PK/C++: an Object-Oriented, Logic-Based, Executable Specification Language

Robert B. Terwilliger

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

Peter A. Kirslis

AT&T
11900 North Pecos St.
Denver, Colorado 80234

Technical Report CU-CS-400-88 (July 1988)
Department of Computer Science University of Colorado at Boulder

Preprint July 7, 1988

This research was supported in part by a gift from AT&T and NSF Grant CCR-8809418.

PK/C++: an Object-Oriented, Logic-Based, Executable Specification Language

Robert B. Terwilliger
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

Peter A. Kirslis
AT&T
11900 North Pecos St.
Denver, Colorado 80234

Abstract

ENCOMPASS is an environment that supports software development using formal techniques similar to the Vienna Development Method (VDM). In ENCOMPASS, software can be specified using the PLEASE family of executable specification languages. PK/C++ is the latest member of the PLEASE family. PK/C++ differs from its predecessor by having C++ rather than Ada® as its base language, by having an operational as well as declarative semantics, and by being based on flat rather than standard Prolog. Using PK/C++, software components are first specified using a combination of conventional programming languages and predicate logic. These abstract components are then incrementally refined into components in an implementation language. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. PK/C++ specifications may be used in proofs of correctness. They are also executable; therefore, initial specifications can be validated and refinements can be verified using testing-based techniques. We believe the use of PK/C++ will enhance the development process. In this paper, we give an overview of ENCOMPASS, describe PK/C++ in reasonable detail, and give an example of development using the language.

1. Introduction

In most cases, the efficient production of software remains an elusive goal. One of the most important problems is *quality*; many of the systems produced do not satisfy their purchasers in either functionality, performance or reliability. We consider a system *correct* if it satisfies its users in all the above criteria. Depending on the model of software development used, the software quality problem can be subdivided in a number of ways. Initially, a system exists only as an idea in the minds of its users or purchasers. In our model, the first step in the development process is the creation of a *specification* which precisely describes the properties and qualities of the software to be constructed [13]. Unfortunately, current methods do not guarantee that the specification correctly or completely describes the customers' desires. A specification is *validated* when it is shown to correctly state the customers' requirements [13]. It has been suggested that *prototyping* and the use of *executable specification languages* can enhance the communication between customers and developers [43]; providing prototypes for experimentation and

---

Ada® is a trademark of the US Government, Ada Joint Program Office.

This research was supported in part by a gift from AT&T and NSF Grant CCR-8809418.

evaluation should enhance the validation process. In general, the specification need not be executable; it must be translated into an implementation. Depending on the method used for translation, the exact relationship between the specification and implementation may be unknown. An implementation is *verified* when it is shown to satisfy its specification [13]. Many techniques can be used to certify this relationship including testing [23], technical review [12], and formal methods [19].

One solution to the verification problem is VDM (the Vienna Development Method) [4, 9, 17]. VDM supports the top-down development of software specified in a notation suitable for formal verification. In this method, components are first written using a combination of conventional programming languages and predicate logic. These *abstract components* are then incrementally refined into components in an implementation language. VDM alone can not ensure the production of correct software; most simply, it does not address the validation problem. Also, VDM relies exclusively on formal methods, while many feel that no one method alone can ensure correctness. Despite these problems, VDM is used in industrial environments to enhance the development process [4, 24, 30]. In these situations, formal specifications serve mostly as a tool for precise communication, and the major impact on methodology is that more time is spent on specification and design. However, the methods do prove useful in practice. VDM could prove even more useful if it was applied more formally and supported by automated tools.

ENCOMPASS [33, 36, 38, 40] is an integrated environment which supports incremental software development in a manner similar to VDM. ENCOMPASS extends VDM with the use of executable specifications and testing-based verification methods. It automates these techniques and integrates them smoothly into the traditional life-cycle. In ENCOMPASS, software is specified using a combination of natural language and the PLEASE [34, 35, 39] family of wide-spectrum, executable specification languages. PLEASE specifications may be used in proofs of correctness. They are also executable; therefore, initial specifications can be validated and refinements can be verified using any combination of testing, peer review and formally-based techniques. ENCOMPASS is an environment for the *rigorous* [17] development of programs; parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques.

Our approach to executable specifications has changed dramatically since we began our work. Our initial vision was of a purely declarative specification language and an extremely intelligent translation system that would automatically produce Prolog procedures from predicate logic assertions. Our experience has led us to believe that this is not a realistic approach in the short term. Therefore, the latest member of the PLEASE family, PK/C++

(Please Kernel on C++), differs from its predecessor by being based on C++ rather than Ada, having an operational as well as declarative semantics, and being based on flat (unification but no backtracking) rather than standard Prolog. We feel these changes will significantly enhance its use as a practical specification vehicle. Being based on C++ rather than Ada makes the power of object-oriented programming available. Having an operational as well as declarative semantics allows the programmer to hand optimize PK/C++ specifications for improved execution. Basing our execution strategy on flat rather than standard Prolog allows much simpler and somewhat more efficient implementation techniques to be used.

In the rest of this paper we describe ENCOMPASS and PK/C++ in more detail. In section two, we give an overview of ENCOMPASS, including a brief summary of VDM and a description of our previous work on executable specifications. In section three, we describe PK/C++ in more detail; specifically, the problems we encountered in our previous work and the corrective actions taken in PK/C++. In section four, we give an example of software development using PK/C++, including: producing a formal, but non-executable, specification of a function; producing a prototype for the same function using the logic programming features of PK/C++; optimizing the executable specification; and verifying that the optimized and original specifications are equivalent. In section five we describe the current status of the system and the issues which have arisen so far in the implementation of PK/C++, and in section six we summarize and draw some conclusions from our experience.

2. ENCOMPASS

ENCOMPASS is an environment to support a formal development method similar to VDM [5, 6, 18, 33-40]. In ENCOMPASS, the general approach is to address the validation problem using executable specifications and to address the verification problem using a combination of formal, testing, and peer review techniques. We can better understand ENCOMPASS after looking at VDM in more detail.

2.1. VDM

VDM (the Vienna Development Method) supports the construction of software using stepwise refinement and notations suitable for formal verification [1, 3, 4, 9, 16, 17, 24, 30]. In VDM, software is first specified using a combination of conventional programming languages and assertions written in first-order, predicate logic. To increase the expressive power of these specifications, the high-level types *set*, *list*, and *map* are added to the language. *Abstract components* specified with predicate logic assertions are then incrementally refined into components in the

implementation or *base* language. The refinements are performed one at a time, and each is verified before another is applied; therefore, the final components produced by the refinment process satisfy the original specifications. Since each refinement step is small, design and implementation errors can be detected and corrected sooner and at lower cost. VDM alone can not ensure the production of correct software. First of all, it does not address the validation problem. Secondly, it relies exclusively on formal methods although many widely used languages are not formally verifiable. Many feel that no one method alone can ensure the production of correct software [10, 11, 27].

Despite these problems, VDM is used in industrial environments to enhance the development process [4, 24, 30]. In this type of environment, the method is not typically applied in all its formality. Assertions are written using operations and predicates which may not be precisely defined. Verification conditions are generated without the aid of automated tools and proved informally using a peer review system. In these situations, formal specifications serve mostly as a tool for precise communication, and the major impact on methodology is that more time is spent on specification and design. However, the methods do prove useful in practice. Automated tools could enhance VDM both by easing application of the method and by reducing the number of errors. Many feel the cost of these tools is justified, and environments to support VDM are being constructed [2]. We feel that the time is ripe for the construction of environments which *partially* automate formal development methods, and that these environments will eventually prove useful in industrial settings.

2.2. ENCOMPASS

ENCOMPASS [33, 36, 38, 40] integrates a number of tools, methods and data structures to support incremental software development in a manner similar to VDM. ENCOMPASS supports a *phased* or *waterfall* life-cycle [13], extended to allow the use of executable specifications and VDM: a separate phase is added for user validation, and the design and implementation processes are combined into a single refinement phase. In ENCOMPASS, a development progresses through the phases: planning, requirements definition, validation, refinement and system integration. For a more detailed discussion of the ENCOMPASS lifecycle see [33, 34, 38].

In the *requirements definition phase*, software is specified using a combination of natural language and the PLEASE [34, 35, 39] family of wide-spectrum, executable specification languages. At the end of requirements definition it is not known if the software requirements specification correctly and completely states the customer's needs and desires. The *validation phase* attempts to show that any system which satisfies the specification will also

satisfy the customers. If not, then the requirements specification should be corrected before the development proceeds. To aid in the validation process, the PLEASE specifications can be used as prototypes in interactions with the customers. These prototypes may be subjected to a series of tests, be delivered to the customers for experimentation and evaluation, or be installed for production use on a trial basis. We feel the use of prototypes will increase customer/developer communication and enhance the validation process.

In the *refinement phase*, the PLEASE specifications are incrementally transformed into conventional implementations. The refinement phase can be decomposed into a number of steps, each of which consists of a *design transformation* and its associated *verification phase*. Each design transformation creates a new specification, whose relationship to the original is unknown. Before further refinements are performed, a verification phase must show that the refinement is correct. In our model, verification is accomplished using a combination of testing, technical review, and formal methods. PLEASE prototypes can be used to verify the correctness of refinements using testing techniques. Most simply, the prototype produced from a PLEASE specification can be used as a test oracle against which implementations can be compared. In a more complex situation, the prototypes produced from the original and refined specifications can be run on the same data and the results compared; this method gives significant assurance that a refinement is correct at low cost. PLEASE specifications also enhance the verification of system components using proof techniques; for the purpose of formal verification, the refinement process can be viewed as the construction of a proof in the Hoare calculus [15, 19].

ENCOMPASS supports the *rigorous* [17] development of programs. Although detailed mechanical proofs are not required at every step, the framework is present so that they can be constructed if necessary. Proof techniques may be used that range from a very detailed, completely formal proof using mechanical theorem proving to a development annotated with unproven verification conditions. Detailed, mechanical verification may be used on parts of a project, while other, less critical parts may be handled using less expensive techniques. Our experience so far leads us to believe that the complete, mechanical verification of large programs will be prohibitively expensive; however, inexpensive methods can certify a large percentage of the verification conditions generated during a development. By eliminating these trivial verification conditions, the total number is reduced so that those remaining can be more carefully considered by the development personnel. In ENCOMPASS, some modules of a system may be developed using PLEASE and IDEAL, while others are developed using conventional techniques. This flexibility allows formal methods to be used only on the most critical portions of the system where the increased

expense is justified.

ENCOMPASS provides support for all aspects of this development paradigm including simple tools for configuration control [18] and project management [5]. Many of the tools in ENCOMPASS are independent of the language used for development, but others are specific to PLEASE.

2.3. PLEASE

ENCOMPASS is founded on the PLEASE [34, 35, 39] family of wide-spectrum, executable specification languages. The design of these languages is a compromise between a number of conflicting goals. First, to increase their practical application the specifications must support the construction of software using conventional programming languages. Second, to enable the construction of high quality software we must be able to incrementally refine PLEASE specifications into formally verified implementations. Unfortunately, there is a conflict between the first and second goals. Most conventional languages were not designed with program verification in mind; therefore, they contain constructs for which no simple formal semantics have been developed. Although PLEASE uses the syntax of the implementation language, the constructs do not necessarily have identical semantics. The semantics of PLEASE constructs are defined using Hoare calculus proof rules [38]. When PLEASE is used within an encapsulated environment, special tools manipulate and display the abstract syntax in a format suitable for humans. Programs with the desired behavior are automatically created in the implementation language from the PLEASE abstract syntax trees.

The third design requirement is that PLEASE must allow the specification of software using pre- and post-conditions written in predicate logic; the more powerful the specification method, the better. Fourth, the language must allow the rapid, automatic construction of executable prototypes from these specifications; the prototypes should be as efficient as possible. Unfortunately, there is a conflict between the third and fourth goals. A fairly powerful specification method would use pre- and post-conditions written in the full first-order, predicate logic. A resolution theorem prover for first-order logic could be used to construct prototypes; however, the performance of these prototypes would be very poor. The emergence of logic programming as a technology, most notably Prolog [8], suggests that these techniques may provide a good compromise. Although in one sense not as powerful as full first-order logic, Prolog allows much more efficient implementation techniques to be used. By restricting the specifications to a logic with an efficient, Prolog-style implementation, reasonable specification power is combined

with implementation efficiency.

We have designed and implemented a number of logic-based executable specification languages during the course of our research, and our approach has undergone significant modification.

3. PK/C++

The latest member of the PLEASE family is PK/C++ (Please Kernel on C++). We have two major goals for the language. First and foremost, we want to produce an executable specification language practical enough that we can develop significant software using the methods we have described. Second, we want to investigate how executable, logic-based specifications and VDM-style development methods interact with object-oriented programming. With these goals in mind, we have made PK/C++ differ from its predecessor by being based on C++ rather than Ada, having an operational as well as declarative semantics, and being based on flat (unification but no backtracking) rather than standard Prolog. Although the use of C++ does not alter our fundamental approach, it is significant for its addition of object-oriented programming to our research.

We developed the initial version of PLEASE using a Pascal derivative [35]. Later, we felt that the lack of advanced separate compilation facilities and overloading was limiting our progress and we began using Ada [41] as the base language [34, 39]. The choice of Ada proved successful for a number of reasons, the most significant being the existence of a large body of work on the Ada-based specification language ANNA [20, 21]. PLEASE/Ada may be considered a subset of ANNA specifically chosen to support VDM. Recently, we have come to believe that an object-oriented language would expedite our research; therefore, C++ [32] was chosen as the new base language. We hope the contrast between Ada generics and C++ inheritance will provide some interesting insights. Also, at a more pragmatic level, the more "open" implementation of C++ may eliminate many of the minor crises we experienced in developing PLEASE/Ada (for example, using C++ we can inspect or modify the intermediate code or binaries produced by the system while in some Ada systems this is not possible).

3.1. Operational Semantics

PK/C++ also differs from its predecessors in having a well defined operational semantics. Our initial vision was of a purely declarative specification language and an extremely intelligent translation system. The specification would contain pre- and post-conditions written in "pure" Prolog which a knowledge-based translation tool would automatically transform into Prolog procedures, ordering clauses and adding "cuts" as appropriate. Our experience

has led us to believe that this is not a realistic approach in the short term. We believe it is too difficult for a translator to construct a prototype with reasonable efficiency and chances of termination from a purely declarative specification. We feel it is necessary to allow the programmer to control the order in which clauses are evaluated in a manner similar to Prolog. In PK/C++, the programmer has control of the evaluation order; therefore, specifications can be hand optimized to increase performance.

For example, Figure 1 shows the PLEASE/Ada specification of a component to compute the factorial of a natural number. This specification defines a package *factorial*, which provides a procedure by the same name. In PLEASE/Ada, procedures are defined using pre- and post-conditions which are designated by *in(...)* and *out(...)* respectively. The pre-condition for a procedure specifies the conditions the input data must satisfy before procedure execution begins. The pre-condition for *factorial* is *true*; the type declarations for the parameters give all the requirements for the input. The post-condition for a procedure states the conditions the output data must satisfy after procedure execution has completed. The post-condition for *factorial* is *is_fact(x,y)*; the predicate *is_fact* must be true of the parameters to *factorial* after execution is complete. The predicate *is_fact* is not pre-defined; it was developed as part of the *factorial* specification. The definition of *is_fact* states that *X* factorial is equal to *Y* if *X* equals zero and *Y* equals one, or if *X* minus one factorial is equal to *T1* and *Y* equals *T1* times *X* (in other words,

```
package factorial_pkg is

     --: predicate is_fact( X,Y : in out natural ) is true if
     --:     T1 : natural ;
     --: begin
     --:     X = 0 and Y = 1
     --:         or
     --:     is_fact(X-1,T1) and Y = T1 * X
     --: end is_fact ;

     procedure factorial( X : in natural ; Y : out natural ) ;
          --| where in( true ),
          --|       out( is_fact(X,Y) ) ;

end factorial_pkg ;
```

Figure 1. PLEASE/Ada specification of *factorial* procedure

is_fact(X,Y) is true if $(X = 0 \wedge Y = 1) \vee ((X\text{-}1)! = T1 \wedge Y = T1 * X)$.

In PLEASE/Ada, the evaluation order for clauses in predicate definitions was not defined; it was assumed the translator would use heuristics to reorder the clauses for correct execution if necessary. In practice, this proved unrealistically difficult; small changes in the original specification could produce dramatic changes in the viability of the prototypes produced by the translator. For example, reordering the last clause of the *is_fact* definition to read *Y = T1 * X and is_fact(X-1,T1)* would produce a non functional prototype. We feel the only practical, short term solution to this problem is to provide a well defined operational semantics for the language. For example, Figure 2 shows the equivalent PK/C++ specification of *factorial.* In PK/C++ the operational semantics are well defined: the clauses in the predicate are considered in a left to right, top to bottom order as in Prolog. Therefore, the programmer can control the execution to achieve reasonable performance; however, it does add to his task in writing the specification. The ordering problem has not been eliminated, but now it is the programmer's problem.

3.2. Flat vs Standard Prolog

The final difference between PK/C++ and its predecessors is that it is based on flat rather than standard Prolog; by "flat" we mean Prolog with unification but no backtracking (this reduces the search to a flat tree or list). This approach allows much simpler and somewhat more efficient implementation techniques to be used. It is similar to the techniques being used to develop very efficient concurrent logic programming implementations for systems or kernel languages [29]. In this approach, we are trading off logical completeness for simplicity; in a simple sense,

```
predicate is_fact( natural& X , natural& Y ) {
        natural T1 ;
        return( X == 0 && Y == 1
                        ||
                is_fact(X-1,T1) && Y == T1*X
        ); }

void factorial( natural& X , natural& Y ) {
        pre(  true ) ;
        post( is_fact(X,Y) ) ; }
```

Figure 2. PK/C++ specification of *factorial* function

the semantics are the same. We can view the execution of a Prolog procedure as the search for a tuple of the corresponding relation deducible from the contents of the data base. If the procedure succeeds, then the tuple it finds is in the relation. If the procedure fails then we know nothing; a tuple might exist, but Prolog was unable to find it. These statements both hold true for flat Prolog, but there will be more tuples that flat Prolog can not find because it will not backtrack to explore all the possibilities. However, the use of flat Prolog has allowed us to construct a very simple and efficient prototype implementation of PK/C++.

For example, consider the following PLEASE/Ada definition of the predicate *permutation*:

```
--: predicate permutation( L1, L2 : in out natural_list ) is true if
--:       Front, Back : natural_list ;
--: begin
--:       L1 = [] and L2 = []
--:            or
--:       L1 = append(Front,cons(hd(L2),Back)) and
--:            permutation(append(Front,Back),tl(L2))
--: end ;
```

The definition states that two lists are permutations of each other if both of the lists are empty, or if the first element in the second list is somewhere in the first list and the remainder of the two lists are permutations of each other. This predicate definition can be translated into the following Prolog procedure which will function as a *generator*:

```
permutation(L1,L2) ←
       eq(L1,[]),eq(L2,[]).
permutation(L1,L2) ←
       eq(L1,Temp3),
       hd(L2,Temp1),
       cons(Temp1,Back,Temp2),
       append(Front,Temp2,Temp3),
       append(Front,Back,Temp4),
       tl(L2,Temp5),
       permutation(Temp4,Temp5).
```

In other words, this procedure can be used to generate all the permutations of a list in sequence using backtracking. In PLEASE/Ada this is a very executable specification; it can also be used to check if two lists are permutations of each other. The same specification in PK/C++ is easy to read, and still useful for communication, but is not executable in any useful way. It can not in general be used to determine if two lists are permutations of each other, and it will not generate all the permutations of a list in order. However, if it returns *true* the values of *L1* and *L2* will be permutations of each other; it will just not return *true* very often. However, the elimination of backtracking

dramatically simplifies the implementation of the language. In logic programming, variables are set to values during the unification process. With backtracking these "assignments" may have to be undone; therefore, more complicated data structures are required than when backtracking is not present.

To clarify our model further and show how PK/C++ specifications enhance the development process, we will consider an example of software construction. We will follow the development through requirements definition, validation of the specification using a PK/C++ prototype, a refinement optimizing the performance of the prototype, and verification of the optimization.

4. An Example of Software Development

Assume that a customer needs a component which sorts a list of integers. The component should take a possibly unsorted list as input and produce a sorted list that is a permutation of the original as output. In the requirements definition phase, the customer discusses his needs with the systems analyst and a requirements specification is produced. Along with other documentation, this specification might contain a component specified in PK/C++.

4.1. Specifying a Component

Figure 3 shows the PK/C++ specification of such a component[1]. The specification defines a function *sort*, which takes two arguments: the first is a possibly unsorted input list, the second is a sorted list produced as output. The specification defines the predicates *permutation* and *sorted*, as well as giving pre- and post-conditions for the function. The pre-condition for *sort* is simply *true*; the type declarations for the parameters give all the requirements for the input. The post-condition for *sort* states that the output is a permutation of the input and the output is sorted. The definition of *permutation* states that two lists are permutations of each other if both of the lists are empty, or if the first element in the second list is somewhere in the first list and the remainder of the two lists are permutations of each other. The definition of *sorted* states that a list is sorted if it is empty, or if it has one element, or if the first element is less than or equal to the second element and the remainder of the list is sorted.

In PK/C++, predicates are specified using Horn clauses: a subset of predicate logic which is also the basis for Prolog [7, 8]. This approach allows a simpler implementation than for full Prolog; however, there are drawbacks. For example, in pure Horn clause programming there is no way to specify the falsehood of formulae; for example, the fact that a list containing the elements 2, 1 (in that order) can never be sorted. The solution used in Prolog is the

[1] In PK/C++, as in C++, ||, && and == are the symbols for logical "or", "and" and equality respectively.

```
predicate sorted( list& L ) { return(
        L == emptylist
                ||
        tl(L) == emptylist
                ||
        hd(L) <= hd(tl(L)) && sorted(tl(L)) ); }

predicate permutation( list& L1 , list& L2 ) {
        list Front, Back ;
        return( L1 == emptylist && L2 == emptylist
                                ||
                L1 == append(Front,cons(hd(L2),Back)) &&
                permutation(append(Front,Back),tl(L2))
        ); }

void sort( list& L1 , list& L2 ) {
        pre( true ) ;
        post(-permutation(L1,L2) && sorted(L2) ) ; }
```

Figure 3. Declarative specification of *sort* function

closed world assumption: if a fact is not provably true then it is assumed to be false[2]. We believe the closed world assumption is fundamentally unsuited for use in incremental software development. In such a situation, each step in the development will add more information to an incomplete knowledge-base; therefore, it is not valid to assume a statement is false simply because it is not yet provable. At present the only way to specify negative information in PK/C++ is to define a new predicate which is by convention the negation of the original; for example, *not_sorted.*

It is unclear if the code in Figure 3 is better viewed as requirements, specification, or design. It certainly states requirements that the software must satisfy; however, in one sense the specification has already constrained the possible implementations to those using a function with two lists as arguments. This is an example of a general problem (or feature) of VDM-style techniques; in such methods the specification, design and implementation phases are fused together into a single refinement process. Actually, the methods are constraining only if we adopt a very simple and completely formal approach to their application. For example, the specification in Figure 3 could be refined into a piece of in-line code which performed an in-place sort on an array; however, the mathematics required to formally verify this refinement are far beyond those currently implemented in ENCOMPASS. We view the above scenario as a further argument for rigorous, rather than rigidly formal, software development. While many

[2] Unfortunately, the closed world assumption may cause inconsistencies for full first-order logic [25].

situations can now be handled with formal techniques, real software development still provides many situations which are not easily handled in this manner.

After the requirements specification has been created, it must be validated. The systems analyst can discuss the specification with the customer and obtain test data and expected results for the system. This process can be enhanced with the use of a PK/C++ prototype. Unfortunately, the specification in Figure 3 is not executable in any useful way. Specifically, it can not be used as a prototype for the *sort* function: taking *L1* as input and producing *L2* as output. In PLEASE/Ada the equivalent specification is executable using a "generate and test" paradigm; *permutation* generates all the permutations of *L1* in order until *sorted* rates one as acceptable. This will not work in PK/C++ because there is no backtracking to automatically generate all permutations in sequence. However, we can easily write a PK/C++ specification which will serve as a prototype.

4.2. Prototyping the Specification

For example, Figure 4 shows a PK/C++ specification which implements an insertion sort algorithm. The pre-

```
predicate insert_in_sorted_order( type& E , list& L1 , list& L2 ) {
        list L3 ;
        return( L1 == emptylist && L2 == cons(E,emptylist)
                                ||
                E <= hd(L1) && L2 == cons(E,L1)
                                ||
                insert_in_sorted_order(E,tl(L1),L3) && L2 == cons(hd(L1),L3)
        ); }

predicate insertion_sorted( list& L1 , list& L2 ) {
        list L3 ;
        return( L1 == emptylist && L2 == emptylist
                                ||
                insertion_sorted(tl(L1),L3) &&
                insert_in_sorted_order(hd(L1),L3,L2)
        ); }

void sort( list& L1 , list& L2 ) {
        pre( true ) ;
        post( insertion_sorted(L1,L2) ) ; }
```

Figure 4. Executable specification of *sort* function

condition for *sort* is still *true*, but now the post-condition is *insertion_sorted(L1,L2)*; in other words, the predicate *insertion_sorted* must hold for *L1* and *L2*. The definition of *insertion_sorted* implements a recursive insertion sort. We can read this as: if *L1* and *L2* are both the empty list then return *true*, else if the tail of *L1* insertion sorts to *L3* and inserting the first element in *L1* in *L3* produces *L2* then return *true*, else return *false*. The predicate *insert_in_sorted_order* is true if *L2* is the result of inserting *E* into *L1* while maintaining the sorted order. The definition of *insert_in_sorted_order* uses a sequential search to insert *E* in the correct location. This specification of *sort* is usable as a prototype and in fact is reasonably efficient, insertion sort being an $O(n^2)$ algorithm.

The prototype contains no conventional assignment statements; the equality predicate (==) subsumes their function and is more amenable to machine reasoning. The value of *L2* is set in the *unification* procedure (used to implement equality in PK/C++). For example, if *sort* is called with *L1* set to the empty list then *L2* will be set to the empty list in the unification procedure called from the first line of *insertion_sorted*. In theorem proving, the *most general unifier* of two formulae is a set of (text) substitutions which when applied to both make the formulae identical. For example, the most general unifier of *f(X)* and *f(1)* is *[1/X]*, read as "substitute *1* for *X*". Unification in Prolog in not an exact implementation of this process; for example, there is no most general unifier for f(X) and X, but in Prolog these two terms can be unified to create a recursive data structure. Unification in PK/C++ is like unification in Prolog, so its semantics are not those used in theorem proving [31]; however, in practice this has not been a problem.

Although we have produced a usable prototype, we have not solved all our problems. First, we do not know the precise relationship between the specifications in Figure 3 and Figure 4. Although this is an interesting problem from an intellectual point of view, and can be addressed using formal techniques, in practice it need not be considered. In practice, we can verify the equivalence between Figure 3 and Figure 4 using peer review, and since the insertion sort specification is written using predicates and pre- and post-conditions, it can be used as the formal specification in further development. However, before development continues we must be sure the customers are satisfied. The prototype should first be checked for correct execution on the acceptance tests compiled earlier. If it performs correctly in these instances then it can be delivered to the customers for evaluation[3]. If the prototype does not perform correctly then we know the specification is invalid and it must be revised and re-validated before the development continues.

[3] We note that no amount of testing can guarantee the correctness of the specification.

4.3. Optimization of the Prototype

Assume that the customers are satisfied with the function of the prototype, but are somewhat concerned about the performance - especially on large inputs[4]. To improve performance in this respect, the specification can be rewritten to implement an O(n lg n) sorting algorithm. For example, Figure 5 shows a PK/C++ *sort* specification which implements a quicksort algorithm. The specification contains two predicates: *partition*, which divides *L* into two sub-lists *Low* and *High* so that all the members of *Low* are less than or equal to the selected element and all the members of *High* are greater; and *quick_ sorted*, which uses *partition* to implement the quicksort itself. To sort the input list, *partition* is called to divide the input list into two sublists which are then sorted recursively and combined to form a sorted permutation of the original.

The performance of this prototype will be quite good, although not as good as a C++ program written using arrays instead of lists. Our general approach to optimization involves two steps. Before investing substantial time

```
predicate partition( type& E, list& L, list& Low , list& High ) {
        list Temp ;
        return( L == emptylist && Low == emptylist && High == emptylist
                                ||
                E <= hd(L) && partition(E,tl(L),Temp,High) && Low == cons(hd(L),Temp)
                                ||
                partition(E,tl(L),Low,Temp) && High == cons(hd(L),Temp)
        ); }

predicate quick_sorted( list& L1 , list& L2 ) {
        list Low, High, Slow, Shigh ;
        return( L1 == emptylist && L2 == emptylist
                                ||
                partition(hd(L1),tl(L1),Low,High) &&
                quick_sorted(Low,Slow) && quick_sorted(High,Shigh) &&
                L2 == append(Slow,cons(hd(L1),Shigh))
        ); }


void sort( list& L1 , list& L2 ) {
        pre( true ) ;
        post( quick_sorted(L1,L2) ) ; }
```

Figure 5. Optimized executable specification of *sort* function

[4] Data on the actual performance of this prototype appears in section 5.2.

in hand optimization of code, we believe that the actual performance of the system should be measured and the performance bottle necks determined. Only after this has been accomplished should detailed hand optimization begin. In other words: code that doesn't run much doesn't have to run fast. For many parts of the system PK/C++ prototypes will provide adequate performance. For system components which are genuinely performance critical hand coded C++ functions can be used. We believe that PK/C++ specifications are even easier to read than pure C++ code (a debatable claim) and that they are much more amenable to machine reasoning techniques [37].

We have now produced a *sort* prototype with increased performance. However, before we continue we must certify that the optimized and original specifications are equivalent.

4.4. Verifying the Refinement

A number of different methods may be used to show that the optimized specification is equivalent to the original. In the most informal case, either inspection of the original and optimized specifications by a senior designer, or a peer review process might be used. A more rigorous approach might run the original and optimized prototypes on the same test data and compare the results; this method gives significant assurance at low cost. In the most rigorous case, mathematical reasoning would be used. ENCOMPASS provides tools to support the construction of formal proofs of correctness. For example, ISLET [33] is a syntax-directed program/proof editor, similar to [26], in which the refinement process is viewed as the incremental construction of a proof in the Hoare calculus [15, 19, 22]. ISLET provides commands to add, delete and refine constructs; as the specification is transformed into an implementation (and the proof is constructed) verification conditions are automatically generated. These are first algebraically simplified and then subjected to a number of simple proof tactics; if these fail, input is generated for TED, a general purpose proof management system [14]. Unfortunately, at present ISLET is not robust enough to be used in major development efforts.

More practically, we can certify the equivalence of the original and refined specifications using a combination of testing and peer review techniques. We feel peer review is an important component in the verification process; many refinements can only be assessed in the context of experience. For example, consider a refinement in which the list-based specification of *sort* is transformed into one which takes arrays as parameters. From a formal viewpoint, this situation is difficult as we are transforming a potentially infinite data structure (list) into a finite data structure (array), thereby introducing error elements (index out of bounds) which may not have previously existed.

We can use testing techniques with the normal mapping between lists and arrays, but we should now test boundary conditions. Also, we must decide how big to make the array; as the validated specification used lists, the users may use extremely large inputs. At present, we feel situations such as this can best be addressed by a senior designer.

Once the design transformation has been verified, the new specification may be refined further and the process repeated until an implementation is produced. This then is the ENCOMPASS development paradigm: describe the system using an executable specification language, validate the specification by creating a prototype, and then refine the specification into an implementation.

5. System Implementation and Status

The ENCOMPASS environment has been under development since 1984; a prototype implementation of ENCOMPASS for PLEASE/Ada has been running under Berkeley UNIX® on Sun workstations since 1986. The prototype is written in a combination of C, Csh, Prolog and Ada and supports the construction of software using the Verdix Ada Development System [42]. PLEASE/Ada and ENCOMPASS have been used to develop a number of programs, including specification, prototyping, and mechanical verification. The current implementation of PLEASE/Ada is based on the UNSW Prolog interpreter [28]. The Prolog interpreter and Ada program run as separate processes and communicate through pipes[5]. This implementation is somewhat expensive; for example, on a Sun 2/170 there is a five CPU second overhead to start the Prolog interpreter, but this is incurred only once during program execution. A procedure call from Ada to Prolog costs about forty milliseconds excluding parameter conversion. As an example of actual performance, the prototype produced from a sort specification equivalent to the one in Figure 3 can process a list of length four in an average of .9 seconds and a list of length five in an average of 4.7 seconds. This poor performance is due both to the use of an exponential algorithm and the poor implementation of PLEASE/Ada. PK/C++ demonstrates dramatically increased performance relative to its predecessor.

5.1. Implementation of PK/C++

PK/C++ has been under development since the Spring of 1988. The current implementation is based on C++ version 1.2.1 and runs under both System V UNIX and Berkeley UNIX, on both AT&T 3b2s and Sun workstations. The architecture is extremely simple; no compiler or pre-processor for the language is used. The system consists of

---

UNIX® is a trademark of AT&T Bell Laboratories.

[5]Pipes are a buffering mechanism implemented in UNIX.

a number of predefined classes which are linked in with all PK/C++ programs. These classes include definitions of the VDM types, such as *set*, *list*, and *map*, as well as code to implement operations required for all types, such as unification, run-time type checking and error handling.

Figure 6 shows a simplified version of the class hierarchy for these routines. The root of the tree is the class *Type*; all object classes used in PK/C++ specifications must be descendants of *Type*. This hierarchy is easily extensible; new object classes can be added as descendants of *Type*, thereby inheriting the unification, type checking and error handling routines implemented there. For example, a stack can be seen as a list with restricted operations; elements can only be added or removed in the first position. We can model this relationship by adding a class *Stack* as a class derived from *List* in the PK/C++ hierarchy. The *Stack* class uses the member functions and data structure of the list class to implement the new type; *Stack* defines a new set of operations for its public interface and elects to not propagate the list operations not valid for stacks to this interface. The *Stack* class must also be assigned a type tag and provide function bodies for the virtual functions equality (==) and assignment (=); by doing so, the type-checking and unification code defined in the base class *Type* can be reused without modification.

Many desirable features of our implementation are due to the use of C++ virtual and/or overloaded functions and operators to provide polymorphic routines. In C++, built in operators whose first operand is a class object can

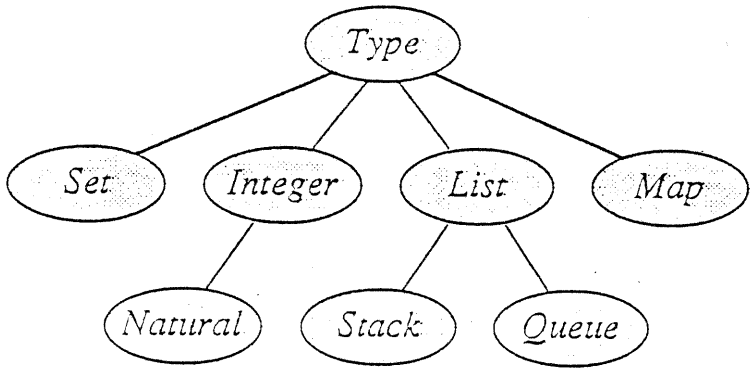


Figure 6. Simplified class hierarchy for PK/C++ support routines

be overloaded so that the compiler calls a user-defined routine. In fact, a number of user-defined routines can be supplied for an operator, each of which takes a different type as its second argument. Function names can also be overloaded in this manner. In addition, both operators and functions can be declared to be *virtual*, which permits the routine to be declared (but not necessarily implemented) in the parent class. Classes derived from this parent may provide an implementation for these routines. Instances of derived class objects can then be passed about as base class types, and when a base class virtual function is applied to the object the system determines at run time the correct derived class routine to call. If none is defined in the derived class, then the routine defined in the base class will be called.

For example, we declare the assignment operator (=) as virtual in the base class *Type*, taking an object of class *Type* as its second argument. Therefore, we can write a single assignment statement to assign any object derived from *Type* to any other object derived from *Type* without knowing how to perform the assignment. Each derived class supplies an implementation of "operator =". Since the second argument is a base type, we require the routine to check its type at run time and generate an error if the type is not appropriate. This permits our type hierarchy to be extensible: new types can be added without requiring any changes to any of the implementations of "operator =" already in place. Alternatively, we could have defined multiple versions of the virtual assignment operator in the base class, each taking a specific second derived type as an argument. In this case we would not need to make an explicit run time type check on the second argument, but we would sacrifice extensibility of the type hierarchy: any new type to be added would require a new virtual operator = to be added to the base class, and modifications to be made to all the classes which need to process this new type. In PK/C++, we elected to follow the former approach.

By taking this approach, we have strived to maintain strong type checking while permitting the use of "generic" routines. For example, an issue arises in the *List* class as to how to specify the type of objects contained in the list. We have written the *List* code to operate on objects of class *Type*, but with the requirement that lists be homogeneous. This allows the same code to be used to manipulate lists with any element type in the PK/C++ hierarchy; however, extra measures must be taken to provide type safety. For example, unless run-time type checking is performed by the *List* class member functions, using this setup it would be possible to append a list of lists to a list of integers. Our present approach uses additional data structures to provide structural type checking[5]. For example, the type designator for a List of Integers is itself a list structure containing the two tags "list" and "integer", and the

[6] While C++ uses name typing for classes, our use of structural typing allows us to have "generics" without needing to perform an instantiation for each new type.

type designator for a List of Lists of Naturals is a list structure containing the three tags "list", "list" and "natural". The type designators for objects are checked before operations are performed: for two lists to be appended, their type designators must be equivalent.

Another implementation issue is the choice of references or pointers as arguments to and return values from functions on objects. In C++, a program variable can be defined as being a *reference* to an object of a particular type. References must be assigned a value upon declaration, and become an alternate name for that object. Once initialized, a reference cannot be changed; any operators applied to the reference act on the object referenced. (The address-of operator (&) will return the address of the object being referenced, not the address of the reference itself.) By passing arguments to functions and predicates by reference, we get the same efficiency as by passing an explicit pointer to the object, without requiring the use (and visibility) of pointers in our implementation. This allows the function to change the value of the argument passed to it, which is essential for us to provide unification in predicates. None of the functions in the interface of classes in the type hierarchy take pointers to types as parameters; all either pass the parameter by reference or value. We wish to avoid the use of pointers in our client code since we want the user to think at a higher conceptual level while producing a specification; the use of references permits us to do this. The user normally has access to the "address-of" operator (&), but if we wish, we can overload it to force control to pass to our own routine, which could then print a message and terminate execution, effectively eliminating the use of pointers with our data types.

C++ also provides a reasonably elegant implementation of the notions of logical variable and unification. In PK/C++, program variables are equivalent to universally quantified variables in the logic used for annotations. In other words, a PK/C++ annotation says that a formula must hold true for all possible values of the program variables; at a particular point during execution, the program variables hold particular values which must satisfy the annotations. A PK/C++ logical variable, on the other hand, represents an existentially quantified variable in the annotation logic. In other words, there must exist a value for each logical variable that satisfies the annotations. Using this view we can see the "execution" of post-conditions as the search for values of the logical variables which satisfy the given formulae. The only thing missing is a way to set program variables to reflect the results of this search. In PK/C++ we provide an explicit operation to transform a program variable into a logical variable so that it can be set in an annotation. For example, the output variable of the *sort* function must be a logical variable so that

the post-condition can set it to an appropriate value.

Logical variables and unification are implemented in the *Type* class so that this code can be reused in all descendant classes. Logical variables are implemented using a *set* bit: if the bit is set then the object represents a program variable and has a value; if it is not set then the object is a logical variable and can be instantiated during unification. To implement unification, each object contains a *unify* field which is a pointer to an object of the same type. We can view unification as the construction of equivalence classes of variables; when two variables are unified their equivalence classes are merged. In PK/C++, the equivalence class of a logical variable is implemented as a circularly linked list using the *unify* field; when two variables are unified these lists are merged. We can view the unification of a constant to a variable as assigning the value to all variables in the equivalence class. In PK/C++, when a logical variable is unified with a value, the value is assigned to all the variables in the equivalence class using the virtual function defined for that data type, and their *set* bits are turned on. Thus they will be treated as constants from then on for the purposes of unification, until their *set* bits are turned off.

To fully implement PK/C++, features other than logical variables and unification are needed. First of all, predicates must be implemented. In PK/C++, predicates are implemented as C++ functions which return either *true* or *false*. The formulae used in predicate definitions have the standard C++ operational semantics; except for the == symbol which in PK/C++ stands for unification. The formulae used to define predicates in PK/C++ may contain recursive calls to other predicates; these calls have the standard C++ semantics. Pre- and post-conditions are implemented as simple functions which take formulae as arguments. As C++ uses eager evaluation, the formulae are invoked (and return *true* or *false*) before the functions are called. *Pre* and *post* simply check that the formulae returned *true* and raise the appropriate exception if necessary. In PK/C++ lists are implemented using pointers and all storage is allocated on the heap. This use of pointers is hidden in the implementation, and is not visible to users of the classes.

5.2. Performance of PK/C++

One of the major goals for PK/C++ was an implementation efficient enough that reasonably large systems could be prototyped using the language. Although we believe the performance of PK/C++ is still far below its ultimate potential, we are pleased with the current implementation. From the timing data we have gathered we can show that PK/C++ prototypes run about two orders of magnitude faster than in the PLEASE/Ada system, but still

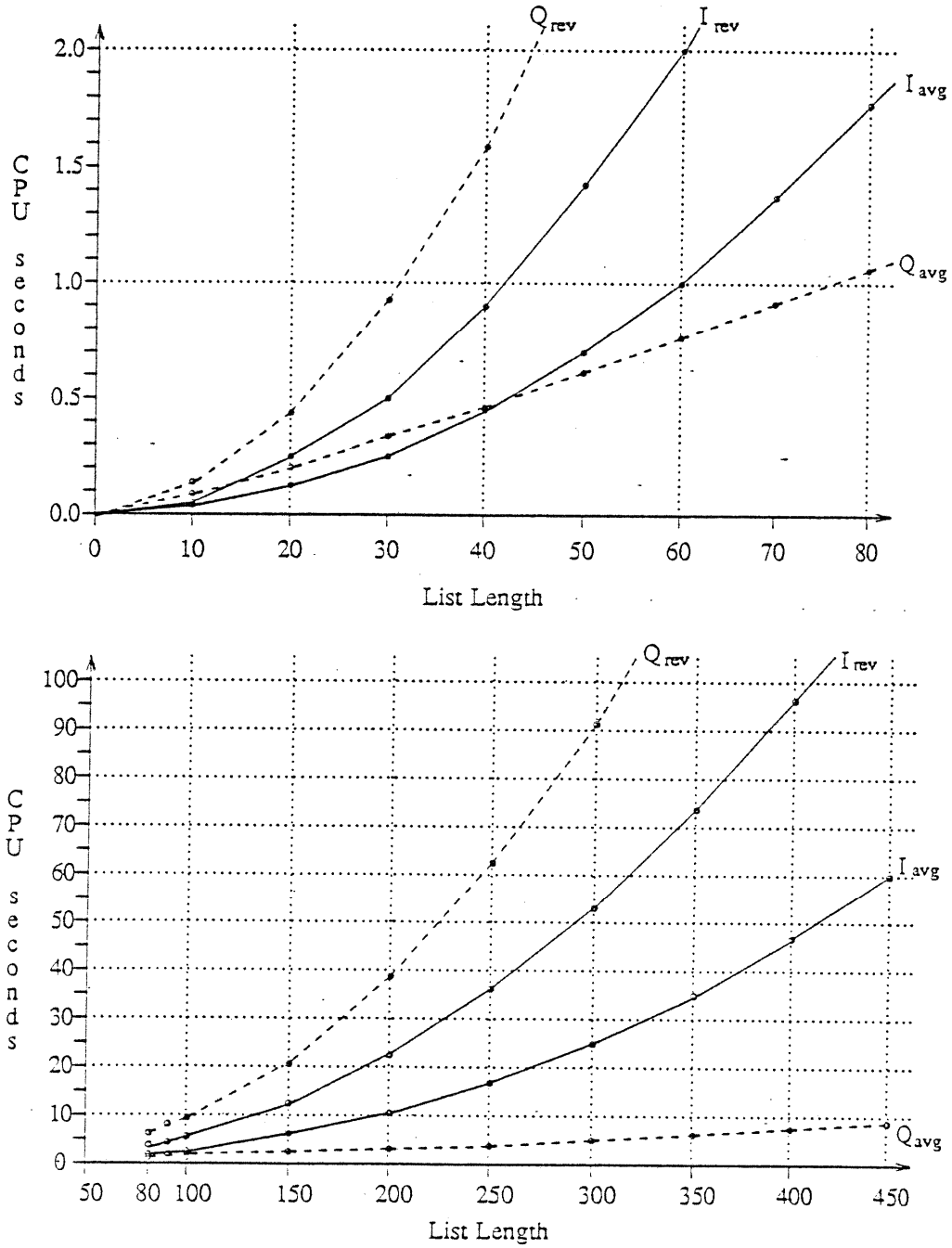
two orders of magnitude slower than hand coded C programs. For example, Figure 7 and Figure 8 show execution times for the PK/C++ *sort* specifications given in this paper, as well as for a hand coded C program which performs an in-place quicksort on an array. The insertion sort and quicksort executable specifications as well as the quicksort program were run on lists of lengths 10 through 100 by increments of 10, and from lengths of length 150 to 500 by increments of 50. Both of the executable specifications used lists of integers, while the quicksort implementation used arrays of *ints*. Both data structures were initialized with (pseudo) random sequences of integers[7]. Each of our "average" cases was determined by using ten different data sets of each length, with each sort being run ten times to better measure execution time for each run. The results of these one hundred runs for each length were then averaged to get the data point entered in the table. Our worst case for each specification was determined using lists in strictly decreasing reverse order. Ten runs were made for each list length, with the average value of these ten runs entered in the table. For each length, the same data that was used in the average quicksort specification example was also used in the C program.

6. Summary and Conclusions

ENCOMPASS is an environment to support a formal development method similar to VDM [5, 6, 18, 33–40]. In ENCOMPASS, the general approach is to address the validation problem using executable specifications and to address the verification problem using a combination of formal, testing, and peer review based techniques. ENCOMPASS supports a traditional lifecycle, extended to allow the use of executable specifications and formal methods similar to VDM. In ENCOMPASS, software is specified using a combination of natural language and the PLEASE [34, 35, 39] family of wide-spectrum, executable specification languages. PLEASE specifications can be incrementally refined into conventional implementations using IDEAL, an environment for programming-in-the-small which supports verification using peer review, testing or proof techniques. ENCOMPASS is an environment for the *rigorous* [17] development of programs; parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques.

Our approach to executable specifications has changed dramatically since we began our work; PK/C++ (Please Kernel on C++) is the latest member of the PLEASE family. PK/C++ differs from its predecessor by being based on C++ rather than Ada, having an operational as well as declarative semantics, and being based on flat rather

[7] We used the UNIX *drand* library routine, which employs a linear congruential algorithm with 48-bit arithmetic to produce non-negative double-precision floating-point values uniformly distributed over the interval [0.0,1.0). These values were then multiplied by a constant to produce the range of integers used in the sorting examples.



I_{avg} = insertion sort, average case Q_{avg} = quick sort, average case
I_{rev} = insertion sort, worst case Q_{rev} = quick sort, worst case

Figure 7. CPU seconds of execution vs list length for PK/C++ specifications

List Length	CPU time in seconds				
	Insertion Sort (Executable Spec)		Quicksort (Executable Spec)		Quicksort (Implemented)
	average	worst	average	worst	average
10	.04	.06	.09	.13	.00040
20	.12	.24	.20	.44	.00095
30	.26	.51	.33	.92	.0015
40	.45	.91	.46	1.59	.0022
50	.70	1.42	.61	2.43	.0029
60	1.01	2.02	.76	3.45	.0035
70	1.37	2.75	.91	4.64	.0042
80	1.78	3.59	1.07	6.02	.0049
90	2.28	4.54	1.23	7.57	.0057
100	2.77	5.60	1.41	9.32	.0064
150	6.19	12.5	2.27	21.3	.010
200	10.9	22.3	3.19	39.6	.014
250	17.0	36.4	4.1	62.6	.018
300	25.2	53.0	5.1	91.1	.023
350	35.2	73.0	6.2	126.	.027
400	47.	96.	7.2	–	.032
450	60.	123.	8.2	–	.036
500	74.	153.	9.3	–	.041

Figure 8. Run times of the executable specifications and C program

than standard Prolog. We feel these changes will significantly enhance its use as a practical specification vehicle. PK/C++ allows a procedure or function to be specified using pre- and post-conditions, a data type to have an invariant, and an implementation to be completely annotated. PK/C++ specifications may be used in proofs of correctness. They can also be executable; these prototypes can enhance the development process in a number of ways. Prototypes can increase the communication between customer and developer, thereby enhancing the validation process. PK/C++ prototypes can be used in experiments performed to guide the design process. Prototypes produced from a PK/C++ specification and its refinement can be run on the same test data and the results compared; this method can give significant assurance that a refinement is correct at a low cost. Since PK/C++ has an operational as

well as declarative semantics, the programmer can hand optimize PK/C++ specifications for improved execution.

We have completed a prototype implementation of PK/C++; it is written in C++ version 1.2.1 and runs on both AT&T 3b2s and Sun workstations. Basing our execution strategy on flat (unification but no backtracking) rather than standard Prolog allowed much simpler and somewhat more efficient implementation techniques to be used. Being based on C++ rather than Ada makes the power of object-oriented programming available. C++ classes and inheritance allow us to implement generics while retaining strong type checking and permitting extensibility. However, we are not sure our use of run-time structural typing is the final solution to the problem. Although we do not believe the current implementation demonstrates the full potential of the language, we are pleased with its performance. The execution data we have gathered shows that PK/C++ runs about two orders of magnitude faster than PLEASE/Ada, but is still two orders of magnitude slower than a C++ implementation. We feel we can improve its performance with further research. We believe that the use of methods similar to those based on PK/C++ specifications will enhance the design, development, validation and verification of software.

7. References

1. Bjorner, D. and C. B. Jones, *Formal Specification and Software Development*, Prentice-Hall, Englewood Cliffs, N.J., 1982.
2. Bjorner, D., T. Denvir, E. Meiling and J. S. Pedersen, "The RAISE Project - Fundamental Issues and Requirements", RAISE/DDC/EM/1, Dansk Datamatik Center, 1985.
3. Bjorner, D., "On The Use of Formal Methods in Software Development", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 17-29.
4. Bloomfield, R. E. and P. K. D. Froome, "The Application of Formal Methods to the Assessment of High Integrity Software", *IEEE Transactions on Software Engineering SE-12, 9* (September 1986), 988-993.
5. Campbell, R. H. and R. B. Terwilliger, "The SAGA Approach to Automated Project Management", in *International Workshop on Advanced Programming Environments*, Carter, L. R. (editor), Springer-Verlag Lecture Notes in Computer Science, New York, 1986, 145-159.
6. Campbell, R. H., H. Render, R. N. Sum, Jr. and R. B. Terwilliger, "Automating the Software Development Process", *Proceedings of the 1988 ACM Computer Science Conference*, February 1988.
7. Chang, C. and R. C. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.
8. Clocksin, W. F. and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
9. Cottam, I. D., "The Rigorous Development of a System Version Control Program", *IEEE Transactions on Software Engineering SE-10, 3* (March 1984), 143-154.
10. DeMillo, R. A., R. J. Lipton and A. J. Perlis, "Social Processes and Proofs of Theorems", *Communications of the ACM 22, 5* (May 1979), 271-280.
11. Dijkstra, E. W., "Structured Programming", in *Software Engineering Principles*, Buxton, J. N. and B. Randall (editor), NATO Science Committee, Brussels, Belgium, 1970.
12. Fagan, M. E., "Advances in Software Inspections", *IEEE Transactions on Software Engineering SE-12, 7* (July 1986), 744-751.
13. Fairley, R., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
14. Hammerslag, D. H., S. N. Kamin and R. H. Campbell, "Tree-Oriented Interactive Processing with an Application to Theorem-Proving", *Proceedings of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives*, December 1985.
15. Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM 12, 10* (October 1969), 576-580.
16. Jackson, M. I., "Developing Ada Programs Using the Vienna Development Method (VDM)", *Software - Practice and Experience 15, 3* (March 1985), 305-318.
17. Jones, C. B., *Software Development: A Rigorous Approach*, Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

18. Kirslis, P. A., R. B. Terwilliger and R. H. Campbell, "The SAGA Approach to Large Program Development in an Integrated Modular Environment", *Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large*, June 1985, 44-53.

19. Loeckx, J. and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, New York, 1984.

20. Luckham, D. C. and F. W. Henke, "An Overview of Anna, a Specification Language for Ada", *IEEE Software 2, 2* (March 1985), 9-22.

21. Luckham, D. C., D. P. Helmbold, S. Meldal, D. L. Bryan and M. A. Haberler, "Task Sequencing Language for Specifying Distributed Ada Systems, TSL-1", Report No. CSL-TR-87-334, Computer Systems Laboratory, Stanford University, July 1987.

22. Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

23. Meyers, G. J., *The Art of Software Testing*, John Wiley & Sons, New York, 1979.

24. Oest, O. N., "VDM From Research to Practice", *Information Processing*, 1986, 527-533.

25. Reiter, R., "On Closed World Data Bases", in *Logic and Data Bases*, Gallaire, H. and J. Minker (editor), Plenum Press, 1978.

26. Reps, T. and B. Alpern, "Interactive Proof Checking", *Proceedings of the 11th ACM Symposium on the Principles of Programming Languages*, January 1984, 36-45.

27. Richardson, D. J. and L. A. Clarke, "Partition Analysis: A Method Combining Testing and Verification", *IEEE Transactions on Software Engineering SE-11, 12* (December 1985), 1477-1490.

28. Sammut, C. A. and R. A. Sammut, "The Implementation of UNSW-Prolog", *The Australian Computer Journal 15, 2* (May 1983), 58-64.

29. Shapiro, E., "Concurrent Prolog: A Progress Report", *IEEE Computer 19*, 8 (August 1986), 44-58.

30. Shaw, R. C., P. N. Hudson and N. W. Davis, "Introduction of A Formal Technique into a Software Development Environment (Early Observations)", *Software Engineering Notes 9, 2* (April 1984), 54-79.

31. Stickel, M. E., "A Prolog Technology Theorem Prover", *Proceedings of the International Symposium on Logic Programming*, February 1984, 211-217.

32. Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986.

33. Terwilliger, R. B. and R. H. Campbell, "ENCOMPASS: an Environment for the Incremental Development of Software", Report No. UIUCDCS-R-86-1296, Dept. of Computer Science, University of Illinois at Urbana-Champaign (also to appear in the *Journal of Systems and Software*), September 1986.

34. Terwilliger, R. B. and R. H. Campbell, "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS-R-86-1295, Dept. of Computer Science, University of Illinois at Urbana-Champaign (also to appear in the *Journal of Systems and Software*), September 1986.

35. Terwilliger, R. B. and R. H. Campbell, "PLEASE: Predicate Logic based ExecutAble SpEcifications", *Proceedings of the 1986 ACM Computer Science Conference*, February 1986, 349-358.

36. Terwilliger, R. B. and R. H. Campbell, "ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications", *Proceedings of the 19th Hawaii International Conference on System Sciences*, January 1986, 436-447.

37. Terwilliger, R. B., "An Example of Knowledge-Based Development in ENCOMPASS", *Proceedings of the Third Annual Conference on Artificial Intelligence & Ada*, George Mason University, October 1987, 40-55.

38. Terwilliger, R. B., "ENCOMPASS: an Environment for Incremental Software Development using Executable, Logic-Based Specifications", Report No. UIUCDCS-R-87-1356 (Ph.D. Dissertation), Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1987.

39. Terwilliger, R. B. and R. H. Campbell, "PLEASE: a Language for Incremental Software Development", *Proceedings of the 4th International Workshop on Software Specification and Design*, April 1987, 249-256.

40. Terwilliger, R. B. and R. H. Campbell, "An Early Report on ENCOMPASS", *Proceedings of the 10th International Conference on Software Engineering*, April 1988, 344-354.

41. "Reference Manual for the ADA Programming Language", American National Standards Institute/MIL-STD-1815A-1983, U.S. Dept. Defense, 1983.

42. *VADS Reference Manual*, Verdix Corporation, Chantilly, Virginia, 1986.

43. "Special Issue on Rapid Prototyping: Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop", *Software Engineering Notes 7, 5* (December 1982).