Reusing Software Developments

Allen Goldberg *
Kestrel Institute
3260 Hillview Ave.
Palo Alto, CA 94304

Abstract

Software development environments of the future will be characterized by extensive *reuse* of previous work. This paper addresses the issue of *reusability* in the context in which design is achieved by the transformational development of formal specifications into efficient implementations. It explores how an implementation of a modified specification can be realized by *replaying* the transformational derivation of the original and modifying it as required by changes made to the specification. Our approach is to structure derivations using the notion of tactics, and *record derivation histories as an execution trace of the application of tactics*. One key idea is that tactics are compositional: higher level tactics are constructed from more rudimentary using defined control primitives. This is similar to the approach used in LCF[12] and NuPRL[1, 8].

Given such a derivation history and a modified specification, the correspondence problem [21, 20] addresses how during replay a correspondence between program parts of the original and modified program is established.

Our approach uses a combination of name association, structural properties, and associating components to one another by intensional descriptions of objects defined in the transformations themselves.

An implementation of a rudimentary replay mechanism for our interactive development system is described. For example with the system we can first derive a program from a specification that computes some basic statistics such as mean, variance, frequency data, etc. The derivation is about 15 steps; it involves deriving an efficient means of computing frequency data, combining loops and selecting data structures. We can then modify the specification by adding the ability to compute the maximum or mode and replay the steps of the previous derivation.

1 Introduction

We are addressing the issue of *reusability* in the context in which software design is achieved by a transformational development of a formal specification of the problem into an efficient implementation. This paper explores how a specification derived as a modification of an existing design can be realized by *replaying* the transformational derivation of the original and modifying it as required by changes made to the specification. We believe tools to support such incremental reuse of designs will become an essential, integral part of software development environments of the future.

Reuse of the product, or *component reuse* is crucial to bottom-up programming. Reusable components, such as user interface packages, mathematical subroutine libraries, graphics standards, UNIX utilities, database systems etc., provide a powerful set of primitives that define a virtual machine base from which applications can

*Supported by RADC contract F30602-88-C-0127, and NSF Grant DMC-8617759. Views and conclusions contained within this report are the author's and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

be constructed. Component reuse can be enhanced by improved programming language support and improved programming environments. The need for improved programming language support is exemplified by the lack of data abstraction available in current languages. To use a scientific subroutine package that manipulates matrices requires agreement of the representation of matrices in both the package and the application. Providing an abstraction mechanism is insufficient unless it supports conversion between differing representations. An example of environmental support needed for component reuse is an indexing and retrieval system to locate relevant components.

Reuse is emerging as an important means of enhancing software productivity. It is worthwhile to distinguish the *reuse of the product* of a software development effort, i.e. code, *from reuse of the knowledge utilized in the generation of the product*.

It has been frequently observed that much of the knowledge used to create a component does not appear explicitly in the component and often not in its accompanying documentation. This knowledge is applied in making design decisions, such as choice of problem decomposition, data representation, and algorithm choice. It is claimed that loss of this information contributes to the high cost of maintenance. Reusing this knowledge is *design reuse*. To achieve design reuse, the issue of capturing and representing designs must be faced.

The basis for design capture we consider is to formalize software development within a transformational framework. In this model specifications are written in formal specification language. Implementations are derived from specifications by application of consistency-preserving transformations to an annotated abstract syntax tree representation. In our model this process is semi-automatic; transformations are applied automatically and manually. At a given point in the derivation many transformations are potentially applicable. Implementors express design decisions by selecting one of the many possible transformations applicable at each step of a derivation. Recording their selections creates a design record which captures this information.

The transformational approach applied to software development has been extensively studied [10, 17, 27, 28]. Powerful, generic techniques such as data refinement, finite differencing, loop combining, inversion, algorithm design, etc. have been developed. The field also benefits from related work in compiler optimization, software specification, theorem proving, and programming language theory and practice.

Program transformation systems are a promising but not-well developed technology. Existing systems have focussed on deriving implementations for medium-scale combinatorial computing problems [26, 9]. Ongoing work at Kestrel Institute has led to the development of a transformational system, called *KIDS*[30, 31] on which our replay work is based. Using the system we have been able to carry out derivations that carry non-trivial examples through many semantic levels and apply a wide range of design and optimization techniques. For example in one derivation we derive from a high-level specification of a topological sort a LISP implementation which is as efficient as any hand-coded version [5]. The derivation is over 40 steps long where a step involves such diverse activity as inverting maps, computing containment relations among set-theoretic data structures, simplifying expressions, combining loops, and selecting data structures.

The reuse problem in this context is to capture the design decisions expressed as the manual selection of transformation rules, so that these decisions can be *replayed* on a specification similar to the original. In a conventional SDE developers rely on a very tight loop in which they execute, modify and re-execute programs. In a transformation environment where compilation is replaced by semi-automated transformation, that loop is no longer tight. One use of replay is to tighten the loop, by replaying the transformations of the original derivation to the modified specification. Within the transformational model maintenance is done by modifying specifications and rederiving an implementation. Thus replay is essential to this attractive approach to software maintenance.

The transformational methodology supports design reuse in two interesting ways. First the creation of transformations and tactics formalizes general design knowledge in a highly reusable way. Second replay reuses design decisions made for related specifications.

The availability of a mature transformational system such as *KIDS* has proved invaluable for experimentation with replay. Conversely everyday use of *KIDS* has motivated the creation of a replay capability.

In this paper we report progress in the following areas:

- An approach to representing a *design history* was developed. The approach is to structure the derivation system using the notion of tactics, and record derivation histories as an execution trace of the application of tactics. One key idea is that tactics are compositional: higher level tactics are constructed from more rudimentary using defined control primitives. This is similar to the approach used in LCF[12, 18] and NuPRL[1, 8].

- An approach to the *correspondence problem* is described [21, 20]. The correspondence problem addresses how during replay a correspondence between program parts of the original and modified program is established Our approach uses a combination of name association, structural properties, and associating components to one another by descriptions of objects defined in the transformations themselves.

- An implementation of a rudimentary derivation management and replay mechanism for *KIDS* is described. Using the system we were able to perform a number of interesting rederivations. We have also built up a set of tools for derivation management, including the ability to store and reload derivations, browse derivations, highlight changes to program text, etc.

2 Approach

2.1 Recording Derivations

2.1.1 Representation of the Design

The first important technical problem faced in this work is the representation of the design history to be used as the basis for the replay mechanism. The process of development has been described as starting with a specification and applying a linear sequence transformations to yield an implementation. However just recording the linear sequence of development steps is inadequate. An analogy with mathematical proofs is revealing. Formally a mathematical proof is also just a linear sequence of formulas obtained by applying inference rules. This is an appropriate view for providing a simple meta-theory (e.g. to prove the soundness of the system), but for little else. Just as a mathematical proof has structure (lemmas, case analysis, formation of induction hypothesis, reformulation, etc.) and is constructed and explained in terms of that structure, a similar, but formal, structure must be devised for transformational developments.

What kinds of structures do we observe in software developments that must be formalized? We survey a few here to motivate our solution.

- One common structure is the virtual machine model. Here the specification is expressed in terms of an abstract language and then mapped in phases to successively lower-level virtual machine or language levels. Compilers are often constructed along this paradigm. The source language is at this highest abstract level. In the first phase this may be mapped to retargetable intermediate code, at a lower abstract level, and then in a second phase to assembly language. Traditional compilers rarely involve more than two phases. Boyle's [6] Lisp to Fortran transformation system goes through 7 phases, each mapping to a different virtual machine level.

- A second common structure is that of stepwise problem decomposition. A problem is decomposed into components and the implementation of each of the components proceeds independently.

- A third is the exhaustive application of a set rewriting rules. This is typical of routine simplification steps or steps that rewrite the program into a normal form. It is common to apply this strategy in conjunction with the first. Each phase mapping between language levels is the exhaustive application of a set of rewriting rules.

- A fourth structure is that of case analysis. This is of course used to express strategies which are conditioned on the form of the specification or other information about the specification supplied to the system.

This is a representative but not exhaustive list of high-level development steps found in systems that formally map specifications into implementations. Observe that these high-level development steps are compositions of more elemental steps and that they are expressible in terms of common control structures found in ordinary programming such as conditional, sequential composition, parallel composition and iteration. For example, a problem decomposition step is the sequential composition of a step which divides the problem in sub-problems, and a step consisting of the parallel composition of steps that solve the subproblems. Parallel decomposition does not impose a temporal order on development steps when no logical dependency exists.

This suggests a straight-forward approach to the problem of structuring derivations. The development system is constructed from a set of primitive operators, using composition mechanisms such as the ones described above. The resulting composite operators are called *tactics*.

This is the approach taken in LCF and NuPrl, systems aimed at the construction of mathematical proofs, not programs. It is also the approach taken by Wile [34] The recorded derivation is simply a trace of the execution of the tactics. This is a direct implementation of the notion of process programming [25] in a transformational context.

A different approach can be based on AI-style planning theory [11]. Here the description of the development step is given in terms of a *goal*– a declaratively stated postcondition that describes properties of the intended result of the step. For example, a step which transforms code into a normal form would be expressed by a declarative description of the form to be achieved. In addition to a goal structure, there are *methods*, which are operations that may be used to achieve a goal. It becomes the task of the system to synthesize a meta-program of methods whose result achieves the goal. The planning approach is a weak method because synthesizing plans is a difficult problem, and because declarative specification of post-conditions is often unwieldy.

2.1.2 An Elementary Tactic Language

This section describes an elementary tactic language sufficient to illustrate the interaction of the replay mechanism and the tactic language. A full tactic language is under development.

The tactic language is a control language. The computation responsible for transforming programs lies within *primitive tactics* written in some other language, which in our case is Refine [32] Primitive tactics are represented by Refine procedures which are called by the tactic language interpreter. The form of a primitive tactic is:

procedure-name (*parameter-list*)
    [**returns** *identifier-list*]

The *identifier-list*, and *parameter-list* are each lists, separated by commas, of an identifier followed by a colon followed by a type expression. The procedure is called supplying actual parameters, which generally are nodes of the abstract syntax tree (AST) representing the program. The procedure transforms the program as a side-effect. It returns a list of values which are then bound to the variables appearing in identifier list following the keyword **returns**. These variables are called *tactic variables*. It also returns an indication of whether the tactic succeeded or failed.

The tactic variables appearing in the identifier list must be declared in a containing tactic called an *abstraction tactic*. An abstraction tactic allows the construction of a tactic with a name, formal parameters, local variables and a body. These tactics have the form:

> *tactic-name* (*parameter-list*) =
> **let** *identifier-list* **in** *tactic*
>   **returns** *identifier-list*

An abstraction tactic is invoked the same way as a primitive tactic. The formal parameters are bound to the actual values, the local tactic variables are allocated and the tactic following the keyword **in** is executed. The tactic fails if the tactic following the keyword **in** fails.

Primitive tactics are composed using control primitives. The most elementary is *sequential composition*. This is simply denoted as $tactic_1;tactic_2;\ldots;tactic_n$. It represents the tactic which executes each tactic sequentially. This tactic fails if any of its sub-tactics fail.

The parallel execution of tactics is denoted $tactic_1 \| tactic_2 \| \ldots \| tactic_n$. It represents the tactic which executes each tactic once in any order or conceptually at least, in parallel. Parallel composition is used when there is no logical dependence among the tactics, and so no temporal order on their execution should be specified. This tactic fails if any of its sub-tactics fail.

The conditional tactic has the form

> **if** *condition* **then** *tactic*
>   **elseif** *condition* ... **else** *tactic*

The condition must be a function call which returns a boolean value. The tactic fails if the subtactic that executes fails.

The syntax $tactic_1?tactic_2$ denotes a tactic which executes $tactic_1$; if this fails it executes $tactic_2$. This is a useful exception handling mechanism.

Finally there is a repetition tactic.

> **while** *condition* **do** *tactic*

Example. This is a tactic that will exhaustively find and combine all pairs of loops that may be merged within a program part p, which is passed as a parameter.

> *Combine-Loops(p: program-part)* =
>   **let** *Loop-1 : program-part,*
>   *Loop-2 : program-part,*
>   *Combined-Loop : program-part*
>   **in**
>   **while** *exists-combinable-loops(p)*
>   *(Find-Combinable-Loops(p)*
>   **returns** *Loop1, Loop-2;*
>   *Merge-Loops(Loop-1, Loop-2)*
>   **returns** *Combined-Loop;*
>   *Simplify(Combined-Loop))*

A tactic such as *Combine-Loops* may be incorporated into another tactic or may be invoked directly by the user.

2.2 The Replay Problem

The replay problem is: given an original program P, its derivation history D, and a modified program P', utilize.

Parameter Correspondence. The execution of a tactic may cause a tactic variable to be bound to some code. Code bound to the same variable corresponds.

Structure correspondence is a weak syntatic-based notion used in the absence of stronger heuristic and semantic information generated by the other correspondence methods. It can be made more powerful by the adoption of program dependence graphs [13, 14] as the underlying representation instead of annotated abstract syntax trees. PDG's incorporate data and control flow dependencies into the representation and factor out syntactic differences that do not contribute to semantic behavior. Using PDGs instead of ASTs would require a major revision to the *KIDS* system and a better treatment by PDGs of non-scalar variables.

Parameter correspondence is a powerful notion, because it captures a semantic correspondence. Often when a tactic is applied it creates a code segment. Suppose that when the tactic is replayed a new code segment is created. With respect to the semantics encoded in the tactic both code segments play the same role and a correspondence is established. For example, a divide-and-conquer algorithm design tactic will generate identifiable code components such as code for the base case; code for dividing the problem into subproblems, etc. Parameter correspondence would identify, say, code for the base case in each derivation as corresponding.

Our replay algorithm maintains a binary relation called the *correspondence relation*. The first and second components of the relation are nodes of the AST taken from the derivation execution trace of D and $D^{'}$ respectively. Intuitively, a pair is in the correspondence relation if there is some evidence that the two pieces of abstract syntax represent code playing the same role in intermediate versions of P and $P^{'}$.

The correspondence relation is initialized as follows. The language that P and $P^{'}$ are written in is a single-assignment functional language. It has a binding construct, known as `let*`, and iteration construct `for*`. These constructs introduce local names and expressions defining the value denoted by name. A heuristic of name equivalence is implemented by initializing the correspondence relation to include pairs of AST nodes from P and $P^{'}$ that define the same variable name within corresponding program scopes. As replay proceeds the correspondence relation will be updated.

Replay proceeds by re-executing each step of the execution trace D, starting with $P^{'}$ instead of P, and using the correspondence relation to substitute actual parameters from $P^{'}$ and its derivatives for values from P. How the step is replayed is described by a case analysis based on the type of the tactic. The tactic may be a primitive tactic, an abstraction tactic, an repetition tactic, a conditional tactic, etc.

If the step to be replayed is the execution of a primitive tactic, $pt(p_1, \ldots, p_m)$ `returns` $id_1, \ldots id_n$ then the tactic pt is invoked. Actual parameters must be supplied for $p_1, \ldots, p_m$. If a parameter is a tactic variable, its current value is used. If it is a node in the AST for P, call it B, then a corresponding node in $P^{'}$ is obtained as follows: First the correspondence relation is checked. If B is paired with a corresponding node $B^{'}$ then use $B^{'}$ as the actual parameter. Otherwise starting at B traverse up the AST to the first node A for which A appears in the correspondence relation paired with some node $A^{'}$, recording the labels on the edges traversed. If there is no such A then stop at the root. Then starting at $A^{'}$ or the root of $P^{'}$, move down the AST following the same labels in reverse order to arrive at a node $B^{'}$. Use $B^{'}$ as the value of the actual parameter corresponding to B. An example is given in Figure 1. If the paths do not correspond then replay fails on that step and manual intervention is necessary. This heuristic of using path correspondence implements the structure heuristic. It recognizes that designed artifacts have component structure and substructure. In other words, components are recursively divided into subcomponents, and this parts hierarchy can be used to find corresponding components.

With all its parameters instantiated the primitive tactic is applied. If it fails, replay has failed on that step and the user is informed. Otherwise the tactic may return values to tactic variables with the `returns` clause. Parameter correspondence is implemented by augmenting the correspondence relation with pairs of AST nodes that were returned as the values of the same tactic variable

Figure 1: Establishing a Correspondence

in the original and replayed derivation. Furthermore, if any new variables were introduced by the tactic, the nodes representing those variables are made to correspond.

Other tactics are handled similarly with the exception of conditional and repetition. Suppose a conditional tactic is executed and in the original derivation the condition evaluates to true and the `then` branch is executed. If the condition during replay evaluates to false the `else` branch is executed with the correspondence relation used to instantiate parameters as described above. The correspondence relation will not be updated when executing the `else` branch. Upon conclusion of the `else` branch, normal replay continues with the step following the conditional. A similar strategy is applied to repetitions.

2.3 An Initial Implementation

In our current implementation, we have not implemented a tactic language so that each tactic is primitive. This means that parameter correspondence cannot be used, since derivation structuring information is not present. However the implementation follows the described mechanism in all other respects. We have successfully used the replay mechanism on a number of examples. The results are described in the next section.

Experience using our system has suggested many features that would make a replay system user-friendly.

Viewing. Currently the system displays a window showing all the derivation steps. The user can mouse on any step and display the program as it appears prior to the execution of the step. The user may initiate a new derivation path from that step and the resulting tree of derivations is displayed. A desired feature is the ability to have more selected views, especially when the tactic language is implemented. For example we may wish to see an "executive" view that only shows the top-level development steps. A user may wish to explode a derivation step to see its sub-tactics. A user may wish to only see tactics that succeeded; or tactics relevant to a specified part of the program.

Editing. Prior to replay the user may wish to make edit changes to the derivation, anticipating where replay may fail. For example a sequence of transformations that applied to some program part may be abstracted and reapplied to a newly introduced object. Or the user may wish to edit the derivation and reapply it to the same specification to quickly generate a new implementation.

Debugging. Replay is the reexecution of a "process program." Thus we can imagine a set of debugging tools that are entered at breakpoints or when the replay mechanism fails. The debugger will allow tactic variables to be examined or changed, examine frames of tactic invocation, and perform other activities usually provided by a debugger.

3 Results

We have used the replay mechanism on a simple example of computing basic statistics such as the mean, variance, and frequency. Figure 2 shows the initial specification. An explanation of the operators appearing in the program can be found in [5]. Figure 3 shows the development just prior to data structure selection. Each of the high-level operators such as *reduce* has been refined into loops, and these loops have been fused together so that a single pass is made over the input, and so no intermediate expressions are required. The efficiency of the computation of the map *freq* has been speeded up asymptotically by iterator inversion. Data structure selection will choose an array implementation for *freq* and the input sequence.

Next we modify the program by changing the definition of *freq* to yield histogram data, in which ranges of data values are counted, and by the inclusion of the computation of the maximum value. Figure 4 shows the modified program. Figure 5 shows the result of replay.

Even though the definition of *freq* was changed the original development was successfully applied. The other development steps, that were independent of the change, were also replayed. Finally Figure 6, shows additional development steps needed to incorporate the computation of the maximum value into the main loop of the program.

A second, more involved example is based on a scheduling problem in which precedence constrained jobs are scheduled on a uni-processor system (only one job may be scheduled at a time). In [5] we outlined this complex derivation which requires over 40 steps. We modified the specification to solve the problem of multi-processing scheduling and were able to replay successfully all of the steps of the derivation.

Figure 2: The Initial Specification

Figure 3: The Implementation of the Original Specification

Figure 4: The Modified Specification

Figure 5: The Modified Specification after Replay

Figure 6: The Final Implementation of the Modified Program

Because of the existence of a large existing base of software there is work on recovery of design knowledge from code. In [2] he emphasizes the existence of semantic clues in documentation and variable names that will aid in design recovery. We have adopted in our use of name correspondence this idea. Examples of work on design recovery can be found in [35, 33, 16].

Our tactic language is similar to [34, 18]. A richer more theoretical approach is being pursued by [29, 15] using the Deva language.

Closer to the spirit of the work reported here is work done at Rutgers University. Their work is couched in a transformational framework. Two domains are addressed: circuit designs [22] and heuristic search algorithms [24, 23].

4 Related Work

The literature on software reuse is very extensive, but most of it deals with component reuse, i.e. the reuse of subroutines. A collection of papers, edited by Biggerstaff and Perlis [3, 4] emphasizes *generative systems*, such as ours which offer design reuse and the promise greater productivity improvements in the long run. Many of the existing transformational systems are described in the collection. This is an excellent survey of the field. See also [19] for a perspective on the reuse of design plans.

There is also an extensive Artificial Intelligence literature on analogy and machine learning. Representative of work of this kind is [7].

5 Conclusions

Initial experiments with the replay system has been encouraging. Furthermore the tactic approach appears to be a sound and useful basis for making transformation systems productive vehicles for formal software development activities. While the varied use of analogy in its full generality is not captured in our work the ability to support evolutionary development and maintenance appears feasible. Without such a mechanism interactive formal development of prgrams would be impractical.

Acknowledgements. I would like to thank Greg Fisher and Tom Pressburger for useful discussions and for, along with Limei Gilham, implementing the replay system.

References

[1] Bates, J. L., and Constable, R. L. Proofs as programs. *ACM Transactions on Programming Languages and Systems 7*, 1 (January 1985), 113–136.

[2] Biggerstaff, T. J. *Design Recovery for Maintenance and Reuse.* Tech. Rep. STP-378-88, MCC Corporation, November 1988.

[3] Biggerstaff, T. J. *Software Reusability, Vol. 1: Concepts and Models.* ACM Press, New York, 1989.

[4] Biggerstaff, T. J. *Software Reusability, Vol. 2: Applications and Experience.* ACM Press, New York, 1989.

[5] Blaine, L., Goldberg, A., Pressburger, T., Qian, X., Roberts, T., and Westfold, S. *Progress on the KBSA Performance Estimation Assistant.* Tech. Rep. KES.U.88.11, Kestrel Institute, May 1988.

[6] Boyle, J. M., and Muralidharn, M. N. Program reusability through program transformation. *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 574–588.

[7] Carbonell, J. Derivational analogy: a theory of reconstructive problem solving and expertise acquisition. In *Machine Learning: An Artificial Intelligence Approach*, R. Michalski, J. Carbonell, and T. Mitchell, Eds., Morgan Kaufmann, Los Altos, CA., 1986, pp. 371–392.

[8] Constable, R. L. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, New York, 1986.

[9] Darlington, J.D. et al. A functional programming environment supporting execution, partial execution and transformation. In *PARLE 89: Parallel Architectures & Languages Europe, Vol. I: Parallel Architectures*, E. Odijk, M. Rem, and J. Syre, Eds., Springer-Verlag, New York, 1989, pp. 286–305. Lecture Notes in Computer Science, Vol. 365.

[10] Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[11] Fickas, S. F. Automating the transformational development of software. *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1268–1278.

[12] Gordon, M. J., Milner, A. J., and Wadsworth, C. P. *Edinburgh LCF: A Mechanised Logic of Computation*. Springer-Verlag, Berlin, 1979. Lecture Notes in Computer Science, Vol. 78.

[13] Horwitz, S., Prins, J., and Reps, T. Integrating non-interfering versions of programs. In *Fifteenth ACM Symposium on Principles of Programming Languages* (San Diego, CA, January 13–15, 1988), ACM, pp. 133–145.

[14] Horwitz, S., Prins, J., and Reps, T. On the adequacy of program dependence graphs for representing programs. In *Fifteenth ACM Symposium on Principles of Programming Languages* (San Diego, CA, January 13–15, 1988), ACM, pp. 146–157.

[15] Hussain, F. A., de Groote, P., Jacquard, R., Jähnichen, S., Nguyen, T. T., Sintzoff, M., and Weber, M. *Esprit Project ToolUse – Requirements and Feasibility Studies for a Development Language*. Tech. Rep. GMD 214, Gesellschaft für Mathematik und Datenverarbeitung mbH, July 1986.

[16] Letovsky, S., and Soloway, E. Delocalized plans and program comprehension. *IEEE Software 3*, 3 (May 1986), 41–49.

[17] Meertens, L. *Program Specification and Transformation (Proceedings of the IFIP TC2/WG 2.1 Working Conference)*. North-Holland, Amsterdam, 1987.

[18] Milner, R. The use of machines to assist in rigorous proof. In *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, Eds., Prentice-Hall, Englewood Cliffs, NJ, 1985, pp. 77–87.

[19] Mostow, J. Design by derivational analogy: issues in the automated replay of design plans. *Artificial Intelligence 40*, 1–3 (September 1989), 119–184.

[20] Mostow, J. Some requirements for effective replay of derivations. In *Proceedings of the Third International Machine Learning Workshop* (Skytop, PA, June 1985), Rutgers University, pp. 129–132.

[21] Mostow, J. Toward better models of the design process. *AI Magazine 6*, 1 (Spring 1985), 44–57.

[22] Mostow, J., and Barley, M. *Automated Reuse of Design Plans*. Tech. Rep. ML-TR-14, Rutgers University, May 1987.

[23] Mostow, J., and Fisher, G. Replaying transformational derivations of heuristic search algorithms in DIOGENES. In *Proceedings of the DARPA Case-Based Reasoning Workshop* (Pensicola, FL, May 1989). Available as Rutgers AI/Design Project Working Paper Number 113-3.

[24] Mostow, J., and Fisher, G. Replaying transformational derivations of heuristic search algorithms in DIOGENES. In *Proceedings of the AAAI 1989 Spring Symposium on AI and Software Engineering* (Palo Alto, CA, March 1989). Available as Rutgers AI/Design Project Working Paper Number 113-1.

[25] Osterweil, L. Software processes are software too. In *9th International Conference*

on Software Engineering (Monterey, CA, March 30–April 2, 1987), pp. 2–13.

[26] Paige, R., and Henglein, F. Mechanical translation of set theoretic problem specifications into efficient RAM code – a case study. *Journal of Symbolic Computation 4*, 2 (1987), 207–232.

[27] Partsch, H., and Steinbrüggen, R. Program transformation systems. *ACM Computing Surveys 15*, 3 (September 1983), 199–236.

[28] Pepper, P., Ed. *Program Transformation and Programming Environments.* Springer-Verlag, New York, 1983.

[29] Sintzoff, M. *Desiderata for a Design Calculus.* Tech. Rep., RM 85–13, Université Catholique de Louvain, June 1985.

[30] Smith, D. R. KIDS – a knowledge-based software development system. In *Proceedings of the Workshop on Automating Software Design* (St. Paul, MN, August 25, 1988). (also Technical Report KES.U.88.7, Kestrel Institute, October 1988).

[31] Smith, D. R. KIDS – a semi-automatic program development system. to appear in *IEEE Transactions on Software Engineering special issue on Formal Methods*, September 1990.

[32] Smith, D. R. *Structure and Design of Global Search Algorithms.* Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987. to appear in *Acta Informatica.*

[33] Soloway, E., and Johnson, W. L. PROUST: knowledge-based program understanding. *IEEE Transactions on Software Engineering SE-11*, 3 (March 1985), 267–275.

[34] Wile, D. S. Program developments: formal explanations of implementations. *Communications of the ACM 26*, 11 (November 1983), 902–911.

[35] Wills, L. M. *Automated Program Recognition.* Tech. Rep. MIT-AI-904, MIT AI Laboratory, February 1987.