Proceedings of the Fourth Systems Reengineering Technology Workshop, NSWC. John Hopkins University APL Research Center Report RMI-94-003, Monterey CA, February, 1994.

Knowledge-Based, Metalanguage-Based Object Abstraction for Automatic Program Transformation

Romel Rivera
Member, IEEE

Xinotech Research, Inc.
1313 Fifth Street Southeast. Suite 213
Minneapolis, MN 55414
romel.rivera@xinotech.com

Abstract

This paper describes Xinotech Research's knowledge based, metalanguage-based programming environment to support automatic program transformation and object abstraction for forward and reverse engineering. In this environment. both knowledge extraction and knowledge abstraction are metalanguage-based and thus language independent. The transformation engine is accessible through the interactive syntax-directed tools for program construction or for massive reengineering. This transformation infrastructure is operational for Ada, and can be applied to transform existing programs to support object-oriented methodologies, to port existing software to new libraries and platforms, to translate automatically between languages. to change the meaning of programs, or to enforce the semantics of applications or programming standards. It also supports specification and prototyping languages, and can be retargeted to other programming languages.

1. Introduction

"By now it is hard to imagine that any computer professional has not become aware of the bottleneck in software development. For both commercial and government applications, the annual bill for software is rising at a rapid pace. For example, the U.S. Department of Defense (DoD) spent over $3 billion on software in 1980 and their expenses are projected to grow to $30 billion per year by 1990 (DoD Annual report FY '81). Moreover, these costs are only the tip of the iceberg, as the impact of faulty software, delayed software, and continuing maintenance Costs drive the real costs even higher" [12].

Ten years later, K.A. Banniuck confirms the above prediction. His study estimates that software expenditures in 1990 were over $185 billion worldwide with approximately $90 billion being spent in the U.S., and of that, $27 billion spent by the DoD. [2]

"We might well ask, why this phenomenal growth in the cost of software? There are several major reasons. One is the fact that the requirements for new software systems are more complex than ever before. A second reason for the rising cost of software is the increased demand for qualified software professionals. A third reason, is the fact that our software development tools and methodologies have not continued to dramatically improve our ability to develop software.

"...It has for a long time been recognized that one fundamental weakness of software creation is the fact that an entirely new software system is usually constructed 'from scratch'. This is clearly an unfortunate situation, as studies have shown that much of the code of one system is virtually identical to previously written code. For example, a study done at the Missile Systems Division of Raytheon Company observed that 40-60 percent of actual program code was repeated in more than one application [11]. Therefore the idea of reusability would seem to hold one answer to increasing software productivity. And yet the simple notion of reusability (i.e., code reusability) has been considered by computer professionals over the years but has never been entirely successful." [12]

Methodologies for reusability must be seamlessly integrated into the design, coding, testing and maintenance phases of the software cycle, which according to E. Horowitz [12], account for 87% of the total life-cycle effort.

Any methodology that can be proposed, could fail to be implemented if it is not supported by tools that synthesize software understanding and automate the transformation of programs so that reusability and modernization techniques can be applied automatically and on a large scale. The task of manual application of evolutionary transformations on large software systems would be found overwhelming and quickly discarded. The problem is that in order to create these tools, they need to be supported by an environment infrastructure with the following properties:

1. Structural and semantic knowledge of the programming language (e.g. Ada).
2. Reusable language knowledge that:
   i. Supports quick fabrication of a multitude of language tools outside the realm of language translation, and
   ii. Allows customization of the created tools.
3. Formal Specifications. The production of language tools requires that the realization of structural and semantic language rules be available to the tool designer so that they can be applied to implement particular transformations, measurements and analyses, infer or complement other rules, etc. For example, a successful environment must provide the ability to specify new applications such as transformations to modularize source code, thus complementing the typical, hard-coded, predefined functions of object code generation. Language knowledge reusability is best supported by a system where languages can be defined with formal specifications, independent and separate from the application tools.
4. If forward and reverse engineering tasks are to be unified, it is essential that certain tools be interactive. This means that the formal mechanisms to manipulate language structures must be incremental. Traditionally, incrementality has been supported through domain-dependent, algorithmic approaches. For the sake of generality, it is desirable that incrementality be derived from the semantics of the metalanguage. From these requirements, it is clear why language-based tools to automate the otherwise impossible task of manual transformation of source code have not proliferated and matured.

2. The Xinotech Environment

The environment is designed to be language-independent. Knowledge extraction (e.g. parsing, creating abstract syntax trees, and deriving semantic attributes) is expressed using a formal notation called XinoML, *the Xinotech Meta-Language*. Knowledge abstraction (the process of recognizing program patterns and transforming them into higher-level structures) is expressed through an XinoML component called XPAL, *the Xinotech Pattern Abstraction Language*. The system can thus be re-targeted for other languages and applications at a fraction of the original cost.

XPAL is designed to express complex program patterns and to specify transformations of these patterns into more cohesive higher-level concepts. The alternative approach of using an intermediate "universal" language to which programs are first transformed, causes the unnecessary loss of the original model and still does not provide the means to tailor transformations.

Only with a pattern language does the task of specifying a vast evolving library of patterns and their transformations become feasible, allowing pattern specification to become an application-oriented task.

XPAL makes use of a complete semantic notation and a comprehensive semantic library. Because XPAL is a component of XinoML, the extraction meta-language, XPAL has access to XSSL (XinoML's semantic notation) as well as all of the semantic equations written to properly define a particular programming language. For example, writing patterns that require the use of language scoping rules can be done by simply referring to the corresponding semantic equations.

XPAL transformations can also be used as the vehicle to formalize and document the implicit relations needed to abstract object oriented (00) models from non-00 programs.

The environment is designed to support interactive software development, including syntax-directed construction, graphical abstraction, and standards and guidelines detection and enforcement. All these tools are built on top of the metalanguage engine. Pattern transformations are available interactively through these tools' user interfaces. Transformation libraries have been developed to support object orientation, conversion to Ada 9X from Ada 83, and translation to Ada from CMS-2 and Jovial.

3. XinoML, the Xinotech Meta–Language

The language-based, language-independent infrastructure of the Xinotech environment is provided by the implementation of XinoML. XinoML is a highly-readable language for specifying the abstract grammar, external syntax (views) and semantics of languages. XinoML is an *environment metalanguage,* because it supports the design, implementation, embedding, revision and evolution of the various languages used in a software development environment, such as specification, documentation, design, programming, testing, and configuration languages. XinoML provides support for quick language prototyping, reusable language descriptions through module decomposition and inheritance, inter- and intra-language transformations, and separation of embedded and annotation languages. It provides an open architecture for integration to other traditional semantic analysis tools such as STARS ASIS for Ada.

XinoML supports *modules* for the hierarchical decomposition of languages. Modules are collections of related symbols. Modularization allows the language designer to logically divide the specification to enhance its readability. A language specification can import modules from other XinoML specifications. This encourages reusability when prototyping new languages.

A *construct* is defined in terms of its intrinsic language properties, such as abstract grammar, views (unparsing syntax) and semantics. Other clauses describe details for the environment, such as menus, placeholders, etc.

XSSL, *the Xinotech Semantic Specification Language*, is a component of XinoML. XSSL is a general notation: it supports, e.g., the expression of Ada scoping rules, type checking, data flow relations, and language translation. XSSL supports structured types and generalized lists, and it incorporates efficient abbreviation schemes to reduce the complexity of expressions due to explicit semantic flow. It uses object-oriented encapsulation to achieve the reuse of semantic structures throughout multiple constructs. XSSL supports incremental evaluation as well as the semantics of inter-compilation-unit relations.

4. XPAL and Pattern Abstraction

Pattern or *plan* abstraction is the transformation process of automatically condensing or abstracting low-level source code patterns found in existing software into high-level program concepts. XPAL. *the Xinotech Pattern Abstraction Language*, a declarative, constraint language, is the vehicle to express these program patterns and their transformations. Since XPAL is a component of XinoML, the Xinotech Meta-Language, these transformations can be written for any language specified with XinoML. Therefore, the entire mechanism is language independent.

Pattern abstraction is valuable because it recognizes implied or concealed relationships in low-level source code and, by representing them with existing higher-level structures, makes the relationships explicit and conceptual, and the code more cohesive and less fragmented. This reduces the complexity of the representation while increasing the expressive power of the resulting programs, thus enhancing its maintainability, understandability and reusability. This process is the inverse of top-down synthesis, such as program compilation.

In XPAL, patterns can be specified in terms of other patterns. Because XSSL, a component of XinoML, is a general notation for expressing the semantics of languages, patterns can use or complement these semantic equations. The approach traditionally taken in designing reengineering environments is that of providing some semantic capabilities through a limited set of hard-coded functions. In the XinoML family, graph operations, such as transitive closures for data and control flow, can be specified on the relationships characterized by XSSL equations. A language this comprehensive makes pattern abstraction very powerful.

Advantages of having XPAL as a component of XinoML. Because the XPAL notation is embedded within XinoML, it has the advantages of full access to the abstract grammar and semantics of the programming language, access to a general semantic notation, the use of XinoML extraction mechanisms, such as parsing views, to express tree patterns textually, and the use of multiple views which allow syntactical transformations to be expressed in the syntax of the programming language.

5. The Xinotech Program Composer

The Composer is the central application tool built on top of the XinoML language infrastructure. It is a syntax-driven, interactive semantic tool for the design and construction of programs. Programs are managed as abstract syntax trees (AST), with multiple textual representations or views. An incremental parser and an incremental unparser provide the mappings between the textual and the AST representations. One of the main areas of concern during the design of the Composer was the functionality and behavior of its incremental bottom-up parser. This parser was designed to support a smooth left-to-right insertion while providing full interactive language support such as automatic template generation, placeholders, menus, and formatting while typing. The user can select levels of template generation during insertion. Templates are non-intrusive, since the user can type over to skip optional clauses. Text files not created with the Composer are automatically imported the first time they are opened.

Views can be used to create multiple formatting schemes, or to combine or isolate programs with embedded documentation and/or PDL structures. The Composer supports browsing through libraries, and provides program outlines from any point in the program.

6. The Graph Abstractor

The Graph Abstractor is an analysis and maintenance tool designed to display XSSL-generated semantic relations. These relations can be displayed graphically or structurally. The Graph Abstractor is designed to minimize the size complexity of graphs and isolate the relations of current relevance.

7. The Guideliner

The Xinotech Guideliner is an interactive program analyzer. It verifies adherence to programming guidelines, standards and metrics, and transforms programs automatically to comply with these guidelines. These guidelines are written using XPAL. The design goals of the Guideliner were as follows:

1. To provide an integrated, incremental capability to prevent and/or detect and flag user-defined guideline deviations during interactive program construction with the Composer.
2. To provide batch processing to obtain detailed and statistical reports regarding non-compliance with user-defined guidelines and standards. This can be useful during the quality assurance phase of code acceptance from contractors.
3. To provide the automatic translation of source code to comply with user-defined guidelines and standards. This process can be applied to any source code, regardless of whether it was created with the Composer.
4. To provide a wide range of metrics measurements that can be requested by the user as part of the guidelines and standards to be analyzed.

8. Reengineering Applications

XPAL is a general language for program recognition and transformation. It can be used to:

1. Translate programs from one language to another, such as CMS-2 or Jovial to Ada.
2. Detect and correct violations of user-defined guidelines and standards.
3. Transform existing non-00 programs into object-oriented programs.
4. Port existing programs from one supporting library to another. This helps automate migration to newer standard libraries, or to different operating systems and hardware platforms. As new libraries are created, existing applications can be searched for potential matches, so that the application can be modernized and expressed in terms of the new reusable components.
5. Modify the meaning of programs. Transformations can be written to modify existing programs so that they perform new functions, thus helping create new applications from existing ones.
6. Apply isolated transformations interactively. XPAL libraries can be created to generate bodies out of package specifications, to split packages or procedures, improve module decomposition, etc.

8.1 Language Translation

Typically, language conversion is an abstraction process, very much the opposite of top-down synthesis or compilation. This is the case whenever the target language is a higher-level language, as in the case of translating CMS-2, Jovial or FORTRAN to Ada. Compilation technologies do not lend themselves well to this process, and pattern abstraction is highly desirable so that low-level, implicit, global relationships can be identified and abstracted into explicit higher-level constructs. XPAL was designed to support such abstraction. These are some examples of XPAL applications when converting CMS-2 to Ada:

1. Patterns can be defined to map different operating-system dependent multi-tasking models in CMS-2 to the construct-based tasking model in Ada. These transformations can be done very effectively since they are a classic example of implied relationships made explicit by the abstractor. Patterns can be written for the following:
   i. Building the multi-thread task structures out of CMS-2 modules and entry point tables.
   ii. Building the "Message_Center" task out of the specification of the message broadcasting table for the linker.
   iii. Abstracting concurrent critical regions by localizing and encapsulating the shared data into tasks, from the fragmented test-and-set protected access semaphores found in CMS-2. Such abstraction supports code migration towards an object-oriented methodology.
   iv. Customizing patterns to support the direct translation of CMS-2 library procedures for some of these functions (e.g. critical regions), if they exist.
2. Abstracting block structure such *as for, while* and exit-based closed loops from goto-based control flow.
3. Creating procedures to modularize code or to eliminate unstructured loops, and creating enumeration types from sets of constants and related variables.

These are some of the advantages of XPAL-based translation:

1. *Fully Customizable.* This is a requirement for the case of CMS-2 or Jovial to Ada, since the translation will depend on the dialect, the executive in use, and library and other environment dependencies, as well as on the customization of the translated code to Ada guidelines such as the STARS Ada Reusability Guidelines.
2. *Fully reusable during subsequent system evolution.* Components developed for translation, since they are language-independent, can be used interactively during continuing Ada development (as Ada-to-Ada reengineering tools).
3. *Powerful dual translation and development environments.* Part of the success of the reverse-engineering process (i.e. translation) depends on how well it is integrated with the forward-engineering process (i.e. development). Such integration dictates the success of the translation system for interactive use.

4. *High-quality of the resulting code.* By devising sophisticated schemes for code abstraction, the transiator designer can make more comprehensive use of the features of the target language (e.g. Ada). This results in more condensed and readable code. By not discarding the original implementation through a very-high-level intermediate language, this approach is able to maintain comparable efficiency levels.
5. *Predictability.* The Xinotech approach, using external specifications for the translation, allows the user to verify and approve *in anticipation,* the ways in which source language structures have been chosen to be translated. In a system where the implementation was discarded, the efficiency of the resulting code would be completely unpredictable.
6. *Life Cycle Orientation.* The XPAL-based approach takes into account the fact that the translated system will continue to evolve, so tailored patterns can prepare it for further growth, by supporting 2167A documentation generation and traceability with the PDL of choice, extraction of high-level graphs, and compliance with user-defined standards.
7. *Formally Specified Translation.* Another advantage of using formal specifications is that they provide a highly modular and functional decomposition of the translation system, resulting in an accessible mechanism for verifying the translator's reliability.
8. *Low-risk Development Path.* This is the result of two factors: predictability, and the fact that this technology is implemented progressively, with practical appreciable benefits available from day one. These benefits continue to grow in proportion to the resources invested in the project. Its success can be measured and monitored throughout the development effort.

8.2. Support for Ada 9X Compliance and Ada 9X Philosophy

The Xinotech transformation environment includes a set of Ada 9X transformation libraries to support Ada 9X compliance as well as Ada 9X philosophy. In mm, these libraries are managed by the Guideliner's user interface.

Support for Ada 9X Compliance. The environment provides a library of transformations to automatically translate the 9X violations in existing Ada 83 sources to the Ada 9X standard. These transformations can be applied interactively or in batch mode: the result is compilable Ada 9X code. This library is used to translate to 9X for compliance, even though the resulting code may not be object oriented (00) or otherwise embody Ada 9X philosophy in any way.

Support for Ada 9X Philosophy. An additional library transforms 9X-compliant programs into a model supporting 00 and 9X philosophy. The 00 Ada 9X programs resulting from these transformations take advantage of 9X-specific features for modularization, object-orientation, parallelism and synchronization. Examples:

1. Transforming a package into a hierarchy with children packages. This supports improved modularization by allowing the direct sharing of declarations among a closely-related family of packages.
2. Transforming Ada structures to support explicit Ada 9X vectorization. A few of these cases can be detected automatically. Conversely, the user is able to invoke these transformations interactively.
3. Transforming a synchronization model into one with explicit protected records. In some cases, the old synchronization model can be derived from the usage of a particular library.
4. Transforming record types with variants to tagged types with extensions. This transformation is requested by the user for a particular record type with variants. The particular record type is analyzed to determine if the transformation is possible, and if so, the transformation is performed. This transformation takes advantage of multiple dispatching to enhance the readability, object-orientation, and reusability of the code. The simplest such case involves a record with a single variant whose discriminant is a value of an enumeration type.

8.3 Real-Time Prototyping Environments

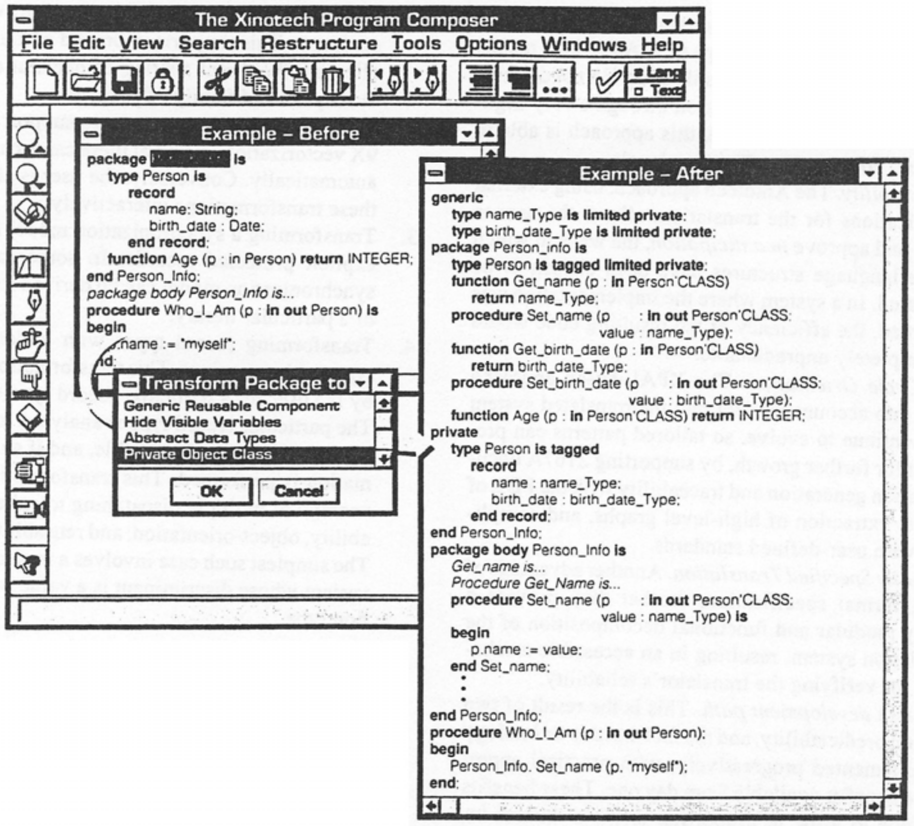
XPAL can be applied to support specification or prototyping languages such as Luqi's PSDL. [19], [22] Besides providing an integrated, interactive front-end for PSDL, XPAL can be used to verify adherence to design methodologies, to synchronize graphical with structured editing, and to map between specification and implementation languages.

8.4 Object Abstraction

Object abstraction is the process of recognizing relationships in existing, non object-oriented (00) Ada programs, and transforming these programs into a higher-level, object-oriented architecture with reusable components.

00 design methodologies have been in use for some time, and are very useful in helping to understand the behavior of systems and relationships between components (objects). It seems natural that obtaining an object-oriented design view of existing non-00 source programs through reverse-engineering will:

1. Help us understand the intended behavior of a system . and its relationships.
2. Allow us to capture this 00 design in an 00 design language that can be manipulated textually or graphically by design tools, thus making it possible to use forward engineering (FE) technology to analyze, modify and browse through the design.
3. Allow us to restructure or redesign the existing code so that it conforms to the recaptured OO design.



```
package [illegible] is
   type Person is
      record
         name: String;
         birth_date : Date;
      end record;
   function Age (p : in Person) return INTEGER;
end Person_Info;
package body Person_Info is...
procedure Who_I_Am (p : in out Person) is
begin
   p.name := "myself";
end;
```

```
generic
   type name_Type is limited private;
   type birth_date_Type is limited private;
package Person_info is
   type Person is tagged limited private;
   function Get_name (p : in Person'CLASS)
      return name_Type;
   procedure Set_name (p       : in out Person'CLASS;
                       value : name_Type);
   function Get_birth_date (p : in Person'CLASS)
      return birth_date_Type;
   procedure Set_birth_date (p       : in out Person'CLASS;
                             value : birth_date_Type);
   function Age (p : in Person'CLASS) return INTEGER;
private
   type Person is tagged
      record
         name : name_Type;
         birth_date : birth_date_Type;
      end record;
end Person_Info;
package body Person_Info is
   Get_name is...
   Procedure Get_Name is...
   procedure Set_name (p       : in out Person'CLASS;
                       value : name_Type) is
   begin
      p.name := value;
   end Set_name;
    •
    •
    •
end Person_Info;
procedure Who_I_Am (p : in out Person);
begin
   Person_Info. Set_name (p. "myself");
end;
```

Fig. 1. The package in the left window is automatically transformed into a private object class. Transformation rules (not shown) are specified in XPAL, the Xinotech Pattern Abstraction Language.

Examples of XPAL for Object Abstraction.

1. Transforming exported data objects into abstract data types. Data objects will be hidden, and made available only through access methods (procedures). This includes the automatic creation of initialization and finalization methods for the data types.
2. Transforming program units into reusable blueprints (e.g. generic units in Ada).
3. Transforming sets of variables into object classes by hiding them in structured types with access methods.
4. Transforming variant record types into a base class with subclasses (e.g. Ada 9X tagged types with extensions). These transformations will take advantage of multiple dispatching to enhance readability, object-orientation and reusability.

9. Benefits

9.1 Benefits for Ada 9X

This environment represents a rather extensive solution for Ada reengineering, because it automates the evolutionary migration, from the legacy systems written in the proprietary languages of the sixties, towards the full, object-based, design philosophy of Ada 9X. For example, it can be used to:

1. Translate CMS-2 or Jovial programs into Ada.
2. Translate Ada 83 programs into Ada 9X.
3. Support the object-orientation of existing Ada code, according to the philosophy of the new Ada 9X features, thus enhancing reusability.
4. Automate the porting of existing Ada applications to new Ada 9X standard libraries, thus enhancing the inter-changeability of the application components.
5. Automate transformations to change the meaning of existing programs, thus supporting the adaptation of existing programs to new applications.

9.2 General Benefits

Support for All Languages in the Life Cycle. Pattern abstraction can be applied to all the languages in the software life cycle, from specification languages, to 00 design languages, to annotation languages, programming languages, etc. XPAL may be used to automate top-down translation during program development, or to abstract design and specifications during reverse engineering.

Interactive Transformation Environment. Transformations can also be applied interactively during program construction. Forward and reverse engineering are thus integrated in a single homogeneous environment.

Support for Multiple Programming Languages. Through XinoML, the same homogeneous language-based environment is available for many programming languages. This is particularly attractive for translation between dialects. The Xinotech environment can also be instantiated (very cost effectively), for specialized languages, such as VHDL and database languages.

Open Architectures. The existing Xinotech environment supports the client-server model of an open heterogeneous architecture with a graphical user interface.

An Integrated Environment. Xinotech's approach was to create an integrated semantic environment for syntax-directed program construction, as well as for analysis and transformation. Forward and reverse engineering are indistinguishable. Vast transformation libraries can be expressed and customized with a metalanguage for pattern abstraction.

10. Bibliography

[1] Ada 9X Project. Ada 9X Requirements. Office of the Under Secretary of Defense for Acquisition, Washington. D.C., December 1990.

[2] K..A. Bannick. Breakdown of Software Expenditures in the Department of Defense, United States and in the World. Master's Thesis, Naval Postgraduate School, Monterey, CA, Sept. 1991.

[3] B. Barding, C. Thompson. Composable Ada Software Components and the Re-Export Paradigm —Parts 1 and 2. ACM SIGAda Letters VIII (1); pp. 58-79, 1988.

[4] Boyle, J.M., Muralidaran, M.N. Program Reusability Through Program Transformation. IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984

[5] C.L. Braun,J.B. Goodenough, R.S. Eanes. Ada Reusability Guidelines. Technical Report 3285-2-208/2, SofTech, Inc., Waltham, Massachusetts, Revised 1991.

[6] P.T. Breuer and K. Lano. Creating Specifications from Code: Reverse-engineering Techniques. Journal of Software Maintenance: Research and Practice, John Wiley and Sons. 1991. Reprinted in Software Reengineering, by Robert S. Arnold. IEEE 1993.

[7] Gianluigi Caldiera, Victor Basili. Identifying and Qualifying Reusable Software Components. IEEE Computer, Feb 1991. Reprinted in Software Reengineering, by Robert S. Arnold, IEEE 1993.

[8] G. Canfora, A. Cimitile, and U. de Carlini. A Logic-Based Approach to Reverse Engineering Tools Production. IEEE Trans. on Software Eng., Vol. 18, No. 12, December 1992.

[9] A. Cimitile and U. de Carlini. Reverse engineering: Algorithms for Program Graph Production. Software Practice and Experience, Vol 21. pp 519-537, 1991.

[10] W. Cunningham, K. Beck. Constructing Abstractions for Object-Oriented Applications, Journal of Object-Oriented Programming, 2,2,17-19, August 1989.

[11] W.L. Frank. What limits to software gains ? Computerworld, pp 65-70, May 4, 1981.

[12] E. Horowitz, J.B.Munson. An Expansive View of Reusable Software. Software Reusability, Vol. I, Edited by T.J. Biggerstaffand A.J. Perlis. ACM Press, 1989.

[13] S. Horwitz and T. Teiteibaum. Generating Editing Environments Based on Relations and Attributes. ACM Trans. on Programming Languages and Systems, Vol 8, No 4, Oct 1986.

[14] Ivar Jacobson, Fredrik Lindstrom. Re-engineering of old systems to an object-oriented architecture. Proc. OOPSLA, 1991. Also reprinted in Software Reengineering, by Robert S. Arnold, IEEE 1993.

[15] Gail E. Kaiser, Simon Kaplan. Parallel and Distributed Incremental Attribute Algorithms for Multiuser Software Development Environments. ACM Transactions on Software Engineering Methodology, January 1993, Volume 2, Number 1.

[16] K. Koskimies, 0. Nurmi, J. Paaki. The Design of a Language Processor Generator. Software -Practice and Experience, Vol. 18 (2), Feb. 1988.

[17] Richard D. Linger. Software Maintenance as an Engineering Discipline. Proc. Conf. on Software Maintenance, pp 292-297. Reprinted in Software Reengineering, by Robert Arnold, IEEE 1993.

[18] S.S. Liu, and K..R. Johmann. A Tool Specification Language for Software Maintenance: Part 1 — Language Design, Part II —Usage. SERC Technical Report 36F, CSci Dept., University of Florida at Gainsville, November 1989.

[19] Luqi, V. Berzins, R. Yeh. A Prototyping Language for Real-Time Software. IEEE Trans. Soft. Eng., vol. 14, October 1988.

[20] D. Maier,and D.S. Warren. "Computing with logic". The Benjamin/Cummings Publishing Co. Menio Park, CA, 1988.

[21] B. Meyer. Software Reusability: The Case for Object-Oriented Design. IEEE Software 4(2), 50-64, 1987.

[22] F. Naveda. Specifying a Prototyping Language in the Cornel] Synthesizer and the Xinotech Program Composer for an Integrated Programming Environment. Proceedings 2nd IEEE International Conference on Systems Integration, IEEE, June 15-18, 1992.

[23] D. Pamas, P. Clements, D. Weiss. Enhancing reusability with information hiding. In Proc. Workshop Reusability in Programming, Sept. 1983, pp 240-247.

[24] William W. Pugh Jr. Incremental Computation and the Incremental Evaluation of Functional Programs. Ph.D. Dissertation, Comell University, 1988.

[25] T. W. Reps, T. Teitelbaum, A. Demers. Incremental Context-Dependent Analysis for Language-Based Editors. ACM Transactions on Programming Languages and Systems, Vol. 5, No 3, July 1983.

[26] D.S. Rosenblum. A Methodology for the Design of Ada Transformation Tools in a DIANA Environment. IEEE Software 2(2):24-33, March ,1985. Also as Stanford CSL Technical Report 85-269, February, 1985.

[27] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. Proc. Int'l Conf. on Software Engineering, IEEE 1991. Also reprinted in Software Reengineering, by Robert S. Arnold, IEEE 1993.

[28] I. Silva-Lepe. Abstracting graphed-based specifications of object-oriented programs. Tech. Report NU-CCS-92-4, College of Computer Science, Northeastern University, March 1992.

[29] A.I. Wasserman. P.A. Pircher, R.J. Muller. The Object-Oriented Structured Design Notation for Software Design Representation, IEEE Computer, March 1990.

[30] Waters, R.C. Program Translation via Abstraction and Reimplementation. IEEE Trans. on Software Eng., August 1988