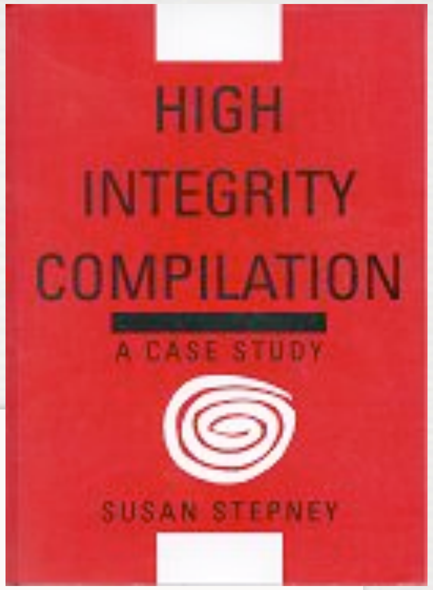
(TOWARDS) DEMONSTRABLY CORRECT COMPILATION OF JAVA BYTECODE

Michael Leuschel
University of Düsseldorf

FMCO 2008
Nice Sophia-Antipolis

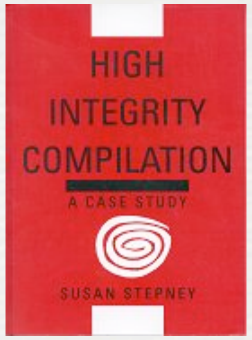
PART 1: BACKGROUND

BACKGROUND



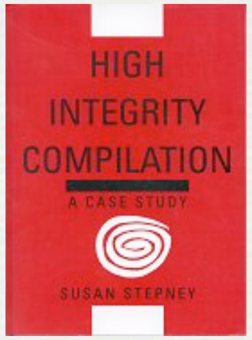
- DeCCo (Demonstrably Correct Compiler)
  - By Susan Stepney, Logica + AWE [1992-2001]
  - From: PASP
    - Pascal-like language
  - To: ASP
    - Custom RISC Processor
- One major step for Hoare's Grand Challenge

DECCO: PROCESS

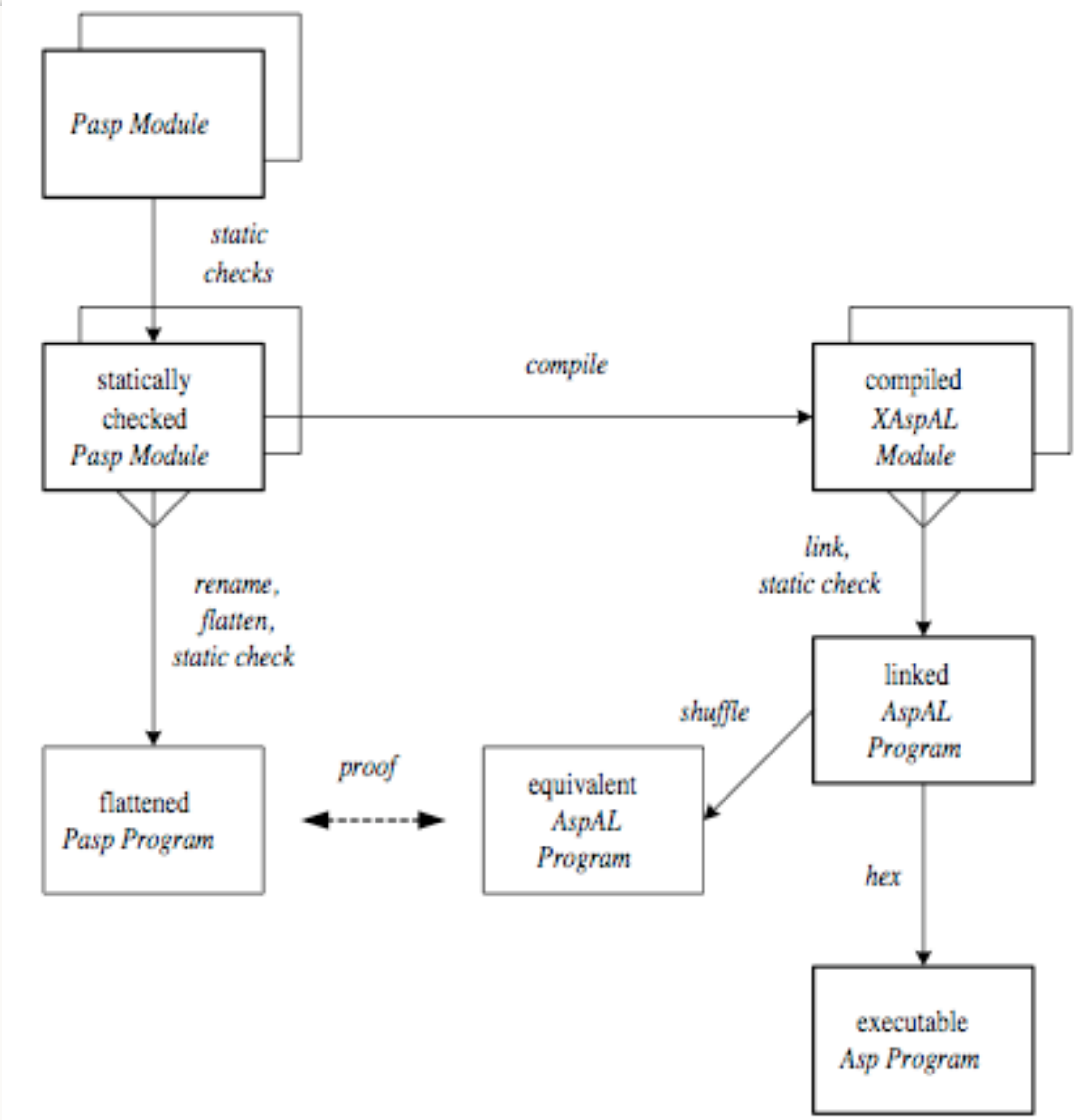


- Z Specification of PASP
- Z Specification of ASP
  (+ ASPAL + XASPAL)
- Translation Rules in Z:
  - PASP → ASP
    - Proven **by hand**
    - Translated **by hand** into Prolog DCGT

DECCO: PROCESS



- Z Specification of PASP
- Z Specification of ASP
  (+ ASPAL + XASPAL)
- Translation Rules in Z:
  - PASP → ASP
    - Proven **by hand**
    - Translated **by hand** into Prolog DCGT



“We believe that the methodology provides us with a high level of confidence in the correctness of the embedded software required to drive high integrity controllers.”

Source: Decco Website
http://www-users.cs.york.ac.uk/~susan/bib/ss/hic.htm

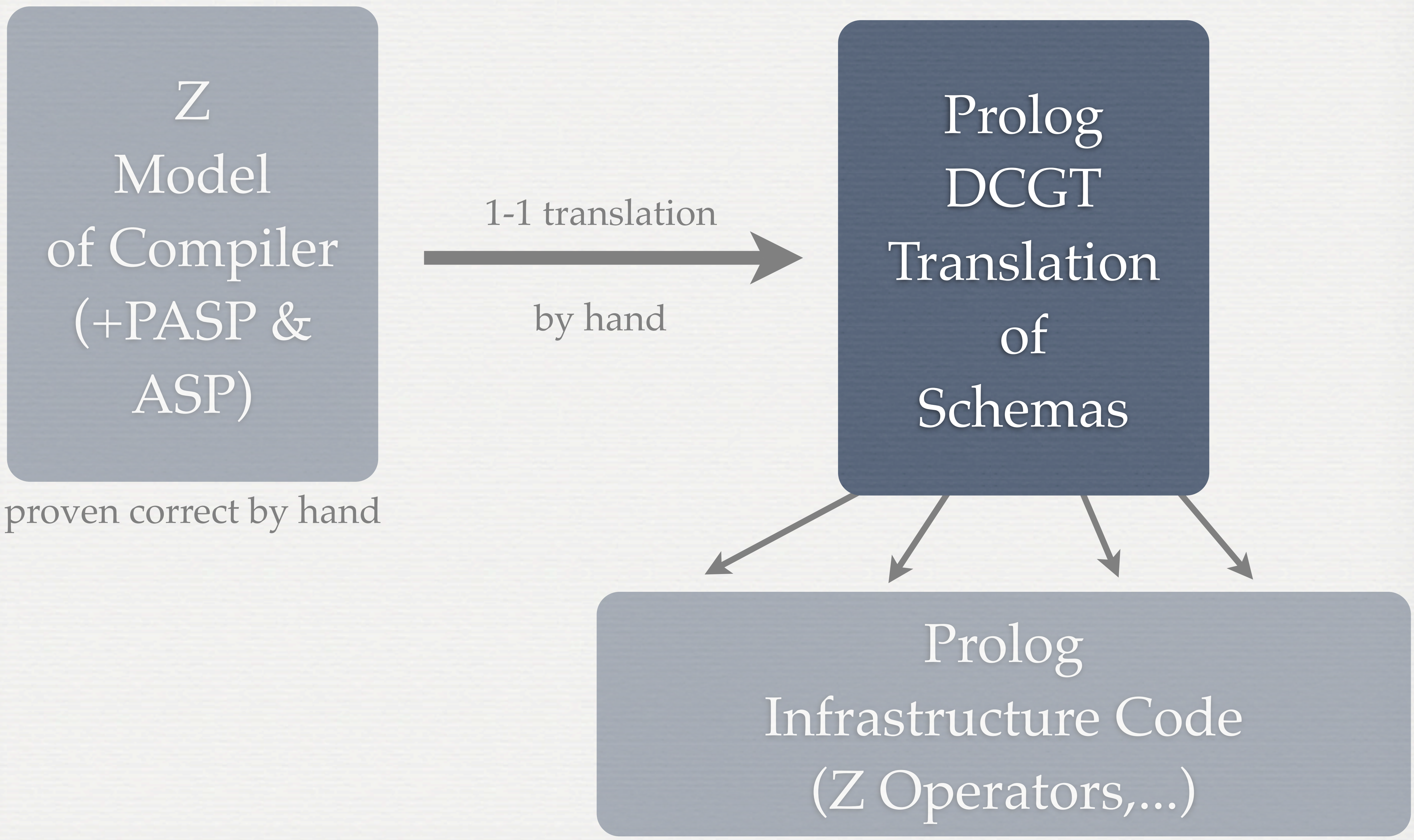
DRAWBACKS

- tied to PASP, difficult to get PASP programmers
- proven by hand + translation by hand
- translation Z → Prolog only correct under certain assumptions
- Prolog code was hard to maintain, performance issues, a few bugs

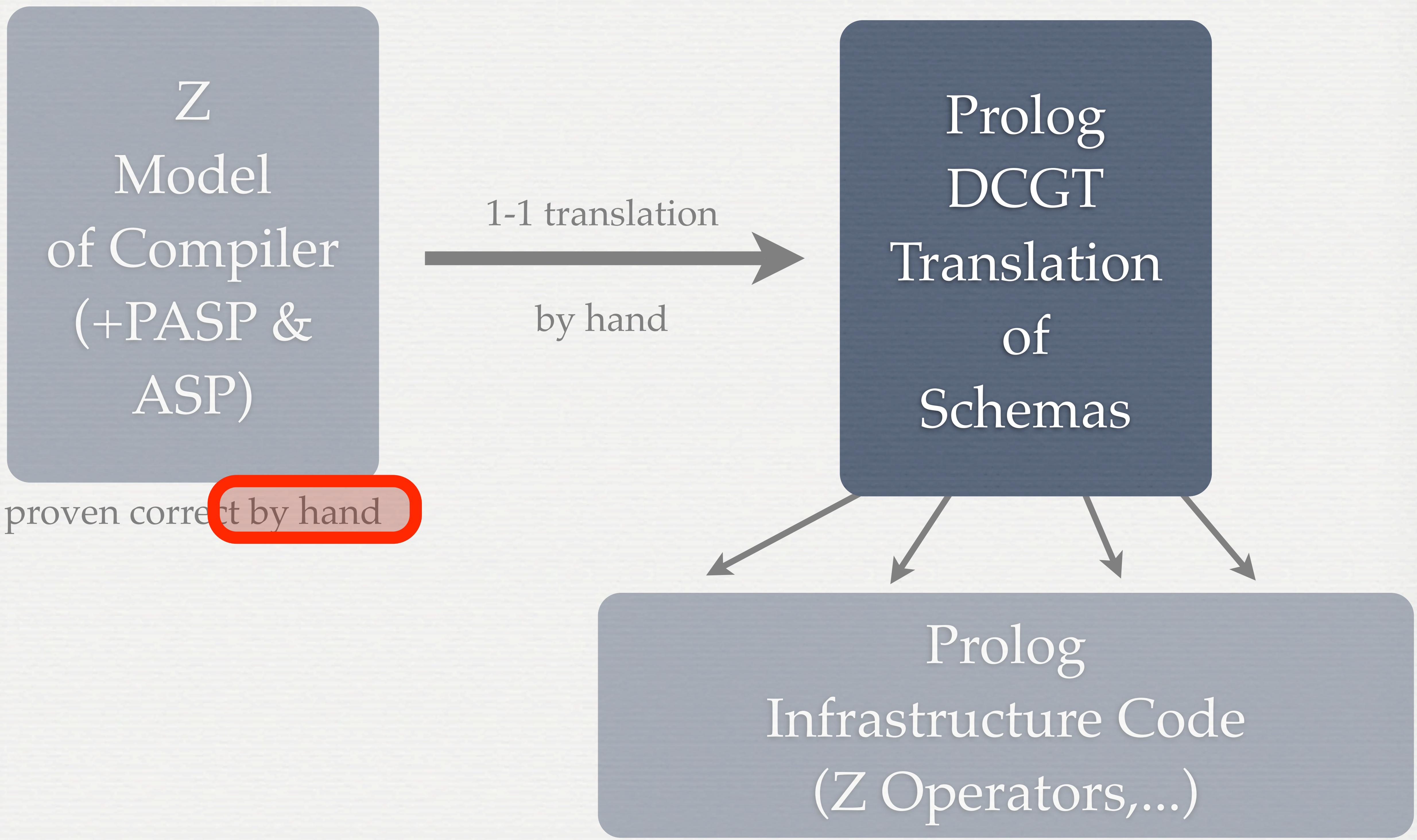
JASP PROJECT

- Investigate existing Decco system
- Move from PASP to Java Bytecode
- Provide recommendations for future developments
  - Adapt existing Decco System for JavaBC ?
    - Move from Prolog to Haskell ?
  - Investigate other alternatives, ...

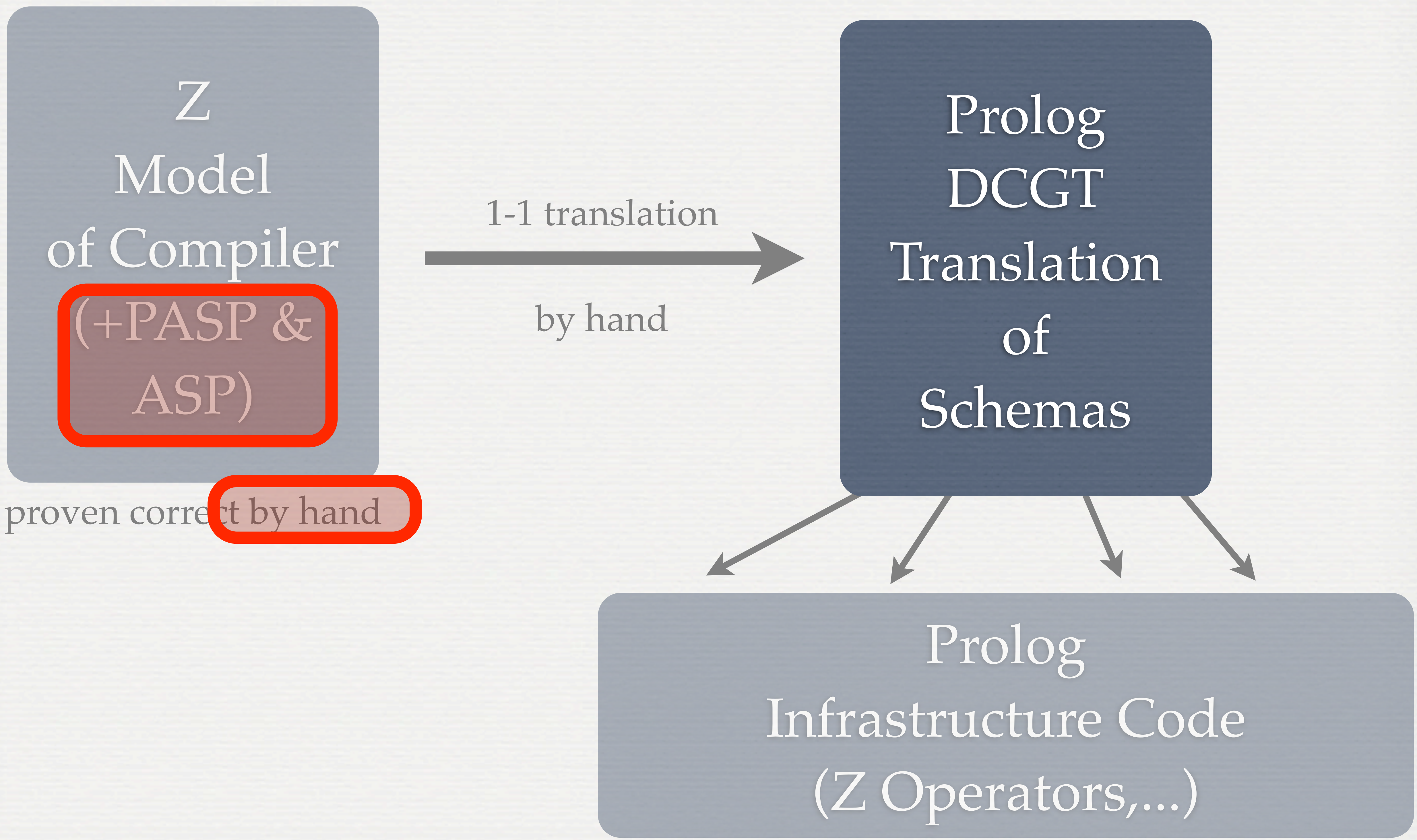
DECCO COMPILER



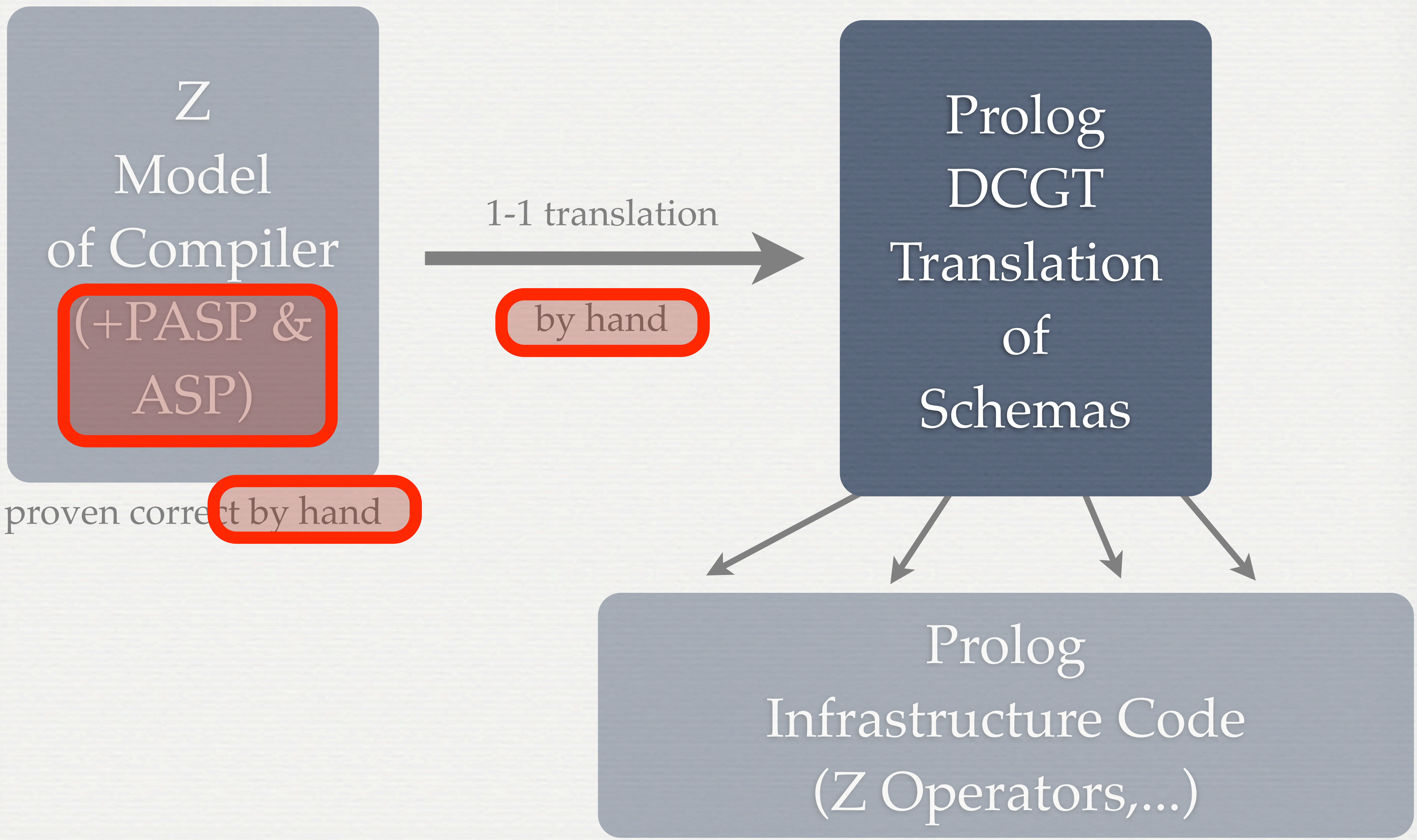
DECCO COMPILER



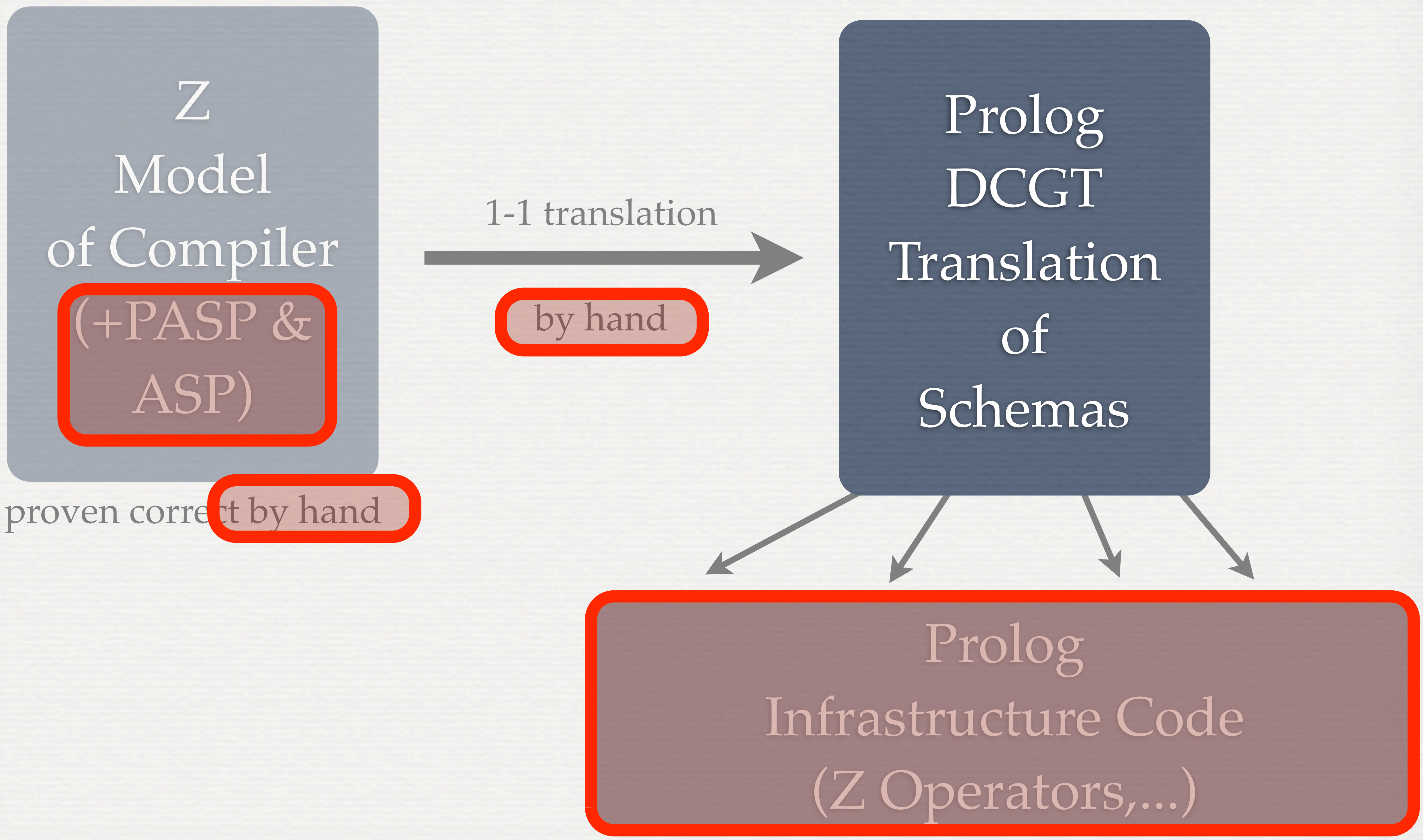
DECCO COMPILER



DECCO COMPILER



DECCO COMPILER



THE PROLOG SYSTEM

- LPA WinProlog:
  - only on Windows, no modules
  - idiosyncratic features were used
  - default mode gives no warnings
    - singleton variables, predicate redefinition, ...

JASP CONCLUSIONS

- Try to automate more of compiler construction
- Move from Z to B (or other approaches)
  - Automatic Code generation
  - Formal proofs
  - Tool support

PART 2:
A LITTLE BACKGROUND ABOUT B

A quick overview of B

B-Method

- Invented by Abrial
- Successor of Z
- Allows to write high-level specifications & code (B0)
- Aimed at tool support

B: Logical Predicates

logic	ASCII	meaning
$P \vee Q$	`P or Q`	or
$P \wedge Q$	`P & Q`	and
$\neg P$	`not(P)`	negation
$P \Rightarrow Q$	`P => Q`	implication
$\forall x : T \bullet P$	`!x.(x:T => P)`	for all
$\exists x \bullet P$	`# x.P`	there exists

B: SETS

Sets	ASCII	meaning
$S \cup T$	`S \/ T`	union
$S \cap T$	`S /\T`	intersection
$e \in S$	`e:S`	member of
$e \notin S$	`e/:S`	not member of
$S \subseteq T$	`S<:T`	subset
$S \setminus T$	`S - T`	set subtraction
$\mathbb{P}\, S$	`POW(S)`	power set
$\mid S \mid$	`card(S)`	size
$\mathbb{N}$	`NAT`	naturals
$\mathbb{N}_1$	`NAT1`	positive numbers

{x,y,... | P} set comprehensions

(partial list)

B: Relations

Relations	ASCII	meaning
$x \mapsto y$	`x |-> y`	x maps to y
$dom(R)$	`dom(R)`	domain of R
$ran(R)$	`ran(R)`	range of R
$U \lhd R$	`U <| R`	domain restriction
$U \unlhd R$	`U <<| R`	domain anti-restriction
$R \rhd U$	`R |> U`	range restriction
$R \unrhd U$	`R |>> U`	range anti-restriction
$R(\| U \|)$	`R[U]`	relational image
R^{-1}	`R~`	relational inverse
$R0 \fatsemi R1$	`R0 ; R1`	relational composition

(partial list)

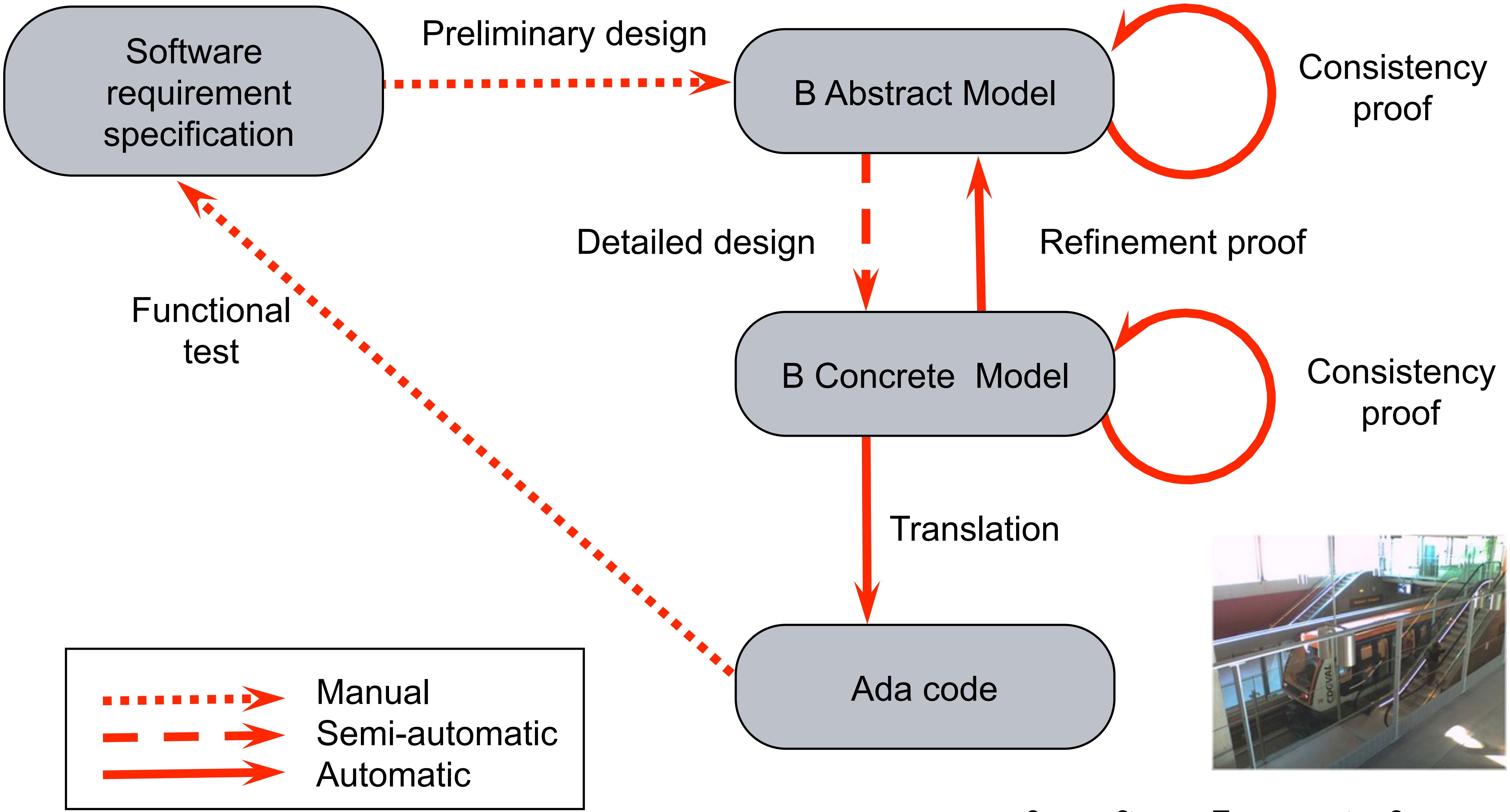
B: Functions

Function	ASCII	meaning
$S \nrightarrow T$	`S +-> T`	partial function
$S \rightarrow T$	`S --> T`	total function
$S ⤔ T$	`S >+> T`	partial injection
$S \rightarrowtail T$	`S >-> T`	total injection
$S ⤀ T$	`S +->> T`	partial surjection
$S \twoheadrightarrow T$	`S -->> T`	total surjection
$S ⤖ T$	`S >->> T`	(total) bijection

f(x) function application

%(x,y,..).(P|E) lambda abstraction

B Development Process



Source: Siemens Transportation Systems

Utilisation de la méthode B développée par

CLEARSY
SYSTEM ENGINEERING

ETHZ

Southampton

Newcastle

Aabo

Nokia

Bosch

Siemens Transportation

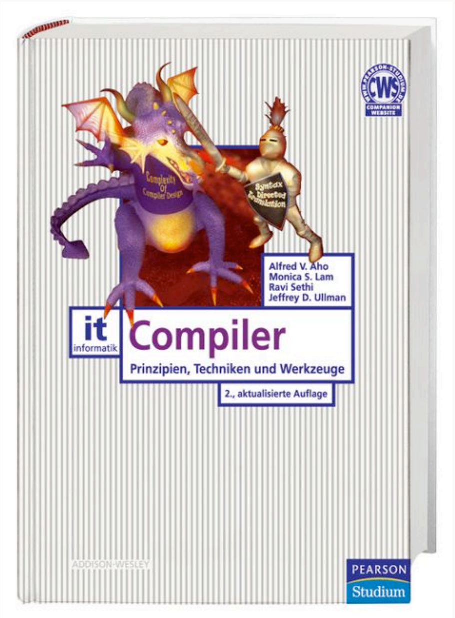
SAP

Space Systems Finland

PART 3: COMPILER CONSTRUCTION WITH B

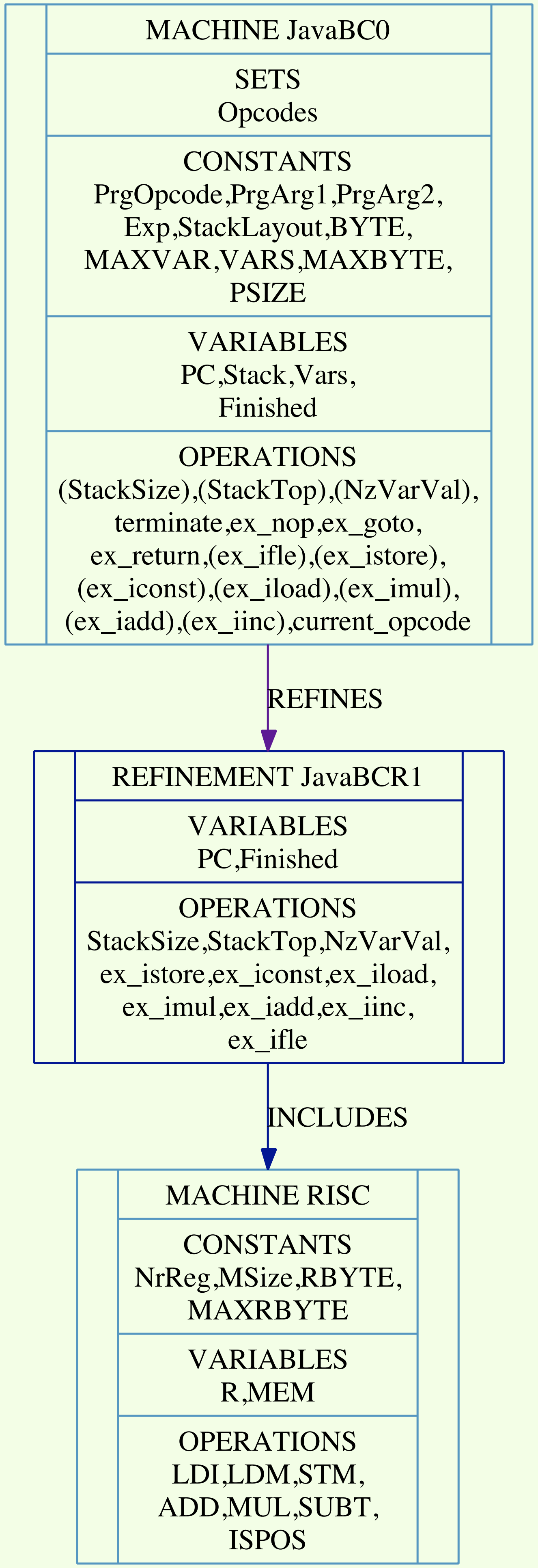
JASP: FIRST EXPERIMENT

- Small Subset of Java Bytecode
  - no methods, objects, ...
  - istore, iload, iconst, imul, ...
- Simple model of the processor
  - Three-Address code of Dragon Book
  - LDI, LDM, STM, MUL, ...



IDEA

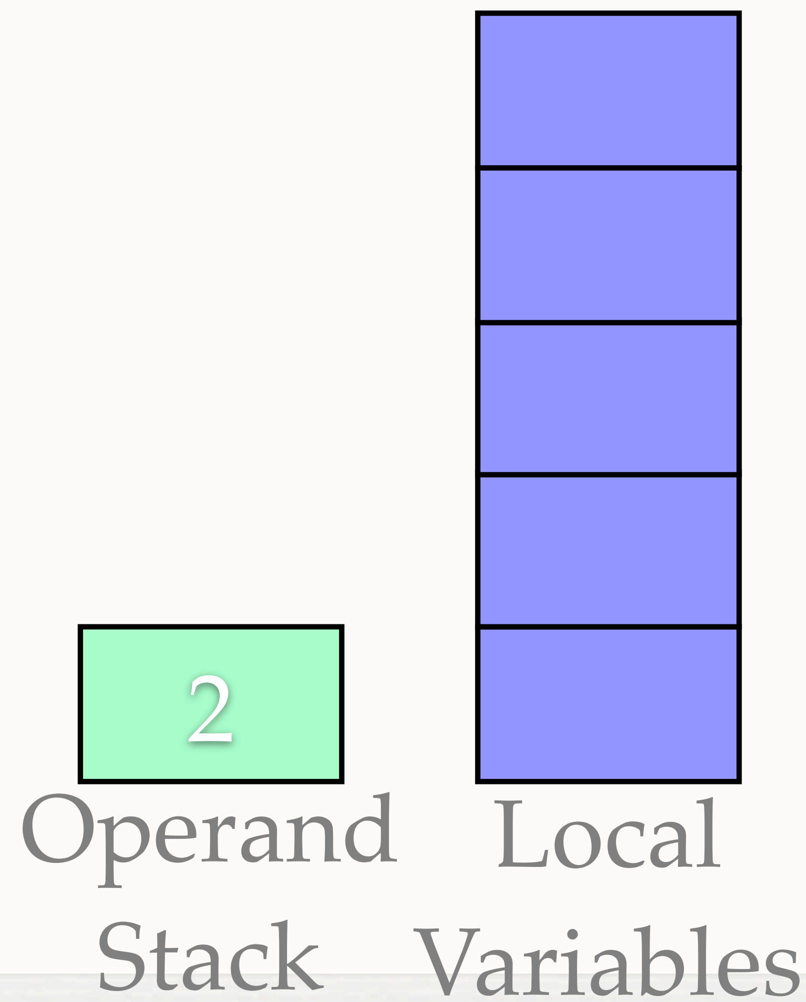
- Model JavaBC as B
- Model RISC as B
- Refine JavaBC into compiled version
  - opcodes translated into RISC
  - correctness established by B refinement



EXAMPLE BYTECODE

```
public class Power {
public static void main(String args[])
 {
  int base = 2;
  int exp = 5;
  int i = exp;
  int res = 1;
  while (i>0) {
    i--;
    res = res*base;
  }
  System.out.println(res);
 }
}
```

```
0: iconst_2
1: istore_1
2: iconst_5
3: istore_2
4: iload_2
5: istore_3
6: iconst_1
7: istore      4
9: iload_3
10: ifle  25
13: iinc 3, -1
16: iload      4
18: iload_1
19: imul
20: istore      4
22: goto      9
25: return
```



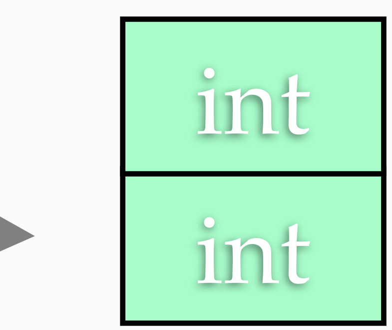
HOW TO COMPILE

- How to compile to RISC with limited memory and registers (2) ?
  - Local variables statically known: ok
  - What about the stack ??

STACK LAYOUT

```
0: iconst_2
1: istore_1
2: iconst_5
3: istore_2
4: iload_2
5: istore_3
6: iconst_1
7: istore      4
9: iload_3
10: ifle  25
13: iinc 3, -1
16: iload      4
18: iload_1
19: imul
20: istore     4
22: goto       9
25: return
```

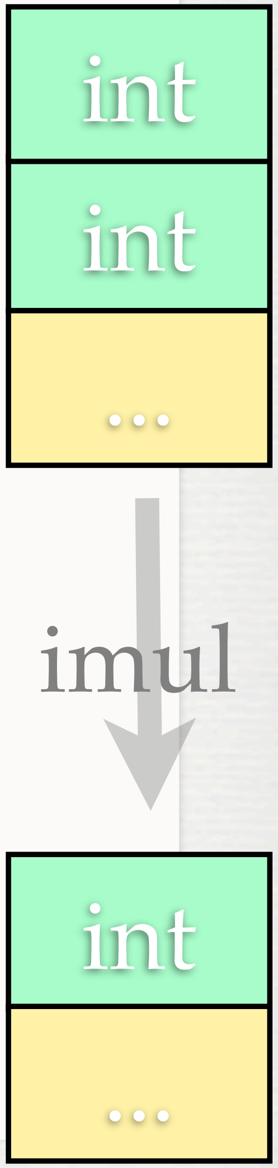
Every program point:
same stack layout,
no matter which path



Operand
Stack

HOW TO COMPILE

- Infer stack layout:
  - for every program point: size of stack
  - upper bound must exist
  - treat like local variables !
    - no need to maintain a stack pointer !!

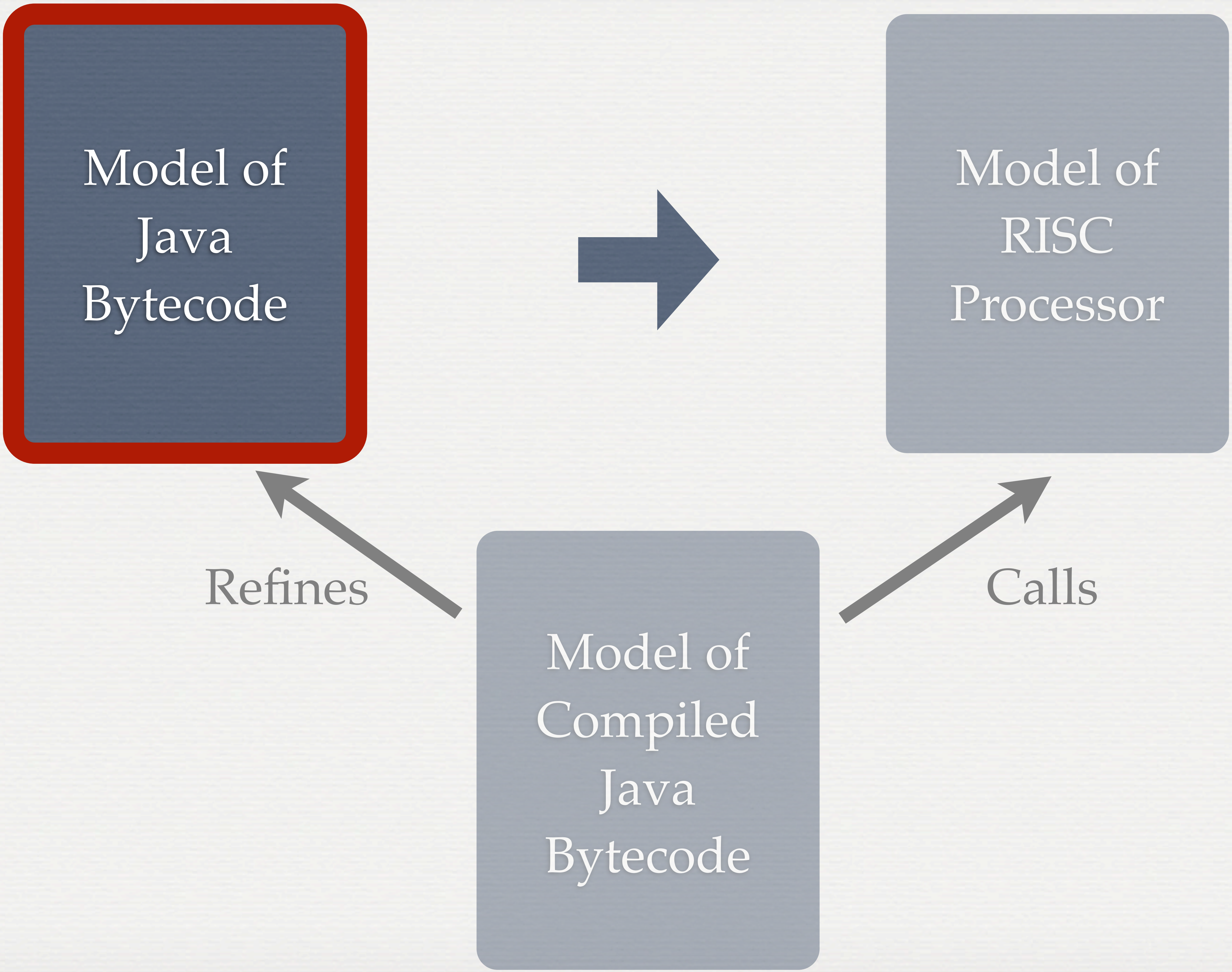


INFERRING STACK LAYOUT

- By abstract interpretation
  - Prolog interpreter for Java BC
  - run it on abstract domain of types {int, ...}
  - [Demo]
- In Java 6:
  - Stacklayout actually already in class file

TRUSTING STACKLAYOUT INFO

- Remember: we want formally verified compilation
- How can we trust the code that computed the stack layout info?
- We don't have to !
  - Build properties of correct stack layout into B formal model
  - Computed stack layout needs to be checked for those properties


Model of Java Bytecode
Model of RISC Processor
Refines
Calls
Model of Compiled Java Bytecode

THE B MODEL OF JAVABC

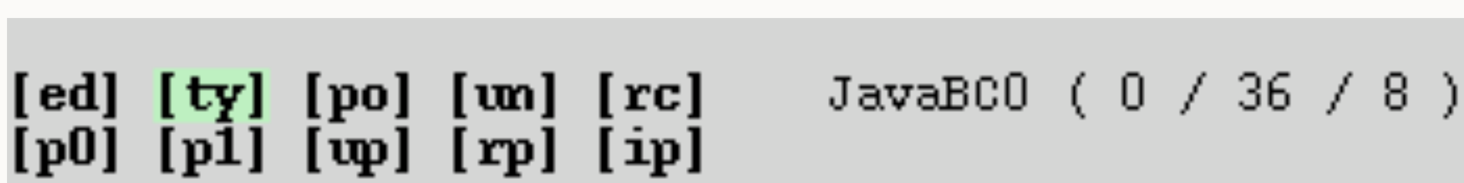
```
PROPERTIES
  PSIZE : NATURAL1 &
  PrgOpcode: 1..PSIZE --> Opcodes &
  PrgArg1: 1..PSIZE --> VARS &
  PrgArg2: 1..PSIZE --> BYTE &
...
StackLayout: 1..PSIZE --> VARS
/* for each Program Point: indicate size of stack */
   &
  StackLayout(1) = 0 & /* Initially stack is empty */
...
 !pc1.(pc1:1..PSIZE =>
       ((PrgOpcode(pc1)/=goto &
    PrgOpcode(pc1)/=return) => pc1+1 <= PSIZE ))
 &
  !pc2.(pc2:1..PSIZE & PrgOpcode(pc2) = goto
           => (PrgArg1(pc2):1..PSIZE &
                   StackLayout(PrgArg1(pc2)) = StackLayout(pc2)) )
 ...
```

```
INVARIANT
  PC: 1..PSIZE &
  Stack: seq(INTEGER) &
  Vars: VARS +->INTEGER &
  Finished: BOOL &
  size(Stack) = StackLayout(PC)
```

THE B MODEL OF JAVABC

```
OPERATIONS
   ex_iload(A1) = PRE
PrgOpcode(PC) = iload  &
A1=PrgArg1(PC) &  A1:dom(Vars)
THEN
        AdvancePC ||
        Stack := Stack <- Vars(A1)
     END;
```

- Proven correct:
  - PC remains within program bounds
  - statically computed Stack Layout is always correct if properties satisfied



Quote

"Every formal model I have seen, proven or not, which has not been animated contained errors"

Christophe Metayer, Systerel

(liberal translation from French based on verbal communication)

Quote

“Every formal model I have seen, proven or not, which has not been animated contained errors”

Christophe Metayer, Systerel

(liberal translation from French based on verbal communication)

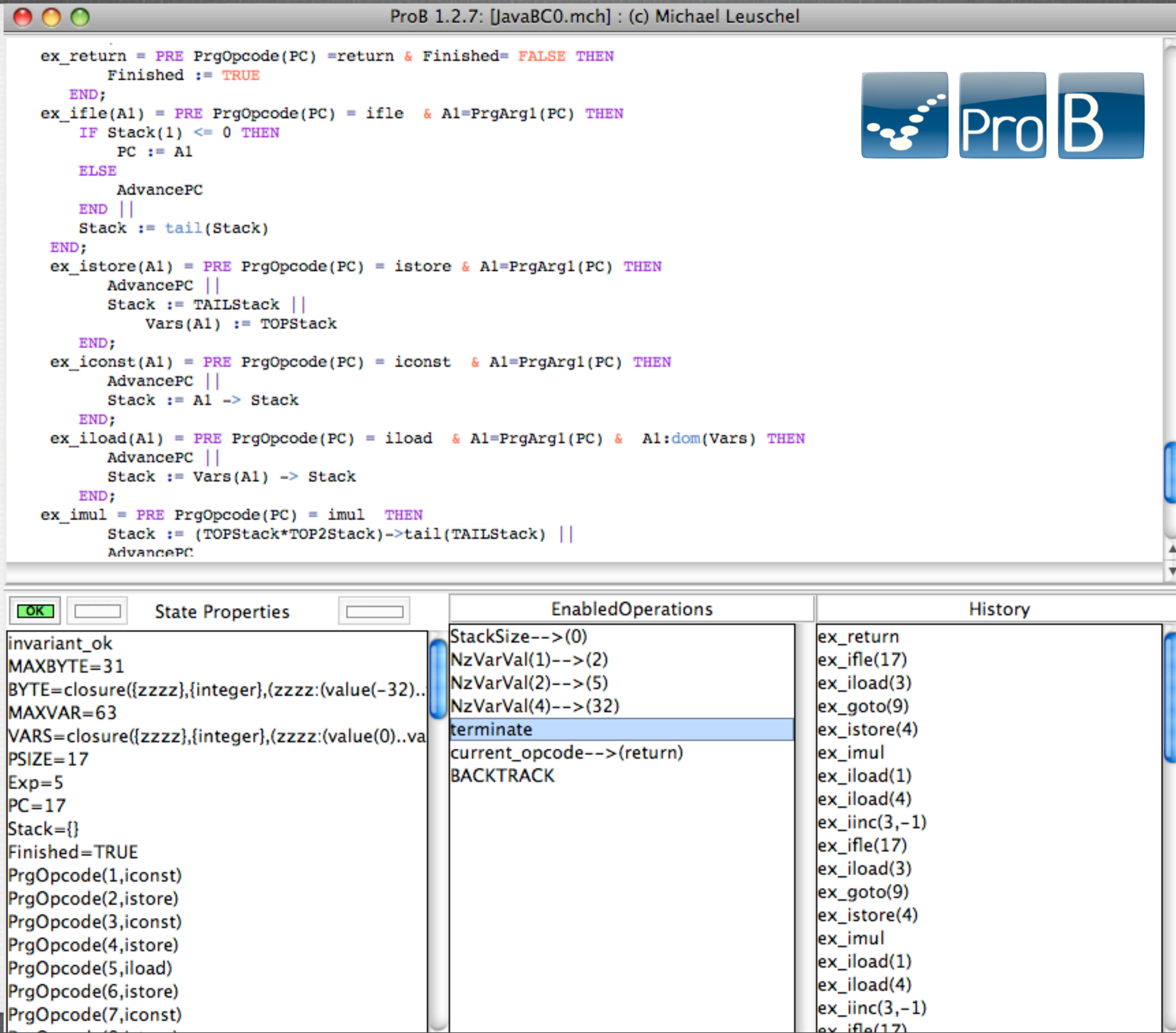
OUR TOOL

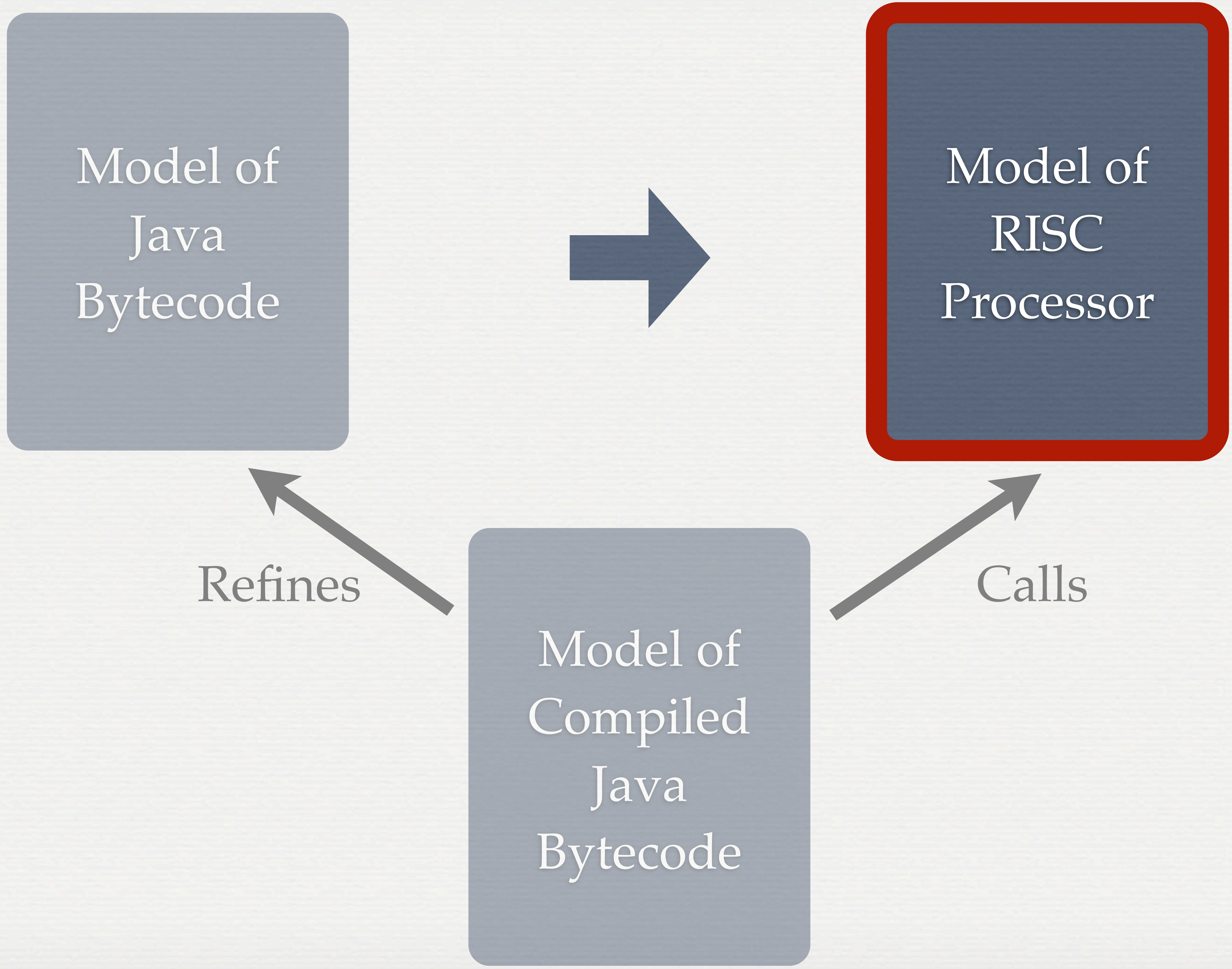
Languages: B,
CSP, Z, CSP | | B, ...

Model Checking
(LTL, Symmetry)
Animation
Refinement Checking

Used for
Teaching B

Industrial
Applications

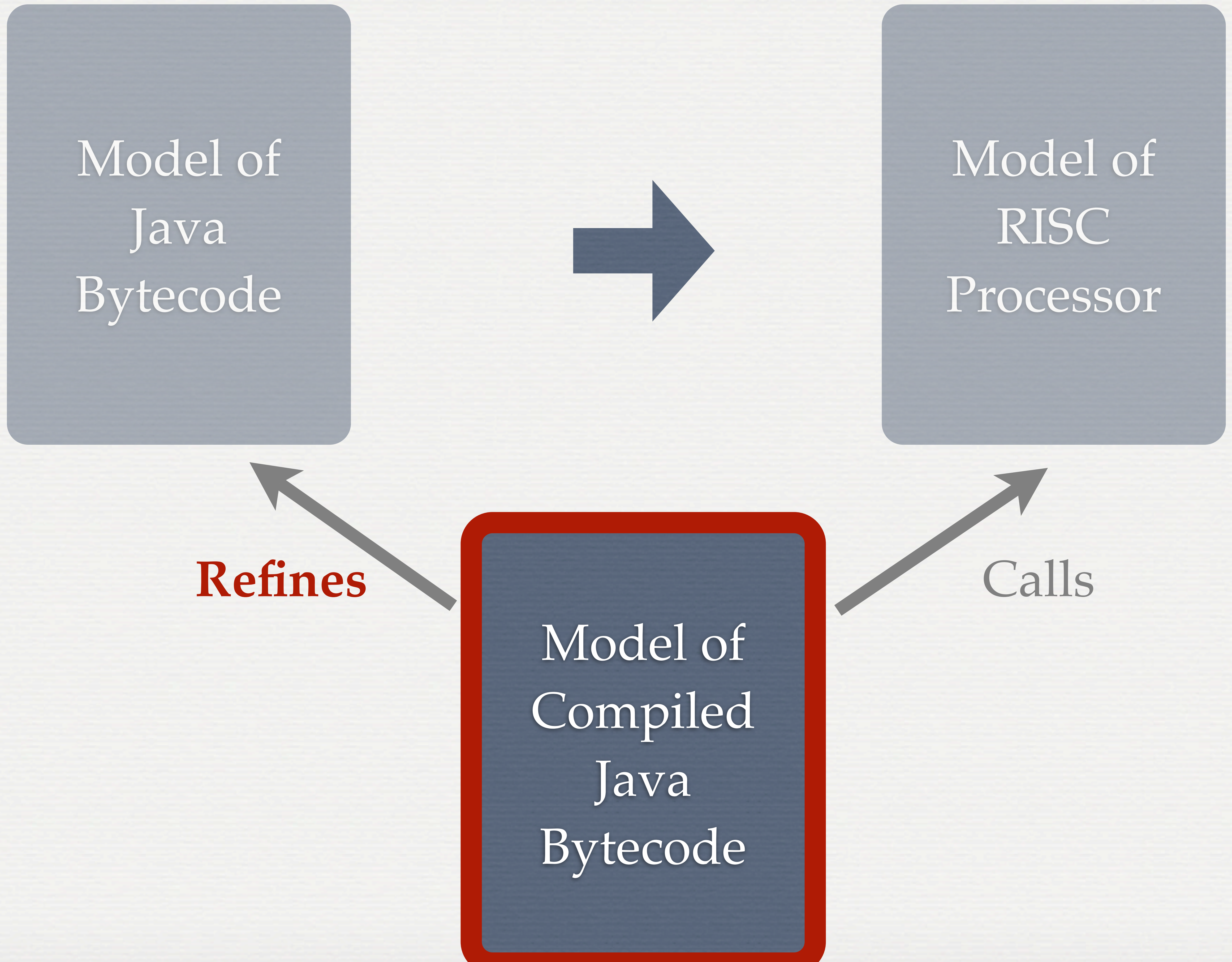

ProB 1.2.7: [JavaBC0.mch] : (c) Michael Leuschel
ex_return = PRE PrgOpcode(PC) =return & Finished= FALSE THEN
Finished := TRUE
END;
ex_ifle(A1) = PRE PrgOpcode(PC) = ifle & A1=PrgArg1(PC) THEN
IF Stack(1) <= 0 THEN
PC := A1
ELSE
AdvancePC
END ||
Stack := tail(Stack)
END;
ex_istore(A1) = PRE PrgOpcode(PC) = istore & A1=PrgArg1(PC) THEN
AdvancePC ||
Stack := TAILStack ||
Vars(A1) := TOPStack
END;
ex_iconst(A1) = PRE PrgOpcode(PC) = iconst & A1=PrgArg1(PC) THEN
AdvancePC ||
Stack := A1 -> Stack
END;
ex_iload(A1) = PRE PrgOpcode(PC) = iload & A1=PrgArg1(PC) & A1:dom(Vars) THEN
AdvancePC ||
Stack := Vars(A1) -> Stack
END;
ex_imul = PRE PrgOpcode(PC) = imul THEN
Stack := (TOPStack*TOP2Stack)->tail(TAILStack) ||
AdvancePC
OK
State Properties
invariant_ok
MAXBYTE=31
BYTE=closure({zzzz},{integer},(zzzz:(value(-32)..
MAXVAR=63
VARS=closure({zzzz},{integer},(zzzz:(value(0)..va
PSIZE=17
Exp=5
PC=17
Stack={}
Finished=TRUE
PrgOpcode(1,iconst)
PrgOpcode(2,istore)
PrgOpcode(3,iconst)
PrgOpcode(4,istore)
PrgOpcode(5,iload)
PrgOpcode(6,istore)
PrgOpcode(7,iconst)
EnabledOperations
StackSize-->(0)
NzVarVal(1)-->(2)
NzVarVal(2)-->(5)
NzVarVal(4)-->(32)
terminate
current_opcode-->(return)
BACKTRACK
History
ex_return
ex_ifle(17)
ex_iload(3)
ex_goto(9)
ex_istore(4)
ex_imul
ex_iload(1)
ex_iload(4)
ex_iinc(3,-1)
ex_ifle(17)
ex_iload(3)
ex_goto(9)
ex_istore(4)
ex_imul
ex_iload(1)
ex_iload(4)
ex_iinc(3,-1)



Model of Java Bytecode
Model of RISC Processor
Refines
Calls
Model of Compiled Java Bytecode

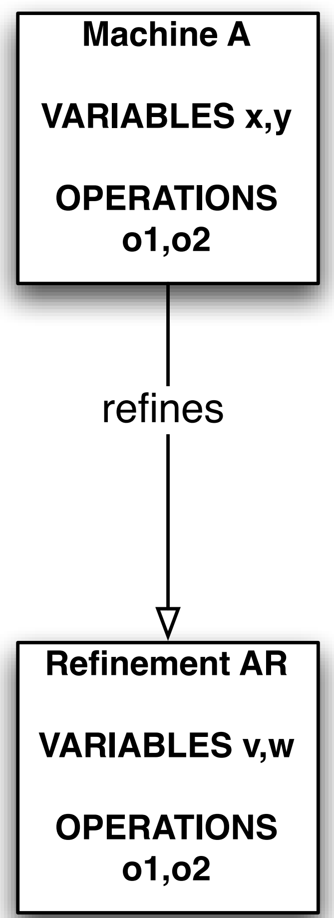
B MODEL OF RISC

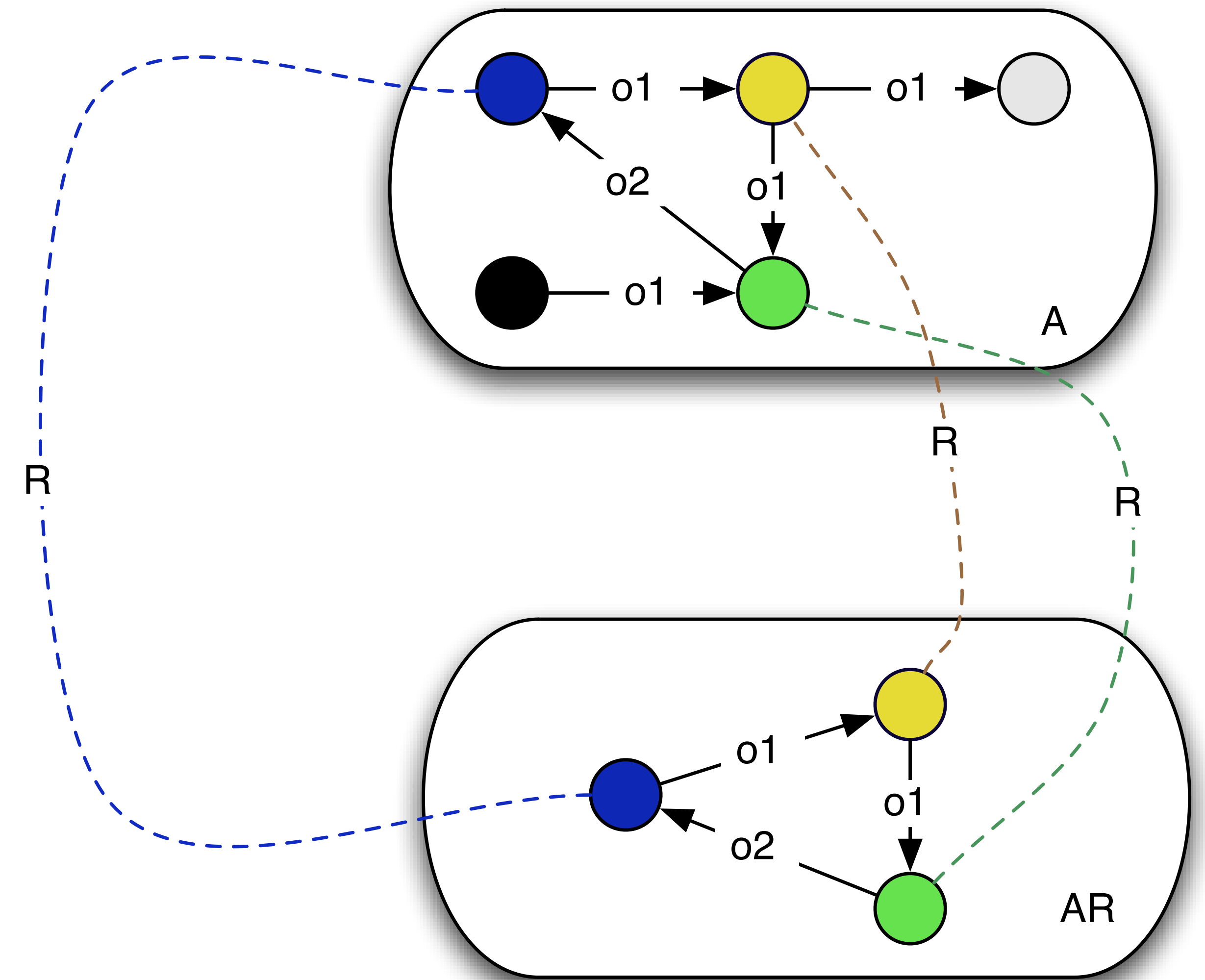
```
MACHINE RISC
CONSTANTS
  NrReg,  /* Number of registers */   MSize,   /* Memory Size */
  RBYTE,  MAXRBYTE
PROPERTIES
  MAXRBYTE = 31 &  /* 127 & */
  NrReg:INT & NrReg>1 &  MSize:INTEGER & MSize>1 &
  RBYTE = (-MAXRBYTE-1)..MAXRBYTE &
  NrReg =2 & MSize = 4*(MAXRBYTE+1)-1
VARIABLES
   R,  /* Register Contents */   MEM  /* Memory Contents */
INVARIANT
   R: 1..NrReg --> INTEGER & MEM: 0..MSize --> INTEGER
INITIALISATION
   R := %x.(x:1..NrReg | 0) ||    MEM := %y.(y:0..MSize | 0)
OPERATIONS
   LDI(r,imm) = PRE r:1..NrReg & imm:RBYTE THEN
     R(r) := imm  END;
   LDM(r,mem) = PRE mem:0..MSize & r:1..NrReg THEN
     R(r) := MEM(mem)   END;
   STM(r,mem) = PRE mem:0..MSize & r:1..NrReg THEN
      MEM(mem) := R(r)  END;
   ADD(r1,r2,r3) = PRE r1: 1..NrReg & r2: 1..NrReg & r3: 1..NrReg THEN
      R(r1) := R(r2)+R(r3)  END;
   MUL(r1,r2,r3) = PRE r1: 1..NrReg & r2: 1..NrReg & r3: 1..NrReg THEN
      R(r1) := R(r2)*R(r3)   END;
     SUBT(r1,r2,r3) = PRE r1: 1..NrReg & r2: 1..NrReg & r3: 1..NrReg THEN
      R(r1) := R(r2)-R(r3)  END;
   res <-- ISPOS(r) = PRE r:1..NrReg THEN
     IF R(r)> 0 THEN res := TRUE
      ELSE   res := FALSE  END
```



Model of Java Bytecode
Model of RISC Processor
Refines
Calls
Model of Compiled Java Bytecode

Example - B Refinement





$$c \in init(AR) \implies \exists a \in init(A) \cdot c\ R\ a$$

$$c\ R\ a \land c\ COP\ c' \implies \exists a' \cdot a\ AOP\ a' \land c'\ R\ a'$$

COMPILATION BY REFINEMENT

Phase 1: without PC

```
ex_iload(A1) = PRE
            PrgOpcode(PC)=iload &
            PrgArg1(PC)=A1 THEN
    LDM(1,A1);
    STM(1,TOP+1);
    AdvancePC
  END;
```

```
REFINEMENT JavaBCR1
REFINES JavaBC0
INCLUDES RISC
```

```
VARIABLES
  PC, Finished   /* Stack, Vars */
INVARIANT
  !v.(v:dom(Vars) => Vars(v) = MEM(v))
&
  !sv.(sv:dom(Stack) => Stack(sv) =
MEM(STACKOFFSET+sv))

  /*  Memory Layout:
  0:       var(0)
  1:       var(1)
   ...
  MAXVAR:   var(MAXVAR)

  MAXVAR+1  Stack(1)
  ...
 */
```

COMPILATION BY REFINEMENT

Phase 1: without PC

```
ex_iload(A1) = PRE
            PrgOpcode(PC)=iload &
            PrgArg1(PC)=A1 THEN
    LDM(1,A1);
    STM(1,TOP+1);
    AdvancePC
  END;
```

```
REFINEMENT JavaBCR1
REFINES JavaBC0
INCLUDES RISC
```

```
VARIABLES
 PC, Finished   /* Stack, Vars */
INVARIANT
  !v.(v:dom(Vars) => Vars(v) = MEM(v))
&
  !sv.(sv:dom(Stack) => Stack(sv) =
MEM(STACKOFFSET+sv))

   /*  Memory Layout:
   0:      var(0)
   1:      var(1)
    ...
   MAXVAR:   var(MAXVAR)

   MAXVAR+1  Stack(1)
   ...
 */
```

FURTHER DEVELOPMENTS

- Add Program Counter in RISC
- Construct an explicit representation of the compiled program

```
iconst 2
iconst2
imul
istore 1
iload 1
return
```

```
RProg(0,ldi1)   RArg(0,2)       /* LDI 1,2 */
RProg(1,stm1)   RArg(1,64)      /* STM 1, 64 */
RProg(2,ldi1)   RArg(2,2)
RProg(3,stm1)   RArg(3,65)
RProg(4,ldm1)   RArg(4,65)
RProg(5,ldm2)   RArg(5,64)
RProg(6,mul112) RArg(6,0)
RProg(7,stm1)   RArg(7,64)
RProg(8,ldm1)   RArg(8,64)
RProg(9,stm1)   RArg(9,1)
RProg(10,ldm1)  RArg(10,1)
RProg(11,stm1)  RArg(11,64)
RProg(12,hlt)   RArg(12,0)
RProg(13,hlt)   RArg(13,0)
```

FINDINGS

- Several errors in translation were uncovered
  - We found a subtle error in the translation of iconst: argument provided to a RISC operation was in the range 0..63, RISC machine expected -32..31.
  - In one case stack elements added at wrong side
  - ...
- Tool Support (Prover, Animator, Model Checker) very important !

CODE OPTIMISATIONS

```
ex_iload_istore(CA1,SA1) =
  PRE PrgOpcode(PC) = iload  &
        CA1=PrgArg1(PC) &
        PrgOpcode(PC+1) = istore &
          SA1=PrgArg1(PC+1) THEN
    PC := PC+2  ||
    Vars(SA1) := Vars(CA1)
  END;
```

ALTERNATE APPROACHES

- Compilation by partial evaluation
- Haskell, Isabelle
- CiaoPP [Univ. Madrid]
- Related work Coq [Leroy et al], ASM [Börger et al]

SOME RELATED WORK

- Goos,Zimmerman: Verifix Project
- Xavier Leroy: Compcert compiler (CMinor to PowerPC), 36000 lines of Coq (13 % compiler)
- Pnueli: Translation Validation
- ...

KEY ASPECTS OF OUR

- Key aspects of our work:
  - **mixture** of proof and validation
  - start from **intermediate** level bytecode
  - compilation as B **refinement**
    - (but no inductively defined datastructures in B yet)

CONCLUSIONS

- DeCCo: informal parts contain serious flaws
- Formally verified compilation by B refinement
  - Various tools very useful in the process:
    - Prover, Automated Refinement Checker
    - Animator, Model Checker to validate spec
- A formally verified compiler for JavaBC feasible
- Step towards Grand Challenge

THANKS TO THE STUPS TEAM

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

- Jens Bendisposto
- Carl Friedrich Bolz
- Nadine Elbeshausen
- Fabian Fritz
- **Marc Fontaine**
- Michael Jastram
- Li Luo
- Daniel Plagge
- Mireille Samia
- Corinna Spermann

deploy

ProB

Research Areas:

Animation
Model Checking
B, CSP, Z, ...
Compilers & Interpreters
Static Analysis
Partial Evaluation, JIT
Logic Programming
Prolog, Haskell
Java RCP
Requirements
Python

Thank you !