

Reusable and Adaptable Strategies for Generative Programming

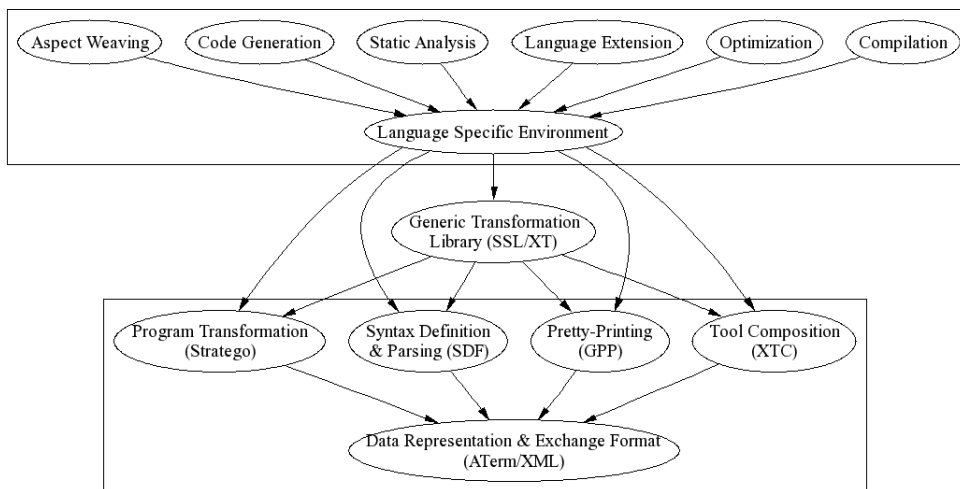
Position paper for the GPCE'04 Software Transformation Systems Workshop

Martin Bravenboer and Eelco Visser
Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089 3508 TB
Utrecht, The Netherlands {martin,visser}@cs.uu.nl

July 2004

Generative programming aims at increasing programmer productivity by automating programming tasks using some form of automatic program generation or transformation, such as code generation from a domain-specific language, aspect weaving, optimization, or specialization of a generic program to a particular context. Key for achieving this aim is the construction of *tools* that implement the automating transformations. If generative programming is to become a staple ingredient of the software engineering process, the construction of generative tools itself should be automated as much as possible. This requires an infrastructure with support for the common tasks in the construction of transformation systems. In the **Stratego/XT** project we have built a **generic infrastructure for program transformation**. In this position paper we give an outline of this infrastructure and indicate where we think the challenges for research and development of program transformations systems lie in the coming years.

Stratego/XT and its applications are organized in five layers (see diagram). **(1)** At the bottom layer is the **substrate** for a transformation system, that is the data representation and exchange format, for which we use the Annotated Term Format (ATerm) as basis, and XML where necessary to communicate with external tools. This substrate allows transformation systems to be rigorously componentized. **(2)** The **foundation** of any transformation system are syntax definition and parsing, pretty-printing, program transformation, and tool composition. The syntax definition formalism SDF provides modular syntax definition and parsing, supporting easy combination of languages. The pretty-printing package GPP supports rendering structured program representations as program text. The program transformation language Stratego supports concise implementation of program transformations by means of rewrite rules and programmable strategies for control of their application. Finally, the XTC library supports composition of simple transformation tools into complex ones. These are all generic facilities that are needed in any



transformation for any language. **(3)** In the middle is a library of transformations and transformation utilities that are not specific for a language, but not usable in all transformations either. The Stratego Standard Library (SSL) provides a host of generic rewriting strategies, and the XT toolset provides utilities for generating parts of a transformation system. **(4)** Near the top are specializations of the generic infrastructure to specific object languages. Such a **language specific environment** consists of a syntax definition for a language along with utilities such as semantic analysis, variable renaming, and module flattening. **(5)** Finally, at the top are the actual **generative tools** such as compilers, language extensions, static analysis tools, and aspect weavers. These tools are implemented as compositions of tools from the lower layers extended with one or more components implementing the specific transformation under consideration.

Where are we now in this development? The basic infrastructure (bottom two/three) layers is well established, readily available, and deployed in several projects. Although these components are gradually improved and extended, they form a reasonably stable development platform. The **next frontier** is the expansion of the infrastructure to generative applications. The tools in the top-level layer are the ones that matter since they are to be used by **application programmers**. Although we have also built a range of prototype language specific environments and generators to experiment with and validate our infrastructure, these do not yet form a product line by themselves. The main challenges to larger scale deployment are availability of *standard language specific components*, lack of documented *reusable strategies* at all levels of granularity, and an approach to make transformation *strategies adaptable*.

Although the languages and tools provided by Stratego/XT make the construction of transformation systems a lot easier than doing it from scratch using a general purpose programming language, it can still be a lot of work. Before a particular generative tool can be created, the hurdle of creating a language-specific environment must be taken. Therefore, the routine production of generative tools requires a library of **standard language specific components**, that is, front-ends for many standard and non-standard languages. The expectation that such front-ends will appear as spin-offs from production of specific generative tools is unjustified. The hardness and tediousness of this problem seems to be caused by the sheer complexity and irregularity of programming languages. Nonetheless, investigation of a higher-level and more declarative solution to the implementation of front-ends for real programming languages would be worthwhile.

Another challenge for transformation systems is to raise the level of reuse by providing **reusable strategies** at all levels of granularity, from micro-level transformations to macro-level tool compositions upto *process strategies*. As with all reuse, a sensible documentation and indexing of available solutions is key to benefitting from the available components. This is clearly a problem for new and even for more experienced users of Stratego/XT. With some 170 executable components and over 1200 strategy definitions in the library, it can be hard to locate the right ones for a specific transformation. While the available components form a very expressive programming environment, supporting the easy composition of all kinds of transformation systems, it requires knowledge of the available components and the ways to compose them. The cause of this problem is that the available components live at a lower level of abstraction than the problem being solved. To improve the productivity of meta-programmers we need higher-level **semantic strategies** that capture all aspects of some type of transformation, from the composition of tools to the traversals used in the actual transformation. Thus, one would expect reusable strategies for compilation, code generation, and aspect weaving, for example. For example, currently, the production of a code generator involves finding components for parsing, pre-processing, and pretty-printing and writing the actual generation strategy and rules. Often the code for such a generator will look very much like other generators, i.e. follow the same design pattern. A reusable code generation strategy should provide a standard composition and a pluggable traversal strategy that only needs to be instantiated with the name of the input and output languages and the generation rules. An ontology for such semantic strategies can then provide access to the components of the infrastructure starting at the right level of abstraction.

It is not to be expected that such reusable strategies will be right for every possible situation. Therefore, these strategies should be **open and adaptable**. In the first place the strategies should be adaptable to the specific *languages* being transformed. In the second place strategies

should be adaptable to the specific *program* that is being transformed or generated. Regarding the first point, languages differ in large and small ways from the standard model. A reusable strategy based on some general model of data-flow, say, needs to be adapted to the specific data-flow rules of the language under consideration. Regarding the second point: It is not uncommon that the programs produced by generators, especially those that generate high-level programs, are further edited by application programmers in order to fit the generated code in their application. Generators that produce code templates even assume this mode of work. Of course, this produces a maintenance nightmare; when the source of generation changes, all modified files need to be merged with the newly generated ones, undoing all benefits of generation. Thus, it should be possible to adapt the product of generation without physically modifying it, either by means of input to the generator, by generating code that can be adapted (e.g. through subclassing), or by applying further transformations to the generated code (e.g. using aspects).