Monitoring the Correctness of Software
(invited paper)

James M. Bieman Hwei Yin

Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096
bieman@cs.colostate.edu, yin@cs.colostate.edu

To appear in
Proceedings ISMM International Symposium on Engineering and Industrial Applications
Long Beach, California, December 1991

Technical Report CS–91–121

September 1991

Abstract

Software must perform its intended functions without failures when used in applications that require extremely high reliability. This paper explores one technique for achieving "ultra" reliable software. We have developed a notation for specifying runtime data invariants, and have implemented a system that monitors software to insure that specified invariants are satisfied.

1 Introduction

Many engineering and industrial applications require ultra-reliable software — software that will not fail. Such safety critical applications include medical software, space-flight software, software that controls hazardous industrial processes, systems that monitor expensive scientific experiments, etc. In many such safety critical applications the need for reliability justifies extra effort and expense to insure that the software performs correctly. Our research is directed towards creating automated support for the development of ultra-reliable software.

As Fred Brooks states, there is no silver bullet that will magically provide us with reliable software produced quickly and at low cost [1]. Developing reliable software is difficult and requires discipline in both specifying system functionality and implementing systems correctly. Approaches for developing highly reliable software includes the use of formal methods [7, 9, 6], and rigorous testing methods [2, 8, 5]. Testing cannot guarantee that software is correct [14], and verification requires enormous human effort and is subject to errors [4]. Automated support is necessary to help us insure software correctness.

In this paper, we describe the Prosper experimental system which supports run time correctness monitoring of software. Section 2 describes what we mean by correctness monitoring and introduces the Prosper system for specifying and monitoring correctness properties. Section 3 shows how our system works on a small example program, a sort routine. Section 4 provides additional details concerning the foundations and unique features of our system. Possible applications of our techniques are described in Section 5. Section 6 summarizes our results and describes our research plans.

2 Correctness Monitoring in Prosper

Our system is designed to monitor the behavior of a software system and indicate when system behavior does not meet its specification. Software can be specified in terms of data invariants. As an example, consider a data type `intlist` which consists of all sequences of integer values. We can define an invariant assertion that specifies that a particular `intlist` object L is ordered:

$$\forall i[i > 0 \wedge i < \text{length}(L) \Rightarrow L_i < L_{i+1}]$$

where L_i is the i'th element of `intlist` L. Assertions of this sort can be inserted as comments at particular points in a program to document that the assertion should hold whenever execution reaches this point. Pre and post conditions of software functions are also examples of invariants. Verification (proof of correctness techniques) must be used to demonstrate that an invariant cannot be violated during any program execution. In general, verification cannot be automated and thus requires significant human labor.

Rather than use verification to check correctness, Prosper uses its type system to monitor correctness at runtime. Assume that a software system makes use of integer lists. We can use a Prosper type, `intlist`, to represent integer lists. We can write assertions concerning integer lists as Prosper Boolean functions. For example, an assertion can determine whether a particular `intlist` is sorted:

`IsOrdered:  intlist ⇒ boolean`

In Prosper, the `IsOrdered` function is defined as follows (note that `(xt (intlist.L))` is used to specify formal parameters):

```
(define IsOrdered
 (fun (xt (intlist.L)) boolean
  (cond ((< (length L) 2) true)
        ((<= (first L) (first (tail L)))
         (IsOrdered (tail L)))
        (true false))))
```

Our current Prosper implementation uses a Lisp-like syntax, while the original Prosper design uses an ML-like syntax [10]. The second line in the above definition specifies the type of the function `IsOrdered`; the function inputs an `intlist` value which is bound to the parameter `L` and outputs a Boolean value. The body of the function is essentially Lisp code.

If we want a particular `intlist` variable to be maintained as an ordered list, we can define a new Prosper type `SortedList` which uses `IsOrdered` as a characteristic function:

```
(define SortedList (SELECT IsOrdered))
```

The Prosper `SELECT` function generates a new type from a characteristic function. In our example, `SELECT` uses `IsOrdered` as a characteristic function to define `SortedList`. Whenever an `intlist` value is bound to a parameter or identifier specified as of type `SortedList`, the Prosper type checker will determine whether the list is ordered using the `IsOrdered` function. An attempt to bind an unordered list to a `SortedList` parameter will cause a run time type error.

3 Monitoring a Sort Function

We can use the Prosper type system to ensure that important assertions such as function pre and post conditions are satisfied. A Post condition specifies the relation between the inputs and outputs of a function and can be a function's primary specification. In the following discussion, we show how the Prosper type system can be used to monitor the correctness of a program that sorts integer lists.

A function `sort` that sorts `intlist` values might have the post condition:

`Ordered(sort(L))` $\wedge$ `Permutation( L, sort(L))`

which states that the output of `sort` is an ordered permutation of its input.

We can write a Prosper function `SortPost` which is equivalent to the above post condition. The input to `SortPost` is two `intlist` values. `SortPost` outputs `true` if the second list is ordered and a permutation of the first list. `SortPost` has the following type signature:

`SortPost:  intlist` $\times$ `intlist` $\Rightarrow$ `boolean`

We can also write a *curried* version of `SortPost` which processes its arguments one at a time. The first input to the curried function is an `intlist`, say `L`, and the output is a new function. This new function takes a second `intlist` as input and returns `true` only when the second argument is a sorted permutation of the first argument `L`. The curried version of `SortPost` has the following signature:

`SortPost:  intlist` $\Rightarrow$ (`intlist` $\Rightarrow$ `boolean`)

The Prosper function to sort an `intlist` can have the following form:

```
(define Sort
 (fun (xt (intlist.L)) (SELECT (SortPost L))
      (SortBody L)))
```

where `SortBody` is a sorting function of type `intlist` $\Rightarrow$ `intlist`. Correctness monitoring is specified in the second line of the above function. We specify that the input to `Sort` is an `intlist` object `L`. We also specify that the output type must satisfy the characteristic function produced by `SortPost` when the input list is `L`. That is, `(SortPost L)` produces a new function, that evaluates as true only when the input to this new function is a permutation of `L` and ordered. The output of `Sort` must satisfy this characteristic function.

The execution and monitoring of `Sort` can be demonstrated with an example:

1. Assume `Sort` is invoked with list `M` as input: `(Sort M)`
2. The input list `M` is bound to formal parameter `L`.
3. The function body, `SortBody` is invoked with list `M`. `SortBody` produces a new (hopefully) sorted list as output.
4. `SortPost` is invoked with list `M` as input. `SortPost` produces a characteristic function as output. This characteristic function accepts a list as input and outputs true if and only if the input is ordered and a permutation of `M`.
5. `SELECT` defines a new `type`, a lists that satisfy the characteristic function produced in 4. This data type includes only one value, an integer list that is the legal output when `Sort` is invoked on input list `M`.
6. The output of `SortBody` is checked to see if it is of the output type defined in 5 above.

Thus, incorrect output is flagged immediately. Note that step 3 can be performed concurrently with steps 4 and 5.

The foregoing Prosper sort function is a simple example that demonstrates our technique for correctness monitoring. If the function `SortBody` produces incorrect results, the Prosper system will identify the error at a point close to the source.

4 Unique Features of Prosper

Our system is based on a functional language for specifying correctness properties; these properties are monitored during execution. The language uses higher-order functions, types as values, dynamic type checking, and parameterized type expressions. This combination of features for the purpose of correctness monitoring is unique.

Correctness properties are specified as (possibly) higher order Boolean functions, and correctness monitoring is performed by the type checking system. Higher order functions in Prosper may generate new functions, which can be used later for monitoring or other purposes. The `SortPost` function in Section 3 is an example higher order function. `SortPost` generates a new function as its output. The built in Prosper function `SELECT` is also higher order; its input is a function.

Types in Prosper are treated (almost) as normal data values in that types may be passed as arguments to functions and produced by Prosper progams. To avoid logical paradoxes that may result from the use of "type type" [11], types and normal values are kept in a hierarchy. Because types are treated as values and may be included in run time computation, much type checking, and correctness monitoring, must be performed at run time.

The enforcement mechanism encorporates specification assertions, expressed as Boolean functions, into type expressions. This approach was inspired by Nordström and Petersson [12] who suggest that types can completely specify programs. However the technique suggested by Nordström and Petersson is undecidable in general and relies on correctness proofs. We limit ourselves to executable type expressions so that correctness monitoring can be automated.

Prosper correctness monitoring makes use of two mechanisms for introducing parameters into type expressions allowing the creation of dependent types [3, 13]. The mechanism used in the sort example, *element parameters*, allows us to bind an identifier to a run time data value. We can use this binding in another part of the type expression or in the function body. We are thus able to specify that the output type of a function depends on the input value as in the example sorting program in Section 3. The other mechanism for parameterizing type expressions, *type parameters*, binds a name to the type of a run time data value. Type parameters can be used to specify generic functions.

The function `IsIntOrBool` uses a type parameter:

```
(define IsIntOrBool
 (fun (xt (T:TYPE)) boolean
  (cond ((= T integer) true)
        ((= T boolean) true))
        (true false))))
```

`IsIntOrBool` can be applied to values of any type and produces a Boolean result. The type parameter `T` is bound to the type of the data input to `IsIntOrBool`. For example, if the input to `IsIntOrBool` is the integer 7, `T` is bound to the type `integer`, not the value 7. If the input is the character string "silly", `T` is bound to type `string`. Thus `(IsIntOrBool 7)` evaluates to `true` and `(IsIntOrBool "silly")` evaluates to `false`.

We can use the `IsIntOrBool` function with the `SELECT` function to create a new type `IntBool` which consists of all possible integer and boolean values:

```
(define IntBool (SELECT IsIntOrBool))
```

As in the type `SortedList`, `IsIntOrBool` is used as the characteristic function to define a new type.

The combination of higer ordered functions, parameterized type expressions, and our treatment of types as values provides great flexibility for the monitoring of sofware correctness. Now we describe potential applications.

5 Potential Applications

Although our research into correctness monitoring is at an initial stage, we forsee a number of potential applications. The Prosper system can be used to monitor a program to insure that specified data invariants are satisfied. The sorting example in Section 3 is one demonstration of Prosper's correctness monitoring capabilities. The system can be used to help develop specifications, software prototypes, and should prove especially useful as a software testing tool.

We are currently testing Prosper for the development of flexible, highly parameterized executable specifications. We can specify complex data relationships in a very expressive manner using Prosper type and element parameters. Since Prosper specifications can execute, they are essentially system prototypes. We can view Prosper as the kernel of a specification system. Starting with primitive executable constructs we build towards higher level executable specification structures. When a Prosper specification is complete, a system prototype is automatically generated.

Because of the runtime nature of Prosper type checking and monitoring, a significant amount of computation is required. In the `Sort` example, the Prosper `SortPost` function is $\mathbf{O}(n^2)$, while sorting `SortBody` is $\mathbf{O}(n \log n)$. Although such overhead is often unacceptable in a final system, automated assertion enforcement is valuable in a prototype. A developer will get an early warning when system invariants are violated.

Our techniques have the potential to be combined in a software development environment with other languages. For example, correctness properties may be specified and monitored with a Prosper-like system, while the main system functionality is performed by a more traditional language such as C or Ada.

During testing, post condition checking is a critical activity. Often this checking is done by human analysts who read program output to determine correctness. An automated system, like Prosper, can dramatically speed up and improve the reliability of post condition checking.

6 Conclusions

We have implemented a system that can monitor the correctness of software. The system uses dynamic type checking, parameterized type expressions, and higher-order functions. These mechanisms allow us to include arbitrary data invariants as restrictions on legal values of complex data types. At runtime, the system can determine whether assertions are satisfied, as long as the assertions concern data values and can be expressed as an executable function. The system is flexible enough to enforce both the pre and post conditions of programs.

Our system demonstrates one technique, runtime correctness monitoring, as a tool to aid in the development of ultra-reliable software. Prosper can serve as a language for prototype development, as a specification language, or as a support tool for development in other languages. Correctness monitoring is especially appropriate during software testing.

The current Prosper system is implemented in Kyoto Common Lisp and runs on Sun SPARCS workstations. We have made a few minor syntax simplifications in the forgoing examples to ease the explanation. Because of the performance overhead of our system, Prosper correctness monitoring is currently appropriate only for systems processing limited sized data sets. However, we are working on improving system performance. See, [15] for additional details concerning the implemented Prosper system, and see [10] for the underlying foundation and design of Prosper.

References

[1] F. Brooks. No silver bullet – essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[2] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–148. North-Holland, 1981.

[3] A. Demers and J. Donahue. Datatypes, parameters, and typechecking. *Proc. 7th ACM Symposium on Principles of Programming Languages*, pages 12–23, 1980.

[4] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):803–820, May 1979.

[5] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offut, and K. N. King. An extended overview of the mothra sofware testing environment. *Proc. ACM SIGSOFT/IEEE Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, July 1988.

[6] S.J. Garland, J.V. Guttag, and J.J. Horning. Debugging larch shared language specifications. *IEEE Trans. Software Engineering*, 16(9):1044–1057, September 1990.

[7] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[8] W. E. Howden. A functional approach to program testing and analysis. *IEEE Trans. Software Engineering*, SE-12(10):997–1005, October 1986.

[9] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, London, 1986.

[10] J. Leszczyłowski and J.M. Bieman. PROSPER: A language for specification by prototyping. *Computer Languages*, 14(3):165–180, 1989.

[11] A. R. Meyer and M. B. Reinhold. Type is not a type. *Proc. 13th ACM Symposium on Principles of Programming Languages*, pages 287–288, January 1986.

[12] B. Nordstrom and K. Petersson. Types and specifications. *Information Processing 83*, 9:915–920, 1983.

[13] J. Reynolds. Towards a theory of type structure. *Lecture Notes in Computer Science*, 19:408–425, 1974.

[14] L. J. White. Basic mathematical definitions and results in testing. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 13–24. North-Holland, 1981.

[15] Hwei Yin. Implementation of a Dynamically Typed Functional Language. Master's thesis, Department of Computer Science, Colorado State University, 1991.