Disentangling the Implementation of Local-to-Global Transformations in a Rewrite Rule Transformation System[*]

Thomas Cleenewerck
Vrije Universiteit Brussel
Brussel, Belgium
tcleenew@vub.ac.be

Theo D'Hondt
Vrije Universiteit Brussel
Brussel, Belgium
tjdhondt@vub.ac.be

ABSTRACT

Transformation rules are often used to implement compilers for domain-specific languages. In an ideal situation, each transformation rule is a modular unit transforming one input element of the source program into a new element of the output program. However, in practice, transformation rules must be written which take one input element and produce several new elements belonging to various locations in the output program, the so-called local-to-global transformations. The implementation of such transformations is very complex and tightly coupled which imposes severe constraints on maintenance and evolvability. In this paper, we propose a transformation architecture on top of rewrite rules to loosen this coupling. The resulting transformation system combines the simplicity and modularity properties of rewrite rules with a new semi-automatic composition system that enables the implementation of local-to-global transformations without hampering maintenance and future evolutions.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: Program transformation; D.3.2 [**Language Classifications**]: Specialized application languages

Keywords

Maintainability, Evolvability, Rewrite rules, Program Transformations

1. INTRODUCTION

A domain-specific language (DSL) raises the abstraction level of a programming language to the domain level. The compiler for a DSL is often implemented using a general transformation system. Such a DSL compiler transforms a source program into a target program by means of transformation rules that implement a mapping of the abstract syntax tree (AST) from the source to the target language.

The most dominating kind of transformations used today are rewrite rules [6, 15, 8]. A rewrite rule specifies a substitution of a part of the source AST by a part of the target AST. This means that a rewrite rule is only concerned with transforming a part of the tree, without context information. However, implementing DSLs merely by means of such 'in place' transformations is hardly ever possible. Quite often, a single transformation must exert some output on other parts of the target AST. In the survey of Jonne van Wijngaarden et. al. [14] these kinds of transformations are called *local-to-global* transformations.

The implementation of local-to-global transformations solely by means of rewrite rules is very complex and tangled. Extensions of transformation systems to the rewrite rule paradigm with various data acquisition techniques like queries and tree traversals [13] did not reduce the tangling. Besides other problems like reusability which we do not address in this paper, maintenance and evolution of DSL compilers consisting of such transformations are quite a nightmare.

We present a mechanism to complement rewrite rules such that they can implement local-to-global transformations without sacrificing maintainability and evolvability. This is achieved by providing a semi-automatic process that will treat the outputs which cannot be handled by the currently available 'in-place' substitution mechanism.

In the following section 2, we investigate the fundamental problem that forces a developer to write non-modular rewrite rules. Section 3 introduces our solution architecture, which explains the architecture in more detail. In section 4 the architecture is evaluated and the implications of the approach are discussed. The related work is discussed in section 5 and the paper concludes in section 6.

2. LOCAL-TO-GLOBAL TRANSFORMATIONS

We conceived a small DSL language to illustrate the problems with the implementation of local-to-global transformation with rewrite rules.

2.1 Motivating Example

The small DSL language we use as an example is called the WIZ language. It was designed to describe installation

[*]This research is partially performed in the context of the e-VRT Advanced Media project (funded by the Flemish Government) which consists of a joint collaboration between VRT, VUB, UG, and IMEC.

```
W  wizard( (
P    page (
I      inputfield("appname" defvalue("T"))

I      inputfield("appdir"
D        defvalue( stats(( return( +(+(
K                (key(programpath) "/") appname)) )))))
   )))
```

Figure 1: The source AST of installation program written in WIZ. The prefixes W, P, I, D, K correspond to the wizard, page, inputfield, defvalue and key transformations that take these nodes as input.

```
W class (datamember("Wizard" "wizard")
I  datamember("Inputfield" "appname")
I  datamember("Inputfield" "appdir")
K  datamember("Key" "programpath"))
W  method("main" (...) (stats (
K    assign( var("programpath") new("Key"))

W    assign( var("wizard") new("Wizard"))
P    assign( var("page") new("Page"))

I    assign( var("appname") new("InputField"))
I    assign( var("appdir") new("InputField"))

I    mcall( var("appname") setListener
I        (... mcall( () "appdirdefvalue" ()) ... ))

I    mcall( var("appname") setValue ("T"))
I    mcall( var("appdir") setValue
D        (mcall( () "defvalue" ())))

P    mcall( var("page") "add" (
I         var("appname")))
P    mcall( var("page") "add" (
I          var("appdir")))

W    mcall( var("wizard") "add" (
P        var("page")))
W   )))
D   method("defvalue" () (stats (
D      return( +(+( mcall(
K             var("programpath") "getValue" ()) "/")
D          mcall( var("appname") "getValue" ())))
D   )))
```

Figure 2: The target AST obtained after transforming the sample installation program (figure 1). The prefixes W, P, I, D, K correspond to the wizard, page, inputfield, defvalue and key transformations which produce these nodes.

wizards. A program written in WIZ is transformed into a Java program on top of a framework. The purpose of WIZ is to hide the developers from tedious glue code and encapsulate the best practices to instantiate the underlying framework. The surface syntax of WIZ is of no importance for this paper, instead we describe how one uses the language and how the transformations on the AST implement the translation to Java.

The example WIZ program of figure 1 describes a wizard consisting of one page. The page contains an inputfield for the application name (`appname`) and an inputfield for the installation directory (`appdir`). The default value of the installation directory is the result of the concatenation of the registry key `programpath` (holding the default location for applications) and the application name (`appname`). When the application name is changed, the default value for the programpath (which depends on it) is recomputed. The desired result after transforming this specification is shown in figure 2.

The implementation of the compiler for the WIZ language is divided in 5 transformations. Each of the transformations corresponds to the translation of one source AST node to its Java counter parts. The source AST nodes are `wizard`, `page`, `inputfield`, `defvalue` and `key` which are respectively translated by the W, P, I, D, K transformations. In figure 1 the nodes are prefixed by the transformation that takes the prefixed node as input. The result nodes produced by those transformations are shown in figure 2 and are also prefixed using the same prefix. This clearly shows that the target ast nodes of produced by the transformations are scattered in the desired target program. This scattering is not an accidental situation, but rather a structural problem between the two languages. The resulting nodes of one transformation may not all substitute the source node. Only results of a particular type may be substituted with the source node, to insure the resulting AST is type correct. The results that do not fit must be put in other places in the tree. For example, the source `key` node is part of an expression, therefore the results the `key` transformation produces must be an expression in order to yield a correct target program. The nonlocal datamember and the statement must consequently be put in other locations of the target program. Besides typing problems, some result nodes belong to a different part in the target program due to grammatical constraints or to particular semantical constraints imposed by the DSL developer.

2.2 Implementation using rewrite rules

Rewrite rules essentially describe 'in-place' substitutions, the results produced by the rewrite rule must all fit into the location of the node under transformation, which we will refer to as the *local results*. In order to deal with the other results that do not fit in the location, the so called *non-local results*, we need a workaround. One can think of several workarounds but they all boil down to the following strategy. Because most of the rewrite rule systems have been extended with some kind of traversal mechanism [13], we will only consider such systems in the discussion of the workaround. A local-to-global transformation is broken up into several rewrite rules and traversals, each one responsible for a single result. The idea is to write for each transformation a primary rewrite rule that produces both local and non-local results contained in a cons cell, and for each non-local result a secondary traversal to collect the non-local and feed it into a secondary rewrite rule that matches the correct target location in order to relocate the non-local to the desired location.

The evolutions and maintainability problems of this strategy are caused by these latter rewrite rules and traversals. They operate on a tree that has been fully rewritten into target nodes, consequently the matching clauses of these rewrite rules and traversals are entirely expressed in terms of target nodes. This raises 3 problems: the secondary traversals and rewrite rules (1) heavily depend on the outcome of the primary rewrite rules and their interactions, (2) must be carefully scheduled in order not to violate the context conditions of other secondary traversals and rewrite rules and (3) depend on the exact properties of the AST at a certain stage during transformation. A full description and explanation

```
class (datamember("Wizard" "wizard")
 datamember("Inputfield" "appname")
 method("main" (...) (stats (

   assign( var("wizard") new("Wizard"))
   assign( var("page") new("Page"))

    assign( var("appname") new("InputField"))

    mcall( var("appname") setValue ("T"))

    mcall( var("page") "add" (var("appname")))
    mcall( var("page") "add" (
     CONS( LOCAL( var("appdir")
     (NONLOCAL(
       assign( var("appdir") new("InputField"))
       mcall( var("appname") setListener
            (... mcall( () "appdirdefvalue" ()) ... ))
       mcall( var("appdir") setValue
         CONS( LOCAL( mcall( () "defvalue" ()) )
         (NONLOCAL(
            method("defvalue" () (stats (
            return( +(+( CONS(
             (LOCAL( mcall( var("programpath") "getValue" ()) "/"))
             (NONLOCAL( datamember("Key" "programpath")) )
              NONLOCAL( assign( var("programpath") new("Key")) )))
             mcall( var("appname") "getValue" ())))
           )))
         ))
       )))
      NONLOCAL( datamember("Inputfield" "appdir") ))))))

     mcall( var("wizard") "add" (var("page")))
  ))))
```

Figure 3: The state of the target AST right before the relocation and integration of the nonlocals produced by the inputfield transformation is initiated.

of those problems can be found in [5].

Any small change in the grammar of the source language or in the implementation of the transformations easily breaks most of the above assumptions made in the implementation of the secondary traversals and rewrite rules.

3. RESOLUTION OF NON-LOCAL TARGET NODES

After analyzing local-to-global transformations more carefully, we identified three different responsibilities: (1) producing the target AST nodes, (2) navigating through the target AST to find the location of the non-local nodes and (3) adding the produced target AST nodes in that location. In contrast to the existing manual and functional (stating rather how than what) approaches, in our approach these last two responsibilities are handled by a composition system that is driven by a declarative specification stating the target location of each non-local and the integration of the non-locals into the tree. Moreover we will show that part of the required specifications can be automatically derived from the target language grammar, which yields a semi-automatic composition system.

In our solution architecture, a modular rewrite rule produces both local and non-local target nodes. Afterwards, the non-local target nodes are relocated to the position where the non-locals can be integrated in the target tree by means of separate relocation paths and subsequently integrated by means of integration policies. The relocation step merely moves the non-locals to the position in the target tree where they can be integrated. The integration step further integrates the non-local in this subtree. In the following subsections, we describe each of the steps. Due to the page limit, we are not able to discuss them in great detail.

3.1 Generation Step

Because rewrite rules are inherently 'in place' transformations, the target nodes described by the rewrite rules are the local target nodes. In order to include the non-local target nodes and distinguish them from the local target nodes, the rewrite rules attach the non-local nodes to a special `NONLOCALS(...)` node, which is in turn, together with the local target nodes, part of a cons cell:

```
KEY(NAME) =
   CONS(LOCAL( mcall( var(NAME) "getValue" ()) )
        NONLOCAL( "datamember" datamember( "Key" NAME) ))
```

We deliberately chose not to introduce a special notion for attaching and distinguishing non-locals for simplicity reasons. A crucial part of the system is that non-locals can cary any kind of information, in the above example the non-local is tagged with a name. This is heavily exploited by the semi-automatic composition system to identify the non-locals and to guide the integration process of the non-local.

3.2 Relocation Step

The relocation step is executed after the execution of all modular rewrite rules. In this step, the non-local target AST nodes are moved to their proper location in the target AST. The correct location for the non-local AST nodes is determined by (1) the grammar of the target language and (2) custom target program specific semantics. The relocation for non-locals, which correct location is based the grammar of the target language, is driven by the target language grammar and can even be automated. The relocation for non-locals, which correct location is based on custom semantics, needs to be written manually.

3.2.1 Automatic Relocation

The automatic relocation of a non-local target node tries to move the node to a location in the target tree where it can be integrated. In general, a non-local `NL` can be integrated in a node `E` if `NL` can be a child of `E` according to the grammar of the target language. The following relocation algorithm is executed for each non-local target node:

For each non-local target node `NL`, attached to a target node `E`, we first check if `NL` has a custom relocation path. If so, the non-local is moved to the position pointed to by the path. If there is no custom relocation path, we move the non-local `NL` upward in the target AST. The non-local `NL`, now becomes a non-local of the parent `E'` of `E`. If the non-local `NL` can be integrated in `E'` or in one of the indirect children of `E'`, the relocation process moves the `NL` to the correct child. The correct child is determined by a depth-first, left-to-right strategy. Otherwise, we keep on moving upward in the target AST until the root is reached.

Let us illustrate how this automatic relocation process works for the inputfield transformation. The state of the parse tree at that time is shown in figure 3. The inputfield transformation produced a couple of non-local statements and a non-local datamember. The automatic relocation process for the datamember will move the datamember upward in the tree until it finds a node which, according to the grammar of Java, can contain other datamembers. The first node which is capable of doing so, is the `class` node (line 1). Similarly, the automatic relocation process for the statements will yield, the `stats` node (line 3) (i.e. a node that holds several statements) of the `main` method node (line 3). The actual integration of the non-local nodes will be performed by the integration step explained in the following section.

Automatic relocation only works in cases where the location of the non-locals can be determined unambiguously or in cases when there is a simple satisfying strategy to disambiguate. Automatic relocation is very well suited for other domain specific languages and general purpose languages for constructs like classes, datamembers and methods. The location of expressions and statements cannot be unambiguously determined but the nearest possible location is often the desired one. However, automatic resolution fails for most types of non-locals in languages, like for example HTML, where most language constructs may be used in every other language construct. For these language constructs custom relocation must be used.

3.2.2 Custom Relocation

In case automatic relocation fails or the relocation is dependent on the particular semantics of the target program, the custom relocation must be defined by the user. Let us immediately explain the technique with an example.

Consider the non-local statement produced by the `key` transformation. The non-local statement (line 22 in figure 3), initializes the non-local datamember that was produced by the same transformation. The proper place of the initialization statement is at the beginning of the `main` method. The automatic relocation of the statement would lift the statement to the enclosing `defvalue` method body (line 18) instead of the `main` method body. Therefore, the following custom relocation path must be provided.

```
NONLOCAL(INITIALIZE, NODE) OF KEY =
    NONLOCAL( INITIALIZE, RELOCATE(NODE, //wizard))
```

The above rule will substitute the initialize non-local node (denoted by `NONLOCAL(INITIALIZE, NODE)` produced by transforming the `key` source AST node) into a non-local with holding the relocation path. The path `//wizard` points to the `main` method target AST node obtained by transforming the source AST ancestor `wizard`.

Custom relocation paths are attached to a non-local target node by means of a 'special' rewrite rule. In contrast to a normal rewrite rule, this 'special' rule is expressed in terms of the source language and refers to the name of the non-local and the original source node that got rewritten. The relocation paths are also expressed in terms of the source AST. As a consequence, the relocation paths express that the non-local node should be relocated to the top level target AST node, which is the result of transforming the source AST node pointed to by the custom relocation paths, and not the *exact location* in the top level target AST node. The further integration into the top level node is be accomplished by means of integration policies, which are discussed in the next section. The language to express the path is the XPath [1].

Expressing and attaching the relocation paths in terms of the source language does not suffer from the problems of the workaround (section 2.2). The advantages are threefold: increased expressiveness and robustness, and reduced complexity. The expressiveness of the paths benefits from the expressiveness of the source language. The paths are also more robust to changes to the produced target AST. No detailed knowledge is required to further integrate the non-locals into the produced top level node AST node, it is captured in separate integration policies.

All the non-local nodes of the local-to-global transformations of the WIZ language (except one non-local of the key transformation) can be relocated using the automatic relocation process. No hard to specify traversals and rewrite rules had to be written and no scheduling puzzles had to be disentangled. Even the specification of the location of the non-local statement (of the example above) only involved source language concepts instead of the former workaround's match and conditions clauses which heavily depended on the produced target nodes.

3.3 Integration Step

Now that the correct location of the non-locals in the parse tree has been found, the non-locals must be integrated in the target AST node on this location. Integration is necessary to ensure that the non-locals do not violate language constraints and obey the semantics of a particular integration policy. For example, the integration of the non-local statements, produced by the inputfield transformation, into the `main` method body must obey the specific user-defined order of the different statements (first creation of *all* inputfields, followed by the *attachment* of listeners and finally their initialization with default values). Also the integration of the non-local datamember, produced by the same transformation, into the `main` class must enforce the uniqueness constraint of the datamembers within a class.

During the integration of a non-local into a target AST node, the correspondence of the non-local with the already existing children is checked. If a correspondence is found, the two non-locals are combined with one another. The correspondence check facilitates many necessary and interesting mechanisms. The two most important mechanisms are the combination of partial results and the enforcement of uniqueness constraints [5].

Integration is a close interplay of three different policies: integration, correspondence and combination. The integration policy is called for each non-local that was relocated to an AST node. Its task is to integrate the non-local in the AST node itself or in one of its children. The correspondence policy is used by the integration policy to check if nodes in the AST need to be combined with the non-local. The correspondence policy is more like a predicate defined on nodes to check if they correspond. The combination policy performs the combination between nodes.

In our present solution, policies are implemented as normal rewrite rules.

Integration is specified by a triple of an `integration`, `correspondence` and `combination` node containing the name of the policy and if necessary some additional information. Similar to the custom relocation paths, the policies are attached to the non-local by means of the 'special' rewrite rule (cfr previous section).

The advantage of this schema is that the different policies can be effectively reused for all non-locals of the same type and can be specialized for particular non-locals. But the separation of the integration process in these three policies has also other advantages. Firstly, the three different policies are nicely separated in different concerns. Secondly, a default integration policy of the integration process can be automatically generated provided that the other policies are defined (see the example below).

The developer only needs to write correspondence and combination policies for the kind of non-locals that are produced by the rewrite rules. And given the reusability of the policies, the effort needed to perform the integration is reduced to a minimum.

Let us illustrate how the integration process for the nonlocal `datamember` node produced by the `key` transformation works.

```
NONLOCAL( DATAMEMBER , NODE) OF KEY = NONLOCAL( DATAMEMBER ,
      ((INTEGRATION(DEFAULT),
          CORRESPONDENCE(DEFAULT),
          COMBINATION(DEFAULT)) ,NODE))
```

The above 'special' rewrite rule attaches the default policies to the nonlocal named `datamember` produced by the `key` transformation, by rewriting the matched nonlocal `NODE` in a triple.

The default correspondence policy for datamembers stating that two datamembers correspond when their names and types are equal, is captured by the following `correspond` rewrite rule.

```
CORRESPOND(DEFAULT,
           DATAMEMBER(NAME,TYPE), DATAMEMBER(NAME,TYPE)) = TRUE
```

Subsequently the integration policy must be defined to integrate the `datamember` node into the `class` node. Below the standard and automatically generated integration policy is given for child nodes of a `class` target node. When the `correspond` rewrite rule succeeds (line 1-2), the two corresponding nodes `NODE2 and NODE` are combined together (line 8), otherwise the non-local `NODE` is just appended to the contents of the `class` node (line 12). The latter append action is the actual default strategy to integrate a non-local into a another structure.

```
(1)    NODE* = NODE1* NODE2  NODE2*
(2)    CORRESPOND(POLICY2, NODE2 , NODE) = TRUE
(3)    ===============>
(4)    INTEGRATE(DEFAULT,CLASS(NODE*),
(5)       NONLOCAL( NAME, ((INTEGRATION(DEFAULT),
(6)                    CORRESPONDENCE(POLICY2),
(7)                    COMBINATION(POLICY3)) ,NODE))
(8)  =  CLASS(NODE1* COMBINE(POLICY3, NODE2, NODE) NODE2*)

(9)    INTEGRATE(DEFAULT,CLASS(NODE*),  NONLOCAL( NAME ,
(10)        ((INTEGRATION(DEFAULT),
(11)          CORRESPONDENCE(POLICY2),
(12)          COMBINATION(POLICY3)) ,NODE))) =  CLASS(NODE* NODE)
```

4. DISCUSSION

The process of handling non-locals is divided into a number of separated modules: the generation, the automatic relocation, the custom relocation, and the three integration, combination and correspondence policies. The division into these modules has several advantages (1) the complex implementation of local-to-global transformations is disentangled in to manageable modules, (2) the impact of maintenance and future evolutions of one transformation is localized to the implementation parts of the modules of that transformation, and (3) each of the modules can be reused and even some of them can be automated.

For the above WIZ language only one custom relocation was needed, the rest of the relocation process was performed automatically. Also for other general purpose languages like Visual Basic and C++ experiments showed that the relocation significantly reduced the complexities involved with the implementation of local-to-global transformations. The automatic relocation process is not beneficial for languages like HTML where most language constructs can be composed with most of the other language constructs: the automatic relocation of those language constructs always yields their parents.

The automatic relocation process is now entirely driven by the grammar of the target language. We are currently looking into the idea, to guide the relocation process with other information and declarations instead of merely using the types of the nonlocals. As such, the developer may specify extra places in the target AST where a non-local can integrate, besides those already defined by the grammar. One possible approach is to use the correspondence rules of the integration phase. Another possibility would be to create extra declarations about the kind of node in the AST a particular kind of non-local node may integrate with. This would allow a far more fine grained and sophisticated relocation process. One possible extension would be to guide the automatic relocation of the variable declarations based on their use sites.

The integration of non-locals is incremental. Incremental integration is very flexible, since the parts that need to be integrated with one another must not be all available before the integration can start. The downside of this flexibility is that some integration strategies are much harder to write in an incremental way.

The validation of our approach is based on early experiments including the implementation of the compiler for the WIZ language used in this paper. Future work definitely includes a report on a complete industrial case study consisting of multiple evolved versions to further validate our approach.

The mechanisms proposed in this paper were implemented on top of the linglet transformation system (LTS) (previously known as the keyword based programming toolkit [4]). According to the component based DSL development philosophy of LTS, the linglets (language components) are standalone and modular components that do not depend on other linglets or engines to produce their results. Nor can a linglet access the whole target tree to do some computations. In other words, in contrast with rewrite rule systems, implementing tangled linglets and linglets that highly depend on the produced results of other linglets is not allowed in LTS. The modularity of the system immediately acted as additional validation mechanism, testing whether the implementation of our semi-automatic mechanism obeyed this. Since modularity reduces the dependencies, this modularity is a key enabler for a better maintainable and evolvable language implementation.

5. RELATED WORK

In the survey of Jonne van Wijngaarden et. al. [14] current day transformation systems are compared and classified according to three transformation mechanics i.e. scope, direction and stage. Transformations with a wide target scope and a single node source scope are called local-to-global transformations. The survey explored whether the basic mechanisms like querying and traversing the parse tree are available in current day transformation systems in order to implement local-to-global transformations.

In this paper we went a step further and investigated the repercussions of those mechanisms on the complexity, maintainability and evolvability of DSL implementations. We found that those mechanisms easily result into over-complex implementations and severely constrain maintainability and evolvability.

ASF+SDF [12] , Stratego [15], XSLT [3] provide a very powerful traversal system but the traversals to locate, collect, and redistribute the non-locals must be written against an intermediate parse tree consisting only of target nodes. Consequently the implementation easily results in a tangled web of traversals and rewrite rules as was discussed in section 2.2. Attribute grammars [7] and similar systems like intentional programming [10, 11] have an implicit query system using inherited and synthesized attributes. The drawback of this mechanism is the lack of control over the source of the information, a feature which is vital for custom relocation strategies and for the specific integration of a non-local. Furthermore to prevent cylces, the implicit scheduling mechanism for the computations of attributes would require to decouple the production of the local from the non-local results. This decoupling has some drawbacks which were discussed and tackled in [15].

Also in contrast to our approach none of the current day systems exploit the grammar to provide a semi-automatic relocation and composition system, and none of the systems provide an adaptable and modular integration mechanism that is driven declaratively.

Nevertheless, the solutions presented in this paper are complementary to the above transformation systems and can be implemented on most of them, provided that they support the basic traversal and query mechanisms. We were able to successfully implement this architecture on top of the linglet transformation system [4]. The architecture could even be implemented on top of those systems which are not based on rewrite rules e.g. Jostraca [9] and JTS [2].

6. CONCLUSION

Transformation systems, in particular rewrite rule systems, fail to deal with local-to-global transformations without corrupting maintainability and future evolvability. Our solution architecture enables the use of modular and simple rewrite rules to implement local-to-global transformations hereby avoiding complex, tangled traversals and rewrite rules and scheduling nightmares. The non-local target nodes are moved to their correct position using a semi-automatic relocation mechanism and subsequently integrated by integration policies in the tree. The result is a general mechanism on top of the rewrite rule paradigm to deal with the non-local results of transformation systems where the above dependencies are reduced to a strict minimum. This has a positive impact on the maintainability and future evolvability of the implementation. The disentangling of the complex implementation of local-to-global transformations (1) localized the impact of maintenance and future evolutions of one transformation to the implementation parts of that same transformation i.e. generation, relocation and integration and (2) limited the impact of maintenance and future evolutions is to specific modules and (3) enables great reuse potential.

7. ACKNOWLEDGMENTS

We would like to thank Johan Brichau, Tom Tourwe, Johan Fabry and Kris Gybels for comments on this paper.

8. REFERENCES

[1] S. B. Anders Berglund. Xml path language (xpath) 2.0 w3c working draft 15 november 2002, 2002.

[2] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE.

[3] J. Clark. Xsl transformations (xslt) version 1.0 w3c recommendation 16 november 1999, 1999.

[4] T. Cleenewerck. Component-based dsl development. In *Proceedings of GPCE'03 Conference, Lecture Notes in Computer Science 2830*, pages 245–264. Springer-Verlag, 2003.

[5] T. Cleenewerck. A semi-automatic composition system for local-to-global transformations. Technical Report PRG-RR-01-21, Programming Technology Lab (PROG), Vrije Universiteit Brussel, 2004.

[6] J. Cordy, T. Dean, A. Malton, and K. Schneider. Software engineering by source transformation - experience with txl. *SCAM'01 - IEEE 1nd International Workshop on Source Code Analysis and Manipulation*, pages 168–178, November 2001.

[7] D. E. Knuth. Semantics of context-free languages. In *Mathematical Systems Theory*, pages 168–178, 1968.

[8] N. J. M. The draco approach to constructing software from reusable components. In C. Rich and R. C. Waters, editors, *Artificial Intelligence and Software Engineering*, pages 525–535, 1986.

[9] R. J. Rodger. Jostraca: a template engine for generative programming. position paper for the ecoop2002 workshop on generative programming, 2002.

[10] C. Simonyi. The death of computer languages, the birth of intentional programming, 1995.

[11] C. Simonyi. Intentional programming - innovation in the legacy age, 1996.

[12] M. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual.* Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, July 2002.

[13] M. G. J. Van Den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[14] J. van Wijngaarden and E. Visser. Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Universiteit Utrecht, 2003.

[15] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.