A 15 Year Perspective on Automatic Programming

ROBERT BALZER

(Invited Paper)

Abstract—**Automatic programming consists not only of an automatic compiler, but also some means of acquiring the high-level specification to be compiled, some means of determining that it is the intended specification, and some (interactive) means of translating this high-level specification into a lower-level one which can be automatically compiled.**

We have been working on this extended automatic programming problem for nearly 15 years, and this paper presents our perspective and approach to this problem and justifies it in terms of our successes and failures. Much of our recent work centers on an operational testbed incorporating usable aspects of this technology. This testbed is being used as a prototyping vehicle for our own research and will soon be released to the research community as a framework for development and evolution of Common Lisp systems.

Index Terms—**Automatic programming, evolution, explanation, knowledge base, maintenance, prototyping, specification, transformation.**

INTRODUCTION

THE notion of automatic programming has fascinated our field since at least 1954 when the term was used to describe early Fortran compilers. The allure of this goal is based on the recognition that programming was, and still is, the bottleneck in the use of computers. Over this period we have witnessed phenomenal improvements in hardware capability with corresponding decreases in cost. These price/performance gains reached the point in the early 1980's that individual dedicated processes became cost effective, giving birth to the personal computer industry. This hardware revolution apparently will continue for at least the rest of the decade.

Software productivity has progressed much more slowly. Despite the development of high-level, and then higher-level, languages, structured methods for developing software, and tools for configuration management and tracking requirements, bugs, and changes, the net gain is nowhere close to that realized in hardware.

But the basic nature of programming remains unchanged. It is largely an informal, person-centered activity which results in a highly detailed formal object. This manual conversion from informal requirements to programs is error prone and labor intensive. Software is produced via some form of the "waterfall" model in which a linear progression of phases is expected to yield correct results. Rework and maintenance are afterthoughts.

Unfortunately, this existing software paradigm contains two fundamental flaws which exacerbate the maintenance problem.

First, there is no technology for managing the knowledge-intensive activities which constitute the software development processes. These processes, which convert a specification into an efficient implementation, are informal, human intensive, and largely undocumented. It is just this information, and the rationale behind each step, that is crucial, but unavailable, for maintenance. (This failure also causes problems for the other lifecycle phases, but is particularly acute for maintenance because of the time lag and personnel changeover normally involved, which precludes reliance on the informal mechanisms such as "walking down the hall," typically used among the other phases.)

Second, maintenance is performed on source code (i.e., the implementation). All of the programmer's skill and knowledge has already been applied in optimizing this form (the source code). These optimizations spread information; that is, they take advantage of what is known elsewhere and substitute complex but efficient realizations for (simple) abstractions.

Both of these effects exacerbate the maintenance problem by making the system harder to understand, by decreasing the dependences among the parts (especially since these dependences are implicit), and by delocalizing information.

With these two fundamental flaws, plus the fact that we assign our most junior people to this onerous task, it is no wonder that maintenance is such a major problem with the existing software paradigm.

Against this background, it is clear why automatic programming commands so much interest and appeal. It would directly obviate both these fundamental flaws through full automation of the compilation process.

But programming is more than just compilation. It also involves some means of acquiring the specification to be compiled and some means of determining that it is the intended specification. Furthermore, if one believes, as we do, that optimization cannot be fully automated, it also involves some interactive means of translating this high-level specification into a lower-level one which can be automatically compiled.

This is the extended automatic programming problem, and the main issue is how to solve it in a way that still obviates the two fundamental flaws in the current paradigm.

ISI's Software Sciences Division has been pursuing this

Manuscript received June 3, 1985; revised August 2, 1985. This work was supported in part by the Defense Advanced Research Projects Agency under Contract MD903 81 G 0335, in part by the National Science Foundation under Contract F30602 81 K 0056, and in part by RADC under Contract MCS-8304190.

The author is with the Information Sciences Institute, University of Southern California, Marina del Rey, CA 90292.

goal for nearly 15 years. This paper describes the perspective and approach of this group to the automatic programming problem and justifies it in terms of the successes and failures upon which it was built.

The Extended Automatic Programming Problem

Automatic programming has traditionally been viewed as a compilation problem in which a formal specification is complied into an implementation. At any point in time, the term has usually been reserved for optimizations which are beyond the then current state of the compiler art. Today, automatic register allocation would certainly not be classified as automatic programming, but automatic data structure or algorithm selection would be.

Thus, automatic programming systems can be characterized by the types of optimizations they can handle and the range of situations in which those optimizations can be employed. This gives rise to two complementary approaches to automatic programming. In the first (bottom-up) approach, the field advances by the addition of an optimization that can be automatically compiled and the creation of a specification language which allows the corresponding implementation issue to be suppressed from specifications. In the second (top-down) approach, which gives up full automation, a desired specification language is adopted, and the gap between it and the level that can be automatically compiled is bridged interactively. It is clear that this second approach builds upon the first and extends its applicability by including optimizations and/or situations which cannot be handled automatically.

Hence, there are really two components of automatic programming: a fully automatic compiler and an interactive front-end which bridges the gap between a high-level specification and the capabilities of the automatic compiler. In addition, there is the issue of how the initial specification was derived. It has grown increasingly clear that writing such formal specifications is difficult and error-prone. Even though these specifications are much simpler than their implementations, they are nonetheless sufficiently complex that several iterations are required to get them to match the user's intent. Supporting this acquisition of the specification constitutes the third and final component of the extended automatic programming problem.

This paper is organized into sections corresponding to each of these three components of the extended automatic programming problem, captured as a software development paradigm, as shown in Fig. 1. Throughout this paper we will elaborate this diagram to obtain our goal software lifecycle paradigm. In fact, our group has focused its efforts much more on the specification acquisition and interactive translation components than it has on the traditional automatic compiler task.

Specification Acquisition

Informal to Formal

Our entry into this field was preceded by a survey [3] which categorized then current work. We identified specification acquisition as a major problem and decided to focus our initial efforts on the earliest aspects of this task—namely, converting informal specifications into formal ones. Our SAFE project [8] in the early 1970's took a (parsed) natural language specification as input and produced a formal specification as output. This system was able to correctly handle several small (up to a dozen sentences) but highly informal specifications (see Fig. 2(a) and (b).

The basis for its capability was its use of semantic support to resolve ambiguity. Its input was assumed to be an informal description of a well-formed process. Hence, it searched the space of possible disambiguations for a well-formed interpretation. This search space was incrementally generated because each informal construct was disambiguated as it was needed in the dynamic execution context of the hypothesized specification. This dynamic execution context was generated via symbolic evaluation.

This use of the dynamic execution context as the basis of semantic disambiguation was strong enough that, at least for the simple specifications on which the system was tested, most ambiguities were locally resolved, and rather little (chronological) backtracking was needed.

Many key aspects of our approach to automatic programming (as explained later in the paper) found their first expression in this system: executable specifications and the use of symbolic evaluation to analyze each execution, an explicit domain model (constructed by the system), and the database view as a specification abstraction of software systems (as embodied in AP3 [18], the knowledge representation system for SAFE).

Our modest success with SAFE led us to consider the problems of scaling the system up to handle realistic sized specifications. We recognized that we were solving two hard problems simultaneously: the conversion from informal to formal, and the translation from high-level formal to low-level formal. The first was the task we wanted to work on, but the latter was required because a suitable high-level formal specification language did not exist.

Formal Specification

We decided to embark on a "short" detour (from which we have not yet emerged) to develop an appropriate high-level specification language. The result of this effort was the Gist language [1], [20].

Gist differs from other formal specification languages in its attempt to minimize the translation from the way we think about processes to the way we write about them by formalizing the constructs used in natural language. We designed Gist from an explicit set of requirements [4] which embodied this goal and the goal of using such specifications in an automation-based software lifecycle (evolved from Fig. 1). It is indeed surprising and unfortunate that so few languages have been designed this way.

This requirements-based design resulted in a language with a novel combination of features.

Global Database: We conceive of the world in terms of objects, their relationships to one another, and the set of operations that can be performed on them. At the specification level, all these objects must be uniformly accessible.

Operational: Gist specifications are operational (i.e.,

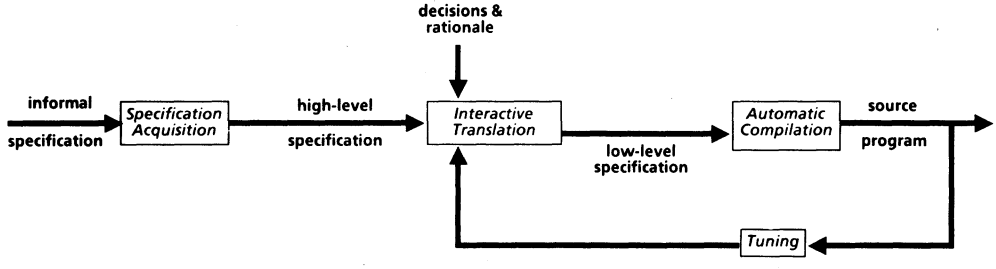

Fig. 1. Extended automatic programming paradigm (initial version).

have an executable semantics) because they describe systems which display behavior (i.e., modify the objects defining some environment). These systems are defined in terms of a set of interacting agents (i.e., concurrent processes) which contribute to the system's overall behavior. The Gist specification must constructively generate this set of intended system behaviors. This operational semantics also allows the specification to be reasoned about as programs (although highly inefficient ones).

Perfect Knowledge: The distinction between explicit and implicit information must be hidden at the specification level (this is an optimization issue). It must appear that all derivable information is already contained in the global database. This implies the need for inference and/or derivation rules, which are self-organizing (used as needed). Thus, any computation which does not change state, but merely the form of information, is expressed via such rules (as in Prolog).

Descriptive Reference: This is the ability to access objects via a description of their relationships with other objects. Since the relationships may be used to access any of the objects participating in the relationship, associative retrieval is implied. If the description is underconstrained, then one member of the set of described objects is nondeterministically selected.

Historical Reference: Just as descriptive reference is used to access objects by descriptions from the current state, historical reference allows this same access capability from previous states. The absence of such a capability forces the specifier to create memory structures to hold needed historical information and to maintain them dynamically. Such compilation activities should have no place in specifications.

Constraint Avoidance: The semantics of Gist is that the generated set of behaviors does not violate any constraints. Gist specifications normally contain much nondeterminism (in descriptive and historical references, and in the interleaving of concurrent processes), which generates alternative possible behaviors. Only those behaviors that do not violate any constraints are actually generated. Thus, constraints can be thought of as pruning a set of overly general behaviors to the acceptable subset. Gist specifications usually specify a simple, overly general process for performing some task and then some constraints which "force" it to generate only the desired behaviors.

Closed System: In order to define the interactions of the specified system with its environment, that environment must also be specified. This allows the entire specification to be simulated and analyzed.

The resulting language was used to specify several real applications [15]–[17] and has recently been adopted as the specification language for a software development environment being built by a software engineering course at USC. An example of a Gist specification is given in Fig. 3.

Specification Readability

We believe that Gist successfully allows specifiers to create formal specifications which are cognitively close to their conceptualization of the specified system. We had assumed that this cognitive closeness would result in specifications that were also easy to read.

Unfortunately, this assumption was false. Gist specifications were nearly as hard to read as those in other formal specification languages. We soon realized that the problem was not particular to Gist, but extant across the entire class of formal specification languages. In their effort to be formal, all these languages have scrubbed out the mechanisms which make informal languages understandable, such as summaries and overviews, alternative points of view, diagrams, and examples.

To partially overcome this problem, we built a paraphraser [30] that takes a formal Gist specification as input and produces a natural language paraphrase as output. This paraphrase both summarizes and reorganizes the specifications to provide an alternative point of view. Fig. 3 shows an example of a Gist specification and its paraphrase.

As can be readily seen from this example, the paraphraser produces a remarkable improvement in readability. In fact, it does better than that. It also helps uncover errors in the specification (places where the specification differs from intent). We found such errors in specifications we had carefully constructed and "knew" were correct. The paraphraser made it obvious that the formal specification did not match our intent (the paraphraser cannot detect such problems itself, but it makes it easy for us to do so).

We are starting a joint effort with TRW to convert the current laboratory prototype paraphraser into a practical tool and to make it applicable to their specification language, RSL [10].

Symbolic Evaluation and Behavior Explanation

The paraphraser pointed out that what we write is often not what we had intended. But it does not help uncover mismatches between our intent and what we really wanted. A specification, after all, is really a generator of behaviors, and the real question is whether the intended set of behaviors was generated. While the paraphraser can help

```
* ((MESSAGE ((RECEIVED) FROM (THE "AUTODIN-ASC"))) (ARE PROCESSED) FOR (AUTOMATIC DISTRIBUTION
ASSIGNMENT))

* ((THE MESSAGE) (IS DISTRIBUTED) TO (EACH ((ASSIGNED)) OFFICE))

* ((THE NUMBER OF (COPIES OF (A MESSAGE) ((DISTRIBUTED) TO (AN OFFICE)))) (IS) (A FUNCTION OF
(WHETHER ((THE OFFICE) (IS ASSIGNED) FOR (("ACTION") OR ("INFORMATION"))))))

* ((THE RULES FOR ((EDITING) (MESSAGES))) (ARE) (: ((REPLACE) (ALL LINE-FEEDS) WITH (SPACES))
((SAVE) (ONLY (ALPHANUMERIC CHARACTERS) AND (SPACES))) ((ELIMINATE) (ALL REDUNDANT SPACES))))

* (((TO EDIT) (THE TEXT PORTION OF (THE MESSAGE))) (IS) (NECESSARY))

* (THEN (THE MESSAGE) (IS SEARCHED) FOR (ALL KEYS))

* (WHEN ((A KEY) (IS LOCATED) IN (A MESSAGE)) ((PERFORM) (THE ACTION ((ASSOCIATED) WITH (THAT
TYPE OF (KEY))))))

* ((THE ACTION FOR (TYPE-0 KEYS)) (IS) (: (IF ((NO OFFICE) (HAS BEEN ASSIGNED) TO (THE MESSAGE)
FOR ("ACTION")) ((THE "ACTION" OFFICE FROM (THE KEY)) (IS ASSIGNED) TO (THE MESSAGE) FOR
("ACTION"))) (IF ((THERE IS) ALREADY (AN "ACTION" OFFICE FOR (THE MESSAGE))) ((THE "ACTION"
OFFICE FROM (THE KEY)) (IS TREATED) AS (AN "INFORMATION" OFFICE))) (((LABEL OFFS! (ALL
"INFORMATION" OFFICES FROM (THE KEY))) (ARE ASSIGNED) TO (THE MESSAGE)) IF ((REF OFFS! THEY)
(HAVE (NOT) (ALREADY) BEEN ASSIGNED) FOR (("ACTION") OR ("INFORMATION"))))))

* ((THE ACTION FOR (TYPE-1 KEYS)) (IS) (: (IF ((THE KEY) (IS) ((FIRST TYPE-1 KEY ((FOUND) IN
(THE MESSAGE)))) THEN ((THE KEY) (IS USED) TO ((DETERMINE) (THE "ACTION" OFFICE)))) (OTHERWISE
(THE KEY) (IS USED) TO ((DETERMINE) (ONLY "INFORMATION" OFFICES)))))
```

Fig. 2. (a) Actual input for message processing example.

us detect some of these mismatches by making the generator (i.e., the specification) more understandable, many mismatches can only be detected by examining the behaviors themselves.

There are three classes of tools relevant to this task. The first is a theorem prover. It could be used to prove that all behaviors have some desired set of properties. We have not investigated this approach because it is hard to completely characterize the intended behavior via such properties, and because this approach only checks the expected. A major problem is that unexpected behaviors occur. The remaining two classes directly examine behaviors, and so are able to detect these unexpected behaviors.

The second class of tools is an interpreter for the specification language. Since we have required that our specification language be operational, all specifications are directly executable (although potentially infinitely inefficient). The major advantage of this type of tool is that it provides a means for augmenting and testing the understanding of the specification through examples. Its major problem is the narrow, case by case, feedback that it provides.

This has caused us to focus our efforts on the remaining class of tools: symbolic evaluation. Symbolic evaluation differs from normal evaluation by allowing the test case to be partially specified [11]. Those aspects not specified are treated symbolically with all cases (relevant to the specification) automatically explored. This provides a means of exploring entire classes of test cases simultaneously. These cases are only subdivided to the extent that the specification makes such distinctions. These subcases are automatically generated and explored. The size of the test case explored by symbolic evaluation is determined by the degree to which it is only partially specified (in the limit, when the case is completely specified, symbolic evaluation reduces to normal interpretation).

We have built a prototype evaluator for Gist. It produces a complex case-based exploration of the symbolic test case. It retains only the "interesting" (as defined by a set of heuristics) consequences of the behavior generated for the test case. To make this behavior understandable, we have also built a natural language behavior explainer (see [27] for details of this work).

Our intent is to provide both static (via the specification paraphraser) and dynamic (via the symbolic evaluator) means for validating a specification. We imagine an intense iterative cycle during the creation of a specification in which the validation cycles progress from the static feedback provided by the specification paraphraser to the dynamic feedback provided by the symbolic evaluator. During this period the specification is being used as a fully functional (although highly inefficient) prototype of the target system. These augmentations to the software lifecycle are shown in Fig. 4.

Maintenance

We also realized that the revisions made to the specification during this validation cycle are just like those that arise during maintenance. This realization suggested a radical change in how maintenance is accomplished: modify the specification and reimplement. This change in the software lifecycle (as shown in Fig. 5) resolves the fundamental flaw in the current software lifecycle. By performing maintenance directly on the specification, where information is localized and loosely coupled, the task is greatly simplified because the optimization process, which

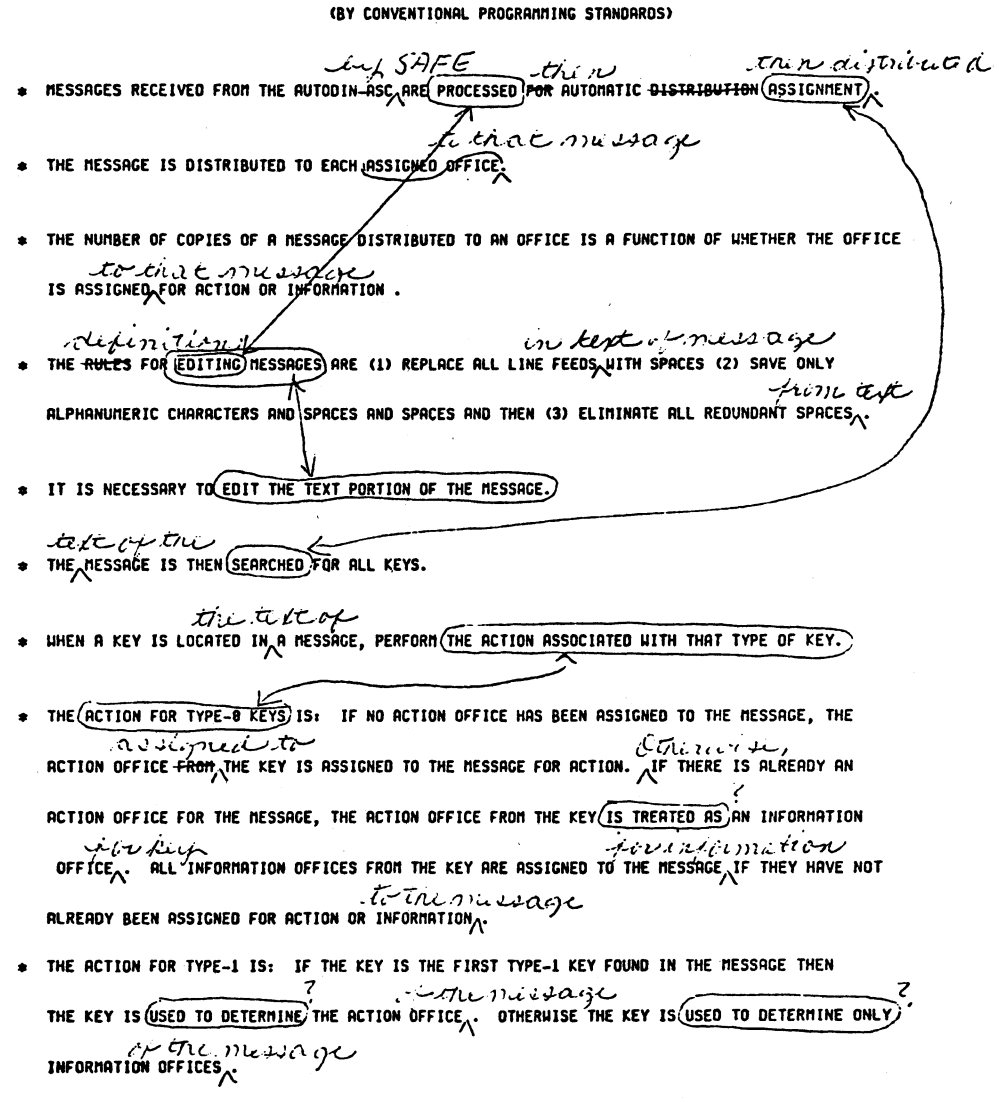

Fig. 2. (*Continued.*) (b) Hand annotated ambiguities in message processing example.

spreads information and builds up (largely implicit) interconnections between the parts, has not yet occurred. In the current lifecycle, we attempt to maintain the optimized source code, with predictable consequences.

At the specification level, such maintenance is almost always simple (if not trivial), usually explainable in a few sentences. This corresponds to the fact that the logical trajectory of a system changes slowly, in part limited by users' ability to absorb such changes. It is only at the implementation level that these small logical changes can have large, global, and complex effects. However, rather than trying to "patch" the previous implementation to obtain the next one, these problems are avoided by deriving the new implementation from the revised specification. To the extent that the implementation process is automated and guaranteed to produce valid implementations, this option becomes feasible. Notice that this is what we have always done with compilers (it is just that currently, much of the optimization process has already been manually performed because existing compilers only accept lower-level implementation-oriented languages, and maintaining this portion manually is difficult and error prone).

Incremental Specification

We noted above that maintenance modifications can usually be explained in a few sentences. This is because natural language provides us with a metalanguage mechanism which enables previous utterances to be modified. This capability is crucial for our major explanation strategy: gradual elaboration. We explain things by starting with a simplified kernel, which is repeatedly elaborated and revised. Each cycle is a better approximation of the final description. We simplify earlier stages to improve comprehension, and in so doing, tell "white lies," which are corrected by later revisions.

These simplified overview approximations are central to our ability to communicate with one another, but, as noted earlier, are totally lacking from our formal specification languages. The lack of this capability, and the metalevel mechanisms for modifying previous utterances upon which it depends, constrains the current generation of formal specification languages to a "batch" semantics in which the information is order independent. Hence, the entire specification must be understood at once, and desired changes must be added to the specification by re-

```
The network

type location() supertype of
 < source(SOURCE_OUTLET | pipe);
  pipe(CONNECTION_TO_SWITCH_OR_BIN | (switch union bin) ::unique);
  switch(SWITCH_OUTLET | pipe :2, SWITCH_SETTING | pipe)
        where always required
              switch:SWITCH_SETTING = switch:SWITCH_OUTLET end;
  bin()
 >;

Packages - the objects moving through the network

type package(LOCATION | location, DESTINATION | bin);
```

There are packages, sensors, package_routers and environments.

Bins, switchs, pipes and sources are locations.

Each switch has 2 switch_outlets which are pipes. Each switch has one switch_setting which is a pipe.

Always required:

The switch_setting of a switch must be a switch_outlet of the switch.

Each pipe has one connection_to_switch_or_bin which is a switch or bin.

Each switch or bin is the connection_to_switch_or_bin of one pipe.

Each source has one source_outlet which is a pipe.

Each package has one location. Each package has one destination which is a bin.

Bins and switchs are sensors.

Fig. 3. Example of a Gist specification.

writing the whole specification with the change integrated in. This "compilation" process produces a larger, more complex batch specification which must be understood as a whole. Clearly, this process breaks down after a certain size and complexity are reached.

Instead, we need to learn how to tell "white lies" in these formal languages and how to define metamechanisms which modify previous utterances. These capabilities would create a new generation of "incremental" specification languages. We have begun to explore the dimensions along which change can occur in such languages [19], and to define a complete set of high-level modifications to the domain model portion of specifications in such languages [7].

Defining such an incremental specification language and understanding its implications on our automated software development lifecycle is our major specification goal.

Interactive Translation

After obtaining a valid high-level specification via the iterative and/or incremental derivation process described above, this high-level specification must be interactively translated into a low-level specification that can be automatically compiled. The amount of work involved in this phase is dependent on the distance between these two specification levels.

In our efforts, this gap is quite large because we have consciously defined Gist to minimize the difference between the way we think about processes and the way we write about them. This decision maximized the work involved in this iterative translation phase. It includes "traditional" automatic programming issues of representation and algorithm selection as well as several issues specific to Gist, such as algorithm organization (to guarantee the nonviolation of constraints), cache detection and organization (to handle inferencing), state saving (to handle historical reference), and algorithm decomposition (to break the single Gist specification and its global database into handleable modules).

Any transformation-based approach, such as ours, is inevitably drawn to the notion of a wide spectrum language, as first enunciated by Bauer [9], in which a single language contains both specification and implementation constructs. Each translation is from and into the same language because each only affects a small portion of the specification (the transformations generally either replace higher-level constructs with lower-level ones within the same language, or reorganize constructs at the same level). Use of a wide spectrum language has two other pragmatic benefits. First, it allows transformations to be less order dependent (if there is a succession of languages, then transformations can only be employed while their language is in effect). Second, it avoids having to build separate transformation languages and analysis packages for each language.

Finally, it should be noted that combining a wide spectrum language with an operational semantics allows a single program semantics and its associated analysis to be used throughout a transformational implementation.

Our first transformation efforts were based on manually applied transformations [5]. It was immediately apparent that the clerical work involved was untenable and that the transformations had to be applied by the system.

We decided to embed such a capability in Popart [31], a grammar-driven development system we were creating. This was a most fortunate choice because this grammar-driven aspect was central to our first major advance in transformations, both in its implementation and in our recognizing the opportunity and need for this capability.

Formal Development Structure

We recorded the sequence of transformation applications to provide a derivation history. To examine this history, it was natural to use the Popart editor. This necessitated building a grammar for this structure, which in turn forced us to consider the language for expressing such structures. This led to the observation that rather than representing the derivation history directly, we should describe how to generate that history, i.e., a program for transformational derivation (called a formal development).

The resulting language, Paddle [32], enabled us to operationally specify the structure of the sequence of transformations needed to implement some system. That specification could be executed to obtain the implementation. However, we could also use the Popart editor to modify the formal development and then reexecute it.

Paddle's contribution lies in the combination of these two capabilities. It formally represented a structured transformational derivation, and the fact that this representation was generative provided us with the insight that such derivations should be iteratively and incrementally developed.

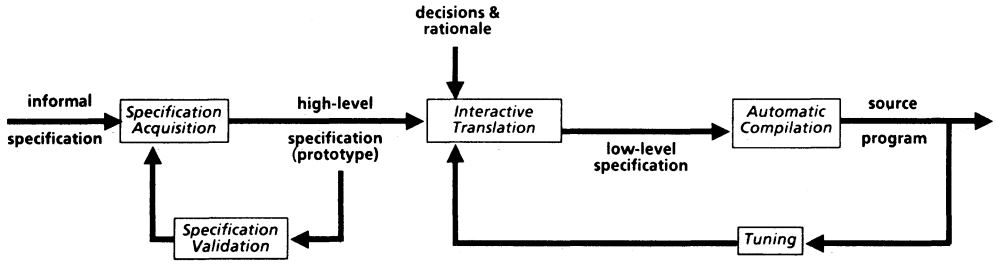

Fig. 4. Extended automatic programming paradigm (intermediate version 1).

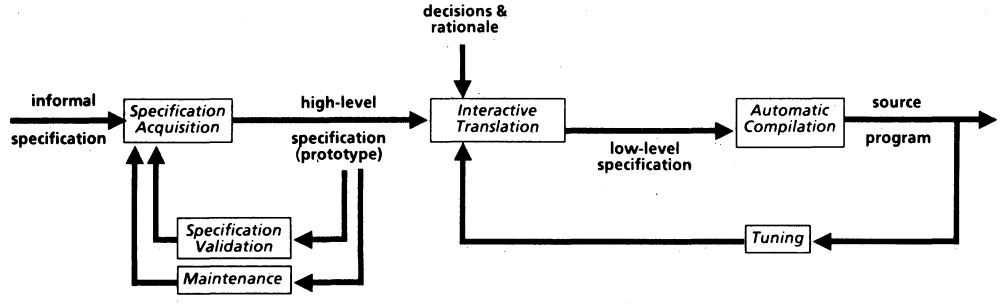

Fig. 5. Extended automatic programming paradigm (intermediate version 2).

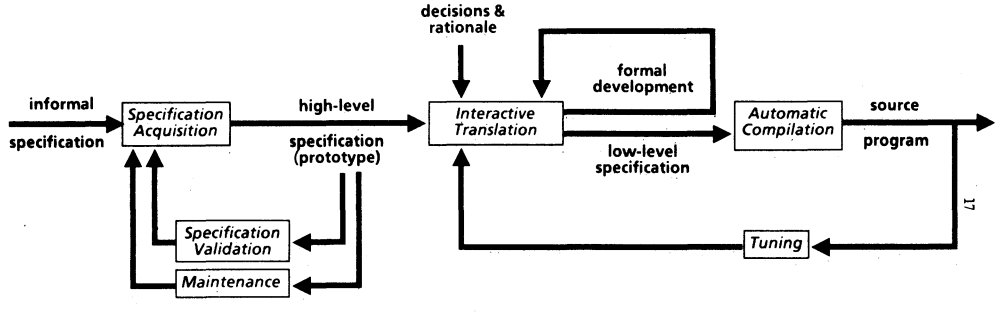

Fig. 6. Extended automatic programming paradigm.

In fact, it is also the basis for our approach to maintenance. As we described previously, we believe that maintenance should be performed by modifying the specification and rederiving the implementation. This strategy is ideal when the implementation is fully automatic because the modified specification then only needs to be "recompiled."

Replay

However, in an interactive translation approach, such as our own, while much of the effort has been automated, there is still considerable human involvement. This makes it unfeasible to rederive the entire implementation each time the specification is modified. However, most of the implementation decisions remain unchanged from one cycle to the next, so rather than rederiving the implementation from the specification, we modify the Paddle program which generated the previous implementation to alter only appropriate implementation decisions and then "replay" (i.e., rerun) it to generate the new implementation.

Reimplementing a specification by modifying the formal development (the Paddle program) and replaying it represents the final elaboration to our extended automatic programming paradigm (see Fig. 6). The formal development becomes an output of interactive translation in addition to the low-level specification which is fed to the automatic compiler. This formal development is also an additional input to the next cycle of interactive translation, in which it is modified and replayed to generate the next version of the low-level specification.

This replay mechanism, and the formal development on which it is based, is the means by which this paradigm still eliminates the two fundamental flaws in the current software paradigm without requiring the unachievable goal of full automation. The flaw of informal and undocumented translations is eliminated by the formal development which formally documents and records the optimization decisions made by the human developer, thus making them accessible to other developers for later maintenance or enhancement of the system. The flaw of trying to modify optimized source code is eliminated by replay, which provides enough automation of the optimization process that a new implementation can feasibly be rederived directly from the specification.

An earlier version of this extended automatic programming paradigm, which did not differentiate between the interactive translation and automatic compilation phases, was adopted for the Knowledge-Based Software Assistant [21] and included as a long-range objective of the Stars program [6].

Automating Interactive Translation

Achieving such a replay capability is our major transformation goal. The main impediment is the operational rather than specificational nature of Paddle. We currently define how to accomplish the translation from high- to low-level specification rather than describe our optimization decisions directly. We need to define a much higher-level language than Paddle for describing these decisions, and just as with conventional computer languages, back them up with correspondingly high-level translation. Here, this translation involves a problem solver, which satisfies optimization goals by constructing appropriate transformation sequences.

A first step in this direction is the Glitter system [32]. In a narrowly defined and restricted domain, it was able to achieve an order of magnitude decrease in the need for interactive input. This type of automation is triply effective. It not only automates the initial development of the implementation, but even more importantly, it enhances its replay capability in two ways. By raising the formal development's level of description and reducing its operational character, it becomes easier to both understand and modify. The biggest effect, however, is that the formal development often may be replayed without modification because the changes in its operational expansion, necessitated by the revision of the program specification, are being filled in by the problem solver rather than being an explicit part of the Paddle specification, and they can be automatically rederived by the problem solver to fit the new situation.

The creation, modification, and implementation of Paddle's formal development structure is exactly analogous to that of program specifications themselves. It is therefore reasonable to ask whether automatic compilation (the transformation problem solver) can be any more successful here than we argued it could be with program specifications in general. Our answer is that it cannot. Just as with program specification, we will inevitably want to write formal development specifications which are beyond the automatic compilation state of the art. We will therefore need an interactive translation facility to bridge this gap.

In fact, we need to employ our entire extended automatic programming paradigm to create and maintain the formal development. This potentially endless recursion is broken, as always, by the human inability to tolerate more than one or two recursive levels. On an engineering basis, we will limit our specifications at some level to the capabilities of that level's automatic compiler. The price is the corresponding locking of the strategies of that level into an explicit operational structure which must be manually maintained [24].

Thus, our entire approach to interactive translation is based on the ability to represent the translation in a language which is executable and upon which our development tools and technology can be applied. We are indeed fortunate that our initial efforts with Popart led us in this direction.

Transformations

While most of our effort on transformations has focused on the technology (language, tools, control structure, etc.), rather than on individual transformations, we have developed some transformations for areas of special interest for Gist. These transformations map away the specification freedoms provided by Gist: specifically, historical reference, derived relations, demons, associative retrieval, and angelic evaluation (nondeterministic execution which proceeds in such a way that constraints will not be violated) [23].

We have also investigated how constraints defined on a system can be refined into constraints on the parts of that system [14]. As with most of the areas listed above, this issue has not received much attention because, along with other specification languages, it had to be resolved before a formal specification could be written. Gist allows these issues to remain unresolved in the formal specification, i.e., to be a specification freedom.

In general, for each of these freedoms we have identified the implementation issues involved, characterized the space of alternatives, and provided a few example transformations to map a subset of the instances of the freedom into an implementation. We have not attempted the much larger task of building a comprehensive library of such transformations.

One area where we did attempt to build such a comprehensive set of transformations was for implementing derived relations in software caches. We observed that much of the complexity of our programs was due to building and maintaining caches of various kinds. We therefore decided to try to build a system which automated this aspect of programming [25].

This system employed a comprehensive set of transformations for creating and maintaining a variety of different types of caches. Furthermore, it automatically determined when, where, and how to install them (based on an analysis of costs compiled from interactively acquired data). It extends previous work on caching by dealing with real Lisp programs with side-effects and shared list structure and which evolve (via an editor). It was able to cache several Lisp programs, including itself.

However, our decision to have it work on arbitrary Lisp programs greatly complicated its analysis and limited its ability to safely install caches. Revising it to work in the more structured Gist environment should alleviate this difficulty.

Automatic Compilation

As indicated earlier, most of our group's effort has focused on the specification derivation and interactive translation phases. However, recently we have started two significant efforts in automatic compilation: explainable expert systems and annotations.

Narrowed Focus

The Explainable Expert Systems project [28] is specializing our general software development paradigm to the narrower area of expert system construction. This focus,

and the fact that the structure of these systems is regular and declarative, has allowed us to match the specification language with the automatic compiler and (attempt to) eliminate the need for interactive translation. The basis for this effort is Swartout's thesis [26], [29], in which he observed that good explanations of the behavior of a system required not only an ability to paraphrase the code of the system, but also an ability to describe the relationship of the implementation to the (possibly implicit) specification from which it was derived and the rationale behind the design decisions employed in the translation.

Rather than attempting to extract this much information from a human designer, he built a special-purpose automatic programmer which dropped mental breadcrumbs along the way (recorded its design decisions in a development history). We are extending this automatic compiler to build rules from domain principles applied to specific domain facts and to integrate rules together into methods. Each such capability that the automatic compiler can handle can be removed from the input specification. Since so little effort has yet occurred in specifying rather than programming expert systems, considerable progress can be made before we reach the limits of our ability to extend the automatic compiler.

Annotations

Our annotation effort is exploring a quite different approach to automatic compilation. Rather than extending the compiler's ability to make optimization decisions automatically, we are attempting to extend its ability to carry out the decisions of a human designer. These decisions are represented as declarative annotations of the specification. The human designer is interested only in the eventual achievement of these decisions, not in interactively examining the resulting state and using it as the basis for further decisions.

Thus, this annotation approach is really the logical limit of our attempts to get more declarative specifications of the formal development. Our reason for considering it here is that it basically trivializes the interactive translation phase to make syntactic annotations while enhancing the role of the automatic compiler, which follows the guidance provided by those annotations.

We believe that the annotation approach has great promise because it enables the human designer to make decisions in a nonoperational way, which facilitates enhanced replay, decreases the need to interactively examine intermediate states of the implementation, and allows the automatic compiler to consider these decisions in any order, or even jointly.

Our first target of opportunity for annotation is representation selection. Towards this end, we are constructing a new knowledge representation system, AP5 [13], to replace AP3. Besides cleaning up the semantics of AP3, its main purpose is to allow heterogeneous representations of knowledge. AP5 defines relations, its basic unit of representation, as an abstract data type. Any implementation which satisfies the formal interface thus defined is acceptable. Annotations allow the user to select among several predefined implementations, or to define new, special-purpose ones. The AP5 compiler is concerned with translating specifications which use the abstract knowledge base interface to access and manipulate logical data into efficient programs [12]. The term "virtual database" has been coined to describe such data and knowledge bases which logically hold and manage data, which are actually represented heterogeneously and directly accessed and manipulated by compiled database applications. We expect large improvements in efficiency in such applications over the homogeneous and actual (as opposed to virtual) interface currently employed in AP3 programs.

Formalized System Development

Nearly three years ago, we confronted the realization that although we had broadly explored the extended automatic programming problem and conceived a new paradigm for automated software development, all of our insight and experience was based on pedagogical examples. None of our tools or technology had progressed to a usable state.

We believe very strongly in the evolution notion, embedded in our software paradigm, that large, complex systems cannot be predesigned, but rather, must grow and evolve on the basis of feedback from use of earlier simpler versions. We know that our insights and perceptions are necessarily limited and preliminary, and that only by actually experiencing such an environment would they significantly deepen. We also felt that enough pieces of technology had been developed that we could begin to benefit from their use.

We therefore committed ourselves to building an operational testbed capable of supporting our own software department. This testbed will be used to experiment with tools and technologies for supporting our software paradigm and for studying alterations to it. The testbed became operational in January 1984 and, since then, is the only system we have used for all our computing needs. By the end of this year it will be available to the research community as a framework for development and evolution of Common Lisp systems.

To build this testbed, called formalized system development (FSD) [2]), we had to overcome two major obstacles to using our new software paradigm. The first was establishing a connection between our high and low levels of specification. Because we attempted to minimize the difference between the way we think about processes and the way we write about them, we created a large gap between these two levels of specification. We still have a long way to go in creating an appropriate interactive translation framework, stocking it with a suitable and comprehensive set of translations, and providing an adequate level of automation to obtain a usable interactive translation facility, even for quite sophisticated programmers.

This lack of connection has bedeviled our efforts right from the start (with SAFE [8]) and has been the major impediment to our use of this new paradigm. It has iso-

lated our high-level specification language from our technology and relegated its use to pedagogical examples.

Overcoming this problem is simple, but philosophically painful. By suitably lowering the level of our "high-level" specification (hereafter called the prototype specification), we can bring it close enough to the lower level of specification to bridge via interactive translation. The cost of such a lowering of the input prototyping specification is that more optimization issues must be handled manually outside the system, thus complicating maintenance.

We decided to initially limit interactive translation to providing annotations for the automatic compiler (as discussed in the annotations section). We believe we currently understand how to annotate data representations, inference rules, and demon invocations, and how to compile those annotations. This then defines the initial difference between the prototyping and low-level specifications.

The rest of the high-level specification is therefore determined by the capabilities of our automatic compiler. This brings us to our second major obstacle to using our new software paradigm: automatic compilation. The issue is both what can be compiled and what constitutes an adequate level of optimization.

From the very start of our efforts, we were writing our tools in terms of a knowledge representation system, FSD [18], which directly incorporated many aspects of our specification semantics. We found that AP3's direct implementation of these semantic constructs, while far from what could be produced by hand, still provided adequate response. Major portions of the FSD testbed are constructed this way, with the same acceptable performance result. We therefore decided that, in general, this is an appropriate target level for our automatic compiler. In places where further optimization is required, we will use annotations to direct the automatic compiler. This compiler will actually be a two-phase compiler. The first phase will translate the annotated low-level specification language into calls on the AP5 knowledge-base system. The AP5 compiler will then use the annotations to perform any further indicated optimizations. A major reason for our shift from AP3 to AP5 is the latter's capability to compile annotations into heterogeneous representations. We believe that this additional, directed level of optimization will enable us to write the entire FSD system in the prototyping specification language.

We have not yet addressed the simplifications required to convert Gist into a compilable prototyping language. These simplifications are defined by the capabilities of the first phase of the automatic compiler. Because so much of the semantics of Gist is directly captured by AP5, most of the compilation is straightforward. The remaining difficult aspects are all related to specific language constructs.

The most problematical is Gist's constraints. These constraints limit nondeterministic behavior so that the constraints are never violated. This requires indefinite look-ahead, and no effective general implementation is known. Instead, this construct has been replaced by two lower-level ones: a simple constraint which merely signals an error when its pattern is violated, and a consistency rule. Like the simple constraint, the consistency rule specifies, through a knowledge-base pattern, some relationships that are always supposed to be satisfied. However, when this pattern is violated, the consistency rule attempts to "patch" the state, via a set of repair rules that have access to both the old and the new state. If it is successful in reestablishing the relationship in the new state, then the computation continues (with the extra modifications made by the repair rule merged with those of the original transition as a single atomic step, so that the intermediate state, which violated the consistency rule, is not visible). This allows the system to automatically maintain consistency relationships expressed via these rules in an electronic-like manner. Like Prolog [22] rather than spreadsheets, these rules can involve logical as well as numeric relationships, and can be used multidirectionally.

Prototyping the Prototype

The current status of the FSD testbed is that it is fully operational and in daily use within our group at ISI. It supports the semantics of the prototyping specification language described above. However, the syntax of this language is not yet in place. Instead, we currently write in a macro and subroutine extension to AP3.

We are in the process of converting FSD to AP5. This will put the second phase of the automatic compiler in place. We will then bring up the first phase, which translates the Gist-like syntax of the prototyping specification language into AP5. Both of these phases should be operational within a few months.

We originally intended FSD to be just a testbed for software development. However, when we considered the semantic base we were installing for it, we found nothing in it that was particular to software development. Rather, this semantic base defined the way we wanted to interact with computer systems in general. We therefore decided to extend FSD from a software development environment to a general computing environment in which we could also read and compose mail, prepare papers, and maintain personal databases, because such an extension merely involved writing these additional services. Furthermore, the creation and evolution of these extra services would give us an additional insight into the strengths and weaknesses of our software development capabilities.

The FSD environment presents the user with a persistent universe of typed objects (instead of a file-based world), which can be accessed via description (i.e., associatively) and modified by generic operations. All the metadata defining the types, attributes, inheritance, viewing, and printing mechanisms are also represented as manipulable objects in this persistent universe. Consistency rules invisibly link objects to one another so that the consistency conditions are automatically maintained as the objects are modified, either directly by the user or by some service. Automation rules (demons) provide a way of transferring clerical activities to the system so that it gradually becomes a specialized personal assistant.

More and more of our tools and technology are being integrated into this method. This effort is providing us with a wealth of prototyping experience for use on later versions. Our long-term goal is to understand the rules

which govern (or should govern) software development and evolution, and to embed them within our system and paradigm.

Summary

We have incrementally elaborated the simple notion of an automatic compiler into a paradigm for automated software development. These elaborations recognize that

1) the specifications have to be acquired and validated;
2) this validation requires an operational semantics;
3) an interactive translation facility is needed to obtain the lower-level specification that can be automatically compiled;
4) the decisions employed in that interactive translation must be documented and recorded in a formal structure;
5) this formal development is the basis for a replay facility which enables implementations to be rederived from the revised specification on each maintenance cycle.

We showed how this paradigm, even without full automation, eliminated the two fundamental flaws in the current paradigm.

We also described the wide variety of research we have undertaken in support of this paradigm and characterized our successes and failures. Our chief successes are

1) Gist, its validation (via paraphrase, symbolic evaluation, and behavior explanation);
2) Paddle (formal development structure);
3) explication of the paradigm itself;
4) development of FSD (operational testbed).

Our chief failures were

1) the unreadability of Gist (partially resolved by the Gist paraphraser);
2) the inability to translate Gist (even interactively) into an automatically compilable form;
3) the fact that all our tools were laboratory prototypes, which only worked on pedagogical examples and did not interface with one another (partially resolved by FSD).

The chief remaining open research issues are

1) incremental specification;
2) replay;
3) automatic compilation leverage provided by a narrower focus (explainable expert systems) and by annotations;
4) explicating the rules that govern software development and evolution, and embedding them within an operational testbed (FSD).

Acknowledgment

The work reported here is the result, over nearly 15 years, of many colleagues who have participated with the author in this joint endeavor to understand, formalize, and automate the software process. He is grateful to all of these colleagues for making this effort so exciting, rewarding, and productive. He is particularly indebted to N. Goldman for his contributions to SAFE and the formalization of Gist, D. Wile for his development of Popart and Paddle, B. Swartout for bringing explanation into our group and exploring the use of our paradigm for developing expert systems, D. Cohen for his work on symbolic evaluation and the development of AP5, and M. Feather for developing Gist transformations and formalizing its semantics. Finally, he would like to thank all the members of his group for their help in making FSD an operational testbed in which they can explore our software paradigm and the utility of their tools and technology.

References

[1] R. Balzer, D. Cohen, M. Feather, N. Goldman, W. Swartout, and D. Wile, "Operational specification as the basis for specification validation," in *Theory and Practice of Software Technology*, Ferrari, Bolognani, and Goguen, Eds. Amsterdam, The Netherlands: North-Holland, 1983.
[2] R. Balzer, D. Dyer, M. Morgenstern, and R. Neches, "Specification-based computing environments," in *Proc. Nat. Conf. Artif. Intell. AAAI-83*, Inform. Sci. Inst., Univ. Southern Calif., Marina del Rey, 1983.
[3] R. Balzer, "A global view of automatic programming," in *Proc. Third Int. Joint Conf. Artif. Intell.*, Aug. 1983, pp. 494–499.
[4] R. Balzer and N. Goldman, "Principles of good software specification and their implications for specification languages," in *Proc. Specifications Reliable Software Conf.*, Boston, MA, Apr. 1979, pp. 58–67; see also, —, in *Proc. Nat. Comput. Conf.*, 1981.
[5] R. Balzer, "Transformational implementation: An example," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 3–14, Jan. 1981; see also, —, Inform. Sci. Inst., Univ. Southern Calif., Marina del Rey, Res. Rep. RR-79-79, May 1981.
[6] R. Balzer, C. Green, and T. Cheatham, "Software technology in the 1990's using a new paradigm," *Computer*, pp. 39–45, Nov. 1983.
[7] R. Balzer, "Automated enhancement of knowledge representations," in *Proc. Ninth Int. Joint Conf. Artif. Intell.*, Los Angeles, CA, Aug. 18–23, 1985.
[8] R. Balzer, N. Goldman, and D. Wile, "Informality in program specifications," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 94–103, Feb. 1978.
[9] Bauer, F. L., "Programming as an evolutionary process," in *Proc. Second Int. Conf. Software Eng.*, Oct. 1976, pp. 223–234.
[10] T. E. Bell and D. C. Bixler, "A flow-oriented requirements statement language," in *Proc. Symp. Comput. Software Eng., MRI Symp. Series*. Brooklyn, NY: Polytechnic Press, 1976; see also, —, TRW Software Series, TRW-SS-76-02.
[11] D. Cohen, "Symbolic execution of the Gist specification language," in *Proc. Eighth Int. Joint Conf. Artif. Intell., IJCAI-83*, 1983, pp. 17–20.
[12] D. Cohen and N. Goldman, "Efficient compilation of virtual database specifications," Jan. 1985.
[13] D. Cohen, *AP5 Manual*, Inform. Sci. Inst., Univ. Southern Calif., Marina del Rey, draft, 1985.
[14] M. Feather, "Language support for the specification and development of composite systems, draft, Jan. 1985.
[15] —, "Formal specification of a source-data maintenance system," Working Paper, Nov. 6, 1980.
[16] —, "Formal specification of a real system," Working Paper, Nov. 21, 1980.
[17] —, "Package router description and four specifications," Working Paper, Dec. 1980.
[18] N. Goldman, *AP3 Reference Manual*, Inform. Sci. Inst., Univ. Southern Calif., Marina del Rey, 1983.
[19] N. M. Goldman, "Three dimensions of design development," Inform. Sci. Inst., Univ. Southern Calif., Marina del Rey, Tech. Rep. RS-83-2, July 1983.
[20] N. Goldman and D. Wile, "Gist language description," draft, 1980.
[21] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a knowledge-based software assistant," Rome Air Develop. Cent. Tech. Rep. RADC-TR-83-195, Aug. 1983.
[22] R. Kowalski, *Logic for Problem Solving*. Amsterdam, The Netherlands: Elsevier North-Holland, 1979.
[23] P. London and M. S. Feather, "Implementing Specification Freedoms," Inform. Sci. Inst., Univ. Southern Calif., Marina del Rey, Tech. Rep. RR-81-100; see also,——, *Sci. Comput. Programming*, 1982.
[24] J. Mostow, "Why are design derivations hard to replay?" in *Machine Learning: A Guide to Current Research*, T. Mitchell, J. Carbonell, and R. Michalski, Eds. Hingham, MA: Kluwer, 1986, to be published.

[25] ——, "Automating program speedup by deciding what to cache," in *Proc. 9th Int. Joint. Conf. Artif. Intell.*, 1985.
[26] W. Swartout, "Producing explanations and justifications of expert consulting systems," Mass. Inst. Technol., Cambridge, Tech. Rep. 251, 1981.
[27] ——, "The Gist behavior explainer," in *Proc. Nat. Conf. Artif. Intell. AAAI-83*, Inform. Sci. Inst., Univ. Southern Calif., Marina del Rey, 1983; see also, ——, Inst. Inform. Sci., RS-83-3.
[28] ——, "Explainable expert systems," in *Proc. MEDCOMP-83*, Oct. 1983.
[29] ——, "XPLAIN: A system for creating and explaining expert consulting systems," *Artif. Intell.*, vol. 21, no. 3, pp. 285–325, Sept. 1983; see also, ——, Inst. Inform. Sci., Univ. Southern Calif., Marina del Rey, RS-83-4.
[30] ——, "Gist English generator," in *Proc. Nat. Conf. Artif. Intell. AAAI-82*, 1982.
[31] D. Wile, *POPART: Producer of Parsers and Related Tools. System Builders' Manual*, Inform. Sci. Inst., Univ. Southern Calif., Marina del Rey, 1981.
[32] ——, "Program development: Formal explanations of implementations," *Commun., ACM*, pp. 902–911, 1983; see also, ——, ISI RR-82-99.

Robert Balzer received the B.S., M.S., and Ph.D. degrees in electrical engineering from the Carnegie Institute of Technology, Pittsburgh, PA, in 1964, 1965, and 1966, respectively.

After several years at the Rand Corporation, he left to help form the University of Southern California Information Sciences Institute, Marina del Rey. He is currently a Professor of Computer Science and Director of the Software Sciences and Systems Division, staffed by a dozen computer science Ph.D.'s. The division combines artificial intelligence, database, and software engineering techniques to automate the software development process. Current research includes explanation systems, transformation-based development, computing environments, formal specification, symbolic execution, and constraint-based systems.

Dr. Balzer is a past Professor of SIGART (Special Interest Group on Artificial Intelligence), was Program Chairman for the First National Conference of the American Association for Artificial Intelligence, is currently Chairman of the Los Angeles Organizing Committee for the 1985 International Joint Conference on Artificial Intelligence, and is Program Chairman for the 9th International Conference on Software Engineering (whose theme will be "Formalizing and Automating the Software Process").

Automating the Transformational Development of Software

STEPHEN F. FICKAS

***Abstract*—This paper reports on efforts to extend the transformational implementation (TI) model of software development [1]. In particular, we describe a system that uses AI techniques to automate major portions of a transformational implementation. The work has focused on the formalization of the goals, strategies, selection rationale, and finally the transformations used by expert human developers. A system has been constructed that includes representations for each of these problem-solving components, as well as machinery for handling human–system interaction and problem-solving control. We will present the system and illustrate automation issues through two annotated examples.**

***Index Terms*—Knowledge-based software development, program transformation systems.**

I. Introduction

In a previous issue of this TRANSACTIONS, Balzer presented the transformational implementation (TI) model of software development [1]. Since that article appeared, several research efforts have been undertaken to make TI a more useful tool. This paper reports on one such effort.

A general model of software transformation can be summarized as follows:[1] 1) we start with a formal specification P (how we arrive at such a specification is a separate research topic), 2) an agent S applies a transformation T to P to produce a new P, 3) step 2 is repeated until a version of P is produced that meets implementation conditions (e.g., it is compilable, it is efficient). Fig. 1 presents a diagram of the model.

In the TI model, the specification language P is Gist [12], the agent S is an expert human developer, and T is taken from a catalog of correctness-preserving transformations[2] (Section II discusses other bindings of P, S, and T). Hence, the human is responsible for deciding what should be transformed next, and what transformation to use; the system checks the transformation's preconditions and applies it to produce a new state. As Balzer noted in [1], the TI model provides at least two advantages.

1) Focus is shifted away from consistency problems and towards design tradeoffs.

2) The *process* of developing a program is formalized as a set of transformations. Thus, the process itself can be viewed as an objecct of study.

Since Balzer's article appeared, we have attempted to use the TI model on several realistic problems. This work has confirmed one of the article's conjectures:

> "... it is evident that the developer has not been freed to consider design tradeoffs. Instead of a concern for maintaining consistency, the equally consuming task of directing the low level development has been imposed. While the correctness of the program is no longer an issue, keeping track of where one is in a development, and how to accomplish each step in all

Manuscript received January 2, 1985; revised June 19, 1985. This work was supported by the national Science Foundation under Grants MCS-7918792 and DCR-8312578.

The author is with the Department of Computer Science, University of Oregon, Eugene, OR 97403.

[1]A survey and more detailed discussion of transformation systems can be found in [15].

[2]Correctness rests both on Gist's formal semantics and on preconditions attached to transformations. Proofs of correctness are carried out by inspection; there has been no attempt to date to apply a more formal proof method.