Automated Program and Model Transformation Technology

Ira D. Baxter, Ph.D.

idbaxter@semanticdesigns.com

Jeff Gray, Ph.D.

gray@cis.uab.edu

Tutorial Overview

- Part I. Program Transformation
  - Transformation Tool overview and Relevance to ADM
  - Theory of transformation systems
  - Some transformation systems and applications
- Part II. Model Transformation
  - Model Transformation to Support Model Evolution
  - Model-Driven Program Transformation

Who are we?

- Dr. Ira D. Baxter
  - CEO of Semantic Designs
  - 35 years Software Engineering:
    OS, Compilers, Transform Systems, Reuse
  - Architect of Design Maintenance System (DMS)
- Dr. Jeff Gray
  - CIS Department, University of Alabama at Birmingham
    - Assistant Professor
    - Research Focus: Software engineering, model-integrated computing, generative programming, aspect-oriented software development
  - PhD from Vanderbilt University
    - Institute for Software Integrated Systems (ISIS)
  - 2000-2005: DARPA PCES Project on Model-Driven Transformation
  - 2005-2007: NSF CSR Project on Model-driven Performance Analysis

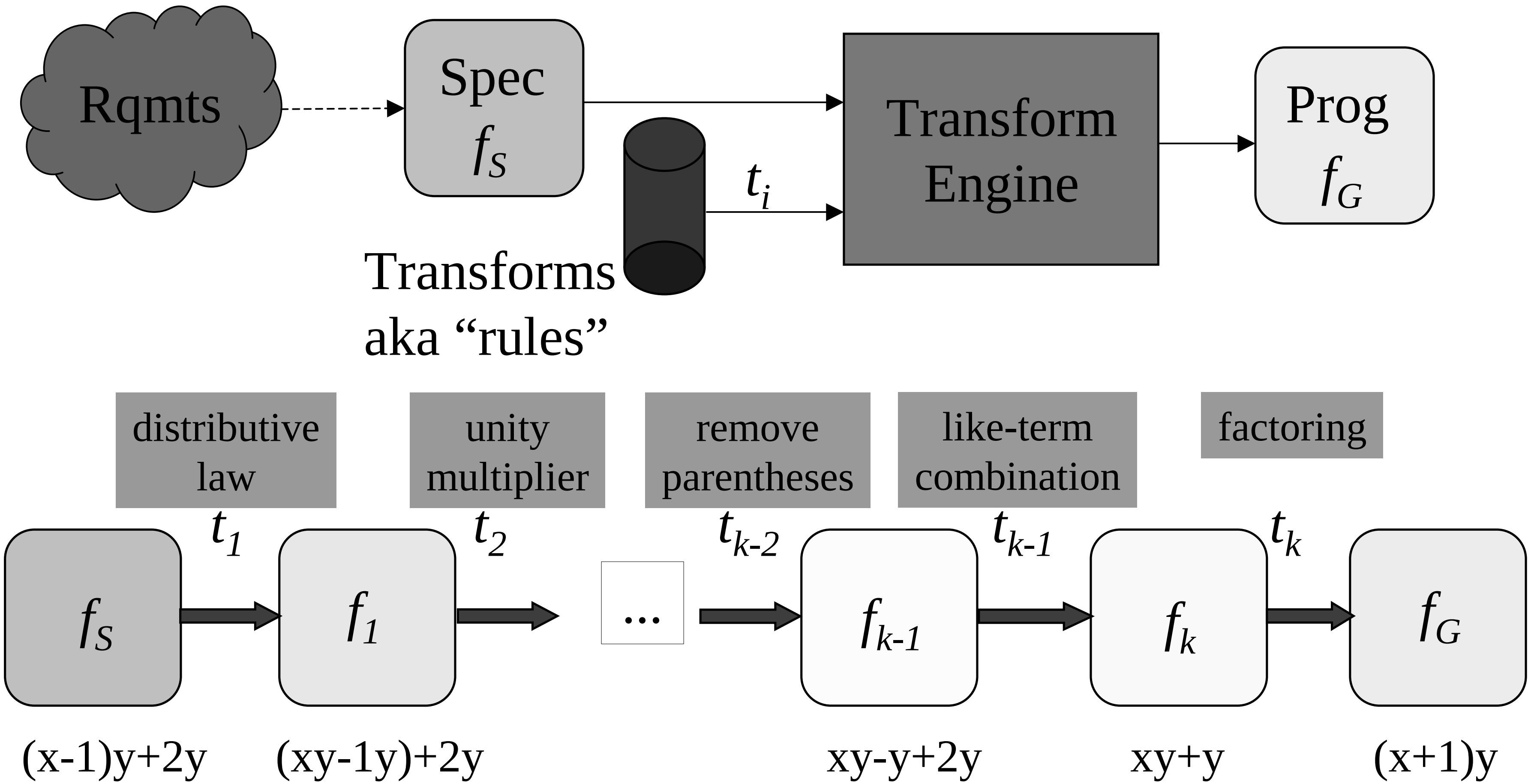
Contents

Introduction

- Theory
- Implementation
- Some Transformation Systems
- Applications
- Summary

Transformation Systems

Stepwise Semiautomatic Conversion of Specs to Code



Spec Transformed to code with Transforms: Summing A Unix File

Specification f_S

```
print sum(<file:"invoices">);
```

Some sequence of transforms $t_1, t_2, \ldots t_k$

Implementation f_G

```
local s,i,n1,f1,v1; s=0;
f1=new filehandle; open f1,"invoices";
n1=0; while not(eof(f1))
   do { position f1,sizeof(v1)*n1;
        read f1,v1; n1=n1+1;
        s=s+v1; }
close f1; free(f1); print s;
```

A Simple Example: Summing A Unix File

Specification

```
print sum(<file:"invoices">);
```

t_1

```
print ( local s,i; s=0;
        for i=1 to length(<file:"invoices">)
          { s=s+<file:"invoices.txt">[i]; }
        s );
```

t_2

```
local s,i; s=0;
for i=1 to length(<file:"invoices">)
    s=s+<file:"invoices txt">[i];
print s;
```

t_3

```
local s,i; s=0;
for i=1 to (local n,f,v; n=0; f=new filehandle;
             open f,"invoices";
             while not(eof(f)) do
                { read f,v; n=n+1; };
             close(f); free(f);
             p)
  s=s+<file:"invoices">[i];
print s;
```

t_4

```
local s,i; s=0;
local t1;
{ local n,f,v; n=0;
  f=new filehandle; open f,"invoices";
  while not(eof(f)) do
       { read f,v; n=n+1};
  close(f); free(f); t1=n};
for i=1 to t1
    { s=s+<file:"invoices">[i]; }
print s;
```

t_5

```
local s,i,t1; s=0;
local n1,f1,v1; n1=0;
f1=new filehandle; open f1,"invoices";
while not(eof(f1)) do
   { read f1,v1; n1=n1+1;};
close(f1); free(f1); t1=n1;
for i=1 to t1
    { s=s+<file:"invoices">[i]; }
print s;
```

t_6

```
local s,i,n1,f1,v1; s=0; n1=0;
f1=new filehandle; open f1,"invoices";
while not(eof(f1)) do
   { read f1,v1; n1=n1+1;};
close(f1); free(f1);
for i=1 to n1
    { s=s+<file:"invoices">[i]; }
print s;
```

t_7

```
local s,i,n1,f1,v1; s=0; n1=0;
f1=new filehandle; open f1,"invoices";
while not(eof(f1))
   do { read f1,v1; n1=n1+1;};
close(f1); free(f1);
for i=1 to n1
  { s=s+(local v,f; f=new filehandle;
         open f,"invoices";
         position f,sizeof(v)*(i-1);
         read f,v; close f; free f; v);
  }
print s;
```

t_8

Naïve intermediate program
Opens file and reads it <u>twice</u>

A Simple Example II

Transforms on this page optimize naïve program into efficient one

t_8

```
local s,i,n1,f1,v1; s=0; n1=0;
f1=new filehandle; open f1,"invoices";
while not(eof(f1))
   do { read f1,v1; n1=n1+1;};
close(f1); free(f1);
for i=1 to p1
  { s=s+(local v,f; f=new filehandle;
         open f,"invoices";
         position f,sizeof(v)*(i-1);
         read f,v; close f; free(f); v);
  }
print s;
```

t_9

```
local s,i,n1,f1,v1; s=0; n1=0;
f1=new filehandle; open f1,"invoices";
while not(eof(f1))
   do { position f1,sizeof(v1)*n1;
        read f1,v1; n1=n1+1;};
close(f1); free(f1);
for i=1 to n1
  { local v2,f2; f2=new filehandle;
    open f2,"invoices";
    position f2,sizeof(v2)*(i-1);
    read f2,v2; close f2; free(f2);
    s=s+v2;
  }
print s;
```

t_{10}

```
local s,i,n1,f1,v1; s=0; n1=0;
f1=new filehandle; open f1,"invoices";
while not(eof(f1))
   do { position f1,sizeof(v1)*n1;
        read f1,v1; n1=n1+1;};
close(f1); free(f1);
Local f2; f2=new filehandle;
for i=1 to n1
  { local v2; open f2,"invoices";
    position f2,sizeof(v2)*(i-1);
    read f2,v2; close f2;
    s=s+v2;
  }
free(f2); print s;
```

```
local s,i,n1,f1,v1; s=0; n1=0;
f1=new filehandle; open f1,"invoices";
while not(eof(f1))
   do { position f1,sizeof(v1)*n1;
        read f1,v1; n1=n1+1;};
close(f1);
for i=1 to n1
  { local v2; open f1,"invoices";
    position f1,sizeof(v2)*(i-1);
    read f1,v2; close f1;
    s=s+v2;
  }
free(f1); print s;
```

```
local s,i,n1,f1,v1, v2; s=0; n1=0;
f1=new filehandle; open f1,"invoices";
while not(eof(f1))
   do { position f1,sizeof(v1)*n1;
        read f1,v1; n1=n1+1;};
close(f1); open f1,"invoices";
for i=0 to n1-1
  { position f1,sizeof(v2)*i;
    read f1,v2; s=s+v2;
  }
close f1; free(f1); print s;
```

```
local s,i,n1,f1,v1,v2; s=0;
f1=new filehandle; open f1,"invoices";
n1=0; while not(eof(f1))
   do { position f1,sizeof(v1)*n1;
        read f1,v1; n1=n1+1;};
i=0; while i<=n1-1
   do { position f1,sizeof(v2)*i;
        read f1,v2; s=s+v2;
        i=i+1;
      }
close f1; free(f1); print s;
```

More transforms

```
local s,i,n1,f1,v1; s=0;
f1=new filehandle; open f1,"invoices";
n1=0; while not(eof(f1))
   do { position f1,sizeof(v1)*n1;
        read f1,v1; n1=n1+1;};
i=0; while i<=n1-1
   do { position f1,sizeof(v1)*i;
        read f1,v1; i=i+1;
        s=s+v1; }
close f1; free(f1); print s;
```

t_k

```
local s,i,n1,f1,v1; s=0;
f1=new filehandle; open f1,"invoices";
n1=0; while not(eof(f1))
   do { position f1,sizeof(v1)*n1;
        read f1,v1; n1=n1+1;
        s=s+v1; }
close f1; free(f1); print s;
```

A Simple Example: A transformed spec

$\downarrow t_2$

```
local s,i; s=0;
for i=1 to length(<file:"invoices">)
    s=s+<file:"invoices.txt">[i];
print s;
```

$\downarrow t_3$

```
local s,i; s=0;
for i=1 to (local n,f,v; n=0; f=new filehandle;
            open f,"invoices";
            while not(eof(f)) do
               { read f,v; n=n+1; };
            close(f); free(f);
            p)
  s=s+<file:"invoices">[i];
print s;
```

A Simple Example:
Some of the Transforms Used

t_3

```
length(<file:"filename">
```

⇒

```
(local n,f,v; n=0; f=new filehandle;
            open f,"filename";
            while not(eof(f)) do
               { read f,v; n=n+1; };
            close(f); free(f);
            p)
```

t_7

```
<file:"filename">[i]
```

⇒

```
(local v,f; f=new filehandle;
          open f,"filename";
          position f,sizeof(v)*(i-1);
          read f,v; close f; free f; v)
```

t_{10}

```
for i=lb to ub
  { code1 // that doesn't touch f
    local f; f=new filehandle;
    code3 // that doesn't free f
    free(f);
    code2 // that doesn't touch f
  }
```

⇒

```
local f; f=new filehandle;
for i=lb to ub
  { code1
    code3
    code2
  }
free(f)
```

Expertise == Number of Rules

- Mathematics
  - Novice (9th grade algebra): $x+0 \Rightarrow x$
  - Amateur (HS Senior): $\sin^2(x)+\cos^2(x) \Rightarrow 1$
  - Journeyman: (Frosh Calculus) integrals
  - Craftsman: (B.S. Math) Linear Algebra, Group Theory
  - Expert: (Ph.D. Mathematics) Category Theory, Topology, …
- Transformation Engines
  - Toy: several rules
  - Useful: 50 rules (simplification/optimization)
  - Powerful: 250 rules (testing, code generation)
  - Indispensable: 2000 rules (massive program translation)

… + "how to use rules" knowledge

- Mathematics
  - Solve for variable, Simplification (novice)
  - Simultaneous equations
  - Change of domain (expert)
  - …
- Transformation Engines
  - Tactic
    - Apply rules till exhaustion (useful)
  - Strategy
    - Metaprograms (sophisticated)

Practical Transformation Machinery

- Theory long established
  - Source to Source transforms ("rewrites")
  - Refinement theory
- Emphasis now focused on *generation* and *change*
- Need integrated mechanisms
  - Parsing, PrettyPrinting
  - Name/Type Resolution, Rewriting, Analyzers
  - Essential technology already developed
- *Practical tools need support for scale*
  - Large real codes
  - Multiple languages
  - Symbolic Computation for analysis and change

Potential Applications of Transformation Systems

- Code Generation
  - Formal Specifications to computer languages
  - Converting Models to code
- Code Modification
  - Program generation of code from specifications
  - Translation from one language to another
  - Instrumentation of code for runtime data collection
  - Program restructuring/refactoring
  - Performance optimization
  - *Massive Change*
- Code Analysis
  - Documentation extraction
  - Reverse Engineering
  - Quality assessment
  - Bug finding

Software Transformation Tools Need Infrastructure Too

- To build a *conventional application*
  - Define specifications
  - Implement application
  - Use *Operating System* to provide standard services
    - File/Terminal I/O, CPU management, Security
- To build an *automated software change tool*
  - Define specifications
  - Implement tool
  - Use …?… to provide standard services
    - Lexing/Parsing
    - *Life after Parsing*
      - Tree/Symbol table building, Analysis management, Transformation, PrettyPrinting, …

Practical Program Transformation Systems

- Criteria
  - Can accept new language definitions easily
  - Can carry out source-to-source transformations
  - Have been used for serious commercial tasks
- Industrial-strength Integrated Transformation Systems
  - Commercial
    - DMS® (Semantic Designs, **www.semanticdesigns.com**)
    - JANUS (Software Revolution, **www.softwarerevolution.com**)
    - Refine5 (Reasoning Systems, **not available**)
  - Academic or open source
    - CodeWorker (**codeworker.free.fr**)
    - TXL (Queen's University, **www.txl.ca**)
    - Stratego/XT (University of Utrecht/CWI **www.program-transformation.org/Stratego/webhome**)
  - Untested: OMG's Query/View/Transformation (QVT)
- *Big Effort to build… not likely to be many of them*

Contents

- Introduction
  Theory
- Implementation
- Some Transformation Systems
- Applications
- Summary

Theory

Concepts and Components

- Transforms and Transformations
- Specifications
- Semantics
- The meaning of “Implementation”
- Control: Navigating the Implementation Space
- Design Capture

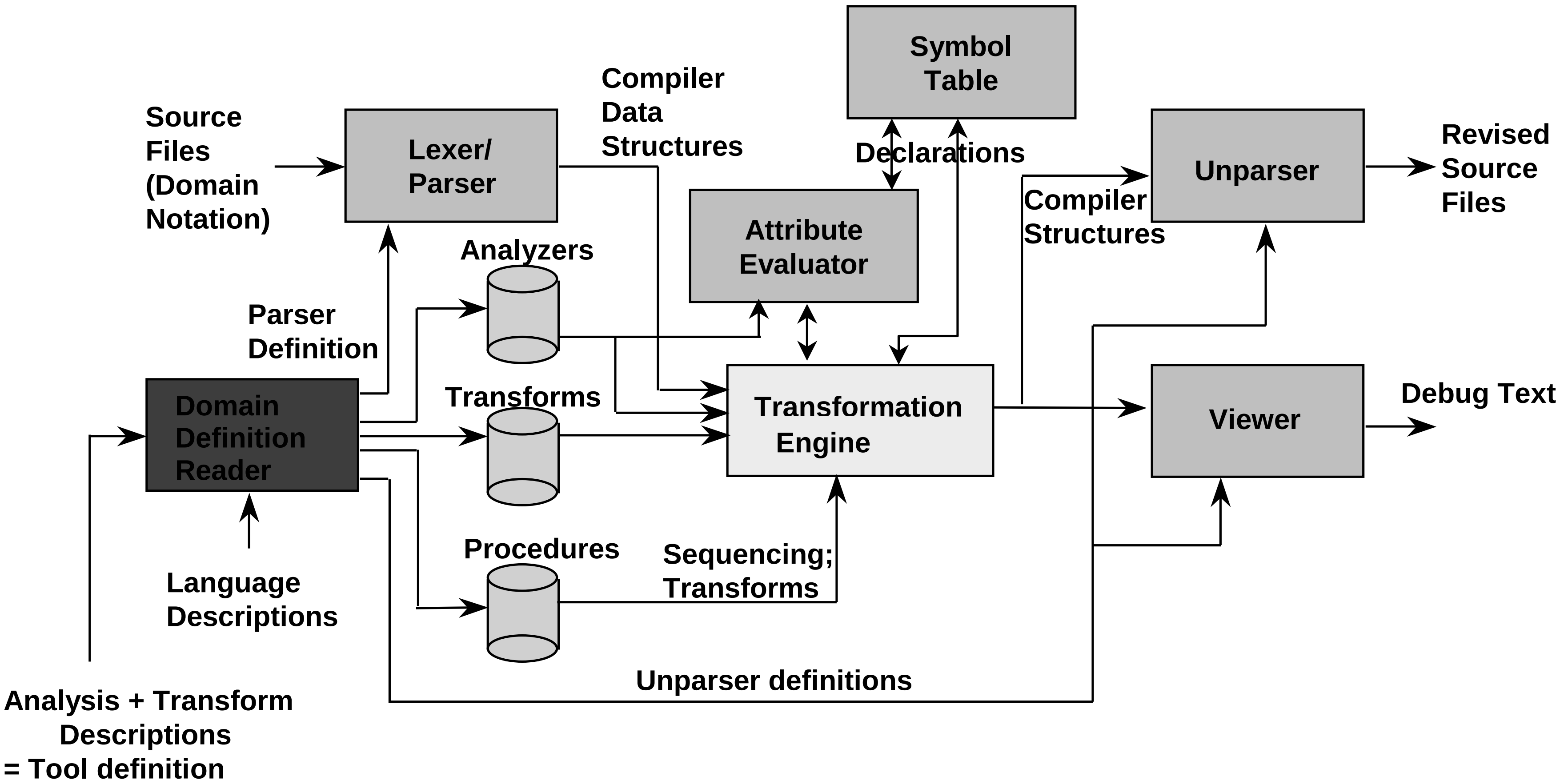
Transformation System *Concepts*

- Domain — *selects instance set of following concepts*
- Schema — *changed by transformation system*
- Semantics — *define meaning of schema*
- Performance Measure — *compute qualities of schemas*
- Performance Predicate — *compares qualities*
- Specification — *defines desired qualities (functional + performance)*
- State — *specification + cached inferences*
- Transform — *state modifiers*
- Locator/locale — *place/area in state*
- Binding — *mapping for variables in transform via locator*
- Transformation — *applied, bound transform*
- Property preservation — *notion of implementation*
- Metaprogram — *program which applies transforms*

Transformation System *Components*

- Formal *Source Language* (**with semantics**) *specification*
- Formal *Target Language* (**with semantics**) *program*
- (Semantic) definition of *Implementation*
- A *Representation* for source/target instances
- A set of *Transforms* applicable to source
- A mechanical *Tool* to apply transforms
- (optional) Means to *apply multiple transforms*
- (optional) Source *Parser*
- (optional) Target *PrettyPrinter*
- (optional) Means to *display* intermediate *states*
- (optional) Means to *undo* transformations
- (optional) Means for performing *analyses*
- (optional) Means for *augment*ing Transformation System databases

Transformation Engine Architecture



Transformation System *Components*: Example

Wang’s Rules

simplify propositional calculus formulas

Given boolean formula over variables with operators:

and, or, not, implies

*determine if formula is a **tautology** or not*

true	*tautology*
or(a,not(a))	*tautology*
and(a,b)	*not tautology*
implies(not(or(a,b)),and(b,not(a)))	*status?*

(Useful as part of precondition evaluator!)

Wang’s Rules

Source to Source transformations on Boolean equation syntax

```
default domain Boolean. -+ and, or have been declared as associative, commutative in grammar

rule not1():term->term: "not(true)" -> "false".
rule not2():term->term: "not(false)" -> "true".
rule not3(x:term):term->term: "not(not(\x))" -> "\x";

rule impl(x:term,y:term):term->term = "implies(\x,\y)" -> "or(not(\x),\y)".
rule and(x:term,y:term):term->term = "and(\x,\y)" -> "not(or(not(\x),not(\y)))".

rule or01(x:term):term->term = "or(true,\x)"-> "true".
rule or02(x:term):term->term = "or(false,\x) " -> "\x".
rule or03(x:term):term->term = "or(\x,not(\x)) " -> " true".
rule or04(x:term,y:term):term->term = "or(\x,or(not(\x),\y)) " -> " true".
rule or05(x:term):term->term = "or(\x,\x) " -> "\x".
rule or06(x:term,y:term):term->term = "or(\x,or(\x,\y)) " -> " or(\x,\y)".
rule or07(x:term,y:term):term->term = "or(not(or(\x,\y)),\x) " -> " or(\x,not(\y))".
rule or08(x:term,y:term,z:term):term->term = "or(not(or(\x,\y)),or(\x,\z)) " -> " or(\x,or(not(\y),\z))".
rule or09(x:term,y:term):term->term = "or(not(or(\x,\y)),not(\x)) " -> " not(\x)".
rule or10(x:term,y:term,z:term):term->term = "or(not(or(\x,\y)),or(not(\x),\z)) " -> " or(not(\x),\z)".
rule or11(x:term,y:term):term->term = "or(not(or(not(\x),\y)),\x) " -> "\x".
rule or12(x:term,y:term,z:term):term->term = "or(not(or(not(\x),\y)),or(\x,\z)) " -> "or(\x,\z)".
rule or13(x:term,y:term):term->term = "or(not(or(not(\x),\y)),not(or(\x,\y))) " -> " not(\y)".
rule or14(x:term,y:term,z:term):term->term = "or(not(or(not(\x),\y)),or(not(or(\x,\y)),\z)) " -> "or(not(\y),\z)".
```

Wang's Rules … in action

Proposition:

implies(not(or(a,b)),and(b,not(a)))

Transformation Steps:

Step implies(not(or(a,b)),**and(b,not(a))**) ⇒ implies(not(or(a,b)),**not(or(not(not(a)),not(b)))**)
by and(x,y) → not(or(not(x),not(y))) ***«and» rewrite***

Step implies(not(or(a,b)),not(or(**not(not(a))**,not(b)))) ⇒ implies(not(or(a,b)),not(or(**a**,not(b))))
by not(not(x)) ⇒ x ***«not3» rewrite***

Step **implies(not(or(a,b)),not(or(a,not(b))))** ⇒ **or(not(not(or(a,b))),not(or(a,not(b))))**
by implies(x,y) → or(not(x),y) ***«impl» rewrite***

Step or(**not(not(or(a,b)))**,not(or(a,not(b)))) ⇒ or(**or(a,b)**,not(or(a,not(b))))
by not(not(x)) ⇒ x ***«not3» rewrite***

Step **or(or(a,b),not(or(a,not(b))))** ⇒ **or(b,a)**
by or(not(or(not(x),y)),or(x,z)) ⇒ or(x,z) ***«or12» rewrite***

Normal form:

or(b,a)

Transformation System *Concepts*
for Wang's Rules

• Domain	*boolean formulas*
• Schema	*abstract syntax trees*
• Semantics	*boolean truth tables*
• Performance Measure	*tautological*
• Performance Predicate	*"tautological?"*
• Specification	*explicit boolean formula; implicit performance predicate*
• State	*transformed boolean formula*
• Transform	*equational rules*
• Locator/locale	*subtree root*
• Binding	*subtrees matching variables*
• Transformation	*transform + locator*
• Property preservation	*same truth tables*
• Metaprogram	*Wang's genius: apply rules, any order!*

Transformation System *Components for Wang's Rules*

- Formal Source Language — *e.g. Propositional terms*
- Formal Target Language — *e.g. Propositional terms*
- Definition of Implementation — *Normalization of propositions*
- Correctness of Implementation — *Rules preserve truth table*
- Representation for Language instances — *Abstract Syntax Trees*
- Tool to apply transforms — *associative-commutative term rewrite engine*
- Means to apply multiple transforms — *e.g. Exhaustive bottom-up application*
- Source Language Parser — *e.g. Propositional term parser*
- Target Language Pretty Printer — *e.g. Propositional term printer*
- Means to display intermediate states — *Trace rewrite applications*
- (optional) Means to undo transformations
  Application of rewrite rule backwards at same position
  Generic undo mechanism on Abstract Syntax Trees
- Means for augmenting Transformation System databases
  Storing internal representation of parsed transformation rules

Theory

- Concepts and Components
  Transforms and Transformations
- Specifications
- Semantics
- The meaning of “Implementation”
- Control: Navigating the Implementation Space
- Design Capture

What is a *transform*?

A partial function from specs/programs to specs/programs

t: Spec $\rightarrow$ Spec

Compilers, YACC, VLSI synthesizers, program restructurers

Often represented as a *rewrite rule with pattern variables*:

eliminate-additive-identity: $x+0 \Rightarrow x$ *optimization*

t(Locator): Spec $\rightarrow$ Spec

implement-sum: sum(*var*,*limit*,*vector*) $\Rightarrow$

begin local s=0,*var*;
do *var*=1 to *limit;*
s=s+*vector*(*var*); enddo
return s
end

refinement

Transforms for Mathematics

eliminate-additive-identity:	$x+0 \Rightarrow x$
cancellation-of-terms:	$x - x \Rightarrow 0$
distributive-law:	$x * (y+z) \Rightarrow x * y + x * z$
reverse-distributive-law:	$x * y + x * z \Rightarrow x * (y+z)$
integer-additive-arithmetic:	$1+1 \Rightarrow 2$ $1+2 \Rightarrow 3$ ... $n1+n2 \Rightarrow n3$ **if** $n3=\mathrm{sum}(n1,n2)$
an-equality-rule:	$x+n = x \Rightarrow \mathrm{false}$ **if** $n \neq 0$

Transform *Applicability Conditions*

Domain of transform defined as applicability predicate
(transforms are partial functions!)

x+0 $\Rightarrow$ x **if** true

analyzers

x / x $\Rightarrow$ 1 **if** $x \neq 0$

$bnf \Rightarrow$ yacc(bnf) if isLALR(bnf)

var = *expr*; call *proc*;	$\Rightarrow$	call *proc*; *var* = *expr*;	**if**	no-side-effects(*proc*) and $var \notin$ references(*proc*)

Transform*ations*

Application of a Transform

Usually applied at a place (*locator*) in the program

t(Locator): Spec $\rightarrow$ Spec

[token]

apply *eliminate-additive-identity @ 2:6*

[line]

[1]	do j = 1 to 10		do j = 1 to 10
[2]	s=s+3+0+2	$\Rightarrow$	s=s+3+2
[3]	p=p+0		p=p+0
[4]	end		end

[token 6]

... rewrite binds *x* to s+3 rewriting s+3+0 ⇨ s+3

Transforms as Procedures

the way they "really" work

- Arbitrary procedure for each transform

eliminate-additive-identity: $x+0 \Rightarrow x$

```
procedure eliminate-additive-identity(line,token)
   if        is-token(@(line,token),[”+”])
      and  is-token(@(line,token)+1,[”0”])
      and  is-last-token-of-additive-expression(@(line,token)-1)
      and  not is-token(@(line,token)+2,[“*”,”/”,”%”])
   then
      delete-tokens([@(line,token)+1,@(line,token)])
   else fail
end procedure
```

- Rewrites are implemented as a *distinguished* procedure
  – implements pattern matching and replacement

Theory

- Concepts and Components
- Transforms and Transformations
  Specifications
- Semantics
- The meaning of “Implementation”
- Control: Navigating the Implementation Space
- Design Capture

Specifications: Stating Desires

Specification: an acceptance predicate *P* over a desired artifact
s is an acceptable program iff *P(s)* is true

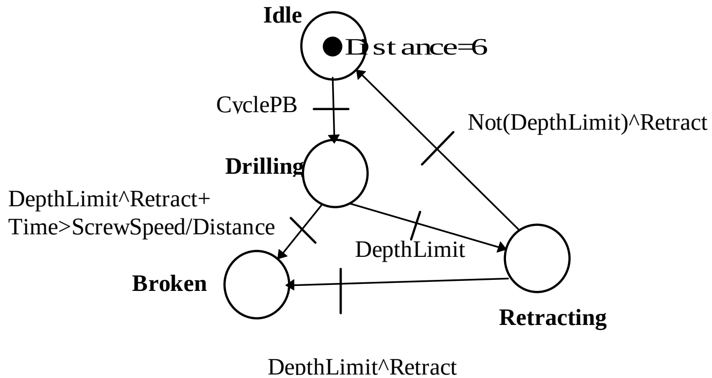
Issues

- What needs to be expressed?
  - *Inputs/Outputs, Function, runtime costs, runtime environment, construction costs, etc..*
- How ambitious are our measurements?
  - *sorted, $O(n^2)$, "low coupling", "usable", "cheap", "profitable", "beautiful"*
- What do we know how to express?
  - *Function, Time, Cost, Metrics*
- How to express?
  - *How formal? Which formalism? One formalism? A Combination?*

Key questions for every specification:

- *What predicate does it represent?*
- *What does it NOT say?*

Practical Specifications



***Somehow** state what programs are acceptable*

•**Functional (f)** specifications:

what a program computes

abstract computations	sum(i,10,quantity(i)*price(i))
pre-/post-conditions	post: exists i: prime(vector(i))
domain specific notations	*Colored Petri net, above right*

•**Performance (P)** specifications:

how well it computes it

target language	targetlanguage=Pascal
size, complexity, structuredness,...	complexity=O(n ln n)

Both required: $P = f_0 \wedge P_{rest}$

Specifications

state more than just what is computed

A full specification (S):

M(S) = f: length(x) = $n \Rightarrow$

$\forall\ i \in 1..n\text{-}1$: $f(x)[i] \leq f(x)[i+1]$

f_S

functionality

and complexity(S) = O(n ln n)

and target-language(S) = Pascal

and souce-lines-of-code(S) $\leq$ 50

and no-gotos(S)

P_{rest}

performance

Types of Specifications

- Natural Language *you can read it*
  - *structured(?) English*
- Operational *can execute it*
  - *procedural, functional*
- Nonoperational *can reason about it*
  - *predicate calculus, pre-/post- conditions*
- Algebraic *can structure it*
  - *abstract data types and their properties*
- Performance *say more than function*
  - *any of the above*
- Unusual properties
  - *divergence, nontermination*
- Domain Specific *problem experts' notation*
- Real-time *significant time constraints*
- Mixtures *address larger problems*

- Text versus Graphical representation *how it is presented*
  - *Doesn't change what can be expressed, only how it is rendered*

Natural Language Specification

- The zero-th Fibonacci number is 1.
- The first Fibonacci number is 1.
- Each other Fibonacci number is the sum of the two largest smaller Fibonacci numbers.

Operational Specification

```
function Fibonacci(x)
  local result
  if x < 2 then
    result = 1
  else begin
    result = Fibonacci(x-1)
    result = result
               + Fibonacci(x-2)
  end
  return result
end
```

Procedural

```
function Fibonacci(x) =
  if x < 2 then
    1
  else
      Fibonacci(x-1)
    + Fibonacci(x-2)
```

Functional

- Essentially programming in an abstract language
- Executable UML is a graphical example

Algebraic Specifications

- Specify primitive datatypes by **functional** *signatures* and *axioms*
- Specify more complex types by combining algebraic specifications *(an algebra over algebras!)*

Spectrum

SPECNAME =
{ **sort** type*1*;
...
sort type*N*;

function*1*: argtype*1* $\rightarrow$ resulttype*1*;
...
function*M*: argtype*M* $\rightarrow$ resulttype*M*;

axioms
formula*1* = formula*2*;
formula*3* $\Rightarrow$ formula*4* = formula*5*;
...
endaxioms;
}

signature

axioms

The Quintessential Stack

"Stackness" defined algebraically

STACK =
{ **sort** Stack α;

empty: Stack α;
push: Stack $\alpha \times \alpha \rightarrow$ Stack α;

Stack α **generated by** empty, push;

pop: Stack $\alpha \rightarrow$ Stack α;
top: Stack $\alpha \rightarrow \alpha$;

axioms
pop(push(s, e)) = s;
top(push(s, e)) = e;
$\neg$ DEF(pop(empty));
$\neg$ DEF(top(empty);
endaxioms;
}

Some Specification Languages

- LOTOS
- RAISE/RSL
- Z
- VDM
- SDL
- Colored Petri Nets

Very "formal"

- IDL
- StateCharts
- OCL (UML)
- YACC/LEX
- VHDL
- SQL

Domain-specific specifications

Property specifications

- "bigO" notation
  O(N^2)

Low level specifications

- C, Java, AWK, ...

OMG Standards as Specifications

- MOF as Syntax
- Model (Instance) as specification
  - Terminology glitch: OMG "model" vs. "semantic model"
    - OMG: "model" is the specification
    - Theory: "model" is a structure that can be interpreted to satisfy a spec
- UML
  - Class Diagrams
  - State Charts
  - Executable UML
- SBVR
  - Specification of Business Rules
- …

Key questions for every specification:

- *What predicate does it represent?*
- *What does it NOT say?*

Theory

- Concepts and Components
- Transforms and Transformations
- Specifications
  Semantics
- The Meaning of “Implementation”
- Control: Navigating the Implementation Space
- Design Capture

What is Semantics?

Communication is done using *Syntax*

Understanding each other comes from a common "agreement" about the *Interpretation* of the *Syntax*

Semantics is the
"agreement" about the *Interpretation* of *Syntax*

Semantics: *Syntax* $\rightarrow$ Set(*Interpretation*)

also called "meaning function"

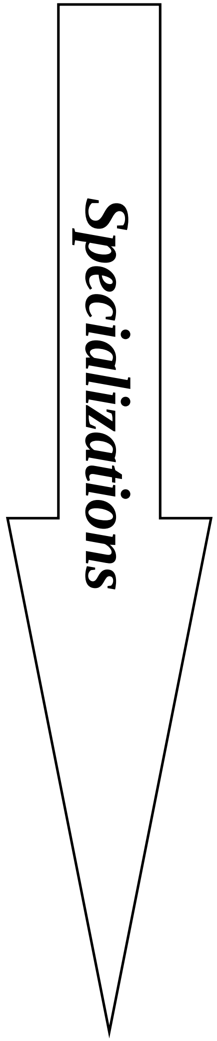
Note the *set* of interpretations in the meaning function!

Example of set of interpretations: meaning of the English word **band**:

- a thin strip of flexible material
- a group of players who perform as an ensemble

Types of Semantics

- Informal Semantics
  - *Everybody is pretty sure they understand*
- Operational Semantics
  - *Executable by well-defined interpreter*
- Transformational Semantics
  - *Map to notation with known semantics*
- Denotational Semantics
  - *Map to lambda calculus*
- Models and Algebraic Semantics
  - *Domain with known semantics ensures theorem truth*



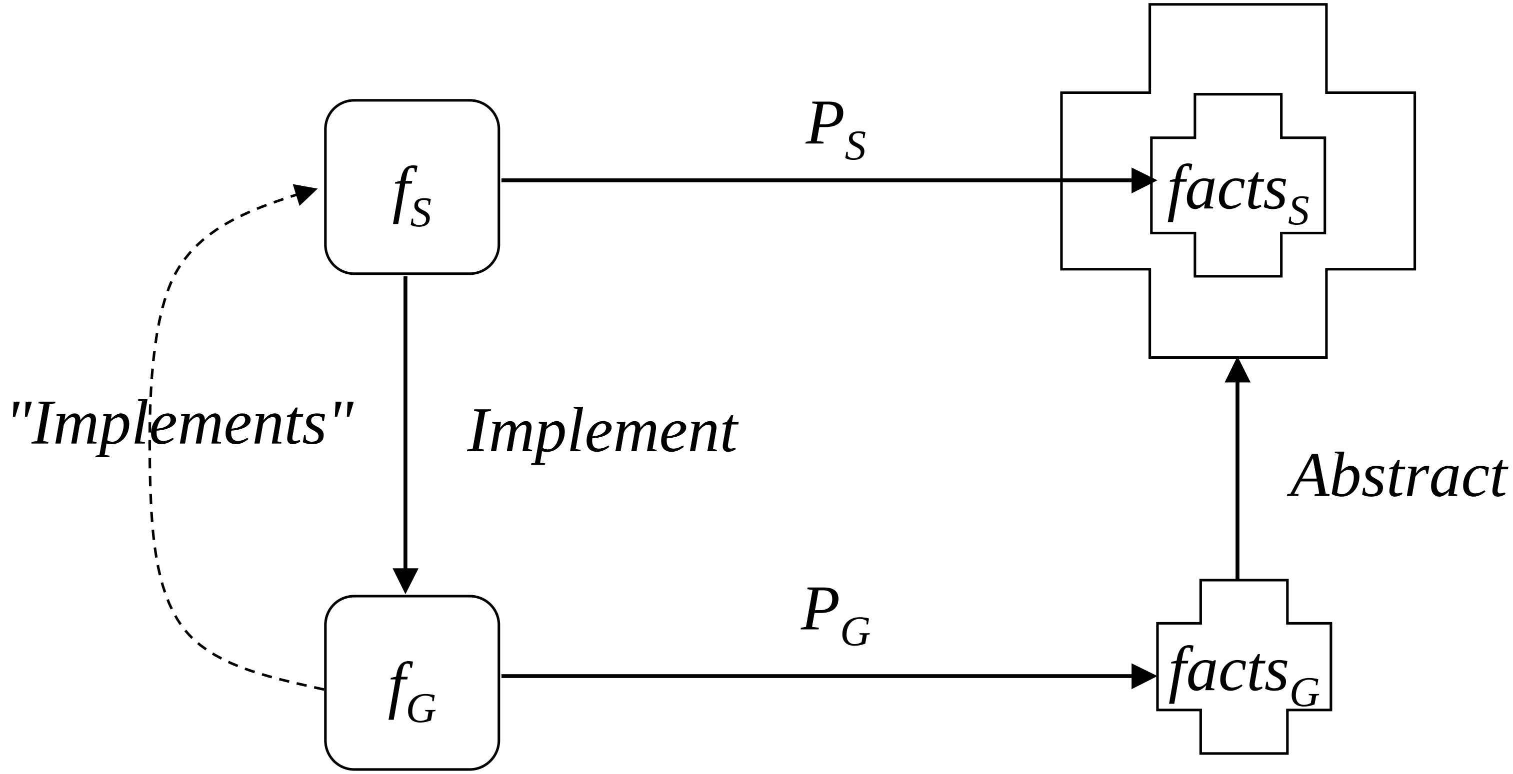
Theory

- Concepts and Components
- Transforms and Transformations
- Specifications
- Semantics

  The Meaning of “Implementation”
- Control: Navigating the Implementation Space
- Design Capture

Implementation - Properties View

How do we know a program implements a specification?

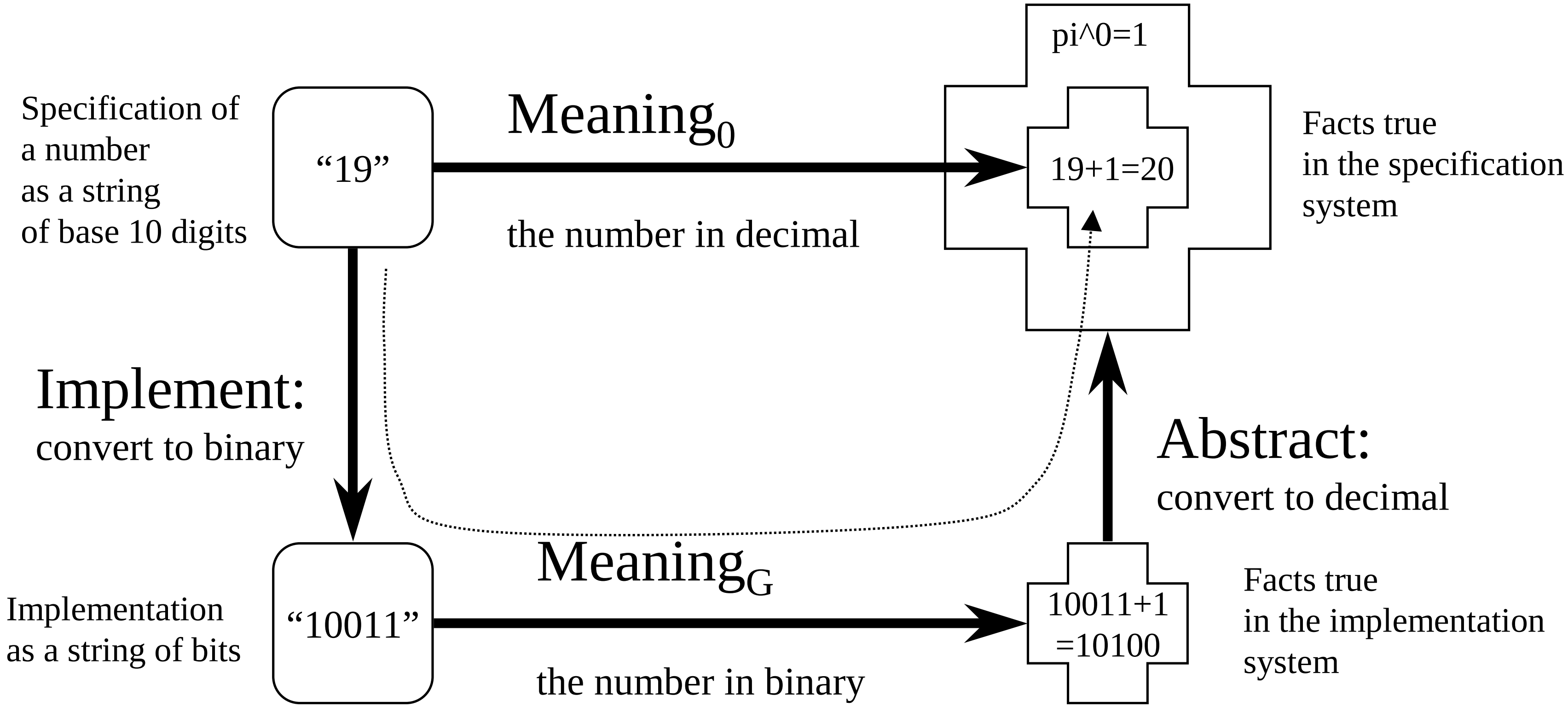


$$P_S(f_S) \subseteq Abstract(P_G(Implement(f_S)))$$

Implementation

A simple example using Numbers

Preservation of facts implied by specification

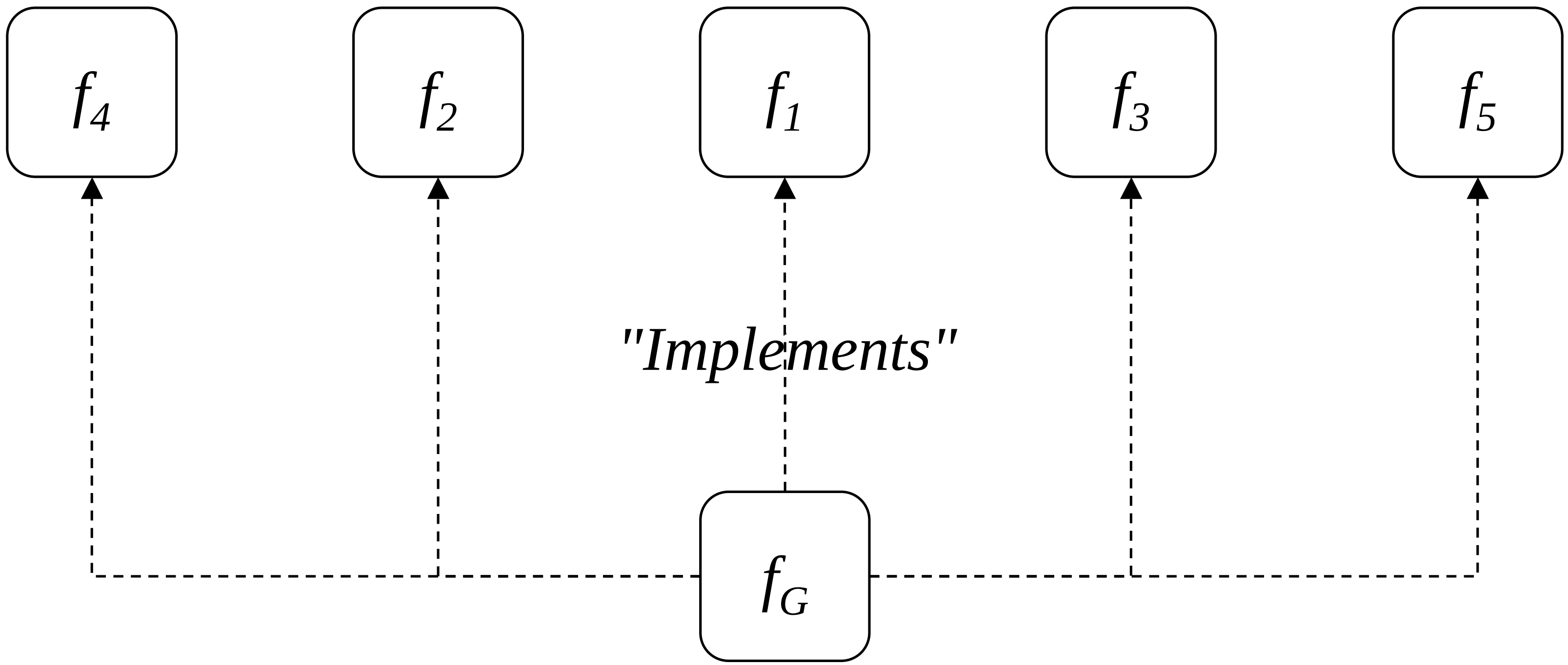


Good DE finds Domain, and <Implement,Abstract> pair!

Implementation with "multiple" specifications

functionality, performance, ….

- Each specification must be satisfied
  - $\Rightarrow$ Conjunction of specifications must be satisfied



Theory

- Concepts and Components
- Transforms and Transformations
- Specifications
- Semantics
- The Meaning of “Implementation”

  Control: Navigating the Implementation Space
- Design Capture

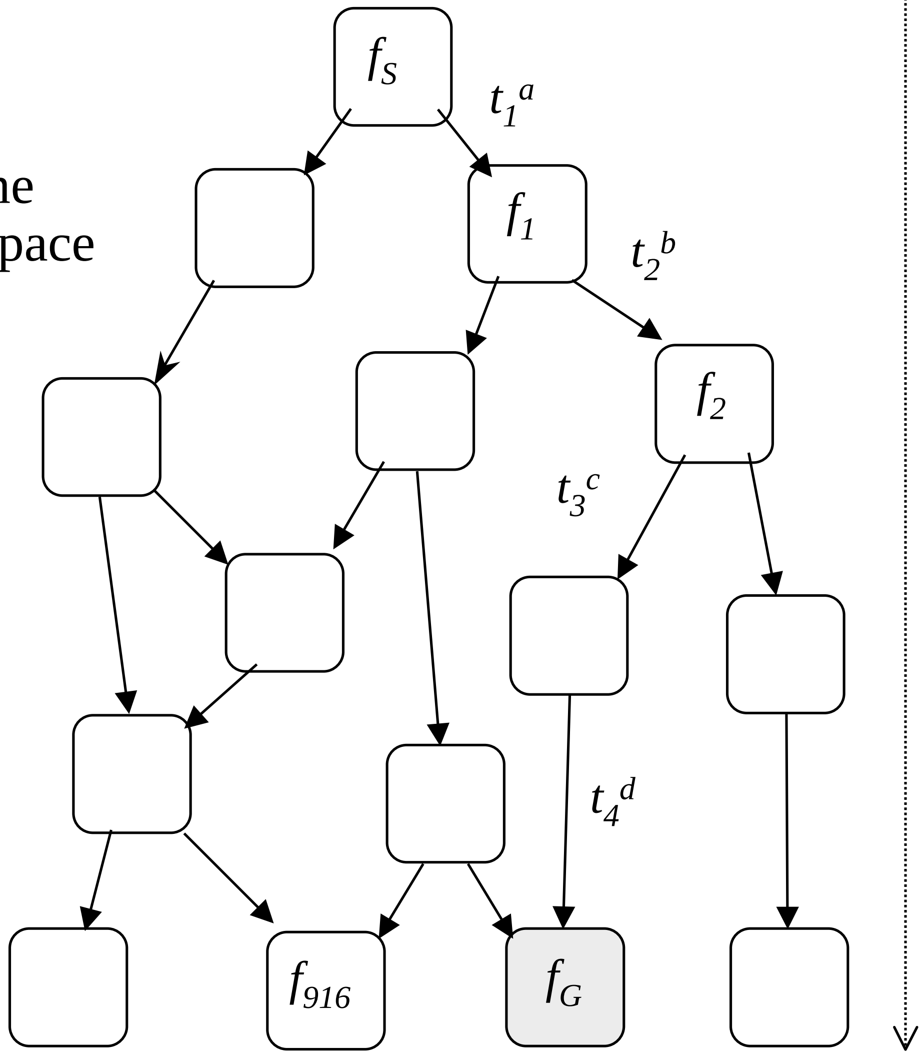
The Control Problem
How to make implementation decisions?

- Transformational implementation of a specification goes through a huge number of intermediate states
- At each intermediate state many transforms are applicable at many locales leading to different possible implementations
- *How do we choose the “right” transformations?*

Control

Navigating the implementation space

Transformations
on specification define
Huge Implementation Space



> 10K
steps!

Navigation Issues

- **Efficacy** (Effectiveness) of transformation choice:
  - How to choose transformation approaching
    - Executability?
    - Desired Performance?

    How do we measure "approaching" effectiveness?
  - How to choose consistent transformations?

    (cf. consistent refinement problem)
- **Efficiency** of transformation choice:
  - How easy for software engineer to choose transformation?
  - How fast for tool to execute transformation?

Navigation Techniques

- Efficacy:
  - Manual selection of transformations
  - Heuristics
  - Repeated domain refinement/optimization
  - Performance-directed search
  - Metaprograms
  - Replay of previous transformation sequences
- Efficiency
  - Top down or bottom up exhaustive application of set of transforms
  - Heuristics
  - Searches
  - Confluent and terminating (Church-Rosser) system of transforms

    application order irrelevant to achieve goal

    e.g derived from equational theory by Knuth-Bendix completion algorithm
  - Metaprograms

Contents

- Introduction
- Theory

  Implementation
- Some Transformation Systems
- Applications
- Summary

Constructing a Transformation System is Hard

Transformation System Components

- Formal *Source Language*
- Formal *Target Language*
- (Semantic) definition of *Implementation*
- A *Representation* for source/target instances
- A set of *Transforms* applicable to source
- A mechanical *Tool* to apply transforms
- Means to *apply multiple transforms*
- Source *Parser*
- Target *PrettyPrinter*
- Means to *display* intermediate *states*
- Means to *undo* transformations
- Means for performing *analyses*
- Means for *augment*ing Transformation System databases

- Each component
  - must be defined
  - must be implemented

- Transformation systems should also provide
  - large number of languages
  - large transform database
  - large metaprogram supply

Let's look at ***some(!)*** *details*

Specification and Program Representation

- Primitive Text (aka character string)
  - only useful for very simple transformations (sed, awk, …)
- Tokenized Text
  - pedagogical but still not very practical
- Abstract Syntax Trees (AST)
  - technology: parsing, term rewriting
- Resolved Abstract Syntax Trees (R-AST)
  - abstract syntax trees and symbol table forming a directed acyclic graph
  - technology: term graph rewriting
- Data- and Control-flow Representations
  - simplify analysis of procedural code for interesting relations
  - technology: graph rewriting
- Arbitrary Graph
  - can capture arbitrary relations
  - technology: graph rewriting

Tokenized Text

The Program as Text:

```
s = 0 . 0 ;
f o r ( i = 1 ; i < 5 ; i + + )
  s =   i % 2 == 1
        ? 2 * s + 0
        : s + i * i ;
```

The Machine View:

```
s = 0.0 ; for ( i = 1
; i < 5 ; i ++ ) s =
i % 2 == 1 ? 2 *
s + 0 : s + i * i ;
```

Incorrect Match!

Rewrite Rule: x `+ 0` $\Rightarrow$ x

- Rule supplier (engineer) has to
  - describe restrictions for x
    "multiplicative expression"
  - describe restrictions for context
    "no factorial sign after 0"

  Hard to express over tokens!

- Machine has to
  - find locale to apply rewrite rule
  - find binding for x
  - check whether binding satisfies restrictions for x
  - check whether locale satisfies context restrictions

Primitive Text to Tokenized Text

- Technology
  - Lexical Specification (with "Regular Expressions") + Lexer Generator

    single "global" lexical spec often insuffícent ⇒ need multiple lexical specs

- Example Lexical Specification (Fragment):

```
#skip "[\ \t\r\n]+"
#token 'for' "for"
#token identifier [STRING]  "[a-zA-Z][a-zA-Z0-9_]*"
 << (ConvertTokenStringToString (. ?:LiteralString) ? 0 0) >>
#token integer     [INTEGER] "[0-9]+"
 << ?:LiteralInteger = (ConvertDecTokenStringToNatural ? 0 0) >>
#token ':'   "\:"            #token ';'    "\;"
#token '+'   "\+"            #token '*'    "\*"
#token '('   "\("            #token ')'    "\)"
```

ignorable characters

type of token value

regular expression

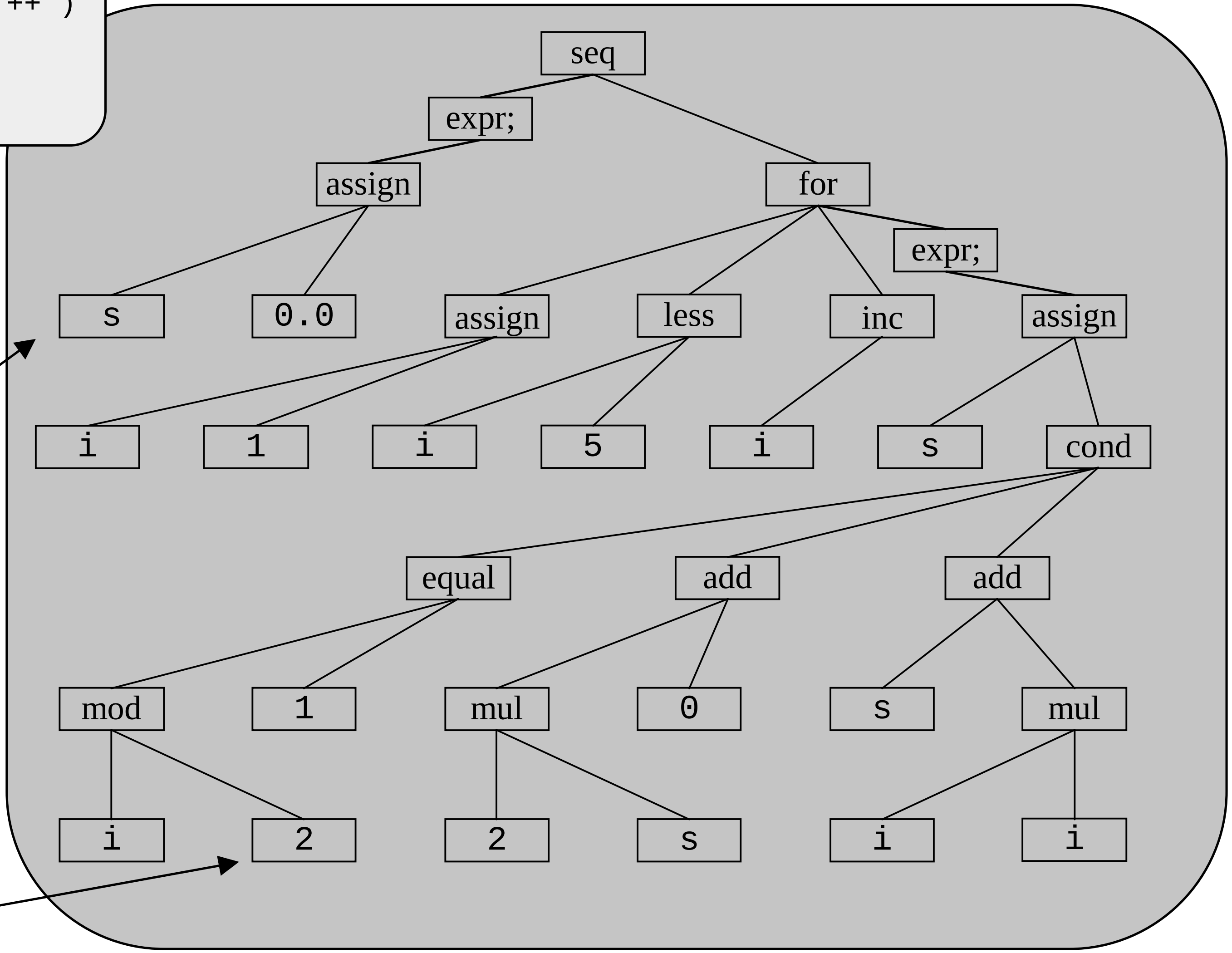
action to be performed when token is recognized

token name

- Note:
  - *Lexical Specification* is *Source Language* Instance
  - *Lexer Generator* is *Transform* producing *Target Language* Code

Abstract Syntax Trees

```
s = 0.0 ;
for ( i = 1 ; i < 5 ; i ++ )
  s =   i % 2 == 1
      ? 2 * s + 0
       : s + i * i ;
```



Really represented as:

identifier *with associated string value* s

Really represented as:

integer *with associated integer value* 2

Tokenized Text to Abstract Syntax Trees

- Technology:
  - Context Free Grammar + Parser Generator

    LL(1), LR(1) often not sufficent ⇒ need more general parser

- Example Grammar (Fragment):

nonterminal

token

```
iteration
     = 'for' '(' expression ';' expression ';' expression ')'
          statement;
statement = expression ';';
expression = identifier '=' conditional;
conditional = expression '?' additive ':' additive;
additive = additive '+' multiplicative;
additive = multiplicative;
primary = identifier;
primary = '(' expression ')';
```

grammar production

terminal token produced by lexer

- Note:
  - *Grammar* is *Source Language* Instance
  - *Parser Generator* is *Transform* producing *Target Language* Code

… and Back to Primitive Text

- Reason:
  - Show engineer transformed specification
  - Provide transformed specification to other tools
- Technology:
  - Hand coded
  - Pretty Printer Specification + Pretty Printer Generator *e.g. box based*
- Example:

```
 iteration-statement
     = ‘for’ ‘(‘ expression ‘;’ expression ‘;’ expression ‘)’ statement;
    <<PrettyPrinter>>: { V(H(‘for’,‘(’,expression,‘;’,expression,‘;’,
                               expression,‘)’,I(statement)); }
additive = additive + multiplicative;
    <<PrettyPrinter>>: { V(I(additive[1]),H(‘+’,multiplicative)); }
```

vertical *indent* *horizontal*

- Note:
  - *Pretty Printer Specification* is *Source Language* Instance
  - *Pretty Printer Generator* is *Transform* producing *Target Language* Code

Winner: Obfuscated “C” Contest

The Maintenance Programmer’s Nightmare

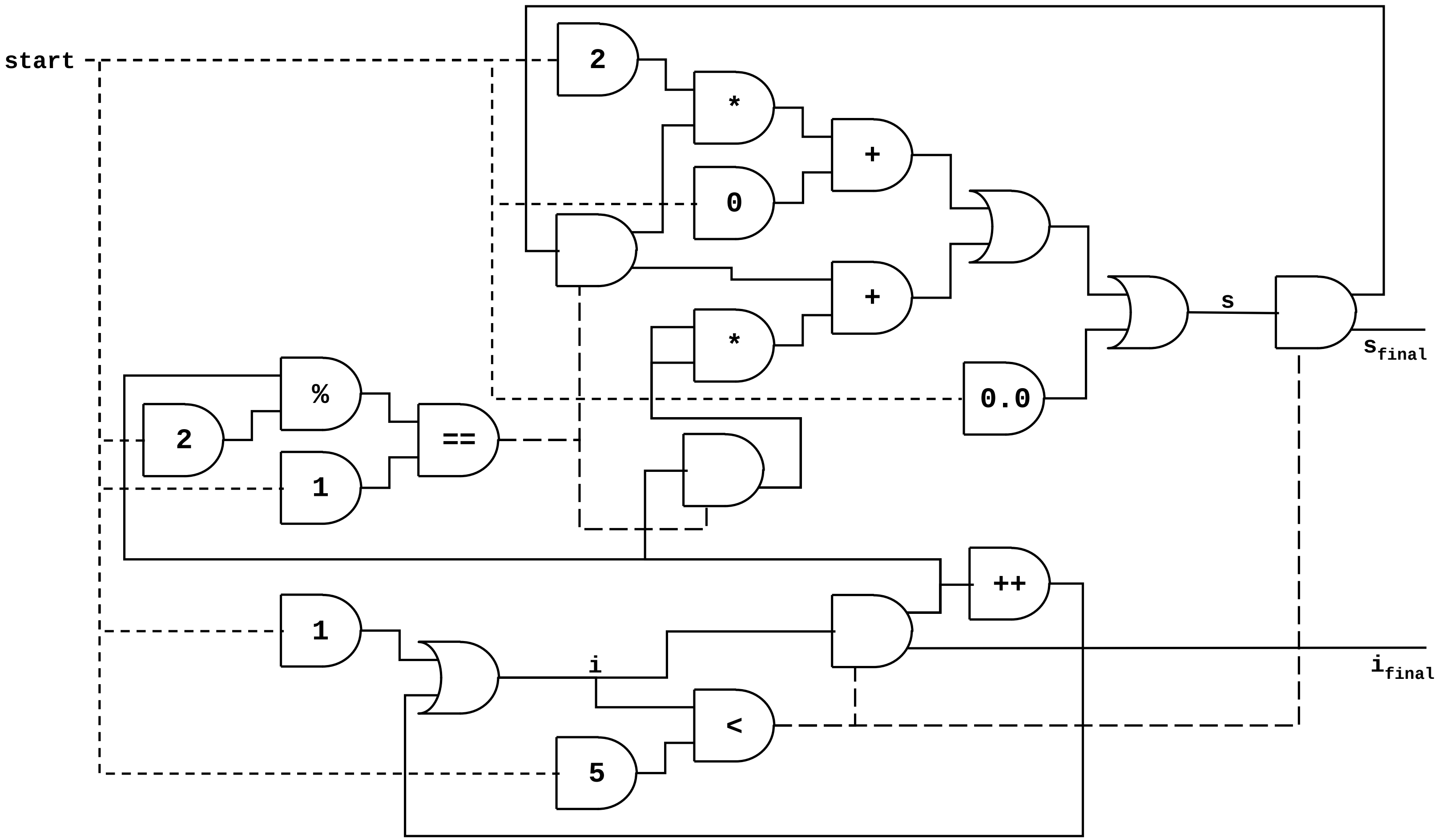
```c
#include                                     <math.h>
#include                                   <sys/time.h>
#include                                   <X11/Xlib.h>
#include                                  <X11/keysym.h>
                                          double L ,o ,P
                                         ,_=dt,T,Z,D=1,d,
                                         s[999],E,h= 8,I,
                                         J,K,w[999],M,m,O
                                        ,n[999],j=33e-3,i=
                                        1E3,r,t, u,v ,W,S=
                                        74.5,l=221,X=7.26,
                                        a,B,A=32.2,c, F,H;
                                        int N,q, C, y,p,U;
                                       Window z; char f[52]
                                    ; GC k; main(){ Display*e=
 XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
; scanf("%lf%lf%lf",y +n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=sin(O)){ struct timeval G={ 0,dt*1e6}
; K= cos(j); N=1e4; M+= H*_; Z=D*K; F+=_*P; r=E*K; W=cos( O); m=K*W; H=K*T; O+=D*_*F/ K+d/K*E*_; B=
sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T*E+ D*B*W; j+=d*_*D-_*F*E; P=W*E*B-T*D; for (o+=(I=D*W+E
*T*B,E*d/K *B+v+B/K*F*D)*_; p<y; ){ T=p[s]+i; E=c-p[w]; D=n[p]-L; K=D*m-B*T-H*E; if(p [n]+w[ p]+p[s
]== 0|K <fabs(W=T*r-I*E +D*P) |fabs(D=t *D+Z *T-a *E)> K)N=1e4; else{ q=W/K *4E2+2e2; C= 2E2+4e2/ K
 *D; N-1E4&& XDrawLine(e ,z,k,N ,U,q,C); N=q; U=C; } ++p; } L+=_* (X*t +P*M+m*l); T=X*X+ l*l+M *M;
  XDrawString(e,z,k ,20,380,f,17); D=v/l*15; i+=(B *l-M*r -X*Z)*_; for(; XPending(e); u *=CS!=N){
                                   XEvent z; XNextEvent(e ,&z);
                                       ++*((N=XLookupKeysym
                                         (&z.xkey,0))-IT?
                                         N-LT? UP-N?& E:&
                                         J:& u: &h); --*(
                                         DN -N? N-DT ?N==
                                         RT?&u: & W:&h:&J
                                          ); } m=15*F/l;
                                          c+=(I=M/ l,l*H
                                          +I*M+a*X)*_; H
                                          =A*r+v*X-F*l+(
                                          E=.1+X*4.9/l,t
                                          =T*m/32-I*T/24
                                           )/S; K=F*M+(
                                           h* 1e4/l-(T+
                                           E*5*T*E)/3e2
                                           )/S-X*d-B*A;
                                           a=2.63 /l*d;
                                           X+=( d*l-T/S
                                            *(.19*E +a
                                            *.64+J/1e3
                                            )-M* v +A*
                                            Z)*_; l +=
                                            K *_; W=d;
                                            sprintf(f,
                                            "%5d  %3d"
                                            "%7d",p =l
                                           /1.7,(C=9E3+
                              O*57.3)%0550,(int)i); d+=T*(.45-14/l*
                             X-a*130-J* .14)*_/125e2+F*_*v; P=(T*(47
                             *I-m* 52+E*94 *D-t*.38+u*.21*E) /1e2+W*
                             179*v)/2312; select(p=0,0,0,0,&G); v-=(
                              W*F-T*(.63*m-I*.086+m*E*19-D*25-.11*u
                               )/107e2)*_; D=cos(o); E=sin(o); } }
```

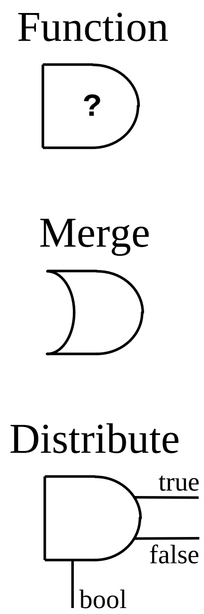
Pretty Printing to *Un*obfuscate

```c
#include <math.h>
#include <sys/time.h>
#include <X11/Xlib.h>
#include <X11/keysym.h>
double L, o, P, _ = dt, T, Z, D = 1, d, s[999], E, h = 8, I, J, K, w[999],
       M, m, O, n[999], j = 3.3e-2, i = 1e3, r, t, u, v, W, S = 7.45e1,
       l = 221, X = 7.26, a, B, A = 3.22e1, c, F, H;
int N, q, C, y, p, U;
Window z;
char f[52];
GC k;
main()
{
  Display * e = XOpenDisplay(0);
  z = RootWindow(e, 0);
  for (XSetForeground(e, k = XCreateGC(e, z, 0, 0), BlackPixel(e, 0));
  scanf("%lf%lf%lf", y + n, w + y, y + s) + 1; y++);
  XSelectInput(e, z = XCreateSimpleWindow(e, z, 0, 0, 400, 400,
               0, 0, WhitePixel(e, 0)), KeyPressMask);
  for (XMapWindow(e, z);; T = sin(O))
    {
      struct timeval G = { 0, dt * 1e6 };
      K = cos(j);
      N = 1e4;
      M += H * _;
      Z = D * K;
      F += _ * P;
      r = E * K;
      W = cos(O);
      m = K * W;
      H = K * T;
      O += D * _ * F / K + d / K * E * _;
      B = sin(j);
      a = B * T * D - E * W;
      XClearWindow(e, z);
      t = T * E + D * B * W;
      j += d * _ * D - _ * F * E;
      P = W * E * B - T * D;
      for (o += (I = D * W + E * T * B, E * d / K * B + v + B / K * F * D) * _; p < y;)
        {
          T = p[s] + i;
          E = c - p[w];
          D = n[p] - L;
          K = D * m - B * T - H * E;
          if (p[n] + w[p] + p[s] == 0 | K < fabs(W = T * r - I * E + D * P) | fabs(D = t * D + Z * T - a * E) > K)
            N = 1e4;
          else
            {
              q = W / K * 4e2 + 2e2;
              C = 2e2 + 4e2 / K * D;
              N - 1e4 && XDrawLine(e, z, k, N, U, q, C);
              N = q;
              U = C;
            }
          ++p;
        }
      L += _ * (X * t + P * M + m * l);
      T = X * X + l * l + M * M;
      XDrawString(e, z, k, 20, 380, f, 17);
      D = v / l * 15;
      i += (B * l - M * r - X * Z) * _;
      for (; XPending(e); u *= CS != N)
        {
          XEvent z;
          XNextEvent(e, & z);
          ++ * ((N = XLookupKeysym(& z.xkey, 0)) - IT ? N - LT ? UP - N ? & E : & J : & u : & h);
          -- * (DN - N ? N - DT ? N == RT ? & u : & W : & h : & J);
        }
      m = 15 * F / l;
      c += (I = M / l, l * H + I * M + a * X) * _;
      H = A * r + v * X - F * l + (E = 1e-1 + X * 4.9 / l, t = T * m / 32 - I * T / 24) / S;
      K = F * M + (h * 1e4 / l - (T + E * 5 * T * E) / 3e2) / S - X * d - B * A;
      a = 2.63 / l * d;
      X += (d * l - T / S * (1.9e-1 * E + a * 6.4e-1 + J / 1e3) - M * v + A * Z) * _;
      l += K * _;
      W = d;
      sprintf(f, "%5d  %3d"
              "%7d", p = l / 1.7, (C = 9e3 + O * 5.73e1) % 0550, (int) i);
      d += T * (4.5e-1 - 14 / l * X - a * 130 - J * 1.4e-1) * _ / 1.25e4 + F * _ * v;
      P = (T * (47 * I - m * 52 + E * 94 * D - t * 3.8e-1 + u * 2.1e-1 * E) / 1e2 + W * 179 * v) / 2312;
      select(p = 0, 0, 0, 0, & G);
      v -= (W * F - T * (6.3e-1 * m - I * 8.6e-2 + m * E * 19 - D * 25 - 1.1e-1 * u) / 1.07e4) * _;
      D = cos(o);
      E = sin(o);
    }
}
```

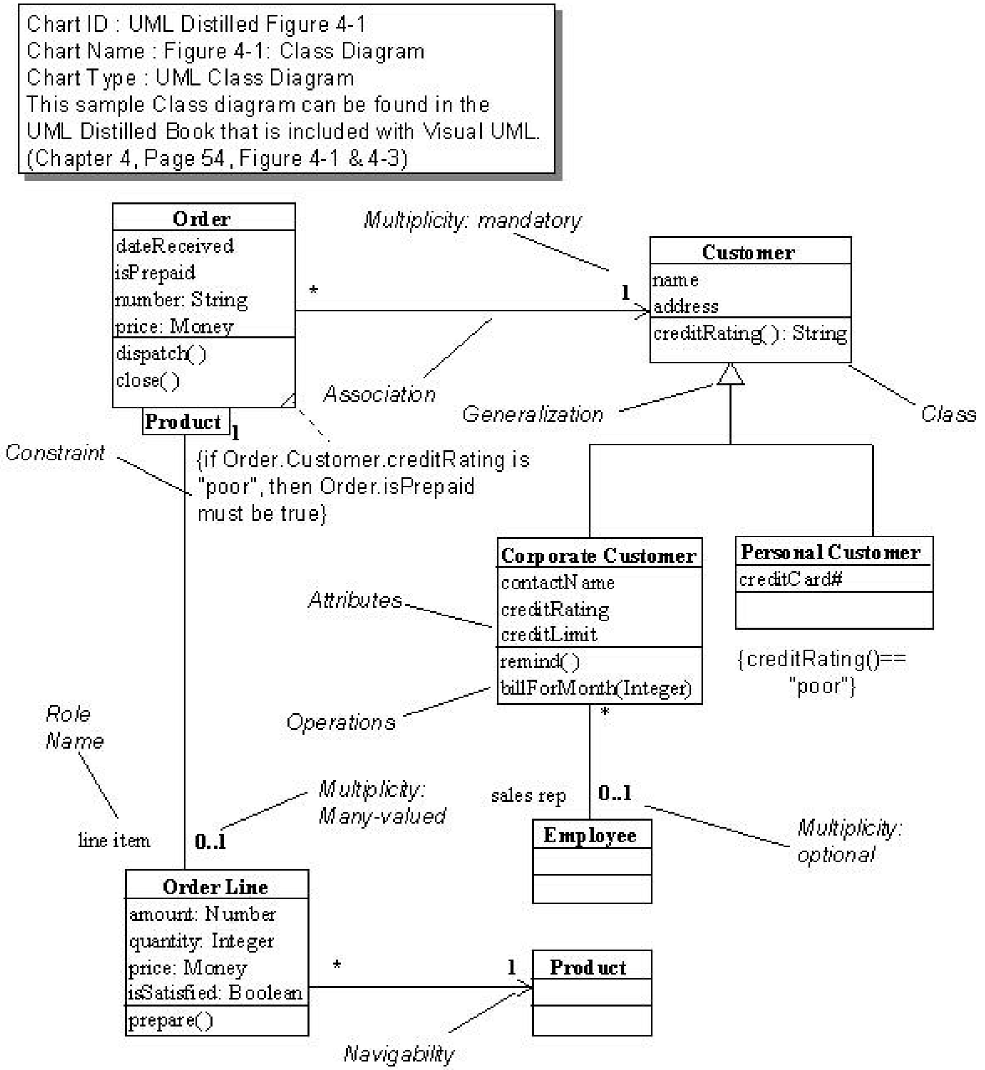
Control and Data-flow Graphs

```
s = 0.0 ;
for ( i = 1 ; i < 5 ; i ++ )
  s =   i % 2 == 1
      ? 2 * s + 0
       : s + i * i ;
```

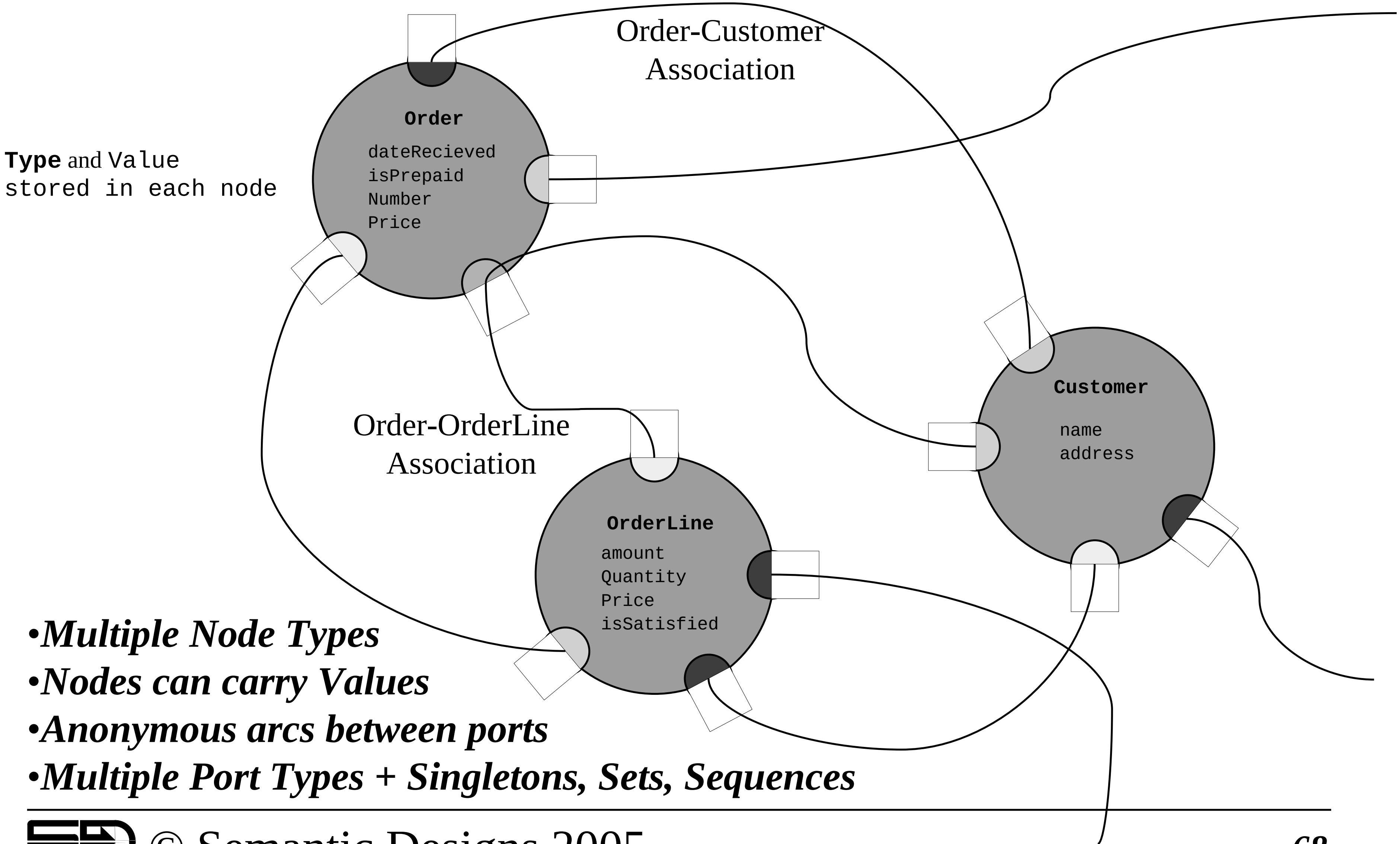




Graphical Specifications



Encode Model Instances as Graphs…



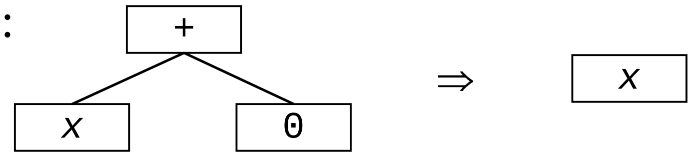
- *Multiple Node Types*
- *Nodes can carry Values*
- *Anonymous arcs between ports*
- *Multiple Port Types + Singletons, Sets, Sequences*

How to Implement Transforms

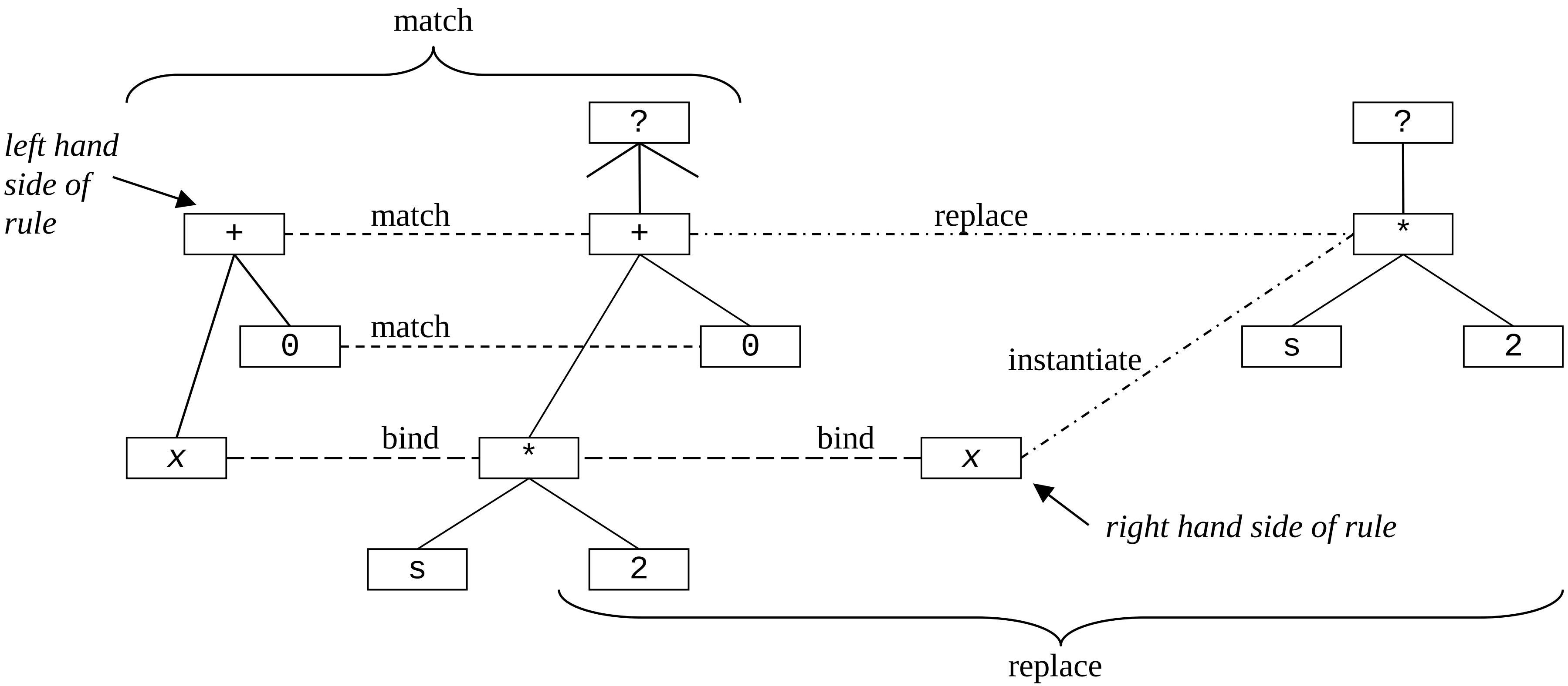
- Arbitrary Procedure
  - Full access to schema representation
  - Arbitrary navigation in and modification of schema representation
  - Fast execution
- Rewrite Rules
  - For chosen schema representation prebuild matching/replacing engine
  - Usually require precondition evaluator
  - Desired: Compilation of set of rewrite rules to efficient procedure

Rewrites on Abstract Syntax Trees

Rewrite Rule:



Action!



Specifying Rewrite Rules

- Technology:
  - Specification in source language with pattern variables + Pattern Parser

- Example:

domain name

fragments in domain syntax

rule

```
default base domain C.
rule not_false() = “! 0”      → ”1”.
rule true_and() = “1 && \e” → “\e”
rule do_while_0(s) = “do \s while(0);” → “\s”
rule for_never(i,c,m,s) = “for (\i=\c; \i<\c; \m) \s” → “”
ruleset simplify = { not_false, true_and,
                     do_while_0, for_never }.
```

set of rules

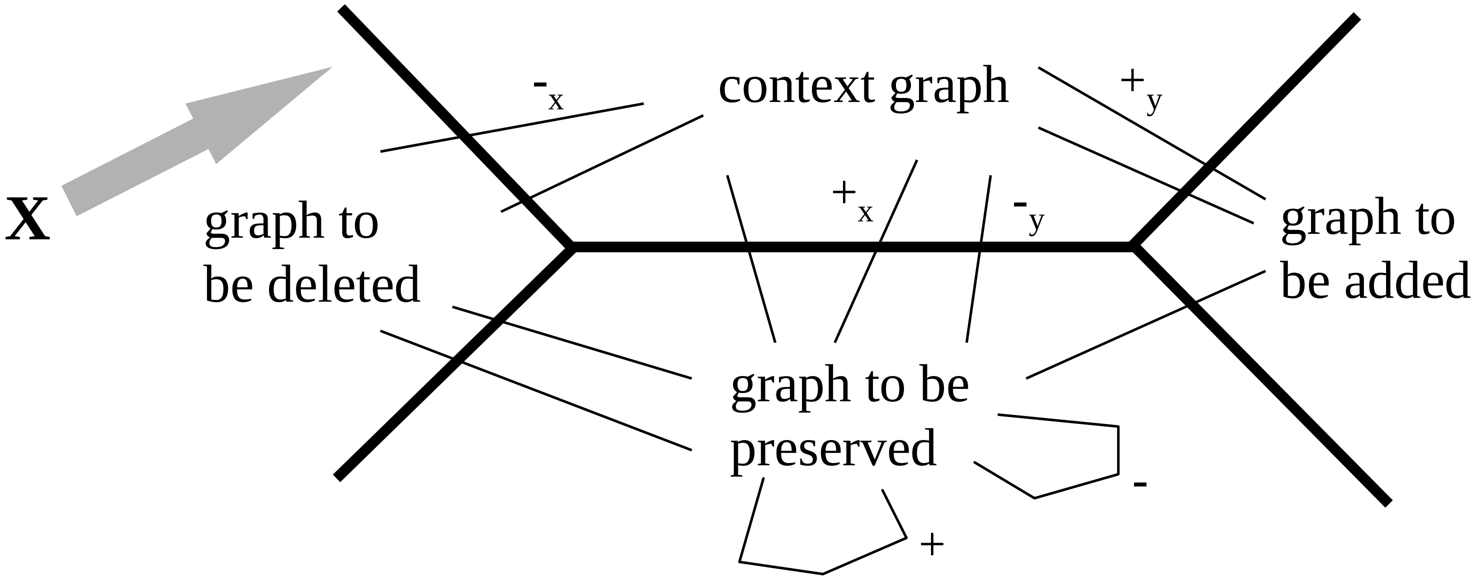
declaration of pattern variables

use of pattern variable

- Note:
  - *Rule Specification Language* Instance
    ⇒ Another language to define and implement.

Graph Rewrites

X notation defines graph rewrite with 4 parts and cross-arcs

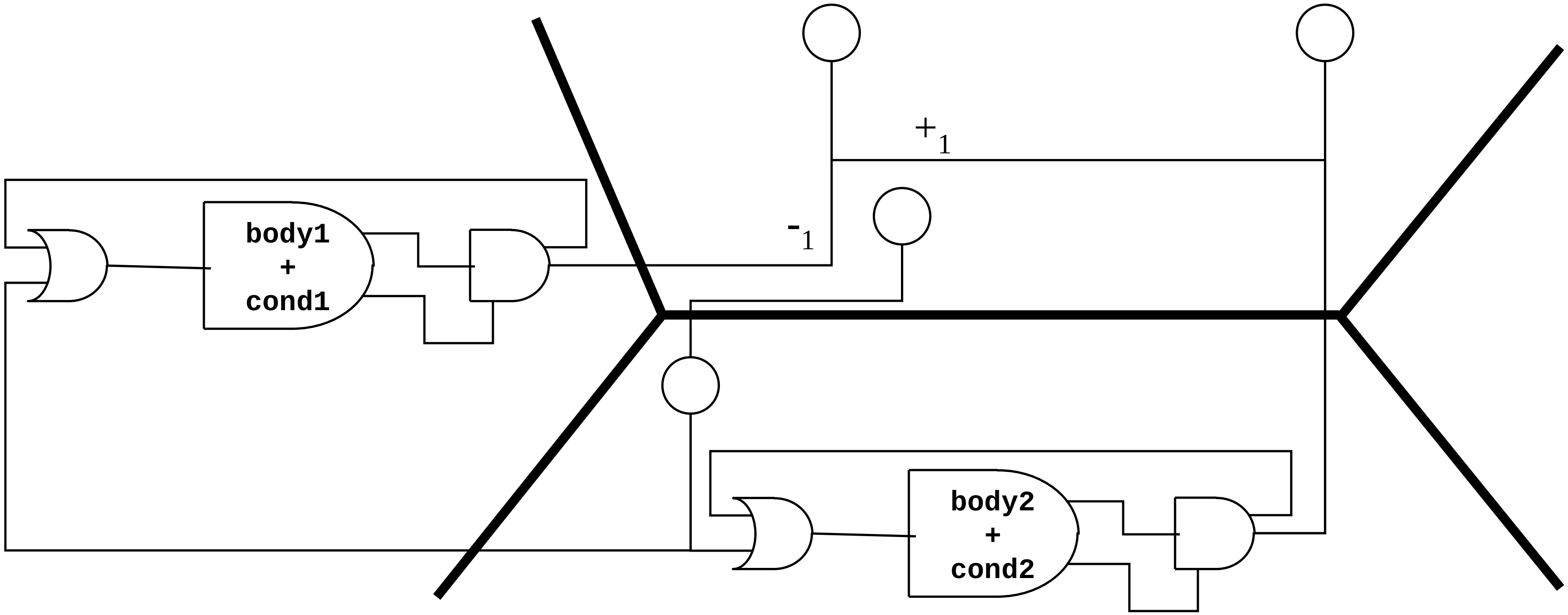


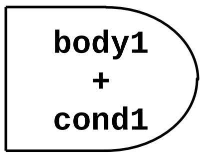
Given a graph transform as an **X**-rewrite, use by:

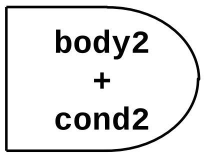
- Match to graph consisting of graph to be deleted and graph to be preserved
- Remove match to deleted graph, arcs tagged "-", "$-_z$"
- Insert added graph, arcs tagged "+", and corresponding arcs "$+_z$"

Dataflow Graph Rewrite

Eliminate Duplicate Loop



if 

= 

Difficult to code on abstract syntax trees!

…

Transformation systems further need:

- Data Bases
  - for transforms, metaprograms
  - for design history
- Development Environment for semi-automatic application of transforms
  - View specifications and transforms
  - Apply transforms
  - Undo transforms
- …
- *Lots of Languages, Transforms, Metaprograms!*

Some Transformation Systems

- Compilers
- Intentional Programming (IP)
  XML/XSLT
- TXL
- Stratego/XT
- Design Maintenance System (DMS)
- OMG QVT

Transformation System Concepts for XML/XSLT

• Domain	*XML satisfying DTD or XSchema*
• Schema	*DOM*
• Semantics	*informal*
• Performance Measure	*none*
• Performance Predicate	*informal*
• Specification	*XML document*
• State	*DOM, possible: state modified by hand-coded extension functions*
• Transform	*XSLT template with root node matching*
• Locator/locale	*pointer to tree node*
• Binding	*explicitly hand-coded variable bindings only*
• Transformation	*implicit*
• Property preservation	*informal*
• Metaprogram	*call of template set ("mode") from within template (programmed in XSLT document)*

Transformation System Components for XML/XSLT

Formal Source Language	*XML satisfying source DTD or XSchema*
Formal Target Language	*XML satisfying target DTD or XSchema*
Definition of Implementation	*informal*
A Representation for source/target instances	*DOM*
A set of Transforms	*XSLT template set ("mode")*
Tool to apply transforms	*XSLT transform engine (e.g. msxsl.exe)*
Means to apply multiple transforms	*call of template sets ("modes") from templates*
Specification Parser	*optionally validating XML parser*
Target PrettyPrinter	*DOM to text printer, other DOM renderers*
Means to display intermediate states	*none*
Means to undo transformations	*none*
Means for performing analyses	*hand-coded extension functions (XSLT1.1)*
Means for augmenting Transformation System databases	*XSLT documents*

XML/XSL
Poor Man's DSL and Code Generator?

- XML: easy way to encode ASTs for a DSL
  - "no parser/prettyprinter" needed; use DOM reader/writer
  - "No DSL syntax → Easy DSL Design/Use!" [Cleaveland00]
    - Wrong! Simply constraining DSL syntax to match XML… *users* lose
  - Use for reengineering exchange format (GXL…)?
    - 1 Million SLOC → 6 million AST nodes → 540mB … good idea?
    - Agreement on DTD/Schema across tools? [typical XML problem]
- XSL: Transforms on ASTs
  - Procedural (functional) transforms
    - Not rewrites → No "matching" → ?x + ?x -> 2*x not easy to code
    - Must build new tree; can't have partial results → scales badly
  - Often augment with "better" procedural escape…(Jscript… not fast)
  - Not clear how present implementations scale
  - Not designed to support complex semantic analyses
- Seductive but not recommended!

Why XML is wrong for Generators

- Generators Goal: enable effective generation of robust code
  - Engineer must understand abstract specification
  - Code generation must bridge abstract-to-concrete gap
  - Generated code must be correct/efficient
- XML attractive, but focuses attention in *wrong* place
  - XML = AST: => easy parsing but… *price = notational convenience*
  - Multiple abstraction levels => different DTDs; how to assemble?
  - XSLT defects:
    - Wrong *notation* to encode transforms
=> slows encoding of reliable transforms
    - Limited to local manipulation of trees
=> long-distance xforms and graph transforms are out of scope
    - Can't directly encode most rewrites: x+x=>2*x; algebraic properties
    - Functional language: will scale badly
- *Moral: Generator builders should suffer,*
so that software specifiers lead easier lives

Some Transformation Systems

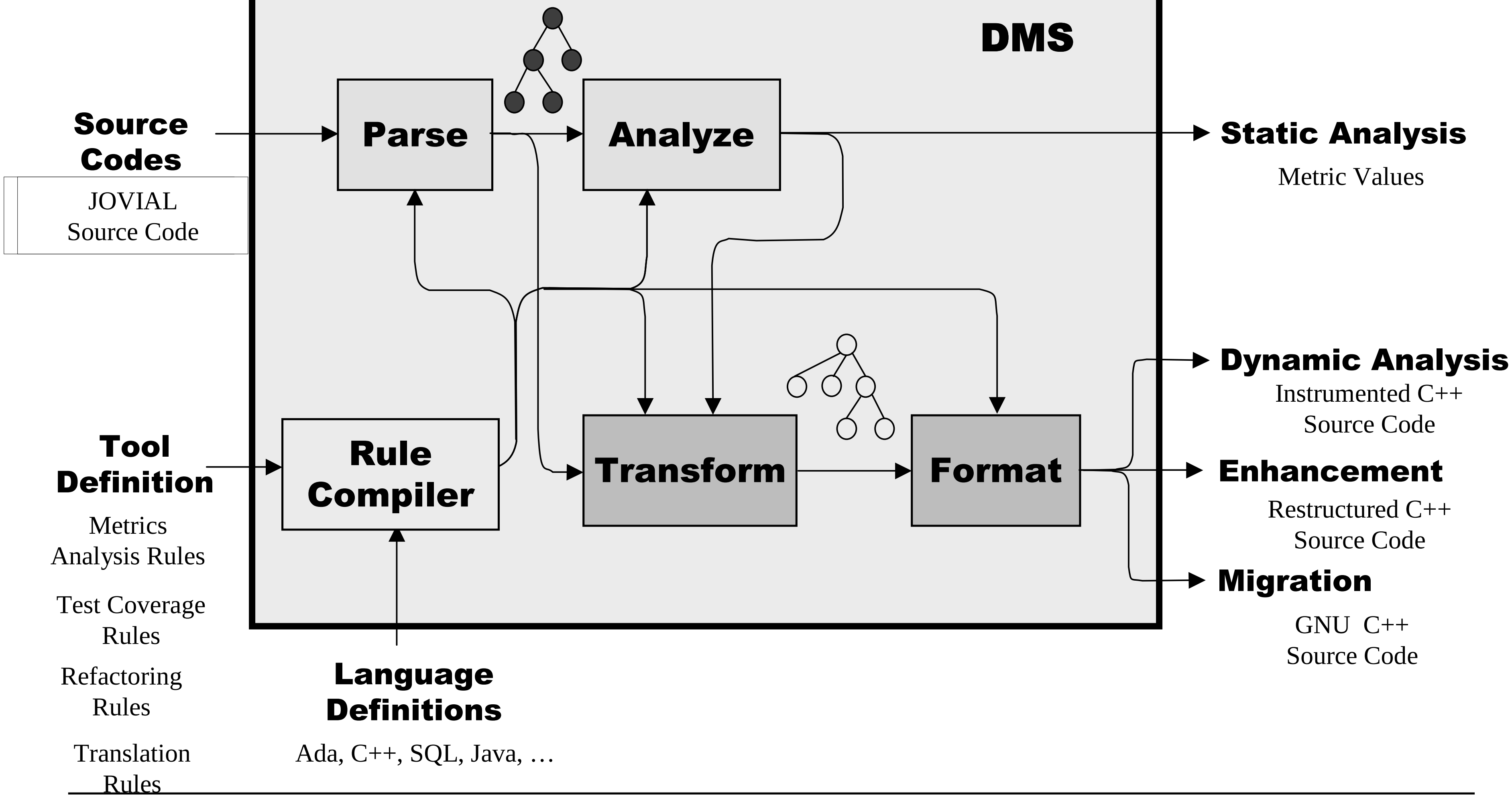
- Compilers
- Intentional Programming (IP)
- XML/XSLT
- TXL
- Stratego/XT

  Design Maintenance System (DMS)
- OMG QVT

DMS Software Reengineering Toolkit

- *Automated* source code analysis and *modification*
  - *Leverage transformation machinery needed to build DMS vision*
- Enables wide variety of SE tasks to be automated
  - *Commercial applications*
    - Source Formatters, Hyperlinked Source Browsers
    - Intellectual property protection by code obfuscation
    - Documentation extraction
    - Metrics
    - Preprocessor conditional simplification
    - Test Coverage and Profiling tools
    - Clone Detection and removal
    - XML DTD compilation
    - DSL code generation: Factory Automation
    - Migrations (JOVIAL to C)
  - *Research applications*
    - Aspect-weaving (U. Alabama Birmingham)
    - FORTRAN optimization (U. Alabama Birmingham)
    - Large-scale C++ component restructuring (SD/Boeing)
    - Code generation/quality checking for spacecraft (NASA/JPL)
    - Architecture Extraction

How DMS Works
Generalized Compiler Technology



DMS Application 1: Custom Refactoring

- Goal: Push Interface (parent) class A down into child B

- Steps
  - Parse the input files to ASTs *preserving comments and lexical formats*
  - Build symbol tables, tie to ASTs
  - Find classes A and B by looking in symbol table
  - Find special method GetInterface in the inheriting class
    - returns a pointer to the parent class.
  - Copy non-overridden members of parent to child class
  - Add base classes of the parent class to into child base class list
  - Replace calls to GetInterface with direct reference to child class
  - Remove GetInterface method from child class
  - Replace uses of the parent class by corresponding uses of child class.
  - Eliminate parent class.
  - Format the syntax tree resulting from these steps.

DMS App 1: Push Interface class A into B

```
namespace N {
  struct X {
    int xf();
  };
  struct Y: X {
    int yf();
    X *GetInterface() { //provide X interface
      return static_cast<X*>(this);
    }
  };
  struct Z {
    int zf();
  };

  class A: Y {
    int af() { }
    public:
    int ag();
  };
  int A::ag() { }

  struct B: X, public A, Z {
    int bf() { }
    int bg();
    public:
    A *GetInterface() { //provide A interface
      return static_cast<A*>(this);
    }
  };

  int B::bg() { }
  int main()
    { B *b;
      b->B::A::ag();
      A *a=b->GetInterface();
      Y *y;
      X *x=y->GetInterface();
    }
}
```

removed

preserved

revised

updated

```
namespace N
{ struct X
    { int xf();
    };
  struct Y : X
    { int yf();
      X *GetInterface()
        { //provide X interface
          return static_cast<X *>(this);
        }
    };
  struct Z
    { int zf();
    };

  class B;
  int B::ag()
    {  } // move me into child
  struct B : X, private Y, Z
    { // this is a class
      private:
      int af()
        {  }
      public:
      int ag();
      public:
      int bf()
        { }
      int bg();
    };

  int B::bg()
    { }
  int main()
    { B *b;
      b->B::ag(); // call method
      B *a = b;
      Y *y;
      X *x = y->GetInterface();
    }  }
```

DMS Application 2:

USAF B2 Bomber: Automated Legacy Migration

DMS Application 2: Automated JOVIAL to C Migration

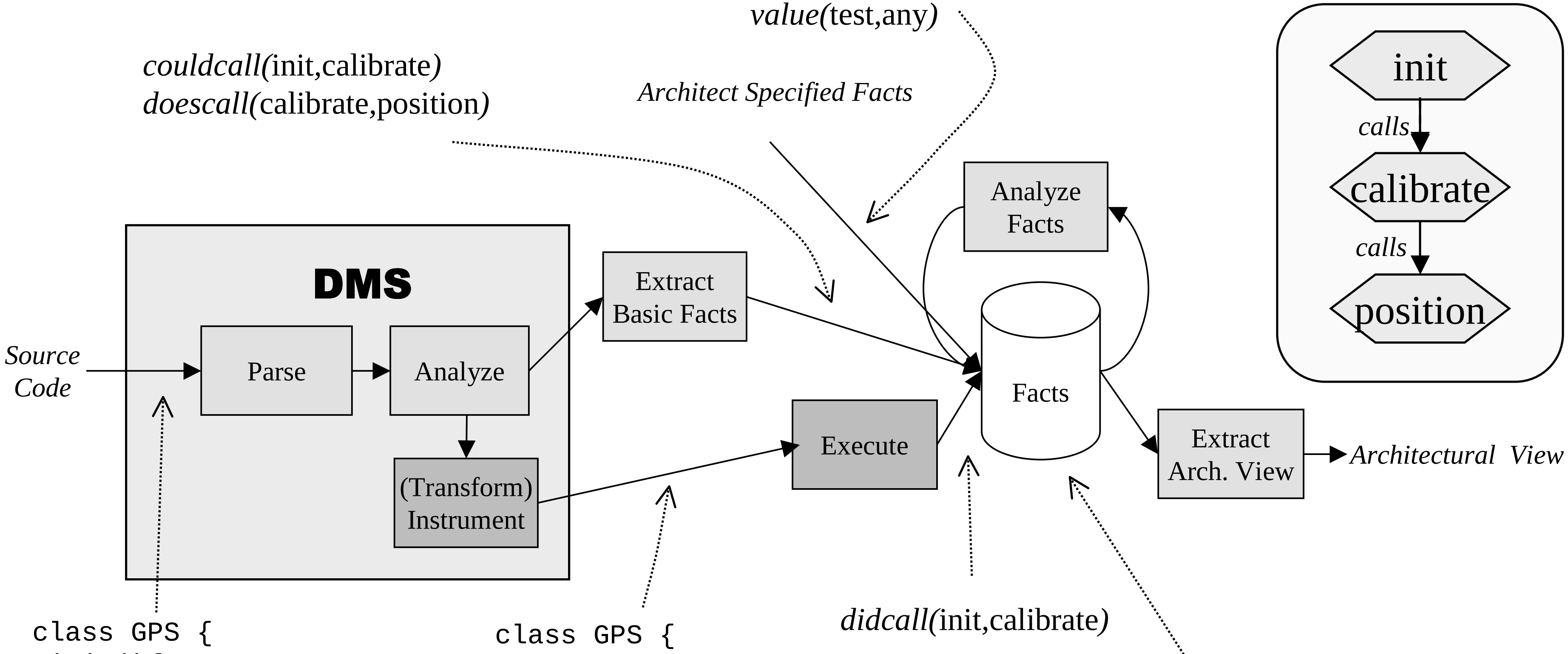
- Problem: aging 16-bit 1750 microprocessors in B2 Bomber
  - 350,000 lines of mission software in JOVIAL
  - Desperately need more memory space and speed
    - Application functionality enhancements pushing boundaries
  - No deep institutional knowledge about code details
  - 2 previous attempts, over 2 years, to migrate to PowerPC

    … *failed*…
- Solution: DMS + Semantic Designs' services
  - 12 months to implement JOVIAL translator
    - Uses DMS source-to-source transformation rules
  - 100% automated translation (some minor input edits)
  - Passes ground simulator for B2

DMS Application 3: *Multiple* Architectural Extraction for *Mixed*-language Systems

- There are many useful views of systems
- Large systems are always implemented in multiple languages
- Need to reason about/modify architecture across *all* languages
- Need to understand each architecture
- Need to relate one view to another

Architecture Extraction Using DMS Combining Static and Dynamic Analyses

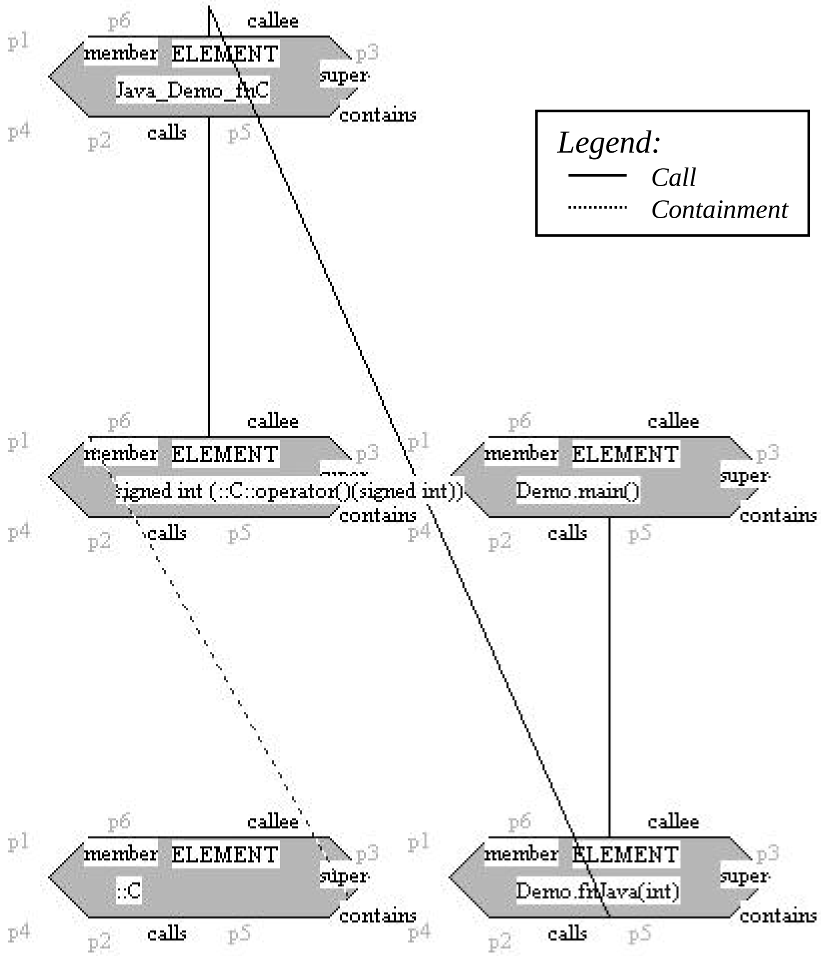



```
class GPS {
 init(){
   if (test)
    calibrate();
    …}
 calibrate(){…position()…}
 position(){…}
}
```

```
class GPS {
 init(){probe(1);
   if (test)
     calibrate();
    …}
 calibrate()(probe(2);…position()…}
 position(){probe(3);…}
}
```

*couldcall(*init,calibrate*)*
*doescall(*calibrate,position*)*
*value(*test,any*)*
*didcall(*init,calibrate*)*

Mixed-language Call Graph: Java, C++




```
public class Demo {
   int fnJava(int x) {
     return fnC(x);
   }
   native int fnC(int p);
   void main() {
     int r = fnJava(3);
   }
}
```

```
#include <jni.h>

class C {
  int operator()(int x) {
    return x+1;
  }
};

extern "C" int Java_Demo_fnC(JNIEnv *env, jobject obj, int p) {
  C c;
  return c(p);
}
```

DMS Toolkit Capabilities

- Parser/Lexer/PrettyPrinter definitions
  - Automatic AST construction from grammar
  - Multimode lexing and Ambiguous parsing
  - Preprocessor parsing, retention of comments
- Existing domains:
  - C, C++, C#, Java, Pascal, Visual Basic
  - COBOL, JCL, FORTRAN, Ada
  - SQL, Progress (4GL)
  - UML (Rational Rose MDL), Extended ER
  - HTML, XML, CORBA IDL
  - Verilog, VHDL
- ASTs using scalable hypergraph foundation
  - Full API for AST navigation/query/change
  - Cache line/node, compact sequence nodes
  - Lockable hash tables for associating values
- Analysis support
  - Attribute evaluator over AST
  - General symbol table manager
  - Scalable Clone detection/removal
  - Control flow graph construction
  - Data flow analysis framework
- SPECTRUM: Algebraic specifications
  - code simplification transforms
  - code performance predicates
  - describe abstract data
- RSL: Rule Specification Language
  - Patterns in DSL syntax
  - Patterns, rewrites, rewrite rule sets
- Term rewriting engine
  - Associative/Commutative rewrites
  - Constant folding on basic types
  - Completion algorithm
- XCL: Transform Control Language
  - Compound transformations
  - Procedures, partial order sequencing
  - Goal directed transforms
- PARLANSE: DMS System Programming Language
  - Custom analyzers/transforms
  - SMP Parallelism
  - Robust exception handling

Knowledge Capture:

DMS Domain Parts

- Syntax (domain notation)
  - External Form (what you can say: string or graphical)
  - Internal Form (How DMS stores it)
  - Parser (how to convert external form to internal form)
  - PrettyPrinter (how to display the Internal Form)
- Semantics (what the domain *means*)
  - Analyzers (how to analyze in the domain)
  - Optimizations (how to optimize in the domain)
  - Refinements (how to transform one domain to another)
  - (Attachments) (procedures to enhance DMS efficiency)

DMS "Target Language" Domains Available

- Heavily tested (parser/prettyprinters + …)
  - ANSI/GCC3/VisualC6 C: name/type resolution
  - ANSI/GCC3/VC/Managed C++: name/type resolution
  - ANSI COBOL85/IBM VSII extensions: name/type resolution
  - Java1.4: name/type resolution; Java 1.5
  - ISO Pascal, JavaScript, PHP, PL/SQL, PARLANSE
- Ready for serious use (parser/prettyprinters)
  - C#, ObjectPascal, XML, (dirty) HTML
  - Verilog, SystemVerilog, VHDL, SystemC
- On the verge of useful:
  - VBScript/VisualBasic, FORTRAN77/95, PL/1
- Available but likely incomplete
  - SQL, Spectrum, IDL, ATL, …

Parsing to Abstract Syntax Trees

A Program Representation analyzable by Computers

- Use DMS lex/grammar domains
  to define language syntax
- DMS generates lexer/parser automatically
  - Also builds *pattern parser*
- Parser reads source file(s)
  - Captures comments + lexical formats
  - Carries out lexical conversions (e.g, FP text -> IEEE binary fp)
  - Builds Abstract Syntax Tree
  - Records Position of every node (file, line, col)

Transforms in DMS

- Arbitrary Procedure
  - Full access to schema representation
  - Arbitrary navigation in and modification of schema representation
  - Fast execution
- Rewrite Rules
  - For chosen schema representation prebuild matching/replacing engine
  - Usually require precondition evaluator
  - Desired: Compilation of set of rewrite rules to efficient procedure
- *DMS Does Both*
  - Rewrite Rules via Source-to-Source Transforms and Patterns
  - Arbitrary procedures via ASTInterface and PARLANSE
  - Both types can build on each other

RSL Capabilities

- Named pattern and rule specification
  - for arbitrary domains/mixtures of domains
  - source-level syntax with parameters
  - patterns and rules accessible by PARLANSE
- Packaging of rules into rule sets
- Conditional rewriting
  - External conditions
  - Matching of subterms
  - Equivalence of subterms under rewriting
- Associative & Associative/Commutative rewriting
  - Enables rewriting for sets and lists (sequence constructs)

Optimization transform
for DMS Rewrite Rule Language

```
default base domain Java;
```

Domain Name

```
rule merge-ifs(\condition1:expression,
               \condition2:expression,
               \then-statements:statement_sequence)
              :statement->statement
   “if (\condition1)
       if (\condition2)
          { \then-statements
          }
   ”
   rewrites to
   “if (\condition1 && \condition2)
       { \then-statements }   ”;
```

Domain Syntax

Conditionality on computed attributes possible

Refinement transform
for DMS Rewrite Rule Language

```
default source domain Transact;
default target domain COBOL;

rule translate-ifs(\data_item_target:identifier,
                   \data_item_source:identifier) =
    “PUT \data_item_target ','
         'LIST' '=' \data_item_source ';' ”
   rewrites to
   “ MOVE \data_item_source TO \data_item_target.”;
```

Source Domain

Target Domain

DMS Pattern
Used to assemble or match code fragments

```
default source domain C;

pattern main_program(s:statement,v:identifier)
           :compilation_unit
   “main ()
      {  int \v;
         \v = 0;
         \s
         cout << \v;
      }".
```

Heavily used to provide target structure of transformed code

Transformation System Concepts: DMS/SRT

- Domain — *arbitrary string or graph language*
- Schema — *hypergraph with domain-imposed topology*
- Semantics — *informal*
- Performance Measure — *informal or attribute evaluator computation*
- Performance Predicate — *same as above*
- Specification — *domain instance + performance predicate*
- State — *schema + cached analyses*
- Transform — *domain-specific procedures + tree rewrites*
- Locator/locale — *domain schema/topology-specific*
- Binding — *substitution for rewrites*
- Transformation — *explicit data structure holding <transform,locale,bindings>*
- Property preservation — *requirement for functional preservation*
- Metaprogram — *domain specific procedures, transform control language*

Transformation System Components: DMS/SRT

- Formal Source Language *domain syntax*
- Formal Target Language *domain syntax*
- Definition of Implementation *preservation of functionality*
- A Representation for source/target instances *hypergraph*
- A set of Transforms *explicitly defined per domain*
- Tool to apply transforms *procedures, term/graph rewrite engines*
- Means to apply multiple transforms *rewrite sets, transform control language*
- Specification Parser *generalized LR parser*
- Target PrettyPrinter *box specification + box layout algorithm*
- Means to display intermediate states *AST:PrintTree*
- Means to undo transformations *none*
- Means for performing analyses *procedures, attribute grammars, rewrites*
- Means for augmenting Transformation System databases *domain definitions*

Some Transformation Systems

- Compilers
- Intentional Programming (IP)
- XML/XSLT
- TXL
- Stratego/XT
- Design Maintenance System (DMS)

  OMG QVT

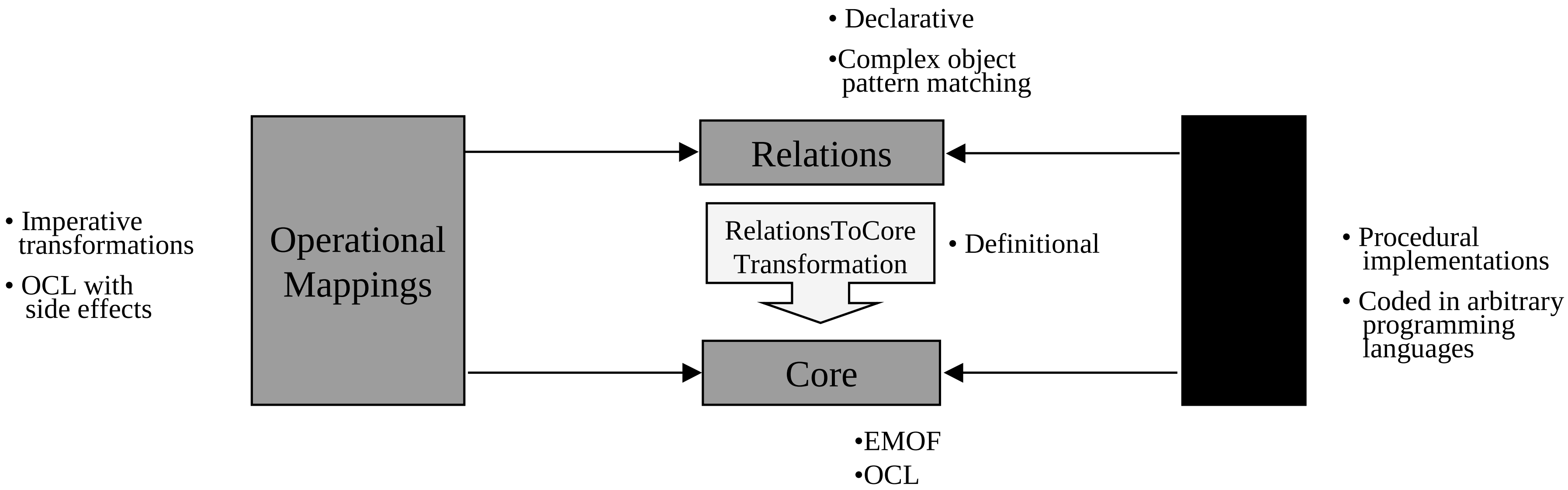
OMG Query/View/Transformation: QVT

Transformations between OMG models

- Purpose: define a model-based Query and Transformation language
  - For metamodel instances defined with MOF 2.0: "graphs"
  - Arbitrary applications to any OMG-defined models
  - QVT:
    - Queries: find subgraphs that match a structure
    - Views: transform a graph into a the same type of graph
    - Transforms: transform a set of graphs of arbitrary type to another set
      - Obvious application: PIM to PSM source to target model transforms
  - Text and Graphical presentations of transforms
  - Declarative, Imperative, and BlackBox specification of transforms
    - Declarative and Imperative versions have Textual languages
    - Declarative version has Graphical language
  - Exchangeable transform representations: Text and XMI versions
- OMG "FAX" recommendation vote in progress

QVT Architecture

- *Relations Language*: Declarative transformations as relations:
- *Core Language*: Core transforms as "virtual machine"
- *Operational Mappings*: Explicit procedural transform language
- *BlockBox*: Access to transforms implemented in other languages



Relations Language
Used to express Relational Transforms

- Key ideas
  - Declarative transforms as a set of *relations* between 2 or more models
  - Relations are defined in terms of *domains*: a set of bindings in a model
    - Terminology glitch: *domain* means "syntax and semantics" for Draco/DMS!
  - *Enforcing* a domain causes missing model elements to be created
  - Text and Graphical presentations of transforms

Relational Transforms:

Text and Graphical Syntax

- Example: Class to Relational Table Relation
- Encodes relations over two models: UML and RDBMS

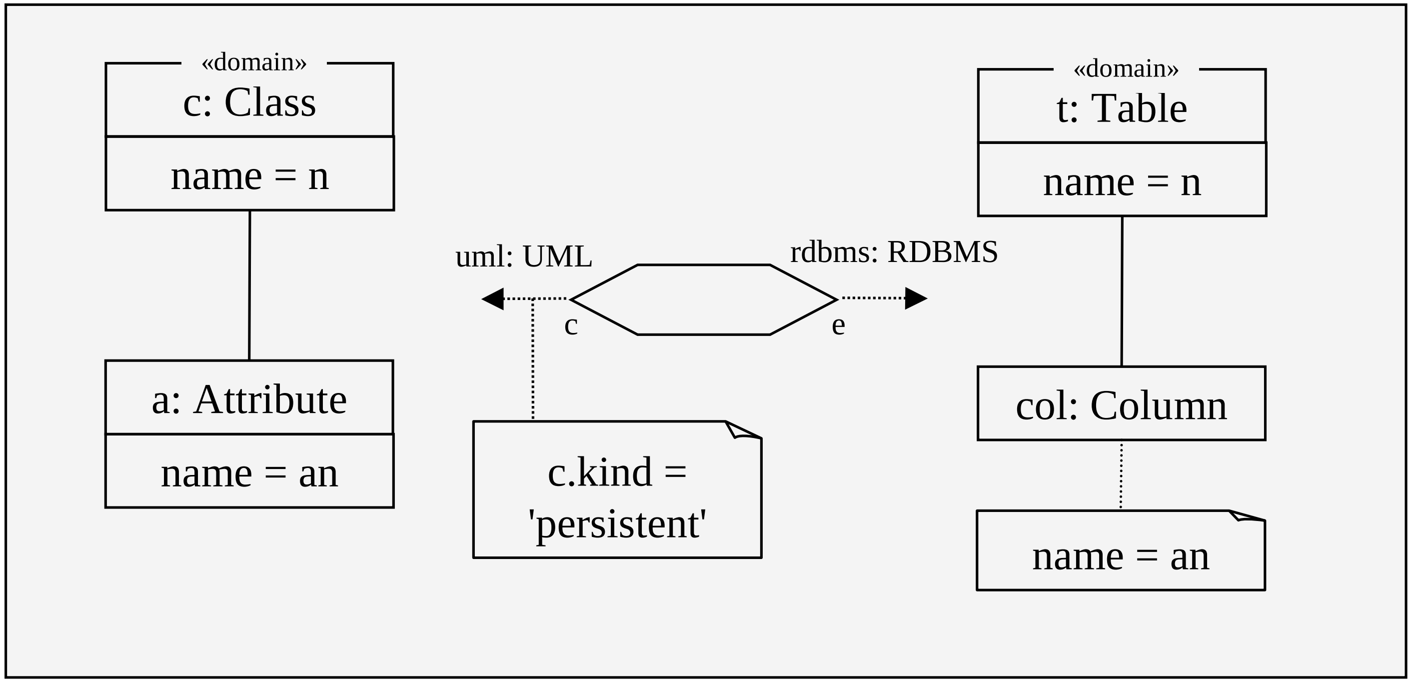
Graphical equivalent

object template expression

Text equivalent:

```
Transformation umlRDBMS(uml: UML, rdbms: RDBMS)
{  top relation UML2REL {
      checkonly domain uml c:Class {name = n, attribute = a:Attribute{name = an},
                                    kind = 'persistent'}
      enforce domain rdbms t:Table { name = n, column = col:Column{name = an}}
} }
```

UML2REL



The Core Language

A simpler language used to define the Relational Language

```
Pattern
{ c1: Class, a1: Attribute, k1: Kind
l1: attrs(c1,a1)
l2: attrs(c1,k1)
where c1.name = 'n'
and a1.name = 'an'
and k1.kind = 'persistent'
}
```

Operational Mapping for UML to RDMS

(QVT definition Appendix A.2.3)

```
-- declaring the transform
modeltype UML uses SimpleUml("omg.qvt-samples.SimpleUML");
modeltype RDBMS uses SimpleRdbms("omg.qvt-samples.SimpleRDBMS");
transformation Uml2Rdb(in srcModel:UML,out RDBMS);

-- defining specific helpers and derived properties
Query UML::Association.isPersistent() =
   (self.source.kind='persistent' and self.destination.kind='persistent');
intermediate property UML::Class:leafAttributes : Sequence(LeafAttribute);

-- defining intermediate data to reference leaf attributes that may
-- appear when struct data types are used
intermediate calss UML::LeafAttribute {
   name:String; kind:String; attr:Attribute;
};

-- defining the default entry point for the module
-- first the tables are created from classes, then the tables are
-- updated with foreign keys implied by the associations
main() {
  srcModel.objects()[#Class]->map class2table(); -- first pass
  srcModel.objects()[#Association]->map asso2table(); -- second pass
}

-- maps a class to a table, with a column per flattened leaf attribute
mapping Class:class2table () : Table
   when { self.kind = 'persistent'; }
{ init { -- performs any needed initialization
    self.leafAttributes := self.attribute->attr2LeafAttrs();
   }
  -- population section for the table
  name := 't_' + self.name;
  column := self.leafAttributes->map LeafAttr2OrdinaryColumn();
  key := object Key { -- nested population section for a 'Key'
      name := 'k_' + self.name; column := t.column[kind='primary'];
   };
}
```

A scripting language using OCL…

```
-- mapping that creates the intermediate leaf attributes data
mapping Attribute::atttr2LeafAttrs (in prefix:String="", in pkind:String="")
   :Sequence(LeafAttribute) {
  init {
    var k := if pkind = "" then self.kind else pkind endif;
    result :=
       if self.type.isKindOf(PrimitiveDataType)
       then -- creates a sequence with a LeafAttribute instance
           Sequence {
              object LeafAttribute (attr:=self; name:=prefix+self.name;kind:=k; }
       else self.type.attribute->map attr2LeafAttrs(self.name+"_",k)
       endif;
   }
}

-- mapping that creates an ordinary column from a leaf attribute
mapping LeafAttribute::leafAttr2OrdinaryColumn (in prefix::String=""): Column {
  name := prefix+self.name;
  kind := self.kind;
  type := if self.attr.type.name='int' then 'NUMBER' else 'VARCHAR' endif ;
}

-- mapping to update a Table with new columns of foreign keys
mapping Association::assoc2table(): Table
  when {self.isPersistent();}
{ init { result := self.destination.resolveone(#Table); }
  foreignKey := self.map assoc2ForeighKey();
  column := result.foreightKey.column;
}

-- mapping to build the foreign keys
mapping Association::asso2ForeignKey {
   name := 'f_' + name;
   refersTo := self.source.resolveone(Table).key;
   column := self.source.leafAttributes[kind='primary']
                -> map leafAttr2ForeighColumn(source.name+'_');
}

-- Mapping to create a Foreign Key from a leaf attribute
-- Inheriting of leafAttr2OrdinaryColumn has the effect to call the
-- inherited rule before entering the property population section
mapping LeafAttrbute::leafAttr22ForeignColumn ( in prefix:String) : Column
  inherits leafAttr2OrdinaryColumn {
   kind := "foreign";
}
```

QVT Observations

- QVT sublanguages
  - Relational Transforms seem reasonable
    - Perhaps OK for some model-to-model transforms
    - Can't express all possible functions of models
      - *Will need black-box transforms to anything practical*
  - Core language
    - Only used for definition; unlikely used in practice
  - Operational Mapping language… *a really bad idea*
    - Likely to be incomplete
    - Not likely to be well defined
- What to use for MDA (model-to-code transforms)?
  - QVT doesn't produce text!
  - Proposed OMG Model to Text RFP… next draft in November, 2005
    - Text generation by use of templates selected by QVT queriers
- Can't use for ADM (code-to-models)
  - QVT doesn't read text!… use OMG proposed ASTM as input?
- As a not-quite-ready standard with no practical experience…
  - *QVT not ready for prime time yet*

Transformation System Concepts: QVT

- Domain *OMG MOF-defined models*
- Schema *not specified (implementation dependent)*
- Semantics *informal*
- Performance Measure *none*
- Performance Predicate *none*
- Specification *OMG MOF model instance set*
- State *same as above*
- Transform *nonprocedural: relations + constraints; procedural: operational mappings language = OCL + side effects*
- Locator/locale *not clearly defined by QVT docs*
- Binding *variable tied to object, association, or value*
- Transformation *trace models*
- Property preservation *informal preservation of model-specific property*
- Metaprogram *operational mappings language control constructs* + `access transformation`

Transformation System Components: QVT

- Formal Source Language *MOF models*
- Formal Target Language *MOF models*
- Definition of Implementation *not defined; informal property preservation*
- A Representation for source/target instances *not defined by QVT*
- A set of Transforms *sequence of declarations in text relational and operational languages*
- Tool to apply transforms *not defined by QVT standard*
- Means to apply multiple transforms *operational mapping language only*
- Specification Parser *not defined by QVT*
- Target PrettyPrinter *not defined by QVT*
- Means to display intermediate states *not defined by QVT*
- Means to undo transformations *not defined by QVT; trace models may help*
- Means for performing analyses *not defined by QVT*
- Means for augmenting Transformation System databases *read text/XMI*

Program and Model Transformation Systems

- Transformation technology maturing
  - Practical value ***now*** for
    - Massive change
    - High quality code generation
- Need generalized compiler-like infrastructure
  - Definable parsers, prettyprinters, transforms, analyzers
  - Need symbol tables for serious applications
  - Must *scale* to application systems with MSLOCs
  - ***Amortizes tool costs over many custom tools***
- ***String and Graph language tools unifiable***
  - ***Same infrastructure can serve both***
  - ***Huge synergy by combining multiple languages and notations***

Where to Learn More

- [Aho/Sethi/Ullman89] *Compilers: Principles, Techniques and Tools*
  - Must-have basic background for compiler theory used in transformation systems
- [Neighbors84] *Draco Approach to Constructing Software*
  - Domain-specific transformations
- [Feather86] *A Survey of Transformation Techniques*
  - Survey of Transformation Systems and Methods
- [Turski86] *The Specification of Computer Programs*
  - Theory of specification and implementation
- [Partsch90] *Specification and Transformation of Programs*
  - Excellent overview of transforms and wide-spectrum languages
- [Smith90] *KIDS: A Semi-Automatic Development System*
  - Advanced transformation system
- [Waters92] *Programmer's Apprentice*
  - Transformational Design Recovery and Modification
- [Jones93] *Partial Evaluation and Automatic Program Generation*
- www.program-transformation.org
- www.semdesigns.com/Company/Publications/TransformBib.pdf

Semantic Designs

21st Century Software Engineering

- Akers, R., Baxter, I., and Mehlich, M. 2004. *Re-Engineering C++ Components Via Automatic Program Transformation*. Invited paper, Partial Evaluation and Program Manipulation Workshop, IEEE Press.
- Baxter, I. 1992. *Design Maintenance Systems*. Communications of the ACM 35(4), April, 1992, Association for Computing Machinery, New York.
- Baxter, I. and Mehlich, M. 1997. *Reverse Engineering is Reverse Forward Engineering*. Working Conference on Reverse Engineering, 1997, IEEE.
- Baxter, I, 2001. *Branch Coverage for Arbitrary Languages Made Easy: Transformation Systems to The Rescue!* International Workshop on Automated Program Analysis, Testing, and Verification, IEEE Press.
- Baxter, I. and Pidgeon, C. 1997. *Software Change Through Design Maintenance*. Proceedings of the International Conference on Software Maintenance ‘97, 1997, IEEE Press.
- Baxter, I., Yahin, A., Moura, L., Sant‘Anna, M., and Bier, L. 1998. *Clone Detection Using Abstract Syntax Trees*. Proceedings of the International Conference on Software Maintenance ‘98, 1998, IEEE Press.

⇒ **Baxter, I., Pidgeon, P., and Mehlich, M. 2004. *DMS: Program Transformations for Practical Scalable Software Evolution*. Proceedings of the International Conference on Software Engineering, 2004, IEEE Press.**

- Mehlich, M. and Baxter, I. 1997. *Mechanical Tool Support for High Integrity Software Development*. Proceedings of Conference on High Integrity Systems ‘97, 1997, IEEE Press.
- Pidgeon, C. 1997. *Organizing and Enabling Domain Engineering to Facilitate Software Maintenance*. Proceedings of The Eighth Annual Workshop on Software Reuse, IEEE.

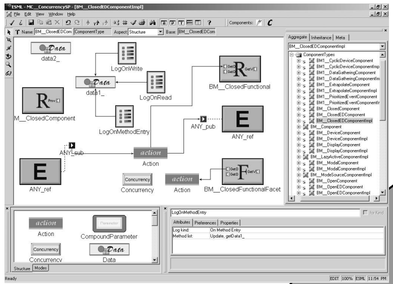
Outline of Section 2

Key Characteristics

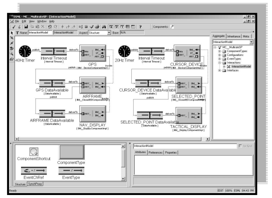
Legacy Evolution

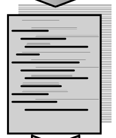
Model-Driven

Software Transformation

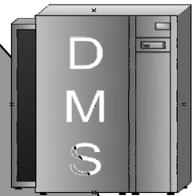


Motivation





DMS Xform Rules



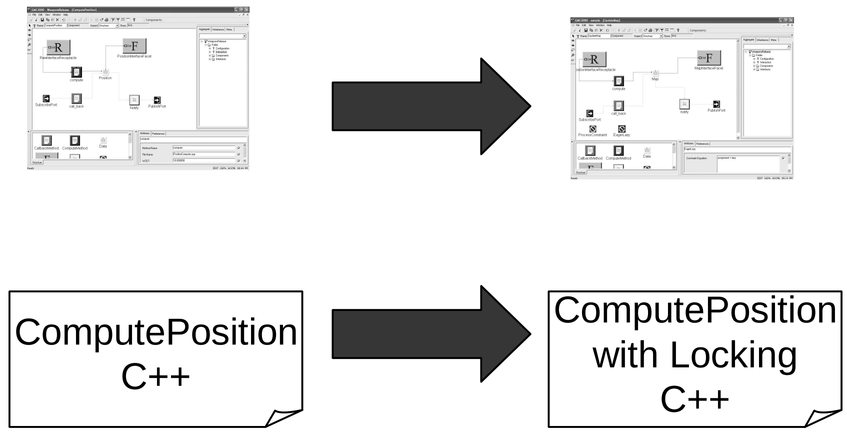
Model-Driven Program Transformation

Case Study and Video Example

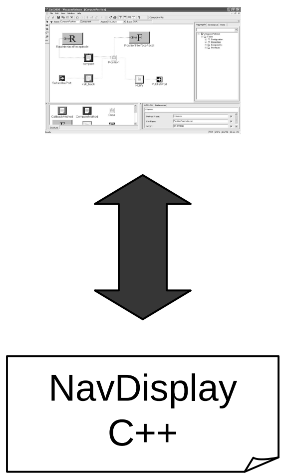
Two-Level Aspect Weaving

Two-Dimensions of Transformation/Translation

- **Horizontal transformation**
  - <u>Transformation</u> within the *same* level of abstraction
  - E.g., Model transformation, code refactoring

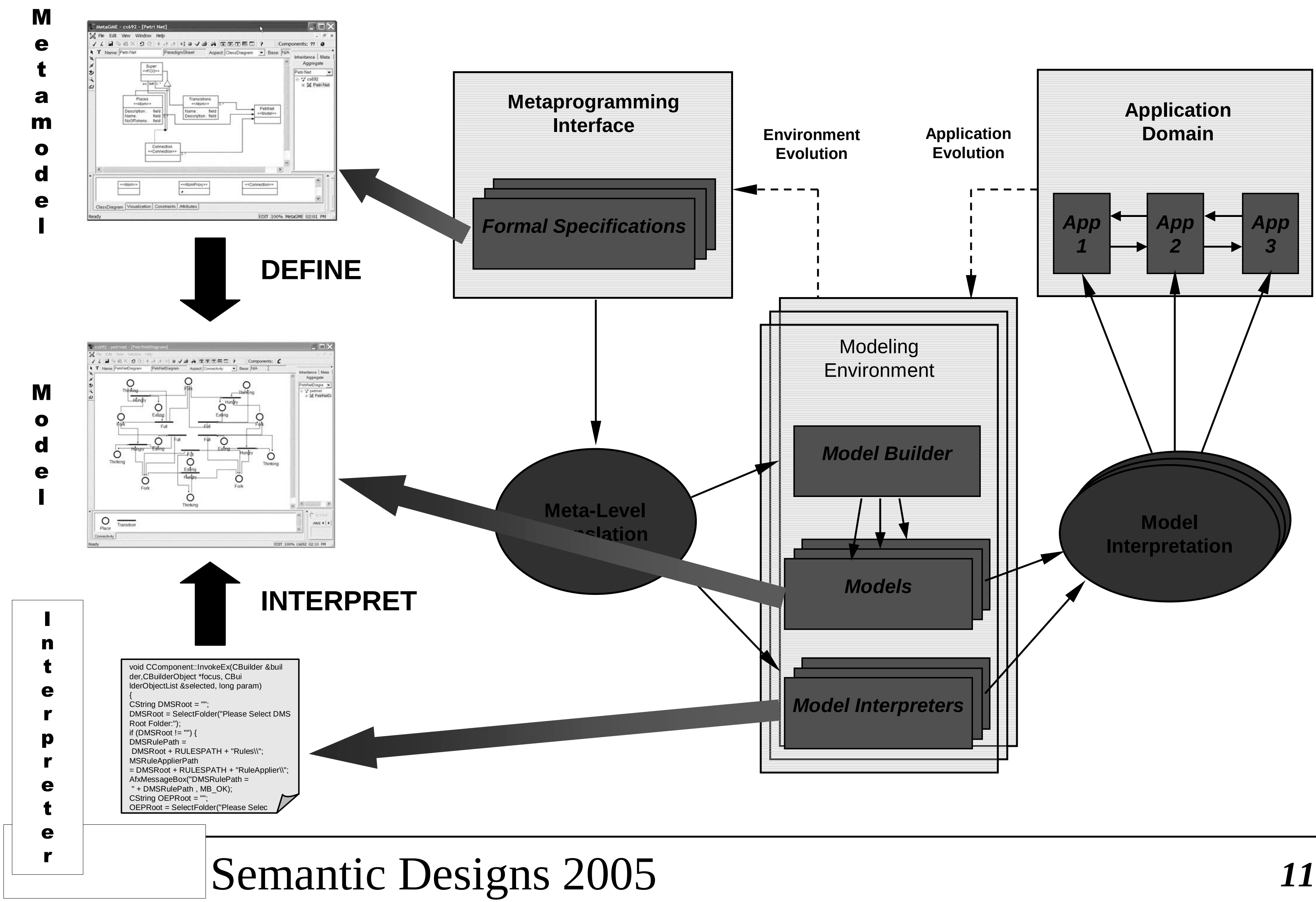


- **Vertical translation**
  - <u>Translation</u>, or synthesis, *between* layers of abstraction
  - E.g., Model interpreters, reverse engineering



Vertical transformation needed!!!

Background: Model Integrated Computing (MIC)

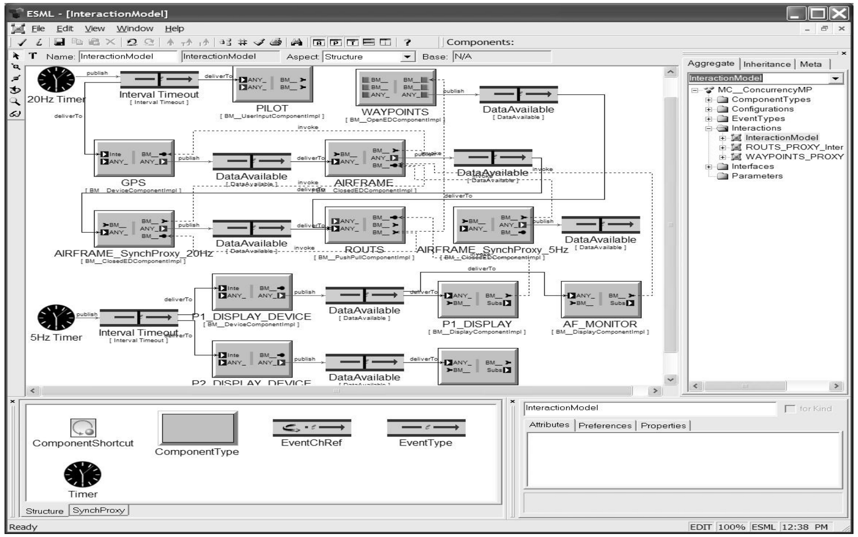


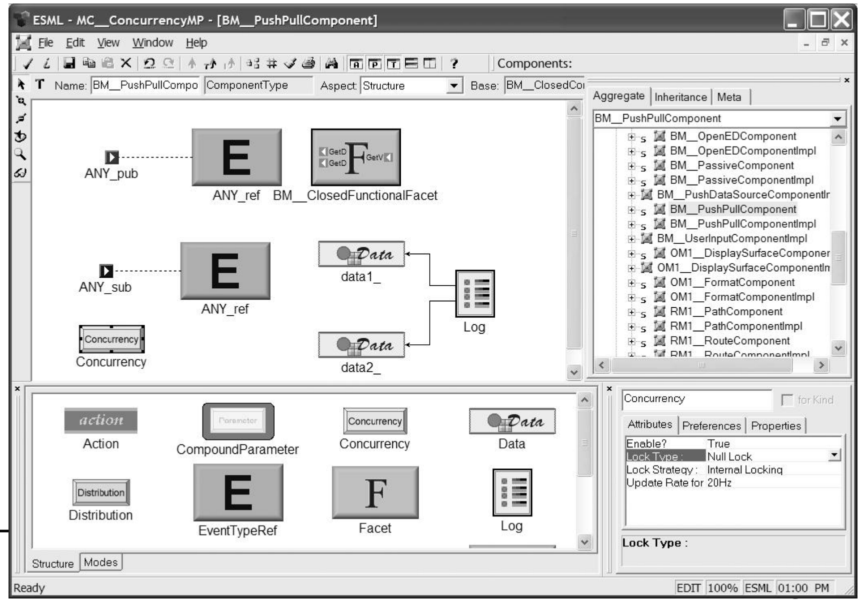
Case Study: Bold Stroke Product Line

- Background Context
  - Mission-control software for Boeing military aircraft (e.g., F-18 E/F, Harrier, UCAV) under development since 1995
  - CORBA event-based systems
  - Thousands of components implemented in over a million lines of C++ code
- Key Challenges
  - Difficult to evolve the underlying source representation to address new requirements; impossible to determine, *a priori*, all of the future adaptation requests
  - Difficult to migrate the source representation to newer component models

Embedded Systems Modeling Language

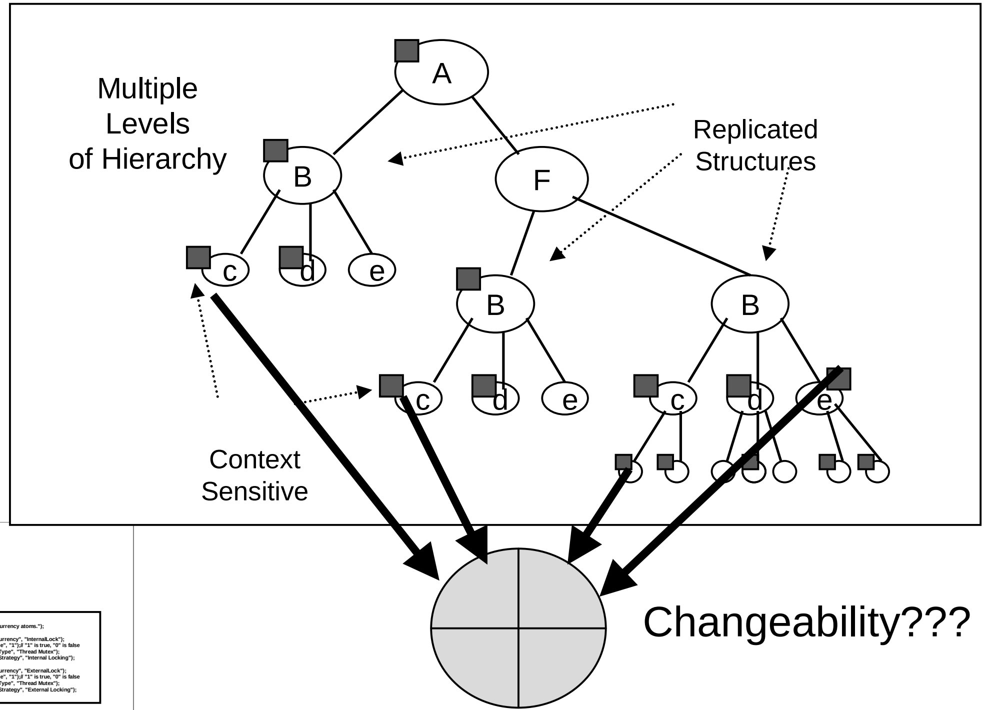
- **ESML**
  - Developed at Vanderbilt/ISIS as a DSML for GME; Large models of Bold Stroke in ESML
  - Multiple views of an embedded system: Components; Component Interaction; Component Configuration
  - Captures the interactions among components via an event channel. System timers and their frequencies are also specified
  - Integration with other tools to provide simulation and model checking.
  - Model interpreters generate various artifacts from models (e.g., XML configuration files that are loaded by Bold Stroke at startup)





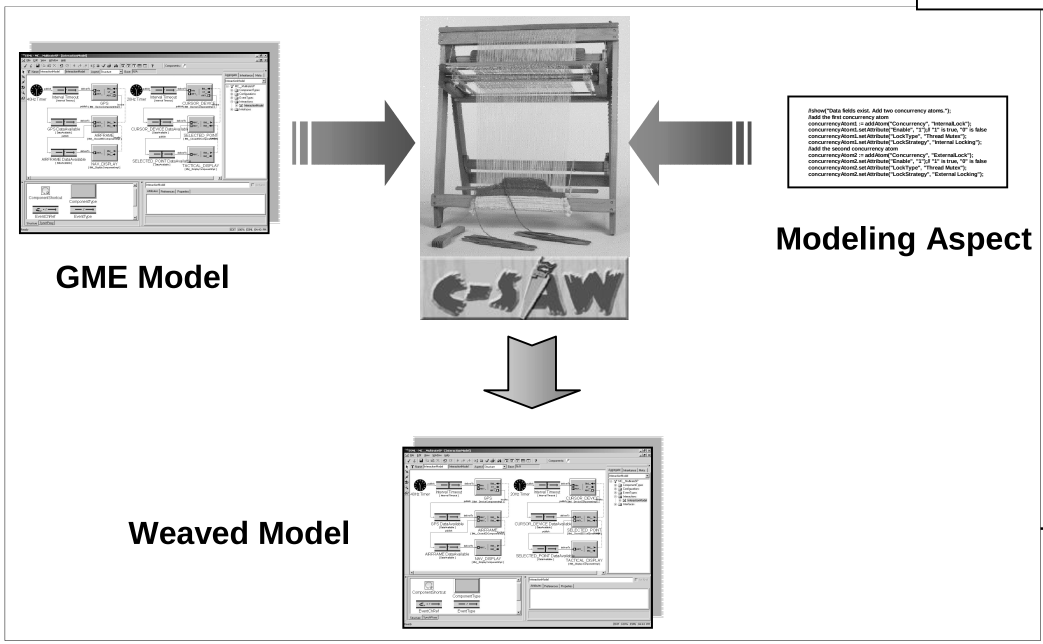
Key Challenge 1:
Crosscutting Constraints in Real-Time/Embedded Models

Crosscutting Constraints



<u>Challenge</u>: Crosscutting in Models

- Base models become constrained to capture a particular design
- Concerns that are related to some global property are dispersed across the model

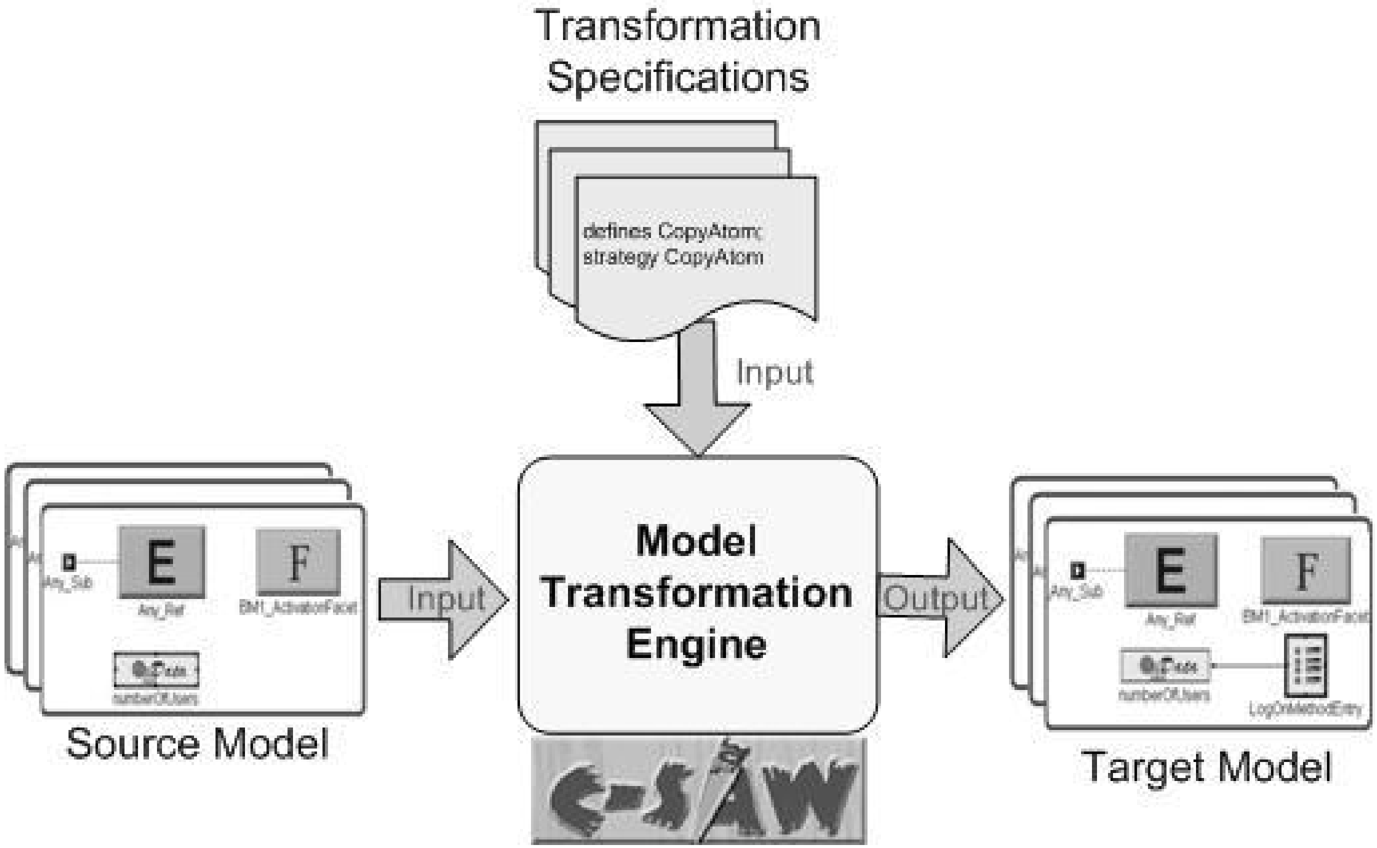


<u>Solution</u>: Model Weaving

- C-SAW is an aspect-oriented weaver at the modeling level

C-SAW: A Model Transformation Engine

-- A model transformation engine assists in the rapid adaptation and evolution of models by weaving additive changes into models.



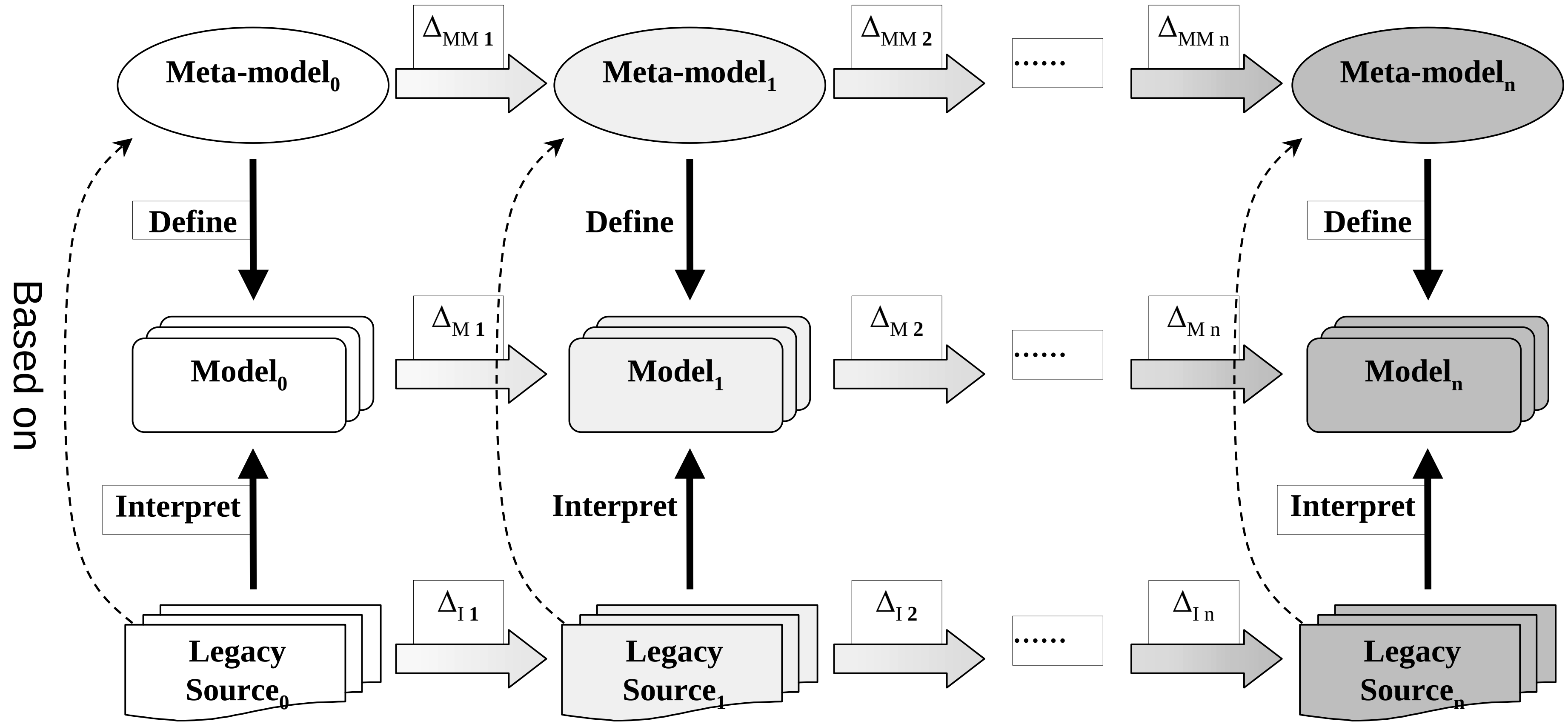
Visualization of changed ESML model

- Requirement: Record the state information of the aircraft according to polices defined in a model
- Under different stages of development, a concern may need to be adjusted based on different contexts e.g., testing on the ground vs. in-flight recording
- Ability to rapidly explore design alternatives representing different policies

Model Transformation to Add LogOnExit (C-SAW)

```
strategy logDataAtoms()
{
  atoms()->select(a | a.kindOf() == "Data")->AddLog();
}
strategy AddLog()
{
  declare parentModel : model;
  declare dataAtom, logAtom : atom;
  dataAtom := self;
  parentModel := parent();
  logAtom:= parentModel.addAtom("Log", "LogOnMethodExit");
  logAtom.setAttribute("Kind", "On Method Exit");
  logAtom.setAttribute("MethodList", "Update");
  parentModel.addConnection("AddLog", logAtom, dataAtom);
}
aspect Start()
{
  rootFolder().findFolder("ComponentTypes").models().
       select(m|m.name().endWith("Impl"))->logDataAtoms();
}
```

Key Challenge 2:
Evolution of legacy models and code

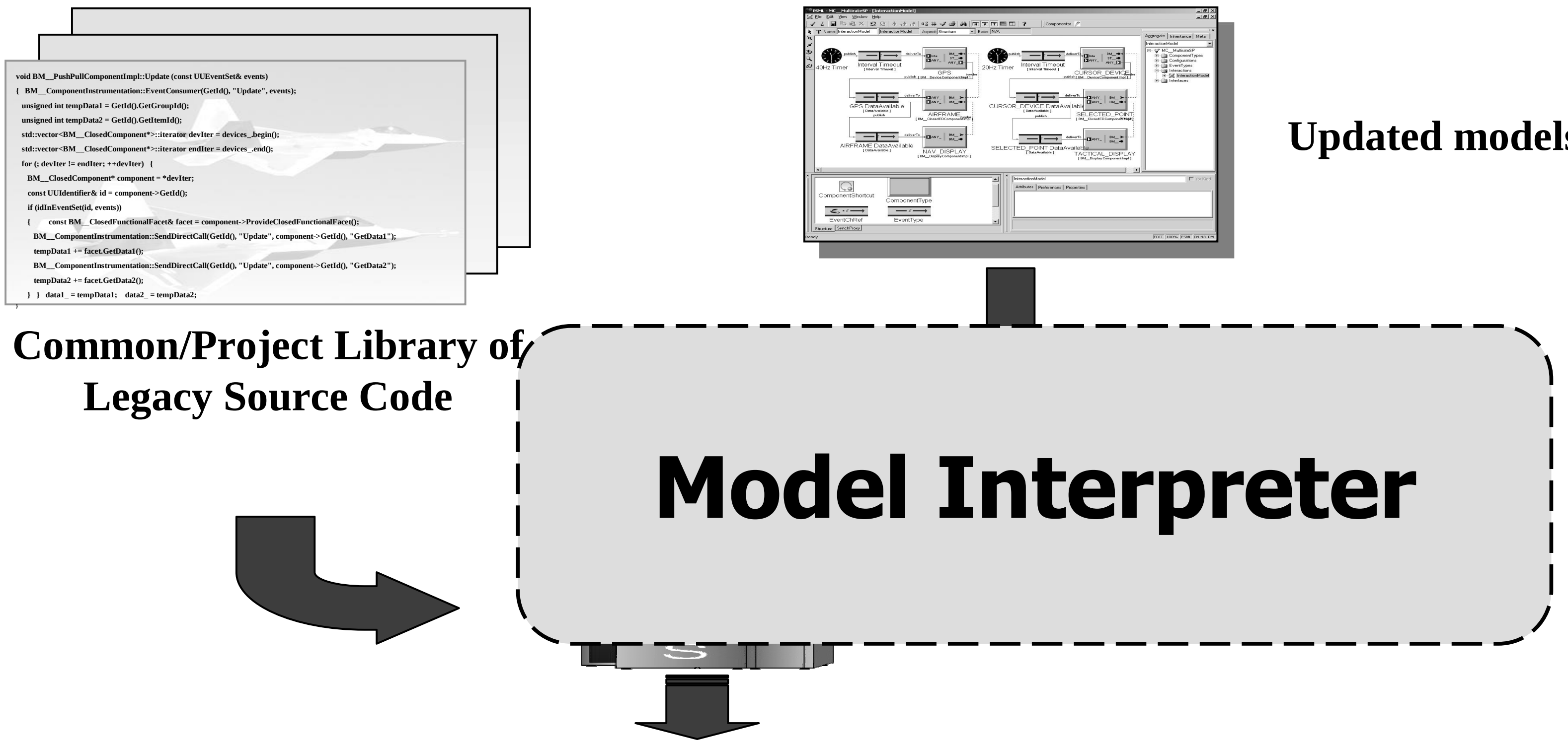


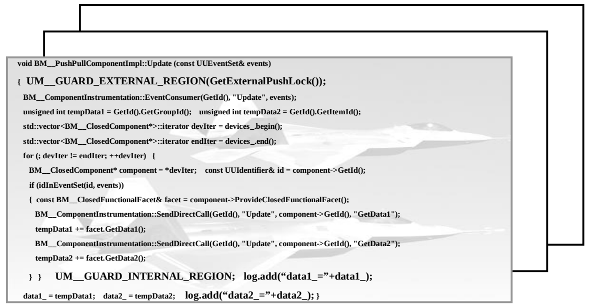
Δ_{MM}: The changes made to the meta-models
Δ_M: The changes reflected in the domain models
Δ_I: The changes reflected in the legacy source

Model-Driven Program Transformation Approach

- General Idea
  - Use “aspects” to insert configuration concerns (e.g., external/internal locking) as an alternative to explicit placement
  - From ESML models, generate the appropriate transformations where needed
- Problem
  - How to parse and execute transformations on “legacy languages”?
- Solution
  - Utilize a mature program transformation engine

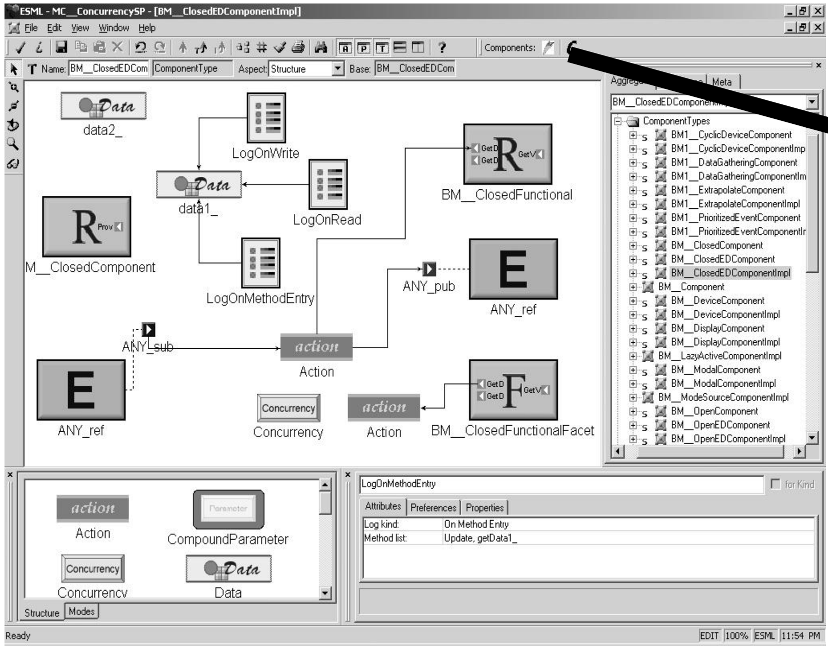
Model-Driven Program Transformation





- Ensures *causal* connection between model changes and the underlying source code of the legacy system
- Large-scale adaptation across multiple source files that are driven by minimal changes to the model properties
- Model interpreters generate transformation rules to modify source

Evolving Concern: Black box data recorder



Generated from Model interpreter

```
default base domain Cpp~VisualCpp6.
  pattern LogStmt() : statement = "log.add(\"data1_=\" +
data1_); ".
  pattern LogOnMethodAspect(s:statement_seq):
statement_seq = " { \s } \LogStmt\(\) ".
  pattern Update(id:identifier): qualified_id = "\id ::
Update".
  rule log_on_Update(ret:decl_specifier_seq, id:identifier,
p:parameter_declaration_clause, s: statement_seq):
function_definition -> function_definition
     = "\ret \Update\(\id\) (\p) { \s } " -> "\ret
\Update\(\id\) (\p) { \LogOnMethodAspect\(\s\) }"
    if ~[modsList:statement_seq .s matches
"\:statement_seq \LogOnMethodAspect\(\modsList\)"].
  rule log_on_Update_cv(ret:decl_specifier_seq,
id:identifier, p:parameter_declaration_clause, s:
statement_seq, cv: cv_qualifier_seq): function_definition
-> function_definition
     = "\ret \Update\(\id\) (\p) \cv { \s } " -> "\ret
\Update\(\id\) (\p) \cv { \LogOnMethodAspect\(\s\) }"    if
~[modsList:statement_seq .s matches "\:statement_seq
\LogOnMethodAspect\(\modsList\)"].
  pattern getData1_(id:identifier): qualified_id = "\id ::
getData1_".
  rule log_on_getData1_(ret:decl_specifier_seq,
id:identifier, p:parameter_declaration_clause, s:
statement_seq): function_definition -> function_definition
     = "\ret \getData1_\(\id\) (\p) { \s } " -> "\ret
\getData1_\(\id\) (\p) { \LogOnMethodAspect\(\s\) }"    if
~[modsList:statement_seq .s matches "\:statement_seq
\LogOnMethodAspect\(\modsList\)"].
  rule log_on_getData1__cv(ret:decl_specifier_seq,
id:identifier, p:parameter_declaration_clause, s:
statement_seq, cv: cv_qualifier_seq): function_definition
-> function_definition
     = "\ret \getData1_\(\id\) (\p) \cv { \s } " -> "\ret
\getData1_\(\id\) (\p) \cv { \LogOnMethodAspect\(\s\) }"
    if ~[modsList:statement_seq .s matches
"\:statement_seq \LogOnMethodAspect\(\modsList\)"].
public ruleset applyrules = { log_on_Update,
log_on_Update_cv, log_on_getData1_,
log_on_getData1__cv }.
```

Program Transformation to Add LogOnExit (DMS)

```
default base domain Cpp~VisualCpp6.

 pattern LogStmt(): statement = "log.add(\"data1_=\" + data1_);".

 pattern LogOnMethodExitAspect(s: statement_seq):
              statement_seq = "\s \LogStmt\(\)".

 pattern JoinPoint(id:identifier): qualified_id = "\id :: Update".

 rule insert_log_on_method_exit(id:identifier, s:statement_seq,
                                 p:parameter_declaration_clause):
     function_definition -> function_definition =
        "void \JoinPoint\(\id\)(\p) {\s} " ->
        "void \JoinPoint\(\id\)(\p) {\LogOnMethodExitAspect\(\s\)}"
 if ~[modsList:statement_seq. s matches
        "\:statement_seq \LogOnMethodExitAspect\(\modsList\)"].

 public ruleset applyrules={insert_log_on_method_exit}.
```

Transformed Code fragment

```
unsigned int BM__ClosedEDComponentImpl::getData1_ () const
 {
        Addlog("data1_=" + data1_);

        UM__GUARD_INTERNAL_REGION;
        BM__ComponentInstrumentation::ReceiveDirectCall(GetId(), "GetData1");

        Addlog("data1_=" + data1_);
        return data1_;
 }

void BM__ClosedEDComponentImpl::Update (const UUEventSet& events)
 {
        Addlog("data1_=" + data1_);

        UM__GUARD_EXTERNAL_REGION(GetExternalPushLock());
        BM__ComponentInstrumentation::EventConsumer(GetId(), "Update", events);
        unsigned int tempData1 = GetId().GetGroupId();
        unsigned int tempData2 = GetId().GetItemId();

        //*** REMOVED: code for implementing Real-time Event Channel

        Addlog("data1_=" + data1_);
        data1_ = tempData1;        //*** REMOVED: actual variable names (proprietary)
        data2_ = tempData2;
 }
```

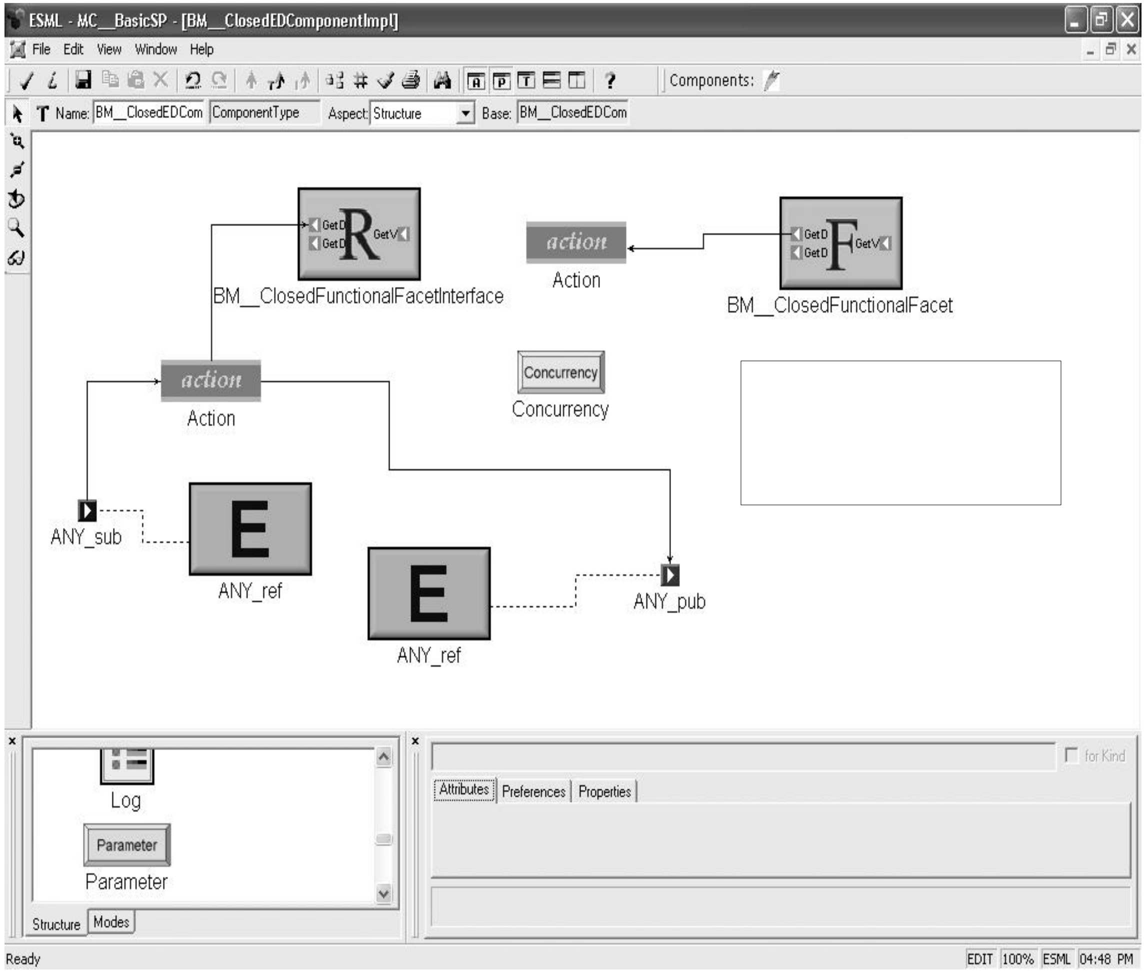
Log on getData1_() method entry

Log on reading data1_

Log on Update() method entry

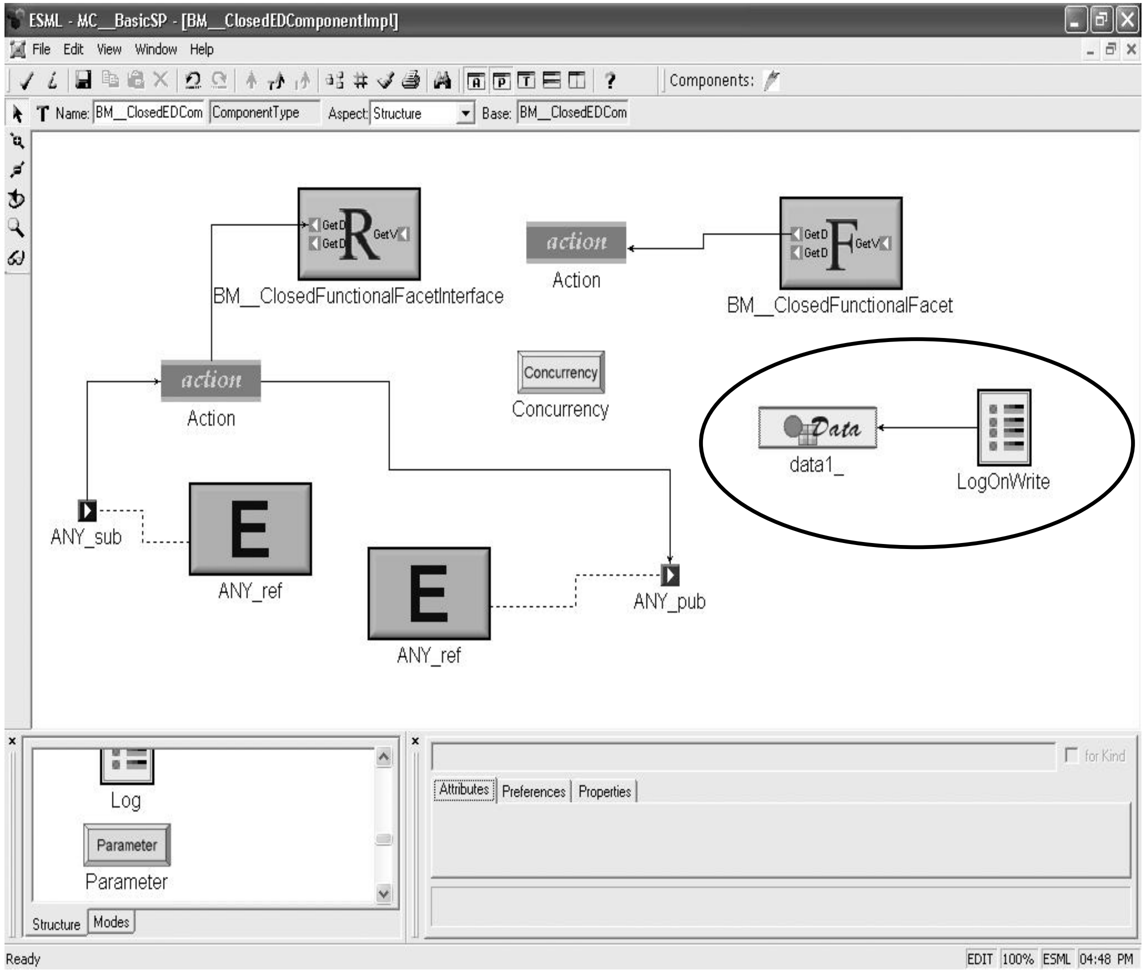
Log on writing data1_

Effect of Model Transformation



Before Weaving

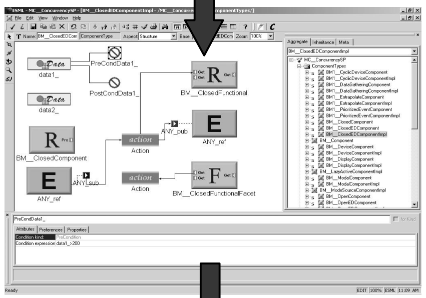
Effect of Model Transformation



After Weaving

Example: C-SAW Assertion Strategy

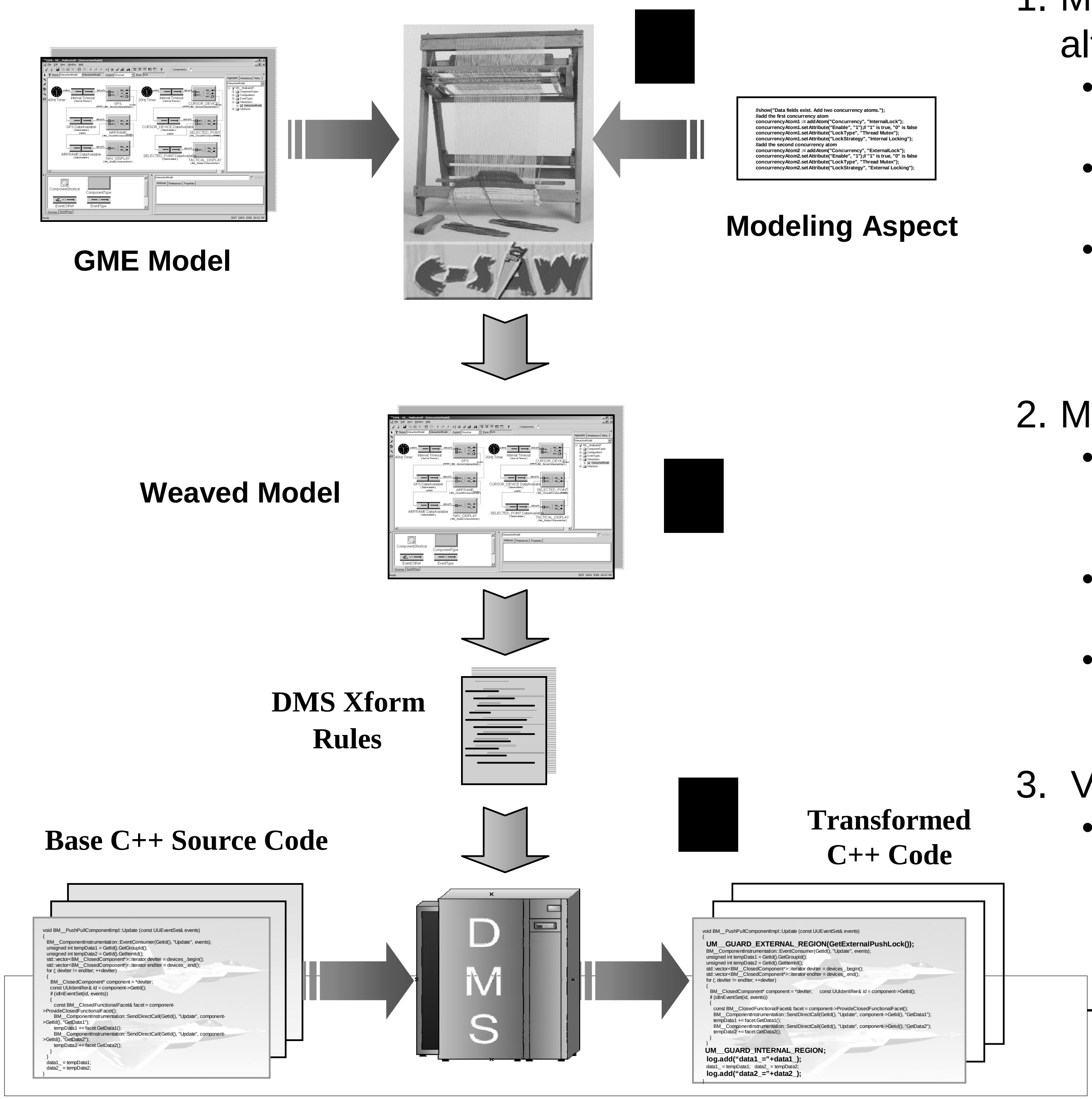
```
strategy FindData1()
{
  atoms()->select(a | a.kindOf()=="Data" and a.name()=="data1_")->AddConds();
}

strategy AddConds()
{
  declare p : model;       declare data, pre, post : atom;
  data := self;
  p := parent();

  pre:=p.addAtom("Condition","PrecondData1_");
  pre.setAttribute("Kind", "PreCondition");
  pre.setAttribute("Expression", "data1_>200");
  p.addConnection("AddCondition", pre, data);

  post:=p.addAtom("Condition", "PostcondData1_");
  post.setAttribute("Kind", "PostCondition");
  post.setAttribute("Expression", "data1_<500");
  p.addConnection("AddCondition", post, data);
}

aspect Start()
{
  rootFolder().findFolder("ComponentTypes").
        models()->select(m|m.name().endWith("Impl"))->FindData1();
}
```



```
default base domain Cpp~VisualCpp6.
pattern assertStmt() :
      statement = "assert(data1_>200);".
pattern aspect(s:statement_seq):
      statement_seq = " \assertStmt\(\){ \s }".
pattern joinpoint(id:identifier):
      qualified_id = "\id :: Update".

rule precondition(ret:decl_specifier_seq,
              id:identifier,
              p:parameter_declaration_clause,
              s:statement_seq):
    function_definition -> function_definition
  =  "\ret \joinpoint \(\id\)(\p){\s}"
  -> "\ret \joinpoint \(\id\)(\p){\aspect\(\s\)}"
  if ~[modsList:statement_seq .s matches
     "\:statement_seq \apect\(\modsList\)"].
public ruleset applyrules = { precondition }.
```

```
void BM__ClosedEDComponent::
        Update(const UUEventSet& events)
{

  assert(data1_>200);              // <- Precondition

  BM_CompInstrumentation::
       EventConsumer(GetId(), "Update", events);
  unsigned int tempData1 = GetId().GetGroupId();
  unsigned int tempData2 = GetId().GetItemId();

  //* REMOVED code for Real-time Event Channel
  //* REMOVED actual variable names (proprietary)

  data1_ = tempData1;
  data2_ = tempData2;

  assert(data1_<500);              // <- Postcondition

}
```

Summary: Two-Level Aspect Weaving



1. Model weaving to explore design alternatives more rapidly
   - Design decisions crosscut model hierarchy
   - Difficult to change models to new configuration
   - Design decisions captured as higher level policy strategies and weaved into models

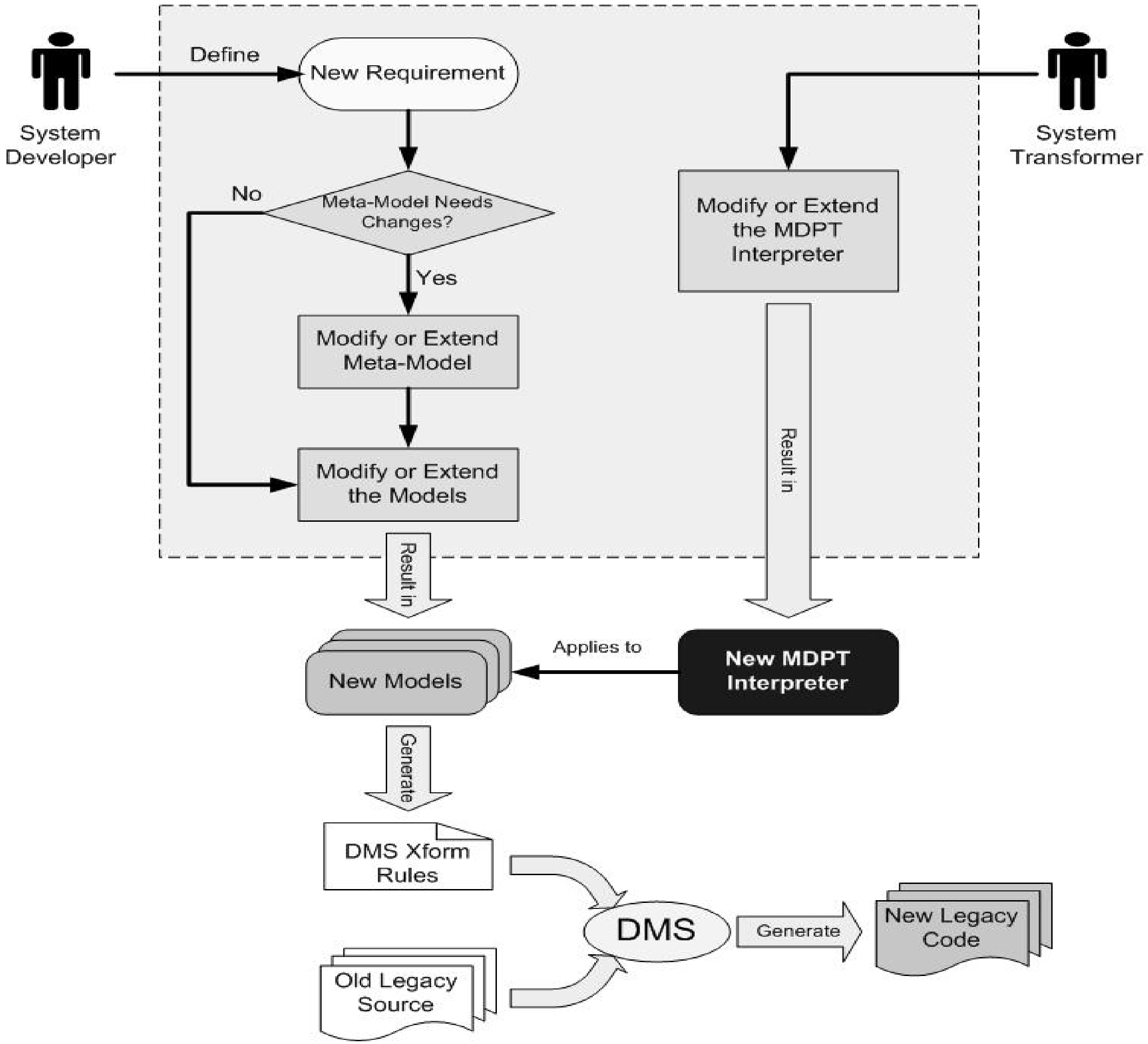
2. Model driven program transformation
   - Ensures causal connection between model changes and represented source code of legacy system
   - Assists in legacy evolution from new properties specified in models
   - Model interpreters generate transformation rules to modify source

3. Video: Bold Stroke Application
   - Apply original Bold Stroke C++ source code and generated transformation rules to DMS; result is a transformed version of Bold Stroke that is consistent with the model specification

Video

- Two-level weaving flight data recorder

Generalization of the control flow for the MDPT process



Conclusion

- Benefits
  - The model-driven program transformation technique is a generative approach for transforming large legacy systems from domain-specific models
  - It provides widespread adaptations across multiple source files according to the evolving model features
- Primary Limitation
  - The current MDPT interpreters are domain-specific and tied to a fixed set of concerns; to address new concerns, the model interpreter needs to be modified
- Future work
  - Generalization of the process for supporting legacy system evolution using MDPT
  - Investigation of related standards, such as the OMG ADM/KDM

For More Information…

Two-Level Aspect Weaving

http://www.cis.uab.edu/Research/C-SAW/
Contains papers, downloads, video demos

C-SAW

Software Composition & Modeling Laboratory
SOFT COM

Department of Computer and Information Sciences
University of Alabama at Birmingham