# Two Application Languages in Software Production

David A. Ladd
ladd@research.att.com

J. Christopher Ramming
jcr@research.att.com

*AT&T Bell Laboratories*

## 1 Introduction

PRL5 is an application-oriented language used to maintain the integrity of databases in the AT&T 5ESS™ telecommunications switch. PRL5 is unusual in that it was explicitly designed to eliminate a number of different coding and inspection steps rather than simply to improve individual productivity. Because PRL5 replaced an earlier high-level language named PRL, which in turn replaced a combination of English and C on the same project, it is possible to trace the effect of several fundamentally different languages on this single project. The linguistic evolution has been away from languages describing computation toward a "declarative" high-level language that has been deliberately restricted to accommodate the requirements of certain analyses.

Algorithms for checking database constraints are no longer specified by human developers; instead, code is generated from static representations of the constraints themselves. These constraint descriptions can be used in more than one way, whereas a program to check constraints is useful only for performing that particular computation. In effect, PRL5 allows the re-use of project information at a high level, before it has been specialized into particular implementations. The effects of this re-use on quality, interval, and cost are tangible. A key lesson is that application-oriented languages should not be designed to describe computation, they should be designed to express useful facts from which one or more computations can be derived.

## 2 Project Background

### 2.1 5ESS Database Constraints

The AT&T 5ESS is a high-capacity, exceptionally reliable digital switching system. The 5ESS software contains millions of lines of code produced and maintained by several thousand developers. At the heart of the 5ESS software is a distributed relational database that contains information about hardware connections, software configuration, and customers. For the switch to function properly this data must conform to certain integrity constraints. Some of these are logical constraints; for example, "call waiting and call forwarding/busy should never be active on the same line." Other constraints exist to document data design choices (redundancy, functional dependencies, distribution rules) that support efficient 5ESS operation and call processing.

### 2.2 Constraint Enforcement: Data Audits and Transaction Guards

5ESS integrity constraints are used in two kinds of software: data audits and transaction guards. *Data audits* check for all data violations; they are time-consuming because the database is large and there are many constraints. Data audits are useful for discovering accidental corruption as the switch is operating and for "cleaning up" switch data at strategically important points, such as when the 5ESS software is upgraded and includes data design changes. *Transaction guards*, on the other hand, ensure that incremental changes to the database leave it in a consistent state. In principle, transaction guards could be implemented by running a complete data audit before committing a transaction. In practice, complete data audits take far too long; transaction guards must be efficient — for instance, subscribing to features like caller-ID ought to be possible without having to re-verify every constraint. Fortunately, most transactions only add, delete, or change small

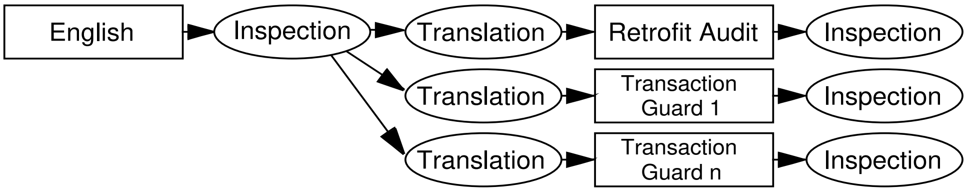


Figure 1: Traditional development with English specifications.

amounts of data and consequently have limited potential for violating constraints. If transaction guards are implemented properly and data is not modified without passing such guards, data should never become inconsistent and in theory data audits are unnecessary. However, to defend in depth against hardware and software failure, both audits are guards are used on the 5ESS.

Over the lifetime of the 5ESS, database constraints have been specified and implemented in three different ways. Initially, English was used to specify constraints; data audits and transaction guards were each written independently from these specifications. Data audits were used off-line to scrub data prior to installing a new software release; during switch operation, transaction guards were relied upon to maintain data integrity — there was no on-line data audit. In a second phase, an imperative high-level language called "PRL" was introduced. PRL is an acronym for Population Rule Language; putting data into the database is called "populating" the database, and the constraints themselves are called "population rules." PRL served simultaneously as the constraint specification language and the data audit implementation language. However, human coders continued to develop transaction guards independently using C. Now, a declarative database constraint language named PRL5 is used. Programs in the new language are executable specifications from which all constraint-related products can be generated: an off-line data audit, a new on-line data audit, a new "residual" data audit, and transaction guards.

## 3 Database Constraints in English and C

Originally, 5ESS development followed the standard software engineering practice of its day by specifying integrity constraints in natural language. But the gap between English and computer languages is large and results in difficulties that subsequent approaches strove to address.

One striking difficulty with English as a specification language is that it is ambiguous and lends itself to misinterpretation. Even seemingly straightforward requirements like those of figure 2 can be confusing.

*For billing purposes, every telephone number must be associated with an active account.*

*No telephone number can have more speed calling destinations than have been paid for.*

Figure 2: Some (contrived) English-language constraint specifications

The notion of a speed call button is familiar to many telephone users — it is a keypad digit used as an abbreviation for a longer "destination" number. Nonetheless, in the context of an implementation, the restriction on "destinations" in this constraint is unclear. Referring to the schema for relation *SpeedCalls* in table 1, it seems that this constraint could be implemented either as a limit on the number of *SpeedCalls* tuples associated with a particular number or as a limit on the number of distinct values in the "destination" fields in these tuples. Here, the choice is probably unimportant, but in a slightly different setting such ambiguities could result in costly errors.

A second pitfall of natural language specifications is that they cannot be compiled automatically and often diverge from their implementations. This problem was compounded in the 5ESS because each constraint was likely to affect several products: data audits as well as transaction guards. In the absence of an accurate mechanism to show the relationship between English specifications and C code, it is difficult to see which code needs to change when specifications are modified and vice-versa. As a result the code diverges from its specification, which in effect becomes "distributed" between the code and English text. Often the English text

| RELATION NAME | ATTRIBUTES | KEY |
|---|---|---|
| SpeedCalls | number | Yes |
| | digit | Yes |
| | destination | No |
| Accounts | account | Yes |
| | status | No |
| TelNums | number | Yes |
| | owner | No |
| | maxspdcalls | No |

Table 1: A sample relation schema

is abandoned to the role of documentation while the executable code is recognized as being more accurate. General-purpose code then becomes the sole repository of hard-won information — customer requirements, architectural decisions, and domain expertise — that is usually impractical or impossible to recover. In 5ESS, because the English constraints were implemented in several products, there was need for a "central point of truth." The English constraints therefore remained important, although over time the implementations were accorded increasing respect as specifications.

An additional problem, not entirely related to the use of English, was that no theory for developing transaction guards existed: only human intuition was available to achieve efficient results. To illustrate this problem, consider the constraint of figure 2:

*For billing purposes, every telephone number must be associated with an active account.*

Now suppose that a number is to be added or deleted. As mentioned earlier, it is possible to run a complete data audit (checking all telephone numbers for this property) before committing a transaction involving the addition or deletion of a number, but that would be too much unnecessary work. Assuming the database is correct before the transaction, a telephone number deletion should require no transaction guard at all. A telephone number addition should be guarded only by a check that the new number can be properly billed. But finding the smallest set of facts that should be checked to ensure continued data integrity is increasingly difficult as the transactions and constraints become more complex. In real life, correct results were rarely achieved (although this did not become clear until later) because specifications and code were not related by any known algorithm. Only a deeper understanding of integrity constraints and their relationship to transactions would lead to better results.

When the constraints were expressed only in English, quality problems could and did occur because of constraint ambiguity, "distributed" specifications, and the lack of an algorithm for generating transaction guards. Development took a long time, since the translation from English to various constraint implementations had to be done by human developers instead of a compiler (see figure 1). Costs were correspondingly high because human developers are expensive.

## 4 Imperative PRL

To eliminate the shortcomings of English as a specification language and thereby to improve interval, cost, and quality, a language called "PRL" [1] was developed. It addressed some but not all of the problems

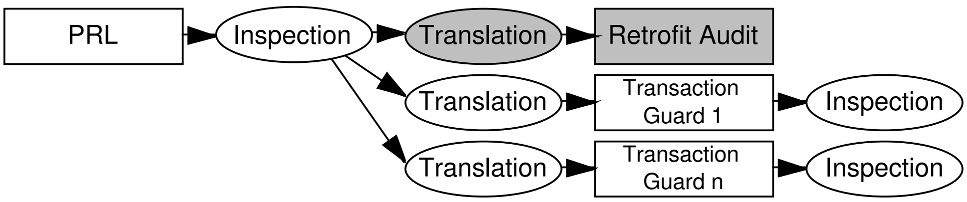


Figure 3: Software development with imperative PRL. Shaded portions are automated.

noted above. Experience with PRL is interesting because it reveals that general algorithmic languages, though powerful, have a limited ability to effect fundamental change in software development; by contrast, a restricted declarative language developed later proved more far-reaching in its consequences.

This original PRL was specially designed for implementing 5ESS data audits. It was an Algol-family language with variables, control flow, and functions and the "right" abstractions for the application domain. For instance, certain statements specified constraints that would raise exceptions when violated — such as "Accounts.status must_equal ACTIVE." It also contained control abstractions such as a "for every" statement that would iterate over tuples in a relation; the body of a "for every" statement typically contained assertions about the current tuple. This control abstraction eliminated the need for programmers to concern themselves with low-level database storage and access details. A "find" statement could perform relation searches for interesting tuples; a "must find exactly one" statement nested in a "for every" worked much like the join of relational algebra. PRL had a built-in knowledge of 5ESS data types — binary coded decimals, various kinds of enumerations and integers, strings, and the tuple types. The PRL compiler also made good use of knowledge about data distribution (over the 5ESS processors), and indices into 5ESS relations, reducing some burden on the programmers.

Figure 4 displays PRL implementations of the English-language constraints introduced in figure 2. Because they use appropriate abstractions and require little extraneous verbiage, imperative PRL programs

```
for every tuple in TelNums
begin
   must find exactly 1 tuple in Accounts
   where(TelNums.owner equals Accounts.account)
   when found
   begin
      Accounts.status must_equal ACTIVE;
  end
end

for every tuple in TelNums
begin
   int cnt;
   cnt = 0;
   for every tuple in SpeedCalls
   where(SpeedCalls.number equals TelNums.number)
   begin
      cnt = cnt + 1;
   end
   TelNums.maxspdcalls must_be_greater_than_or_equal_to cnt;
end
```

Figure 4: An imperative PRL specification

were easier for humans to read than programs in low-level languages such as C. This fact, coupled with the formality and precision of imperative PRL relative to English, caused one organization within the 5ESS to abandon English-language specification in favor of PRL programs that would serve simultaneously as code and specification. Of the two constraints implemented in the example PRL fragment, the first serves considerably better as a specification than the second, which is harder to analyze because it involves both state and control flow. PRL represents a compromise between English and C and was not an optimal specification language. The advantage of PRL, which must be measured against the cost of developing the language and tools, is that its programs can be compiled to produce data audits, eliminating a round of coding and inspection (see figure 3). For the data audit, this automatic code generation reduced interval and improved quality by eliminating errors of coordination and interpretation (although compiler bugs could still introduce faults). In addition, programmer productivity was boosted by the availability of primitive constructs at an appropriate level of abstraction.

In spite of its improvements, PRL suffered the plight of all general-purpose algorithmic languages: programs in such languages are opaque artifacts suitable only for interpretation on a particular machine. Interesting questions about programs in Turing-complete languages are often undecidable. In particular, the analysis needed to produce transaction guards is undecidable for Turing-complete constraint languages (like PRL) unless the transaction language is trivial. In essence, the constraints that were expressed in PRL could not be recovered for use in transaction guards. One implication is that the effect of general-purpose languages on large software development is limited because they can rarely be analyzed. Programs in such languages are a poor source of interesting information; in the case of PRL, this prevented the automatic generation of more than one product.

## 5 Declarative PRL5

Application-oriented computer languages need not be algorithmic; instead, they can describe facts relevant to a particular application. Given an appropriate algorithm, these facts can be used to generate the needed code. As an example, a yacc [3] program describes a formal language, but that description can be used to generate a parser as well as a diagram of the grammar and information about whether the grammar is ambiguous or whether certain productions are un-reducible. The grammar can also be easily transformed into a normal form. Even though yacc parsers can be augmented with reduction-time actions expressed in the host language, the grammar itself is often easily separated for use in other products. Just as the restricted declarative nature of the yacc language allows many different analyses, replacing imperative PRL with a restricted declarative language made it possible to generate data audits and transaction guards from a single source. This shift changed the development process, reduced costs, improved quality, and resulted in specifications that are a good source of information even for unexpected applications.

The missing piece of the puzzle was a procedure for deriving transactions from specifications. This was found in the work of Xiaolei Qian [4], who presented an algorithm (subsequently refined at AT&T[2]) for "differentiating" constraints to arrive at efficient transaction guards. The idea behind the algorithm is to find the weakest precondition of a given transaction with respect to the database constraints. By exploiting the assumption that the database is consistent (e.g., conforms to all constraints) before the transaction is executed, this weakest precondition can be factored into a form which in practice is less complex than the full weakest precondition. Then, at run-time, a transaction can be aborted if it fails to pass this simplified weakest precondition, thereby preventing inconsistencies from creeping into the database. The algorithm assumed a limited transaction language and also a constraint language based on first-order logic.

The new high-level language, dubbed "PRL5," was constructed on the strength of the "differentiation" algorithm and designed to conform to the requirements of that algorithm. PRL5 is based on first-order logic and does not describe computation directly — the notions of program counter, variables, and assignment are missing from the core language. But many of the abstractions useful in 5ESS constraint enforcement were retained by PRL5 language in some form — for instance, the "for every" and "find" control constructs were transformed into similar universal and existential quantification statements.

Figure 6 shows PRL5 implementations of the example constraints. Note that PRL5 programs describe constraints rather than an algorithm for checking whether constraints hold. Interestingly, the first constraint

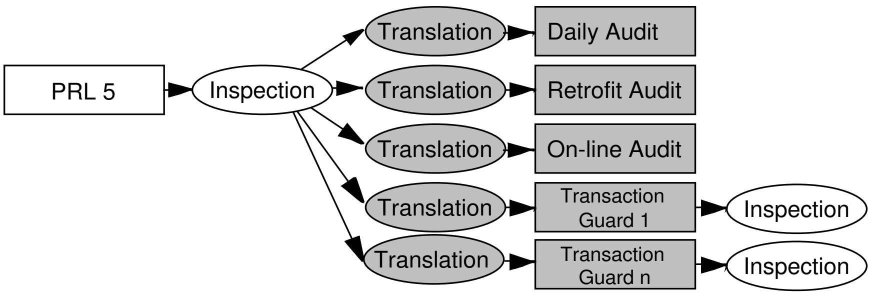


Figure 5: Development with declarative PRL5. Shaded portions are automated.

```
for number in TelNums
begin
  find >= 1 account in Accounts
  where(number.owner equals account.account)
  when_found
  begin
    account.status == ACTIVE;
  end
end

for number in TelNums
begin
  number.maxspdcalls >= count { s in SpeedCalls
                                where (s.number==number.number) };
end
```

Figure 6: A declarative PRL5 specification

looks similar to its imperative PRL equivalent. The important distinction is that the PRL5 "for" statement is defined in terms of universal quantification whereas the imperative PRL "for every" statement is really a looping control flow construct and can include "break," "continue," or other imperative statements such as variable assignment. The second constraint in figure 6, which formerly made use of iteration and stored information in a variable, has been rendered with a "count" expression. The subtle but important difference between this built-in aggregate operator and the invocation of a function call named "count" which simply hides imperative code is that PRL5 aggregates have been carefully considered for their ramifications on the "differentiation" algorithm (one of the important extensions of [2] accounts for such aggregates). Statements in PRL5 are limited to those which are "differentiable," whereas those in imperative PRL are not; PRL programs could check data constraints that cannot be expressed at all in PRL5. Situations that call for more complex constraints must be addressed by changing the assumptions of the system or reorganizing the data to arrive at a more tractable constraint.

```
find ==1 number in TelNums
where (number.number==new_tuple.num)
when_found
begin
   number.maxspdcalls>=1+count {s in SpeedCalls
                              where (new_tuple.num==s.num) };
end
```

Figure 7: Transaction guard for inserting a tuple into relation SpeedCalls

As an example of differentiation, consider the constraints of figure 6 and the simple transactions which either delete or add a tuple to relation SpeedCalls (there is a special transaction language offering inserts, deletes, updates, conditionals, and some iteration; realistic transactions and constraints are usually more complex than those in this example). Deleting from the relation SpeedCalls cannot possibly violate any of the constraints in figure 6. Adding a SpeedCalls tuple involves the check of figure 7.

Note that the variable `new_tuple` is not bound in this check and is a value that must be provided at run-time. An inspection of this transaction guard reveals that an expensive recount is required each time a SpeedCalls tuple is added. A better data design would store the number of tuples in a new attribute and one would expect that differentiation would then yield better results. Such a counter could be added to the TelNums relation as shown in Table 2. A constraint relating numspdcalls to the tuples in relation SpeedCalls

| RELATION NAME | ATTRIBUTES | KEY |
|---|---|---|
| TelNums | number | Yes |
| | owner | No |
| | maxspdcalls | No |
| | numspdcalls | No |

Table 2: A modified "TelNums" relation

owned by the number would need to be added; a check limiting the number of SpeedCalls can then refer to a stored value rather than a computed one. The revised check is shown in figure 8. An auxiliary constraint requires each tuple in SpeedCalls to be related to an existing TelNums tuple.

```
for number in TelNums
begin
   number.numspdcalls<=number.maxspdcalls;
   number.numspdcalls==count { s in SpeedCalls
                            where (s.num==number.number) };
end
for speedcall in SpeedCalls
begin
   find ==1 number in TelNums where (number.num == speedcall.num);
end
```

Figure 8: Constraints revised to reflect the new data design

With these revised constraints, the differentiator will no longer allow a transaction to delete or add to relation SpeedCalls alone: it is necessary to update the TelNums relation simultaneously. Because the differentiator "understands" aggregates such as count, neither adding nor deleting tuples from SpeedCalls will require iteration over the SpeedCalls relation; the transaction guard is merely that of figure 9. This analysis lies at the heart of the PRL5 language, which was designed precisely to accommodate the algorithm that performs it.

```
find ==1 number in TelNums
where (number.number == new_tuple.number)
when_found
begin
   number.numspdcalls+1<=number.maxspdcalls;
end
```

Figure 9: A more efficient transaction guard

## 5.1 The Applications of PRL5

Like yacc code, which has several potential applications, PRL5 code can be (and is) employed in several different tasks. Most importantly, PRL5 is compiled to produce code for several products. Two kinds of data audit — one for use on-switch and one for use off-line — are produced from PRL5 specifications. Numerous transaction guards have been derived, which is particularly significant since they are for the first time being developed in a provably sound fashion. A hybrid partial audit called a "residual check" or "daily audit" is also derived from PRL5 code to double-check the embedded base of transactions developed with the old methodology at the end of each day; this "residual check" is new and has been made possible by the versatility of declarative PRL5 specifications.

In addition, because PRL5 is declarative, it is also the source of some lesser but nonetheless useful analyses that would not have been possible if PRL5 had been a general-purpose language. In particular, PRL5 offers a view of the data design that augments a basic database description enumerating relations, fields, keys, and indices. The relationships between data items can be understood by examining PRL5 constraints for such information as foreign key constraints and functional dependencies; software visualizations revealing this information guide future data design and feature implementation.

PRL5 can also be "optimized" using techniques often unavailable or of limited value when applied to imperative code. Redundant code elimination, logical and semantic transformations, and simplifications are all good optimization candidates. For instance, the absence of state makes "loop jamming" effective for combining separate constraints quantified over the same relation; this results in an implementation that traverses a relation just once instead of many times. Another useful optimization involves the automatic selection of a good lookup method given index information, knowledge about keys, and the ways in which keys are populated. One compelling example involves lookups where keys are partially specified. Depending on certain factors, it may be more efficient to probe for each possible value with a keyed lookup rather than perform a linear search. Which method to use is never specified within PRL5 code; rather, the choice is made by a compiler. If the factors affecting the decision change, an improvement can be effected without relatively risky modifications to the PRL5 constraints themselves.

Another interesting aspect of PRL5 "code" is that terse error descriptions can be extracted at compile-time and presented when constraints are violated. In a general-purpose algorithmic language, a constraint violation can be identified crudely at best, for instance by giving the line number of the failed assertion. But in PRL5, it is possible to exhibit precisely the code that is necessary and sufficient to describe the violated constraint, making the error easier to understand and to fix. These error constraints, being valid PRL5 programs in their own right, were formerly "verbosified" by a program which rendered the constraint in English for the benefit of those unfamiliar with PRL5 syntax; however, lately PRL5 code itself has proven adequate for most descriptive purposes without this additional transformation.

## 5.2 PRL5 Impact

The impact of PRL5 on software development has been to eliminate certain major coding and inspection steps while at the same time offering an increased number of products. These improvements are reflected in figure 5. More constraint-related products are now offered, as compared with the situations detailed in figures 3 and 1; other things being equal, this is a form of productivity improvement. The elimination of coding and inspections affects cost and interval. The single source of constraint information and the technology for automatically compiling these constraints both result in quality improvements.

Although PRL5 was intended to eliminate several development steps, transaction guards continue to be inspected by people even though they are automatically generated (accordingly, this step is drawn with white nodes in figure 5). These inspections prove necessary because differentiation results sometimes surprise developers by being "too inefficient" or, occasionally, outrightly impossible to satisfy. These bad results indicate previously unnoticed problems with the data design, the transaction, or the constraints themselves. Differentiation therefore acts as a useful "sanity check" as well as a code generation step; its results must be heeded. For instance, the transaction guard of figure 7 prompted a small change to the data design that resulted in the better guard of figure 9. Furthermore, if (given the new design) a transaction attempted to add or delete a SpeedCalls tuple without updating "numspeedcalls" in the appropriate TelNums tuple, this mistake would have been revealed by inspecting the transaction guard output. The curious result is that an intended improvement in development interval and cost resulted instead in a quality improvement.

The declarative form has had a more profound effect on development than any imperative language could, because PRL5 is a more useful repository of information. More products can be generated automatically, and time-consuming coding steps as well as inspections have been eliminated. Quality is higher, partially because problems of coordination have been eliminated and partially because the differentiation algorithm provides useful new information.

## 6 Conclusion

The 5ESS database constraint components have evolved significantly since the days when English was considered a suitable specification language. The second and third generations of this projects show two "very high level" languages in stark contrast. The first, being Turing-complete, was effective for describing algorithms to enforce constraints but not the constraints themselves; its use therefore was limited to straightforward execution on a particular machine. The new language, PRL5, is geared toward the description of constraints, and its programs can be used to derive computation for enforcing constraints automatically in a variety of ways. This suggests that although computation is the natural purpose of any computer language, languages that describe computation directly are not always the most useful ones because they tend to be impossible to analyze — they hide information rather than expose it.

The application language approach is to design a domain-specific language that can be analyzed to yield executable code as well as secondary applications. The advantage is in explicit representation of domain information that would otherwise be implicit and inaccessible; this approach reduces the need for alternative documentation in languages such as English, and leaves open the possibility that unforeseen uses of the information can be found. The drawback, of course, is that designing, implementing, and introducing new languages is difficult. Although PRL5 eliminated a number of tasks previously executed by humans, some of these improvements were offset by shifts in resource allocation due to the integration of this new language. Nonetheless, as of this writing, PRL5 successes include the deployment of new on-switch data audits, the automatic generation of transaction guards, and quality improvements that are having a positive impact on the 5ESS.

## Acknowledgements

The work described in this paper was the result of a collaboration between several organizations. Many individuals made significant contributions to the project. Rather than running the risk of omission, we will refrain from listing them all here. A smaller and more manageable number of people offered valuable insights and advice leading to this paper. For this we thank gratefully thank Jackie Antis, Michael Benedikt, Jon Bentley, Tim Griffin, John Meale, John Pyrce, Eric Sumner Jr., and Dave Weiss.

## References

[1] B. N. Desai, D. L. Harris, and R. A. McKee. A formal language for writing data base integrity constraints. In *International Switching Symposium*, 1992.

[2] T. G. Griffin and H. Trickey. Integrity maintenance in a telecommunications switch. *IEEE Data Engineering Bulletin*, June 1994.

[3] S. C. Johnson. Yacc: Yet another compiler compiler. Technical report, Bell Telephone Laboratories, 1975.

[4] X. Qian. *The Deductive Synthesis of Database Transactions*. PhD thesis, Stanford University, 1989.

Chris Ramming received degrees in Computer Science from Yale College (BA '85) and the University of North Carolina at Chapel Hill (MS '89). He joined AT&T Bell Laboratories in 1987 and is a Member of Technical Staff in the Software Production Research department. His current interests include application languages and their use in software production.

David Ladd received the BS and MS degrees in Computer Science from the University of Illinois at Urbana-Champaign in 1987 and 1989. He joined AT&T in 1989, where he is currently a Member of Technical Staff in the Software Production Research Department. His current research interests are software re-engineering and application-oriented languages and environments.