Metaprogram Implementation by Second Order Source Transformation

James R. Cordy*
Medha Shukla Sarkar†
School of Computing, Queen's University
Kingston, Ontario, Canada K7L 3N6
cordy@cs.queensu.ca, msarkar@mtsu.edu

Metaprogramming is the process of specifying generic software templates from which classes of software components, or parts thereof, can be automatically instantiated under direction of a formal design model to produce new software components. In the μ* system [Cordy92], metaprograms are specified using an annotated by-example style accessible to ordinary programmers of the target programming language. Annotations in the form of Prolog-like predicates specify the design conditions under which different parts of the source template are to be instantiated. Instantiation of a source component is then done by providing a design model as a database of Prolog facts from which the design conditions can be evaluated and source component instances automatically generated using Prolog-style deduction.

The implementation of μ* is interesting in the context of this workshop because it is entirely done using the source transformation language TXL [Cordy91,04]. The implementation is achieved in a two stage process in which metaprograms are first translated by source transformation into equivalent TXL programs. These TXL programs are then run with input from a design database to implement instantiation of the metaprograms and generate instantiated source code by source transformation. In essence, this implementation is a second order source transformation.

1. μ* : A Family of Metalanguages

μ* (pronounced "mew-star") is a family of by-example metaprogramming languages that share a common metanotation and implementation. The philosophy of the family is exactly the ideal: the metaprogramming language for each target language consists of the target language itself, augmented with meta-annotations specifying conditions in the design database. For example, μC, the metalanguage for C, consists of C program syntax, optionally annotated with meta-annotations. The syntax of meta-annotations is the same across all target languages. In each case, the syntax of the basic metalanguage is the syntax of the language itself, and the syntax of the meta-annotations is the syntax of μ*. The target language can be any programming or specification language with a formal syntax.

2. By-example Metaprogramming

In μ*, every program written in a target language is a metaprogram unconditionally generating itself. Thus every C program is automatically a μC program, and every Prolog program is a μProlog program. Syntactically contained program fragments (for example, declarations, statements, and so on) are also in general metaprograms for themselves.

The addition of meta-annotations to a metaprogram attaches the metaprogram to the design database and makes generation of the annotated parts conditional on the facts in the database. The range of affected code dependent on a design condition is denoted by enclosing it in backslashes, followed by the meta-annotation and a double backslash to mark its end, as shown in Figure 1. The backslash is the only symbol reserved by μ* it can be replaced with any other single symbol.

Because in many cases the intended role of the affected area in the target source is ambiguous, the role must be given explicitly following the bracketed area, as shown in Figure 1. The role is the name of the intended part of speech in the target language reference syntax (that is, the common name of the entity in the target language, for example *statement* or *declaration* in C) enclosed in square brackets [ ].

3. Generative Metaprograms

The μ* annotation language provides two basic operations: *when*, which includes a section of target source conditionally on the provability of a predicate on the database, and *each*, which generates one copy of the section of target source for every solution to a predicate in the design database (Figure 2). These two operations can be nested to give complex combinations of conditional generation.

The database is searched for solutions to each annotation predicate. When a solution is found, the metavariables in the predicate are bound to the terms found in the solution in the design database. The metavariables can then be instantiated in the target source generated for that solution. Repeated instances of a metavariable in a predicate specify unification in the usual Prolog way, so the predicate *function(F[id])* ***and*** *returns(F,int)* specifies only those entities that are functions in the design that return the type *int*. Figure 2 shows an example specifying a μC metaprogram to generate external C routine declarations for every function entity in a design database.

When programmers write code templates, they often use a pseudo-code style in which descriptive identifiers take the place

```
const char *strsignal(int n)
{
    static char
        buf[sizeof("Signal ")+1+INT_DIGITS];
\
    if (n>=0 && n<NSIG && sys_siglist[n]!= 0)
        return sys_siglist[n];
    sprintf(buf,"Signal %d",n);
\ [statement*]
  when listing
\\
    return buf;
}
```

Figure 1. Trivial Example μC Metaprogram.

The if and sprintf statements enclosed in backslashes are conditionally included in instances of the metaprogram only if "listing" is a design fact in the design database.

* Author's present address: ITC-IRST, Trento, Italy.
† Author's present address: Middle Tennessee State University, U.S.A.

```
\
  extern FType F();
\ [declaration*]
  each function(F[id])
   and returns(F,FType[type])
\\
```

Figure 2. Trivial Generative μC Metaprogram.

The interpretation is that a sequence of declarations is to be generated, one for each "function" entity in the design database. Unification on design facts finds the associated type automatically from the "returns" design fact.

of sections to be filled in later. μ* provides this same feature by allowing metavariable identifiers to take the place of any part of a target source fragment enclosed in backslashes, and by allowing later refinement of the role and source text of the metavariable, either as part of the solution to a predicate, or by using a *where* clause.

A *where* clause is a nested metaprogram that generates a target source fragment and binds it to a metavariable for use in other parts of the metaprogram, for example, the main source text. While in this position paper we do not have room for realistic examples, the nested combination of *when*, *each*, and *where* with Prolog-style predicate solution and unification on design databases gives μ* great power and flexibility while retaining the by-example nature of metaprogram templates. It has been used to specify and generate complex code artifacts in C, Prolog and Turing using design databases describing production software interfaces such as OpenGL.

4. Implementation of μ* Using TXL

μ* is implemented using the TXL source transformation language [Cordy91,04] by translating each μL metaprogram for a target language L to a corresponding TXL source transformation ruleset using TXL source transformation. The generated TXL ruleset is then combined with reference grammars for the target language L and Prolog to create a TXL program that implements the instantiation of the μL metaprogram from a design database of Prolog facts, as shown in Figure 3. The translation of μL metaprograms to corresponding TXL metaprograms is itself achieved using a source transformation specified and implemented in TXL. In essence, this implementation is simply a second order source transformation interpretation of the original μL metaprogram.

The purpose of this position paper is to introduce and explore the possibility of generalizing this technique to the wider implementation of metaprogramming systems using source transformation tools. While in this application the technique is driven by an entity-relationship design database in Prolog form, there is no fundamental reason why the design model could not be represented in any other design notation, including those based on UML [OMG03]. And while the particular source transformation system used here is TXL, there is no reason why the technique would not work with other tools.

References.

[Cordy92] J.R. Cordy and M. Shukla, "Practical Metaprogramming", *Proc. CASCON'92, IBM Centre for Advanced Studies Confernece,* Toronto, November 1992, pp. 215-224.

[Cordy91] J.R. Cordy, C.D. Halpern and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects", *Computer Lang*. 16,1 (January 1991), pp. 97-107.

[Cordy04] J.R. Cordy, "TXL - A Language for Programming Language Tools and Applications", *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications,* Barcelona, Spain, April 2004, pp. 1-27.

[OMG03] Object Management Group, http:/www.omg.org, *Unified Modeling Language Specification v1.5*, March 2003.

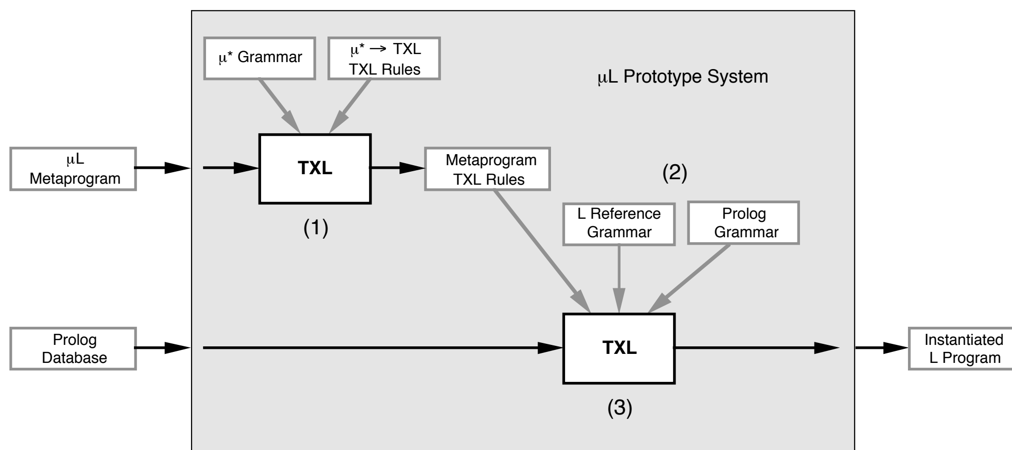

Figure 3. Implementation of μ* Using TXL.

A TXL source transformation is used to translate μL metaprograms to TXL transformation rules for a target language L (1). The result is combined with standard reference grammars for L and Prolog to give a complete TXL program (2), which is then run with a Prolog form entity-relationship design database as input. The design database is transformed by the TXL program to a target language L instantiation of the μL metaprogram (3). The entire process is very efficient, running in seconds on practical examples.