Formalizing Advanced Class Mechanisms

Atsushi Igarashi (Univ. of Tokyo)
joint work with
Benjamin C. Pierce (Univ. of Pennsylvania)
Philip Wadler (Avaya Labs)

Class-Based OO Languages

Main Stream of Object-Oriented Languages (C++, Java, etc.)

- Useful and simple abstraction mechanisms
  - Class as a programming unit
  - Inheritance for incremental programming

Advanced Class Mechanisms

For better abstraction and code reusability

- Parametric classes
  - Type information can be abstracted as parameters (cf. polymorphic functions)
    - templates in C++
    - extensions for Java [GJ, NextGen, etc.]
- Nested classes
  - Classes can be members of a class just as fields/methods (cf. nested functions)
    - virtual classes in Beta [Madsen et al. 89]
    - inner classes in Java 1.1

Goals of the Work

- Clarification of the essence of complex advanced class mechanisms
  - Language semantics
  - Type system
  - Implementation technique
- Feedback to language design

Our Approach

Formal modeling with a tiny class-based language

- Focus on core features
  - Dropping other advanced features such as reflection, concurrency, etc.
  - Making proofs of useful properties tractable
- High-level base language

Why Formalization?

- Rigorous specification of the language
  - Useful for serious users and compiler writers
  - cf. official specification documents written in English prose
- Proof of important properties
  - e.g., type soundness, correctness of compilation
    - Increasing importance of secure languages

Why High-Level Base Language?

- No additional complexity of encoding
  - cf. Object calculi [Abadi&Cardelli] and object encoding in λ-calculus [Pierce&Turner, etc.]
- Focus on complexity of advanced mechanisms

Objects of Study

- Inner classes in Java
  - Member classes
  - Local/anonymous classes
  - Compilation to Java Virtual Machine Language
- GJ [Bracha, Odersky, Stoutamire & Wadler 98]
  - Parametric classes/methods
  - Raw types to interface with old Java code
  - Method type parameter inference
  - Compilation to Java Virtual Machine Language

Contributions

Methodology to study advanced features

- Design/formalization of a foundational calculus: Featherweight Java
- Formalization of the core of ...
  - Inner classes (w/o local/anonymous classes)
  - GJ (w/o type inference)
    - Direct semantics
    - Type system
    - Compilation scheme

Contributions (contd.)

- Proofs of ...
  - soundness of the type system
  - correctness of the compilation scheme
- Discovery of design/compiler bugs

Outline of the Talk

- Featherweight Java (FJ)
- Featherweight Java with Inner Classes (FJI)
- Featherweight GJ (FGJ)
- Related Work
- Concluding Remarks

Featherweight Java

Slogan: "λ-calculus" for class-based OO languages

- A tiny sublanguage of Java with ...
  - top-level class declarations
  - inheritance/subtyping
  - fields/(side-effect-free) methods
  - typecast (dynamic typecheck)
  - method recursion through `this`
- Simple small-step operational semantics
  - `e` → `e'` (w.r.t. a certain set of class defs)
    - realized by syntactic tricks

Why Omitting Assignments?

- Simplifies the formalization significantly
  - No need to model heap
- Does not affect most issues in FGJ/FJI
  - Exceptions (left for future work):
    - Type inference in GJ
    - Compilation of local/anonymous classes

Example of FJ Class

Pair class

```
class Pair extends Object {
  Object fst; Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd;}
  Pair setfst (Object nfst) {
    return new Pair(nfst, this.snd);}}
```

- The single constructor just initializes fields from its arguments
- Every method body is a single `return` statement
- Explicit `this` to access fields/methods of the current class

Syntax of FJ Classes

```
L ::= class C extends D {
        C_1 f_1; …; C_n f_n; K M_1 … M_m }
K ::= C(C_1 f_1,…,C_n f_n){ super(f_1,…,f_i);
      this.f_{i+1}=f_{i+1}; …; this.f_n=f_n; }
M ::= C m(C_1 x_1,…,C_n x_n){ return e; }
e ::= x
    | this
    | new C(e_1,…,e_n)
    | e.f
    | e.m(e_1,…,e_n)
    | (C)e
```

Operational Semantics

- Reduction relation: $e \rightarrow e'$

e.g., **new Pair(a,b).setfst(c)**

→ **[c/nfst, new Pair(a,b)/this]**
new Pair(nfst, this.snd)

= **new Pair(c, new Pair(a,b).snd)**

→ **new Pair(c,b)**

recursion through **this**

$$\frac{\mathit{methodbody}(\mathbf{m}, \mathbf{C}) = (\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{e})}{\mathbf{new\ C}(\mathbf{e}_1,\ldots,\mathbf{e}_n).\mathbf{m}(\mathbf{d}_1,\ldots,\mathbf{d}_m) \rightarrow [\mathbf{d}_i/\mathbf{x}_i, \mathbf{new\ C}(\ldots)/\mathbf{this}]\mathbf{e}}$$

Type System

- Subtyping `C <: D`
  - induced by `extends` clauses in class declarations
- Type judgment $\Gamma \vdash \texttt{e} : \texttt{C}$
- Typing rules following Java's closely
  - object instantiation

$$\frac{\mathit{fields}(\texttt{C}) = \texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n \qquad \Gamma \vdash \texttt{e}_i : \texttt{D}_i \qquad \texttt{D}_i <: \texttt{C}_i}{\Gamma \vdash \texttt{new C}(\texttt{e}_1, \ldots, \texttt{e}_n) : \texttt{C}}$$

Type Soundness

- Theorem: A well-typed FJ program does not cause "no such method/field" errors
  - Proved using standard techniques
    - Subject reduction
      - Reduction preserves well-typedness
    - Progress
      - Reduction proceeds unless the (well-typed) expression becomes a (fully-evaluated) value or fails at a downcast

Outline of the Talk

- Featherweight Java (FJ)
- Featherweight Java with Inner Classes (FJI)
- Featherweight GJ (FGJ)
- Related Work
- Conclusion and Future Work

Inner Classes in Java

- Class declaration in a class
  - as a member just as fields/methods
  - inside a method (a.k.a. local/anonymous classes)
- Direct access to members of its enclosing definitions
  - Programming style similar to nested functions
- Extensively used in event-driven programming (e.g., AWT)

Inner Classes in AWT programming

- Counter and event handler (ActionListener)

```
class Counter {
  int x=0;
  void regButton(Button b) {
    b.addActionListener(new Listener());}
  class Listener implements ActionListener {
    public void actionPerformed(ActionEvent e)
     { x++; }}}
```

  - Method `regButton` associates a listener with a given button
  - Every time the button is pressed, `x` will be incremented through the invocation of `actionPerformed` of the listener object

Complication (1): Scoping Rules

- Directly accessible members are in
  - superclasses
  - enclosing classes
  - even superclasses of enclosing classes

```
class A { Object f; … }
class B { Object f;
  class C extends A {
    Object m() {
      return f;
      // which f?
}}}
```

Complication (2): Inheritance

- Almost any form of inheritance is permitted
  - Inner class is inherited outside of the defined scope

```
class A { …
  class B { … }
}
class C extends A.B { … }
```

  - Inner class can extend its enclosing class

```
class A { …
  class B extends A { … }
}
```

Official Specification by JavaSoft

- Explains what inner classes are and how to compile them to Java Virtual Machine Language
- However, it is not quite sufficient
  - indirect
    - the semantics is given only via compilation
  - informal and sometimes ambiguous
    - written in English prose
    - interpreted differently by different versions of JDK

Main Issues to Deal With

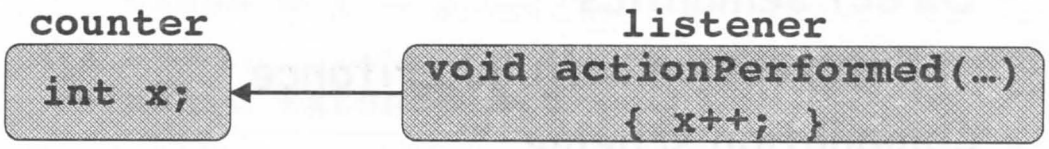
- Syntactic complication
  - Scoping rules
- Semantics complication
  - Direct semantics
    - Interaction with inheritance
  - Compilation scheme

Outline

- Basic constructs about inner classes
- Overview of formalization
- Interaction with inheritance

Enclosing Instances

- Every instance of an inner class is associated with an *enclosing instance* of the enclosing class
  - e.g., `Listener` object has its `Counter` object to increment



  - cf., function closure = code + environment
- `new Listener()` in `Counter` creates a listener whose enclosing instance is `this`

`.this` Notation

- Enclosing instance is accessible with `C.this`
  - `C` is the enclosing class's name

- Direct access to `x` is just an abbreviation of access to the field `x` of the enclosing instance

```
class Counter {
  int x; …
  class Listener … {
    public void actionPerformed(…){
      Counter.this.x++; } … }}
```

Prefix to `new`

- Enclosing instance of a newly created object can be explicitly specified by prefixing `new`
  - `e.new Listener()` creates a new listener where `Counter.this` refers to `e`, which is a `Counter` object
  - `new Listener()` in the class `Counter` is an abbreviation of `this.new Listener()`
- A `Listener` can even be instantiated outside the class `Counter`

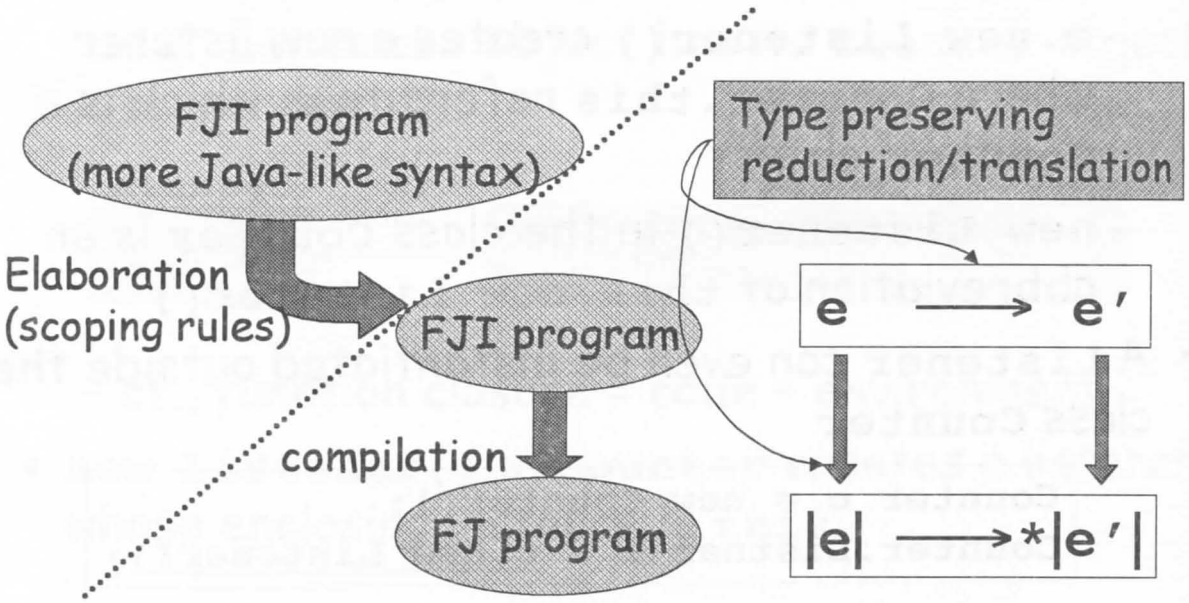
```
Counter c = new Counter();
Counter.Listner l = c.new Listener();
```

Formalization: FJ+Inner Classes (FJI)

- notation `C.this` for enclosing instances
- prefix to `new`
- notations to support inheritance (explained later)

Separating Syntactic Complication

- Elaboration from external to internal language



Example of FJI Program

```
class A {
  class B {
    Object f; Object g;
    B(Object f, Object g){this.f=f; this.g=g;}
    B set_g(Object x) {
      return new B(f, x); }}}
```

Example of FJI Program

```
class A {
  class B {
    Object f; Object g;
    B(Object f, Object g){this.f=f; this.g=g;}
    A.B set_g(Object x) {
      return A.this.new B(this.f, x); }}}
```

- Everything is made explicit by elaboration
  - "fully-qualified" type names
  - `this`/`A.this` before field accesses and `new`

Reduction Relation: e → e′

- When a method is invoked, the redex becomes the method body (the expression after `return`), replacing formal parameters, `this` and `A.this`

```
(new A().new B(a,b)).set_g(c)
→ [c/x, new A()/A.this,
   new A().new B(a,b)/this]
      A.this.new B(this.f, x)
= new A().new B((new A().new B(a,b)).f,c)
→ new A().new B(a,c)
```

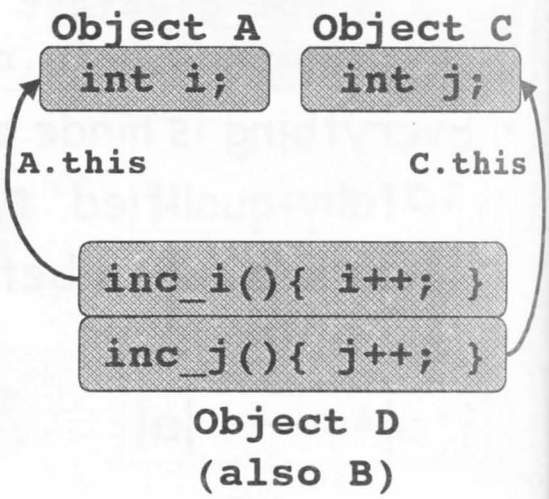
  - The value for `A.this` is immediate from the form of the receiver

Inheritance and Inner Classes

- Inherited methods may require a *different environment* i.e., enclosing instance
  - *How do you create an object of* **`C.D`**?

```
class A {
  int i;
  class B {
    void inc_i(){
      A.this.i++; }}}

class C {
  int j;
  class D extends A.B {
    void inc_j() {
      C.this.j++; }}}
```



Prefix to `super();`

- Constructor of `D` must specify the value for `A.this` used in `B` by prefixing `super();`

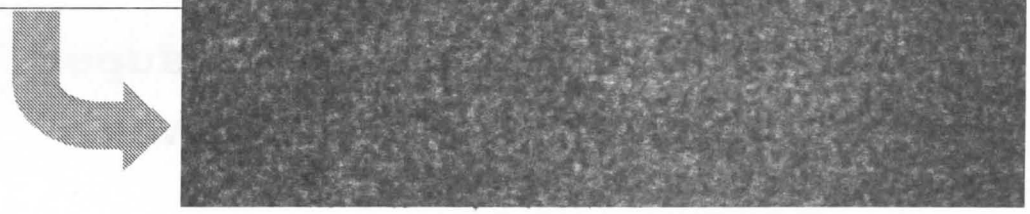
```
class C {
 int j;
 class D extends A.B {
   D(A a){ a.super(); } … }}
```

  - Both `a.new B().inc_i()` and `c.new D(a).inc_i()` increments `i` of `a`
- In FJI, a prefix must be a constructor argument
  - The prefix can be any expression in Java
  - Even under such a restriction, computing `A.this` gets rather complicated

Compilation of Inner Classes

- An inner class is translated to a top-level class with one additional field holding the *direct* enclosing instance
  - `C.this` is translated to a sequence of field accesses `this.`f_1`.….`f_n where $f_1,\ldots,f_n$ are mangled field names
- Formalized as translation from IL to pure FJ

```
class A … {
  class B … { …
    class C … { …
      T m() { return … A.this …; }
}}}
```



Technical Results

- Definition of FJI
  - reduction semantics and type system
    - main complexity in the computation of enclosing instances in method invocations
  - elaboration (scoping rules)
- Formal rules of compilation to pure FJ
  - Closure conversion of class definitions
- Theorem: type soundness
- Theorem: correctness of compilation

What was "Real" Complication?

✓ An instance of an inner class is just like a closure!

✓ Convenient for

- exporting a small piece of code outside
- writing adapters

– Environment is represented by an object called enclosing instance

✖ Liberal subclassing lets methods from one object refer to distinct environments

– By certain restrictions,

- we wouldn't need a prefix to **super**
- simpler semantics could be given

Nested Classes in Simula

- Possible forms of inheritance are rather restricted

– If $C_1 \ldots . C_n <: D_1 \ldots . D_m$, then n=m and $C_i <: D_i$

```
class A {
  class B { … } }

class C extends A {
  class D extends B { … } }
//A.this in B and C.this in D
//always denote the same object
```

- An object can dispense with multiple links to direct enclosing instances
- no need for prefixed **super();**

Nested Classes in Beta

- Enclosing instances are a part of the type

```
class A {
  class B { … }}

class C {
  class D extends a.B { … }
  // a is the enclosing instance A.this
  //for methods in B
}
```

  - some sort of *dependent types*?
  - no need for prefixed `super();`, either

Subtlety in Scoping Rules

✗ Some classes/packages can be inaccessible!

  - Necessity of syntax for "top-level"?
    - c.f. "::" in C++

```
class A { … }
class B { …
  class A {
    // you can't use the top-level class A,
    // or package A here!!
  }
}
```

Outline of the Talk

- Featherweight Java (FJ)
- Featherweight Java with Inner Classes (FJI)
- Featherweight GJ (FGJ)
- Related Work
- Conclusion and Future Work

GJ [Bracha, Odersky, Stoutamire & Wadler]

- Java + ML-style parametric polymorphism
  - Classes and methods can take type parameters
    - Refined type information
- Compilation into JVML
  - By "erasing" type parameters
    - cf., "by-expansion" compilation of C++ templates
- Method type parameter inference
- Raw types for interface with Java

Example of GJ Program

```
class List<X> {
  X hd;   List<X> tl;
  <Y> List<Y> map(Fun<X,Y> f) { … }}

List<A> l = …;
Fun<A,String> f = …;
String s = l.map(f).hd+"Hello";
```

- Type argument `String` for map is inferred
- Type system can give `hd` the type `String`
  - Java programmers simulate this style by giving `hd` the type `Object` and inserting a downcast `(String)` for the access of `hd`

Main Issues on GJ

- Direct semantics
  - not specified in the official document
- Type system and its soundness
  - Subtlety due to the erasure compilation
  - Raw types
    - `List<X>` can be used w/o type parameters
- Correctness of the compilation scheme
  - depending on downcasts
    - run-time check might fail

Featherweight GJ (w/o Raw Types)

FJ + parametric polymorphism

- Tricks similar to FJ
  - Single constructor
  - Method body as a single `return` statement
  - Explicit `this`
  - Explicit method type parameters
    - No type inference (so far)

Example: Polymorphic Pair Class

```
class Pair<X extends Object,
           Y extends Object> extends Object {
  X fst; Y snd;
  Pair(X fst, Y snd) { … }
  <Z extends Object> Pair<Z,Y> setfst(Z nfst) {
    return new Pair<Z,Y>(nfst, this.snd);}
}
```

- Type parameters `X`, `Y`,and `Z` with explicit upper bounds `Object` (i.e., any type is allowed)
- Object instantiation: `new Pair<A,B>(a,b)`
- Method invocation: `e.setfst<B>(b')`

Direct Operational Semantics

Reduction relation as in FJ

- "Type-passing" reduction semantics
  - cf. polymorphic λ-calculus [Girard, Reynolds]

```
e = new Pair<A,B>(a,b)

e.setfst<C>(c)
→ [C/Z, c/nfst, e/this]
     new Pair<Z,B>(nfst, this.snd)
= new Pair<C,B>(c,new Pair<A,B>(a,b).snd)
→ new Pair<C,B>(c,b)
```

- Models programmers' intuition directly

Model of Compilation

Formalized as translation from FGJ to FJ

- Type parameters are erased
  - `Pair<A,B>` ⇒ `Pair`
  - `X` ⇒ `Object`
- Synthetic *downcasts* for recovery of type information of the original program
  - `new Pair<A,B>(a,b).fst`
    ⇒ `(A)(new Pair(a,b).fst)`
    - The type of the field `fst` is `Object` in FJ
  - Synthetic downcasts must not fail !

Compilation of Pair Class

```
class Pair<X extends Object,
           Y extends Object> extends Object {
  X fst; Y snd;
  Pair(X fst, Y snd) { … }
  <Z extends Object> Pair<Z,Y> setfst(Z nfst) {
    return new Pair<Z,Y>(nfst, this.snd);}
}
```

```
class Pair extends Object {
  Object fst; Object snd;
  Pair(Object fst, Object snd) { … }
  Pair setfst(Object nfst) {
    return new Pair(nfst, this.snd);}
}
```

Equivalence of Two Semantics

- Naïve 1-to-n correspondence of reduction steps does *not* hold

FGJ execution: `new C<A,B>(...).m()`

⇓

FJ execution: `(A)new C(...).m()`

- But, it is proved that the execution *results* will agree with each other
  - Synthetic downcasts can never fail, indeed!

Summary of FGJ (w/o Raw Types)

- Intuitive semantics
- Compilation scheme and its correctness
  - Implementation of polymorphism with subtyping and typecast
- Theorems
  - Type Soundness
    - Well-typed FGJ program does not cause "no such method/field" errors
  - Correctness of compilation
    - Compilation preserves well-typedness
    - Compilation preserves execution *results*

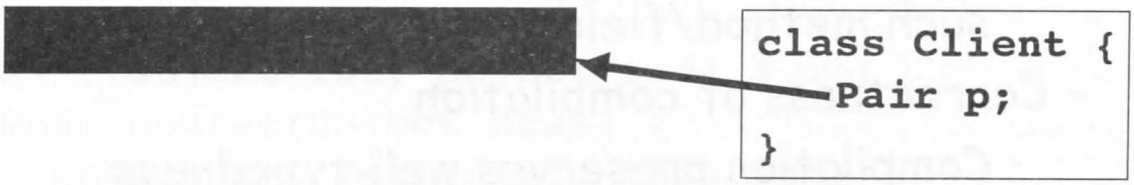
Outline of the Talk

- Featherweight Java (FJ)
- Featherweight Java with Inner Classes (FJI)
- Featherweight GJ (FGJ)
  - Raw types to support program evolution
- Related Work
- Conclusion and Future Work

Raw Types to Support Program Evolution

Interface between parametric classes and old Java classes

- Class `Pair<X,Y>` can be accessed through the raw type `Pair` (w/o type parameters)
  - so that clients of `Pair` do not have to be rewritten as `Pair` is rewritten to `Pair<X,Y>`



```
class Client {
    Pair p;
}
```

How Raw Types Work

E.g., the raw type `Pair`

- provides *erased* types for fields/methods
  - `fst` is given type `Object`
  - `setfst` takes `Object` and returns `Pair`
    - no type parameters required

```
class Test { ……
   A id(Pair p, A a) {
    return (A)(p.setfst(a).fst);  }}
```

- is supertype of `Pair<X,Y>` for any `X` and `Y`

```
new Test().id(new Pair<A,B>(…), a)
```

Unchecked Programs

- Several unsafe operations are permitted to accept more evolving programs
  - unsafe subtyping: `Pair <: Pair<A,B>`
  - method invocation is unsafe if erasure changes argument types
    - `p.setfst(...)` is classified as unsafe
      - argument type `Z` changes to `Object`
- Compiler should issue *unchecked warnings* against such unsafe operations

Some Natural Questions...

- Do unchecked warnings cover all sources of disaster?
- What are they, really?
  - GJ spec. explains raw types in terms of compilation (erasure)
    - "Erased programs are, at least, well-typed Java programs"

Current Results from Formalization

- Type system of FGJ with raw types
  - correspondence to *existential types*
    - Type theory for ADT and modules
    - `Pair` is *some* polymorphic class `Pair<X,Y>`
- Direct reduction semantics
  - Type arguments for polymorphic methods may be missing!

```
new Test().id(new Pair<A,B>(…), a)
→(A)new Pair<A,B>.setfst<?>(a).fst
```

    - Appropriate type arguments have to be 'made up'
      - Introduction of *the bottom type*

Current Results (contd.)

- The original type system is found flawed!!
  - Invocation of raw constructor should be unsafe if erasure changes argument types
- Repaired type system is proved to be sound

More Questions to Answer

- How do raw types affect the correctness of compilation?
- Do raw types really help program evolution?
  - Evolution theorem?
    - "Sensible evolution preserves well-typedness"

Related Work

- Official specification documents
  [Bracha et al. 98, JavaSoft 97]
  - Indirect
    - Semantics as compilation scheme
      - Hard to reason about program behavior
  - Informal
    - Written in English prose
    - Including several underspecifications

Related Work (contd.)

- Object calculi [Abadi&Cardelli] and object encoding in λ-calculi [Pierce&Turner, etc.]
  - Complexity of encoding of simple classes
- Models of class-based languages [Fisher&Mitchell, etc.]
  - Focusing on basic class mechanisms
- Core languages for Java [Drossopoulou&Eisenbach, etc.]
  - Too large to deal with complex extensions

Conclusions

Formalization works !

- Rigorous understanding of complex language mechanisms
  - Semantics and typing rules
  - Compilation scheme
- Proofs of useful properties
  - Type soundness
  - Correctness of compilation scheme
- Discovery of design bugs and underspecifications

Future Work

- GJ's type inference
  - Bottom type plays an important role
  - Interaction with side-effects
- Local/anonymous classes
  - Method parameters as free variables in a class declaration
  - Interaction with side-effects
- Access restriction (`public`, `private`, ...) and inner classes

Future Work (contd.)

- Connection with lower-level calculus
  - structural vs. name-based subtyping
  - denotational semantics of FJ [Studer00]
- Study of erasure compilation and raw types under a simpler setting

フォーマルメソッドのコンポーネントソフトウェア開発への適用について

2000年10月31日

PFU/JAIST 松本 充広

概要

フォーマルメソッド：射影型振舞仕様

仕様言語：CafeOBJ

コンポーネントソフトウェア開発：

ｺﾝﾎﾟｰﾈﾝﾄを組合せることでｿﾌﾄｳｪｱを開発

フォーマルメソッドの適用部分：

ｺﾝﾎﾟｰﾈﾝﾄの仕様，組合せ方の記述 を用意

↓

組合せの正当性保証，組合せ部の生成

概要

(DeRemore et al. 1976)

Programming-in-the-Small:

- モジュール生成のためのプログラミング

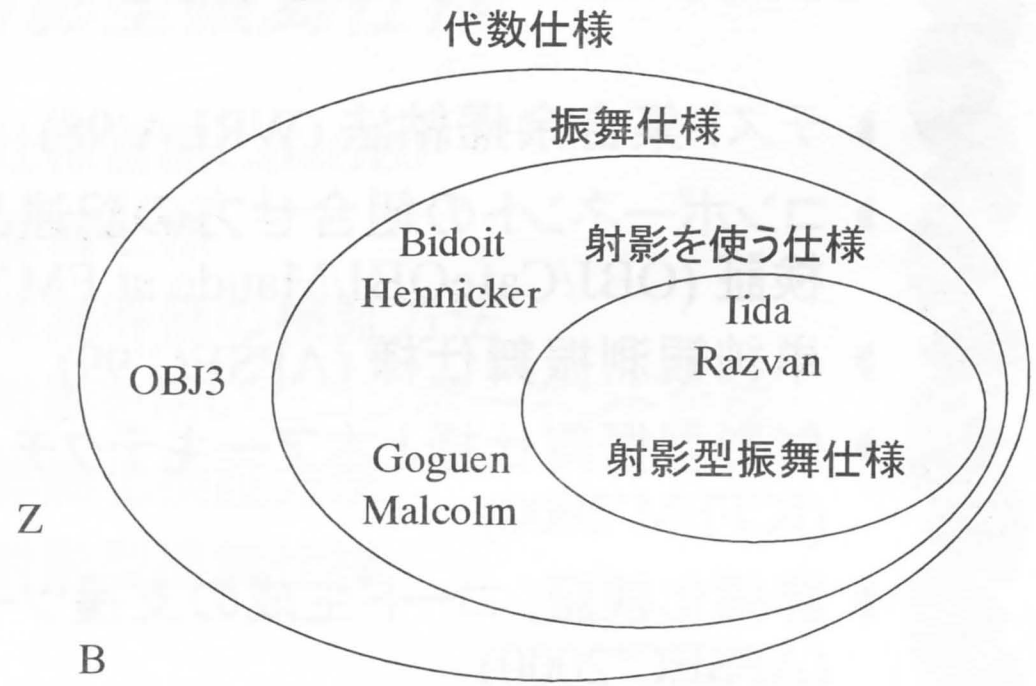
Programming-in-the-Large:

- 組合せ部生成のためのプログラミング

B, VDM-SL → Programming-in-the-Small

CafeOBJ → Programming-in-the-Large

射影型振舞仕様の位置付け



射影型振舞仕様までの流れ

振舞等価の概念？ (Moore 1956)

↓

振舞理論 (Bidoit and Hennicker 1996)

隠蔽代数 (Goguen and Malcolm 1997)

↓

射影の導入 (Iida, Matsumoto, et al. 1998)

↓

射影型振舞仕様 (Matsumoto 2000)

現在までに行った研究

- テスト集合余帰納法 (WRLA'98)
- コンポーネントの組合せ方の記述と詳細化検証 (OBJ/CafeOBJ/Maude at FM'99)
- 単純観測振舞仕様 (APSEC'99)
- 射影型振舞仕様と木アーキテクチャ (ICFEM'2000)
- 詳細化検証，コード生成の支援ツール (APSEC'2000)

発表の流れ

- 射影型振舞仕様(PSBS)
- コンポーネントソフトウェア開発(CBSD)
- CBSDへのPSBSの適用

射影型振舞仕様

- 仕様言語CafeOBJ
- 振舞仕様
- 振舞等価の検証方法
- 射影を用いたコンポーネント合成
- 詳細化検証方法
- 射影型振舞仕様

仕様言語CafeOBJ

仕様言語

- 数式で記述し，同じ言葉を違った意味に解釈するという問題を除去
- 仕様に対する要求抽出の補助
- 論理レベルのバグの除去

歴史

前身: OBJ3

1992～: 二木研を中心に研究（IPAのプロジェクト等）

特徴

- 振舞仕様，書換え論理のサポート

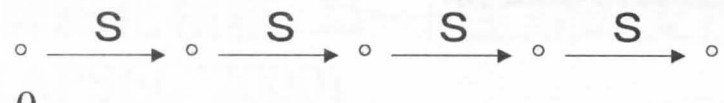
仕様言語CafeOBJ

自然数の仕様

```
mod! NAT { [ Nat ]
  op 0 : -> Nat        op s_ : Nat -> Nat
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq 0 + N = N .       eq s(M) + N = s(M + N) . }
```

モデル

代数 ＝（集合，集合上の関数）



仕様言語CafeOBJ

自然数の仕様

```
mod! NAT { [ Nat ]
  op 0 : -> Nat        op s_ : Nat -> Nat
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq 0 + N = N .       eq s(M) + N = s(M + N) . }
```

性質の検証

s(s(0)) + s(0) = 0 + s(s(s(0))) (2 + 1 = 0 +3)

s(s(0)) + s(0) → s(s(0)+s(0)) → ... → s(s(s(0)))

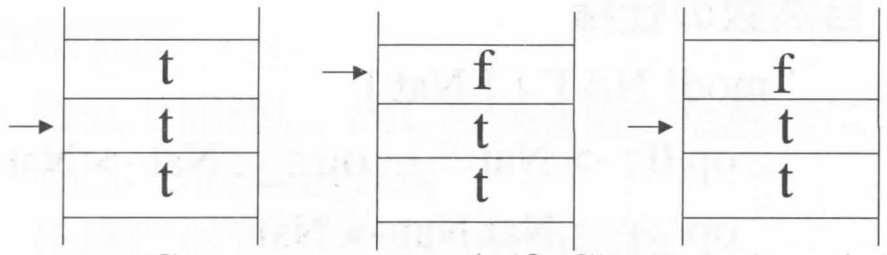
0 + s(s(s(0))) → s(s(s(0)))

```
red s(s(0)) + s(0) == 0 + s(s(s(0))) .
```

射影型振舞仕様

- 仕様言語CafeOBJ
- 振舞仕様
- 振舞等価の検証方法
- 射影を用いたコンポーネント合成
- 詳細化検証方法
- 射影型振舞仕様

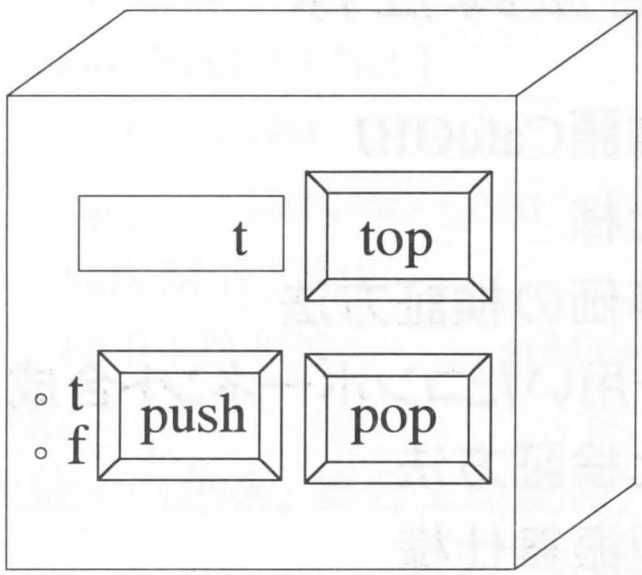
振舞仕様



~~S = pop(push(f, S))~~

top(S) = top(pop(push(f, S))
top(pop(S)) = top(pop(pop(push(f, S))))
top(push(t, S)) = top(push(t, pop(push(f, S))))

振舞仕様



観測
- top

操作
- push
- pop

ブラックボックス（ブール値を蓄えるスタック）

振舞仕様

システムをブラックボックスと見なして仕様記述

ブール値を蓄えるスタックの振舞仕様

```
mod* STK { pr(BOOL)  *[ Stk ]*
   bop top_  : Stk -> Bool
   bop push  : Bool Stk -> Stk
   bop pop_  : Stk -> Stk
   var S : Stk    var B : Bool
   eq top push(B, S) = B .
   beq pop push(B, S) = S . }
```

bop : 振舞演算，beq : 振舞等式（振舞等価）

振舞仕様

top(S) = top(pop(push(f, S))

top(pop(S)) = top(pop(pop(push(f, S))))

任意の c . c(S) = c(pop(push(f, S))).

コンテキスト(c)

- 操作を何度か行った後に行う観測
- 振舞演算の列

振舞等価

任意の c . c(s) = c(t). ←→ beq s = t .

射影型振舞仕様

- 仕様言語CafeOBJ
- 振舞仕様
- 振舞等価の検証方法
- 射影を用いたコンポーネント合成
- 詳細化検証方法
- 射影型振舞仕様

振舞等価の検証方法

コンテキスト帰納法 (Hennicker 1990)

- コンテキストの長さに関する帰納法
- 補題の発見が必要

余帰納法 (Bidoit and Hennicker 1996, Goguen and Malcolm 1997)

- 基底を与えるコンテキスト帰納法

テスト集合余帰納法 (1998)

- 基底の自動生成が特徴

射影型振舞仕様

- 仕様言語CafeOBJ
- 振舞仕様
- 振舞等価の検証方法
- 射影を用いたコンポーネント合成
- 詳細化検証方法
- 射影型振舞仕様

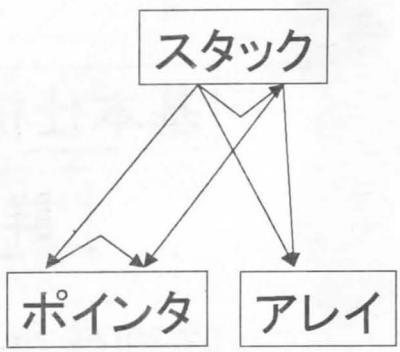
射影を用いたコンポーネント合成

(状態の)射影

- 振舞演算間の

対応関係を定義

eq ptr pop S = p ptr S .
eq arr pop S = arr S .



複合コンポーネントの仕様

•等式群は，射影演算を用いた振舞演算間の対応関係の定義(原始コンポーネントの組合せ方の記述)

射影型振舞仕様

- 仕様言語CafeOBJ
- 振舞仕様
- 振舞等価の検証方法
- 射影を用いたコンポーネント合成
- 詳細化検証方法
- 射影型振舞仕様

詳細化検証方法

基本仕様

↓ 詳細化

詳細仕様1

↓ 詳細化

詳細仕様2

詳細化検証で行うこと

- 生成物間の整合性チェック

詳細化検証方法

例）元の仕様: スタックの仕様

詳細化した仕様: ポインタ付きアレイによる
スタックの仕様

詳細化した仕様上で，次の性質が成り立つことを示す.

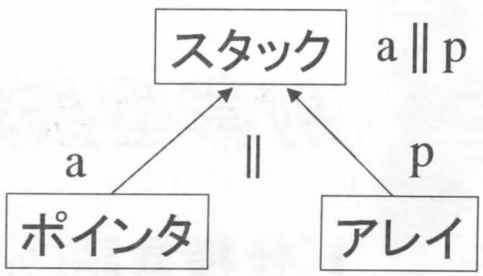
eq top push(B, S) = B .

beq pop push(B, S) = S .

この時，redコマンドを利用できる.

特徴: 計算機支援が得られること

詳細化検証方法



Bidoit, Hennicker 1996

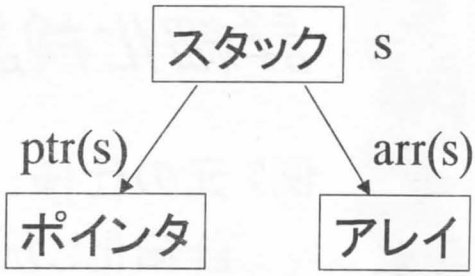
Goguen, Malcolm 1997 の方法

- 複合ｺﾝﾎﾟｰﾈﾝﾄの正しい状態（a || p）を保証するため，到達可能制限を満たすモデルのみを仕様のモデルと考える

↓

- 状態に関する帰納法，補題の発見が必要になることがある

詳細化検証方法



射影を用いる 方法

- 複合コンポーネントの状態は，常に正しいので，モデルに制限を設ける必要がない

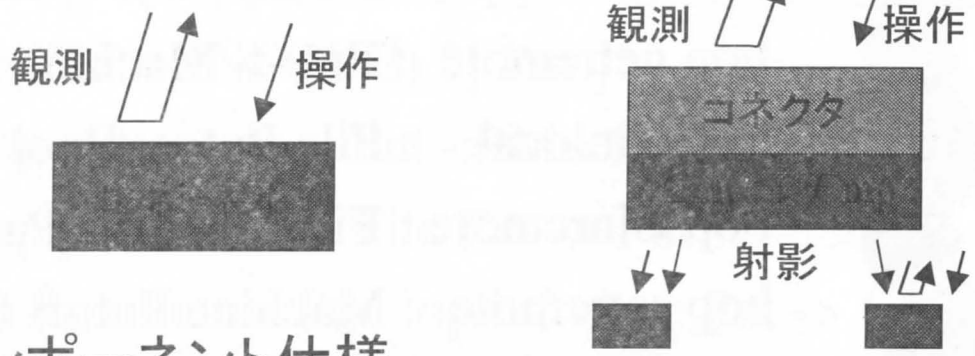
↓

- 状態に関する帰納法，補題の発見が不要

射影型振舞仕様

- 仕様言語CafeOBJ
- 振舞仕様
- 振舞等価の検証方法
- 射影を用いたコンポーネント合成
- 詳細化検証方法
- 射影型振舞仕様

射影型振舞仕様



コンポーネント仕様

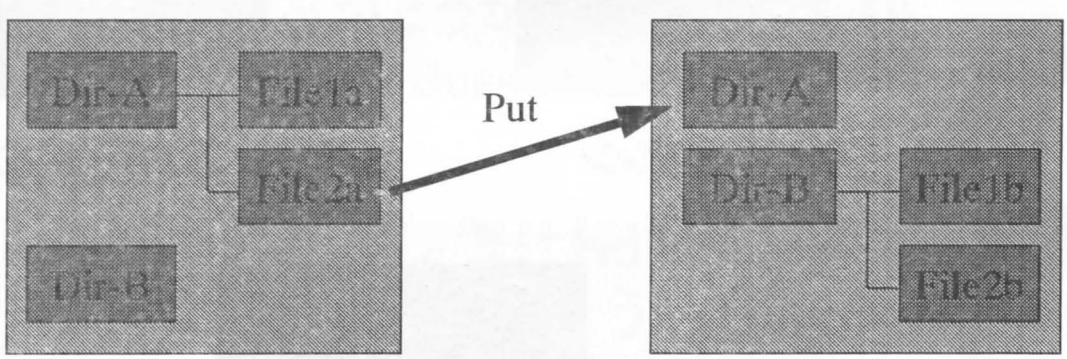
- 操作の観測結果への影響を記述

コネクタ仕様

- コネクタの観測(操作)と原始コンポーネントの観測(操作)との間の対応関係を記述

コンポーネント仕様の例

Put : ローカルマシンからリモートマシンへのファイル転送プログラム

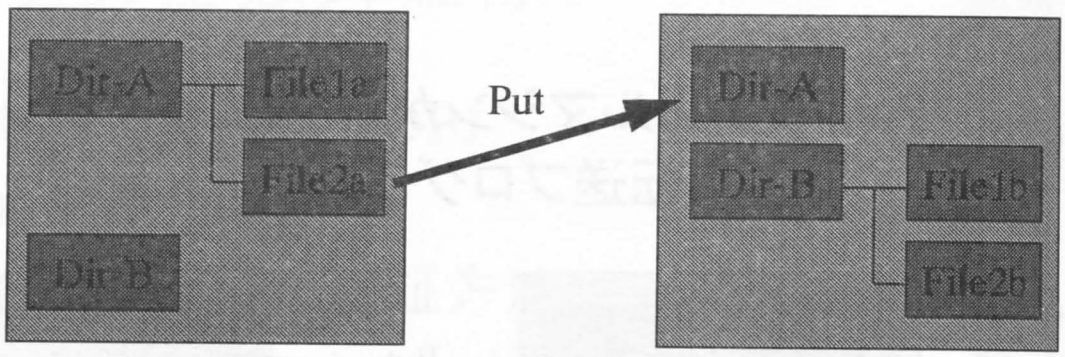


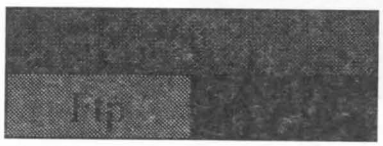
ローカルマシン　　リモートマシン

コンポーネント仕様の例

```
mod* PUT { pr(BOOL + ...)  *[ Put ]*
  bop getremote  : Put -> Machine
  bop isinlocal    : File Put -> Bool
  bop isinremote : File Machine Put -> Bool
  bop setremote  : Machine Put -> Put
  bop put            : File Put -> Put
  var P : Put   vars I J : File   var M : Machine
  eq isinlocal(I,put(J,P)) = isinlocal(I,P) .
  ceq isinremote(I,M,put(J,P)) = t
      if  (I == J) and (getremote(P) == M) .
```

コネクタ仕様の例





Put

コネクタ仕様の例

```
mod* PUT { pr(BOOL + MACHINE +FILE)
 pr(FTP + INFODB) *[ Put ]*
 bop getremote  : Put -> Machine    ・・・
 bop put            : File Put -> Put
 op ftp : Put -> Ftp   op infodb  : Put -> Infodb
 eq getremote(P) = getmachine(infodb(P)) .
 eq ftp(put(I,P))
 = put(I,getmachine(infodb(P)),
       name(getmachine(infodb(P))),
       passwd(getmachine(infodb(P))),ftp(P)) .
```

射影型振舞仕様

一般の振舞仕様

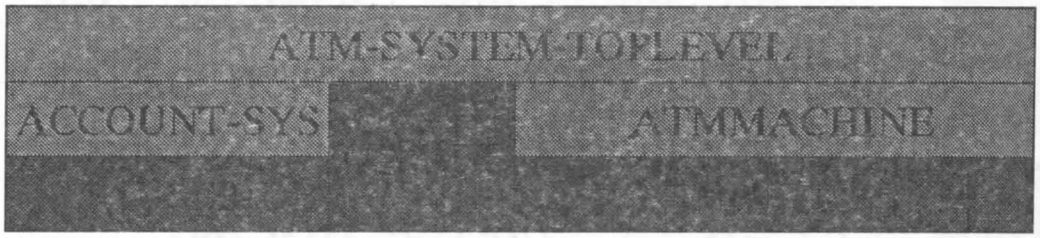
- 任意の振舞仕様に対し，任意の振舞等式が成立するかどうかを判定する推論システムは存在しない (Buss et al. 2000)

射影型振舞仕様

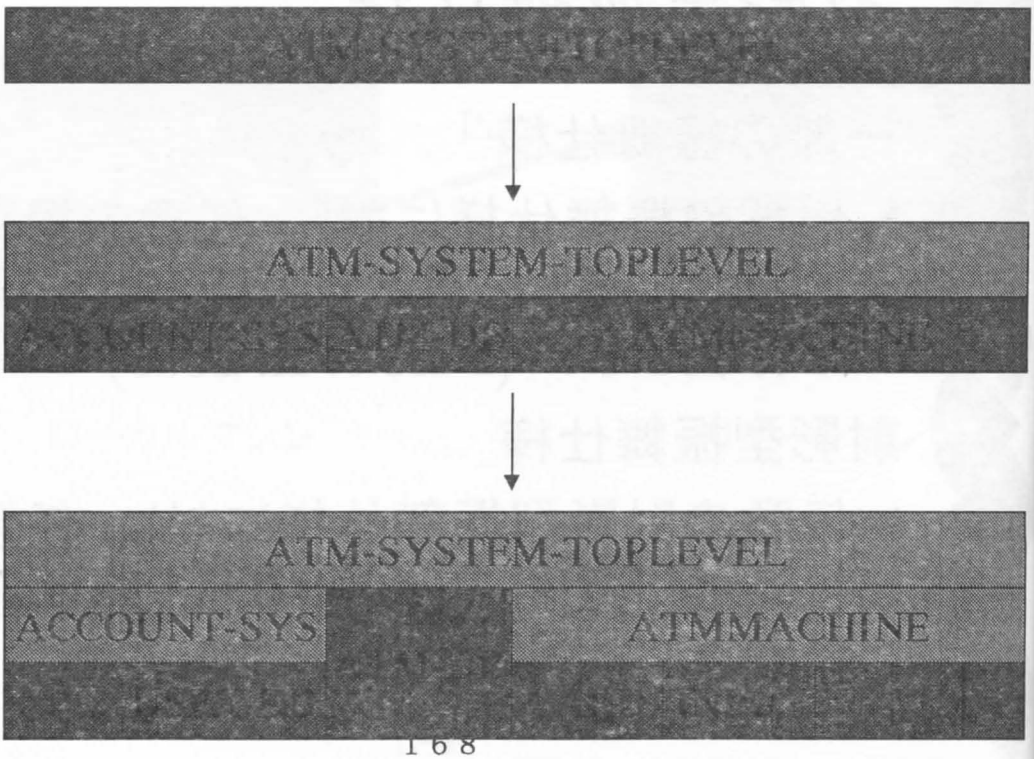
- 任意の射影型振舞仕様に対し，任意の振舞等式が成立するかどうかを等式推論（項書換えシステム）で判定できる.

射影型振舞仕様

The ATM system (Matsumoto et al. 1999)



段階的詳細化



発表の流れ

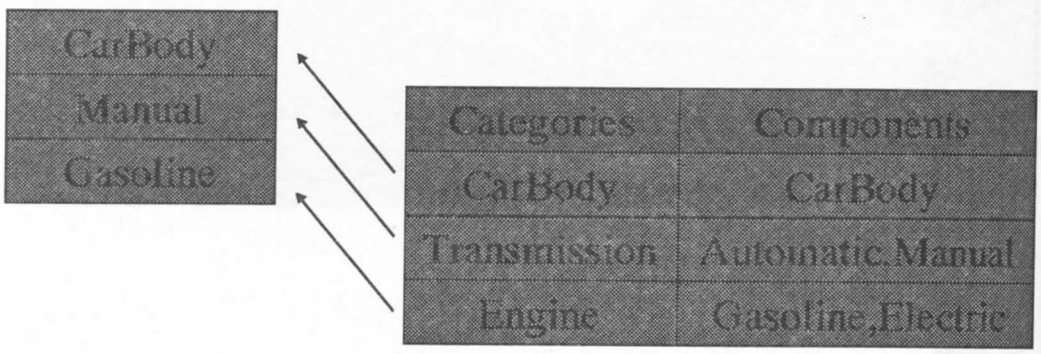
- 射影型振舞仕様(PSBS)
- コンポーネントソフトウェア開発(CBSD)
- CBSDへのPSBSの適用

コンポーネントソフトウェア開発

- コンポーネントソフトウェア開発
- 標準技術

コンポーネントソフトウェア開発

- コンポーネントライブラリを用意
- ソフトウェアは，コンポーネントを組み合わせることで作成



Component library

コンポーネントソフトウェア開発

アーキテクチャミスマッチ (Garlan et al. 1994)

異なるアーキテクチャを持つコンポーネントは結合できない（仮定の対立）.

- The existance of X library vs. No existance of X library
- Reply routine vs. Callback routine

アーキテクチャミスマッチを避ける為の手段

- ソフトウェアアーキテクチャの固定
- Assumption listの使用（Bass et al. 1998）

コンポーネントソフトウェア開発

Barrocaらによるコンポーネントライブラリの検討（Barroca et al. 1999）

- 一般向け4000個のコンポーネントライブラリを利用したスェーデンの電話会社のプロジェクトの失敗
- 特殊向け100個のコンポーネントライブラリを利用したHPの電話交換システムの成功

↓

- 対象ドメインを固定すべき
- ライブラリの大きさは100個以下にすべき

コンポーネントソフトウェア開発

コンポーネントの再利用に必要なこと

- ユーザとメーカ間の共通理解（標準技術）

↓

コンポーネント開発に関する技術の満たすべき要件

- 標準技術と統合できる形で提供できること

コンポーネントソフトウェア開発

- コンポーネントソフトウェア開発
- 標準技術

標準技術

標準技術の満たすべき条件

- ソフトウェア技術者に受け入れられること
- フィードバックにより進化していくこと
- コンポーネントソフトウェア開発を考慮していること

↓

UML関連技術が有望

標準技術

UMLのフィードバックによる進化

- RFP を作成
- TF の投票による RFP の承認 / 非承認
- RTF による revision up

標準技術

UML関連のコンポーネントソフトウェア開発への考慮

- カタルシス法
- UML Profile for EDOC
- Reusable asset specification
- UML2．0

発表の流れ

- 射影型振舞仕様(PSBS)
- コンポーネントソフトウェア開発(CBSD)
- CBSDへのPSBSの適用

CBSDへのPSBSの適用

- Programming-in-the-Large
- 木アーキテクチャ
- 開発支援ツール
- 標準技術との統合へ

Programming-in-the-Large

- DeRemerらの研究(DeRemer et al. 1976)に始まり，その後MIL，ADLとして発展.

MIL75

- 属性文法を用いてモジュールの階層構造を記述
- 設定したアクセス権どおりにリソースにアクセスしているかを検証
- 各モジュールが，設定したリソースを提供しているかを検証

Programming-in-the-Large

MIL

- リソース管理等のNon-functionalな性質の記述，検証

↓

ADL(Wright(Garlan94),Rapide(Luckham95))

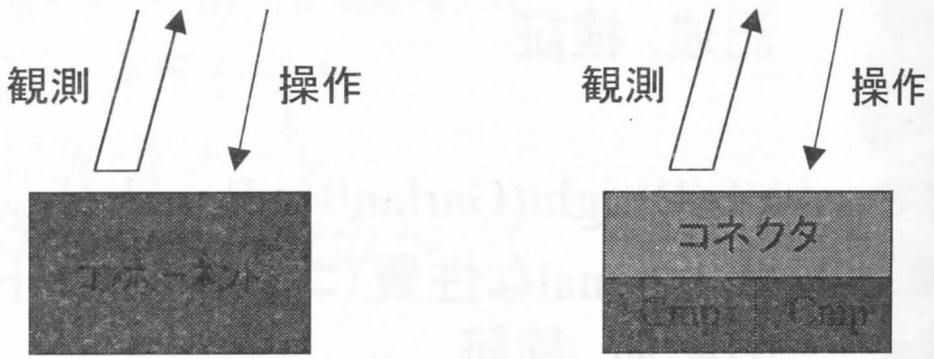
- Functionalな性質（コンポーネントの振舞い）の記述，検証
- コンポーネント間のcommunication protocolの記述

CBSDへのPSBSの適用

- Programming-in-the-Large
- 木アーキテクチャ
- 開発支援ツール
- 標準技術との統合へ

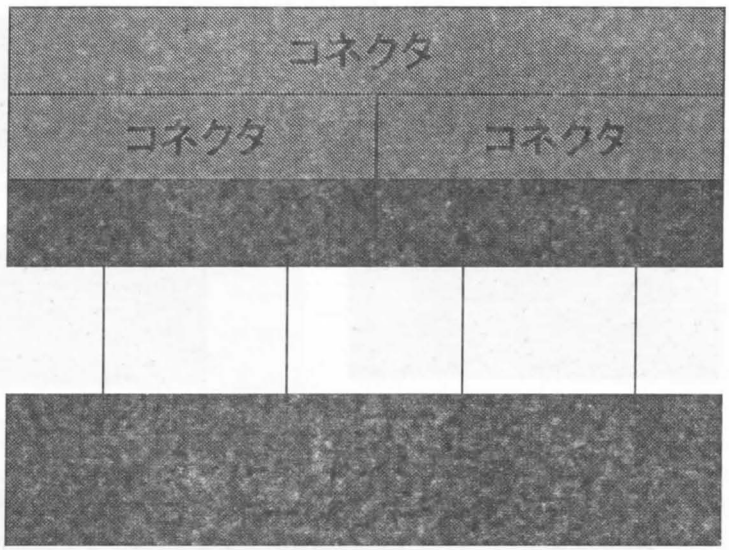
木アーキテクチャ

コンポーネントのイベントモデル



- コンポーネントのイベントモデルは，状態を持つ

木アーキテクチャ

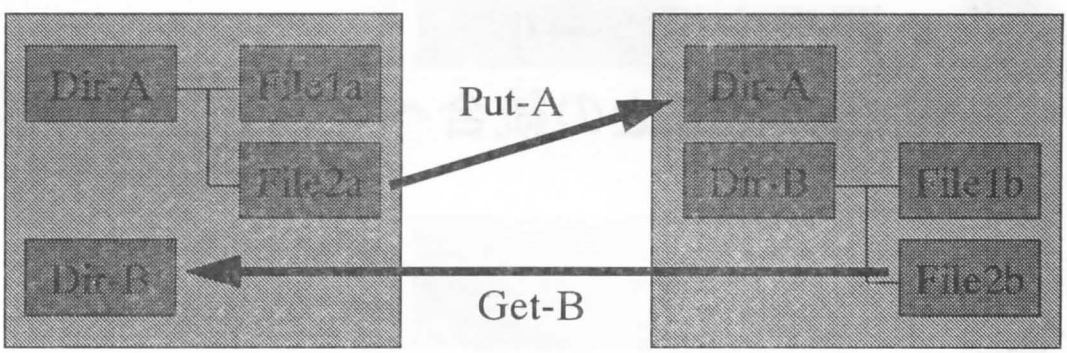


- ソフトウェアファミリ(ドメイン)を固定
- イベントモデルでコンポーネントをモデル化

木アーキテクチャの例

ソフトウェアファミリ : ファイル転送プログラム

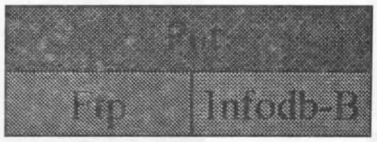
Put-A, Put-B, Get-A, Get-B



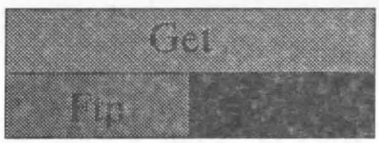
ローカルマシン　　リモートマシン

木アーキテクチャの例

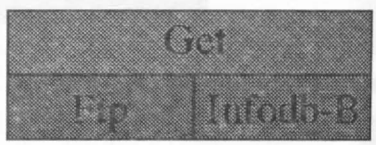
Put-A



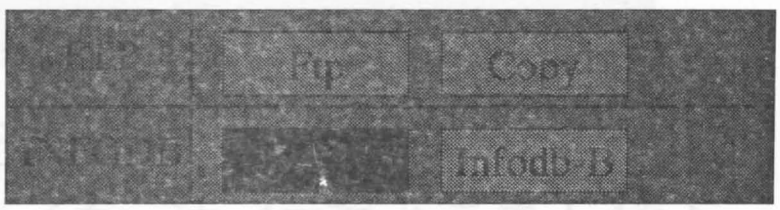
Put-B



Get-A



Get-B



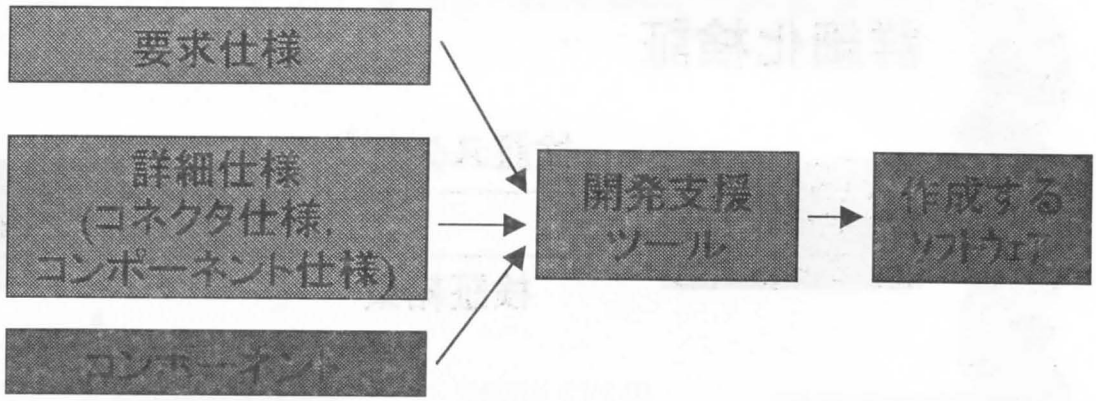
コンポーネントライブラリ

CBSDへのPSBSの適用

- Programming-in-the-Large
- 木アーキテクチャ
- 開発支援ツール
- 標準技術との統合へ

開発支援ツール

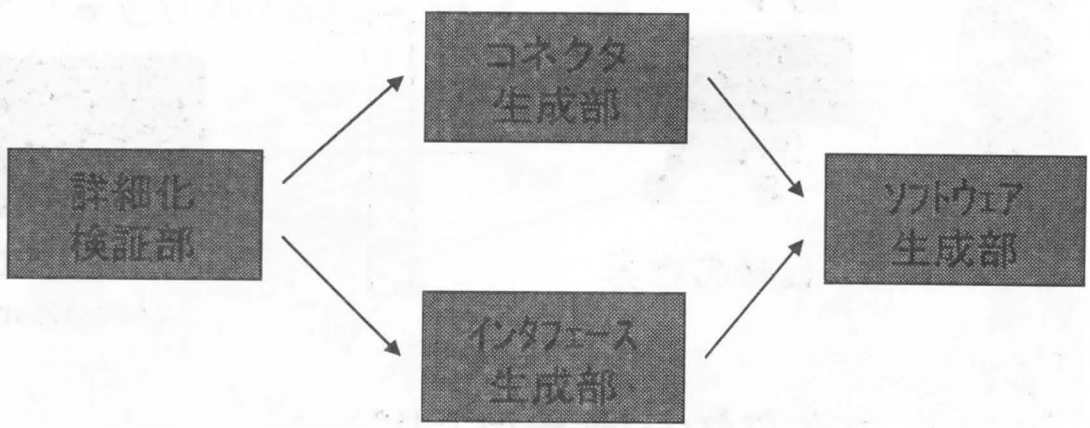
開発支援ツールの入出力



- 詳細仕様(コンポーネントの組合せ方)が要求仕様を満足することを自動検証
- コネクタの自動生成

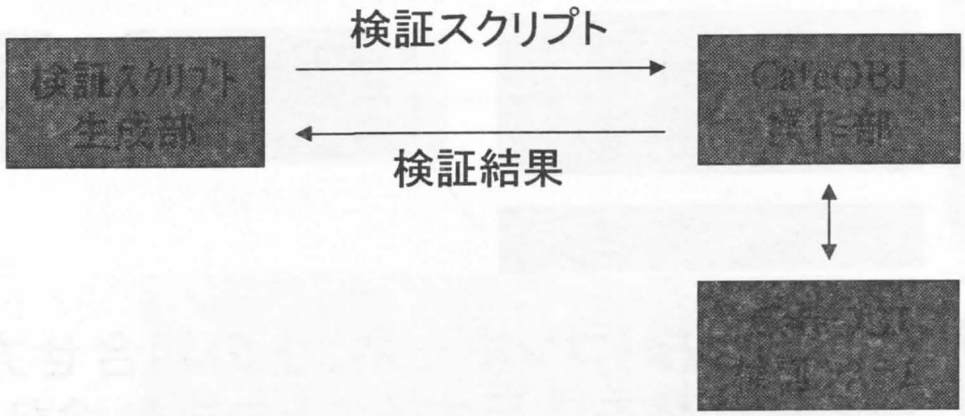
開発支援ツール

開発支援ツールの構造



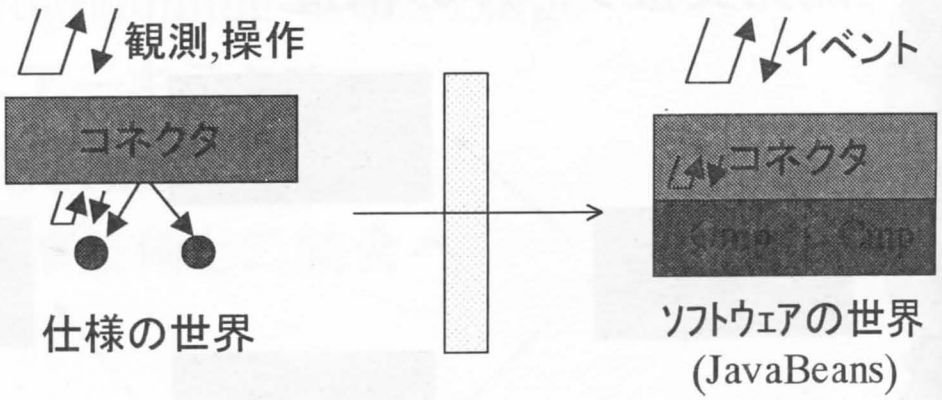
開発支援ツール

詳細化検証



開発支援ツール

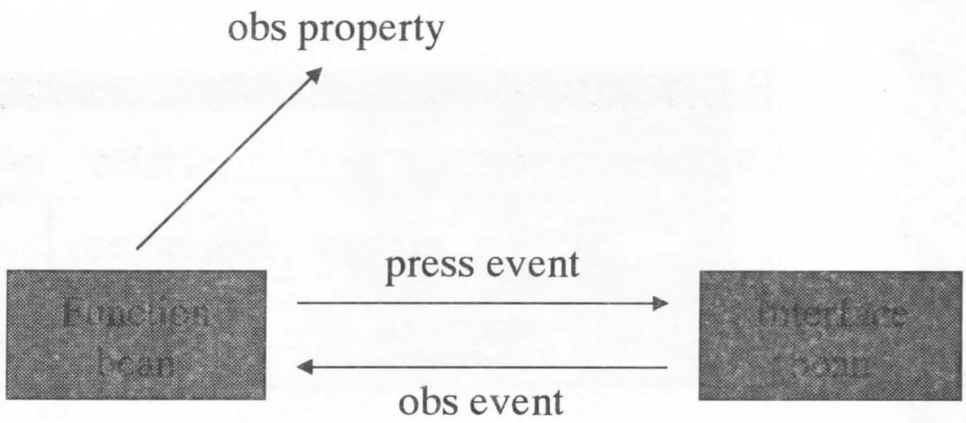
コネクタの自動生成



- コネクタの構造は同じ

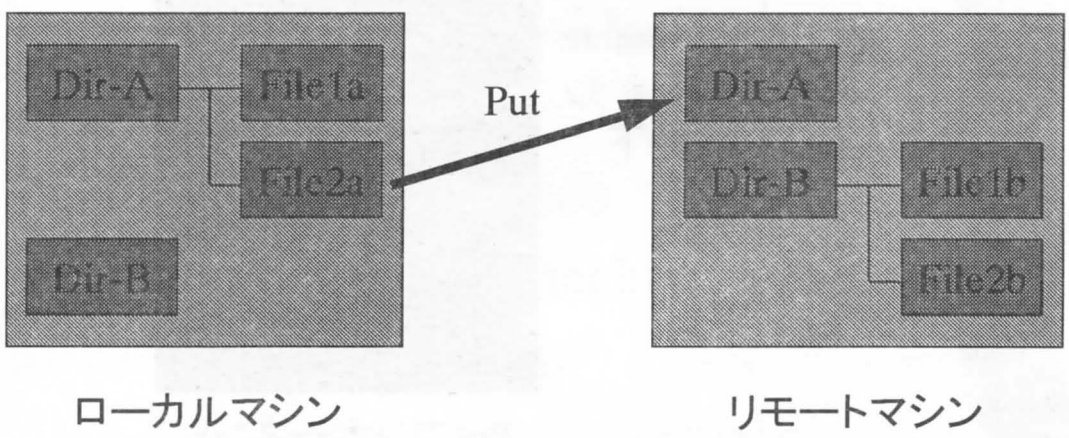
開発支援ツール

コンポーネントのJavaBeansによる実現



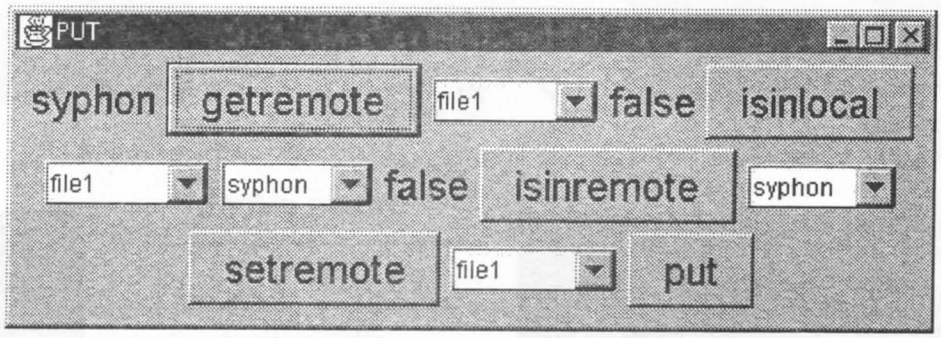
開発支援ツール

Put : ローカルマシンからリモートマシンへのファイル転送プログラム



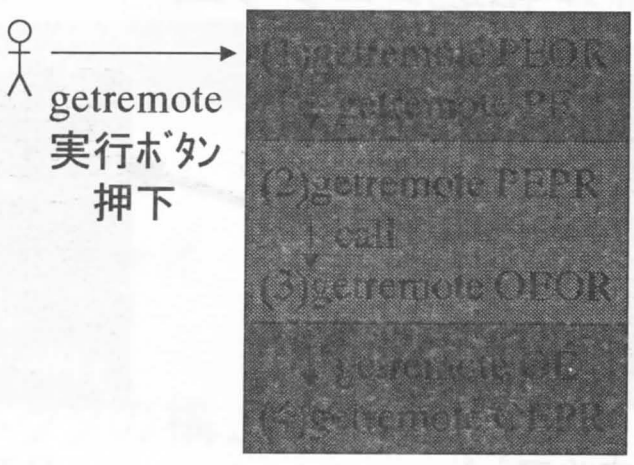
開発支援ツール

Putコンポーネントの外見



開発支援ツール

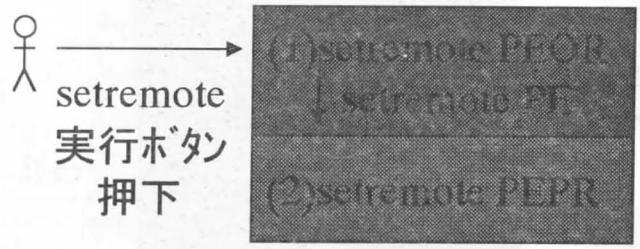
コンポーネントにおける
観測(getremote実行ボタン押下)時の処理



Put コンポーネント

開発支援ツール

コンポーネントにおける
操作(setremote実行ボタン押下)時の処理



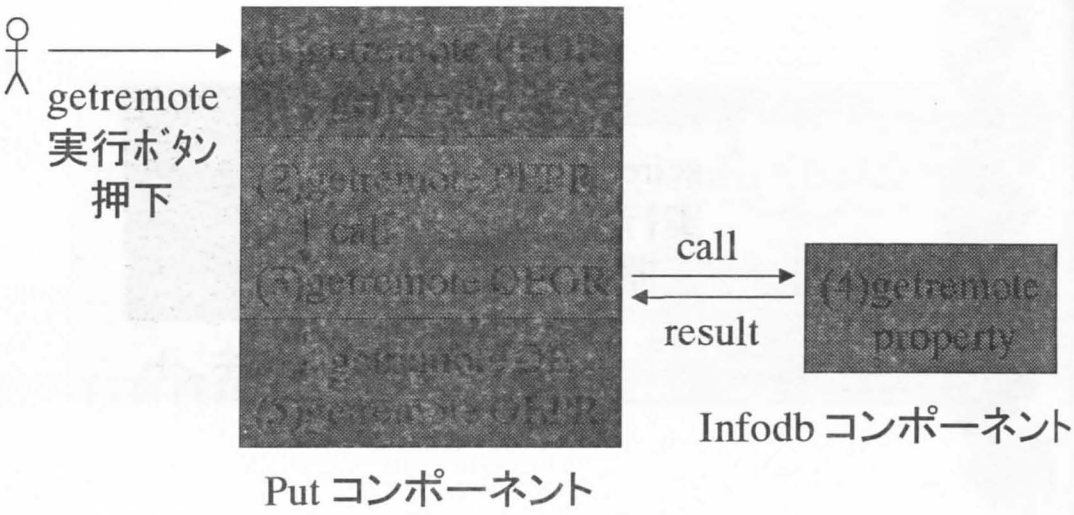
Put コンポーネント

開発支援ツール

```
mod* PUT { pr(BOOL + MACHINE +FILE)
 pr(FTP + INFODB) *[ Put ]*
 bop getremote  : Put -> Machine    ・・・
 op ftp : Put -> Ftp
 op infodb  : Put -> Infodb
 eq getremote(P) = getmachine(infodb(P)) .
                :
```

開発支援ツール

複合コンポーネントにおける
観測(getremote実行ボタン押下)時の処理

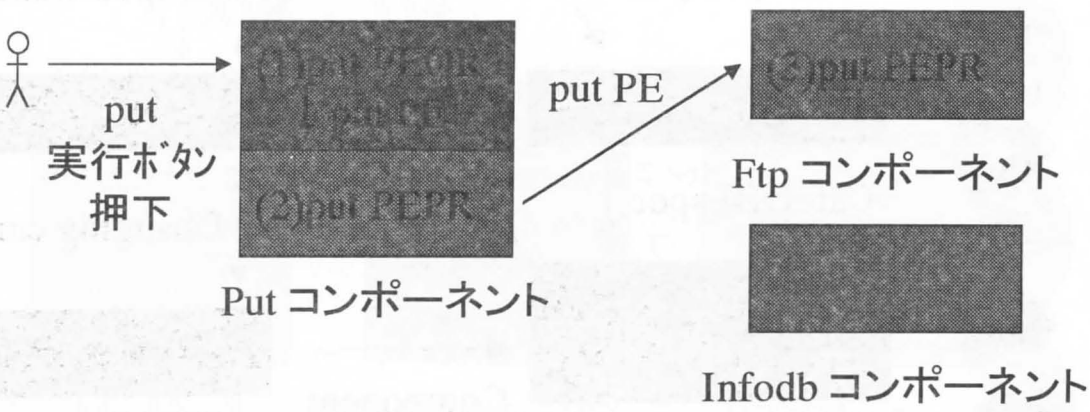


開発支援ツール

```
mod* PUT { pr(BOOL + MACHINE +FILE)
 pr(FTP + INFODB) *[ Put ]*
 bop put          : File Put -> Put    ・・・
 op ftp : Put -> Ftp
 op infodb  : Put -> Infodb
 eq ftp(put(I,P))
 = put(I,getmachine(infodb(P)),
       name(getmachine(infodb(P))),
       passwd(getmachine(infodb(P))),ftp(P)) .
 eq infodb(put(I,P)) = infodb(P) . ・・・
```

開発支援ツール

複合コンポーネントにおける操作(put実行ボタン押下)時の処理

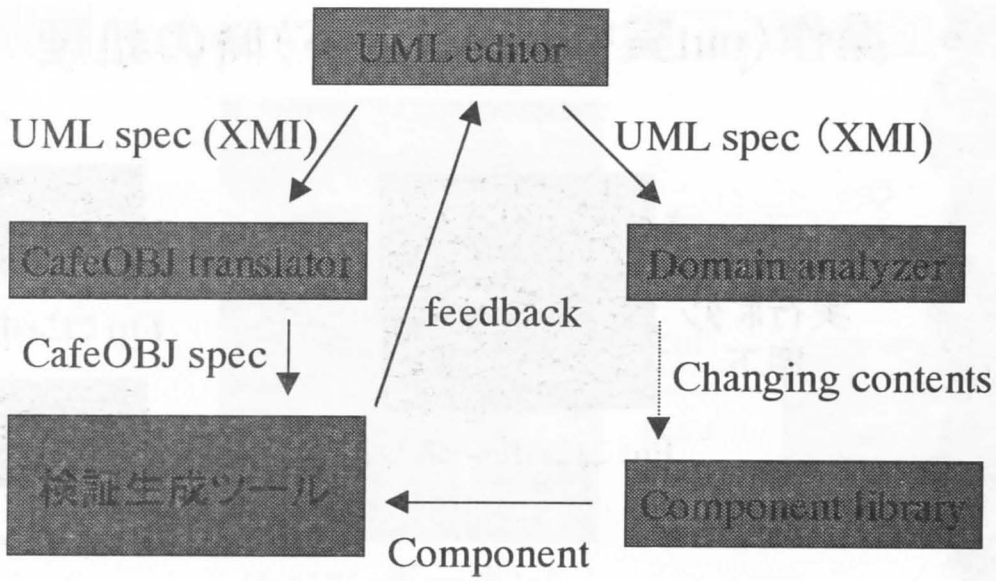


CBSDへのPSBSの適用

- Programming-in-the-Large
- 木アーキテクチャ
- 開発支援ツール
- 標準技術との統合へ

標準技術との統合へ

開発支援ツールの将来像



標準技術との統合へ

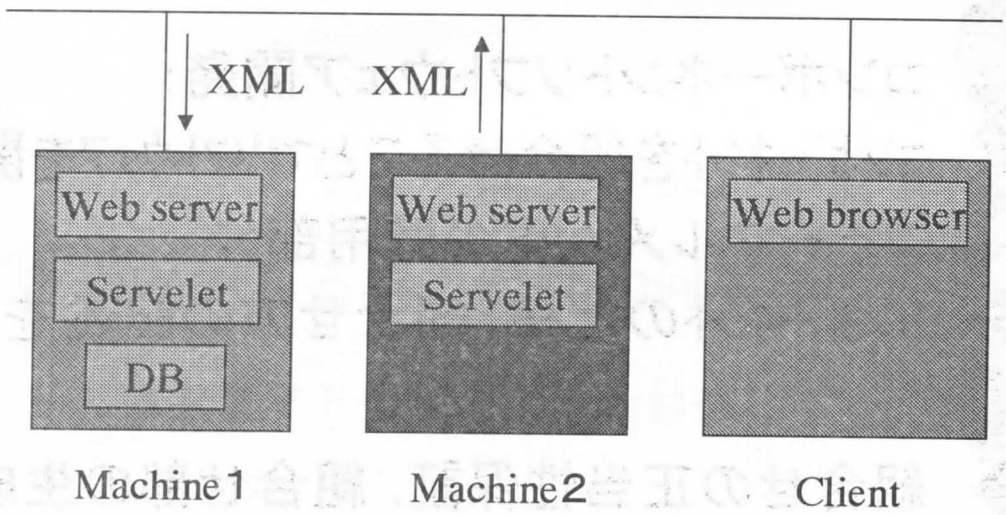
(1) 開発支援ツールの作成

- 射影型振舞仕様作成に必要な情報を記述できる体系を，UMLの拡張として実現
  （Stereo type　or　UML Profile）
- この実現をサポートするツールの作成
  （Rose tool　or　Visio enterprise）

標準技術との統合へ

(2) コンポーネントライブラリの作成

- コンポーネントソフトウェアはサーバ



標準技術との統合へ

（予備実験）

対象ドメイン

- 航空券予約システム

コンポーネント

- 各航空会社のデータを操作するコンポーネント
- 航空券を選択するコンポーネント

まとめ

フォーマルメソッド：射影型振舞仕様

仕様言語：CafeOBJ

コンポーネントソフトウェア開発：

ｺﾝﾎﾟｰﾈﾝﾄを組合せることでｿﾌﾄｳｪｱを開発

フォーマルメソッドの適用部分：

ｺﾝﾎﾟｰﾈﾝﾄの仕様，組合せ方の記述 を用意

↓

組合せの正当性保証，組合せ部の生成

ソフトウェア工学に於いて
「数学的である事」とは如何なる事か？
— 精密ソフトウェア工学を目指して —

こんどう ひでたか
金藤 栄孝

(株) 日立製作所　システム開発研究所

SIGFM, 9 March 2001

ソフトウェアでの「理論」と「実践」との関係

理論計算機科学 : ソフトウェア工学

≅ ? : ?

余談

私は「(理論) 計算機科学」(“(theoretical) computer science”)
という用語が大嫌いである.

何故なら,「計算機」という製品分野の名称と,本質的には真理探求の学を
表わす「科学」を繋ぐのは,カテゴリ・エラー以外の何物でもないから.
(「船舶工学」とは呼んでも「船舶科学」とは呼ばないでしょう.)

最近,“informatics”(「情報学」?)なる語が使われているが,
この語の語幹の “information” (「情報」) という言葉も
ソフトウェアの基礎分野を表わす目的には範囲が広すぎる.

計算機ソフトウェアの本質は「計算できる事」にこそ有るのだから,
それを基礎付ける科学分野を表わす名称には,

“computationics”(訳語としては「算理学」)

を提案したい.

Hoare & He (1998) の立場

≅ 物理科学の各分野 : 従来の各工学分野

しかし,

物理法則は**帰納的**に基礎付けられる. 一方,

計算は**演繹的**理論に基づいている.

この他にも, Hoare & He は,

物理科学に対する**素朴過ぎる「信仰」**

が散見される.

例えば, 現状の物理学は, 彼らがその本で主張している様に,

統一的でも**完全な還元主義的**成功を収めている訳でもない!

例えば,「何故, 飛行機が飛ぶのか?」の原子レベルでの説明, 即ち,

Navier-Stokes 方程式が, 何故, あの形なのかの原子レベルでの説明は,

未だ判っていない.

我々の提案

	理論計算機科学	:	ソフトウェア工学
≅	数理論理学	:	数学
=	数学を 分析する 事	:	数学を 行なう 事

即ち,

	ソフトウェアを 基礎付ける 事	:	ソフトウェアを 開発する 事
≅	数学を 基礎付ける 事	:	数学理論を 生み出す 事

言い替えると，**我々のテーゼ**とは：

ソフトウェアは数学的に開発されるべきである．

しかしながら，

「数学的である事」とは如何なる事か？

「数学的である事 ($Mathematicalness$)」は，

通常の数学者の視点

に基づいて定義する．

即ち，部外者である我々ではなく，

普通の数学者が「数学だと思う」ものが数学！

形式的仕様は，しばしば

数学的である

と主張される．しかし，

形式的手法は**本当に数学的なのか？**

(前に述べた我々の意味に於いて)

FM は何らかの（数理）論理学 の体系に基づいている

という事を勘案すると，前の質問は，次の様に言い替えても良い．

論理的である事」 ⟺ 「数学的である事」？

論理学的である事」 $\Longleftrightarrow$ 「数学的である事」？

($\Leftarrow$): 自明に真；

∵）数学は本質的には演繹的科学であるから.

($\Rightarrow$): ？！？これは真とは限らない！？！

論理学者の視点 *vs* 数学者の視点

論理学者の（数学を分析する時の）視点は，

還元主義的である！

例：

$$\forall x \in G.\, e \cdot x = x,$$

$$\forall x \in G.\, (x)^{-1} \cdot x = e,$$

$$\forall x, y, z \in G.\, x \cdot (y \cdot z) = (x \cdot y) \cdot z.$$

論理学的な視点からは，これらの論理式は，定数と関数記号とを持つ一階の理論の文以上の何物でもない.

数学者にとっては，これらの公理の定める数学的構造は，殆ど，数学として有意で面白いか否かの基準ですらある.

冗談

論理学者とは，帰納法に最も関心を持っている人種である.

数学者とは，帰納法に関心のない（帰納法が出て来たら自分の仕事は終わったと思う）人種である.

© Hidetaka Kondoh, 1997.

形式化の非効率性

例１：

X を集合，$\sqsubseteq$ と $\underset{\sim}{\sqsubseteq}$ を X 上の半順序，$\simeq$ と $\approx$ を，各々，それらの半順序から導かれる同値関係とする．この時，

$$\forall x, y \in X.\,(x \sqsubseteq y \supset x \underset{\sim}{\sqsubseteq} y)$$
$$\supset \forall x, y \in X.\,(x \simeq y \supset x \approx y)$$

この言明は直感的には自明であるにも拘らず，NK での形式的な証明では，約 10 ステップを要する．

例2：簡単なプログラムの正当性証明からの抜粋

$$NotYetFound(i, j-1) \wedge a[i,j] = 0$$
$$\wedge\, i \in [1..m] \wedge j \in [1..n]$$
$$\supset (MinimalIndex(i,j) \vee NowhereZero \wedge i > m)$$

where

$$NotYetFound(i,j) \triangleq$$
$$\forall k \in [1..m], l \in [1..n].$$
$$(\langle k,l\rangle \sqsubseteq \langle i,j\rangle \supset a[k,l] \neq 0)$$

$\sqsubseteq$: the lexicographic order on $\mathbb{N} \times \mathbb{N}$

$$MinimalIndex(i,j) \triangleq$$
$$\forall k \in [1..m], l \in [1..n].$$
$$\Big((a[k,l] = 0 \supset \langle i,j\rangle \sqsubseteq \langle k,l\rangle)$$
$$\wedge\, i \in [1..m] \wedge j \in [1..n]\Big)$$

$$NowhereZero \triangleq \forall k \in [1..m], l \in [1..n]., a[k,l] \neq 0$$

意味的に考えると自明にも拘らず，形式的にこれを証明するには約20ステップを要する.

教訓

人間にとっては極めて**自明な**「証明ステップ」でさえも，形式的証明では**10ステップかそれ以上**を必要とする.

即ち,

形式的証明は数学での証明と比べて，**規模が文字通り桁違いに大きい**

普通の数学者の「生産性」

形式的数学者 (補助概念)

（仮想的なクラスの）数学者で，通常の数学者と同様，**数学を行なう（即ち，新しい数学理論を生み出す）**事を生業とするが，それを**形式的に**行なう点が通常の数学者とは異なる.

注意：

形式的数学者を，既存の数学理論を形式言語で記述された形に翻訳／収集する活動をしている研究者（大抵は，自動定理証明等の研究者）とは**混同しない様に**せよ.

数学者の生み出す**数学的内容に関する彼の生産性は，彼が形式的数学者であると考える**事で測る事ができる.

（即ち，論文や書籍を形式的な言語に翻訳した時の量）

例 1　**Barendregt の "*The Lambda Calculus*"：**

約 600 ページ,

約 30 行 / ページ,

換算尺度 (informal→formal) 20

（この本の証明には殆ど自明でないギャップは無いので極めて小さい値）

⇓

この本の内容は約 400,000 formal lines (LOC) に相当すると見積もる事が出来る.

例２．

難しい数学の論文誌に載る長さが 20 ページで 30 行／ページの論文（そいつらの証明は，大抵，我々の様な**素人では到底埋め難い「ギャップ」だらけ**なのであるが）

⇓

その数学的な内容の推定評価量は，約 60,000 ～ 600,000 LOC！

教訓

形式化（特に，**形式的証明**）は，内容に関する生産性を大幅に低下させる！

観察

Software Formalists（形式的な検証の研究者）は，彼ら自身の論文を形式的に書いた事もなければ，彼ら自身の（彼らが研究しているFMの性質に関する）メタ定理を形式的に証明した事もない！

事実

ソフトウェア技術者は，既にハンディキャップを背負っている．即ち，彼らの最終成果物であるプログラムは，完全に形式的な代物でなければならない．

社会的対応関係: ソフトウェア *vs* 数学

或いは，**連中は何で給料を稼いでいるのか？**

ソフトウェア技術者	数学者
プログラム	(新しい定理を含んだ) 新しい数学の理論 (に関する論文)
正当性の証明	定理の証明

⇓

何れの場合も，証明の位置付けは，生産物そのものではなく，

品質保証の手段

注記

ソフトウェアの正しさは，ソフトウェア技術者の主たる目的の一つである（或いは，「あるべきである」）が，

正当性の証明自身の（絶対的な）正しさは，彼らの主要な職務の範囲外である.

テーゼ

プログラムは，再帰的関数を実現する形式的対象である.

従って，**プログラムは形式的な枠組の下で開発されるべきである.**

⇓

形式的手法

このテーゼに基づく**最も極端なアプローチ**は：

構成的プログラミング

De Bruijn-Curry-Howard の対応: プログラム *vs* 論理

プログラミング	論理
仕様	命題
プログラム	証明

限界

この視点は，programming-in-smalls のスケールであり，現実のソフトウェア・システムの大規模スケールには対応していない．（純粋な数理論理学的視点では，「補題」の設定の善し悪しの判定が行なえないのと同様）

アンチテーゼ

プログラムは，単に計算機への指令の為だけでなく，人間同士のコミュニケーションの為のものでもある.

従って，**プログラムは楽しく読めるものであるべきだ.**

⇓

Knuth の文芸的プログラミング

人間の活動の中で，プログラミングに最も類似した活動は **文芸作品を書く事**である.

注記

この視点は**エンジニアリングでは無視できない.**

（即ち，保守の問題）

シンテーゼ

プログラムは**形式的な理論体系**に基づいており，同時に計算機だけでなく**人間によっても**読まれねばならない.

しかも，殆どのプログラムは**自明でない規模**である.

故に，**プログラムは，その論理的な特性が良く判っている「良い構造」を用いて設計されねばならない.**

⇓

精密ソフトウェア工学

ソフトウェア開発に最も類似した人間活動は**数学の本を書く事**である.

構造的対応：ソフトウェア *v.s.* 数学

アイデア

構成的プログラミングでのプログラミングと論理との対応関係を，より programming-in-larges のスケールで考えよう.

ソフトウェア開発	数学的活動
基本制御構造	論証の基本ステップ
イディオム	証明テクニック
抽象データ型	数学的概念に関する理論
デザイン・パターン	証明の戦術
アーキテクチャ・スタイル	理論構築の戦略
ソフトウェア・システム	数学理論
ドメイン	数学の分野

基本制御構造 *v.s.* 論証の基本ステップ

この対応は exact である. (*cf.* 構成的プログラミング)

イディオム *v.s.* 証明のテクニック

イディオムとは，複数の**基本制御構造**を特定のパターンで組み合せたものである.

証明テクニックとは，複数の**基本的な論証ステップ**を特定のパターンで組み合わせたものである.
例えば,

Fermat の無限降下法：（初等整数論で良く出て来る）

> **数学的帰納法**と**背理法**とを一定のパターンで組み合わせた論証法.

抽象データ型 *v.s.* 一つの数学的概念に関する理論

抽象データ型とは，（その表現が隠蔽されるべき）共通のデータに作用する**演算の集まり**である.

数学的概念に関する理論とは，その概念の性質を厳密な形で表わしている**命題の集まり**である.

抽象データ型	数学的概念に関する理論
演算の仕様	（概念の性質に関する）命題
抽象データ型の仕様	（概念に関する）理論＝正しい命題の集まり
演算の実現	命題の証明
抽象データ型の実現	理論を構成する命題群の証明の集まり

デザイン・パターン *v.s.* 証明の戦術

デザイン・パターンとは，複数の**抽象データ型（クラス）**とそれらの間の特定の**依存関係（継承）**との組み合わせ方である．

証明の戦術とは，（証明を進める上で必要に応じて導入される補助的な概念の性質を与える）**補題** — **その集まりは，補助概念に関する「小さな」理論を構成する** —とそれら補題や補助概念の間の依存関係に関する知識である．

デザイン・パターン	証明の戦術
抽象データ型（クラス）	補助的な理論としての補題の集合
継承	補題群の間の依存関係

アーキテクチャ・スタイル *v.s.* 理論構築の戦略

アーキテクチャ・スタイルとは，ソフトウェア・システムを構築する為のテンプレートであり，**コンポーネントの部分的な仕様，内部インタフェース，外部インタフェース**を一定の形で組み合わせたものである.

理論構築の戦略とは，**基本的定義，理論が示そうとする目標のパターン（主定理されるべき generic な命題），重要な補題群**の有機的な纏まりである．例えば，

λ- 計算や類似の簡約理論／書き換え理論を構築する場合，

合流性（Church-Rosser 風の定理．即ち，これが理論の目標）を示すには，良く知られたものとしては，大別すると 2 通りの方法がある．（その各々が「理論構築の戦略」である）：

(1) Hindley-Rosen のダイヤモンド補題（この補題自体が generic）を用いる方法；

(2) 並列簡約（この概念自体が generic）を用いる方法.

ソフトウェア・システム *v.s.* 数学理論

ソフトウェア・システム	数学理論
システムの外部仕様	主定理として挙げられた命題
コンポーネントの仕様	補題として示された命題
コンポーネントの実現	補題の証明
ライブラリ・コンポーネント	文献から参照した補題
外部仕様の為の抽象データ型	主定理を述べる為の数学的概念
内部インタフェースの為の抽象データ型	補題を述べる為の補助概念

ドメイン *v.s.* 数学の分野

共に，仕事を進めて行く為に必要な基本的な語彙の集合と基本的な考え方を定める.

ソフトウェア工学で何が欠けているのか？

ソフトウェアのマクロ構造（イディオム，デザイン・パターン，アーキテクチャ・スタイル）の論理的に**厳密な性質を分析し，蓄積し，再利用する**事.

形式的手法は主として**記述する事**に関心がある．しかし
性質の分析の方が記述よりも遥かに重要である．

精密ソフトウェア工学を目指して

- ソフトウェアの様々な粒度の階層で繰り返し現れるマクロ構造の性質を分析し,
- それらの構造に関する**ミニ理論**（「数学理論」と同じ意味で）を構築する事.

この活動は，表現や実現と独立に構造自身を調べるという点で，抽象代数と同じ意味で,

抽象ソフトウェア工学

と呼んでも良い．これこそが，**ソフトウェア開発を直接に支えるソフトウェアの為の学**となる.
（この「抽象ソフトウェア工学」の確立は，アカデミズムの方々のミッション）

その成果に基づく実践としての精密ソフトウェア工学での構造の利用手順は，概ね，

- それらの構造に関して分析から判った論理的な性質を活用して，**それらの構造を，安全に（即ち，バグの入る心配なしに）再利用し，**

- それらの構造を用いて構築されたソフトウェアの正しさを，**個々の構造に対するミニ理論に基づいて効率良く**証明する.

これが，将来あるべきソフトウェア開発の姿である！

応答系の仕様記述

山崎　利治

はじめに

ワープロや組込系などの応答系はそのおかれた環境と交信しながら一定の機能をはたすシステムである．これは変換系と呼ばれることもある伝統的なシステムと違った側面が多く仕様記述や設計にも別の工夫が必要になる．これは，L. Lamport による時間論理の体系である TLA[2] をその応答系の仕様記述に適用する試みの紹介である．

1　応答系とは

変換系 (tranformational system) は入力データを与えプログラムを始動しその停止とともに出力データを得るものであり，プログラムは必ず止まることを要請している．変換系は入力と出力をもち，出力は入力の関数とみることができる．したがって，プログラムの仕様は関数定義，あるいは，前件 (precondition) と後件 (postcondition) の組を与えればよかった．

これに対して**応答系** (reactive system) は系の利用者などの外部環境と相互作用を継続しつつ一定の機能を提供するものであり，つぎのような特徴をもっている．

1．　継続性（非停止性）
2．　対話性
3．　並行性

応答系のプログラムに対しては必ずしも停止することを期待していない．系がもっている時計が止まっては困るわけである．このことは，前件と後件との組としてプログラムの性質が記述できないことを意味し，変換系で行った Hoare 論理などによるプログラムの性質の推論がそのままではできなくなっている．

応答系は環境と交信しながらその機能を果たすのでプロセス間通信が問題になる．交信が同期か非同期かまたそのプロトコルなどを考えなければならない．

応答系は交信問題などによって並行プロセスとして実現されることが多い．その実現が純粋の同時並行でなく差込 (interleaving) による並行性の場合もあるが，並行性が引き起こす非決定性が問題になる．

このような性質をもつ応答系の仕様をつぎのように考えてみる．応答系をその許された挙動の集合とみなす．そこで，挙動を外からみえる系の状態 (状態とは系あるいはプログラムのもつ変数にその値を割り付ける関数である) 成分の変化列として記述する．外からみえない状態成分も便宜上記述したい場合はそれを許してあとで隠蔽すればよい．その状態列を時間論理の述語として表現し，それを仕様と考えるのである．

変換系を議論する機構として抽象プログラム機械があったが，応答系の場合には推移系 (transition system) が役立つ．(札付き) **推移系**とは，**状態**と**札**の集合 S, A があるとき，写像 $\theta : S \to \mathbb{P}(A \times S)$($\mathbb{P}(X)$ は X の部分集合全体を表す) をいう，つまり，θ は状態 s から札 a によって推移する状態 t たちを示すものである．

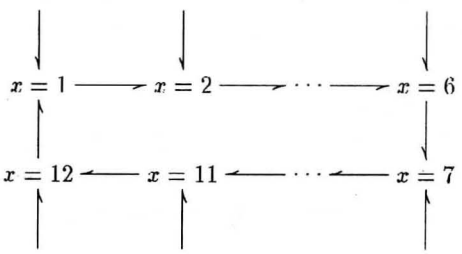
2　例 1

例をみてみよう．つぎは時間だけを表示する時計のプログラムである．これは止まっては困るも

のである．

```
...
const hr ∈ {1..12}
{ x = hr }
do  true → x := (x mod 12) + 1
od
...
```

ここで，プログラム仕様として後件を $\{false\}$ とするのは意味がない．そこでプログラムを推移系と考えればつぎの推移図が書ける．



さて，つぎの状態の無限列はプログラムの実行列になっている．

$\sigma \triangleq x = 11, x = 12, x = 1, \cdots$

状態の無限列を挙動 (実行列) ということにしよう．状態列で，それぞれ連続した２状態を作用 (行動，事象，推移) などといい，前状態と後状態の関係として記述する．たとえば，

$Next \triangleq x' = (x \bmod 12) + 1$

とする．ただし，x' は後状態を参照する．また，初期状態は

$Init \triangleq x \in \{1..12\}$

である．とすると挙動は

$StateSequence \triangleq Init \wedge \Box Next$

と表せ，これを時計プログラムの仕様と考えることができる．つまり，つぎのように考えるのである．ここに挙げたような述語 $StateSequence$ を挙動の上で解釈してその真偽を定める．プライムなしの変数と定数記号による論理式を状態述語といい，この真偽は挙動の最初の状態の上で定める．$Init$ は上の挙動 σ に対してその最初の状態 $x = 11$ で真である．したがって，上の σ に対して真となる．プライムなしおよびプライムつきの変数を含む論理式を作用述語といい，連続した２状態の組でその真偽を定める．$\Box$ は時間論理の「常に」演算子である．$\Box Next$ は，$Next$ が作用述語であるから続いた２状態に対して真偽が定まるが「常に」演算子をつけた場合，それを時間述語というが，すべての連続２状態に対して真の場合に真と定める．$\Box Next$ は上の σ に対しては真となる．つまり，順序対列

$(x = 11, x = 12), (x = 12, x = 1), \cdots$

を考えれば，そのそれぞれに対して $Next$ は真となる．

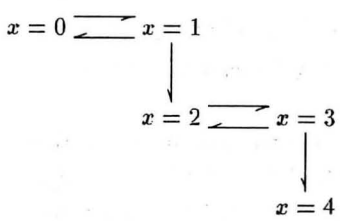
時間述語の厳密な解釈やプログラムの仕様としてこれで十分かなどについてはいまは問わない．

3 例２

つぎのプログラムは非決定性を含む例である．

```
{ x = 0 }
do  x ≤ 3 → x := x + 1
□   odd(x) → x := x − 1
od
```

これもつぎのような推移図が書ける．



またプログラムの実行に伴う状態列の全体は正規表現風に

$(AB)^\omega + (AB)^+(CD)^\omega + (AB)^+(CD)^+E^\omega$

と書いていいだろう，ただし，$A, B, \cdots$ は $A : x = 0$, $B : x = 1$, $C : x = 2$, $D : x = 3$, $E : x = 4$ を表している (有限状態列となるものに対してはその後に最終状態を無限個連接して無限列ととする)．

例１のように考えればプログラムの仕様がつぎの

ように記述できる.

$Init \triangleq x = 0$
$Inc \triangleq x \leq 3 \wedge x' = x + 1$
$Dec \triangleq \mathrm{odd}(x) \wedge x' = x - 1$
$Next \triangleq Inc \vee Dec$
$Spec \triangleq Init \wedge \Box Next$

ここで非決定性に付随する公平性について補足する. 挙動 $ABABABAB\cdots$ は公平さを欠くといえる. 状態 B から状態 C へ推移する機会, つまり, 作用 Inc が実行される機会が断続的ではあるが無限回あるにも拘わらず一度もそうなっていないからである.

普通, 公平さはつぎのように定義されている.

挙動 σ が作用 A に対して弱公平であるとは, A を実行できる状態が無限回連続して生起していれば必ず A が実行されているときにいい, 連続しての条件を除いたとき強公平という.

$ABABAB\cdots$ は作用 (B, C) に関して, また, $ABABCDCDCD\cdots$ は作用 (D, E) に関して強公平 (無論弱公平) ではない. すなわち, それらの作用を実行できる状態が無限回起こっているのに実行されていないからである. 強公平を保証すれば例題プログラムは必ず E 状態に到達するだろう.

4 TLA

ここで例1,2で試みたような仕様記述を実務で使うために Leslie Lamport の TLA('Temporal Logic of Actions)[2] を紹介する. これはプログラム (システム, 成分) の記述と推論のための離散線形時間論理体系である. TLA ではプログラムとその性質をおなじ言語で記述する. TLA は**作用** (行動, 事象, 推移) を基本概念とし, これによって TLA は, いわゆる, stuttering 不変な言語になっている. この性質は仕様作成に際して定義対象の抽象・洗練, 分解・合成の面で役に立つ.

TLA による仕様は対象のもつ変数の時間的変化を記述するものである. 変数, 状態, 状態の対である作用, 状態の無限列である挙動が基本概念である.

4.1 構文

まず, TLA 構文を述べる. 基本式としてつぎを定義する.

$c, \cdots$	**定数記号**
$m, \cdots$	**時間非依存変数** (rigid variable)
$x, x', \cdots$	(時間依存) **変数** (flexible variable)(プライムなし, プライムつき)
$f, f', \cdots$	**状態関数** (state function)(プライムなし変数, 時間非依存変数, 定数記号を用いた (非論理) 式)
$p, p', \cdots$	**状態述語** (state predicate)(プライムなし変数, 時間非依存変数, 定数記号を用いた論理式)
$A, \cdots$	**作用** (action)(プライムなし, プライムつき変数, 時間非依存変数, 定数記号を用いた論理式)

基本式の上に **TLA 式 (時間述語)**(TLA formula, temporal predicate) をつぎのように定義する (これは stuttering 不変のために通常の時間論理式に比べ著しく制限したものになっている).

$$F, G ::= p \mid \mathrm{Enabled}\, A \mid \Box[A]_f \mid F \wedge G \mid \neg F \mid \Box F \mid \exists m : F \mid \exists x : F$$

便宜上つぎの省略形を定義しておく.

$F \vee G \triangleq \neg(\neg F \wedge \neg G)$
$F \Rightarrow G \triangleq \neg(F \wedge \neg G)$
$\mathrm{Unchanged} f \triangleq f' = f$
$\Diamond F \triangleq \neg\Box\neg F$
$[A]_f \triangleq A \vee (f' = f)$
$\langle A\rangle_f \triangleq A \wedge (f' \neq f)$
$\mathrm{WF}_f(A) \triangleq \Box\Diamond\neg(\mathrm{Enabled}\, A) \vee \Box\Diamond\langle A\rangle_f$

$\mathrm{SF}_f(A) \triangleq$
$\Diamond\Box\neg(\mathrm{Enabled}A) \vee \Box\Diamond\langle A\rangle_f$

4.2 意味

TLA 式の意味を以下に定義する． 順順に定義していく．

$d, \cdots$	**値** 例 -1, 22, $''abc''$, $\{1..12\}, \cdots$
$\xi \cdots$	**時間非依存変数解釈** (時間非依存変数と定数記号から値への関数)
$s, t, \cdots$	**状態** (変数から値への関数)
$\sigma, \tau, \cdots$	**挙動** (状態の無限列)

以上をもとに，状態の対に対する基本式の意味をつぎのように定義する．

$[\![c]\!](s, l) \triangleq \xi(c)$
$[\![m]\!](s, t) \triangleq \xi(m)$
$[\![x]\!](s, t) \triangleq s(x)$
$[\![x']\!](s, t) \triangleq t(x)$
$[\![f]\!](s, t) \triangleq f(\forall v : [\![v]\!]s/v)$
$[\![f']\!](s, t) \triangleq f(\forall v : [\![v]\!]t/v)$
$[\![p]\!](s, t) \triangleq p(\forall v : [\![v]\!]s/v)$
$[\![A]\!](s, t) \triangleq A(\forall v : [\![v]\!]s/v, [\![v]\!]t/v')$
$[\![\mathrm{Enabled}A]\!](s, t) \triangleq \exists u : [\![A]\!](s, u)$

$(\forall v : \cdots/v, \cdots/v')$ はすべての変数 v を $\cdots$ で置換したものを表す．
つぎに挙動に対する TLA 式の意味をつぎのように定義する．

$[\![p]\!]\sigma \triangleq [\![p]\!](\sigma_0, \sigma_1)$
$[\![\mathrm{Enabled}A]\!]\sigma \triangleq [\![\mathrm{Enabled}A]\!](\sigma_0, \sigma_1)$
$[\![F \wedge G]\!]\sigma \triangleq [\![F]\!]\sigma \wedge [\![F]\!]\sigma$
$[\![\neg F]\!]\sigma \triangleq \neg[\![F]\!]\sigma$
$[\![\Box F]\!]\sigma \triangleq \forall n \in \mathbb{N} : [\![F]\!]\sigma^n$
$[\![\exists m : F]\!]\sigma \triangleq \exists c \in Val : [\![F]\!]\sigma$
$[\![\exists x : F]\!]\sigma \triangleq \exists \rho, \tau : (\natural\sigma = \natural\rho) \wedge (\rho =_x \tau)$
$\wedge [\![F]\!]\tau$

以上の定義で σ_i は挙動 σ の i(0 から始まる) 番目状態を指し，σ^n は σ の σ_n に始まる無限列 $(\sigma_n, \sigma_{n+1}, \cdots)$ を指す．$\natural, =_x$ はつぎによって定義するが，$\natural\sigma$ は挙動 σ のなかで連続した同一状態有限列を単一状態に置換した挙動，ただし，前半の有限列を除いて全部同一の状態列はそのままとした挙動のことであり，$\sigma =_x \tau$ は挙動 σ と τ の状態対 $(\sigma_i, \tau_i)(\forall i \in \mathbb{N})$ が x に対する値を除いて同一であることを主張するものである．

$$\natural\sigma \triangleq \begin{cases} \sigma & \text{if } \forall n \in \mathbb{N} : \sigma_n = \mathrm{hd}\sigma \\ \natural(\mathrm{tl}\sigma) & \text{if } \mathrm{hd}\sigma = \mathrm{hd}(\mathrm{tl}\sigma) \\ \mathrm{hd}\sigma \circ \natural(\mathrm{tl}\sigma) & \text{otherwise} \end{cases}$$

$\sigma =_x \tau \triangleq \forall n \in \mathbb{N} : \forall v \neq x : [\![v]\!]\sigma_n = [\![v]\!]\tau_n$
つぎに式とその意味について若干補足する．
状態述語 p がある挙動に対して真であるとはその挙動の最初の状態に対して真であるときまたそのときである．
任意の作用 A に対して，述語 Enabled$A$ を，それがある状態で真であるのはその状態からはじまる A 段階が存在するときであると定義する．例 $\mathrm{Enabled}(y = x' + n) \equiv \exists c : y = c + n$
演算子 $\Box$ は時間論理の「常に」(いつも always, 今後ずっと henceforth) 演算子である．すなわち，$\Box F$ は F が未来に対して常に真であることを意味する．具体的には，挙動 σ が $\Box F$ を満たすとは，すべての $n \in \mathbb{N}$ に対して σ^n が F を満たすことである，つまり，σ のすべての状態が F を満たすのである．
演算子 $\Diamond$ は時間論理の「いつか，やがて」(sometime, eventually) 演算子である．$\Diamond F$ はやがていつか F が真になるときがくることを意味する．具体的には，ある $n \in \mathbb{N}$ が存在して σ^n が F を満たす，つまり，n 時点では F を満たすというものである．
$\Box[A]_f$ は未来において隣り合ったすべての状態の対が作用 A かあるいは $f' = f$ を満たすことを意味する．これは作用 A のうちに状態関数 f を変えない部分段階の存在を許すというもので，仕様の

モデルは洗練された仕様のモデルでもなければならないという要請に基づくものである.
式 $\exists x.P$ は P が成立する x の値を選択できる方法があることを意味し x を隠蔽する目的をもつ.

5 安全性と活性

仕様作成において安全性と活性 (応答性) の記述が一番大切である. **安全性** (safety) とは**悪いことは決して起こらない**という性質であり, $\Box F$ の形の式で表現できる. これに対して**活性** (liveness) とは**良いことはやがて起こる**という性質であり, $\Box\Diamond F$ あるいは $\Diamond\Box F$ の形の式で表現できる.
安全性については, すでに例 1,2 の $\Box Next$ の例でみたように, $\Box F$ は挙動のすべての状態 (の組) に対して F が真となるわけであるから, 挙動のいつの時点でも必ず F が成立する, つまり, F 以外は決して起こらないことを示すものである.
仕様として安全性だけを書いたのでは不足である. なにもしない消極的な系も仕様を満足するからである. したがって活性を明示的に書かなければならない. ところで活性の方は時間演算子が 2 個付いているからすこしややこしい.
式 $\Box\Diamond F$ がある挙動に対して真であるのは, その挙動のすべての状態 (時点 n) において $\Diamond F$ が真であるときであり, そして, 時点 n において $\Diamond F$ が真であるのは ある時点 $m \geq n$ にて F が真であるときである. すなわち,

$$[\![\Box\Diamond F]\!]\sigma \equiv \forall n \in \mathbb{N} : \exists m \in \mathbb{N} : [\![F]\!]\sigma^{n+m}$$

$\forall n : \exists m : p(n+m)$ の形の式は $p(i)$ が無限に多くの i の値に対して真となることを主張している. したがって, 挙動が $\Box\Diamond F$ を満たすとはその挙動の中で無限回 F が真となることである. このことを F が**無限回しばしば (infinitely often)** 真になるなどという.
式 $\Diamond\Box F$ はやがて, つまり, 未来のある時点以降は常に F が真になることを主張している. 形式的には

$$[\![\Diamond\Box F]\!]\sigma \equiv \exists m \in \mathbb{N} : \forall n \in \mathbb{N} : [\![F]\!]\sigma^{n+m}$$

換言すれば, 挙動が $\Diamond\Box F$ を満たすとはある時点が存在してその時点以降は永続して F が真になることである. このことを F が**ほとんど常に (almost always)** 真になるなどという.
さて, 作用 A についての**弱公平性** (weak fairness) をつぎのように定義する.

$$\begin{aligned} \mathrm{WF}_f(A) &\equiv \Box(\Box\mathrm{Enabled}\langle A\rangle_f \Rightarrow \Diamond\langle A\rangle_f) \\ &\equiv \Box\Diamond\neg\mathrm{Enabled}\langle A\rangle_f \vee \Box\Diamond\langle A\rangle_f \\ &\equiv \Diamond\Box\mathrm{Enabled}\langle A\rangle_f \Rightarrow \Box\Diamond\langle A\rangle_f \end{aligned}$$

これらは, それぞれ日語でつぎのようにいえるだろう.
「つぎが常に成り立つ : A の実行可能が継続すれば A はやがて実行される.」
「A が無限回しばしば実行不可能であるか, あるいは, A が無限回しばしば実行される.」
「A が未来のある時点以降実行可能が継続すれば A が無限回しばしば実行される.」
また, 作用 A についての**強公平性** (strong fairness) をつぎのように定義する.

$$\begin{aligned} \mathrm{SF}_f(A) &\equiv \Diamond\Box\neg\mathrm{Enabled}\langle A\rangle_f \vee \Box\Diamond\langle A\rangle_f \\ &\equiv \Box\Diamond\mathrm{Enabled}\langle A\rangle_f \Rightarrow \Box\Diamond\langle A\rangle_f \end{aligned}$$

「A がほとんど常に実行不能になるか, あるいは, A が無限回しばしば実行される.」
「A が無限回しばしば実行可能になれば A が無限回しばしば実行される.」
以上の定義において, 作用が $\langle A\rangle_f$ の形であることに注意したい.
時間演算子を伴った式の扱いのためにつぎの成立を注意する.
$\Box\Diamond\neg F \equiv \Box\neg\Box F \equiv \neg\Diamond\Box F$
$\neg\Box F \equiv \Diamond\neg F$
$\Diamond(F \vee G) \equiv \Diamond F \vee \Diamond G$
$\Box\Diamond(F \vee G) \equiv \Box\Diamond F \vee \Box\Diamond G$
例 1,2 で仕様と称したものはこの活性を欠いていた. ともに $\mathrm{WF}_x(Next)$ を連言させなければならない.

6 stuttering 不変性

TLA では作用と時間演算子を伴う TLA 式は $[A]_f$, あるいは, $\langle A \rangle_f$ の形に限られている. これは stuttering 不変性を考慮した結果である. 挙動の隣り合った２状態が同一のとき, その２状態を stuttering 段階という. TLA 式の真偽はそれを定める挙動のなかで同じ状態の有限個の反復を追加あるいは除去しても変わらない. この性質を stuttering 不変であるという. この不変性はつぎのような事情によるものである.

プログラムを扱う伝統的な時間論理は次状態演算子 ○ を使った. プログラム

$Prog1 \quad :: x := x + 1$

は

$\Box(x = 0 \Rightarrow \bigcirc(x = 1)) \cdots (*)$

を満たす. ところで,

$Prog2 \quad :: \textbf{local}\ t;$
$t := 1\ ;\ x := x + t$

は $(*)$ を満たさない. $Prog2$ を $Prog1$ の洗練と考え $(*)$ を $Prog1$ の仕様とみなすとき, $Prog2$ が $(*)$ を満たさないのは具合が悪い.

また,

$Prog3 \quad :: y := y - 1$

は

$\Box(y = 1 \Rightarrow \bigcirc(y = 0)) \cdots (\dagger)$

を満たす. つぎの並行合成を考える.

$Prog4 \quad :: Prog1 \parallel Prog3$

この場合 $Prog4$ が $(*) \wedge (\dagger)$ を満たすことを期待したい. しかし, 実際は $(*)$ も $(\dagger)$ も満たさない. つまり, プログラム成分の仕様が並行合成の仕様の一部にならない. このような状況は実際の仕様作成に際しては好ましくない. 時間論理式が stuttering 不変であると以上の状況が好転する. 時間論理式の stuttering 不変性が望まれた所以である.

7 TLA による仕様

TLA による仕様記述の一例を挙げる. これは環境と系の作用の同時並行実行を許さない系またはその成分 (component) の差込方式による記述例である. 系は１個の入力（環境）変数 e, １個の出力変数 c, １個の内部変数 x をもつとする. この系の仕様はつぎの形式で記述できる:

$$\exists x.\ (Init \wedge \Box[Next]_{c,x} \wedge Live)$$

ただし：

$Init$ は c と x の初期値を表す状態述語.

$Next$ は系の段階を記述する作用. この系の場合入力変数を不変に保つから $Next$ は $e' = e$ を含意する.

c, x は $Next$ の添え字に現れる. というのは $\Box[Next]_{c,x}$ が c と x とを不変に保つ状態変化を許容するからである.

$Live$ はそれぞれが $\mathrm{WF}_{c,x}(A)$ か $\mathrm{SF}_{c,x}(A)$ の形の公平性条件の連言である. 一般に $Next$ は複数の作用の選言になる.

この方式による仕様作成はほぼつぎのような段階を追うことになるだろう.

(0) 系を記述する部分と抽象度を選べ.

(1) 系のもつ変数を選び, その型と初期値を書け. 定数引数やその仮定が定まる.

(2) 次状態関係を書け (作用の例を挙げてみよ).

(3) 仕様の時間に関する部分を書け. 安全性と活性 (公平性) を加味せよ. 仕様は初期述語, 次状態作用, 公平性の連言になる.

(4) 仕様から導出できる定理を述べよ.

注意　系の機能については最初は無視し, あとで次状態作用に選言として追加せよ.

8 非同期交信界面

早速仕様記述に応用してみよう．つぎは非同期データ送受信界面（2 phase handshake protocol)の仕様例である．データを val 上で送り，rdy と ack 回線を同期のために使う．送信側はつぎのデータを送信する前に受信通知を受信しなければならない．

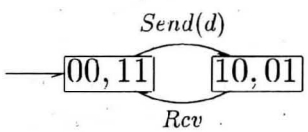
状況を図示すれば

$$Sender \xrightarrow{c} Receiver$$

交信状況はつぎのようなものである．

$$\left(\begin{array}{lcl} v & \mapsto & d0 \\ rdy & \mapsto & 0 \\ ack & \mapsto & 0 \end{array}\right) \xrightarrow{Send(d1)} \left(\begin{array}{lcl} v & \mapsto & d1 \\ rdy & \mapsto & 1 \\ ack & \mapsto & 0 \end{array}\right) \xrightarrow{Ack} \left(\begin{array}{lcl} v & \mapsto & d1 \\ rdy & \mapsto & 1 \\ ack & \mapsto & 1 \end{array}\right)$$

さらにこの界面はつぎのような推移図として説明できる．



つぎが TLA による記述例である．

module *Chan*

CONSTANTS D, $Two \triangleq \{0,1\}$

VARIABLE c

$TInv \triangleq c \in [v : D,\ rdy : Two,\ ack : Two]$

$Init \triangleq TInv \wedge c.rdy = c.ack$

$Send(d) \triangleq c.rdy = c.ack$

$\quad \wedge c' = [c \text{ EXCEPT } !.v = d,\ !.rdy = 1 - @]$

$Rcv \triangleq c.rdy \neq c.ack$

$\quad \wedge c' = [c \text{ EXCEPT } !.ack = 1 - @]$

$Next \triangleq (\exists d \in D : Send(d)) \vee Rcv$

$Spec \triangleq Init \wedge \Box[Next]_c \wedge \mathrm{WF}_c(Rcv)$

THEOREM $Spec \Rightarrow \Box TInv$

記述中の $[\,r$ EXCEPT $!.c_1 = e_1, \cdots, !.c_n = e_n\,]$ は r においてそれぞれの i に対して $r.c_i$ を e_i で置き換えてえられたレコードを表示する．また，式 c_i に出現する@は $r.c_i$ を意味する．

9 炭坑排水制御系

例題として炭坑の浸水に対して排水ポンプを作動させる制御プログラムの作成を考える [1][3][4]．浸水してもメタン濃度が高ければポンプを運転することはできない．

系はつぎの構成成分をもっている．

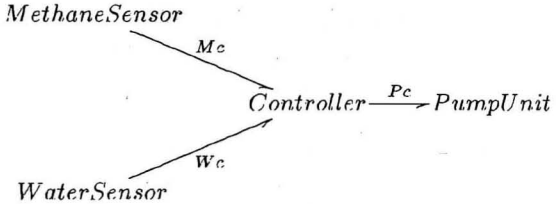
ポンプは制御プログラムからポンプの「起動」と「停止」の命令を受ける．

メタン感知器は制御プログラムへ坑内メタン濃度が「安全」か「危険」を知らせる．

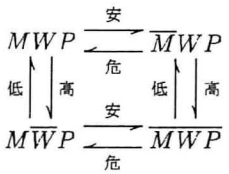
水位感知器は制御プログラムへ坑内水位が「高い」か「低い」を知らせる．そこでプログラムはつぎを要請する．

(1) メタン濃度安全かつ水位高ならばポンプを作動
(2) メタン濃度安全かつ水位低ならばポンプを停止
(3) メタン濃度危険かつ水位低ならばポンプを停止
(4) メタン濃度危険かつ水位高ならばポンプを停止

状況はじ次図のとおりである．



この系を推移図として表せば



図の中の文字はつぎのような意味をもっている：

$M \equiv Mcrit$ $MWP \equiv M \wedge W \wedge P$

$W \equiv Wlow$ $M\overline{W}P \equiv M \wedge \neg W \wedge P$

$P \equiv Poff$

つぎが TLA による記述例である.

module *MinePumpController*

CONSTANTS $M = \{$安,危$\}$
$W = \{$低,高$\}$
$P = \{$入,断$\}$

VARIABLE m, w, p, s

$Mc \triangleq$ **instance** $Chan$ **with** $D \leftarrow M, c \leftarrow m$

$Wc \triangleq$ **instance** $Chan$ **with** $D \leftarrow W, c \leftarrow w$

$Mc \triangleq$ **instance** $Chan$ **with** $D \leftarrow P, c \leftarrow p$

$TInv1 \triangleq s \in [ms : M,\ ws : W,\ ps : P]$

$Mcrit \triangleq s.ms =$危

$Wlow \triangleq s.ws =$低

$Poff \triangleq s.ps =$断

$Tinv \triangleq Mc!TInv \wedge Wc!TInd \wedge Pc!Inv \wedge TInv1$

$Init \triangleq Tinv \wedge Mc!Init \wedge Wc!Init \wedge Pc!Init \wedge$
$Mcrit \wedge Wlow \wedge Poff$

$MSend(d) \triangleq Mc!Send(d) \wedge Mc!Rcv \wedge$
UNCHANGED $\langle wc \rangle \wedge$
case $Mcrit \wedge Poff \wedge d =$安 $\rightarrow$
$Pc!Send($入$) \wedge$
$s' = [s$ EXCEPT $!.ms = d, !.ps =$入$]$
$\neg Poff \wedge d =$危 $\rightarrow Pc!Send($断$) \wedge$
$s' = [s$ EXCEPT $!.ms = d, !.ps =$断$]$
other $\rightarrow s' = [s$ EXCEPT $!.ms = d,] \wedge$
UNCHANGED $\langle pc \rangle$

$WSend(d) \triangleq Wc!Send(d) \wedge Wc!Rcv \wedge$
UNCHANGED $\langle mc \rangle \wedge$
case $\neg Poff \wedge d =$低 $\rightarrow Pc!Send($断$) \wedge$
$s' = [s$ EXCEPT $!.ws = d, !.ps =$断$]$
$\neg Poff \wedge d =$低 $\rightarrow$
$Poff \wedge \neg Mcrit \wedge d =$高 $\rightarrow Pc!Send($入$) \wedge$
$s' = [s$ EXCEPT $!.ws = d, !.ps =$入$]$
other $\rightarrow s' = [s$ EXCEPT $!.ws = d] \wedge$
UNCHANGED $\langle pc \rangle$

$PRcv \triangleq Pc!Rcv$

$Next \triangleq (\exists d \in M : MSend(d)) \vee$
$(\exists d \in W : WSend(d)) \vee PRcv$

$Spec \triangleq Init \wedge \Box[Next]_{\langle mc,wc,pc,s \rangle} \wedge$
$\mathrm{WF}_s(PRcv)$

まとめ

TLA を応答系の仕様記述に利用することを紹介したが，TLA 自身の説明も不充分である．形式的な証明については完全に省いた．実際に利用するために必要な仕様の洗練，実現，また，分解，合成についても一切説明していない．実時間系 (real-time system) や混合系 (hybrid system) のための機構についても同様である．しかし，TLA は組込系 (embedded system) などで是非採用・実用化してみたいと思う．

設計パターンやコンポーネントなどについてはこの枠組で厳密な議論ができると考えている．

参考文献

[1] I. J. Hayes and B. Mahony, A case study in timed refinement: A mine pump. IEEE TSE, 1992.

[2] L. Lamport, The temporal logic of actions. ACM TOPLAS,1994.

[3] K. Lano, Formal Object-Oriented Development. Springer, 1995.

[4] ibid., The B Language and Method. Springer, 1996.

[5] 田辺誠他，コンピュータサイエンス入門　論理とプログラム意味論．岩波，1999.

ソフトウェア技術者協会
〒160-0004　東京都 新宿区 四谷 3 - 12 丸正ビル 5F
Tel: 03 - 3356 - 1077　Fax: 03 - 3356 - 1072
E-mail: sea@sea.or.jp
URL: http://www.iijnet.or.jp/sea