This work is supported by the DARPA Information Exploitation Office (DARPA/IXO), under the Program Composition for Embedded Systems (PCES) program.

Supporting Software Evolution through Model-Driven Program Transformation

Jing Zhang (Advisor: Jeff Gray)
Department of Computer and Information Sciences – The University of Alabama at Birmingham
http://www.cis.uab.edu/zhangj

PROJECT OBJECTIVES

This project represents an investigation into two research objectives for supporting software evolution in Model-Driven Software Development (MDSD) through Model-Driven Program Transformation (MDPT):

- A generalization process for supporting evolution of legacy software through model-driven program transformation techniques;
- Evolution of model interpreters in the presence of meta-model schema changes.

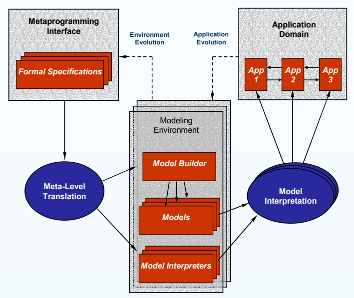
This project will assist in elevating models and model transformations to first-class development artifacts and facilitate further the ability to evolve model interpreters, as well as a generalized process for transforming legacy software using model-driven techniques.

BACKGROUND: DOMAIN-SPECIFIC MODELING

Key Characteristics of MIC

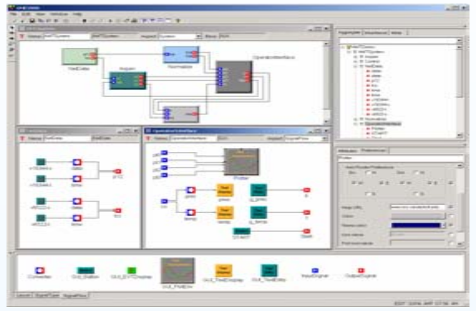
- Meta-modeling is used to define a domain modeling language and the constraints within that domain.
- From the meta-model, a modeling environment is created for a specific domain.
- Domain experts work within the generated environment to create specific instances of domain models.
- The model interpreters traverse the internal representation of the model and generate new artifacts (e.g., XML configuration files, source code).

Model Integrated Computing (MIC)



Generic Modeling Environment (GME)

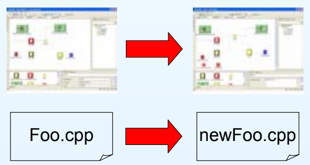
- Extendable/modular component-based architecture that supports MIC.
- Consists of a meta-programmable graphical editor, a model constraint checker, and meta-modeling environment.
- Used in numerous industrial domains: automotive, avionics, electrical utilities, digital signal processing, chemical plants.
- The GME is available from ISIS/Vanderbilt University; See http://www.isis.vanderbilt.edu/Projects/gme/.



TWO-DIMENSIONS OF TRANSFORMATION/TRANSLATION

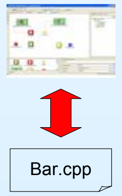
Horizontal transformation

- Transformation within the same abstraction level
  E.g., Model transformation, code refactoring.



Vertical translation

- Translation, or synthesis, between abstraction layers
  E.g., MIC interpreters, reverse engineering.



Consequence: Vertical *transformation* is needed!

KEY CHALLENGES

Challenge 1: Application of Model-Driven Software Development to Legacy Software Evolution

- Hard to maintain the fidelity between the model properties and the legacy source code.
- No support for parsing and invasively transforming legacy source code from higher-level models.

Challenge 2: Evolution of Model Interpreters in the Presence of Meta-model Schema Change

- Each evolution of the meta-model breaks the interpreter that was defined on the previous version.
- Changes to the API of the modeling tool may also necessitate adaptations to model interpreters.

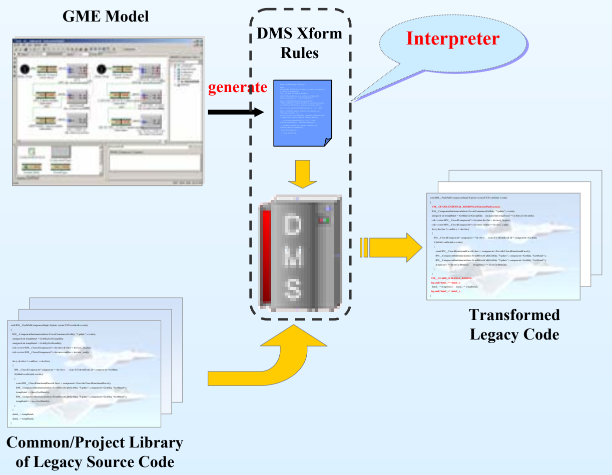
APPROACH 1: MODEL-DRIVEN PROGRAM TRANSFORMATION (MDPT)

MDPT for Supporting Legacy System Evolution

- Based on the unification of a program transformation system with a meta-modeling environment; specifically, uniting the Design Maintenance System (DMS) with GME.
- Key approach: the construction of model interpreters that generate DMS transformation rules from the model specifications.
- Along with the base implementation of a large application, the rules can be fed into DMS to invasively modify a large cross-section of the legacy source.
- DMS rules are transparent to the modeler and the whole transformation process is automated by the model interpreters.

Benefits

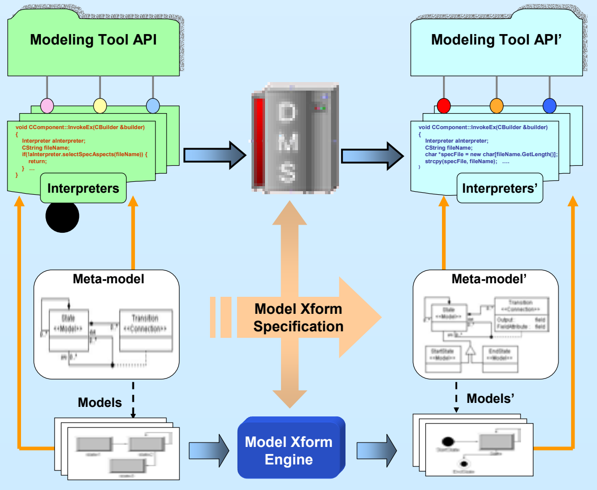
- Ensures causal connection between model changes and the underlying source code of the legacy system.
- Assists in legacy evolution from new properties specified in models.
- Model interpreters generate transformation rules to modify source.



APPROACH 2: MODEL INTERPRETER EVOLUTION ARCHITECHTURE (MIEA)

Goal: Automation of Model Interpreter Evolution

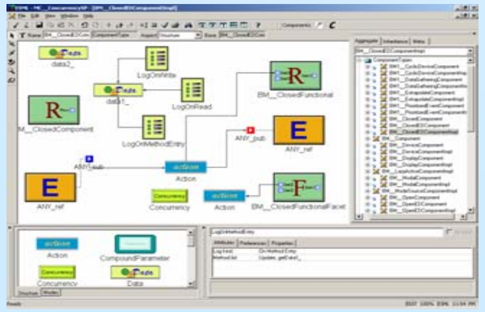
- As the meta-model evolves, a formal specification is defined to represent each step of the model transformation.
- The specification language consists of two parts:
  1) a pattern description of the interpreter signature;
  2) the replacement rule to perform the transformation.
- The transformation specification will be used to generate the DMS rewriting rules to transform the original interpreter into a new one that matches the changes to the meta-model.
- Model Interpreter Evolution in the presence of Modeling Tool API changes.



CASE STUDY: SUPPORTING A BLACK BOX DATA RECORDER IN A LARGE AVIONICS SYSTEM

Description

In avionics systems, a black box recorder is specified by a logging policy in the source program. This case study demonstrates the process of automatic adaptation of log statements into large source adaptations driven by the model-driven program transformation approach.



Generated DMS transformation rules

```
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 }.
```

According to the model properties, "log.add" statements are inserted automatically into a large section of legacy source files by the MDPT engine

Transformed Source Code Fragment

```
unsigned int BM__ClosedEDComponentImpl::getData1_ () const
  {   log.add("data1_=" + data1_);
      UM__GUARD_INTERNAL_REGION;
      BM__ComponentInstrumentation::ReceiveDirectCall(GetId(), "GetData1");
      log.add("data1_=" + data1_);
      return data1_;
  }

void BM__ClosedEDComponentImpl::Update (const UUEventSet& events)
  {   log.add("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
      log.add("data1_=" + data1_);
      data1_ = tempData1; //*** REMOVED: actual variable names (proprietary)
      data2_ = tempData2;
  }
```

http://www.cis.uab.edu/gray/Research/C-SAW/