Program Specification and Testing
Within an Integrated Programming Environment

Robert Cartwright

Rice MASC TR83-27.

Computer Science Program
Department of Mathematical Sciences
Rice University
Houston, Texas 77251
# Table of Contents

1. Project Summary .................................................................................................................. 1
2. Project Description ................................................................................................................ 2
   1. Background and Motivation ............................................................................................... 2
2. Overview of the Research Project ......................................................................................... 3
3. Description of Subprojects .................................................................................................. 4
   1. Mathematical Foundations of Program Specification ....................................................... 4
   2. Testing Methodology ......................................................................................................... 11
   3. A Program Specification and Testing System ................................................................. 15
3. Research Facilities ............................................................................................................... 21
4. References ............................................................................................................................ 22
5. Appendix: Overview of \( \mathbb{R}^n \) ....................................................................................... 27
1. Project Summary

Despite the growing importance of program specifications in programming methodology, there are virtually no practical software tools that exploit them. In fact, most programming environments do not accommodate program specifications at all. Programming environment research has largely neglected program specifications in favor of issues such syntax directed editors, source level debuggers, and program management systems.

In contrast, we believe that formal specifications should play a central role in the programming process and that programming environments should actively assist the programmer in this regard. In the presence of formal specifications, a much more systematic approach to software development and validation is possible. Fortunately, most of the technology required to construct such an environment already exists. Fifteen years of vigorous research in program semantics and verification have laid most of the necessary theoretical groundwork. This primary goal of this research project is to adapt and extend that work to form a comprehensive practical discipline of program specification and testing.

Our research program breaks down into three parts. First, we are exploring the mathematical foundations of program specifications and applying the understanding that we gain to subject of specification language design. The success of a specification and testing system hinges on the quality of the underlying specification language. We are particularly concerned with the special problems posed by floating point arithmetic and concurrency. Second, we are developing more sophisticated methods—employing a judicious combination of verification and testing—for certifying that programs implement their formal specifications. Finally, to test the effectiveness of ideas, we are building an experimental program specification and testing system as an extension of the \( R^2 \) Programming Environment already under development at Rice.
2. Project Description

2.1. Background and Motivation

Despite enormous strides in programming methodology, program testing is still a black art. Finding errors in programs is so difficult that many experts in programming methodology stress error prevention and ignore the issue of error detection entirely. Although this point of view may have pedagogic value, it is unrealistic. Since programmers are fallible human beings, they will inevitably make mistakes. For this reason, we believe that a comprehensive discipline of programming should include systematic methods for testing programs as well as writing them. Moreover, since program testing is a laborious clerical procedure (like compilation), these methods should be almost entirely automatic.

The biggest obstacle to developing automatic methods for validating programs is the absence of precise specifications. Conventional documentation written in informal prose is typically incomplete and ambiguous. In the context of informal documentation, program testing is an amorphous activity that cannot be automated (except as a questionable exercise in artificial intelligence). Consequently, the development of systematic methods for validating programs hinges on the feasibility of documenting programs with formal specifications.

Although writing intelligible formal specifications is a non-trivial endeavor, we believe that, given a sufficiently expressive specification language, it is much less time consuming than conventional testing which reportedly [Beiz 83] consumes half of the resources involved in software development. In current programming practice, the primary impediment to writing formal specifications is the absence of satisfactory formal specification languages and tools (such as syntax-directed editors and interpreters) that support them. With the exception of a few limited, experimental systems (such as SIMPL [Brand 76], HDM [Levi 79], FDM/ina Jo [Loca 80], AFFIRM [Gerh 79], and PRL [Bate 83]), programming environment research has neglected program specifications in favor of issues such syntax directed editors, source level debuggers, and program management systems. In contrast, we believe that formal specifications should play a central role in the programming process and that programming environments should actively assist the programmer in this regard.

At Rice, the Laboratory for Computer Science and Engineering has recently embarked on a five year project dubbed R² to build a reactive programming environment specifically designed to support numerical (scientific) computation. A program specification and testing system based on very high level executable specifications is an integral part of the planned environment. The laboratory
staff and equipment for are partially supported by NSF CER grant MCS 82-21884. This report describes the the project specification and testing component of $R^n$ and the theoretical investigations underlying it. An overview of the $R^n$ project appears in Appendix I.

2.2. Overview of the Research Project

The $R^n$ Specification and Testing Project has three distinct components.

First, we are studying the mathematical foundations of specification languages and logics with an emphasis on specification language design and the special problems posed by floating point arithmetic and concurrency. For program specifications to be useful, they obviously must be clear, succinct, and logically meaningful—qualities that critically depend on the expressiveness and elegance of the specification language. In addition, program specifications should be written in a form that facilitates program validation (testing the consistency of the program with its specifications on test data)—a property that depends on the the semantic structure of the specification language. We have designed a specification language TTL (Tentatively Typed LISP) that we believe embodies these qualities. As our understanding of specifications grows, we will continue to extend and refine it.

Second, we are developing more sophisticated methods—employing a judicious combination of verification and testing—for certifying that programs implement formal specifications. Although the broad outlines of a satisfactory theory of program specification and testing are beginning to emerge [Cart 81], there are many important issues that are still unresolved. In particular, programs involving either concurrency or extensive floating point arithmetic lie outside the scope of established specification and testing methods.

Finally, we are building an experimental language independent program testing system and incorporate it as an integral component the $R^n$ Programming Environment (under the supervision of Professors Ken Kennedy and Robert Hood). Obviously, the strengths and weaknesses of various approaches to program specification and validation cannot be accurately determined without constructing experimental implementations that programmers can use. We believe that the value of a programming environment will be significantly enhanced by incorporating program specifications and test data in the program data base managed by the environment. If specifications are available, a sophisticated programming environment can maintain the semantic consistency of the program data base—detecting anomalies between the interface actually supported by a program module and the interface required by the programs that import it.
Similarly, it can ensure that all installed programs are "correct"\(^1\) by confirming that a program is consistent with its specifications as part of the installation procedure.

2.3. Description of Subprojects

2.3.1. Mathematical Foundations of Program Specification

Our theoretical investigations are currently focussed on extending and refining of the specification language TTL [Cart 80, 81, 83] and the mathematical theory of data domains and types underlying it. In our view, specification languages are extremely high level programming languages. They are supposed to describe the behavior of a program at the highest possible level of abstraction—omitting all unnecessary implementation details.

In the context of conventional procedural programming languages, specifications are typically formulated as logical formulas (called assertions) that are attached to various points in the program. Each assertion specifies an invariant on the program state that always holds at the corresponding point in the program. Although it is possible in principle to annotate programs with assertions written in a closed specification language containing a small fixed set of operations (functions and predicates), specifications expressed in this form are extremely difficult to understand (just like machine language programs). To be intelligible, specifications must describe the intuitive abstractions that a program implements. Consequently, all practical specification languages are open; they include mechanisms that enable the programmer to define the abstract data domains and operations that are appropriate for his particular application.

In fact, these mechanisms form the core of a specification language. The assertions annotating a program are simply formulas over a collection of operations defined by a specification "program". The "program" is typically is much more important and interesting than the assertions themselves. For example, in the specification of a compiler, the specification "program" must contain definitions of the semantics of the source and target languages. The annotation inside the compiler merely relates the meaning of the fragment of target code generated to the meaning of the fragment of source text processed.

In our experience, it is possible to restrict a specification language to computable (in the sense of Scott [Scot 76,81]) operations without seriously limiting the expressiveness of the language. In this case, a specification "program"  

\(^1\)In the sense of correctly processing all the inputs in an extensive, representative set of test cases.
literally is a program written in an extremely high level programming language. In the context of program testing, the advantages of adhering to this discipline are enormous, because it enables us to evaluate assertions on test data and check the conformance of a program execution with the program's specifications.

In accordance with this philosophy, we have designed the specification language TTL, a very high level applicative programming language based on the universal domain CONSO developed by Cartwright and Demers [Cart 83]. CONSO is a universal domain consisting of lazily evaluated binary trees (LISP S-expressions) over a countable set of atoms At. It includes the universal domains Pω [Scot 76] and Tω [Plot 78] as trivial subspaces. From the standpoint of program testing, CONSO has a critical advantage over other universal domains (including Pω and Tω): CONSO includes finite total elements corresponding to ordinary "first-order" data objects such as characters, natural numbers, strings, and finite arrays. In informal terms, finite total elements are the answers generated by terminating computations. TTL is a very high level specification language for identifying elements of CONSO analogous to the LAMBDADA languages for Pω and Tω. The critical difference between TTL and the LAMBDADA languages is that is possible to construct an extensionally faithful\(^2\) interpreter for TTL. The existence of such an interpreter is critical feature of TTL since we intend to test programs by evaluating assertions written in TTL on actual test data.

Since CONSO includes finite total elements, it is possible to reduce an arbitrary TTL expression denoting a finite total element to an extensional canonical form. The requisite interpreter is very similar to an "eager-beaver" interpreter for a lazy dialect of LISP; it incrementally outputs a description of the denoted a tree—terminating if the tree is finite and total (all leaves are convergent). In contrast, an interpreter for LAMBDADA (either the Pω or Tω formulation) must either behave intensionally (as in normal order reduction of the untyped lambda calculus) or run forever on every input expression because there is the possibility that additional information about the denoted element must be generated.

An easy way to understand the difference between CONSO and ordinary universal domains, is to identify the isomorphic images of the domains Pω and Tω inside CONSO. Pω and Tω are isomorphic to the subdomains Pω and Tω defined by the equations

\(^2\)For an interpreter to be extensionally faithful, it must behave identically for all expressions that denote the same abstract object. If the expressions α and β are equal, an extensionally faithful interpreter cannot terminate for α and diverge for β even if produces the same set of outputs in both cases.
\[ P_\omega = \text{Triv} \times P_\omega \]
\[ T_\omega = \text{Bool} \times T_\omega \]

where Triv denotes the flat domain containing the elements \{⊥, true\}, Bool denotes the flat domain containing the elements \{⊥, true, false\}, and \( \times \) denotes the Cartesian product construction (where \( (\bot, \bot) \) is the least element of the Cartesian universe). Every tree in \( P_\omega \) and \( T_\omega \) contains at least one leaf that is \( \bot \). Hence, any attempt to evaluate (in an extensionally faithful fashion) an expression denoting an element of \( P_\omega \) or \( T_\omega \) will diverge.

Although TTL is a spiritual descendant of Pure LISP, it is a much more expressive and flexible specification language, because it includes a sophisticated data type definition facility and much more powerful primitive operations. In TTL, the programmer can define any algebraic structure with the following two properties:

(i) The universe (carrier) is an effectively-presented finitary complete partial order.\(^3\)

(ii) Every operation is computable (in the sense described by Scott [Scot 76, 81]).

This class of algebras is much richer than the class definable by algebraic specification (recursively enumerable sets of equations) since neither the equality relation nor the inequality relation on variable-free expressions is forced to be recursively enumerable [ADJ 80] [Kami 83].\(^4\) Yet the specifications are constructive in a very strong sense: the specified domain is guaranteed to exist, and every element is the least upper bound of a set of finitely generated elements (using the primitive domain elements). Moreover, if the domain does not include any infinite elements—a characteristic of many domains encountered in practice—then every element of the domain is finitely generated.

A concrete illustration of the expressiveness of TTL appears in the context of specifying floating point programs. As we discuss in detail in Section 2.3.3.1, an elegant way to specify many floating point programs is to relate the computed approximate answer to the exact answer expressed in terms of operations on the representable real numbers. Fortunately, within TTL, it is easy to define the domain consisting the representable real numbers; they are simply floating point numbers with unbounded exponents and infinite (lazily evaluated) fractions. In contrast, the specification of the representable reals is not possible in more

\(^3\)A complete partial order \( D \) is finitary if and only if the set of finite elements forms a basis for \( D \). See [Scot 81].

\(^4\)In the two formulations of algebraic specification—initial semantics [ADJ 80] or final semantics [Kami 83], either the equality relation (initial semantics) or the inequality relation (final semantics) is recursively enumerable.
restrictive frameworks such as algebraic specification, since neither the equality or inequality relation on representable reals is recursively enumerable (the tail of a fraction may diverge).

Although the design of the core of TTL is essentially complete, there are four important language extensions that we are investigating. First, we are seriously considering adding a finite top element (T)—denoting inconsistency—to CONSω. In CONSω without T, it is difficult to embed the domain of continuous functions over CONSω within CONSω; the most elegant solution is a generalization of the ingenious embedding developed by Plotkin for Tω [Plot 78]. In contrast, an extremely simple embedding very similar to the standard embedding used in Pω [Stoy 76] is possible if we add T to the domain and the corresponding binary operation lub (least upper bound) to TTL. The two obvious disadvantages of this approach are:

(i) The definitions of several important technical terms such as total element become more complex, because T must be treated as a special case.

(ii) The addition of T to the domain forces us to accept a slightly weaker form of extensionally faithful interpretation which we call weak interpretation. In the weak interpretation of an expression denoting T, it is perfectly acceptable for the interpreter to generate a finite total element (which technically approximates T) and halt. Without this revision in our notion of extensionally faithful interpretation, the TTL interpreter is forced to run forever on all expressions that do not denote T (just like interpreters for the LAMBDA languages) because there is a possibility that subsequent evaluation will produce T.

At the moment, we are strongly leaning in the direction of including T in CONSω, since the preceding disadvantages seem relatively minor in comparison to the advantage of much simpler embeddings for function spaces. Moreover, the addition of T in the domain has two other desirable consequences. First, the lub operation is a significant extension (from the perspective of programming convenience) to TTL. A surprising property of lub is that its implementation in a TTL interpreter is a straightforward generalization of the amb (ambiguous) operation invented over twenty years ago by McCarthy [McCa 63], but largely ignored because it is not referentially transparent. In the context of TTL, the notion of weak interpretation allows us to implement the referentially transparent lub operation in a way that is not necessarily referentially transparent when the meaning of the application is T (since the interpreter can return any finite total

---

5McCarthy's amb operation did not accommodate higher order objects; on ordinary "first order"
element). In TTL, McCarthy's \textit{amb} operation has an elegant mathematical characterization that is purely denotational.

The second desirable consequence of adding $\top$ to $\text{CONS}^\omega$ is that it provides a convenient framework for formalizing static type-checking within the Domain Theory [Scot 76,81] and for developing MacQueen and Sethi's [MacQ 82] conception of types as ideals. In $\text{CONS}^\omega \cup \{ \top \}$, an expression is type consistent if and only if for all possible bindings of its free variables (compatible with their type declarations), it does not denote $\top$. Since a weak interpreter for TTL (with $\top$) may behave unpredictably on any expression denoting $\top$, it is imperative that we develop type-checking rules that guarantee that all well-typed TTL expressions are consistent.

In Sethi and MacQueen's conception of type, a type is the kernel (set of points mapped to $\bot$) of a strict, continuous function.\footnote{In very recent, unpublished joint work with Plotkin, they have proposed relaxing the continuity constraint to accommodate function spaces in greater generality [MacQ 83].} In $\text{CONS}^\omega \cup \{ \top \}$, it is easy to show that every such type is also the image of a particularly well-behaved finitary retraction that maps all elements outside its image into $\top$. Hence, in this framework, Sethi and MacQueen's types are a proper subset of the finitary retracts (Scott's notion of type [Scot 76]) of the universal domain.

The second major extension to TTL that we are investigating is the addition of a sophisticated type checking and inference system. Although TTL has a very powerful data type definition facility, the existing static and dynamic type checking mechanisms are very weak. Consequently we want to adapt and extend applicable ideas from the type checking schemes developed for Russell [Deme 80], the polymorphic typed lambda calculus [Mili 78], and Smalltalk [Suzu 82].

The third and most extensive addition to TTL that we are studying is the incorporation of constructs for describing concurrency. At the moment, TTL does not include any provisions for describing concurrent computations. Although it is possible to document the \textit{safety}\footnote{A safety property has the form: for all states $s$ encountered during program execution, the predicate $\pi(s)$ must hold ("bad things can't happen").} (partial correctness) properties of a concurrent program by using a sequential assertion language and auxiliary variables [0wic 76], we want to extend TTL so that it can describe the \textit{liveness}\footnote{A liveness property has the form: a state $s$ such that the predicate $\phi(s)$ holds will be encountered during program execution ("good things must happen").} [0wic 82] properties as well. For the sake of simplicity, we want to avoid introducing an explicit notion of time. Consequently, we favor an implicit approach to time such objects the implementation of \textit{lab} is identical to \textit{amb}.\footnote{A liveness property has the form: a state $s$ such that the predicate $\phi(s)$ holds will be encountered during program execution ("good things must happen").}
as that employed in "temporal" logics [Pnue 79] [Lamp 80] where a computation is formalized as a sequence of states (appearing in chronological order). Before we adopt a specific design, however, we want to deepen our understanding of the semantics of very high level "applicative" languages accommodating concurrency such as data flow languages [McGr 82] and the mu calculus [Ward 80], a generalization of the lambda calculus accommodating concurrency.

The final addition to TTL that we are exploring is a major expansion the standard library of TTL type and function definitions. Since TTL has a very rich data type definition facility, almost none of the types and operations that are built into other specification and programming languages are primitives in TTL (the LAMBDA languages and, to a lesser degree, Algol 68 share this property). For this reason, TTL includes a standard library (analogous to the "standard prelude" in Algol 68) that is automatically included in every TTL specification.

In fact, two of the most interesting features of TTL, the definition of sequences and "multi-ary" function application (which is built on top of the primitive notion of curried function application), are actually part of the standard library rather than the language core. Although TTL is a spiritual descendant of LISP, it strongly breaks with LISP tradition in its treatment of both of these issues.

In LISP, the existence of two separate notions of sequence—S-expressions and lists—with overlapping representations is a frequent source of confusion. Similarly, the presence of three different forms of function application (corresponding to EXPR, LEXPR, and FEXPR "functions") forces the programmer to think at a much lower level of abstraction than mathematical functions.

In contrast, TTL (more precisely the TTL standard library) relies on a single comprehensive notion of sequence and function application, which are closely intertwined. Every function (other than a single binary operation for building sequences) takes a single evaluated argument that is a sequence. In an application, the sequence argument α is pattern matched against a formal parameter list, binding the parameters to elements of α. The individual elements of the sequence argument correspond to the "arguments" in a conventional multi-ary application. If the notions of sequence and parameter list are properly formulated, this form of application is very flexible and powerful, yet referentially transparent (unlike LISP FEXPR's). Moreover, with a judicious choice of notation, the syntax of ordinary function definitions and applications is identical to that in a conventional multi-ary formulation of functions.
As an illustration of the power of this approach, consider the problem of accommodating call-by-name argument evaluation in function applications. If the definition of sequences includes the proper form of lazy sequence (a sequence containing elements that may be divergent) [Cart 82], then the suspended elements in a sequence argument are passed by name (assuming the binding process does not force evaluation) while the evaluated ones are passed by value. Similarly, a function can take a variable number of "arguments" simply by matching a single formal parameter against the actual sequence argument and accepting responsibility for breaking it down into "arguments". The same framework also supports the composition of functions that produce sequence results with functions that take multiple "arguments" (including those that are non-strict or take a variable number of arguments)—an operation that is very awkward in LISP.  

The elegance and utility of this definition of function application critically depend on how sequences are defined. We have already observed that the existence of lazy sequences provides the capability to pass "arguments" (sequence elements) by name instead of by value. But the question of how to define sequences is more subtle than simply choosing whether or not to support lazy evaluation.

To avoid the confusion between atomic objects and singleton sequences, TTL defines sequences such that every data object is a sequence, eliminating the possibility of forming data objects that are not sequences. As a consequence, every primitive operation on sequences—e.g., cons, head (car), tail (cdr)—is total for the entire program data domain (including atoms), yet there are no anomalies in the behavior of any operation.

The key idea underlying the TTL formulation of sequences is that an atom (a primitive element of a sequence) should always be interpreted as singleton sequence consisting of that object. In the context of character strings, this notion is very natural: on an abstract level, the string "A" can be identified with exactly the same object as the character 'A. The only difference between the set of strings over an atom set At and a satisfactory formulation of sequences over At is that arbitrary sequences (rather than just atoms) may appear as elements of sequences. However, it is easy to generalize the notion of string to eliminate this difference: all that is required is an embed operation (denoted by enclosing

---

9The details depend on the particular LISP dialect, since this form of composition requires explicit access to the APPLY operation. I am not aware of any LISP dialect where this composition can be written in a uniform way for EXPR's, LEXPR's, and FEXPR's.

10Even the question of deciding what form of laziness to include in the definition of sequences has
pointed brackets \( <> \) that converts an arbitrary string to a string element (a string of length 1). Consequently, in TTL, we formalize sequences over the atom set \( At \) simply as generalized strings over \( At \).

Ironically, the universe of generalized strings over an atom set \( At \) is identical to the set of S-expressions (binary trees) over \( At \) assuming that the empty tree is not an atom in \( At \). However, neither the standard notation for S-expressions nor the definition of the primitive operations \( \text{head} (\text{car}) \) and \( \text{tail} (\text{cdr}) \), is compatible with the intuitive idea of generalized strings. In the domain of generalized strings, the \( \text{head} \) and \( \text{tail} \) operations treat atoms as sequences consisting of a single element. For any atom \( a \),

\[
\begin{align*}
\text{head}(a) &= a \\
\text{tail}(a) &= <> \quad \text{(the empty sequence)}
\end{align*}
\]

Among the new type definitions and operations that we want to enter in the TTL library are the primitive types and operations of Turner's functional language KRC [Turn 81] and Backus's FP language [Back 78]. With the exception of minor modifications for the sake of uniformity, we expect to incorporate both languages in their entirety within TTL.

### 2.3.2. Testing Methodology

The second major focus of the \( \mathbb{R}^n \) Specification and Testing Project is testing methodology. In particular, we are extending the theory of formal testing [Cart 81] to encompass programs utilizing floating point arithmetic and concurrency. Our approach to program testing radically differs from more conventional methods [Beiz 83] [Chan 81] because it is based on formal program specifications. For this reason, formal program testing is much closer in spirit to program verification than it is to conventional testing.

The core of a formal testing system is a verification condition generator and simplifier similar to those embedded in the Stanford Pascal Verifier [Igar 75], [Nels 79]. Like a program verification system, a formal testing system takes a program annotated with formal specifications as input, generates the corresponding verification conditions, and passes them through a simplifier. A fast simplifier incorporating sophisticated, computationally efficient decision procedures such as the Stanford Pascal Simplifier [Nels 79] will automatically prove or refute many verification conditions.

After the simplification step, the similarity between formal testing systems and formal verification systems ends. Instead of trying to prove the remaining
verification conditions, a formal testing system simply tests each one by evaluating it on a representative set of data values. The test cases are heuristically generated by the system under programmer guidance. If no errors are detected, the tested verification conditions are accepted as true statements, assuming the test data is adequate.

To establish the adequacy of the test data, the programmer can apply a local form of mutation analysis [Cart 81], which is much more computationally efficient than the global approach originally proposed in the literature [Budd 80]. In local mutation analysis, a mutation analyzer applies mutant operators to each program fragment S corresponding to a verification condition and produces a set of program fragments which are identical to S except for minor clerical errors. A set of test data is adequate for fragment S if for each mutant that is not semantically equivalent to S (does not satisfy its verification condition), the set includes an input value that falsifies the verification condition for S.

2.3.2.1. Floating Point Programs

It is particularly difficult to establish the intuitive correctness of floating point programs, because they typically do not have precise logical specifications. A well written numerical "code" implementing a stable algorithm will produce an answer close to the exact one as long as the specific problem is not ill-conditioned. On the other hand, if the particular problem is ill-conditioned (e.g. solving a near singular system of linear equations), a good code will usually abort. In most cases, the programmer/numerical analyst cannot give a succinct mathematical definition specifying the exact boundary between ill-conditioned and well-conditioned problems. Similarly, he cannot give a tight bound on the error in the solution. Worst case analyses (interval arithmetic) grossly overstate the errors in typical computations, because they ignore the effects of cancellation in the propagated round-off error.

Since the behavior of numerical programs is not accurately captured by standard logical specifications, we intend to investigate the feasibility of annotating them with "soft" specifications that do not necessarily hold in pathological cases. Soft specifications indicate how the program behaves almost all of the time. For example, a soft specification for Gaussian elimination with partial pivoting would assert a much tighter error bound than is provable by backward error analysis which is exponential in the size of the matrix [Wilk 65]. Such a specification is not always true either; reducing a pathological matrix (e.g. the example on p. 212 of [Wilk 65]) to upper triangular form will generate larger errors than the asserted bounds. Soft specifications are statements of the
numerical analyst's intuition about the behavior of "typical" computations. Although they obviously cannot be rigorously proven (since they may be false in pathological cases), they can be systematically tested. Formal testing (including local mutation analysis) is applicable to soft specifications as well as to "hard" ones. If a test case fails, the numerical analyst must decide which of three alternatives is true:

1. The test case is a pathological problem.
2. His program is incorrect.
3. His specifications make excessive claims.

If the program fails only the pathological cases of an adequate test data set, the program is correct (in a "soft" sense). In some cases, the numerical analyst may also wish to check that the program aborts rather than produce catastrophic errors. Establishing this claim requires formulating a different set of specifications that make much weaker claims about error bounds. In this case, the specifications that are checked are "hard" (always supposed to be true), but exceedingly tedious to prove. As in the case of most nonnumerical programs, formal testing is a practical alternative to formal verification (assuming the specifications can be evaluated).

The biggest obstacle to annotating numerical programs with specifications is that specification language must support real numbers as data objects. Otherwise there is no natural way to relate the computed values to the real numbers that they are supposed to approximate. On the other hand, it is unclear how to represent the real numbers as concrete data objects; the set is uncountably large. Moreover, formal testing requires that program assertions be executable, a property that is incompatible with expressions involving arbitrary real numbers. Fortunately, the specification language TTL accommodates an attractive solution to the problem.

There are obviously only countably many real numbers that are definable in any finite system of notation. For the representation $r$ of a real number $x$ to be computationally useful, there must be an algorithm that given $r$ and a tolerance $\delta$ generates a rational number $x'$ that differs from $x$ by less than $\delta$. Such real numbers are called *representable*. It is easy to prove that a real number is representable iff there is an algorithm for computing each digit in its decimal (or, equivalently, any other base) expansion.

In our experiments with testing floating point programs, we intend to define the real numbers as normalized binary radix floating point numbers with infinite fractions. Hence, a representable real number will simply be a pair consisting of
an integer exponent and a lazy sequence of bits (denoting the infinite fraction).

On the representable reals all normal arithmetic operations, except relational operators, are computable total functions. Relational operators (e.g. =, <, >) are computable partial functions that diverge on equal arguments. For this reason, we plan to use ternary relational operators that take an extra argument specifying a positive rational number as a tolerance. In this format, it is possible to define the relational operators on representable reals so that they are total.\textsuperscript{11} In the context of specifying the behavior of numerical programs, these ternary relational operators are natural primitives for stating the relationship between floating point numbers and the real numbers that they approximate.

2.3.2.2. Concurrent Programs

Concurrent programs are another class of programs for which are very difficult to validate. Conventional testing often fails to uncover timing dependent race conditions. In addition erroneous behavior is often difficult to reproduce, because an execution trace may depend on external events (e.g. the completion of input-output operation, a clock interrupt) that are not exactly reproducible. In an attempt to overcome this problem, are developing formal testing methods that are applicable to concurrent programs expressed in terms of high level synchronization and communication primitives (e.g., remote procedure calls, asynchronous message passing, and monitors [Andr 82]). Formal testing is obviously applicable to any class of annotated concurrent programs that supports the automatic generation of executable verification conditions. Owicki and Gries [Owic 76] have demonstrated that for an interesting class\textsuperscript{12} of annotated concurrent programs, it is possible to generate the verification conditions automatically if the assertions are restricted to discussing safety properties. The generated set of verification conditions contains all of the verification conditions produced by a conventional sequential analysis of the program plus a set of non-interference conditions. Ironically, the safety properties of a program are exactly those that can be expressed simply by annotating a concurrent program with appropriate assertions written in a "sequential" specification language like TTL. Although the formal development is still incomplete, recent work by Schlichting and Schneider [Schl 82] suggests that the same techniques may be

\textsuperscript{11}On the boundary, the results of a test may be ambiguous depending on the particular binary expansions denoting the real numbers being compared. Every number denoted by a terminating expansion has two representations (e.g., the number one has the two binary expansions

\textsuperscript{12}Those that can be expressed in terms of conditional critical regions allowing access to shared global variables.
applicable to a wider variety of synchronization and communication primitives.

Besides generalizing the theory of formal testing to accommodate the validation of safety properties of concurrent programs, we are beginning to study the much more difficult problem of validating liveness properties [Owic 82]. Unfortunately, liveness properties cannot be expressed by annotating a concurrent program with assertions written in a "sequential" specification language (like TTL). Our progress in this area clearly depends on the success of our efforts to extend TTL to describe concurrent computations.

Since concurrent programs are particularly difficult to write and to validate, we believe that the potential rewards of formal testing are even greater for concurrent programs than they are for sequential ones. It is important to note that formal testing checks conditions (the non-interference conditions) that have no analog in conventional testing based on program execution.

2.3.3. A Program Specification and Testing System

To provide an experimental testbed for our theoretical ideas, we are building a language independent program testing system consisting of an intelligent editor for constructing annotated programs, an interpreter for executing program fragments and assertions, a verification generator, a simplifier, a test data generator, and a mutation analyzer. Although we are initially concentrating on FORTRAN, we expect the same system to work for other languages such as Pascal, Modula II, C and LISP with relatively little extension. The system is being written in a lexically scoped LISP dialect [Steel 78]; it heavily relies on the interactive environment supported by LISP.

2.3.3.1. Implementation Strategy

The key idea underlying the implementation of the testing system is that it is possible to devise a reasonably compact common abstract syntax for a variety of widely used procedural and applicative programming languages (LISP, TTL, FORTRAN, Pascal, Modula II, C, Sequential Ada) and formulate it as an extension of the abstract syntax for LISP. Each constructor in the common abstract syntax (that is not already a LISP primitive) is implemented by a LISP function (in some cases a FEXPR). The common abstract syntax can be viewed as an abstract intermediate language that is implemented by defining LISP functions that are the meanings (denotations) of the abstract syntax constructors. Hence, the common abstract syntax is effectively an extension of the abstract syntax of LISP. Obviously, not all constructs in the common abstract syntax are present in every programming language. Moreover, many constructs are not utilized in their full
generality in a particular language.

We are focusing our initial implementation efforts on building an intelligent editor that builds abstract syntax trees (LISP list structures), an interpreter for the abstract syntax, and verification generator, since it is possible to construct a primitive, but useful programming environment solely from those three components. In such a system, the programmer is responsible for generating his own test data.

We are implementing the editor and interpreter in conjunction with the R² Programming Environment project, since we firmly believe that our testing system should be an integral part of a comprehensive programming environment.

2.3.3.2. Test Data Generation and Mutation Analysis

Once the primitive testing system is operational, we will tackle the problem of designing and implementing the two remaining components of the testing system: an automatic test data generator and a mutation analyzer.

Since automatically choosing a representative set of test data for a program fragment is a very difficult problem, we expect that any practical test data generation system will frequently have to rely on programmer guidance. On the other hand, generating test data is less difficult in the context of a formal testing system than it is in general because formal specifications provide invaluable information. The basic paradigm that we intend to employ is to symbolically evaluate the verification condition deferring the binding of every variable until the last possible moment. At each predicate forcing a variable binding, we make a "non-deterministic" choice for the binding from a small set of heuristically generated values based on the particular predicate and its arguments. The generation of test cases proceeds by backtracking until all possible non-deterministic choices have been tried (given we have imposed a small bound on the maximum recursion depth allowed in evaluating recursively defined predicates and functions in the specification language). To guide the generation of test data, we may try to exploit some of the heuristics employed in the Boyer-Moore LISP Verifier [Boye 75, Boye 79] to identify induction variables. For each induction variable, the test data generator should generate test values corresponding to the base and induction steps in an inductive proof. Another obvious heuristic for generating important test cases is to use the test outputs from each path as test inputs to subsequent paths.

A good way to measure the performance of an automatic test data generation system is to use mutation analysis [Budd 80]. Mutation analysis is a formal procedure for assessing the adequacy of the data set used to test a program. In the
context of conventional program testing (where there are no formal specifications), it has the following formulation. Given a program P as input, a mutation analyzer performs two operations. First, it exhaustively generates a very large set of programs that are identical to P except that each one contains a single clerical error (or more accurately an inverse clerical error). Second, it executes each mutant on the test data and uses the results to partition the set of mutants into two groups: those that are distinguishable on the basis of test data from the original program and those that are not. After this process is complete, the programmer scrutinizes the indistinguishable mutants to determine whether or not they are all semantically equivalent to the original. If the answer is yes, the test data is certified as adequate for P. Under the assumption that either the original program or one of the mutants is correct, it is clear that the original program is correct if and only if it behaves correctly on every input in an adequate set of test data.

Although mutation analysis is an ingenious idea, there are three significant problems that a practical mutation analysis system must overcome. First, determining the set of mutants that are indistinguishable from the original program is a massive computation: every mutant must be executed on the test data until a test case produces a different answer than the original program or until the data is exhausted. For computationally complex programs, the analysis may not be computationally feasible. Second, it is doubtful that the programmer can reliably determine which indistinguishable mutants are equivalent to the original program. The equivalence may depend on the truth of an invariant assertion that the programmer does not know. Finally, if a program contains more than one clerical error, an adequate test data set may not reveal the fact that the program is incorrect. The crux of the problem is that correcting any single error does not necessarily yield a program that is semantically distinct from the original program. A simple example of this anomaly appears below:
<table>
<thead>
<tr>
<th>Incorrect program</th>
<th>Correct program</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>begin</code></td>
<td><code>begin</code></td>
</tr>
<tr>
<td><code>integer x,y,ct;</code></td>
<td><code>integer x,y,ct;</code></td>
</tr>
<tr>
<td><code>read(ct);</code></td>
<td><code>read(ct);</code></td>
</tr>
<tr>
<td><code>x:=ct;</code></td>
<td><code>x:=ct-1;</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>y:=2*ct;</code></td>
<td><code>y:=2*x;</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>end</code></td>
<td><code>end</code></td>
</tr>
</tbody>
</table>

where the variable `x` does not appear in any of the elided portions of the program. In the incorrect program, `x` is a dead variable. Hence, correcting the assignment to `x` will not affect program behavior. Similarly, correcting the assignment to `y` will not affect program behavior because `x` and `ct` have the same value (as a result of the incorrect assignment to `x`). In essence, the "behavioral distance" between programs (measured by testing) is not always consistent with "syntactic distance" between them (measured by counting clerical errors). We do not know how frequently this effect occurs in practice, but it is certainly disconcerting.

Since these problems appear very difficult to solve within the framework of conventional testing, we are skeptical that this formulation of mutation analysis will work very well in practice. On the other hand, in the context of formal testing, the situation looks much more promising, because all three problems appear to have satisfactory solutions.

In a formal testing system, mutation analysis can be reformulated as a local, incremental procedure instead of a monolithic, global one. Instead of analyzing the entire program as a unit, a local mutation analyzer independently analyzes each program fragment corresponding to a verification condition—a chunk of straight line code. Since each fragment is formally specified, the mutant analyzer knows exactly how each fragment is supposed to behave. This reformulation of mutation analysis drastically reduces its computational cost. Although the total number of mutants generated for a program is identical to that generated in the conventional formulation, the cost of executing each one is much less because it is a short straight line code fragment instead of an entire program.

In addition to reducing the computational burden, the existence of formal specifications simplifies the task of determining which mutants are equivalent to the original program. The specifications provide the programmer with helpful
information since they succinctly state what conditions hold at the point of the mutation. Moreover, many mutants can be automatically classified (without programmer assistance) because the equivalence relation can be formalized as an implication between verification conditions which is amenable to proof or refutation by a fast simplifier such as the Nelson-Oppen prover [Nels 79]. Of course, the simplifier will frequently fail to reduce an implication to either true or false, forcing the programmer to intervene.

Local mutation analysis also largely avoids the multiple error problem, because multiple errors cannot foil an incremental testing procedure unless two or more occur within a single program fragment. In the example presented above, there is no problem if two errors occur in different program fragments, because local mutation analysis independently analyzes each fragment.

2.3.3.3. Incorporating Specifications in the Environment Database

As soon as the pilot testing system is operational, we will extend the environment database so that it maintains a test data library (a test data set for each program module) and supports a specification based version control and program installation system. Since this portion of the specification and testing system strongly impacts the core of the R^2 Programming Environment project, it will be designed and implemented by a team consisting of members from both projects.

The core of a programming environment is a program manager that oversees a database of program modules. These modules are connected to each other by various relations—such as inclusion, reference, and revision—which satisfy certain invariants. For example, references to an included module must be consistent with type declarations of that module. Similarly, if a revision of a module does not support the same interface as the original, that fact should be reflected as an attribute of the revision. For the environment to function correctly, the program manager must maintain the integrity of this database by ensuring that the invariants are preserved.

Maintaining the integrity of the revision relation is usually called "version control". The program manager records in the database which modules have been revised and ensures that programs including a revised module are appropriately revised to reflect the changes. Version control systems such as that embedded in the Gandalf programming environment [Habe 82] typically support two different kinds of changes to a program module: revisions to the specifications and revisions to the implementation. The distinction is important because it determines whether programs including the module need to be changed.
A potential problem in programming environments that do not support formal specifications is that the version control system cannot enforce the distinction between these two kinds of revisions beyond ensuring that an implementation revision has the same external interface (type declarations and export lists) as the original. Consequently, the version control system is forced to rely on the questionable judgment of programmers in performing this classification.

A similar problem arises in the program installation process. A program environment maintains a library of "installed" modules that are ostensibly correct. Ideally, an environment should prevent an incorrect module from being installed—particularly if it supersedes a program that is correct. However, in the absence of formal specifications, a programming environment has no mechanism for validating the correctness of a program. It must rely on the programmer's judgment in this regard.

In the \texttt{R} Programming Environment, we intend to overcome these problems in enforcement by using formal specifications in conjunction with the test data library to validate revision classifications and module installations. In the presence of formal specifications, there are three different classes of changes to a module that are important: specification revisions, specification generalizations, and implementation revisions. A specification-\textit{generalization} differs from a specification revision in that it strengthens the specifications of the original. Hence, programs incorporating the module will not be affected by the revision (assuming the implementation is correct).

Given formal specifications and test data for every module, the environment can validate the installation of programs and the classification of module revisions. When a module is created, it is marked invalid until the testing system certifies that it is correct. As part of the testing process, the testing system builds a test data set for the module in the program data base. Before the program manager installs a module (new or revised), it must confirm that it has been validated by the testing system. If a programmer wishes to revise an installed module, he must create a new instance of the module and declare what kind a revision (specification modification, specification generalization, or implementation modification) he intends to make. The program manager responds by creating a copy of the module, classifying it in accordance with the programmer's instructions, and marking it invalid. The environment maintains the integrity of uninstalled revisions by 1) preventing the programmer from modifying the specification of an implementation revision and 2) forcing the test data set for an implementation revision or specification generalization to contain all of original module's test data. In addition, the testing system requires specification
generalizations to pass an extra certification step: the revised specifications must imply the original ones on all test cases in the test data set for the original module. Consequently, the testing system cannot certify the correctness of a specification generalization or implementation revision unless it is consistent (to the accuracy of testing) with the original module's specifications.

3. Research Facilities

The research laboratory supporting this project includes the following equipment:

(1) A VAX 11/780 with 4 megabytes of memory and 1.2 gigabytes of disk running VMS augmented by a Unix Emulator (Phoenix) developed by the Rice Laboratory for Computer Science and Engineering.

(2) Three VAX 11/750 systems, each configured with 4 megabytes of memory and 450 megabytes of disk, running Berkeley Unix.

(3) Eight SUN Workstations linked by a 10 megabit Ethernet to various file servers with 400 megabytes of disk. The SUN machines currently run Berkeley Unix, but we are also experimenting with the V-System developed under the supervision of David Cheriton [Cher 83] at Stanford.

Although we eventually plan to run our specification and testing system on SUN workstations, we do not expect the SUN workstation to meet the computational requirements of our system for at least another year. In its present form, SUN workstation is seriously limited by 1) a 2MB virtual address space, 2) the absence of floating point hardware, and 3) the lack of satisfactory network software (particularly a distributed file system). SUN Microsystems plans to upgrade the performance of the workstation by 1) redesigning the CPU board to support a 16MB virtual address space and hardware floating point instructions and 2) extending the operating system (Berkeley Unix) to support a distributed file system. In the interim, we intend to use VAX Unix machines augmented by high-resolution, bit-mapped terminals that support a reactive user interface similar to that available on the SUN workstations.
4. References

[ADJ 76]

[ADJ 77]

[ADJ 79]

[Andr 82]

[Back 78]

[Bate 83]

[Beiz 83]

[Boye 75]

[Boye 79]

[Bran 76]

[Budd 80]
T. Budd, R. DeMillo, R. Lipton, and F. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," Record Seventh Annual ACM Symposium on Principles of Programming Languages, pp. 213-233.

[Cart 76a]

[Cart 76b]
[Cart 80]

[Cart 81]

[Cart 82]

[Cart 83]

[Cart 83]

[Chan 81]

[Cher 83]

[Demers 80]
A. Demers and J. Donahue, "Data Types, Parameters, and Type Checking", Record Seventh Annual ACM Symposium on Principles of Programming Languages, pp. 12-23.

[Frie 76]

[Gerh 75]

[Gerh 79]

[Gord 77]

[Habe 82]
[Hend 76]

[Hood 83]

[Igar 75]

[Kami 80]

[Kami 83]

[Levi 79]

[Lisk 80]

[Loca 80]

[MacQ 82]

[MacQ 83]

[McCa 63]

[McGr 82]

[Miln 78]

[Nels 79]

[Owic 76]
[Owic 82]

[Plot 78]

[Schl 82]

[Scot 76]

[Scot 81]

[Stee 78]

[Stoy 77]

[Suzu 82]

[Turn 81]

[Ward 80]

[Wilc 65]
Appendix

Overview of
$\mathbb{R}^n$: An Experimental Computer Network
to Support Numerical Computation

Departments of
Mathematical Sciences and
Electrical Engineering

Rice University
Houston, Texas

Principal Investigators
Robert Cartwright
John E. Dennis
J. Robert Jump
Ken Kennedy

Contributing Investigators
C. Sidney Burrus
Faye Briggs
R.J.P. de Figueiredo-
Michael K. Donegan
Robert T. Hood
Don H. Johnson
Thomas W. Parks
Henry H. Rachford
David W. Scott
J. Bartlett Sinclair
Trond Steibaug
Richard A. Tapia
James R. Thompson
Scott K. Warren
Mary F. Wheeler
J.D. Wise

With the support of a five-year, 2.4 million dollar Coordinated Experimental Research Grant grant from the National Science Foundation, the Computer Science Faculty at Rice University is designing and implementing an experimental distributed computing system to support numerical computation. Although local networks of single user machines have already been proven for many nonnumerical applications, the concept has yet to be tried in the context of numerical program development and execution. The Rice Numerical Network, or $\mathbb{R}^n$, will consist of approximately 24 single-user numerical machines equipped with high-resolution bit-mapped screens, a 32-bit central processor, and vector floating point hardware. It will also include several specialized server nodes supporting a
high-performance vector floating point processor and various peripheral devices including a gateway to the CSnet communications network linking the nation's major computer science research centers.

The new facility will support a coherent research program in software systems, computer architecture, and quality numerical software, directed at creating a modern reactive environment for numerical computation. Despite stiff competition from industry and other universities, Rice University has recently assembled the nucleus of computer science faculty required to develop an innovative distributed computing system supporting vector numerical computation and to evaluate its utility as a tool for solving important scientific problems.

5. Introduction

During the last decade, a number of experimental distributed computing systems exploiting inexpensive microprocessor and integrated circuit technology have been developed to support nonnumerical computer applications such as program development, artificial intelligence research, the automated office, and computer-aided design. Some well-known examples are the Xerox personal computer systems (Alto, Dolphin, and Dorado machines), the MIT LISP machine, and the Three Rivers Computer PERQ system. The resulting "reactive" computer systems have proven to be far more productive tools than their conventional predecessors. Today the technology is sufficiently well-established that descendants of these experimental systems are commercially available. As a result, many computer science research centers are replacing their conventional interactive facilities with distributed computing environments.

In contrast to the dramatic evolution of distributed computing systems for nonnumerical applications, numerical computing environments have changed very little during the last decade. Although current scientific machines are cheaper and faster than their predecessors of ten years ago, their architecture is virtually unchanged (except for the inclusion of vector-oriented pipelining). Moreover, the user interface and programming environment supported by numerical software systems is still based largely on batch compilation, debugging, and execution. Instead of adopting distributed architectures, most research centers for numerical computation (such as Los Alamos Scientific Laboratories) are moving in the direction of larger multiprogrammed monolithic machines (e.g. the Cray-1) supporting "smart" terminals. In applications where floating point accuracy is not critical and the programming complexity is low, a
large minicomputer with an attached array processor is a cost-effective alternative to the large mainframe. However, neither approach supports a highly responsive, interactive programming environment. The philosophy of reactive distributed computing systems has yet to be tried in the context of numerical computation.

The primitive environments for developing numerical software are unfortunate, because numerical computation is the most common form of technical computing -- particularly in our geographic region. In the south central United States (New Mexico, Colorado, Texas, Oklahoma, Arkansas, and Louisiana), the primary focus of software research and development is numerical computation to support the booming energy industry (Exxon Production Research, Shell Development, Schlumberger Well Services, Digicon Seismic Software, Los Alamos Scientific Laboratories, etc.). Yet given the primitive state of numerical programming environments, talented graduate students typically choose more glamorous, less frustrating fields of specialization than numerical software systems.

To improve the quality of numerical programming environments, to strengthen our own education and research programs supporting numerical computing, and to pursue an important national research objective, we are building $\mathbb{R}^2$, the Rice Numerical Network, a computer laboratory consisting of a local network of single-user numerical machines augmented by a few peripheral device and vector processor servers. By the end of the project in 1987, each numerical machine will include a high-resolution bit mapped screen, a fast 32-bit processor incorporating a reconfigurable vector floating point unit, hardware to support a large virtual memory, and a small local disk to store frequently accessed user files. The peripheral device and vector processor servers are intended to provide large scale, sharable resources that are too expensive to incorporate in individual nodes.

The new facility will allow us to integrate our ongoing research projects with several new projects to form a coherent research program in software systems, computer architecture, and quality numerical software, directed at creating a modern reactive environment for numerical computation. The projects in this program fall into two categories: projects directly concerned with the design and implementation of the distributed numerical computing system, and application projects that utilize that system.
6. Systems Research Projects

In this section, we describe five projects that will provide tools and facilities for the $\mathbb{R}^n$ workstation. Each of these projects represents a natural extension of current work of Rice faculty. However, the availability of the facilities of $\mathbb{R}^n$ will permit us to apply our research to an exciting new problem: building a system of software tools to support the development, maintenance, and use of numerical software on a high-performance personal computer system.

6.1. Project Summaries

A Portable Unix Interface
(Cartwright, Hood, Kennedy)

In this project we intend to develop a portable Unix interface kit that enables us to quickly implement Unix at the level of C source compatibility on top of any suitable operating system. Subprojects include a retargetable C compiler, a retargetable linker, and the formal specification of Unix system calls. This project is a natural extension of our development of an implementation of Berkeley Unix (called xunix) on top of VAX/VMS.

A Programming Environment for Fortran
(Kennedy, Hood)

In spite of its deficiencies, Fortran remains the standard language for numerical programming, yet most of the sophisticated new tools for program development do not support it. On $\mathbb{R}^n$, we plan to develop a retargetable interactive programming environment that is particularly well-suited for Fortran. It will include an intelligent editor, a project manager system that maintains a database on all the program modules, an interactive source-level debugging system, and an optimizing compiler with extensive interprocedural analysis. This project will build on our previous compiler optimization work and the program manager we are currently developing to interprocedural optimization.

A Numerical Analyst's Workbench
(Donegan, Warren)

The goal of this project is to build a reactive software environment for numerical researchers. This environment will provide a variety of software tools accessible through a single friendly interface characterized by a window-oriented high-resolution graphical display, an object-oriented semantic model (as in Smalltalk), a uniform command syntax based on pointing at the screen, and the ability to use several tools at once and transfer the data from one tool to another easily.

Program Specification and Testing Systems
(Cartwright)

In an attempt to reduce the cost and to improve the reliability of software, this research project will develop methods and tools for systematically testing the correctness of programs, with particular emphasis on those that use floating point computation. The system we envision will employ a hybrid of testing and formal verification to validate the correctness of a
program annotated with very high level specifications. Besides making a program more understandable, the specifications facilitate the automatic generation of plausible test data. To handle floating point arithmetic, the system will accommodate "soft" specifications that may not hold for ill-conditioned problems.

Vector Processors to Support High Level Languages
(Jump, Cartwright, Kennedy)

With the Rice experimental optimizing compiler as a tool, we plan to investigate pipelined "vector" computer architectures with the goal of matching the instruction set to the intermediate code that is generated by vectorizing compilers. We are particularly interested in architectures in which the unit of parallelism is an entire loop body rather than a short chain of arithmetic operations. Using a small microprogrammed vector unit attached to an Apollo node as an experimental vehicle, we shall pay special attention to optimal management of the movement of data between memory and the processor, a fundamental bottleneck on most systems.

7. Applications Research Projects

We expect that the projects in this section will use \( \mathbb{R}^n \) to solve important numerical problems. The availability of a powerful personal workstation should lead to innovative new solution techniques that are not feasible in a conventional environment. Several of the projects described here take advantage of computation intensive interactive computer graphics to involve the human in the problem solving process and help him better understand the problem through visualization. Almost all of the projects use Fortran and would benefit from a rich reactive environment for numerical program development.

7.1. Project Summaries

An Interactive Data Analyzer
(Scott, Thompson)

Estimation of the density functions from experimental data is an important problem arising in many application areas such as pattern recognition. Understanding the structure of the density function from which data arises is often valuable in data analysis (discovering clusters, for example). A fundamental question is how to represent the density function so the human can visualize it. In one dimension a simple graph can be used, but in higher dimensions some form of perspective is required. We plan to use perspective and color to display the four-dimensional density function of three variables. With a joy stick to select our location and orientation, we can explore a space in which density is directly represented by color and shading. By using this system, a human will more readily recognize features of the estimated density and hence of the data.

Numerical Simulation of Physical Systems
(Rachford, Wheeler)

1. Adaptive Systems for Partial Differential Equations (Wheeler). We plan
to use the interactive graphics capabilities of $\mathbb{R}^n$ to enhance our understanding of adaptive finite element methods. We have already proved the theoretical value of techniques that modify the spatial grid size near "fronts", but we have not yet automated these "self-adaptive" techniques because the programs would require solving difficult heuristic search problems in the domain of artificial intelligence. We will use the $\mathbb{R}^n$ node to involve the human directly in the solution as a consultant on grid size. Ideally, the experience that a human expert gains from solving systems this way will make it easier for him to design a fully automatic system.

2. **Computer Modeling in Real Time** (Rachford). An interesting source of challenging numerical and engineering problems is the analysis and control of physical systems, such as as transmission networks, in real time. We have developed an approach to real time modeling that should be applicable to any process that can be formulated as a differential system. However, modeling each new system requires solving many challenging mathematical and engineering problems. Access to several processors that can communicate over a network would be enormously helpful in the development and testing of real time models, since otherwise we are confined to studying systems in the field.

**Effective Algorithms for Nonlinear Optimization**
(Dennis, Steihaug, Tapia)

1. **Using Graphics to Understand and Improve Algorithms** (Dennis). Many researchers derive their new ideas and insight into current methods for solving nonlinear optimization problems from the geometry of local models. In this project we shall use computer graphics to exploit the implicit geometry of these methods for research into improved methods, interactive restarting procedures, and education. One novel idea is the use of graphical "games" to illustrate the simple geometric ideas underlying important methods.

2. **Global Algorithms for Constrained Optimization** (Tapia). Developing effective global algorithms for constrained optimization is now recognized as one of the most difficult and important problems in the area. By using the screen to display the constraint surface and objective function while monitoring the progress of candidate algorithms, we hope to gain sufficient insight to formualte rules for improvement (not necessarily associated with a merit function) that will lead to fast convergence in an automatic algorithm.

3. **Large Scale Nonlinear Problems** (Steihaug). The aim of this research is to devise methods for large scale nonlinear problems, especially nonlinear least squares and the simultaneous solution of nonlinear equations. We have developed iterative methods that require very few evaluations of the objective function $F$ or of $F'$ and do not require storage of any matrix. The efficiency of a 'globalized' method based on our scheme is very sensitive to the choice of convergence tolerances; the correct choice is difficult to predict in advance. An interactive program development environment like $\mathbb{R}^n$ will enormously simplify the task of selecting these tolerances and understanding the practical behavior of the algorithm.

**High Performance Modular Systems Design**
(Jump)

The goal of this project is to explore the possibilities for achieving high performance with modular computing systems constructed from LSI and
VLSI devices. It includes the development and evaluation of modular organizations for which the level of performance is directly related to the number of modules. We plan to use $\mathbb{R}^n$ to interactively simulate interconnection structures for such systems, so that we can quickly identify and concentrate on the most important features.

**Task Assignment in Distributed Processing Systems**  
(Sinclair)

This research deals with the management of distributed task execution. In particular, we wish to find minimum cost assignments of modules of a single task to processors. Our current work focuses on finding optimum static assignments in systems where communication costs vary with time, (possibly caused by variation in network traffic). The NP-completeness of this problem for more than three processors motivates the investigation of heuristic algorithms for determining "good" assignments. We plan to use the $\mathbb{R}^n$ network to obtain performance data.

**Random Access Local Networks**  
(Johnson)

We are interested in studying the topologies and communications protocols for local-area networks, with an emphasis on random-access carrier-sense networks such as Ethernet. Our work is currently focused on optimum access strategies, which are intended to utilize more fully the capacity available for transmission. To date, the research has been primarily theoretical. We plan to construct an experimental carrier-sense network and attach it to an Apollo node. Within this net, the Apollo will monitor network traffic and gather statistics, tasks that will require most of its capacity. Such an experiment would not be feasible on a time-shared utility because it would substantially degrade response time.

**Signal and Image Processing**  
(Burrus, de Figueiredo, Parks)

1. *Digital Signal Processing* (Burrus, Parks). The primary focus of our work is the design and implementation of algorithms for digital filter design. On the $\mathbb{R}^n$ network it should be possible to develop an interactive digital filter design system that is more responsive than existing systems of similar cost because $\mathbb{R}^n$ provides a large physical address space, a fast dedicated processor, and a high-resolution screen ideal for graphic output and pointing. Our algorithms will be even more responsive if vector hardware is provided on the system.

2. *Image Processing and Analysis* (de Figueiredo). We are currently investigating methods for image analysis, particularly multiple image frames arising from motion, and image analysis. In the latter area, we are employing a mixed graph-theoretic and analytic approach. Solutions to these problems are better suited to an $\mathbb{R}^n$ numerical workstation than a time-shared machine because practical implementations require extensive man-machine interaction and large computations.