東北大学「情報科学特論II」
ソフトウェアの基礎理論と検証技術

亀山 幸義（筑波大学　電子・情報工学系）
http://www.is.tsukuba.ac.jp/~kam

2002.10.23

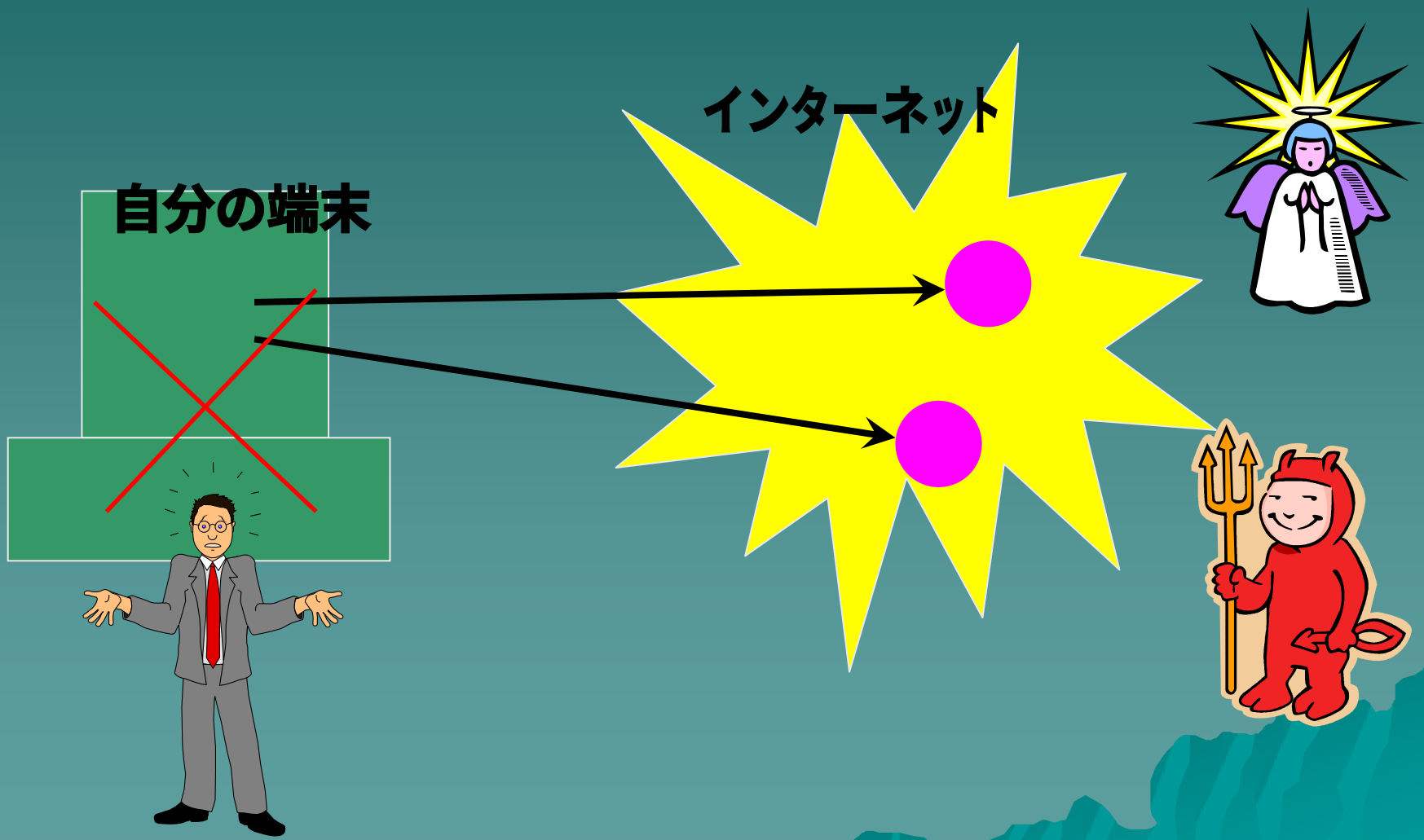
情報社会の将来

- 目に見える変化
  - コンピュータのハードウェアは日進月歩
  - 様々な新しい情報機器の登場、家電の情報化
  - etc. etc. etc.
- 情報社会の鍵は？
  - ソフトウェア
    - ソフトウェアの高機能化、高性能化、使いやすさの向上
    - あらゆる機器が情報化（知能化）される
- 情報社会の弱点は？
  - ソフトウェア

ソフトウェアがひきおこした社会的問題

- 枚挙に暇がない
  - Y2K(西暦2000年) 問題
  - スペースシャトルの打上げ延期
  - 銀行のオンラインシステムのダウン
  - 携帯電話 (電話機) のたび重なる回収
  - ...

インターネットからのソフトウェア取得



未来社会とソフトウェア科学

- 未来の社会
  - 社会の隅々に情報機器がゆきわたる
  - 社会の隅々にソフトウェアがゆきわたる
  - 社会の隅々に不安定要因がゆきわたる？？
- ソフトウェアの目指す方向
  - 高機能、高性能であること
  - 使いやすいこと
  - 安定していること、**高い信頼性**をもつこと

本日の内容

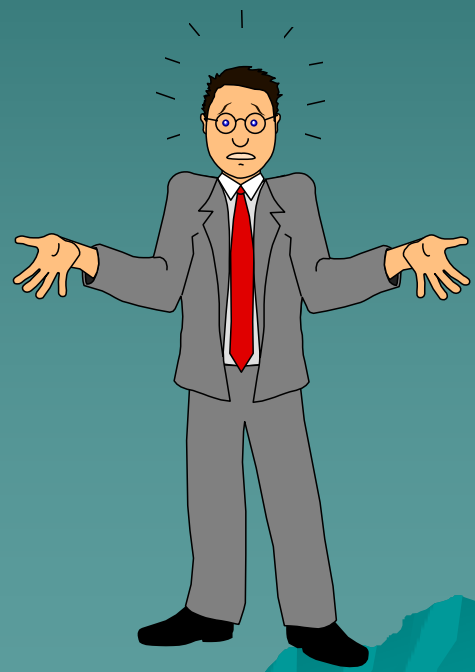
- ソフトウェアの仕様記述と検証
- アプローチ1：定理証明
- アプローチ2：型システム
- アプローチ3：モデル検査
- まとめ

ソフトウェアの検証

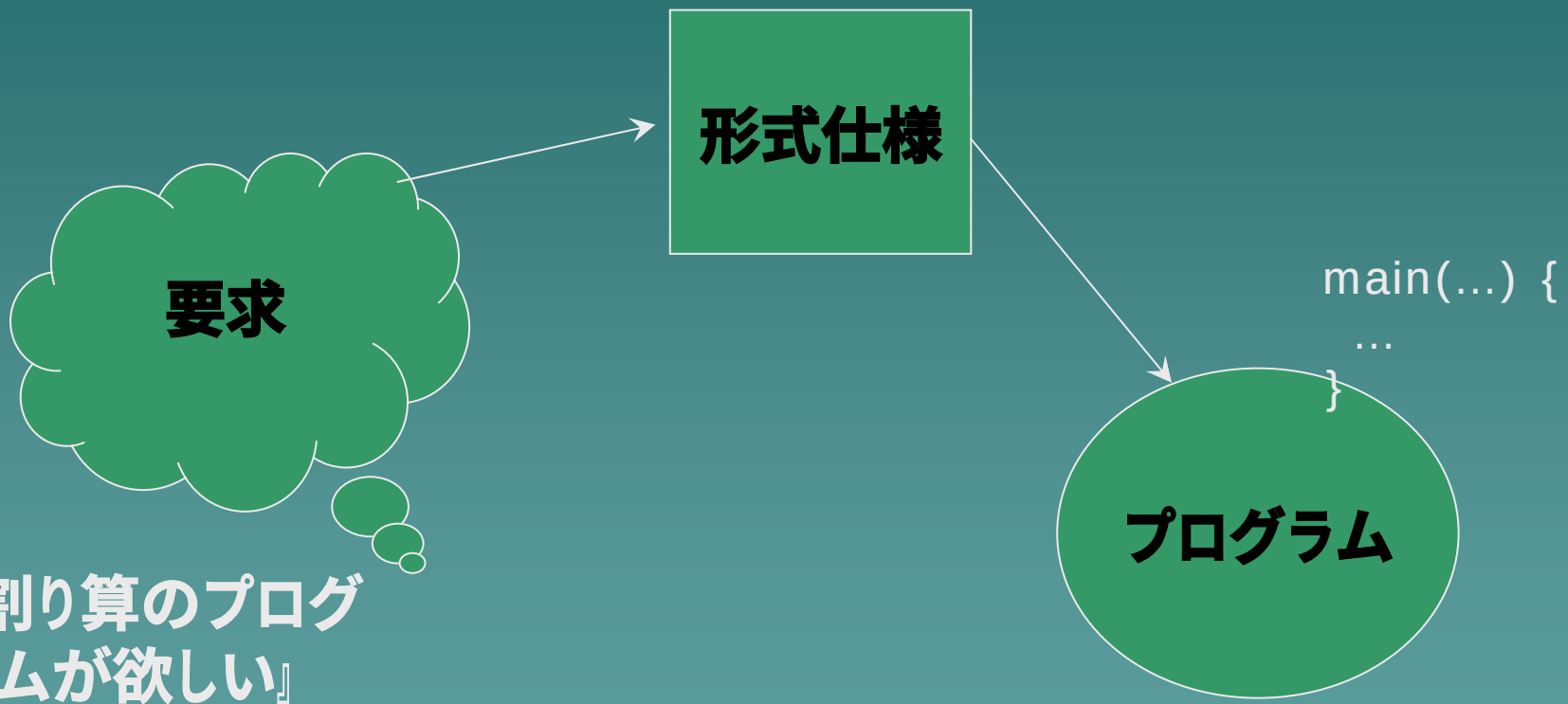
- 「絶対的に正しい」ソフトウェアは存在しない

- 「要求」と「プログラム」
  - 要求： 割り算を計算するプログラムがほしい
  - プログラム：

```
main(...) { .... }
```



プログラムに対する要求と仕様



形式仕様 (Specification)

- 割り算をするプログラムの仕様は？
  - 入力は、被除数x と 除数y (x を y で割る)
  - 出力は 商d と 余りr
  - x, y, d, r の関係を論理式で書くと？

    x = y * d + r

    x = y * d + r ∧ 0≦r<y (Sと記述)

形式仕様＝

形式言語 (**論理式**) で記述された仕様

Hoare論理による形式仕様

- プログラム p の実行前と実行後に成立すべき論理式を記述する

  S {p} T

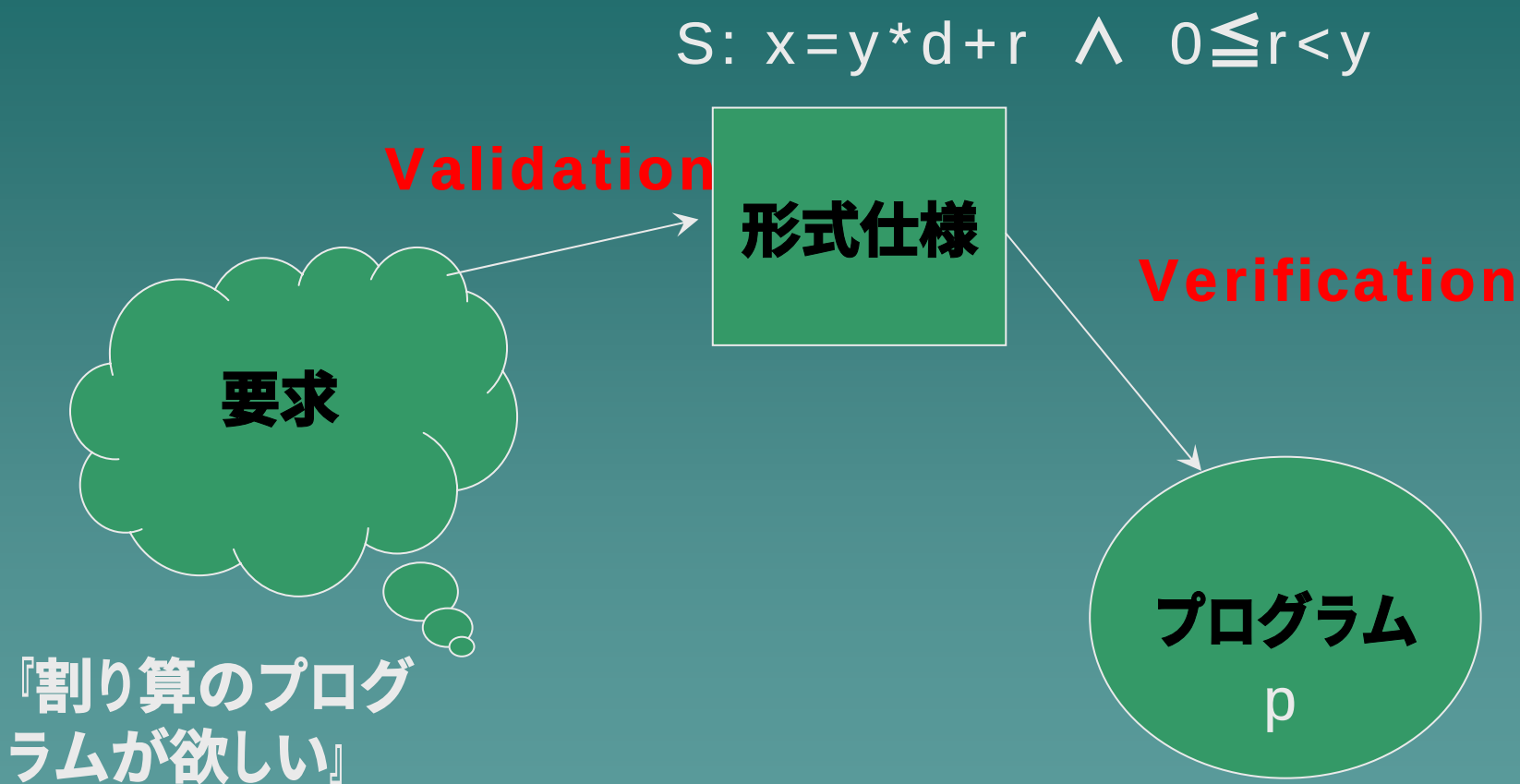
  - 実行前に S が成立するならば、p の実行後には必ず T が成立する
- 割り算をするプログラム p の仕様

  $\{0 \leqq x \wedge 0 < y\}\ p\ \{x = y*d + r \wedge 0 \leqq r < y\}$

形式化ということ

- 形式的定義＝厳密な定義
  - 記号列の上の機械的な操作を仮定して、種々の概念の定義を与える。
- 形式的体系の必要性
  - きちんとした定義がなければ、プログラムの正しさを議論できない。
  - 誰が見ても正しい推論(あるいは計算規則) を与えなければならない
- 形式的体系のメリット
  - ソフトウェアで取り扱うことが出来る！

Validation & Verification



要求、仕様、プログラム

- 仕様Sが割り算を表している、プログラムpが割り算を表している
  (形式的に表現できない)
- プログラム p が仕様Sに関して正しい

$$\forall x.\ \forall y.\ (x = y * p1(x,y) + p2(x,y) \land 0 \leqq p2(x,y) < y)$$

- 仕様Sがおかしくない (妥当である、仕様を満たすプログラムが存在する)

$$\forall x.\ \forall y.\ \exists d.\ \exists r.\ (x = y*d + r \land 0 \leqq r < y)$$

- プログラム p がおかしくない
  0で割り算をしない
  配列がオーバーフローしない
  デッドロックしない
  etc.

ソフトウェアの信頼性の向上

- ソフトウェアの検証
  - ソフトウェアが (完全に) 正しいことを証明する
  - safety-critical system, mission-critical system, …
- ソフトウェアの信頼性向上
  - ソフトウェアの信頼性を相対的に向上させる
  - テスト等に比べて少ないコストでバグをとることができる
  - 一般のソフトウェア

定理証明による
ソフトウェア検証

定理証明による検証

- 最も古くある伝統的な方法
- プログラムが仕様を満たすこと（論理式）を、記号論理の推論による証明する　[数学的な定理証明と同様]
- 研究の蓄積
  - 様々な定理証明技法の提案
  - 計算/プログラムの本質の追及
  - 新しいプログラミング言語の発見
  - etc.

定理証明による検証

- 事実
  - 全自動による定理証明は**原理的に**不可能
  - プログラムの作成に比べて、同じプログラムの検証は、より大きな手間（コスト）がかかる
  - ある種の検証はプログラム作成を含んでいる（構成的プログラミングの原理）
- 検証手段としては「最後の手段」
  - 他の方法が適用できないときは最終的には定理証明を行うしかない

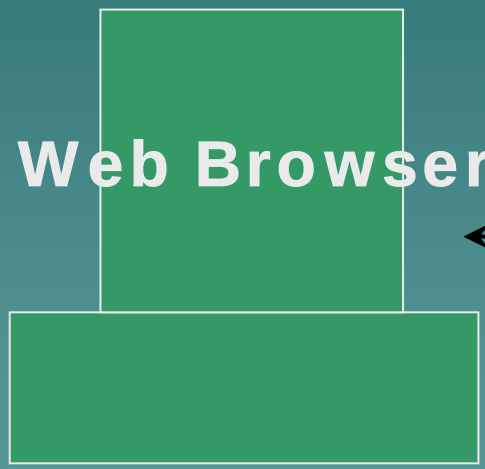
構成的プログラミング

- 形式仕様を論理式で記述する
- 形式仕様の妥当性をある種の論理で証明する
  - 形式仕様を満たす出力値が存在する
- この証明には、プログラムが含まれている！
  - 「ごみ」を落とすことによるプログラムが自動的に得られる。
  - さらに、このプログラムが形式仕様を満たす証明も得られる。

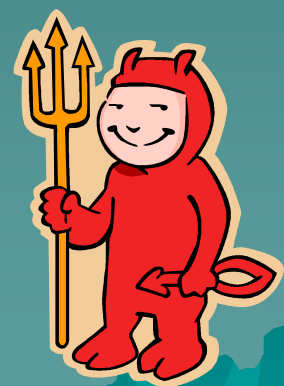
- プログラム作成と、プログラム検証の2つの行為を1つの行為にまとめている
- 理論的には大変興味深い
- 実用化には困難がある
  - プログラムを書くだけでも大変なのに、証明をすることは一般のプログラマにはほとんど不可能
  - 形式仕様自体が１００％完璧なことは少ない。プログラム作成段階で仕様の誤りが見つかる場合もある。
  - 複雑なプログラミングへの拡張が容易でない。

型システムを用いた
プログラムの (部分的な) 検証

ソフトウェアのダウンロード



ダウンロード



ダウンロードしたソフトウェアが正しいことを期待するが、
仮に正しいことまでは言えなくても、自分のマシン上の
データを破壊する等のことはないことを保証したい

型システム

- 型 (type)
  - プログラミング言語におけるデータ型
  - 基本型：整数、実数、真偽値、文字、etc.
  - 複合型；配列、ポインタ、レコード、関数型 etc.
- 型の意義
  - データの種類が理解しやすくなる
  - 変数を取り違えるミスが減る
  - 同じ型のデータは一様な操作を行うことができる (効率的な実行コードの作成につながる)
- 型の整合性
  - プログラム中のすべての操作 (演算) において、必要とするデータの型と実際に渡されるデータの型が整合していなければならない
  - 駄目な例： (int) x = x + 2.3;
  - 駄目な例：prinf(stderr, x * 2);

単純型理論

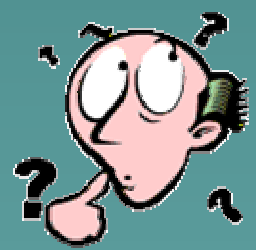
- 型
  - 基本型: K1, …, Kn
  - 複合型: A→B
  - 例: (K1→K2)→ (K1→K2)
- プログラム (ラムダ式)
  - 型A をもつ変数 x^A
  - 変数x^A が引数で、Mを返すプログラム λx^A.M
    - f(x) = x+1 という関数 f は λx^int. x+1
  - プログラムM を関数とみなして、それにプログラム N を引数として与えた結果 MN
    - f(3) のような記法。括弧を省略することが多い。

単純型理論

- 型推論規則
  - x^A　は型Aをもつ
  - Mが型Bをもつとき、λx^A.M　は型A→Bをもつ
  - Mが型A→Bをもち、Nが型Aをもつとき、MNが型 B　をもつ。
  - 例：λx^K1→K2．λy^K1．x y
    は型 (K1→K2)→(K1→K2)をもつ。
  - 例：λx^K1→K2．λy^K1．y x　は型をもたない。
  - 型検査：　プログラムと型が与えられ、確かにその型を持つことを確認すること
  - 型推論：　型のわからないプログラムに対して、その型を推論すること

単純型理論

- 型推論は必ずしも容易ではない
  - λx. λy. x　は型がつく
  - λx.λy.λz. ((x z) (y z))　　は型がつく
  - λx. x x　　は型がつかない



単純型理論における定理

- 型の健全性定理　(の簡単化した形)
  - Mが型Aをもつプログラムで、M　を計算して N が得られたならば、N は型Aをもつ
- 型を持つプログラムの停止性
  - 型を持つプログラムの計算は無限ループすることなく、いつか必ず停止して答えを出す

型理論

- 理論的には、
  - プログラム理論の中でも最も成功した部類
  - 様々な（強力な）型システムの研究、豊富な研究項目
  - 論理との密接な関係
- 実用的には？
  - YES!
  - プログラミング言語の設計への影響
    - 例： Java言語の型システム
  - 型検査、型推論によるソフトウェアの（部分的な）検証

JAVAバイトコードの検証

- 検証の動機
  - JAVAプログラムではなく、それをコンパイルした機械語のコードのみをもらって、自分のマシンで実行することがある
  - 極めて危険
  - もとのJAVAプログラムがない以上、正しい動作をすることの検証は絶望的
  - しかし、最低限、このコードが「悪さをしない」ことは確認してから実行したい（完全な検証ではないが、一部の性質の検証）

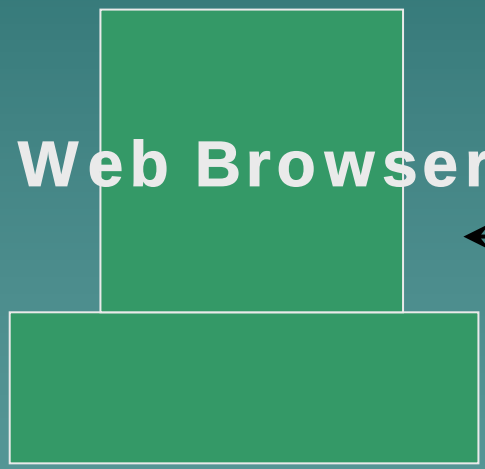
JAVAバイトコードの検証

- バイトコード（機械語のプログラム）だけをもらって、それが「悪さをしない」ことを自動的に検証したい
  - 悪さの例：
    - 実行時に型エラーを起こす
    - スタックをいくらでもたくさん消費する
    - とんでもない番地へジャンプする
    - 確保したメモリ領域外へアクセスする
- これらの性質の検証を型推論、型検査の手法で行う
  - バイトコードに対する型システムを構築する。
    - 「型」の概念を洗練して、上記の性質を含んでいるものだけが片付けできるようにする
  - 型の健全性定理を証明する。
  - 型推論アルゴリズムを作成する。
    - バイトコードがあたえられると、それが正しい型をもつかどうか自動的に推論する

型システムを用いた検証

- うまくいくかどうかは、型システム次第
- 長所
  - うまくいく場合は、いくつかの「最低限欲しい性質」が自動的に検証できる
  - 型理論の豊富な成果から、どのような性質に拡張できるかの目安がつけやすい
  - コンパイラ等に組み込むことも容易
- 欠点
  - プログラム言語ごと、また、検証したい性質ごとに型システムを構築して、型の健全性定理を証明しなければならない
  - 検証したい性質は比較的単純なもののみ

ソフトウェアのダウンロード



ダウンロード

ダウンロードする毎に、型検査を行う

Proof-Carrying-Code: ダウンロードするコードに
「証明のヒント」を付加する方法

モデル検査による
ソフトウェア検証

モデル検査

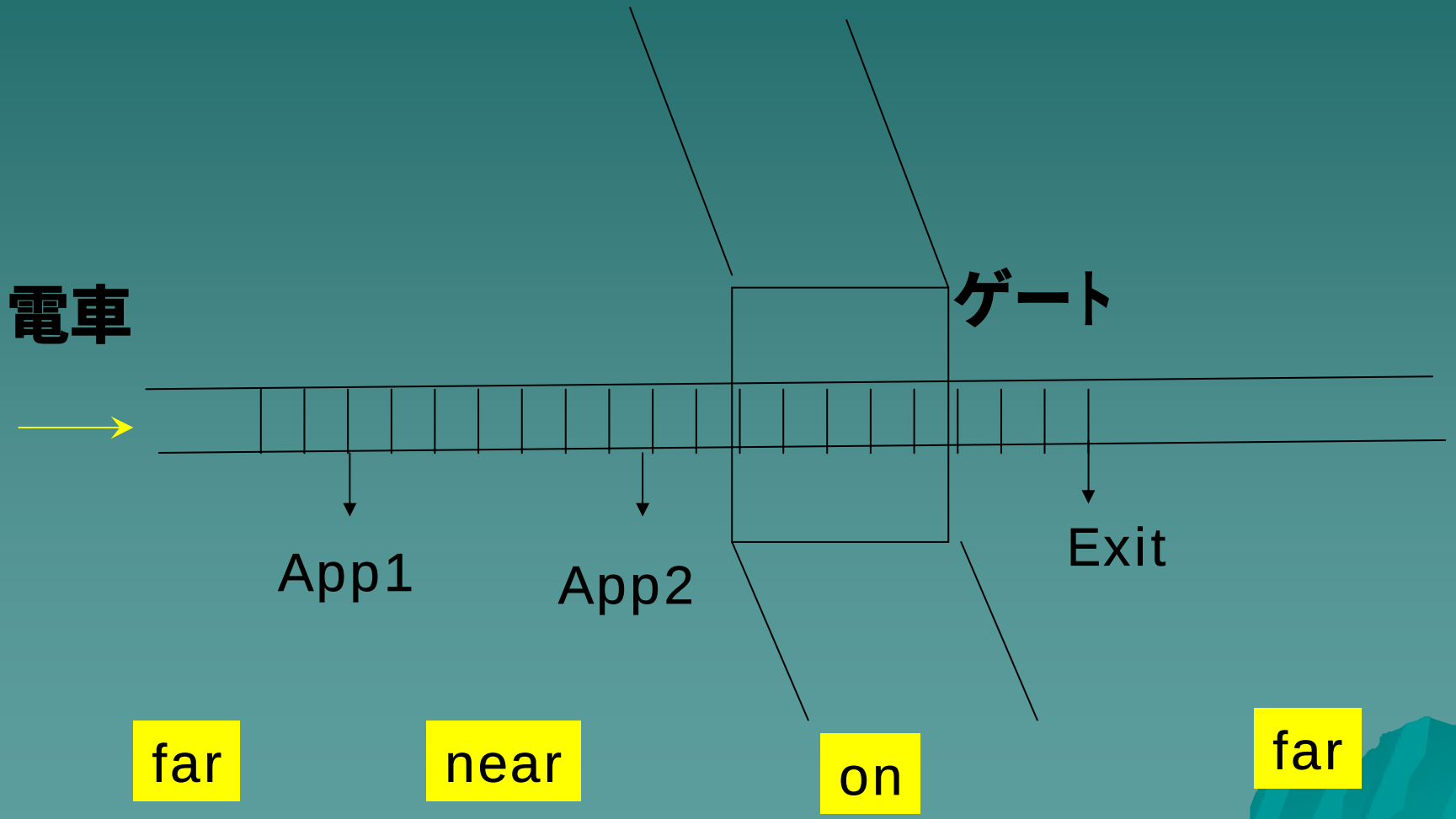
- システムの設計(design)レベルの検証技法
- 対象となるシステムはソフトウェアに限らない
  - ハードウェア、プロトコルなど　リアクティブ・システムとして定式化できるものが対象
  - 状態数が有限のもの
- 全自動
- 検証できなかったときは反例を構成

- 歴史
  - 1980年代 Clark&Emerson, Queille&Sifakisらにより創始
  - 現在、産業界への応用が活発化

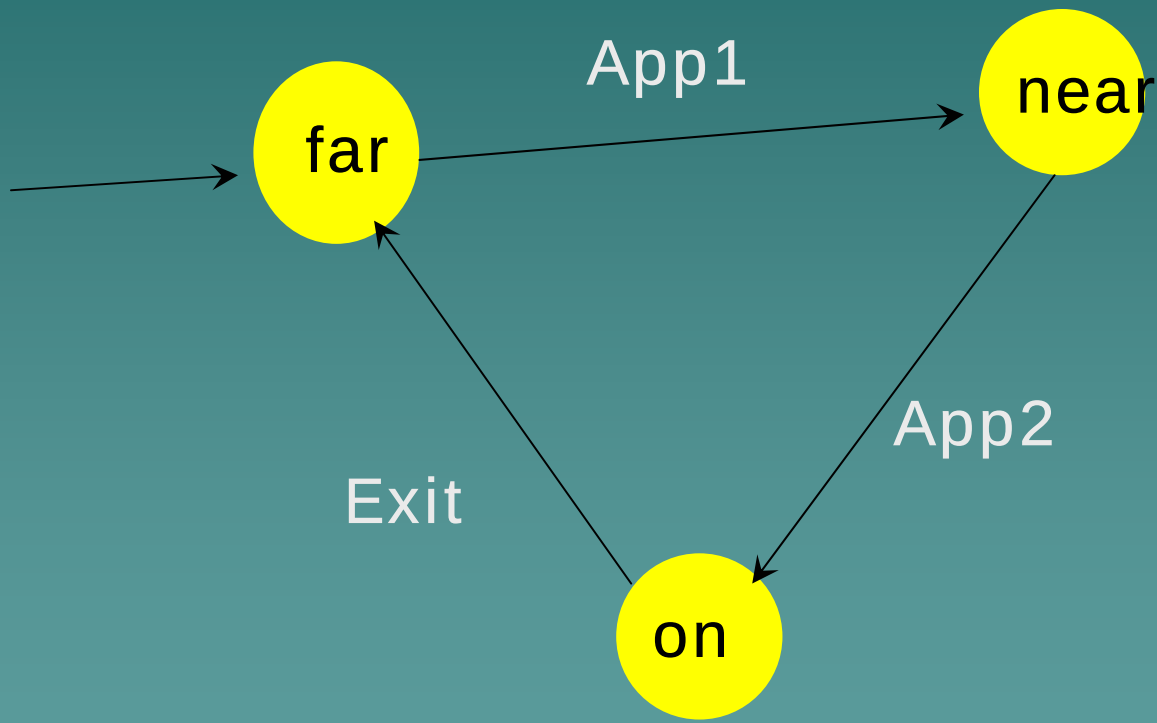
モデル検査の概要

- 遷移系 (transition system, automaton)
- 検証命題
- モデル検査アルゴリズム

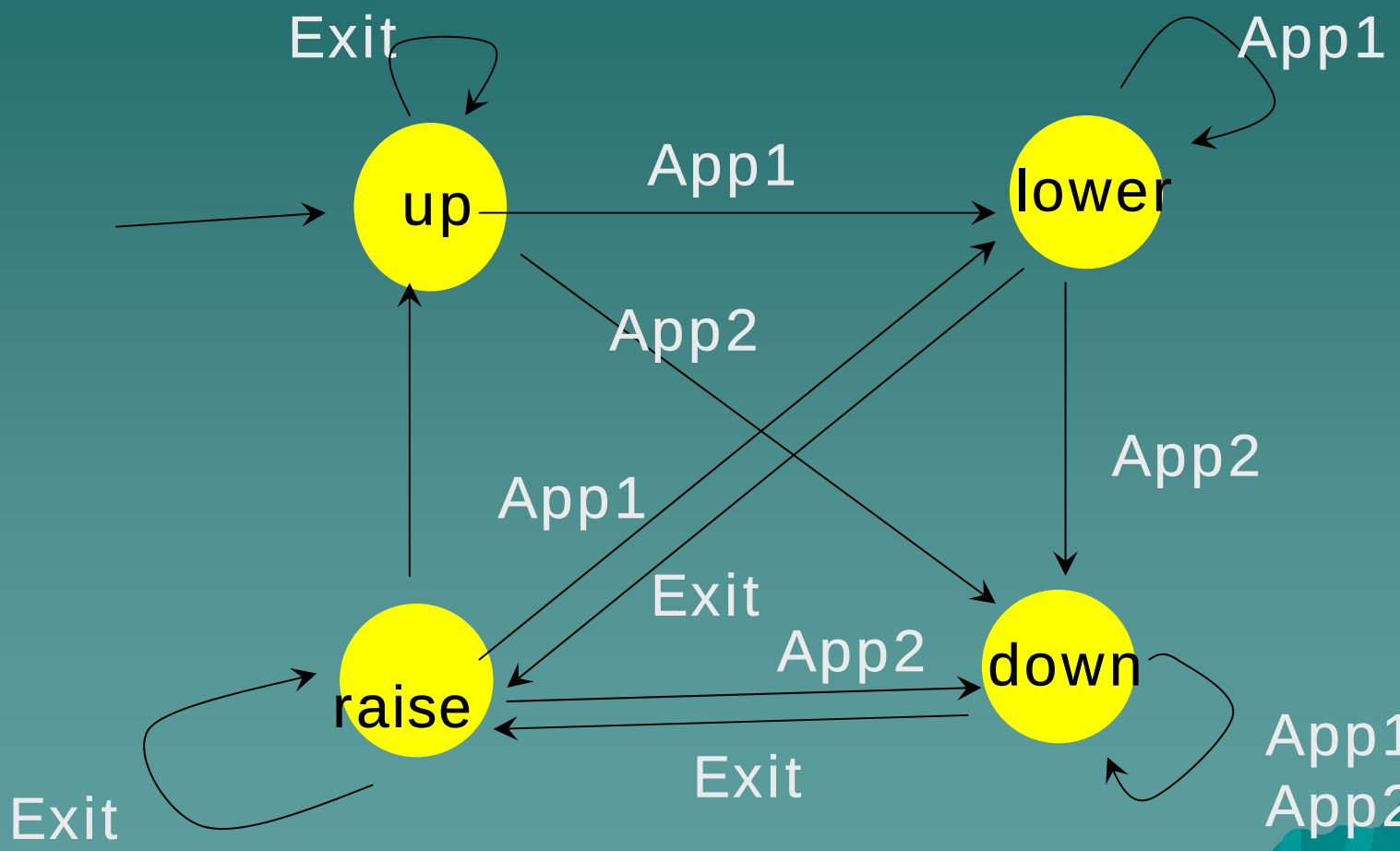
踏切のある鉄道 [Berardら]



電車の遷移系



ゲートの遷移系



電車は2台続けてくることがあるが、このモデルではうまくいかない

検証命題

- この遷移系が満たすべき性質を時相論理(Temporal Logic)の論理式で表現
- Safety: 今からずっと、○○であり続ける

  AG (on ⇒ down)

  いつでも、電車がゲート上にあれば、ゲートはおりている

  AG AX true

  システムがデッドロックしない
- Liveness：将来のある時点で、○○が成立する

  AG(down ⇒ AF up)

  ゲートがおりていれば、いつかはゲートがあがる

モデル検査アルゴリズム

- 検証命題を記述する論理体系の種類ごとに異なるアルゴリズムがある
- CTL論理： 効率的なアルゴリズムが存
- LTL論理： 特定の実行系列に関する非常に複雑な命題を記述できるが、論理式のサイズに関して指数関数的なアルゴリズム

モデル検査の実例

- ソフトウェアよりむしろハードウェア、プロトコル検証で数多い例題
- 例：SMVシステムによる検証
  - キャッシュ一貫性プロトコル
  - IEEE Future bus+
  - T9000マイクロプロセッサの部品
  - 10^1300 状態をもつ論理回路
  - 通信システム

モデル検査の現状と将来

- 現状
  - 多数の実例
  - ハードウェア (CPUなど) 、プロトコル等では設計段階の検査が常識になりつつある
- 問題点
  - 非常に状態数の多いシステム、無限状態を持つシステムへの適用
  - 実時間システムへの適用

モデル検査の研究動向

- 論理体系の拡張
  - より広範囲の検証命題を記述でき、かつ、モデル検査に要する時間が少ない体系
- 実時間を取り入れた検証
  - 「リクエストを出してから、いつか反応がかえってくる」ではなく「5秒以内に反応する」といった時間制約を記述したい
- 巨大な状態数への挑戦
  - Symbolic Model Checking　記号的に状態を表現する
  - 抽象化　　いくつかの状態を1つの状態にまとめる
- 無限個の状態をもつシステムへの挑戦
  - 実は、大半のソフトウェアは無限状態をもつ

まとめ
ソフトウェア検証へのアプローチ

- 定理証明を用いた方法
  - 最強の方法
  - 全自動化できないためコストがかかる
  - 他の方法がとれないときに用いる
- 型システム等の静的解析を用いた方法
  - 検証命題を自由に定義できない (型システムごとに決まる一定のパターンのみ)、弱い意味での検証
  - (多くの場合) 自動化可能
- モデル検査を用いた方法
  - 有限状態システムに適用可能
  - 検証命題はかなり自由に記述できる
  - 時間に関するある程度の表現が可能 (時相論理)
  - ソフトウェア以外のシステムにも適用可能

まとめ

- ソフトウェアにバグ（虫）はつきもの
  - １００万行の小説で1文字も誤字がないものがあるだろうか？
  - ましてや意味的な誤りも一切ないことは期待できない
- しかし、ソフトウェアのバグを減らす技法は進歩している
  - 精力的に研究、既に実用化されているものもある
  - 現在の情報環境ではどこでも必要とされる性質を保証する技法は遠からず来るだろう
  - 一方で、より複雑な環境で動作するプログラムが開発されていく、という、いたちごっこでもある。

問い合わせ等

- 本日の講義資料

  http://logic.is.tsukuba.ac.jp/~kam

  から取得可能にする予定
- 質問等のあて先

  kam@is.tsukuba.ac.jp

参考

- システム検証一般
  - 林晋「プログラム検証論」共立出版
  - システム設計検証研究会
- 定理証明
  - Handbook of Automated Reasoning
- 型システム
  - 龍田真「型理論」
- モデル検査
  - Berard et al. “Systems and Software Verification”, Springer, 1999
  - Huth et al., “Logic in Computer Science”, Cambridge, 2000