Implementing Tag-Driven Transformers with Tango

Vasian Cepa *

Software Technology Group
Department of Computer Science
Darmstadt University of Technology, Germany
cepa@informatik.tu-darmstadt.de

Abstract. Using explicit tags at the programming language level has been attracting a lot of attention recently with technologies like xDoclet [41] which automates EJB [30] related tasks, and .NET attributes [21] which are a vital part of the .NET framework [34]. However there are currently no systematic ways for adding and transforming tag-driven product line specific constructs. This results often in implementations which are not very modular and hence that are difficult to reuse.
In this paper we introduce Tango a specialized language for implementing source to source tag-driven transformers for object-based languages in a systematic way using several layers of abstraction. Tango operates on a fixed, language independent, object-based metamodel and divides the transformation strategy in several well-defined layers which makes it also possible to reuse lower level parts. Tango uses the so-called *inner tags* to communicate semantics between different transformer modules in a uniform way.

1 Introduction

Using tags is an intuitive way to associate custom semantics with elements of interest. Variations of the tag concept have been used all around in computer science. Explicit tags offer a convenient way to quickly customize a language with domain specific constructs reusing its existing features and front-end compiler tools [39]. The programmer does not need to know the way the grammar rules evolve, which makes it easy to introduce domain specific customizations for application families, or proofs of concepts, in a uniform way. This is preferable to *extensible grammar* [24] approaches, which allow a grammar to evolve by adding new constructs which are then mapped to the original kernel grammar via add, update, and delete operations on existing production rules.

We will use the term *tagged grammars* for language grammars designed to support explicit tags directly as part of the language. Several general purpose

* Mira Mezini, Sven Kloppenburg, and Klaus Ostermann provided useful comments and suggestions about the early drafts of this paper. The anonymous reviewers comments also proved very useful for producing the final version.

languages like .NET [34] and Java (JSR 175 [40]) already offer such support. Tags can be associated with any existing structural element in these languages, like classes and methods. They are part of the structural element representation in the AST. Some language technologies, like .NET with its CodeDom API [1], offer supported programmatic access to the annotated source via a generalized AST, which can be used to write pre-processing tools that deal uniformly with explicit tags interpretation [2]. For language technologies that do not offer explicit tag support, tags can be emulated with special comments.

There are however several drawbacks when using tags which relate to the nature of automatic transformations. Usually the semantics of a tag are unclear unless we have full documentation about the tag, that is to know exactly how the tag-driven transformer will handle the tag. The side-effects that a tag transformer introduces also present a problem. Despite the benefits of using explicit tags, these problems make it difficult to use tags in cases when interactions of transformers [35] must be taken into account. It is important then to implement tag transformers in ways that allow grasping their transformation strategy quickly, so it becomes easier to understand and reuse them.

There are several good general purpose transformation frameworks like Stratego [10], DMS [13] and TXL [19] which could be used to address domain specific transformations. These general purpose frameworks have an open metamodel meaning that, we can map any specific language to them. In this context such tools could also be used with explicit tags. However the generality of these frameworks offer no way to abstract the transformation strategy according to a domain of interest. If the domain is fixed then the chances of abstracting the transformation strategy in different levels grow. We introduce here Tango [3] a specialized framework for implementing source to source tag-driven transformers. Tango supports only languages whose metamodels conform to a generalized object-oriented (OO) class based metamodel with explicit tag annotations. Not every transformation is possible with this model. However Tango's common metamodel is useful for many transformations that appear in context of tag specialized constructs for product line [4] applications.

2 Tango Framework Concepts

Tango framework is designed for quickly adding tag-based product line [18] domain specific constructs to existing object-oriented (OO) languages, reusing their core functionality.

Fixed Metamodel - Using tags to decorate class entities is similar in different OO languages. Tango achieves language independence using two concepts which encapsulate language dependent details:

[1] A third party implementation for C# is CS CodeDom Parser [14].

[2] Alternatively tags are saved as part of metadata and could be later processed using reflection like API-s.

[3] The name *Tango* was coined in the following way: tag → tag-go → taggo → tago → tango.

First, all Tango transformers work with a unified internal meta-graph model of input source, called *Class Template* model. It is a generalized and tag annotated common model of the structural AST for a generic OO language. The model is focused on the class construct and is restricted on the structural elements it can contain. Currently it supports only classes, field attributes and methods. This model has been used to define product line specific constructs in MobCon [29], a framework or mobile applications. How a specific language source code is mapped to this model is not addressed directly by Tango. Such mappings can be complicated to implement for a given language, but we can take advantage of the fact that tag-driven transformations are often used sparingly in a project only in well-defined component structures. Only the parts of the metamodel needed by the transformations could be mapped, ignoring for example full namespace support. Advanced users may extend the framework and change its metamodel by mapping new language features to it.

Second, Tango's abstractions are isolated from the rest of the transformer implementation via the concept of a *'code snippet'* [4], which is similar to Stratego's [10] *concrete syntax*. A code snippet is a source code node atomic to Tango. It is the only part of code which is language grammar dependent. Tango requires that the implementation has a way to replace parameters inside the code snippet using templates. An example of a template code snippet that uses Apache Velocity [20] script language is shown in Fig. 1. This code generates the body of method *toRecord* in Fig 4. The details of Velocity language are however outside the scope of this paper. Code snippets allow representing parameterized clusters

```
code StoreFields ($fieldArray) language (Velocity) {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream outputStream = new DataOutputStream(baos);
    #foreach($field in $ fieldArray)
    outputStream.$dp_write($field.type)
        (this.$Tango.makeMethodName(["get", $field.name]()));
    #end
    return baos.toByteArray();
    #macro(dp_Write, $type)
        if($type == string) UTF
        if($type == int) Int
    #end
}
```

Fig. 1. Code Snippet Example

of source graph nodes without dealing with details of such nodes in the rest of transformer implementation. Similar approaches include syntactic unit trees [26] and various template and macro based approaches [15, 16]. Tango code snippets differ from such approaches because they are restricted in representing only non-structural elements for example a block of code inside a method. They can also

[4] This term is borrowed from .NET CodeDom API [31].

be labeled with inner tags (see below) and reprocessed later as atomic elements of the AST.

Controlling Semantics with Inner Tags - Tango introduces the concept of *inner tags* which are a natural extension of explicit tags. They are used only in the inner operations of tag-driven transformers and offer convenient means for specifying the *coupling sets* between composed transformers by removing the need to reinterpret the code. Inner tags are similar to ASF+SDF [25] *placeholders* for saving intermediate results. However inner tags offer a generic uniform model of controlling custom semantics integrated uniformly with the rest of tag-driven transformer operations in Tango. Programmers deal with inner tags in the same way they process explicit tags. All Tango's basic edit operations can specify inner tags to the entities they modify.

While inner tags place inter-transformer interaction semantics, we should note that the alternative is to re-process the AST in each transformer to see if it fulfills a given condition. Using inner tags does not grow transformer coupling which remains the same. Inner tags only make the coupling declarative, avoiding reprocessing. In this context inner tags are used to create arbitrary graph node sets [28], which may not correspond directly to the generalized AST graph nesting structure. With inner tags we are also able to associate more than one label with a node (group) and select the node in more than one set.

Layers - Tango's fixed metamodel allows dividing the transformation strategy in several layers corresponding to the elements of the metamodel. This allows reasoning in different levels of abstraction when we want to understand the transformation strategy. A transformer implementation in Tango is organized in several hierarchical layers: *the workflow layer*, *the transformers layer*, *the member constructors* and *the code snippets layer* as show in Fig. 2. Each layer uses the

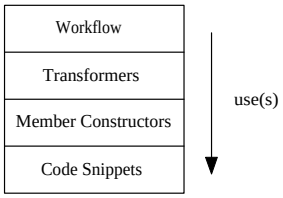

Fig. 2. Tango Framework Layers

elements of the successive lower layer, but cannot create elements of the upper layer. For example class templates are only created in the workflow layer. The transformer layer can only modify the class templates but not create new ones. This way we can know what class templates take part into a transformation only by examining the workflow layer. In the same trend member constructors (see Section 3) are used in the transformers layer only. Finally code snippets are used only by member constructors.

Each layer defines part of the transformation strategy customized to its elements. For example the workflow strategies are similar to Stratego's [10] transformation strategies but simpler. Unlike Stratego which is a general purpose transformation framework Tango's workflow operates only on class templates. The simplicity is a consequence of the specialization for class layer of the metamodel. Strategies of the other lower layers also correspond roughly to some Stratego [10] transversal strategies.

Traceability - Traceability is integrated in the Tango framework. It can be turned on and off for pieces of source code of interest by using a special explicit tag directly in source code. When present, this tag instructs the framework to add special log methods to all methods that are transformed by any Tango transformer. The log methods are decorated with special tags containing all the transformer names which have edited the method. When output code is executed, the trace statements are printed on the console, allowing to know for each executing method its exact transformation history. This centralized tracing capability helps debugging transformation related side-effects.

3 A Feeling of Tango: Transformation Example

In this section we introduce transformer implementation in Tango using the example of Fig. 3 which will be transformed to the code of Fig. 4. This example is a tag based implementation of the standard *GameScore* example which comes with J2ME MIDP [6] documentation. The input code fields are decorated with explicit tags in forms of JavaDoc [33] comments. Tags will be part of generalized AST field elements. The following attribute classes have been used in the code of Fig. 3: (a) *property* - adds accessor / mutator methods for a field (b) *validate* - adds validation code for fields that have mutator methods; *min, max* show the required range for an integer or the required length ranges for a string field (c) *dp* - adds data persistence methods to the component and allows the records to be retrieved sorted. For more details see [29]. The details of the specific Tango syntax shown here may change as the framework is still being developed. Check [36] for the latest version.

The Workflow - The workflow layer defines the structure of a composite transformer at the class level. The workflow operations work only with *class templates* and *transformers*. Tango transformers are functions of form $\tau : < ct_{i1}, ..., ct_{in} > \rightarrow < ct_{o1}, ..., ct_{om} >$, that is they take one or more class template arguments and output one or more class templates. This allows us to compose transformers uniformly, unlike other systems like JMangler [12] that make a distinction between individual transformers and *mergers*. Tango transformation workflow for the example above is listed in Fig. 5. The class template is initialized from a source file in line 2. The implementation of the transformer is split based on the three classes of attributes that are present: *Property, Validation, Persistence*. The composition operator (',') is used in line 3. This composition is equivalent to the following functional representation: '`$CT = Persistence(Validation(Property($CT)));`', which can also be used

```
/**
 * @dp
 * @validate
 * @property
 */
public class GameScore {
/**
 * @dp.pk
 * @validate.min 0
 * @validate.max 100
 * @dp.sort asc
 * @property.accessor
 */
private int score;
/**
 * @dp.pk
 * @validate.min 4
 * @validate.max 32
 * @dp.sort asc
 * @property.both
 */
private String playerName;
/**
 * @property.both
 */
private String Comment;
}
```

Fig. 3. Input Code

```
public class GameScore {
private int score;
private String playerName;
private String Comment;

public getScore() { return score; }

public String setPlayerName(
    string value)
{ playerName = value; }

// ...
public byte[] toRecord()
{
ByteArrayOutputStream baos =
    new ByteArrayOutputStream();
DataOutputStream outputStream =
    new DataOutputStream(baos);
outputStream.writeInt(o.getScore());
outputStream.writeUTF(
        o.getPlayerName());
outputStream.writeUTF(
        o.getComment());
return baos.toByteArray();
}
// ...
}
```

Fig. 4. Output Code

```
1. @using "example.tgt"
2. $CT = read("input.java");
3. $CT = Property, Validation, Persistence;
4. write($CT, "output.java");
```

Fig. 5. Example Workflow File

directly. The implementation of transformers is read from file *'example.tgt'* (line 1). The transformed class template is saved to source code in line 4.

Other supported workflow operations not shown in this example include: the try operation '`$CT = ?(T1, T2 | T3);`' - where '`T3`' is tried only if '`T1, T2`' fails; the creation of empty class templates '`$U = CT("Utility");`' - creates a new class template named `Utility`, and cloning: '`$CT1 = CT2`' - that sets `CT1` to be a deep copy (not a reference) of `CT2`.

Transformers - Fig. 6 shows the complete implementation of the *Property* transformer in Tango. Each transformer has a *precondition* part and an *action* part. The editing operations are allowed only in the *action* part. The split is intended to specify in the beginning the main checks whether a transformer can be applied or not. The *noapply* operator (line 4) tells Tango workflow that this transformer cannot be applied. If a transformer fails to apply the control is returned to the workflow. Theoretically we can define a precondition and a postcondition for any graph rewriting operation [28], but in practice it is cumbersome to enumerate them for each operation (postconditions can be written as preconditions [28]). Tango allows factorizing some preconditions before all the

```
1. transformer Property(~ct) {
2. precondition {
3.     if(not check(~ct, tags(["property"])))
4.         noapply;
5. }
6. action {
7.     $fields = select(~ct, FIELDS, tags(["property.*"]);
8.     if(check($fields, empty())) error;
9.     iterator($field in $fields) {
10.        if(check($field, tags(["property.accessor"])
11.            or tags(["property.both"])))
12.            add(~ct, METHOD, GetMethod($field),
13.                [tag(<"property.accessor", $field.name>)]);
14.        else if(check($field, tags(["property.mutator"])
15.            or tags(["property.both"])))
16.            add(~ct, METHOD, SetMethod($field),
17.                [tag(<"property.mutator.", $field.name>)]);
18.    }
19. }
20. return ~ct; // optional the first argument is returned
21. }
```

Fig. 6. The Property Transformer Implementation

actions, but this convenience results in an *optimistic* precondition requirement that is, the *action* part can also make more specific checks and the transformer may still fail as a consequence of a failed condition in the *action* part.

The argument of the transformer is a class template (`~ct`) (line 1). The actions part will modify this class template to add getter and setter methods for all fields decorated with some form of *property* tag. The fields are selected in line 7 and stored in (list) variable `$fields`. The generic *select* operation is used to filter a set of nodes of the same type that fulfill a given condition. Only predefined types of the supported metamodel like `FIELDS, METHODS` etc, can be used. The third argument of *select* is a *condition*. Several predefined conditions are supported: 'tag' - filters nodes based on tags, 'name' - filters based on names, 'empty' - checks for empty list, and 'count' - checks the number of list items. For conditions that expect string arguments, regular expressions are supported. Users can define additional custom conditions.

The *check* operation (line 8) is similar to *select*. It applies all the conditions given to it to the first argument and returns a boolean value indicating whether conditions have succeeded or not. The individual conditions can be combined using boolean operators: `and, not, or`. The *check* and *select* operations pass the first argument implicitly to all conditions. This makes it easier to understand what a *check* or *select* statement does, given that all conditions work on the first argument of *check* or select, at a cost of a slightly slower implementation.

The *iterator* operation (line 9) is used to apply an operation over all elements of a list. Note that *iterator* and *select* could be a single operation, however it makes sense to separate these operations in cases when we need to process lists, other than those returned by *select*. We have divided the iterations over the metamodel AST between the 'check', 'select', and 'iterate'. The *check* and *select* operators do implicit iterations over *finite* lists of elements. Again, this is

preferable because it makes it easier to understand what a loop does, at the cost of slightly less efficient implementation.

The *add* operation (line 12, 16) is an example of an edit operation. It adds a meta element to the class template given as its first argument. `GetMethod` and `SetMethod` are member constructors (see below). All supported edit operation in the transformers layer (*add, delete, modify*), work upon class templates and use member constructors of the lower layer. In the example the *add* operation also adds an inner tag to the newly added element: `[tag(<"property.accessor", $field.name>)]`. The tag is created by the explicit tag constructor *tag*, combining the field tag type and the field name using the string concatenation operation `<...>`. This inner tag can be used later in a *select* or *check* statement of another transformer, in the same way as an explicit tag.

Note the pattern used in *add* and *select*. We require the type of meta elements to be given explicitly. In the case of *add*, the type could have been deduced from the member constructor type. We do this because it improves the readability of the code. An alternative would be to have distinct operation names for each meta element type, like *addMethod*. However this would require to maintain the Tango parser if its fixed metamodel is ever customized. The modified class template is returned in line 20. For an implementation of the two other transformers of this example see [36].

Member Constructors Layer - This layer defines the actual metamodel member implementations that are used in the transformers layer. Class templates cannot be passed as arguments to the member constructors. Currently Tango defines member constructors for fields, methods and tags. Method bodies are represented as a list of tag decorated code snippets divided into three logical blocks: begin, middle and end (Fig. 7). When an existing method is parsed in code, its method body is represented as a single code-snippet node in the middle block. *Inner tags* can be used to decorate snippets of code. Users can select

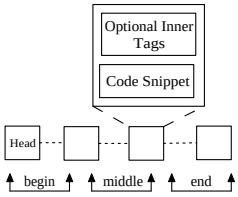

Fig. 7. Method Body as a List

```
1. method ToRecord($fieldArray) {
2.   methodName(makeMethodName(
       ["toRecord"]));
3.   methodReturn(ByteArray());
4.   addBody(StoreFields($fieldArray));
5. }
```

Fig. 8. Method Constructor Example

the elements by their tags and insert new or remove existing code snippets in any block part of the method body list. Fig. 8 shows a member constructor for a method used in the persistence transformer. No direct code is written in this layer, instead, code snippet constructors like *ByteArray* and *StoreFields* are called (lines 3, 4). The method name for a newly created method is set in line 2 and its return type is set in line 4 (as a string of code returned by a code

snippet). The method body is produced by code snippet constructor *StoreFields* (Fig. 1) . The *addBody* operation adds this code snippet at the end of the middle block.

4 Related Work

Variations of tags have been used in many areas in computer science including transformation tools. Tags can be used to reduce [5] transformation search space [22]. Here we are interested in explicit tag usage at the source code level, as a reusable way to introduce product line domain specific extensions to existing general purpose OO languages.

Tango reflects previous experience with a tag-based framework implemented as part of MobCon [29], a specialized generative framework for Java 2 Micro Edition, Mobile Information Device Profile (J2ME MIDP) [6]. Several concepts generalized in Tango like the fixed metamodel and inner tags were used in MobCon use case. Tango adds a layered structure and a specialized syntax to ease using such constructs.

Tango's relation with generic open transformation systems like Stratego [10], TXL [19], DSM [13], ASF+SDF [25] was stated in several places in this paper. Unlike such open AST systems, which can be seen as open generalized compilers, Tango works with a fixed tag annotated object-based metamodel. How specific metamodel mappings are implemented is not directly addressed by Tango. Tango also divides the transformation strategy in several abstraction layers.

Multi-stage programming [39] approaches use special annotations to deal uniformly with multi-stage transformations. These approaches are more driven by the need for optimizations and have a fixed set of well-defined annotations. Tango on the other hand, can be used to implement invasive transformers [3] at a higher level using a *Generalized and Annotated AST* (GAAST) [37] representation of the code, which is an AST-like API that preserves the annotations done at source level and allows accessing them programmatically. GAAST tools belongs to the category of API based generators [27].

Other approaches that generalize some of the ways a transformer works with an AST-like representation: (a) *Filtering related approaches* like JPath [32] that are motivated by XPATH [42] node query language for XML [2]. The idea is to be able to define XPATH like queries over the AST to declaratively select sets of nodes of interest. For example the query '`//class[@name="Class1"]/Method1`' selects and returns all methods named *Method1* found in class named *Class1*; (b) *Mapping related approaches* which build upon filtering related approaches and that are motivated by XSLT [8] and more recently by MOF Query View and Transformation [9] proposals. These approaches define declaratively how to map a set of selected nodes to another set of nodes, enabling this way transformation of one graph to another. These approaches are very general and can be used to implement Tango on top of them.

According to feature based categorization of MDA [7] transformation approaches given in [23], Tango falls into a hybrid of *code-to-code* and *template*

transformer with source-oriented deterministic iterative rules. However given that the transformation of a marked model to marked code is trivial, the approach can be seen also as *model-to-code*. The categorization in [23] is however too general, and many tools could fall into the same category. Tango is unique for its well-structured layers, extensibility of primitive operations, and heavy reliance on inner tags.

Tango can be seen as a domain specialization of graph rewriting systems [28]. We can think of changes that a tag-driven transformer introduces to an annotated class as series of primitive operations. Given that the number of structural elements in a general purpose language is limited, it makes sense to enumerate such operations. If we have a notation for such operations and write a sequence of such high-level operations, we will end up with a *specific language for implementing tag-driven transformers*. The approach is similar to implementing graph based schema evolutions [17], but specialized for OO tag-driven transformers.

Aspect Oriented Programming (AOP) [11] deals with ways to modularize crosscutting concerns systematically. Tags can be seen as a way to specify joinpoints. Selecting tags in a language that supports them can be done with an aspect-oriented enabled tool in the same way as choosing other code elements. However custom tags enforce an explicit programming [1] model. They are used conventionally similar to method calls. This is not a limitation in our case because we use tags to implement domain specific extensions that result in generative mappings back to the core language.

5 Conclusions and Future Work

Custom explicit annotations offer an attractive way to introduce product line [18] domain specific constructs uniformly to general purpose languages, reusing their front-end compiler tools. This convenience has however several problems related to understanding transformations of the code decorated with tags. The answer to understanding the implementation relies on being able to reason about the strategy in various levels of abstraction that correspond to the language model. Tango is a specialized framework for addressing these problems for tag-driven transformers that work with a fixed object-based metamodel. Tango uses layering and restrictions of several operations to promote understandability over efficiency. This not only grows chances of individual transformer reuse, but also enables reuse of elements of lower layers in more than one element of the more abstract upper layers. Tango's inner tags allow treating transformer semantics in a uniform way during composition.

Future work will be concentrated on finalizing Tango grammar and improving the prototype. Possible areas of future work are: expressing and checking tag dependencies declaratively [38] and introduction of non-intrusive transformers.

References

1. K. De Volder G. C. Murphy A. Bryant, A. Catton. Explicit Programming. *In Proc. of AOSD '02, ACM Press*, pages 10–18, 2002.

2. M. Gudgin A. Skonnard. *Essential XML Quick Reference.* Addison-Wesley, 2001.
3. U. Aßmann. *Invasive Software Composition.* Springer-Verlag, 2003.
4. D. Batory, R. Cardone, and Y. Smaragdakis. Object-Oriented Frameworks and Product Lines. In P. Donohoe, editor, *Proceedings of the First Software Product Line Conference*, pages 227–247, 2000.
5. T. J. Biggerstaff. A New Control Structure for Transformation-Based Generators. *Proc. of 6th International Conerence on Software Reuse*, 2000.
6. C. Bloch and A. Wagner. *MIDP 2.0 Style Guide for the Java 2 Platform, Micro Edition.* Addison-Wesley, 2003.
7. D. S. Frankel. *Model Driven Architecture - Applying MDA to Enterprise Computing.* Wiley, 2003.
8. S. St.Laurent D. Tidwell. *XSLT.* O'Relly, 2001.
9. DSCT, IBM, CBOP. MOF Query / Views / Transformations. *Initial Submition, OMG Document*, 2003.
10. E. Visser. Stratego: A Language for Program Transformations Based on Rewriting Strategies. *Rewriting Techniques and Applications (RTA'01), LNCS 2051*, pages 357–361, 2001.
11. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming. *In Proc. ECOOP '97, Springer-Verlag, LNCS 1241*, pages 220–243, 1997.
12. M. Austermann G. Kniesel, P. Costanza. JMangler - A Powerful Back-End for Aspect-Oriented Programming. *In R. Filman, T. Elrad and S. Clarke, M. Aksit (Eds.): "Aspect-oriented Software Development", Prentice Hall*, 2004.
13. I. D. Baxter, C. Pidgeon, M. Mehlich. DMS: Program Transformations For Practical Scalable Software Evolution. *Proc. of International Workshop on Principles of Software Evolution*, 2002.
14. I. Zderadicka. CS CODEDOM Parser. *http://ivanz.webpark.cz/csparser.html*, 2002.
15. Yuuji Ichisugi. Extensible Type System Framework for a Java Pre-Processor: EPP. *staff.aist.go.jp/y-ichisugi/epp/*, 2000.
16. K. Playford J. Bachrach. The Java Syntactic Extender: JSE. *In Proc. of OOPSLA '01, ACM SIGPLAN Notices, Volume 36, Issue 11* , pages 31–42, 2001.
17. W. Kim J. Banerjee. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *Proc. of ACM SIGMOD International Conference on Management of Data*, 1987.
18. J. Bosch. *Design and Use of Software Architectures, Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2002.
19. J. Cordy. TXL - A Language for Programming Language Tools and Applications. *Proc. of 4th Workshop on Language Descriptions, Tools and Applications, LDTA*, 2004.
20. J. Cole J. D. Gradecki. *Mastering Apache Velocity.* John Wiley & Sons Inc, 2003.
21. J. Newkirk and A. Vorontsov. How .NET's Custom Attributes Affect Design. *IEEE SOFTWARE, Volume 19(5)*, pages 18–20, September / October 2002.
22. J, van Wijngaarden, 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. Institute of Information and Computing Sciences, Utrecht University*, 2003.
23. S. Helsen K. Czarnecki. Classification of Model Transformation Approaches. *Proc. of 2nd OOPSLA Workshop Generative Techniques in the context of MDA*, 2003.
24. M. Abadi L. Cardelli, F. Matthes. Extensible Syntax with Lexical Scoping. Technical Report 121, Digital Equipment, 1994.

25. M. G. J. van der Brand, J. Heering, P. Klint, P.A. Oliver. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems, Vol. 24, No. 4*, pages 334–368, 2002.
26. B. Franczyk M. Majkut. Generation of Implementations for the Model Driven Architecture with Syntactic Unit Trees. *Proc. of 2nd OOPSLA Workshop Generative Techniques in the context of MDA*, 2003.
27. M. Voelter. A collection of Patterns for Program Generation. *Proc. EuroPLoP*, 2003.
28. T. Mens. *A Formal Foundation for Object-Oriented Software Evolution*. Ph.D Disseration / Vrije University Brussel, 1999.
29. MobCon Homepage. *http://www.st.informatik.tu-darmstadt.de/static/pages/projects/mobcon/%index.html*, 2003.
30. R. Monson-Haefel. *Enterprise JavaBeans*. Addison-Wesley, 2000.
31. N. Harrison. Using the CodeDOM. *O'Reilly Network* *http://www.ondotnet.com/pub/a/dotnet/2003/02/03/codedom.html*, 2003.
32. P. W. Calnan. EXTRACT: Extensible Transformation and Compiler Technology. *Master of Science Thesis, Worcester Polytechnic Institute,* *http://www.wpi.edu/Pubs/ETD/Available/etd-0429103-152947/*, April 2003.
33. M. Pollack. Code Generation using Javadoc. *JavaWorld,* *http://www.javaworld.com/javaworld/jw-08-2000/jw-0818-javadoc.html*, August 2000.
34. J. Prosise. *Programming Microsoft .NET*. Microsoft Press, 2002.
35. M. Suedholt R. Douence, P. Fradet. Detection and Resolution of Aspect Interactions. *ACM SIGPLAN/SIGSOFT Proc. of GPCE*, 2002.
36. Tango Homepage. *http://www.st.informatik.tu-darmstadt.de/static/staff/Cepa/tango/index%.html*, 2004.
37. V. Cepa, M. Mezini. Language Support for Model-Driven Software Development. *To Appear in (Edior M. Aksit) Special Issue Science of Computer Programming (Elsevier) on MDA: Foundations and Applications Model Driven Architecture*, 2004.
38. .NET Attribute Dependency Checker Tool Homepage. *http://www.st.informatik.tu-darmstadt.de/static/staff/Cepa/tools/adc/i%ndex.html*, 2003.
39. W. Taha, T. Sheard. Multi-stage Programming. *ACM SIGPLAN Notices*, 32(8), 1997.
40. JSR 175: A Metadata Facility for the Java Programming Language. *http://www.jcp.org/en/jsr/detail?id=175*, 2003.
41. xDoxclet Code Generation Engine. *http://xdoclet.sourceforge.net/*, 2003.
42. XML Path Language (XPath). *http://www.w3.org/TR/xpath*, 1999.