Comprehension of Generative Techniques*

Victor Winter†, Christopher Scalzo, Arpit Jain, Brent Kucera, and Azamatbek Mametjanov
Department of Computer Science, University of Nebraska at Omaha

A wide variety of tools, techniques, and development methodologies are presently being employed to facilitate the construction of complex software in a timely and cost-effective fashion. In this context, researchers have demonstrated that a number of design and development goals can be satisfied in a framework in which software artifacts are manipulated through generative techniques based on rewriting. This has lead to the creation of tools and environments in which software artifacts can be subjected to sophisticated manipulations. These tools are referred to as *software/program transformation systems*.

In spite of their demonstrated utility, the software industry has been slow in adopting software transformation systems into their development tool set. One reason for this is that the computational model underlying software transformation systems is very different from computational models underlying mainstream programming paradigms (e.g., objected oriented programming). As a result, programmers are faced with a considerable learning curve when adding a software transformation system into their development tool set.

Benefiting from theoretical advances in recent years, software transformation systems have become considerably more sophisticated than their predecessors. This has significantly increased the problem space to which transformation can be effectively applied. Unfortunately, in spite of their theoretical sophistication, software transformation systems generally provide very little support facilitating the comprehension of transformation-based programs. We believe that this presents a bottleneck – both to the experienced transformational programmer as well as to the novice. A particular negative consequence of this is that, as the sophistication of software transformation systems increases, they become more and more inaccessible to mainstream programmers.

To address this concern, we are exploring the use of *tracing* to facilitate the comprehension of software transformation. In this context, transformation is modelled as a sequence of high-level computational steps which we refer to as a *trace*. We use the term *trace element* to refer to a computational step within a trace. Broadly speaking, we are interested in the relationships that exist between (1) a transformation source program p, (2) a software artifact being transformed t, and (3) the trace $\mathcal{X}$ resulting from the application of p to t. For example, let $r : lhs \rightarrow rhs$ denote a basic rewrite rule belonging to the program p, let t_1 denote a sub-term in t, and let r' denote the trace element in $\mathcal{X}$ corresponding to the instance of r that rewrites t_1 to t'_1. Among other things, we are interested in viewing the relationships between (1) r and p, (2) t_1 and t, (3) r' and $\mathcal{X}$, (4) r' and t_1, and (5) r' and t'_1.

Display and navigation are two major issues that arise in our tracing framework. Trace elements may contain complex values (e.g., terms) and, depending on the sophistication of the transformation system, may have a nontrivial relationship to the transformation source program. The software artifact being transformed may have a complex structure defined by an abstract and/or concrete syntax. We are employing graphical as well as symbolic techniques to display these values and structures.

*This work was partially supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

†vwinter@mail.unomaha.edu

Traces and software artifacts will typically have a scale and complexity making brute-force navigation (e.g., manual scrolling) impractical. Furthermore, only a portion of a trace may be the current focus of attention. Towards this end, we are developing a framework where filters can be used to extract, from a trace, a subsequence of trace elements satisfying particular properties (e.g., all rewrite rules of a particular type, all transformations affecting a particular piece of a software artifact, etc.).

The tracing framework we are developing draws upon related research (e.g., debuggers) that has been developed for other computational paradigms (e.g., Prolog, Java, Lisp, etc.). However, the computational paradigm underlying a given software transformation system will generally have distinct differences from the aforementioned paradigms.

We are applying the ideas discussed in the previous paragraphs to develop a tracing facility for the HATS software transformation system. HATS [2] is a general purpose transformation system in which the manipulation of software artifacts is expressed in a special purpose language called TL [3, 1]. As such, it belongs to the class of meta-tools for generative programming. An effort is currently underway to design and implement a sophisticated tracing capability for TL. Our goal is to provide users with an abstract view of the computational model underlying TL and to use this model to relate the dynamic behavior of a TL program (i.e., its execution) to its static description (i.e., source code).

The abstract computational model we are developing focuses on the high-level computational steps underlying TL transformations. The language TL provides a computational framework, where transformational ideas are expressed in terms of conditional rewrite rules, whose application is controlled by a variety of standard strategic combinators and traversals (e.g., sequential composition, left/right biased choice, top-down and bottom-up traversals, user-defined traversals, etc.). Distinguishing features of TL include: (1) the ability to express transformational ideas in terms of higher-order conditional rewrite rules, (2) the ability to control rule application through a variety of unique combinators including the *transient* combinator and the *hide* combinator, and (3) a semantic foundation, where the failure of rule application behaves like the identity transformation (in contrast to strategic systems in which rule failure yields the strategic constant FAIL).

References

[1] V. Winter. Strategy Construction in the Higher-Order Framework of TL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 124, 2004.

[2] V. Winter and J. Beranek. Program Transformation Using HATS 1.84. In *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, 2006.

[3] V. Winter and M. Subramaniam. Dynamic Strategies, Transient Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.