# Invasive Composition By Transformation Systems

Thomas Cleenewerck

June 30, 2004

## 1 Introduction

The most important strategy to deal with complex systems in computer science is the divide and conquer design paradigm. It works by recursively breaking down a problem into sub-problems until they become simple enough to be solved directly. The solutions to the sub-problems are then composed to give a solution for the whole problem. There are two kinds of composition: non-invasive and invasive composition. The non-invasive composition mechanisms are applicable as long as the kind of components to be composed fit in the dominate decomposition. However it has become clear that there are multiple equally valid and useful decompositions of the same software. In order words, there are often components that do fit and violate the dominate decomposition. There are two ways of dealing with this problem. One approach is to express a software system as a set of multi-dimensional concerns like HyperSpace [OT00]. Another approach is to keep a single dominate decomposition and express the components that violate this decomposition in a crosscutting way like AspectJ. In this later approach such crosscutting components must be invasively composed with the other components.

Quite a lot of the generative programming techniques have been build with the second approach in mind and thus offer various invasive composition mechanisms. Let us briefly discuss the most significant ones. The founder of invasive composition is subject-oriented programming [HO93]. In this model object-oriented code snippets and fragments are composed with one another using correspondence and combination rules. Gray box component models integrate [TG97] through a partial exposure of the internals of the system in terms of an operational model [BW97]. Glass-box composition models use declarative specifications to compose and reason about the composition of components [Bat03]. More recently, aspect-oriented programming (HyperJ and AspectJ) broadened the application of a crosscutting concern to a set of crosscutting points scattered over the entire software system where existing code gets composed with the crosscutting code. In fact, *every concern-specific language* ranging from general purpose languages like the ones discussed above to domain-specific languages enabling the specification of their problem into a more appropriate concern needs invasive composition mechanisms to compose these concerns. Note that the invasive composition mechanisms must not always be visible to the developer. In the case of concern-specific languages, the more domain specific the less visible the invasive composition mechanisms will be. In short, invasive compositions are frequently needed and encountered.

## 2 Position

Invasive composition mechanisms are unfortunately enough still implemented with ad-hoc generators. Hereby losing valuable research results of the three main-stream general purpose transformation paradigms (GPTP): template or rule-based transformations and attribute grammars. *Our position statement is that the reason for the use of ad-hoc generators lies in the fundamental underlying in-place substitution property of those general purpose transformation.*

Template, rule-based transformations and attribute grammars are all based on an *in-place substitution* mechanism: Templates are parameterized target language expressions with escaping variables referring to any kind of domain information necessary. The templates are composed (usually by concatenation) with one another to form the whole solution. In rule based systems, the target language expression produced by rules are substituted with their top-level or pivot nodes. When no more rules apply the complete target language expression is reached. In attribute grammars attributes are attached to their productions and are afterwards also composed into a complete solution. Clearly each transformation module (template, rule or attribute) produces a target language expression which is composed with the others to form a complete solution.

Merely using in-place substitutions to implement an invasive composition mechanism is very cumbersome and troublesome. Invasive composition mechanisms need to exert influence on various parts of other components in the system. Since only in-place substitutions are supported, developers are often tempted and forced to come up with creative work arounds for two problems (1) escaping from their local context to the other parts of the system and (2) implementing their effect in those parts of the system. When these two problems are dealt with naively the escaping and the implementation of their effect highly depends on the implementation details of the rest of the system and its components and thus on the state of the transformation process. Very soon these dependencies clutter up the system and result in a spaghetti code implementation. To keep this more or less manageable a staged transformation process is the most commonly used solution where the escaping and the implementation of their effect is performed in two separate stages. However, this does not reduce the number of dependencies.

Clearly transformation systems are not very suitable to implement invasive composition mechanisms. In order to remedy this situation, we believe that it is necessary to *extend* current transformation systems with a *suite of basic invasive capabilities.* These capabilities should not only facilitate the implementation but render it also more robust to evolutions of the other components of the software system.

Currently we are experimenting with a suite of basic invasive capabilities based on the model presented by subject-oriented programming (SOP) [OKK+96, SCT99]. The extensions for the transformation systems we propose are thus based on SOP. SOP was formulated and founded in terms of object-oriented programming and introduced two kinds of rules: correspondence rules and combination rules. The correspondence rules declare which parts of the components must be combined with one another, the combination is performed by the combination rules. The two rules are externally defined to the components. To apply the SOP ideas in a general setting in transformation systems, a couple of modifications had to be made: (1) generalization and integration of those two rules into the transformation paradigm (2) additional context specifications expressed in declarative source language constructs using paths, (3) automation of tedious context specifications, (4) selection of the most specific and applicable rules and (5) increase of the robustness of the combination rules.

The above extensions are build on top of the Linglet Transformation System (formerly known as Keyword Based Programming [Cle03]). Since our incentive for this research lies in the need for more domain-specific concern-specific languages, the invasive composition mechanisms of current experiments are usually implicit constructs in those languages. Further experiments with other generative techniques are necessary to refine and validate our approach.

## References

[Bat03] Steve Battle. Boxes: black, white, grey and glass box views of webservices. Technical Report HPL-2003-30, HP, 2003.

[BW97] M. Buchi and W. Weck. A plea for grey-box components, 1997.

[Cle03] Thomas Cleenewerck. Component-based dsl development. In *Proceedings of GPCE03 Conference, Lecture Notes in Computer Science 2830*, pages 245–264. Springer-Verlag, 2003.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428. ACM Press, 1993.

[OKK+96] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Theor. Pract. Object Syst.*, 2(3):179–202, 1996.

[OT00] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[SCT99] Harold Ossher Siobhán Clarke, William Harrison and Peri Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices*, 34(10):325–339, 1999.

[TG97] D. Tombros and A. Geppert. Managing heterogeneity in commercially available workflow management systems: A critical evaluation, 1997.