The TAMPR Program Transformation System:
Design and Applications *

James M. Boyle
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, U.S.A.
boyle@mcs.anl.gov

Terence J. Harmer
The Queen's University of Belfast
Department of Computer Science
Belfast, BT7 1NN
Northern Ireland
t.harmer@qub.ac.uk

Victor L. Winter
Sandia National Laboratories
Intelligent Systems and Robotics Center
Dept 9622, MS-0660
P.O. Box 5800
Albuquerque, NM 87185-0535
vlwinte@sandia.gov

January 30, 1996

Abstract

TAMPR is a fully automatic, rewrite-rule based program transformation system. From its initial implementation in

*This work was supported in part by the Mathematical Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and in part by the BM/C3 directorate, Ballistic Missile Defense Organization, U.S. Department of Defense
1970 [1], TAMP has evolved into a powerful tool for generating correct and efficient programs from specifications [4, 2].

Any program transformation system must have a notation for expressing transformations. In order to achieve automatic operation, this notation must be extended to include the capability to specify where, when, how often, and in what order to apply transformations. Such specifications have the potential to become both impossibly complex and immensely fragile, given that the program being transformed is constantly changing as the result of applying transformations. The particular genius of TAMP lies in avoiding the need to specify such details, by letting the program being transformed control the order and location of applying the transformations. Thus, TAMP provides only a minimalist transformation language, distinguished by

- A declarative approach to specifying transformations;
- A restricted repertoire of constructs for analyzing and transforming programs;
- Application of transformations to exhaustion;
- An emphasis on sequences of canonical forms for carrying out large-scale refinements;
- Completely automatic operation; and
- The ability to effortlessly “replay” the application of transformations when either the program being transformed or the transformations themselves change.

We first describe some of the applications of the TAMP system, to suggest the power and practicality of its approach. These applications include the derivation of the programs in the LINPACK package of Fortran subroutines for solving systems of linear equations [7] (in daily use throughout the world for almost 20 years) and the derivation of efficient programs from higher-order functional specifications. Examples of the latter include the derivation of efficient sequential and parallel implementations of the TAMP system itself [2] and the derivation of a program for solving hyperbolic partial differential equations that exceeds the per-
formance of handwritten Fortran code on a Cray vector processor [5].

The main emphasis of this presentation is on the TAMP approach to program transformation, which traces its ancestry from Chomsky’s transformational grammars [6]. This approach is embodied in TAMP’s high-level language for expressing transformations, and we discuss both its design and its philosophical underpinnings. The minimalism of the TAMP language leads to the formulation of transformations that are clear and, for the most part, easy to verify. Transformations are expressed as rewrite rules; no procedural code is used. Simple rewrite rules are extended with subtransformations, which, while again rewrite rules, have restricted scope of application. Subtransformations facilitate expressing such things as substitutions. Simple rewrite rules are also extended with applicability conditions, which provide a facility for exercising fine control over the application of transformations. Applicability conditions are again expressed in terms of rewrite rules, thereby maintaining the conceptual simplicity of the TAMP facilities.

TAMP normally applies sets of transformations to exhaustion—until every program fragment that they match, even ones constructed by prior applications, has been transformed. A set of transformations thus defines a canonical form. The use of canonical forms enables TAMP to apply transformations automatically, because application of a set of transformations terminates when the canonical form it defines is reached. The emphasis on canonical forms leads naturally to the principle of structuring large-scale derivations as a sequence of canonical forms through which a specification is passed in order to construct an efficient executable program [3]. The use of a sequence of canonical forms also provides transformational derivations with a clear structure and facilitates reuse of sets of transformations.

Of course, automatically applied transformations are of little use unless they preserve desired properties of the programs being transformed—usually, their correctness. The
semantic foundation for TAMPR is a denotational semantics for the subject language in which the programs being transformed are expressed. This denotational semantics is the basis for proofs that TAMPR transformations preserve the correctness of (refine) the programs to which they apply.

We conclude with a brief description of the engineering considerations involved in the implementation of TAMPR, including using the system to construct its own implementations.

References


Reprinted (with corrections) in R. D. Luce, R. Bush, E. Galanter eds., 
*Readings in Mathematical Psychology, Vol. II*, John Wiley and Sons, 

and G. W. Stewart, SIAM (Society of Industrial and Applied Mathematics), 1979, Philadelphia, PA.