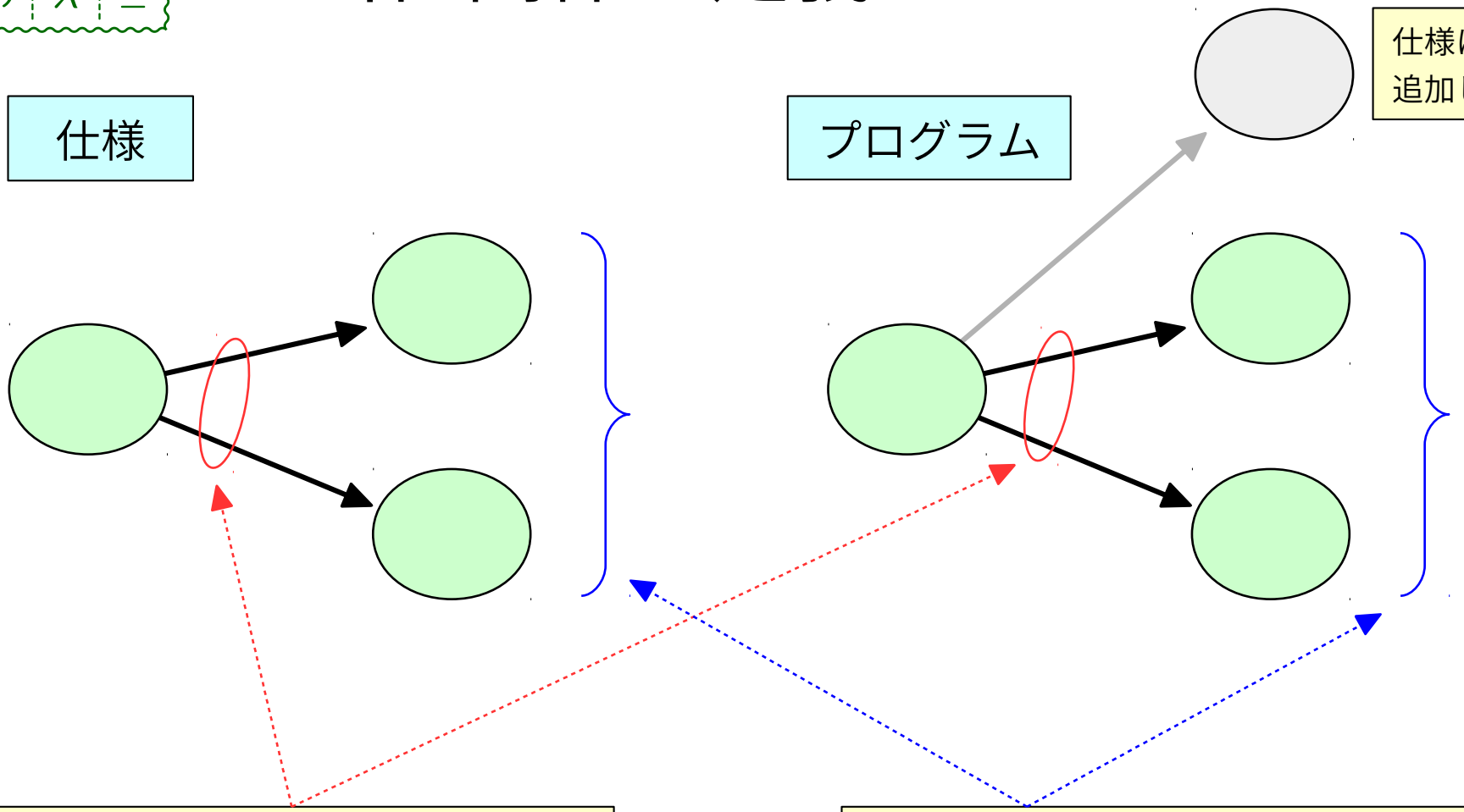
仕様 $\sqsubseteq$ 設計

設計 $\sqsubseteq$ プログラム

$\Rightarrow$ 仕様 $\sqsubseteq$ プログラム



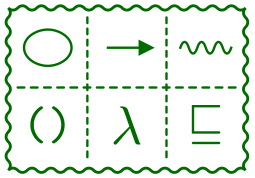
正当性関係の定義



条件 1 ：仕様が要求している
遷移がなければならない

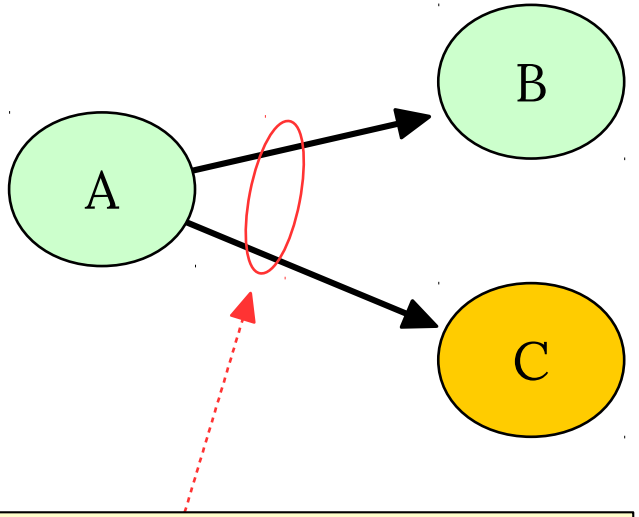
条件 2 ：遷移先は仕様が許容する
中に含まれていなければならない

この 2 条件は置き換え可能であるための条件である



非決定性とテストによる検証

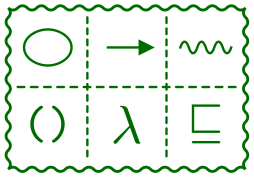
プログラムには非決定性がありうる
同じ状態（環境も含む）で同じ操作をしても結果が異なる場合がある



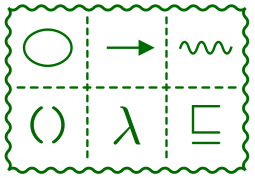
仕様が C を許容しない場合，テストでたまたま B が出たとしても確認できたことにはならない．

ソフトウェアの検証をテストによって完全に行うことはできない

設計の内部情報を使った検証を行うしか方法はない



ソフトウェアの正しさを
数学的に証明する



Hoare 論理とは

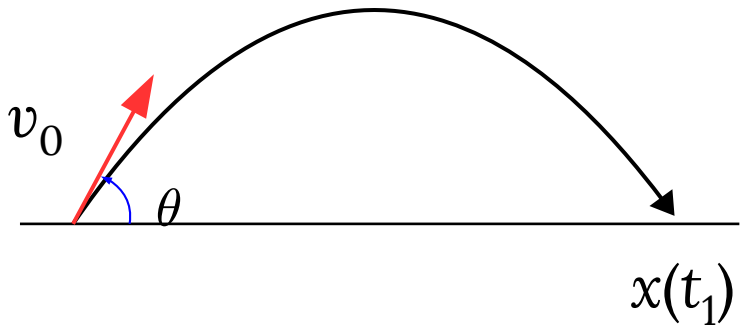
- プログラムの正当性を証明するための形式システム（論理体系）
- プログラミング言語の意味定義にも使われる
  - 公理的意味論という
  - 他に表示的意味論や操作的意味論などがある
- 1969 年 C. A. R. Hoare によって提案された
  - Robert W. Floyd による仕事が基礎にあるため Floyd-Hoare 論理と呼ばれることもある

Photograph by Rama, Wikimedia Commons, Cc-by-sa-2.0-fr

計算の軌跡によって検証する

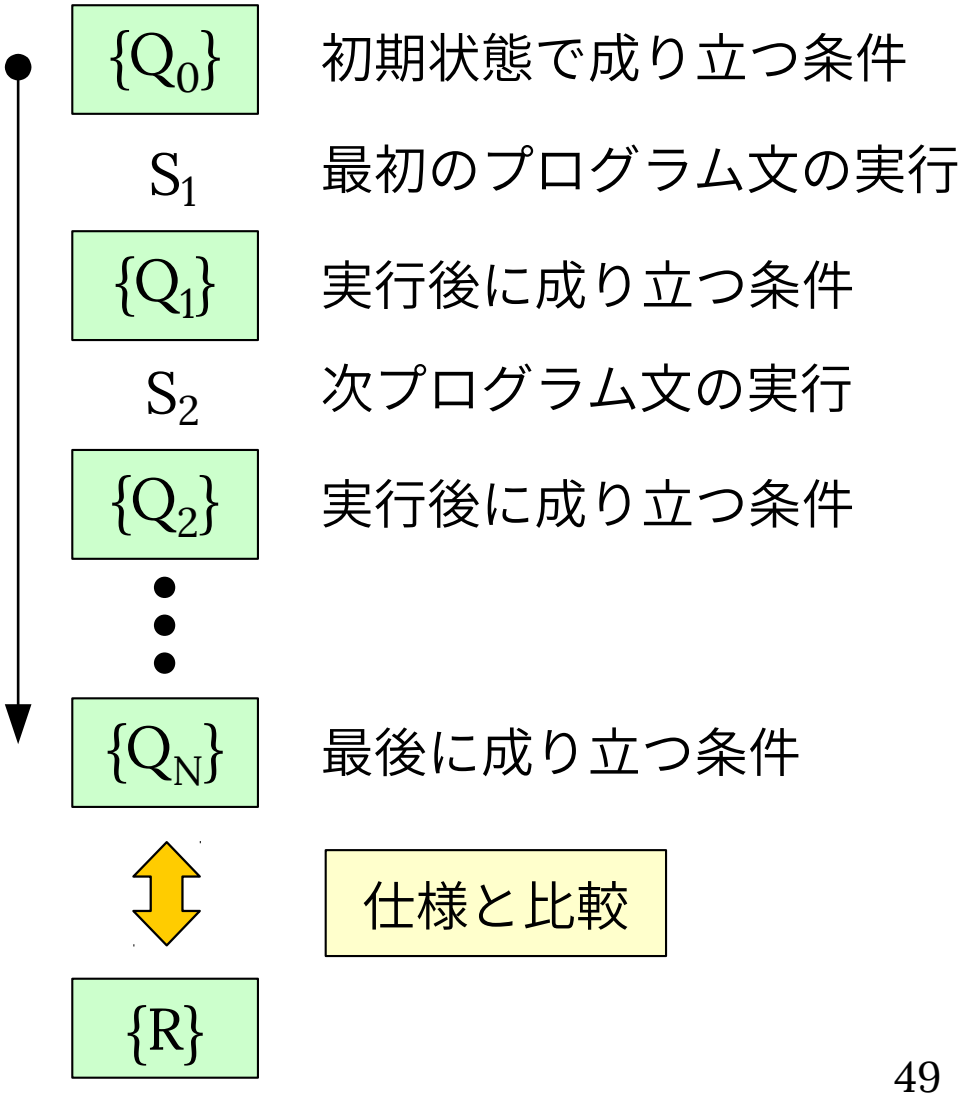
物理系

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{F}(\boldsymbol{x}, t) \Rightarrow \boldsymbol{x} = \boldsymbol{x}(t)$$



微分方程式から運動の記述（軌跡）を求め，それによりある力学変数がある時刻において条件を満たすことを示す

プログラム

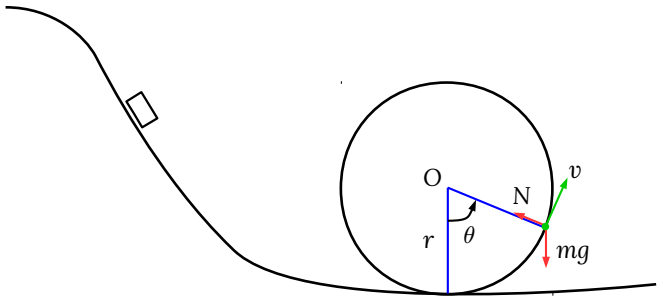


不変条件を使ってループを処理する

物理系

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = F(\boldsymbol{x}, t)$$

- $H(\boldsymbol{x}, \boldsymbol{p}) = E$
- $N = N(\boldsymbol{x}(t)) \geq 0$



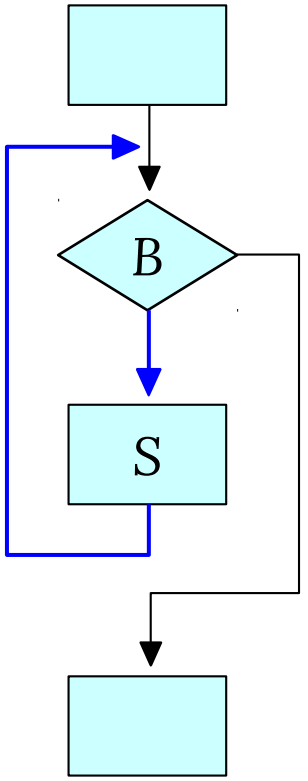
プログラム

{P ∧ B}

S

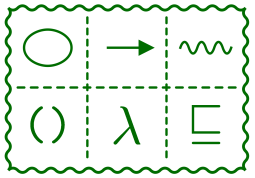
{P}

{P ∧¬B}

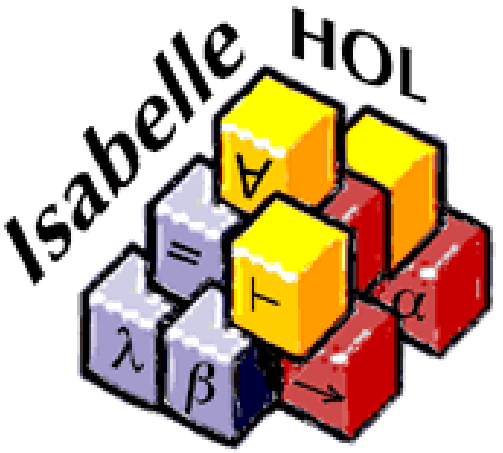


運動が求められない場合でも不変量から期待する結果を示せる場合がある

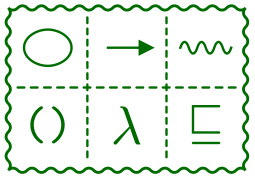
ループの軌跡は一般に求められないがループ前中後で成り立つループ不変条件によってループ後に成り立つ条件を証明できる



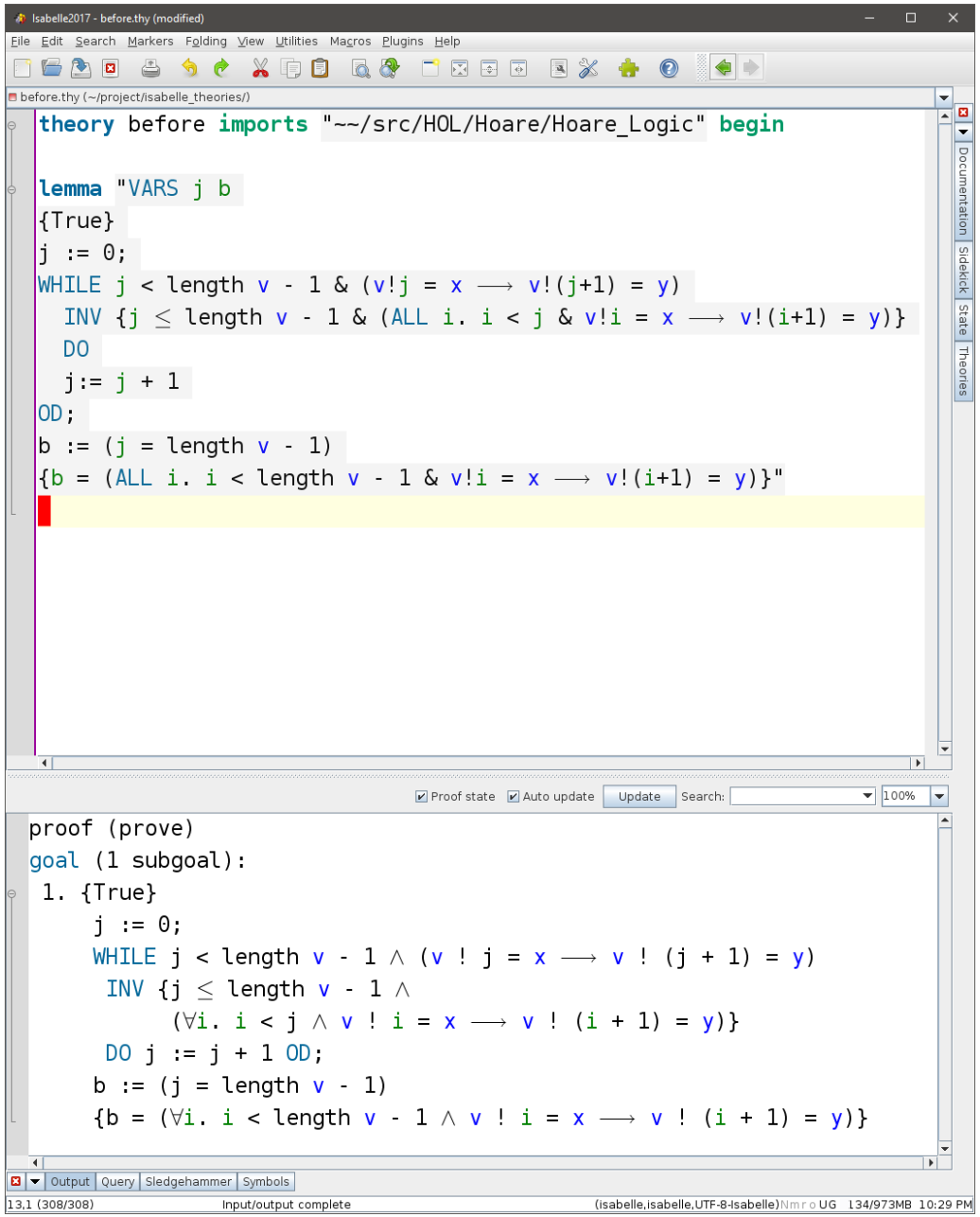
定理証明支援系 Isabelle



- 高階論理に基づく汎用の定理証明支援系
  - https://isabelle.in.tum.de/
- 自動証明能力がとても高い
- 外部の証明プログラムを呼び出す機能を持っている
- Hoare 論理のためのライブラリを持っている
  - プログラムを自動的に論理式に変換してくれる
- オペレーティングシステムに関する証明など大規模証明の実績がある
- ケプラー予想の形式証明 Isabelle と HOL Light によってなされた
  - https://www.cambridge.org/core/services/aop-cambridge-core/content/view/78FBD5E1A3D1BCCB8E0D5B0C463C9FBC/S2050508617000014a.pdf/formal_proof_of_the_kepler_conjecture.pdf

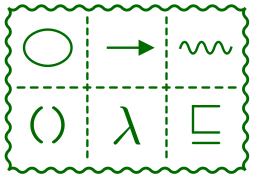


Isabelle による正当性証明

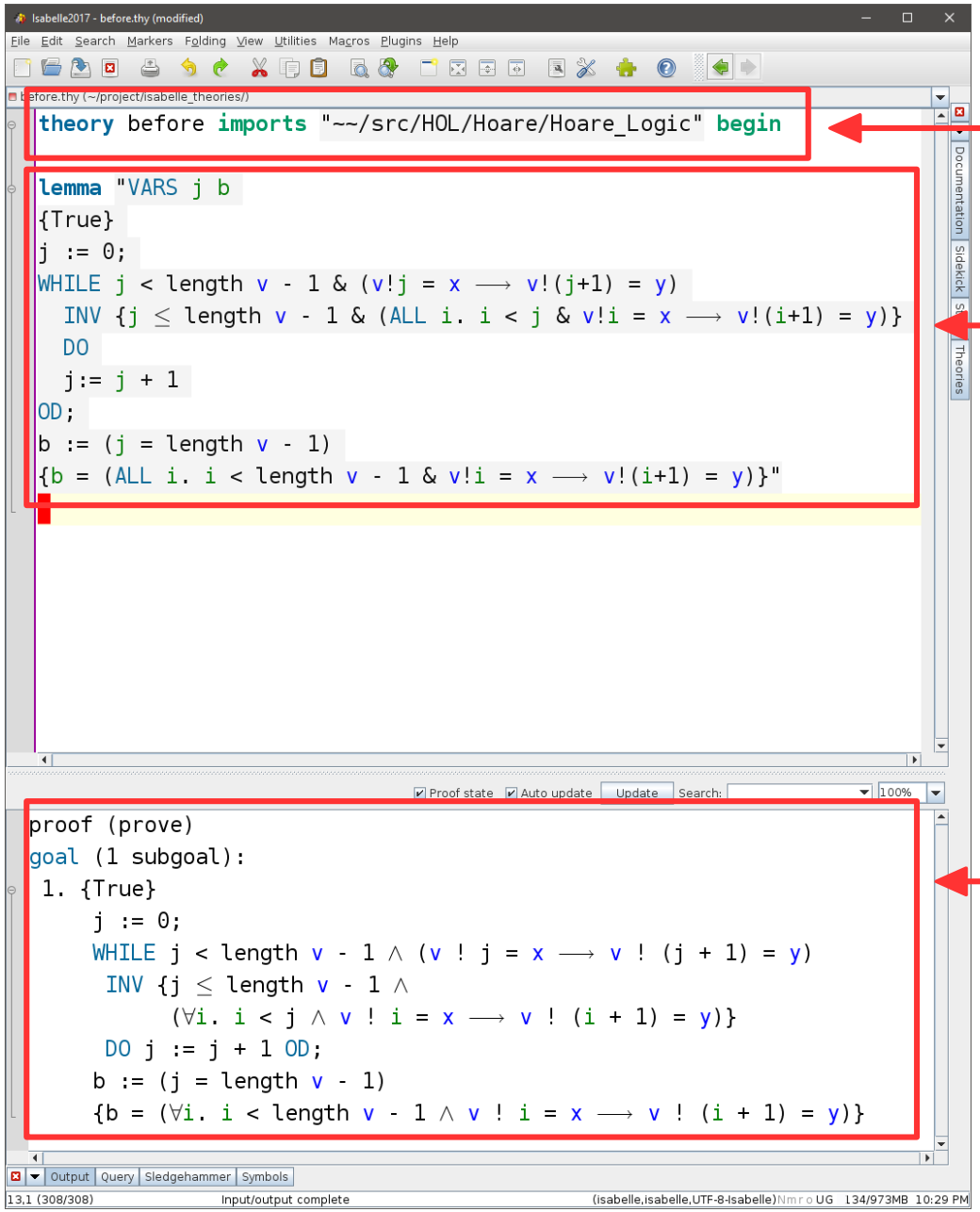


証明したい式と，証明のための
コマンドを入力・編集する

証明の途中経過
証明すべき部分式（サブゴール
という）を表示する



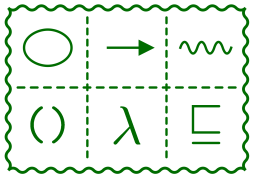
Isabelle による正当性証明



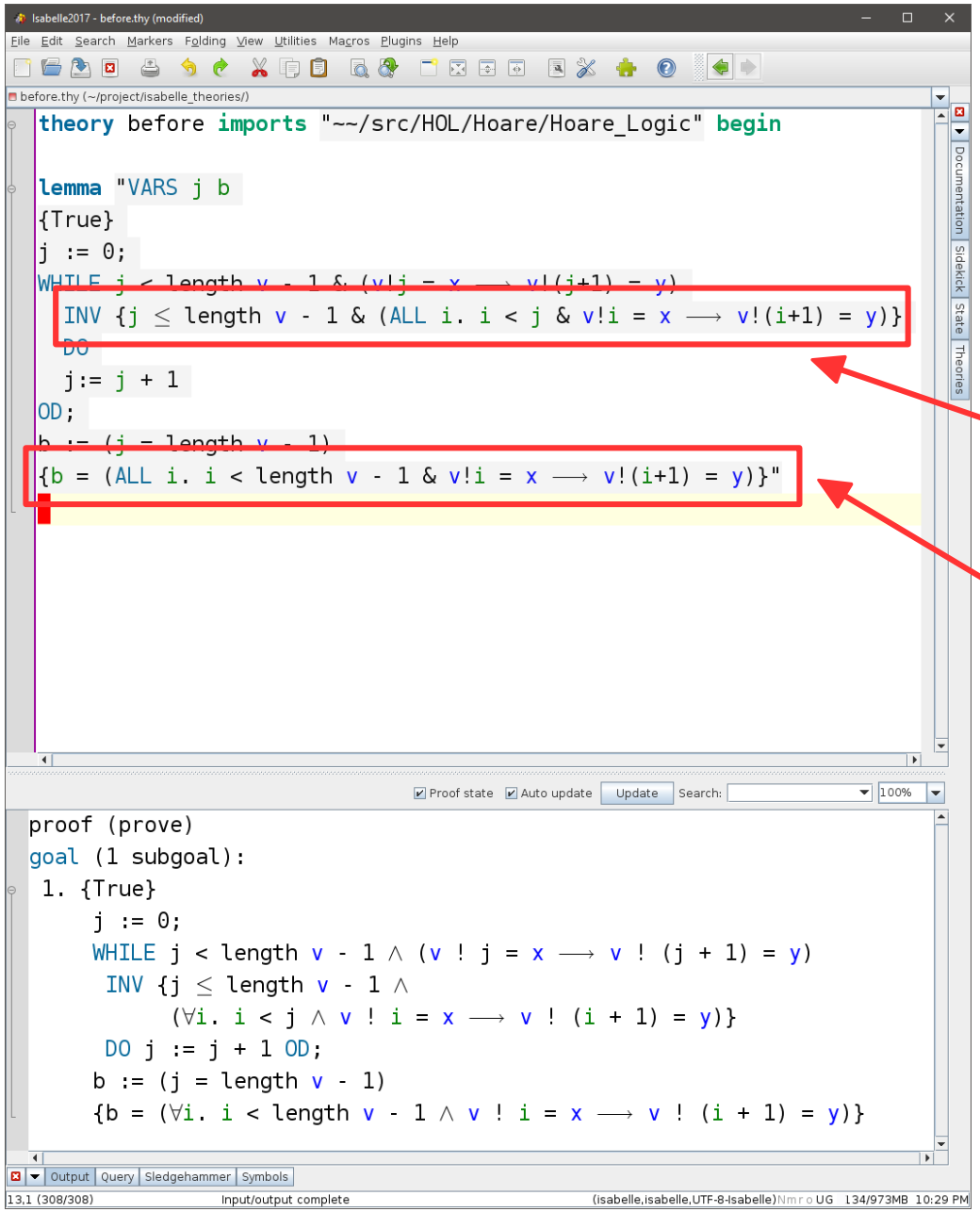
Hoare 論理のライブラリをインポートする

代入文， IF 文 , WHILE 文などのプログラム構文が使える

証明すべきサブゴール
最初は入力した式がそのまま表示される



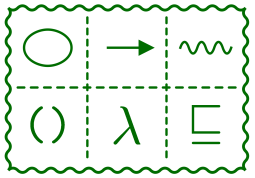
Isabelle による正当性証明



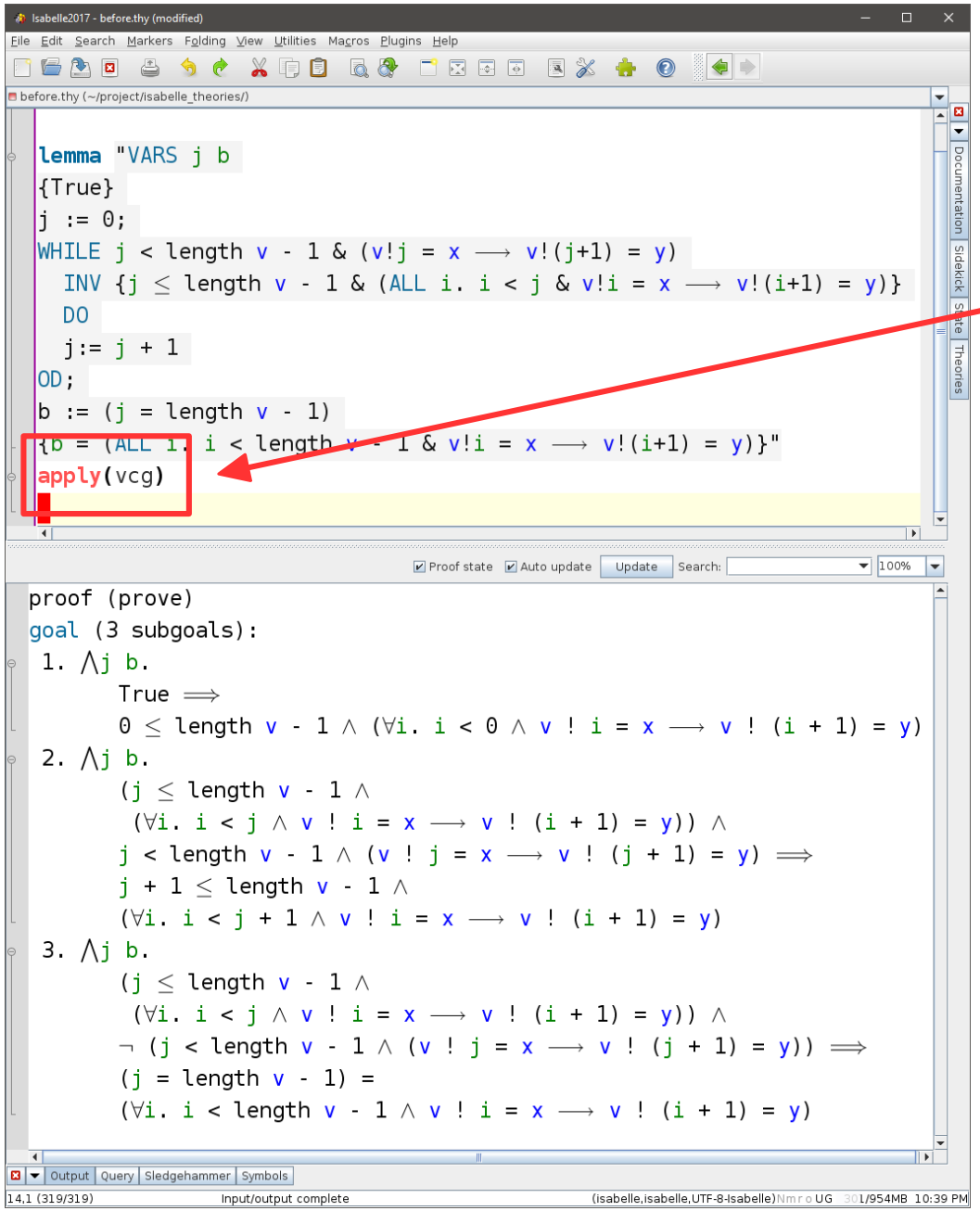
「x の後ろに y がある」ことを判定するプログラムの正当性を証明してみる

ループ不変条件

形式仕様「直後版 A」

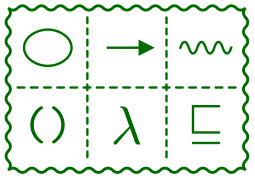


Isabelle による正当性証明

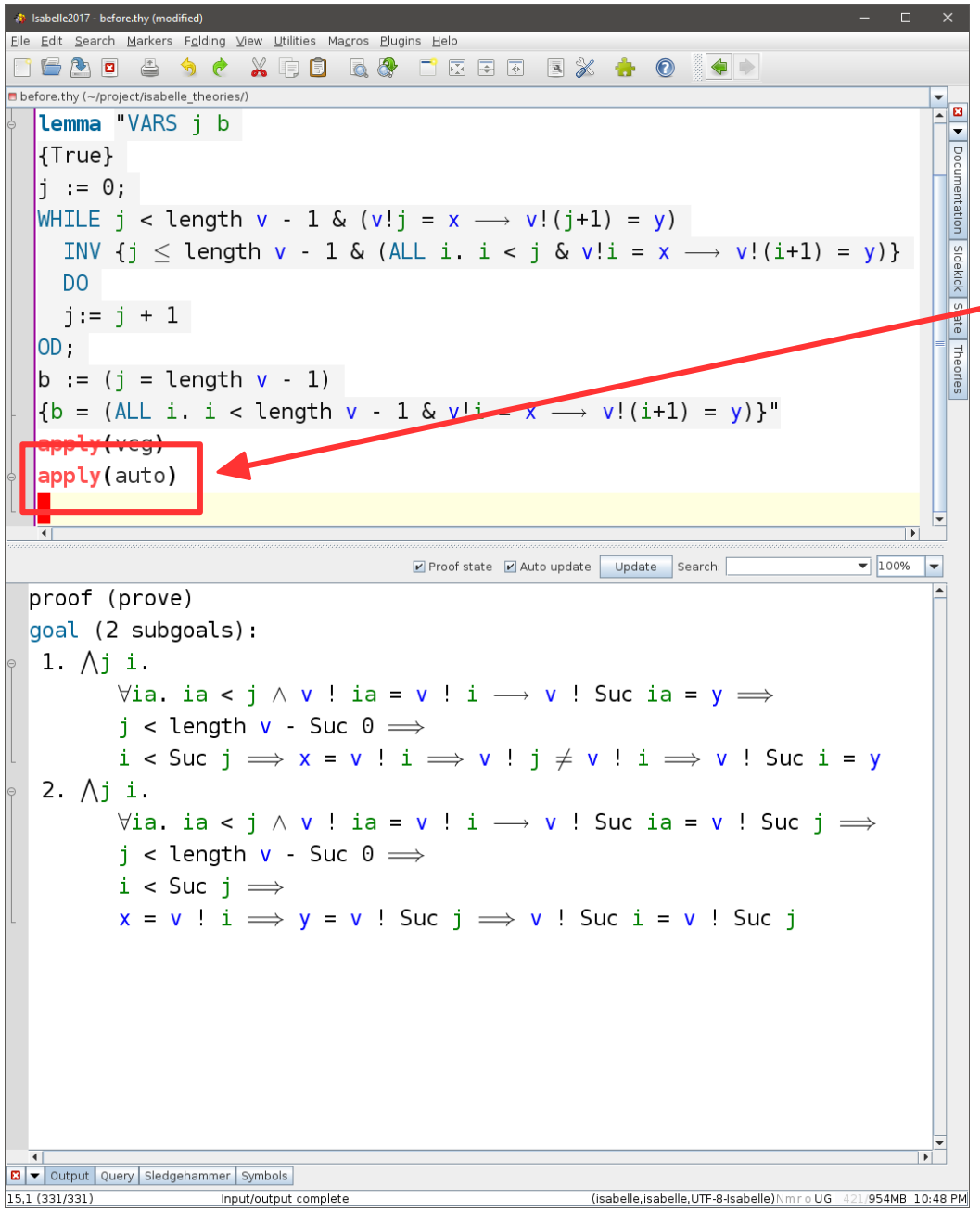


vcg というコマンドを実行すると
プログラムを証明すべき論理式に
変換してくれる

証明すべき論理式（サブゴール）
が 3 つ現れる
これはプログラムの実行軌跡の各
ステップに対応する



Isabelle による正当性証明

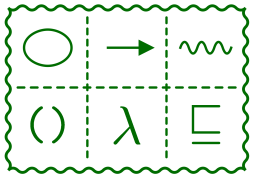


自動証明コマンド auto を実行

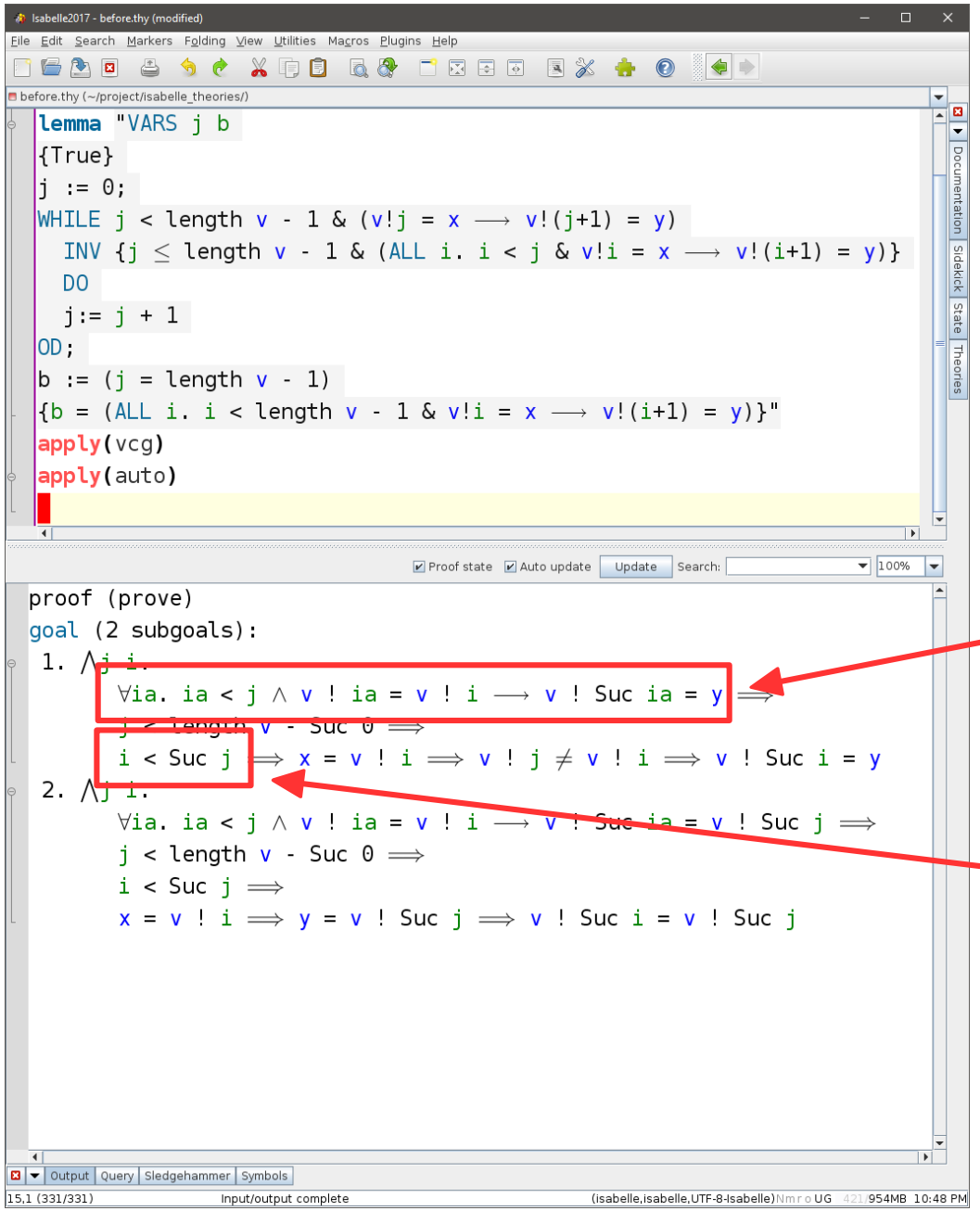
「ガンガンいこうぜ」

3 つあったサブゴールのうち 1 つは自動的に証明されて消えた

残り 2 つも簡約化された



Isabelle による正当性証明

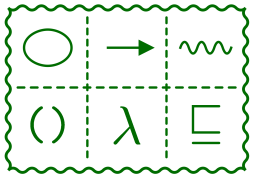


サブゴールを見て証明戦略を考える（人間の仕事）

使い方としては Isabelle にアドバイスを出してあげる感じになる

この前提を使いたい ...

条件が合わないので場合分けを考える



Isabelle による正当性証明

Isabelle2017 - before.thy (modified)

File Edit Search Markers Folding View Utilities Macros Plugins Help

before.thy (~/project/isabelle_theories/)

```
{True}
j := 0;
WHILE j < length v - 1 & (v!j = x ⟶ v!(j+1) = y)
  INV {j ≤ length v - 1 & (ALL i. i < j & v!i = x ⟶ v!(i+1) = y)}
  DO
  j:= j + 1
OD;
b := (j = length v - 1)
{b = (ALL i. i < length v - 1 & v!i = x ⟶ v!(i+1) = y)}"
apply(vcg)
apply(auto)
apply(case_tac "i = j")
```

Proof state | Auto update | Update | Search: | 100%

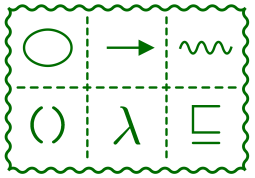
```
proof (prove)
goal (3 subgoals):
 1. ⋀j i.
       ∀ia. ia < j ∧ v ! ia = v ! i ⟶ v ! Suc ia = y ⟹
       j < length v - Suc 0 ⟹
       i < Suc j ⟹
       x = v ! i ⟹ v ! j ≠ v ! i ⟹ i = j ⟹ v ! Suc i = y
 2. ⋀j i.
       ∀ia. ia < j ∧ v ! ia = v ! i ⟶ v ! Suc ia = y ⟹
       j < length v - Suc 0 ⟹
       i < Suc j ⟹
       x = v ! i ⟹ v ! j ≠ v ! i ⟹ i ≠ j ⟹ v ! Suc i = y
 3. ⋀j i.
       ∀ia. ia < j ∧ v ! ia = v ! i ⟶ v ! Suc ia = v ! Suc j ⟹
       j < length v - Suc 0 ⟹
       i < Suc j ⟹
       x = v ! i ⟹ y = v ! Suc j ⟹ v ! Suc i = v ! Suc j
```

Output | Query | Sledgehammer | Symbols

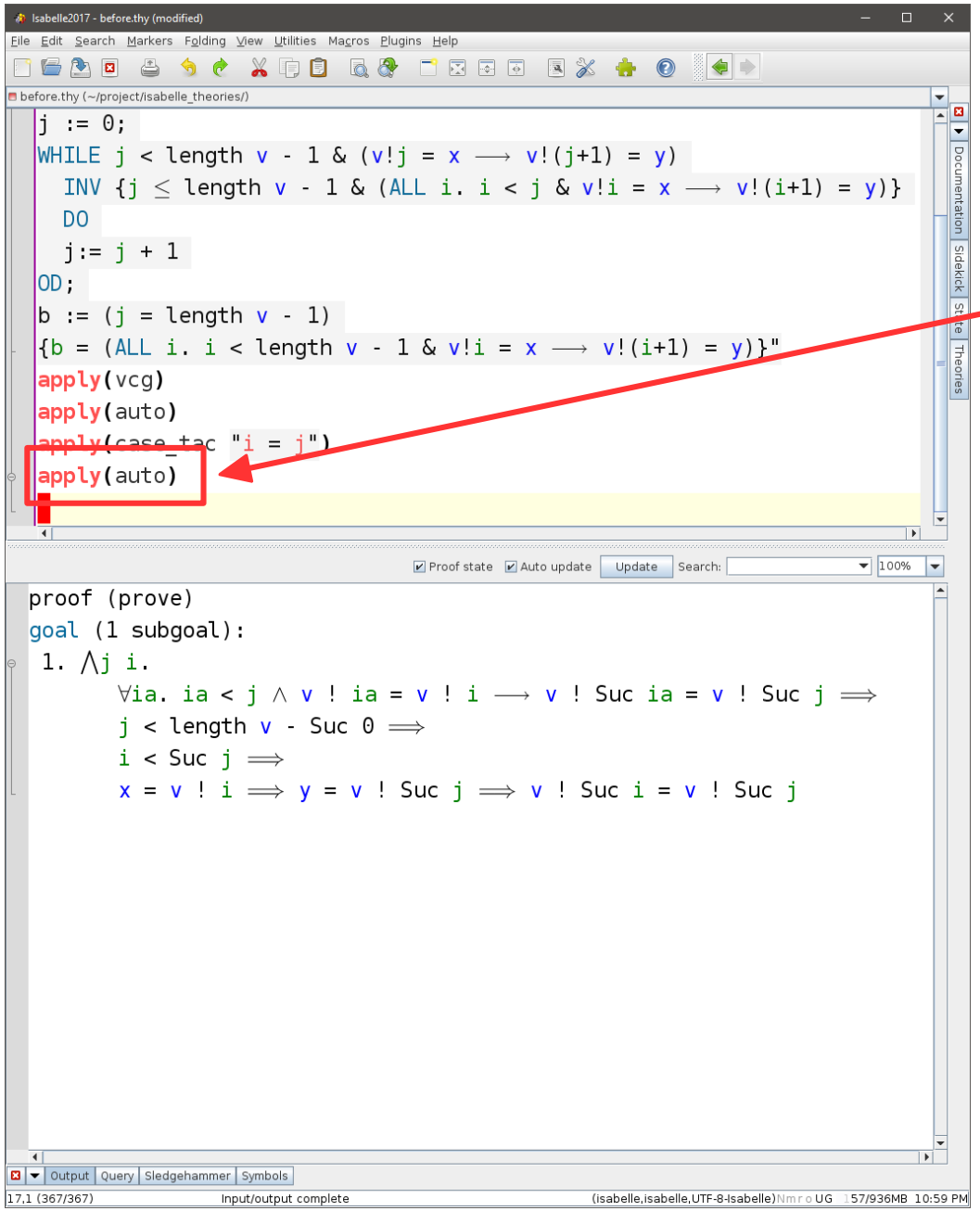
16,1 (355/355) (isabelle,isabelle,UTF-8-Isabelle) UG 116/936MB 10:57 PM

case_tac コマンドを使って i = j の場合とそれ以外で場合分けする

サブゴール 1 が i = j の場合と i ≠ j の場合に分かれたので，サブゴールが 3 つになった



Isabelle による正当性証明

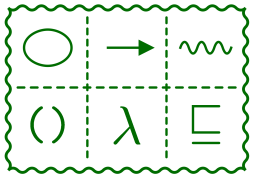


再び auto を発行

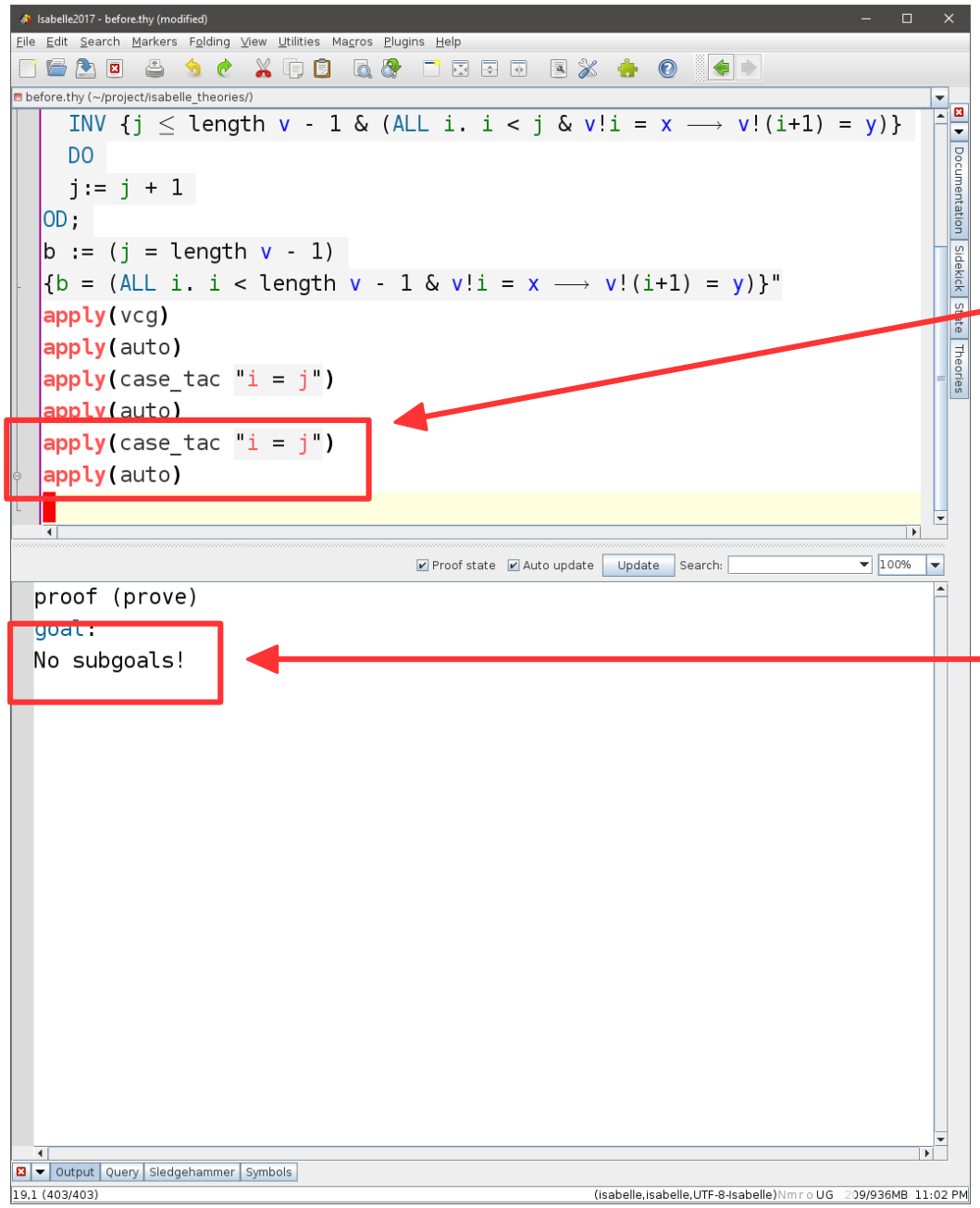
Isabelle が意図を汲んでくれた

サブゴール 2 つとも自動証明で消えた

実にかしこい



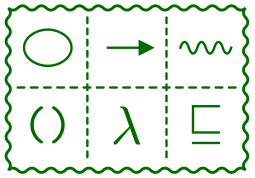
Isabelle による正当性証明



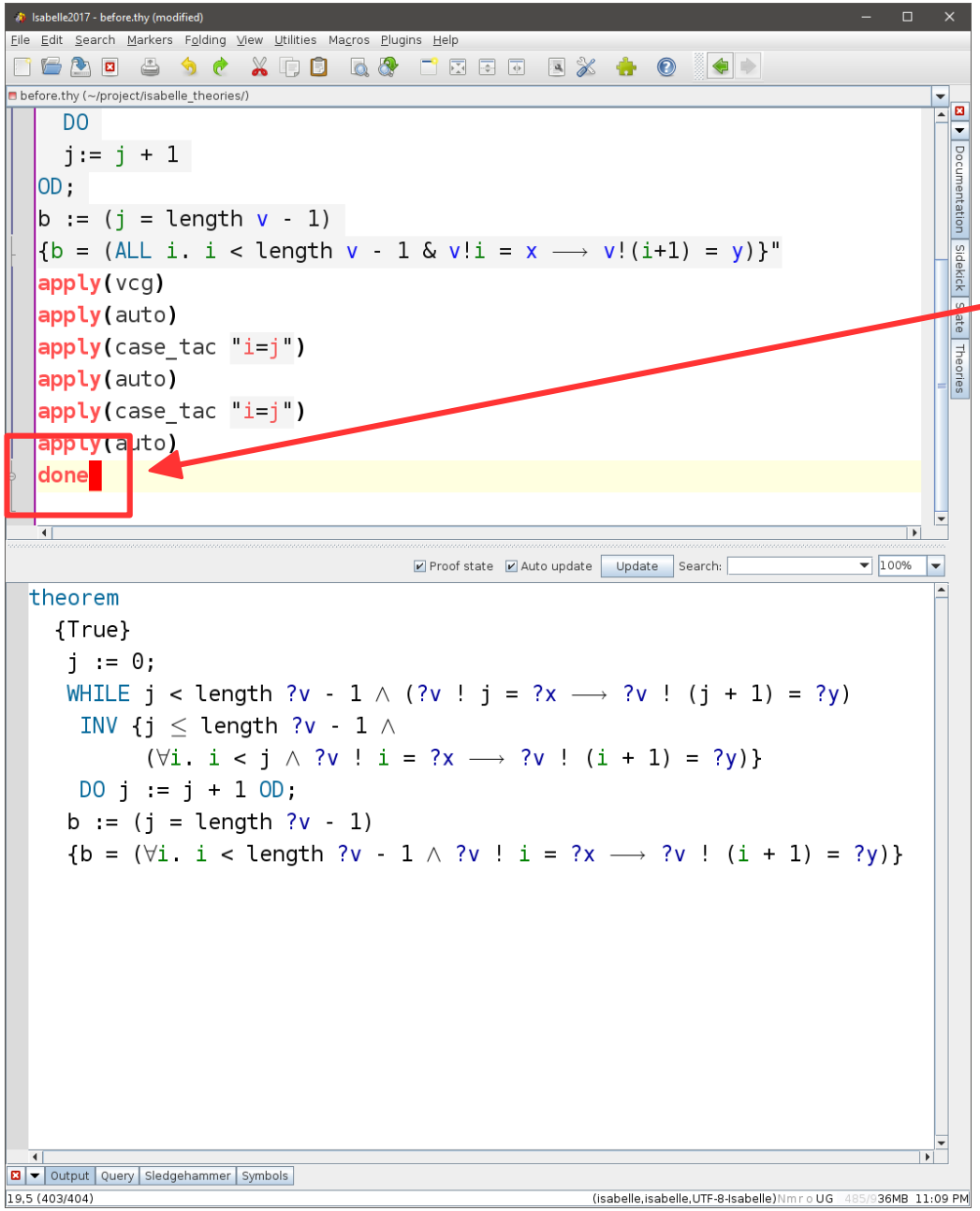
残ったゴールも似た形をしているので，同じ場合分けを適用

証明完了！

このプログラムは数学的な意味で正しさが証明された



Isabelle による正当性証明

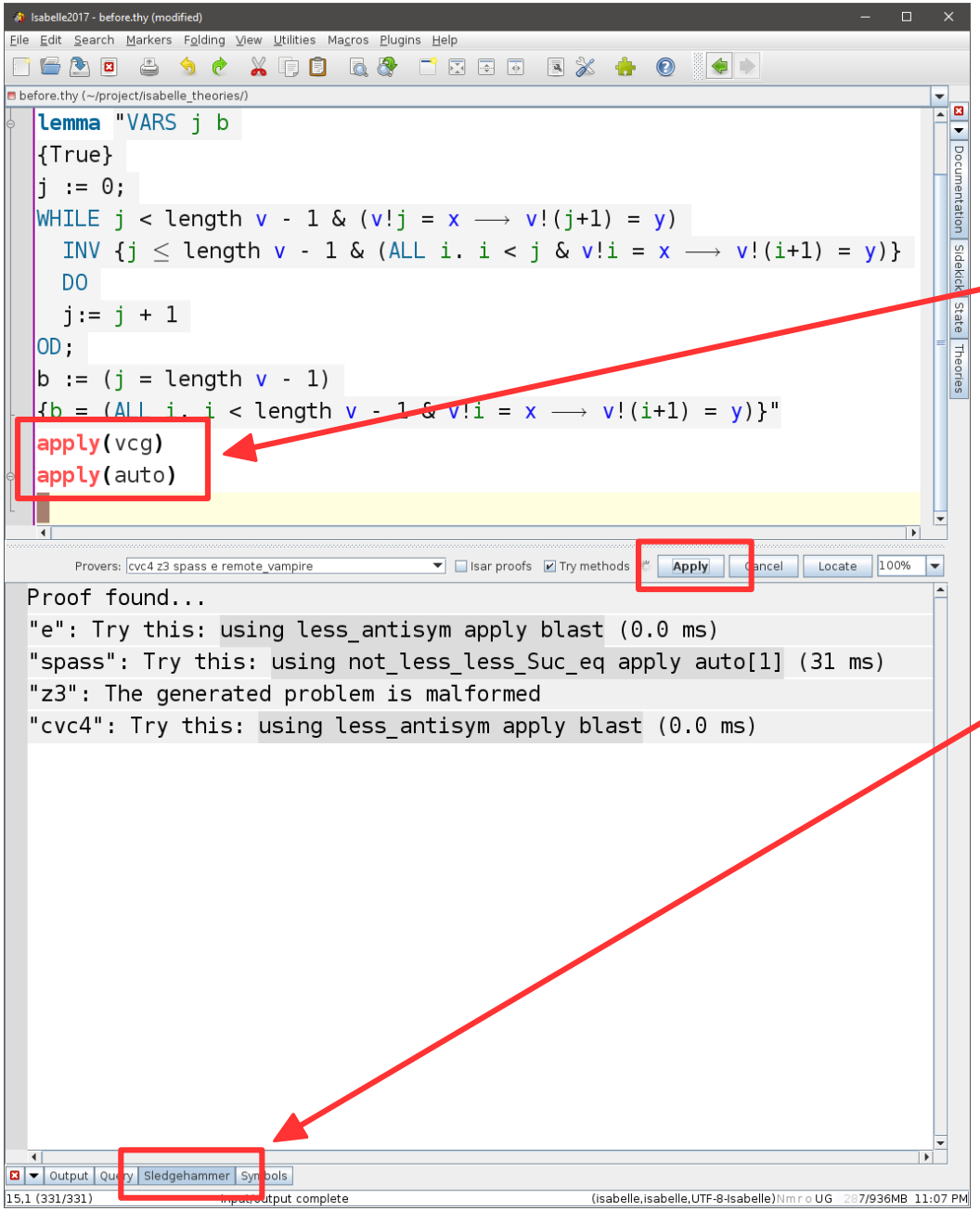


done コマンドで証明を閉じる

最後に証明した定理が表示される

数学の定理の場合はライブラリに登録され，後で利用できる

外部証明ツールの呼び出し



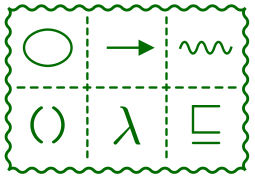
最初の auto まで戻った

Sledgehammer タブを開き

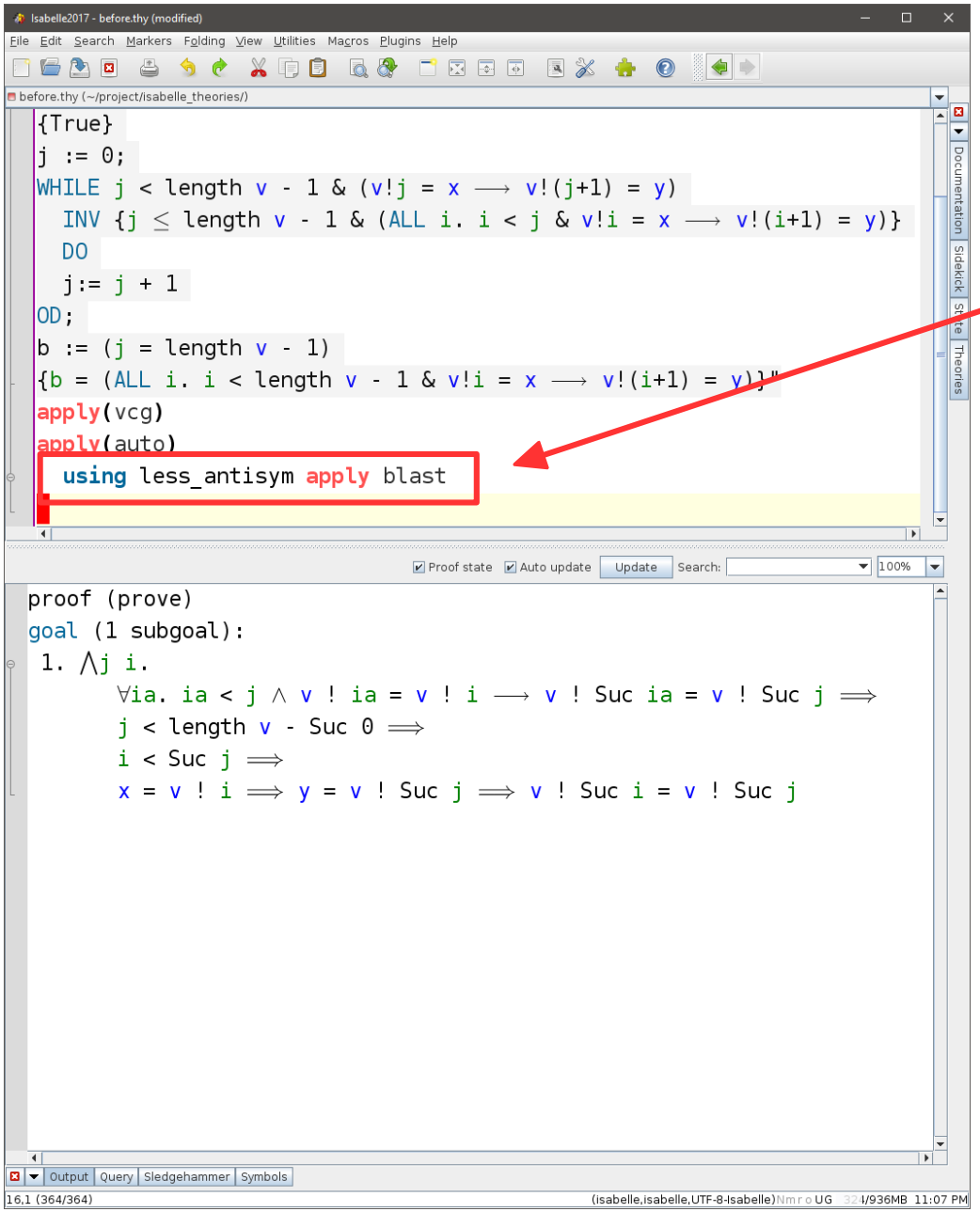
Apply ボタンを押すと

現在のゴールの証明を外注できる

ここでは cvc4, z3, spass,e に並行して依頼を出して z3 以外は瞬殺で答えを返してきた

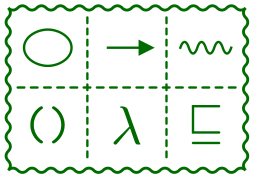


外部証明ツールの呼び出し

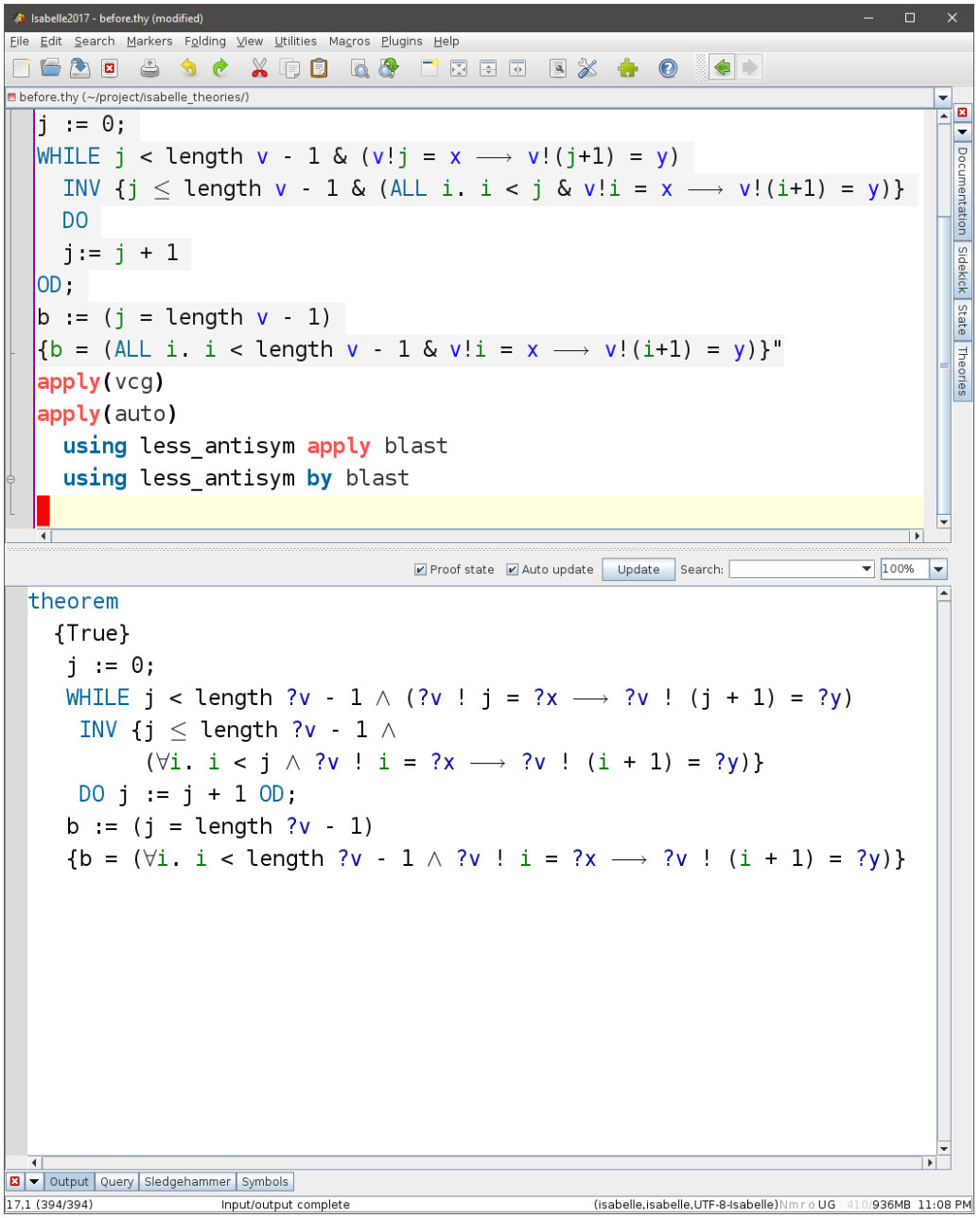


Sledgehammer の結果をクリックすると自動的にコマンド入力される

証明されたので
サブゴールが 1 つになった

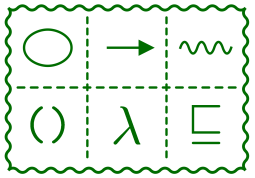


外部証明ツールの呼び出し



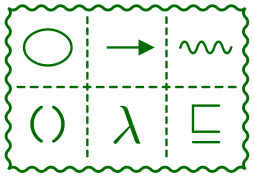
再度 Sledgehammer 実行

こちらも自動的に証明された

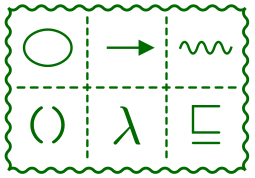


形式証明の実績例

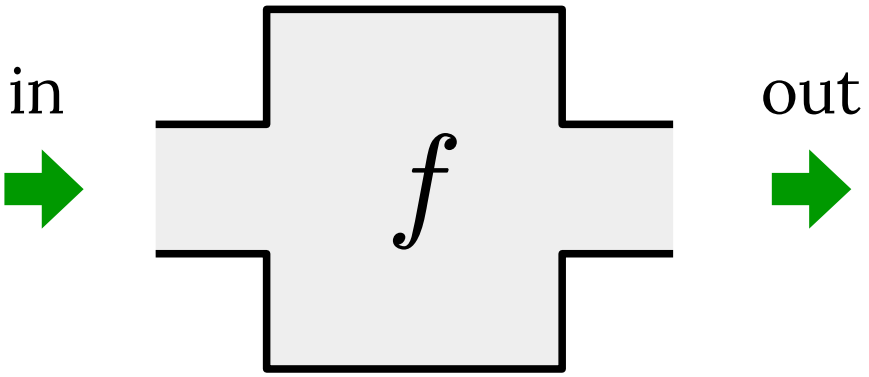
- The seL4 Microkernel (Isabelle)
  - https://researchgate.net/publication/220420663_seL4_Formal_Verification_of_an_Operating-System_Kernel
- The CompCert verified compiler (Coq)
  - http://compcert.inria.fr/doc/
- CakeML: A Verified Implementation of ML (HOL4)
  - https://cakeml.org/



リアクティブシステムの仕様と検証

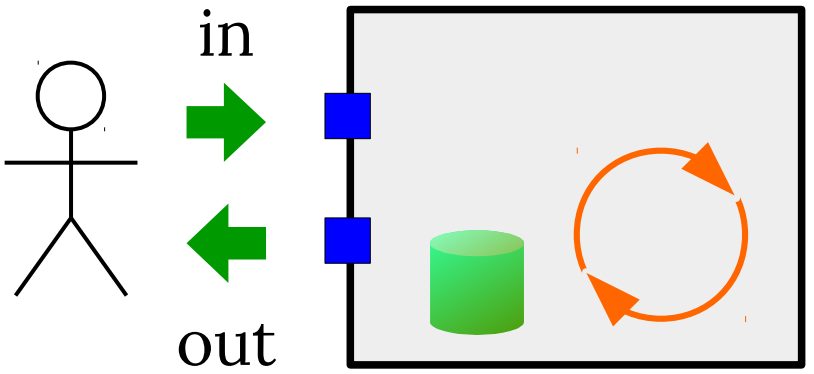


計算システムとリアクティブシステム



計算システム

入力が与えられると計算を実行し
結果を出力して停止する



リアクティブシステム

環境（ユーザや他のシステム）と
相互作用をしながら計算の実行や
サービスの提供を継続的に行う

- 自律的に動作する
- 内部状態を持つ
- 特別な場合を除き停止しない

相互作用とは何か？

基本相互作用

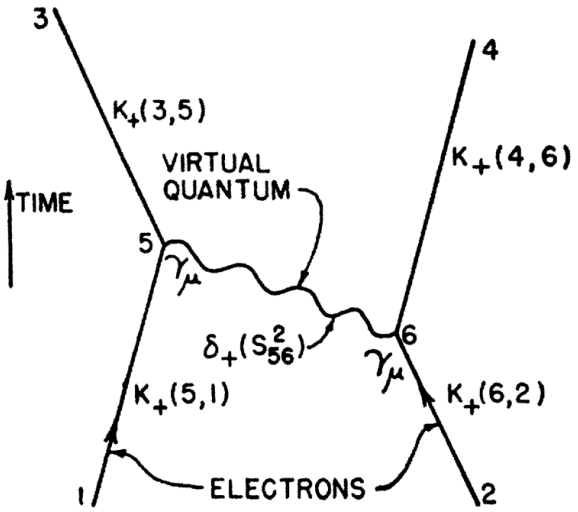
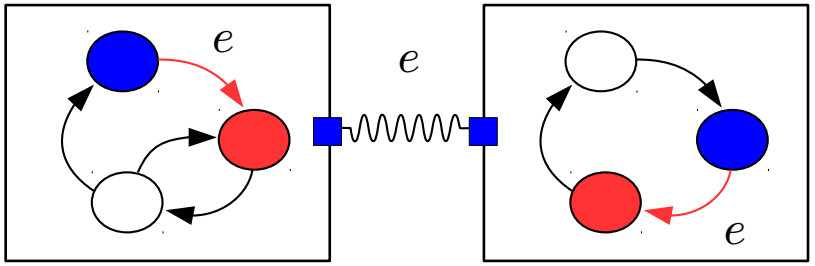


FIG. 1. The fundamental interaction Eq. (4). Exchange of one quantum between two electrons.

力を媒介する粒子を交換

Space-Time Approach to Quantum Electrodynamics,
R. P. Feynman, 1949

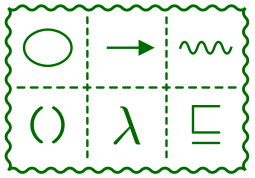
同期型相互作用



リアクティブシステムの基本動作は遷移である
遷移に付随して相互作用が発生すると考える
これをイベントと呼ぶ

リアクティブシステム間での相互作用は
それぞれの遷移が同時に発生することであると考える
これを同期型相互作用という※

※ 基本相互作用の決め方にはいくつもの流儀がある

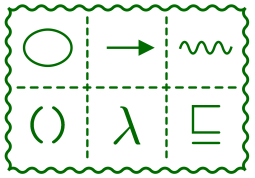


相互作用についての理論の必要性

Building communicating systems is not a well-established science, or even a stable craft; we do not have an agreed repertoire of constructions for building and expressing interactive systems, in the way that we (more-or-less) have for building sequential computer programs.

But nowadays most computing involves interaction - and therefore involves systems with components which are concurrently active. Computer science must therefore rise to the challenge of defining an underlying model, with a small number of basic concepts, in terms of which interactional behaviour can be rigorously described.

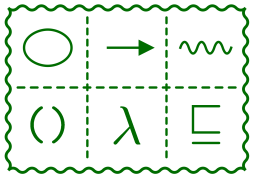
Communicating and Mobile Systems: the π-Calculus, Robin Milner, 1999



CSP とは

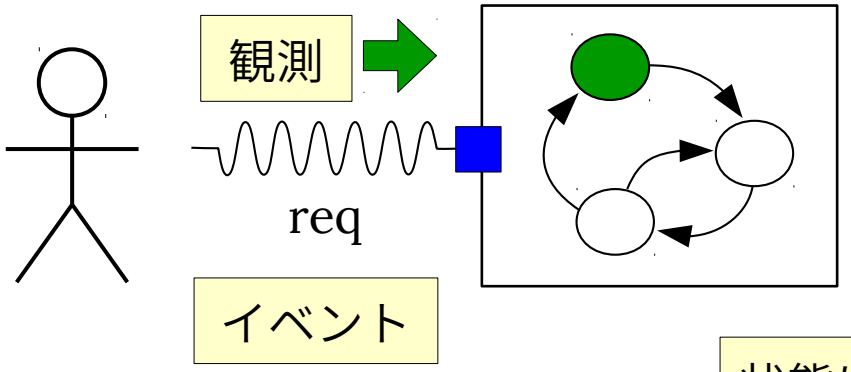
- Communicating Sequential Processes
- 並行システムの振る舞いを記述し，その性質を論証するための数学的理論
  - プロセス代数と呼ばれる理論の 1 つ（他にπ計算など）
- 1978 年 C. A. R. Hoare によって提案された
- 特徴
  - 詳細化関係の定義
  - 豊富な意味論

Photograph by Rama, Wikimedia Commons, Cc-by-sa-2.0-fr



リアクティブシステムの仕様とは？

システムの動作とは外部から観測できるものである



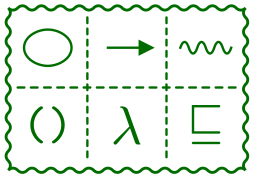
トレース　⟨req, ack, calc, wait, ...⟩

状態は観ることができない※

観測されるイベントを時系列で記録する
これをトレースという
システムの発生しうるトレースの全体が
システムの動作であると考える

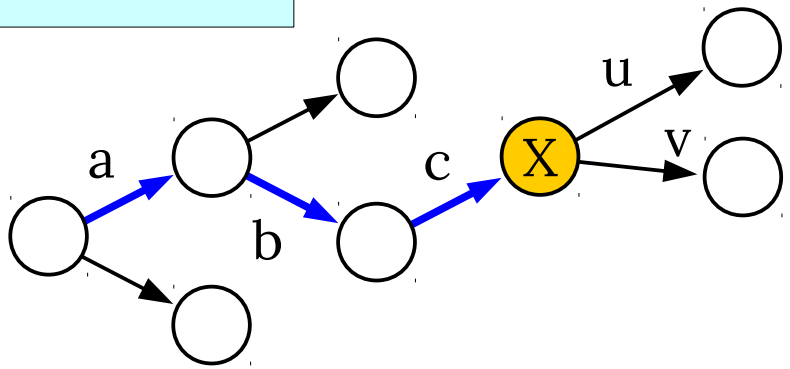
トレースの集合を規定するものを
システムの仕様と考えることができる

※ 相互作用とは別の超越的手段で状態が観測可能とする立場もある

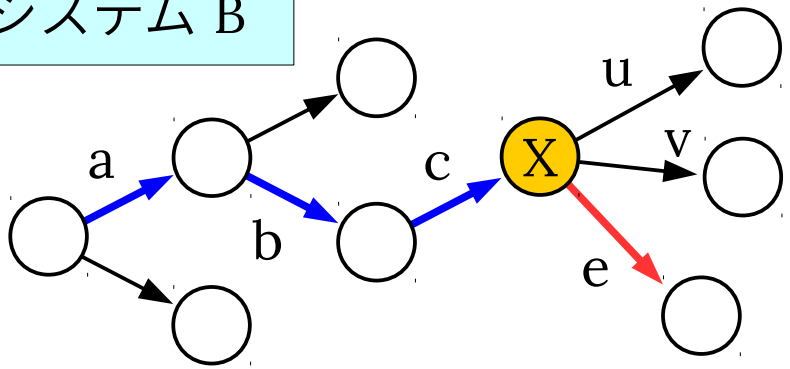


リアクティブシステムの正当性条件

システム A



システム B



仕様 S

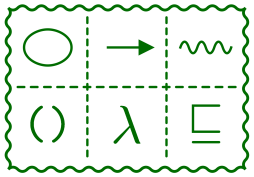
$\langle\rangle$
$\langle a\rangle$
$\langle a, b\rangle$
$\langle a, b, c\rangle$
$\langle a, b, c, u\rangle$
$\langle a, b, c, v\rangle$
...

違反 $\langle a, b, c, e\rangle \notin S$

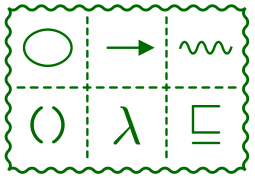
リアクティブシステムの正当性条件

システムが生成するトレースは仕様に含まれていなければならない※

※ 有限長トレースの場合は必要条件



並行・並列システム開発の難しさと対応

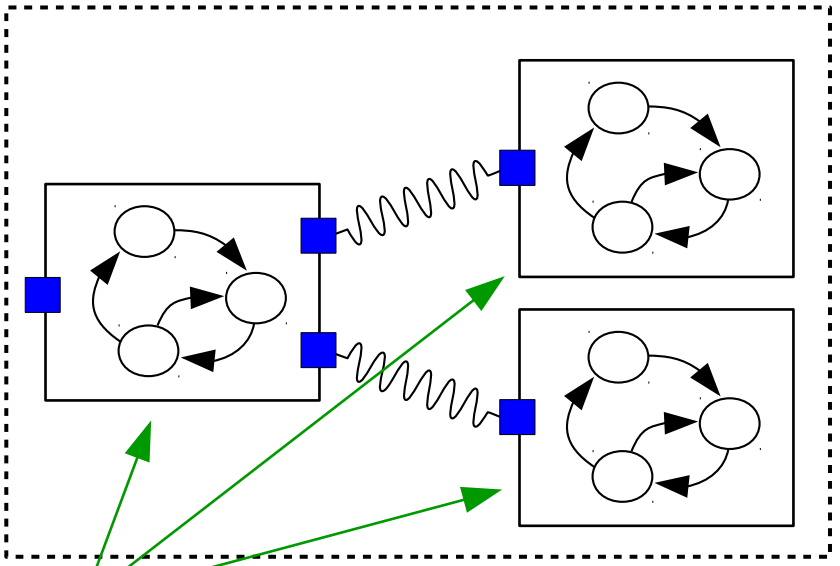


設計における分割と合成

正当性

$S \sqsubseteq$

仕様

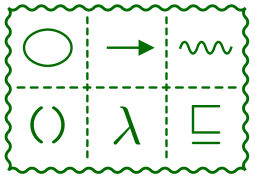


設計で決めるのは
コンポーネントの動作

合成系

合成系の動作はコンポーネントの
動作から完全に決定される

正当性を満たすようにコンポーネントの動作を規定することは難しい



なぜ並行システムを考えるのか？

物理的世界は並行システム

外界やシステムを構成するデバイスの中には
物理法則にしたがい自律的に動作するものがある

設計手法

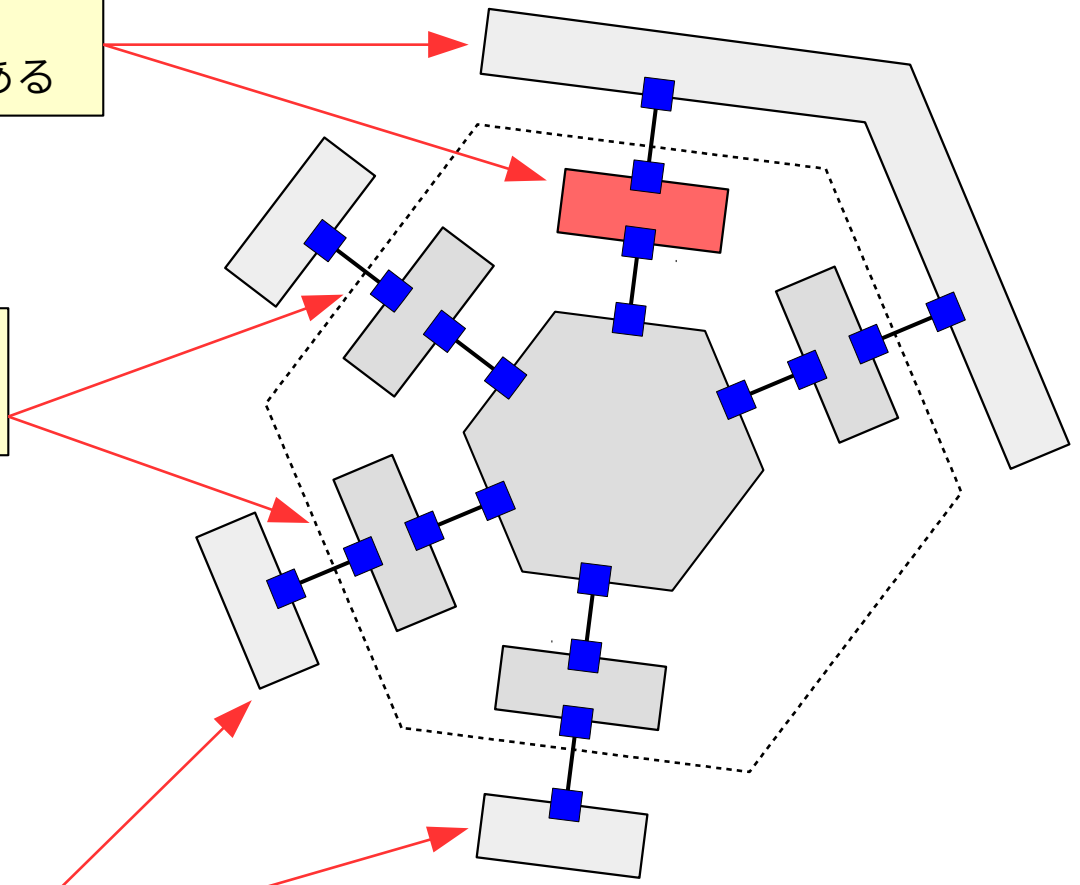
外界と接する境界面ごとにコンポーネント
を用意する設計手法は概念的に明解である※

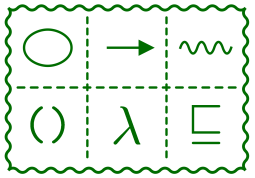
※ しかし振る舞い的に明解になる保証はない

パフォーマンスや応答性の向上

異なるコンポーネント上で計算を並列に実
行できればパフォーマンスを向上できる

各監視対象に専用のコンポーネントを割り
当てれば応答性を改善できる





並行システム特有の難しさ

- 状態爆発

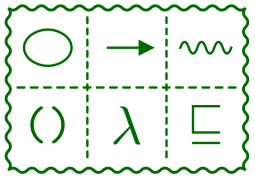
  合成系の状態は各コンポーネントの状態の組になるので，合成系の状態数はコンポーネントの数に対して指数関数的に増加する．状態数が多ければ検証はそれだけ困難になる

- 非決定性※

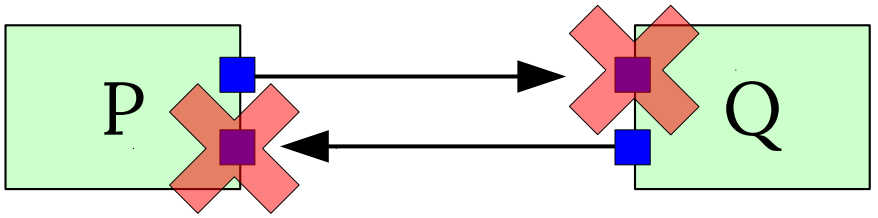
  システムが同じ状態，同じ環境に置かれていても動作が異なる場合がある．非決定性を持つシステムの動作をテストによって保証することはできない

- デッドロック
- ライブロック

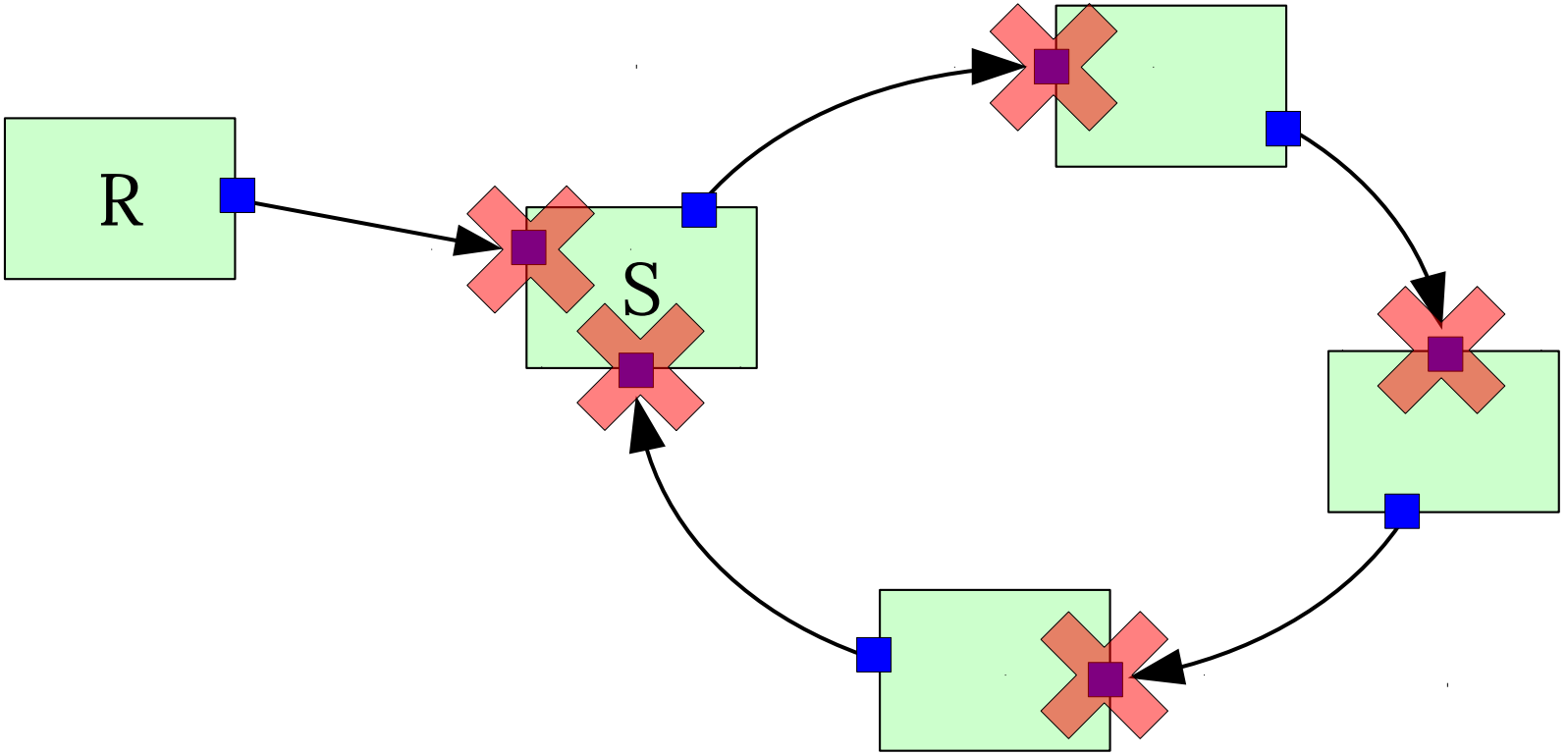
※ 理論的には逐次システムにも非決定性はあるが現実のプログラムに現れることはほとんどない（ ref. Go のライブラリにあるハッシュテーブル）

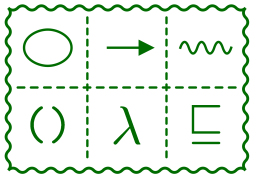


デッドロック



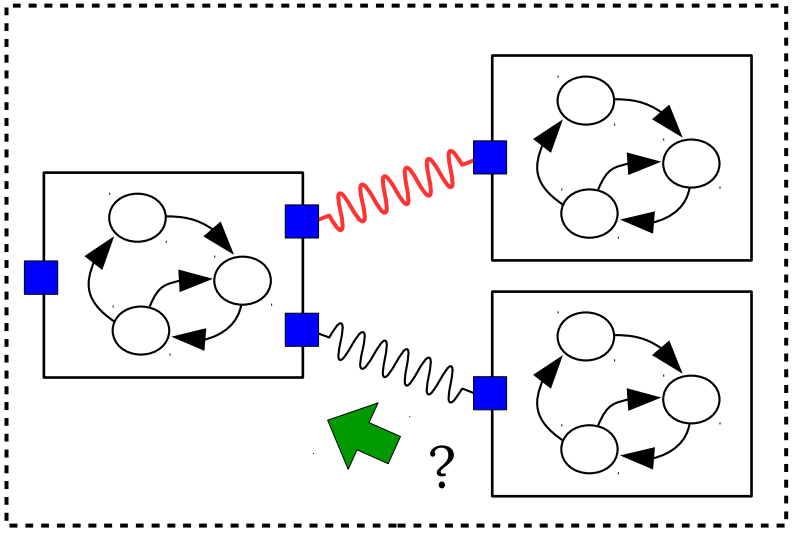
受理されない要求が循環すると
誰も動けなくなってしまう
この現象をデッドロックという

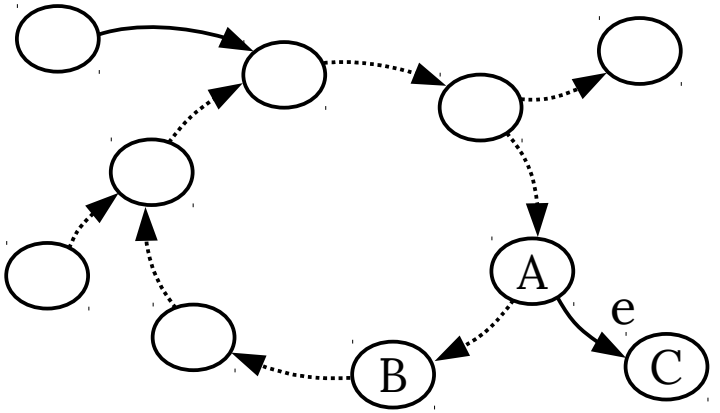


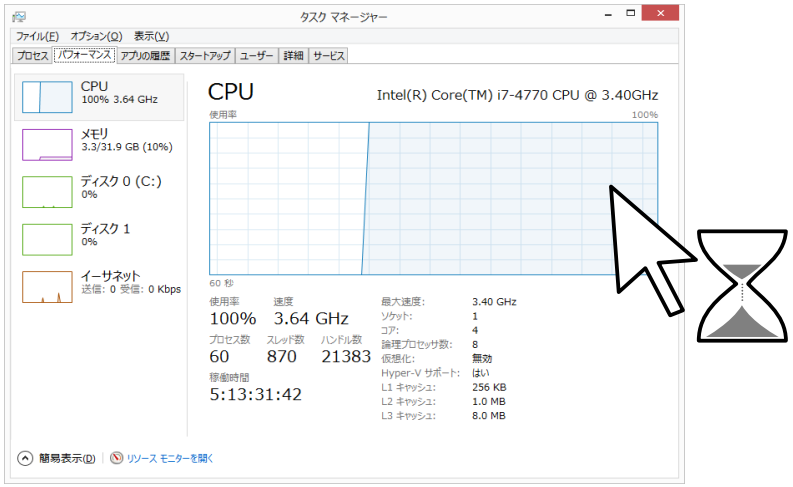


発散（ライブロック）

- 内部遷移の循環ができると，自発的な遷移を繰り返し外部と相互作用ができなくなる．この現象を発散 (divergence) またはライブロックという

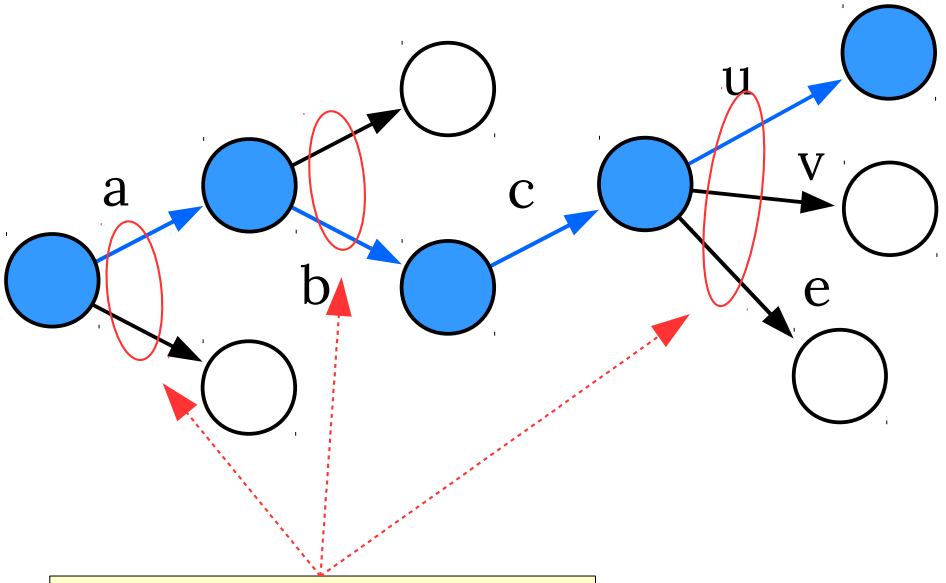






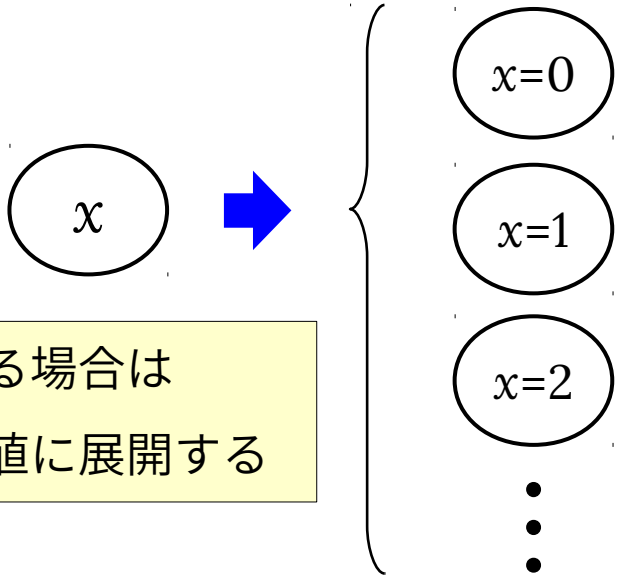
モデル検査とは

システムが生成しうるすべてのトレースを自動的に計算し
仕様に含まれているかどうか検査する力技の方法



すべての分岐を調べる

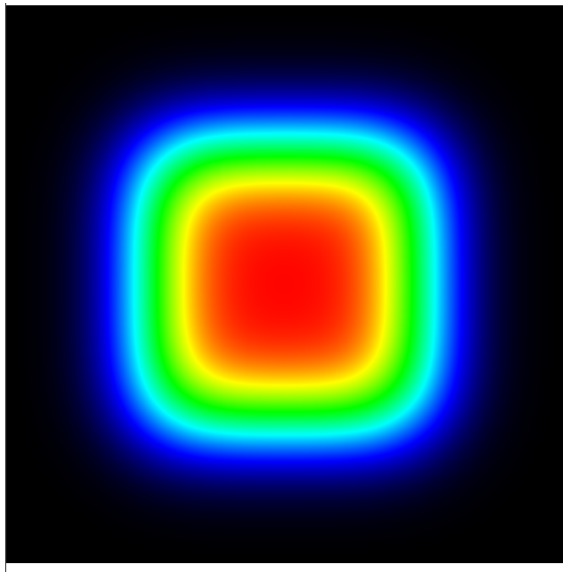
シミュレーションやテストは
1つの経路について調べる



変数がある場合は
具体的な値に展開する

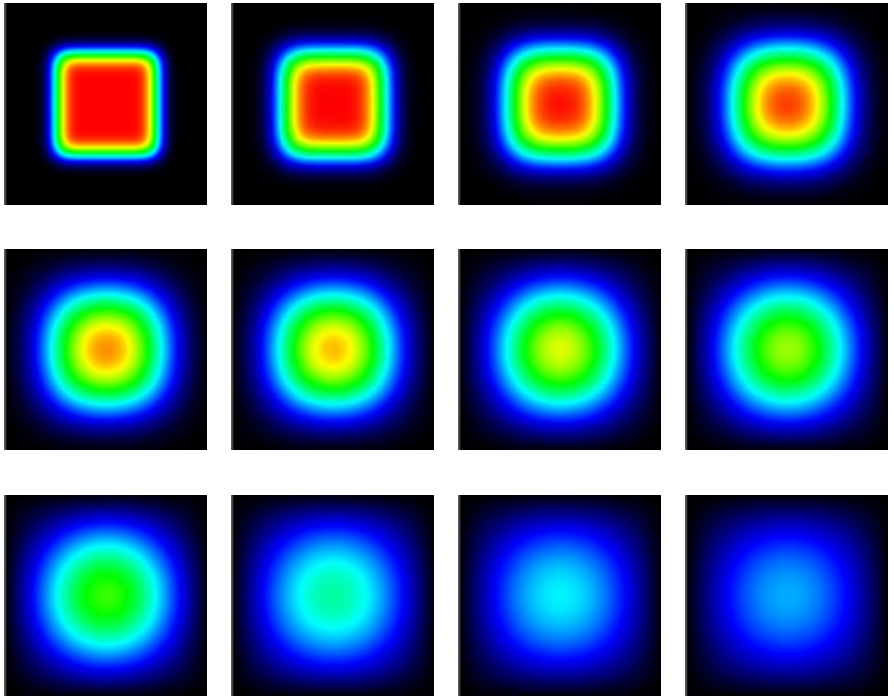
モデル検査のイメージ※

シミュレーション

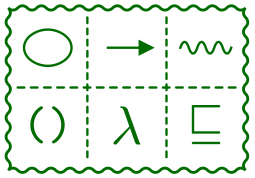


系を特徴づけるパラメータ値を
1つ選択して計算する

モデル検査



可能性のあるパラメータ値について
すべて網羅的かつ自動的に検査する



モデル検査の種類

時相論理によって仕様を記述する

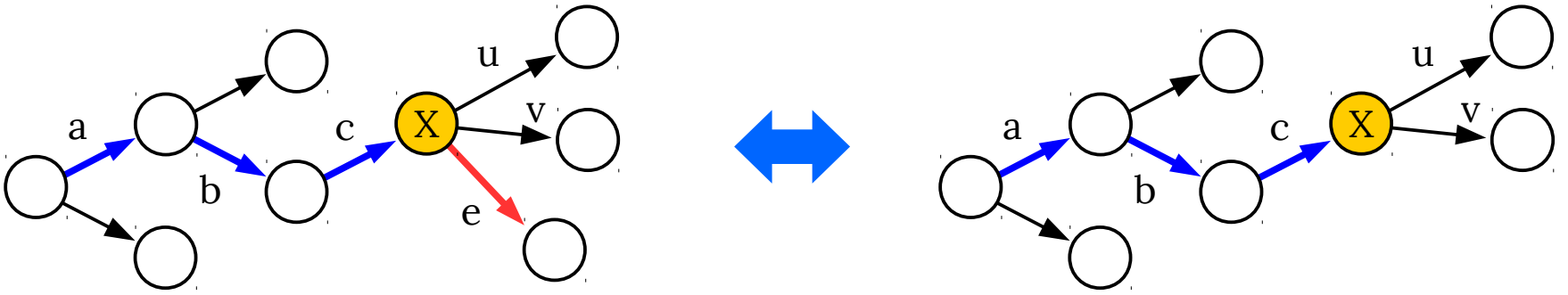
※ 時相論理にも種類がある

時間についての演算子 □, ◇ で拡張した論理式を使って仕様を表す

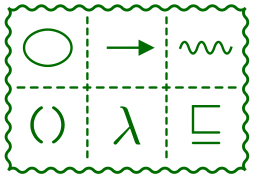
$\Box p$	常に p が成り立つ	$\Box \neg \mathrm{error}$	決してエラーにはならない
$\Diamond p$	いずれ p が成り立つ	$\Diamond \mathrm{print}$	いずれ印刷される

遷移系によって仕様を記述する

2 つの遷移系の動作全体を比較できる

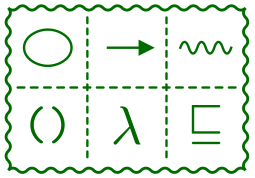


例：画面遷移仕様と設計モデルを比較する

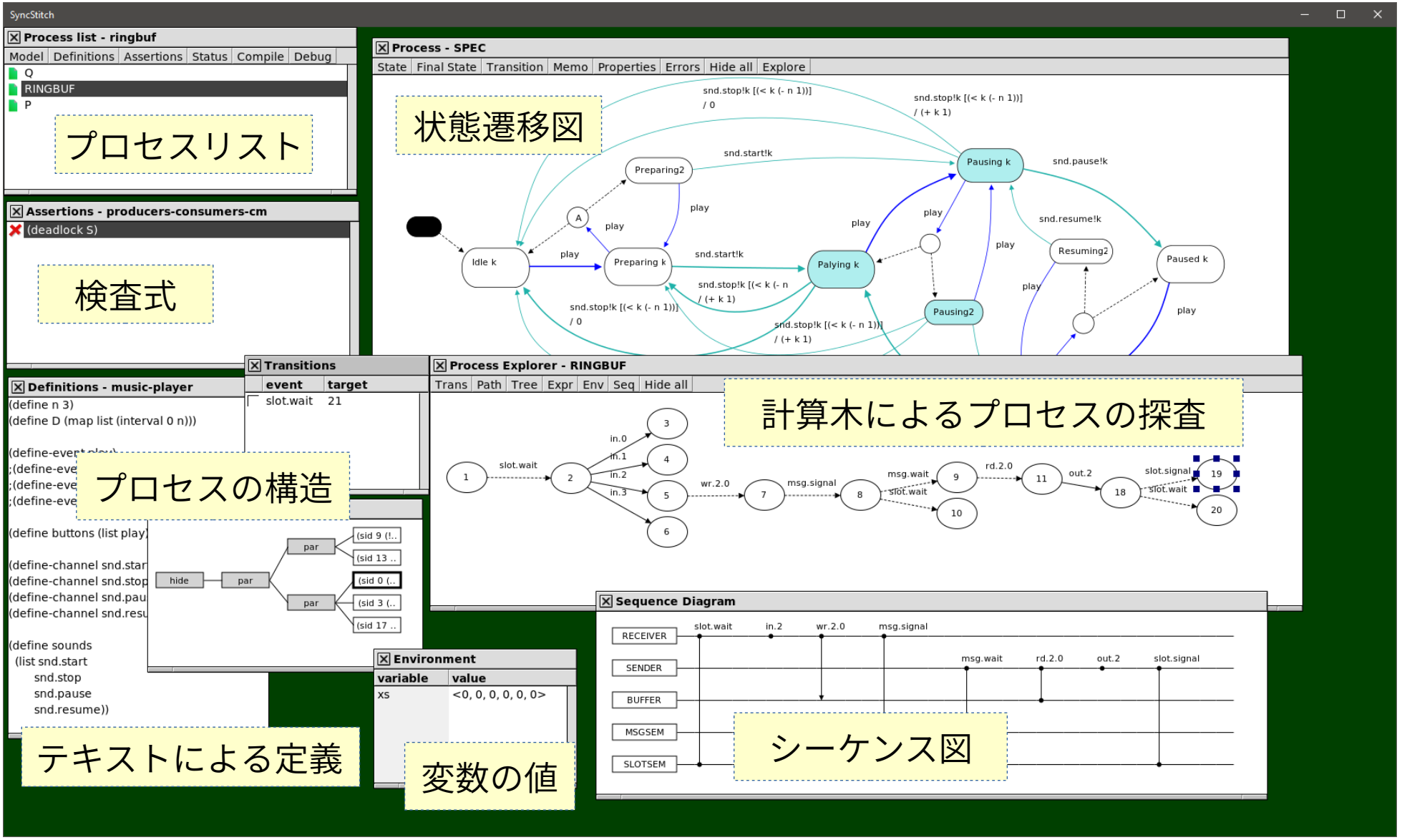


SyncStitch

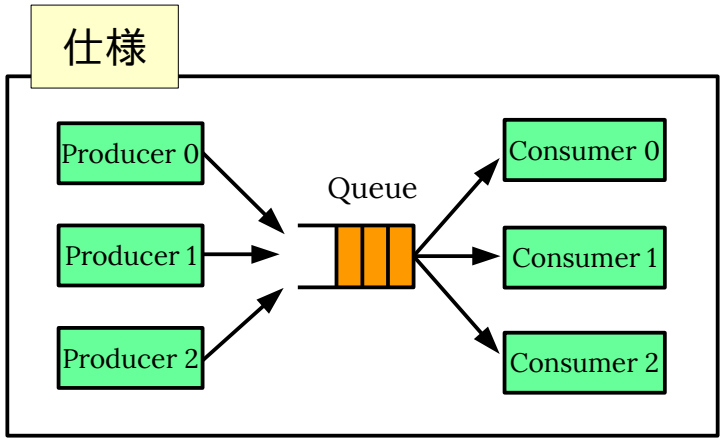
- CSP に基づく並行システムの振る舞いモデル化・検査ツール
- 2 種類のモデル記述方法
  - 状態遷移図
  - モデリング言語 : CSP + 型付き関数型言語
- 4 種類の検査
  - デッドロック
  - 発散
  - トレース集合による詳細化検査
  - 安定失敗集合による詳細化検査
- 分析ツール
  - 計算木、シーケンス図、プロセス構造表示、変数表示
- 並列検査エンジン



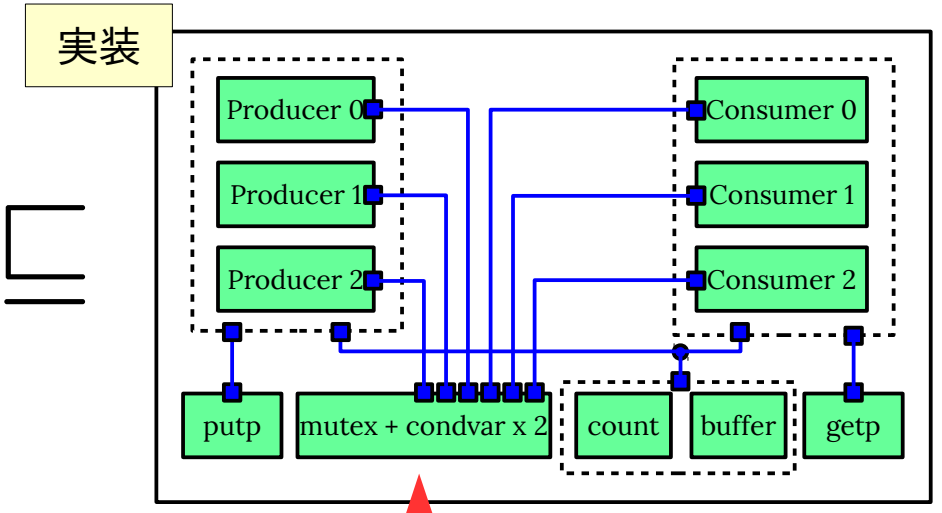
SyncStitch



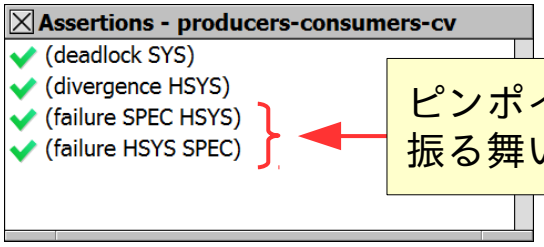
検査例：生産者・消費者問題



⊏



条件変数と
リングバッファ
による解



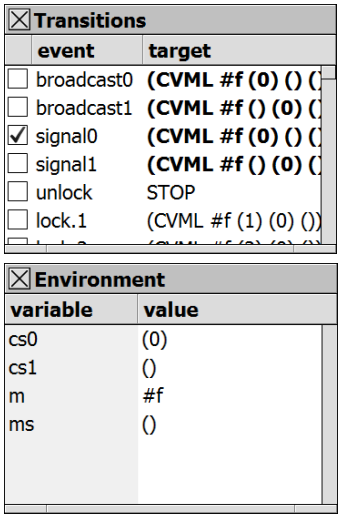
ピンポイントの性質ではなく
振る舞い全体について検査できる

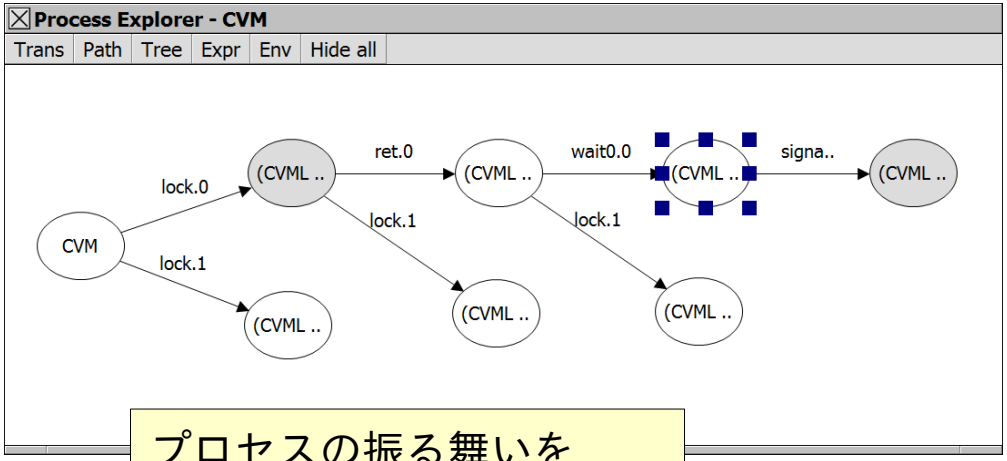
各プロセスはモックやスタブなしに単独で
分析できる（イベント同期型相互作用）

仕様は全体 / 部分，抽象 / 具象
など目的に応じて記述する

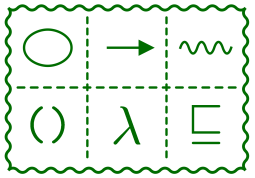
モデルを対話的に修正し検査する
ことで，さまざまな設計を短時間
で確実に試行できる．

有限のモデルしか検査できない





プロセスの振る舞いを
計算木としてみるビュー



分析ビュー

Status - producers-consumers-cv1-deadlock

1701 states
the state space diameter 13
Unfolding SYSTEM
4356848 states
the state space diameter 200
Determinizing SPEC
112 states
Calculating minimal acceptances for SPEC
Checking refinement relation: SPEC [F= SYSTEM

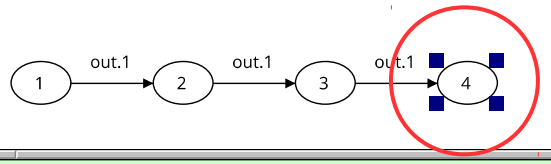
仕様

Transitions

event	target
in.0	5
in.1	**3**
out.0	6
out.1	7

Process Explorer - SPEC

Trans | Path | Accept | Compo | Tree | Expr | Env | Hide all



違反が発見された状態

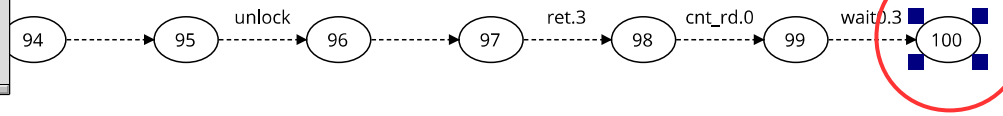
実装

Transitions

event	target
in.0	101
in.0	102
in.1	103
in.1	104
out.1	105

lorer - SYSTEM

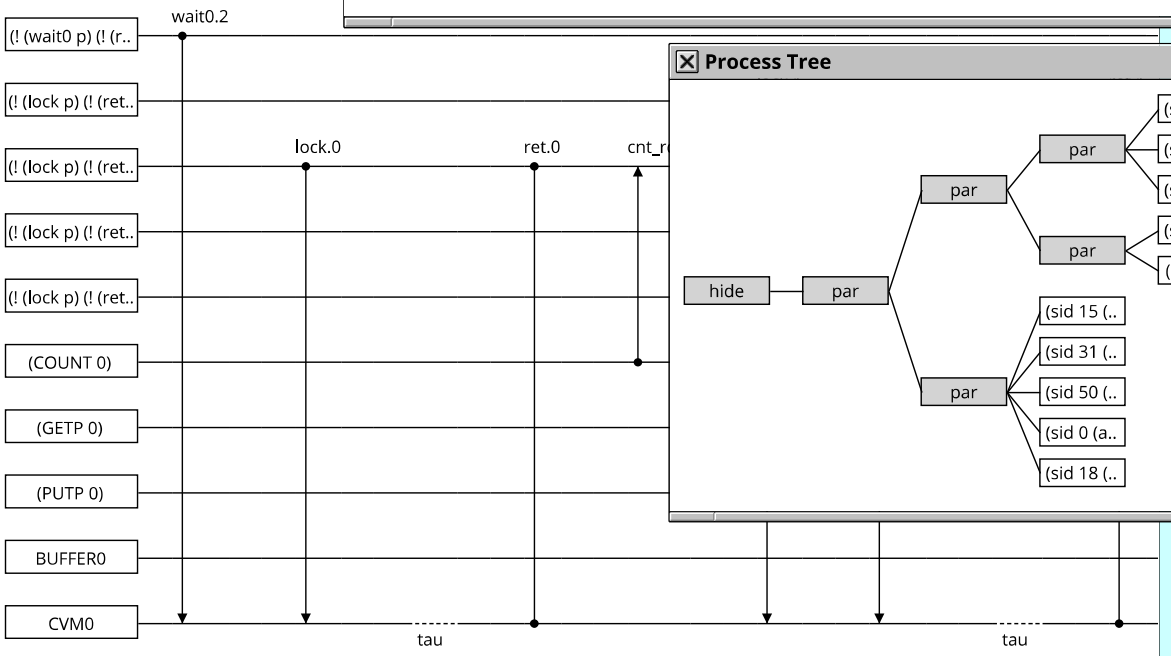
Tree | Expr | Env | Seq | Hide all



Path

	event	target
74	wait0.3	76
75	tau	77
76	ret.2	78
77	cnt_rd.0	79
78	putp_rd.1	80
79	wr.1.1	81
80	putp_wr.0	82
81	cnt_wr.1	83
82	signal0	84
83	tau	85
84	unlock	86
85	tau	87
86	ret.4	88
87	cnt_rd.1	89
88	getp_rd.1	90
89	rd.1.1	91
90	getp_wr.0	92
91	cnt_wr.0	93
92	signal0	94
93	tau	95
94	unlock	96
95	tau	97
96	ret.3	98
97	cnt_rd.0	99
98	wait0.3	100

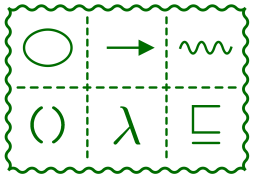
Sequence Diagra



Process Tree

Environment

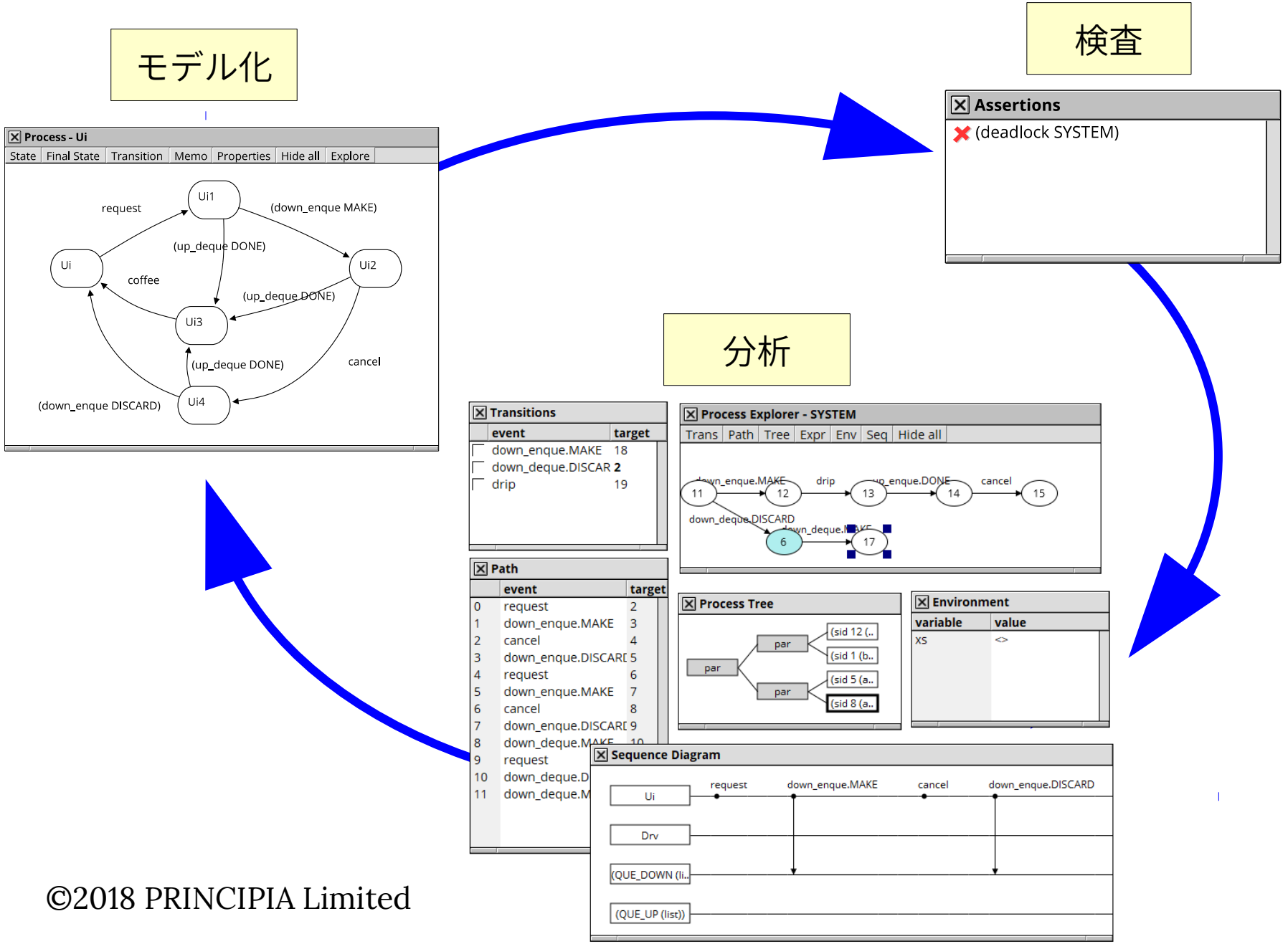
variable	value
m	Unlock
ms	{}
cs0	{3, 1}
cs1	{}

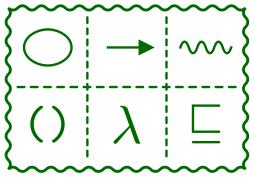


設計は試行錯誤

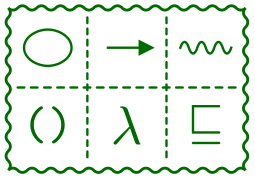
- 表現を作ることで自己フィードバックがかかる
  - 考えが整理される
  - 新しいアイデアが生まれる
  - 間違いに気づく
- 表現を人間が見るためだけのものではもったいない
  - ツールを使えば変換や検査ができる
- 対話的なモデリング・検査ツールは設計を強力に支援する

設計は試行錯誤



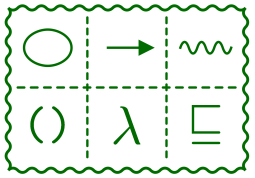


まとめ



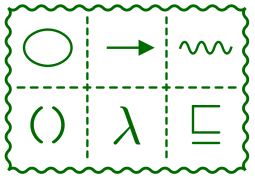
まとめ

- 形式手法はソフトウェアを工学にする
  - 計算機科学を基礎とする
  - 数学を使って厳密な表現を作り検証する
- 主な形式手法
  - 形式仕様
  - 形式検証
  - モデル検査
- 形式手法の効果
  - 開発効率を向上することができる（設計は試行錯誤）
  - 成果物の品質を高める



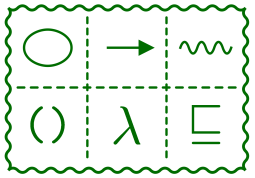
参考文献・リンク：形式手法全般

- ソフト・エッジ : ソフトウェア開発の科学を求めて，中島 震，みわ よしこ，2013
- 「形式手法と仕様記述」の探求に向けて〜形式手法とは？〜，石川 冬樹
  - http://research.nii.ac.jp/~f-ishikawa/work/files/2012-05-11-SQiP-intro.pdf
- 高信頼・高安全ソフトウェアのための数理的検証手法：最新の研究と応用の現状，住井 英二郎
  - http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/RSA.ppt.pdf
- ソフトウェア科学基礎：最先端のソフトウェア開発に求められる数理的基礎，磯部 祥尚 , 粂野 文洋 , 櫻庭 健年 , 田口 研治著 , 本位田 真一監修， 2009
- コンピュータサイエンス入門〈2〉論理とプログラム意味論，田辺 誠 , 中島 玲二 , 長谷川 真人， 1999
- Logic in Computer Science: Modelling and Reasoning about Systems, Michael Huth, Mark Ryan, 2004



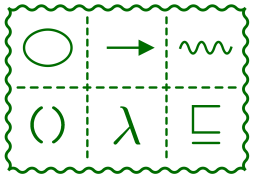
参考文献・リンク：形式仕様

- 手法
  - Z
    - Using Z: Specification, Refinement, and Proof
      - http://cs.cmu.edu/~15819/zedbook.pdf
  - VDM++
    - 仕様の品質を飛躍的に高める手法 VDM++ によるオブジェクト指向システムの高品質設計と検証，酒匂寛 訳，ジョン フィッツジェラルド，ピーター ゴルム ラーセン，ポール マッカージー，ニコ プラット，マーセル バーホフ 著， 2010 ，翔泳社
- 事例
  - フェリカネットワークス
    - https://ipa.go.jp/files/000044283.pdf
  - ライフロボティクス
    - http://jasst.jp/symposium/jasst17tokyo/pdf/D3.pdf
  - 厳密な仕様記述における形式手法成功事例調査報告書 (IPA)
    - https://ipa.go.jp/sec/softwareengineering/reports/20130125.html



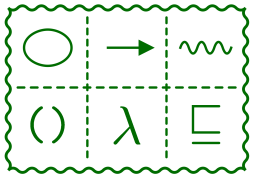
参考文献・リンク：形式検証

- 理論・テキスト
  - プログラム検証論 (情報数学講座 8)，林 晋 著，1995,，共立出版
  - プログラミングの科学 (情報処理シリーズ)，D. グリース (著), 筧 捷彦 (翻訳)，1991，培風館
  - Coq/SSReflect/MathComp による定理証明: フリーソフトではじめる数学の形式化，萩原 学 (著), アフェルト・レナルド (著)，2018，森北出版
  - Concrete Semantics with Isabelle/HOL
    - http://concrete-semantics.org
- ツール
  - Isabelle
    - https://isabelle.in.tum.de/
  - Coq
    - https://coq.inria.fr/
- 事例
  - The seL4 Microkernel
    - https://sel4.systems
  - The CompCert verified compiler
    - http://compcert.inria.fr/doc/
  - CakeML: A Verified Implementation of ML
    - https://cakeml.org/



参考文献・リンク：モデル検査

- テキスト
  - 並行システムの検証と実装：形式手法CSPに基づく高信頼並行システム開発入門，磯部祥尚著， 2012
  - Model Checking, Edmund M. Clarke Jr., Orna Grumberg, Doron Peleg, Calin Belta, 1999
  - SPIN Model Checker, The: Primer and Reference Manual, Holzmann, 2003
  - Concurrency: State Models and Java Programs, Jeff Magee, Jeff Kramer, 2006
- ツール
  - SyncStitch
    - https://principia-m.com/syncstitch/
  - FDR4
    - https://cs.ox.ac.uk/projects/fdr/
  - PAT: Process Analysis Toolkit
    - http://pat.comp.nus.edu.sg/
  - LTSA - Labelled Transition System Analyser
    - https://doc.ic.ac.uk/ltsa/
  - Spin Model Checker
    - http://spinroot.com/spin/whatispin.html
  - NuSMV
    - http://nusmv.fbk.eu/



参考文献・リンク：その他

- CSP
  - http://www.usingcsp.com/
    - 邦訳：ホーア CSP モデルの理論
- π計算
  - Communicating and Mobile Systems: The Pi Calculus, Robin Milner, 1999
- Event-B
  - Event-B: リファインメント・モデリングに基づく形式手法，中島 震 , 來間 啓伸 著， 2015
- CafeOBJ
  - モデルの記述と検証のためのプログラミング入門― CafeOBJ による仕様検証，二木 厚吉， 2017
- Alloy
  - 抽象によるソフトウェア設計ー Alloy ではじめる形式手法ー， Daniel Jackson 著， 中島 震 (監訳), 今井 健男 (翻訳), 酒井 政裕 (翻訳), 遠藤 侑介 (翻訳), 片岡 欣夫 (翻訳) , 2011
- SAT/SMT solver
  - SAT/SMT ソルバの仕組み，酒井 政裕
    - https://slideshare.net/sakai/satsmt