A Language-Independent Approach Towards Software Maintenance using Grammar Adapters

Software Composition & Modeling Laboratory SOFTCOM
Department of Computer and Information Sciences
University of Alabama at Birmingham

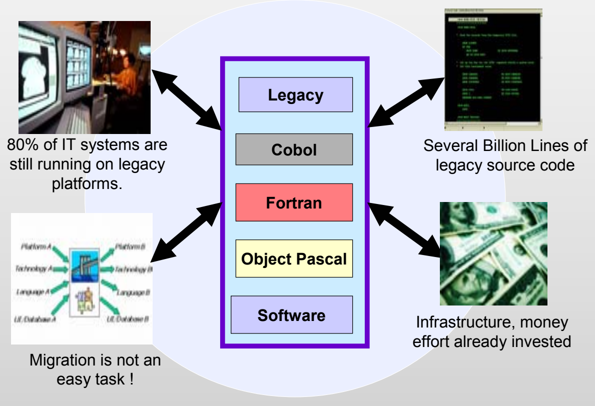
GenaWeave
http://www.cis.uab.edu/gray/Research/GenAWeave

UAB

Suman Roychoudhury (*roychous@cis.uab.edu*) **Advisor: Jeff Gray** **University of Alabama at Birmingham**

A generic platform to construct reusable design maintenance tools that aid software developers in preserving source code at an appropriate level of language abstraction.

Challenges with Legacy System Adaptation



80% of IT systems are still running on legacy platforms.

Several Billion Lines of legacy source code

Migration is not an easy task !

Infrastructure, money effort already invested

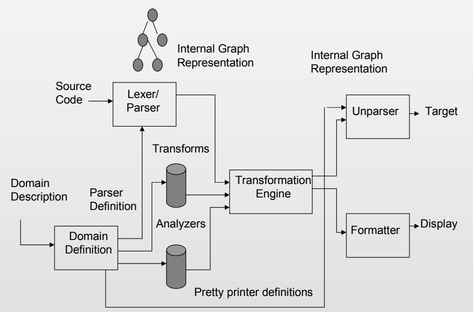
Need to Extend Maintenance Tool Support

Modern tools and methodologies are needed to reduce maintenance cost and effort

We can't throw them away ! Neither can we rewrite the whole legacy application, potential risk is high at no additional business functionality!

Can we extend support for refactoring, AOP and other Software Engineering techniques to these systems?

Generative Reuse using a Transformation System



- Capture and reuse *artifact generation* knowledge
- *Generative Programming* techniques are used to synthesize code from procedures using Lambda calculus
- Utilize a mature program transformation engine (Semantic Designs' DMS), which provides highly scalable parsers for a large set of legacy languages

However, transformation systems can be hard to use and understand because they are not at an appropriate level of abstraction for general software development.

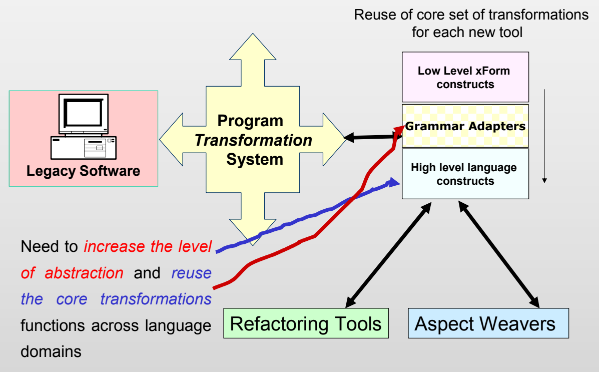
Complexities with Transformation Systems

- The transformation rules are often tied to the base language specification
- The end-programmers have little knowledge of the parsing algorithms and other low-level transformation constructs associated with the system
- The transformations are hard to construct; e.g., a simple rule to add a *tracing functionality* before every method execution in a C++ application looks like:

```
pattern probe_id_pattern1(): unqualified_id = " printf ".
rule insert_probe1(s: statement_seq):
        function_body -> function_body
 = " { \s } " ->
 "{ \probe_id_pattern1\(\)(\"Entering Method…\"); { \s } }".
```

- The transformations applied to one domain may have similarities with another; however, the tool builder often creates new tools from scratch without preserving and reusing the knowledge gained from previous construction

Generic Reuse using Grammar Adapters


Reuse of core set of transformations for each new tool
Low Level xForm constructs
Grammar Adapters
High level language constructs
Legacy Software
Program Transformation System
Refactoring Tools
Aspect Weavers

Need to *increase the level of abstraction* and *reuse the core transformations* functions across language domains

Example Case Study + Implementation Details

The following example shows an extension of basic AOP features in a legacy based application. The examples use complex program transformation rules to show the usefulness of grammar adapters as a language-independent approach to construct transformation maps for generic reuse.

Example:

Synchronization is an important aspect in the design of complex, concurrent embedded systems. The following code fragments show synchronization of database error handling objects written in Object Pascal and Java.

Tasks:

1. To separate the locking aspect (shown in red) using rewrite rules for both Object Pascal and Java
2. To extract the commonalities between the rules and create a transformation map using grammar adapters
3. To specify a high-level aspect language to abstract the transformation rules for the end-programmers

Sample Object Pascal Code

File Name : CLAS_HandleDBError.pas

```
….
function TExDBError.Handle(ServerType:
      TServerType; E: EDBEngineError) : Integer;
begin
  TExHandleCollection(Collection).LockHandle;
  try
   <database error handling code omitted here>
  finally
    TExHandleCollection(Collection).UnLockHandle;
  end;
  ….
end;
```

Synchronization Rules for Object Pascal

File Name : SyncObjectPascalMethods.rsl

```
rule sync_OP_meths (sl:stmt_list, id:IDENTIFIER,
        fps: formal_params, frt:func_result_type):
qual_func_header_decl -> qual_func_header_decl =
"function \method_id\(\id\) \fps : \frt ;
        begin \sl end;" ->
"function  \method_id\(\id\) \fps : \frt ;
        begin \LockStmt\(\);
          try \sl
          finally \UnLockStmt\(\);
        end;"
…
```

Sample Java Code

File Name : CLAS_HandleDBError.java

```
….
public Integer HandleDBError(TServerType
                    ServerType, EDBEngineError E)
{
  ErrorHandlerCollection.LockHandle(Collection);
  try {
     <database error handling code omitted here>
  }
  finally {

  ErrorHandlerCollection.UnLockHandle(Collection);
  }
  ….
}
```

Synchronization Rules for Java

File Name : SyncJavaMethods.rsl

```
rule sync_Java_meths (s_seq:stmt_seq,m_mods:
           meth_modifiers, t:type,id:IDENTIFIER,
           fp:fparams) :
method_declaration -> method_declaration =
"\m_mods \type \method_id\(\id\) \fp
        { \s_seq } ;" ->
"\m_mods \type \method_id\(\id\) \fp
        { \LockStmt\(\);
         try { \s_seq }
         finally { \\UnLockStmt\(\) }
        }; "
…
```

Extract commonalities in the transformation map

File Name : SyncTransformationMap.ag

```
st_list@ag → stml_list@OP
ID_list@ag → id@OP
param_list@ag → paramformal_params@OP
ret_type@ag → func_result_type@OP
meth_signature@ag
  →qual_func_header_decl@OP
……….
st_list@ag → stmt_seq@Java
ID_list@ag → id@Java
param_list@ag → fparams@Java
ret_type@ag → type@Java
meth_signature@ag
  →method_declaration@Java
……….
```

File Name : GenericSyncRule.rsl

```
rule sync_generic (sL:st_list,
   id:ID_list, pl: param_list,
   rt:ret_type):
std_func_decl -> std_fun_decl =
"\meth_signature\(\id\,\pl\,\rt)
     \std_start \sL \std_end" ->
"\meth_signature\(\id\,\pl\,\rt)
     \std_start
        \LockStmt\(\);
            \try_finally_decl \sL
        \UnLockStmt\(\);
     \std_end"
```

File Name : AbstractSync.gl

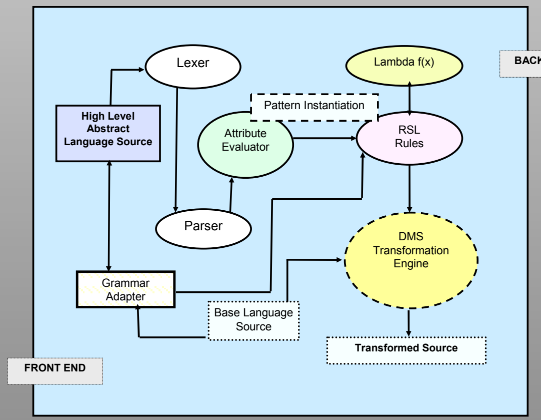
High level abstract aspect

```
aspect generic_synchronize {
  before(): execution (mname@meth_signature) {
       // insert base language specific lock statement
  }
  after():  execution (mname@meth_signature) {
       // insert base language specific unlock statement
  }
```

Advantages

- Generic reuse of the core transformations for faster tool development
- A high level language to drive the low level transforms for end-programmers
- Lower engineering cost raising reliability, performance and quality of software

High Level System Overview + References



Key References:

- Ira Baxter, Christopher Pidgeon, and Michael Mehlich, "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering (ICSE),* Edinburgh, Scotland, May 2004, pp. 625-634
- Jeff Gray, and Suman Roychoudhury, "A Technique for Constructing Aspect Weavers using a Program Transformation Engine," *AOSD '04, International Conference on Aspect-Oriented Software Development,* Lancaster, UK, March 22-26 , 2004, pp. 36-45
- Jeff Gray, Jing Zhang, Yuehua Lin, Suman Roychoudhury, Hui Wu Rajesh Sudarsan, Aniruddha Gokhale, Sandeep Neema, Feng Shi, and Ted Bapty, "Model-Driven Program Transformation of a Large Avionics Framework," *Generative Programming and Component Engineering (GPCE 2004)*, Springer-Verlag LNCS, Vancouver, BC, October 2004.
- Ralf Lämmel, "Grammar Adaptation," *Intl. Symposium of Formal Methods Europe (FME),* Springer-Verlag LNCS 2021, Berlin, Germany, March 2001, pp. 550–570.