Supporting Forward and Reverse Engineering with Multiple Types of Models

Ira Baxter, Ph.D., CEO/CTO
idbaxter@semanticdesigns.com

Thursday, September 21st
Models 2017 Keynote
Austin, Texas
Software Engineering
State of the Art for Program Construction

• Deep Semantic Theory
• Requirements Capture and Traceability
• Formal Specifications in Domain Specific Languages or Models
• Mature Technologies: RDB, RPC, GUIs, …
• Modern languages with exceptions, generics, parallelism, …
• Automated Test Generation
• Configuration Management Tools
• Software Engineering Process and Methods
• **Model-driven engineering**
Model Driven Engineering

Software Development Problem Solved!

• Write a Model of a Desired Program
• Run the off-the-shelf Model-to-Code Generator
• Run Generated Code in Production
• Done!
Problem 1: How Does This work?

Step 1

Step 2

Step 3

"I think you should be more explicit here in Step Two."
Modelling Background

(2017) Brambilla, Cabot, Wimmer

(2017) Starr, Mangogna, Mellor

“With No Mysterious Gaps”
One MDE View

- (to produce code) “Model … must be executable (page 26)” No!

- **ExecutableUML** as typical model

- Distinguishes concrete vs abstract syntax, semantics … but no discussion of latter

- Emphasizes (concrete) **graphical models** syntax = model conformance

- Emphasizes **simple model of generation**: M2M (optional) then M2T

- M2M as **graphical model to graphical model** transforms (Refinements)

- Code generation via Model2Text **Size of semantic gap** from Model to target

- Some references “graph transformation” literature
Alternate MDE View

• Shows one approach in detail

• **ExecutableUML** as “the model”
  • **Classes** with data elements
  • **Statecharts** as class-transition descriptions with signals to other class-statecharts
  • **Abstract actions** to navigate class relations, side-effect class data

• Text encoding of concept xUML into **Pycca syntax**
  • **Actions as explicit C code fragments**
  • **Data declarations as C code fragments**

• **Pycca M2T generator** produces
  • C structs for classes
  • FSA per class with continuations used to signal to other class-FSAs

• **No mysterious gaps … 283 pages**
  *but* pretty weak generator where did Pycca come from?

---

“With No Mysterious Gaps”

(2017) Starr, Mangogna, Mellor
Model Driven Engineering

Software Development Problem Solved?

Problem 2

- Write a Model of a Desired Program
  - Where did my modelling notation come from?
  - What does it mean?
  - How did I get it into the computer?
  - Is it complete wrt Functionality? Performance?
  - Does my model mean what I think it means?

- Run the off-the-shelf Model-to-Code Generator
  - What machinery reads the model?
  - What is my choice of code targets? Is it only one language/technology?
  - How are model transformations specified?
  - How are they sequenced and executed?
  - How do I know they are right? Complete?
  - How long does code generation take to run?

- Run Generated Code in Production
  - Does the generated code need runtime support?
  - How do I debug problems using modelling terms?

- Done?
  - Success breeds discontent: user needs change, external context changes
  - How do I modify my model in an organized way to respond to these demands?
  - Do I regenerate all the code again, even for the parts of the model that don’t change?

Problem 3: Maintenance
How do these tools really work??

- MDE suggests “models” and “transforms” but not a lot of detail
  - Generated systems seem rather “small”
  - Where is the theory?
  - How to improve it?

- We need a different model of model driven engineering!
- => Program Transformations
  - General “model” of specifications: any formal artifact
    
    .... don’t have to executable or complete
  - Can define meaning of specifications using a variety of formalisms
  - Transforms as functions on specifications \( \rightarrow \) composable
    
    realized in a wide variety of ways
  - Correctness as preservation of properties by transforms
  - Ability to operate at same level of abstraction or many levels of abstraction
  - Metaprogramming to realize design choices
  - Ability to produce large systems
  - Ability to choose a variety of different implementations
  - Ability to operate on “Text” part of M2T
  - Perspective to define reverse engineering
Background: Semantic Designs

• Automated Software Engineering Tools since 1996
• *All* tools derived from single **Program Transformation Engine:**
  DMS® Software  Reengineering Toolkit
• **Focus on legacy code analysis/transformation**
• DMS based on 3 key foundations
  • Compiler Technology developed over last 50 years, *generalized*
  • Mathematical notion of A=B realized as mechanical *program transformations*
  • Scale support: large size, many languages, parallel computation for inference

• Some DMS tasks
  • Analysis of code structures at ANZ Bank (16MSLOC COBOL)
  • 100% fully automated migration of  B-2 Stealth Bomber Mission Software
  • Rearchitect large C++ applications in CORBA/RT compatible structure
  • Extraction of process-control models from legacy assembler code for Dow Chemical
DMS Software Reengineering Toolkit

Constant set of program manipulation services

Source Language Files (many files, multi-lingual)

Task Definition (Task Specific Analysis and Transformation Rules)

Domains == Language Definitions
- (Grammar Rules + General Analysis Rules + Formatting Rules for many languages or custom, including
  - C++ Factory
  - C# Configuration
  - HLASM
  - HTML
  - Java
  - Natural
  - SQL

Understanding
- Language parsers
- Compiler data structures
- Deep data flow analysis
- Data flow concept matching

Transformation Engine
- Source Code Patterns
- Source Rewrite Rules
- Condition on analysis results

Designed for the real world
- Millions of lines
- Thousand of files
- Mixed languages
- Parallel processing
- Full Unicode/Native char sets
- Actively used and enhanced for over 20 years

9/28/2017
Case Study: Large Banking System

**Analyze:** How are software elements connected?

**Business Challenge:** Programmers create new defects when making application changes

- Unhappy Customers (**ATMs went offline for a day**)
- Escalating maintenance costs

**Technical Problem:** Code and data dependencies obscured by application (Hogan) architecture

- 16+ Million lines of IBM Enterprise COBOL, JCL extended by Hogan
- 15,000 software components

**Solution:** DMS custom analyzer visualizing Component Connectivity

- Define custom parser for Hogan to DMS
- Parse COBOL, JCL, Hogan DBs
- Compute interconnections
- Graphically display connections

**Benefit:** Impact/change analysis now possible

U.S. Social Security Administration:
Same Problem but 200M SLOC!
Now in use for 3+ years
Case Study: B2 Bomber Mission Software

**Change:** 100% Automated Migration Jovial to C

**Business Challenge:** Existing B-2 Mission software incapable of meeting new requirements

- Legacy JOVIAL software needed to be modernized
- Internal teams unable to re-write application

**Technical Problem:** Legacy Software Complexity

- Failed internal manual and semi-automated translations
- 1.2 million lines Black code; **SD not allowed to see source**

**Solution:** Migrated 100% by DMS

- Define JOVIAL language from scratch to DMS
- Reuse existing definition for C target language
- ~6000 translation rules
- Delivered in 9 months

**Benefit:** Trustworthy solution for critical software

**Operational in**

B2 Bomber fleet
**Case Study: Avionics Software**

**Change:** OS replacement/Architectural shift

**Business Challenge:** Highly successful C++ product line for many Boeing military aircraft

- Hundreds of C++ components, communicating on limited-bandwidth internal aircraft data bus
- Military wants to add video cameras to all aircraft
- Internal bus overwhelmed; desperately need QoS data delivery guarantees

**Technical Challenge:** Replace legacy Boeing RTOS (no QoS) with CORBA/RT (QoS)

- Too big to do by hand: millions of SLOC
- Code architecture must change radically to match

**Solution:** Mass change to replace legacy OS calls then rearchitect

- Define C++ and Facet spec description to DMS
- Add rules to map legacy OS calls to CORBA
- Add rules to reshape code into “facets”

**Benefit:** 98% automated conversion of components

Savings of 1-2 man-months per component

New video components in UAV in live-fire exercise demo
Case Study: Chemical Plants

**Change:** Model/Migrate Software Running Manufacturing Process

**Business Challenge:** Trusted plant-controller computers starting to fail due to age

- Many different plants / **Thousands of control programs**
- Software had to be migrate to modern controller hardware
- Limited resources and time

**Technical Challenge:** Manual conversion impractical for scale

- Can’t be wrong or factory may “blow up”
- Assembly like language difficult to analyze

**Solution:** Automated Tool to recover abstract process control model from “assembly code”

- Define Dowtran from scratch to DMS
- Define **abstractions in terms of data flows** with conditional implementations
- DMS matches legacy code via data flows (“Programmer’s Apprentice”) to produce model
- Generate new controller code from model

**Benefit:** Reliable migration of safety critical software + huge cost savings + **design capture**

Some plants now converted
To a first order approximation, there’s no such thing as “new code”.

There’s only code somebody else wrote yesterday, that you want to change.
Software Engineering
State of Software Maintenance practice

• Theory: How to modify it?
  – How to describe a change?
  – Where to look for place to start?
  – How to make change?
  – How verify change?
  – How to verify rest of system?

• Practice: Key Problems
  – No specification
  – No design documents one can trust
  – Growing scale
  – No repeatable tests
  – Scar tissue from repeated hacking

• How are these systems going to have long lives?
How do we reconcile MBE and Software Maintenance?

• We need a model of software construction

• Then we need a model of maintenance deltas wrt construction
  – How to specify?
  – Where to look for place to start?
  – How to make change?
  – How to determine parts of code that are inconsistent with desired change?
  – How verify change?
  – How to verify rest of system?
So Why the Maintenance Mess?

- System has a Design
  - Problem Domain
  - Implementation Steps
  - Components, connections
  - ...what else?

- Consult Design for Guidance
  - Done!

- Ooops. I forgot the Design!
  - maybe didn’t know how to save it
Conventional Designs are just Artifact Projections

- Don’t explain all properties of artifact
- Don’t provide rationale for chosen structure
- Wrong to call these “designs”... perspectives?
Better Model of Design?

Transformational Explanation

• Based on *transformational program* generation

• Components:
  
  – *Formal Specification*
    
    • Functionality (what program does)
    
    • Performance (other program properties: size, speed, OS, languages)
      
      – Properties of the program, not the construction process
  
  – Transformation steps converting spec into code
    
    • Carry out implementation of Functionality fragments
  
  – Rationale for how steps contributes to desired performance
    
    • Direct contribution: optimizations, refinements
    
    • Indirect contributions: problem decomposition, solution preparation
    
    • Rejected Alternatives
Key Technology: Transformation Systems

Stepwise Semiautomatic Conversion of Specs to Code

Rqmts $\xrightarrow{f_S} \text{Spec} \xrightarrow{c_i} \text{Transform Engine} \xrightarrow{f_G} \text{Prog}$

Transforms aka “Rules”

- distributive law $t_1$
- unity multiplier $t_2$
- remove parentheses $t_{k-2}$
- like-term combination $t_{k-1}$
- factoring $t_k$

$(x-1)y + 2y \xrightarrow{f_S} (xy-1y) + 2y \xrightarrow{f_1} \ldots \xrightarrow{f_{k-1}} xy-y+2y \xrightarrow{f_k} xy+y \xrightarrow{f_G} (x+1)y$
What is a **transform**?
A partial function from specs/programs to specs/programs

\[ t: \text{Spec} \rightarrow \text{Spec} \]

Incredibly useful

*Procedural: Compilers, YACC, VLSI synthesizers, refactorings*

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

\[ x + 0 \Rightarrow x \]

**optimization**

\[ t(\text{Locator}): \text{Spec} \rightarrow \text{Spec} \]

**refinement**

\[
\text{sum}(\text{var}, \text{limit}, \text{vector}) \Rightarrow \\
\text{begin local s}=0, \text{var}; \\
\text{do var}=1 \text{ to limit}; \\
\text{s}=s+\text{vector(var)}; \text{ enddo} \\
\text{return s} \\
\text{end}
\]
The design space caused by multiple transformation choices

(* Symbolic Model *)
Application = Wavepropagation;
ModelType = StressStrain;
Medium = Acoustic,
Boundaries = Absorbing
Dimensionality = 2;
(* Target Properties *)
TargetLanguage = Fortran77;
1am.inFile = “1am.grd”;
(* Algorithm *)
AlgorithmClass = FiniteDifference;
FDMethod = ExplicitMethod;
BoundaryMethod = Taper,
DefaultOrder = 2;
(* Program *)
InlineQ = False;

Sinapse Specification of 3D Sonic Wave Modelling Code
[Kant92: Synthesis of Mathematical Modeling Software]

10,000 lines of CM Fortran

Same function, different performance

Depth >= 10000
Design Space Navigation

How to make implementation decisions?

- Huge number of intermediate states
- At each intermediate state many transforms are applicable
- How does machinery choose the “right” transformations to apply?
- How do we provide guidance? (“metaprogramming”)

MDE world often seems to offer only one choice
The Consistent Refinement Problem

• Not always practical to refine specification as monolith
  – so must “refine” parts of spec “independently”
  – must have separate “refinements” for parts (component transforms)
  – what guarantees that set of component transforms forms a refinement?

• Example:
  – Want to refine stack spec having push and pop actions
  – “Refine” push by adding new cell to linked list
  – “Refine” pop by decrementing pointer to array
  – Resulting program obviously doesn’t work!
  – The pair push ⇒ linked list & pop ⇒ array is not a refinement

• Must some how bundle sets of transforms as a consistent refinement
The Draco Paradigm\textsuperscript{1}  
DSLs and design space navigation

- Define a DSL
  - A Notational system for describing \textit{problems} or \textit{solutions} with shared agreement on meaning among domain experts
  - Tension between ease of problem specification and ability to achieve efficient implementation
    - $\Rightarrow$ Sometimes contain implementation hints
- Specify application in DSL
- Repeat
  - Apply optimizations at DSL level
    - Uses domain-level knowledge \textit{lost} in next step
    - Multiple optimizations added as knowledge as convenient
  - (Consistent) Refinement to lower DSL levels
    - Introduces implementation methods
    - Multiple refinements provide different results/performance
- Stop when final set of DSLs is executable

\textsuperscript{1} [Neighbors78]
A Domain Network (for Sinapse)

Specific Applications

Generic Applications

Computer Science

Execution Model

Target Execution

Sonic Wave Modelling
PDEs + boundaries

Discretized Equations

Matrix Arithmetic

OOP

C++

A bundle of transforms that are consistent

optimize

refine

optimize

refine

optimize

refine

optimize

refine

optimize
A Reusable Domain Network

Specific Applications
- Sonic Wave Modelling
- PDEs + boundaries
- optimize
- refine

Generic Applications
- Discretized Equations
- optimize
- refine

Computer Science
- Matrix Arithmetic
- optimize

Execution Model
- OOP
- refine

Target Execution
- C++

Electronic Funds Transfer
- Money Management

Punch Press Control
- Real Time Control
- Global Navigation

Parallelism / Distributed Computation
- Logic
- Functional

Data Flow
- Prolog
- Haskell
- Occam
Navigating the implementation space using Domains
Design = details from Abstract Implementation Space => specification, transforms, choices

Implementation Steps (1000s of transforms)

Design Choices

(* Symbolic Model *)
Application = Wavepropagation;
ModelType = StressStrain;
Medium = Acoustic,
Boundaries = Absorbing
Dimensionality = 2;
(* Target Properties *)
TargetLanguage = Fortran77;
lam.inFile = “lam.grd”;
(* Algorithm *)
AlgorithmClass = FiniteDifference;
FDMethod = ExplicitMethod;
BoundaryMethod = Taper,
DefaultOrder = 2;
(* Program *)
InlineQ = False;

Sinapse Specification of 3D Sonic Wave Modelling Code
[Kant92: Synthesis of Mathematical Modeling Software]

10,000 lines of CM Fortran
Paradigm: *Design Capture*

= spec, transforms, ...

---

(* Symbolic Model *)
Application = Wavepropagation;
ModelType = StressStrain;
Medium = Acoustic,
Boundaries = Absorbing
Dimensionality = 2;
(* Target Properties *)
TargetLanguage = Fortran77;
1am.inFile = “1am.grd”;
(* Algorithm *)
AlgorithmClass = FiniteDifference;
FDMethod = ExplicitMethod;
BoundaryMethod = Taper,
DefaultOrder = 2;
(* Program *)
InlineQ = False;

*Sinapse Specification of 2D Sonic Wave Modelling Code*
[Kant92: Synthesis of Mathematical Modeling Software]

---

\[ \rho \text{ (continuous)} \rightarrow \rho \text{ (1000:1000:.001)} \]

**Implementation Steps**

(1000s of program transforms)

\[ \rho \text{ (1000:1000:.001)} \rightarrow \text{...array of row ptrs...} \]

\[ x+0 \rightarrow x \]

---

10,000 lines of CM Fortran
Paradigm: *Design Capture with Rationale*

- Transformational Design
  - Functionality Spec ($f_0$) + Derivation
  - + Performance Spec ($G_{rest}$)
    + Justification + Alternatives
- Metaprograms to construct design
  - Goal driven transform application

“[Baxter92 Design Maintenance Systems” CACM]"
Paradigm: *Revising Design with Δs*

- **Transformational Design**
  - Functionality Spec ($f_0$) + Derivation
  - + Performance Spec ($G_{rest}$) + Justification + Alternatives

- **Metaprograms to construct design**
  - Goal driven transform application

- **Incremental Updates as Δs**
  - Specification, Performance, Technology Δs
  - Δs drive design revision: retain transforms that *commute* with delta

\[ \Delta @ p(C_i @ q(f_i)) = C_i @ q'(\Delta' @ p'(f_i)) \]

*Commuting Transforms*

\[ \Delta' @ p' \]

*[Baxter92 Design Maintenance Systems" CACM]*
Reality: What to do when all you have is code?

1. You are here with 10,000 lines of CM Fortran
Practical: *Incremental* Design Recovery

1. You are here with 10,000 lines of CM Fortran

2. Recover design/spec by “running” transforms backwards!

(* Symbolic Model *)
Application = Wavepropagation;
ModelType = StressStrain;
Medium = Acoustic,
Boundaries = Absorbing
Dimensionality = 2;
(* Target Properties *)
TargetLanguage = Fortran77;
1am.inFile = “1am.grd”;
(* Algorithm *)
AlgorithmClass = FiniteDifference;
FDMethod = ExplicitMethod;
BoundaryMethod = Taper,
DefaultOrder = 2;
(* Program *)
InlineQ = False;
Syntax patterns:
Matching idioms to concepts to reverse engineer

int getBankCode(int bn) {
    int bc;
    if (bn > 10 & bn <= 25)
        bc = 3;
    else
        bc = 0;
    return bc;
}

$A$ code idiom
Syntax patterns:
Matching idioms to concepts to reverse engineer

A code idiom

default base domain C~ISO9899c1990.

public pattern
generate_bank_code(bank_number:IDENTIFIER,
    bank_code:IDENTIFIER):statement_seq
= "if (bank_number > 10 & bank_number <= 25)
    bank_code = 3; // bank of ethic
else
    bank_code = 0; // unknown bank number"

Code pattern for idiom
Converting a real semaphore implementation back into abstraction

Baxter, I. and Mehlich, M.
Reverse Engineering is Reverse Forward Engineering.

RT:UnLock ; unlock block of code whose semaphore is in (X)
intds ; lock out the world momentarily
inc scb:count,x ; anybody in queue ?
bgt RT:ITSX ; b/ no, done releasing resource
stx itempx ; save pointer to semaphore
ldx scb:tcbq,x ; pointer to TCB to activate
ldd tcb:nexttcb,x ; find pointer to TCB following that
stx itempd ; save pointer to TCB to activate
ldx itempx ; pointer to semaphore
std scb:tcbq,x ; remove task from SCB queue
ldx itempd ; pointer to TCB to make ready to run
RT:ITSC ; insert task at (X) into ready queue and switch contexts if needed
; Assert: interrupts are disabled here
jsr RT:ITIQ ; insert task into ready queue
ldx RT:TCBQ ; are we still highest priority task ?
cmpx RT:CTCB ; ... ?
beg RT:ITSX ; b/ yes, pass control to caller
ldx #RT:ISCH ; no, force task switch
jmp RT:SIInt ; by interrupt to task scheduler
RT:ITSX inten ; enable interrupts and return to caller
rts

30 transformation rules including de-optimizations
Draco in reverse
Mainframe HLASM to C code

Several hundred transformation rules
including de-optimizations, goto removal
Data Flow patterns:
Matching code with dataflows, *not syntax*

```c
int getBankCode(int bn) {
    int bc;
    if (bn > 10 & bn <= 25)
        bc = 3;
    else
        bc = 0;
    return bc;
}
```

**A code idiom**

```
default base domain C~ISO9899c1990.

public data flow pattern
get_bank_code(bank_number:IDENTIFIER<~,
    bank_code:IDENTIFIER~>):statement_seq
    = "if (\bank_number > 10 & \bank_number <= 25)
        \bank_code = 3; // bank of ethel
    else
        \bank_code = 0; // unknown bank number"
```

**Data flow pattern for idiom**

```
int displayInfo(int rn) {
    boe_lower = 10;
    bank_number = lookup(rn);
    tmp = bank_number;
    boe_upper = 25;
    chkh = tmp <= boe_upper;
    if (!chkh)
        logOutOfRange(tmp);
    chkl = tmp > boe_lower;
    if (!chkl)
        logOutOfRange(tmp);
    chkr = chkh & chkl;
    if (chkr)
        logWithinRange(tmp);
    else {
        u_code = 0;
        printf("Error: Unknown bank number.\n");
        tmp = u_code;
    }
    displayRecord(tmp,rn);
    return tmp;
}
```

**Is the idiom somewhere in here? YES**
x = x + 1;
if x > 3 then y = x;
x=x+1;
if x>3 then y=x;
x = x + 1;
if x > 3 then y = x;
x = x + 1;
if x > 3 then y = x;
\begin{align*}
x &= x + 1; \\
\text{if } x > 3 \text{ then } y &= x;
\end{align*}
What's inside a Computer Program?
A Data Flow Graph

int fibonacci(n)
{ unsigned int fl= 0, fh = 1, i;
  if (n <=1 )
    fh = n;
  else
    for (i= 2; i<=n; i++) {
      int tmp = fh;
      fh =fl + fh;
      fl = tmp;
    }
  print("Fib(%d) = %d\n", n, fh);
  return n;
}

Big example wouldn't fit on football field…

Insight:
Maybe we can abstract away this detail

accumulate(fib#s)
COMPUTE-TOTAL.
MULTIPLY QUANTITY BY PRICE GIVING TOTAL-AMOUNT.
IF TOTAL-AMOUNT > DISCOUNT-THRESHOLD
  MULTIPLY TOTAL-AMOUNT BY DISCOUNT-PERCENT
    GIVING DISCOUNT-AMOUNT
  DIVIDE 100 INTO DISCOUNT-AMOUNT
  ADD ONE TO VAT-RATE GIVING TAX-ADJUSTMENT.
  MULTIPLY TAX-ADJUSTMENT INTO TOTAL-AMOUNT.
  DISPLAY COMPANY-NAME.
  DISPLAY "Total: ", TOTAL-AMOUNT.

COMPUTE-INVOICE.
MULTIPLY AMOUNT BY VAT-RATE GIVING TAX.
Compute INSURANCE = INSURANCE_RATE * AMOUNT.
ADD TAX TO AMOUNT.
ADD INSURANCE TO AMOUNT GIVING INVOICE_TOTAL.

define TaxStyle = { `Added`, `Multiplied` }

data flow pattern ComputeTax_by_multiplying(TaxRate:Constant, Total:IDENTIFIER)
  :StatementSequence
  "Compute $Total = 1.0 + $TaxRate"
  if Value(TaxRate)>0.0 and Value(TaxRate)<1.0;

data flow pattern ComputeTax_by_adding(TaxRate:Constant, Total:IDENTIFIER)
  :StatementSequence
  Temp:IDENTIFIER
  "MULTIPLY $Total BY $TaxRate GIVING $Temp."
  ADD $Temp TO $Total
  if Value(TaxRate)>0.0 and Value(TaxRate)<1.0

data flow pattern ComputeTax(TaxRate:Constant, Total:IDENTIFIER):
  <HowTaxed: TaxStyle>:
    StatementSequence
    case HowTaxed
      when `Added`
        ComputeTax_by_adding(TaxRate,Total)
      when `Multiplied`
        ComputeTax_by_multiplying(TaxRate,Total)
    esac;
Choice/Decision data declarations 1

- Used to enumerate space of implementation choices
  - Each decision represents selection of specific alternative for a choice
  - Often there are complex relations across decisions
  - Stack-as-array cannot realize “pop” using link-list operations
  - Data flow pattern for alternative depends on stack-as-array feature
  - Called *generic* types

- Patterns encode valid decision combinations with arbitrary boolean constraints
  - Matcher generates decision sets producing coherent dataflows

```cpp
generic type stack_implementation =
enum { `stack_via_singly_linked_list`,
   `stack_via_double_linked_list`,
   `stack_via_array_with_index` };
```
Choice/Decision data declarations 2

- **Syntax:** `generic type identifier = typedeclaration;`
- `identifier` is an RSL standard identifier
- **typedeclaration:**
  - `boolean`, with decision being True or False
  - `character` (Unicode)
  - `string` (of Unicode characters)
  - `natural`
  - `natural unsigned_constant .. unsigned_constant`
  - `integer`
  - `integer signed_constant .. signed_constant`
  - `float`
  - `float float_constant .. float_constant`
  - `rational`
  - `rational rational_constant .. rational_constant`
  - `enum { decision_literal_string, … }`
    - with decision_literal_strings being `text` (accent grave)
  - `identifier` (referring to an already named generic type)
  - `*` (RSL attempts to infer the type based its usage)
Matrix Multiply in real programs

• Abstract operation A*B
  – Fundamental to thinking about application
  – Rarely coded that way

• May be implemented in code in many ways
  – Algorithmic variations
    • Triply nested for loops
    • Strassen (recursive decomposition)
    • Library calls (BLAS == Basic Linear Algebra Subprograms)
  – Different data representations
    • Contiguous Memory Block: (row or column major order)
    • Sparse Matrix
    • Upper/Lower Triangular Matrix

• Matcher must find “matrix multiply” in face of variations
Matching abstract concepts using dataflow instead of syntax

private data flow pattern AddInto
(t: IDENTIFIER, -- target being updated
s: IDENTIFIER -- value to add to target)
):statement
= "\t += \s;" ? "\t = \t + \s;".

public data flow pattern MatrixMultiply
<i: Implementation,
ra: Representation, oa: Order,
rb: Representation, ob: Order,
rc: Representation, oc: Order>
(n: IDENTIFIER <~, -- in: matrix size parameter
m: IDENTIFIER <~, -- in: matrix size parameter
p: IDENTIFIER <~, -- in: matrix size parameter
a: IDENTIFIER <~, -- in: source matrix
b: IDENTIFIER <~, -- in: source matrix
c: IDENTIFIER ~> -- out: target matrix
):statement
= case
  when i == 'Explicit Code' then
  [i: IDENTIFIER, j: IDENTIFIER, k: IDENTIFIER,
s: IDENTIFIER,
ta: IDENTIFIER, tb: IDENTIFIER, tc: IDENTIFIER.
  "for (\i=0; \i<n; \i++)
    for (\j=0; \j<p; \j++) {
      \s=0;
      for (\k=0; \k<m; \k++) {
        \ReadElement<r,a,o>(\i,\n,\m,\l,\k,\ta)
        \ReadElement<r,b,o>(\b,\m,\p,\k,\j,\tb)
        tc = ta * tb;
        \AddInto<s,tc>(\s,tc)
      }
    }
  WriteElement<r,c,o>(\c,\n,\p,\i,\j,\s)
  }
  when i == 'BLAS' then
  ...
  esac.

Dataflow Pattern for MatrixMultiply in C (2)
Represents 3456 variants with decisions
Matching abstract concepts using dataflow instead of syntax

Dataflow Pattern for MatrixMultiply for C (1)

Matrix multiply abstraction found in code
Model/Abstraction Based Migration
(Dow Chemical)

AS-IS

Description of Model

Translation
(Reduction) Rules from RLL to DCS concepts

DCS Model: Process Control
Concepts applied to specific factory

Rule Compiler

Parse

Analyze

Target Language

Formatting Rules

DCS Model:

DO(252) IF STEP(255) AND BT(250)
DO(152) IF BT(152) AND AI(1) GT AP(2) AND BO(153) C
DO(144)
DO(152) IF BT(152) AND AI(1) GT AP(2) AND C
DO(153) AND BO(153) AND DC(144)
DO(152) IF BT(152) AND BT(153) AND BO(153) AND C
AI(1) GT AP(2) AND DC(144)
DC(151) IF DO(151) AND BT(151) AND BT(150) AND C
AI(121) GT AP(1) AND AI(121) GT AP(1) C
DC(151) UN D(152)
DO(152) IF BT(152) AND BT(153) AND BT(154) AND C
DO(146) IF BT(151) AND BT(153) AND BT(154) AND C
BO(159) AND BT(154) AND BT(153)
DO(146) IF DO(146) AND BT(146)
DO(145) IF AI(152) GT AP(3,80,100) C
ON DO(145) AND AI(152) GT AP(3,25,100) C

Description of RLL

Guidance

DCS Model:

if(ST4)
timer(T42,4sec)

if(ST2)
timer(T41,4sec)

STIX :=
(ST1 ! ST4 & T42, dn)
& (~ST1 ! ~ S1 & S2)
& first_scan )
STIX :=
( ST2 ! ST1 & S1 & ~S2 )
& (~ST2 ! T41, DN )
STIX :=
( ST3 ! ST2 & T41, DN )
& (~ST3 ! S1 ! ~S2 )
ST4X :=
( ST4 ! ST3 & ~S1 & S2 )
& (~ST4 ! T42, DN )

Translation Rules
from Model to ST

Rule Compiler

Parse

Analyze

Transform

Format
Dowtran

interpreted as Dataflow

then matched by Dataflow Patterns
Various Dowtran Analyzers

- Parse
  - Symbol Table
  - Control Flow
  - Data Flow
  - Forward Slice
  - State Transition Analysis
  - Latch Detection
  - Value Range Analysis
  - Timer Analysis
  - Safety Code Detection
  - Array/Index Range Detection
  - Data Flow Pattern Match
  - Controller Assembly
  - Best Match Selection/Revision
  - Backward Slice
  - Indirection Analysis
  - UI Analysis
  - Project Estimation Data
Connecting Matched Dataflow Patterns using intervening dataflows
Lessons

• Program Transformation is better model than MDE
  – Perspective and theory enable us to understand and improve

• We throw away the design. Price is really high.
  – STOP THAT

• Clean design capture starts with new program
  – We have a theory about how to capture it
  – Can revise transformational designs
    • Gives continuous maintenance model preserving design!

• Apply reverse engineering to legacy software
  – Reconstruct the part of the design you need
  – Switch to continuous maintenance model

• Dataflow patterns provide one kind of RE
  – Proven in practice on real code (Dow Chemical)
Dr. Baxter has been building system software since 1969, when he built a timesharing system on Data General Nova serial #3. In the mid-seventies, he built real-time, single user, multi-user systems and locally distributed OSes on 8 bit CPUs.

Realizing that software engineering was largely enhancement of existing code rather than building new code, and that the OS architectures were conceptually similar but shared no code, he went back to graduate school to learn more about reuse of knowledge in software maintenance. He studied program transformation tools for code generation and modification, obtaining a PhD from UC Irvine in 1990.

At the Schlumberger Computer Science lab, he worked on generation of parallel CM-5 Fortran code for sonic wave models from PDEs. He spent several years as consulting scientist for Rockwell Automation working on automating factory control.

In 1996, he founded Semantic Designs, where he is now CEO and CTO. At SD, he architected DMS, a general purpose program transformation engine, used in commercial software reengineering tasks, and he designed and implemented PARLANSE, a task-parallel, work-stealing programming language in which DMS is implemented.

He has been project lead on applying DMS to re-architect large C++ applications. Recent work includes automated recovery of chemical factory process control models from low-level industrial controller software to enable migrations to new process control platforms.
Abstract: Supporting Forward and Reverse Engineering with Multiple Types of Models

Many model-based tools work with single models, which capture some abstraction of a target software system of interest, with intent to convert the abstract description into a runnable computer program somehow. These tools usually provide some type of model-to-model transforms to carry out operations appropriate for the abstraction level of "the" model, and model-to-text transforms to generate low-level program source code. The model-to-model and model-to-text transforms are treated differently; one difference is that model-to-model transforms (may) compose, but model-to-text transforms by definition do not compose.

We have found it practical to mix high level models of programs with low-level models of source code, using domain-specific notations for each, and applying composable transformations (both reifying and abstracting) to both.

This talk will provide an intuitive unified view of how "models" and "code" can be treated consistently, and how transforms between them may be harnessed for both forward and reverse engineering.

A practical version of such a tool must be able to (meta)model a variety of models and source code, and allow specification and execution of transformations across these.

We will describe an effective tool for reverse engineering "assembly code" for running large-scale chemical plants back to abstract process control models, and then forward engineer those models to a completely different industrial control language, preserving the critical elements of factory control. This realizes the vision of (ADM/MDA) of "architecture-driven modeling of legacy applications into reifiable models. The implementation uses a combination of abstract syntax trees, data-flow graphs, and what amounts to graph-grammars, and mixes the analysis and transformation of these. Special support for reverse engineering low level code is provided by data flow pattern graphs. This reverse/forward engineering tool is realized using a commercial program transformation (DMS).

The resulting tool is being used by a Fortune 100 company to re-engineer the process control code for roughly 1000 factories.