Restrictions: Help in Documenting Client Code Under a Verified Software Paradigm

Jason Kirschenbaum
The Ohio State University,
Columbus OH 43210, USA
Email:kirschen@cse.ohio-state.edu

***Abstract*—This paper presents an idea for documenting abstract client invariants in uses of ADTs in client code.**

I. Introduction

It has long been claimed that software professionals cannot be expected to write mathematically rigorous descriptions of their code such as formal specifications and loop invariants [1]. This contention arguably underestimates the capabilities of software professionals—after all, most of them have not been *taught* either why or how to write such annotations, so it is not surprising they are currently unequipped to do so. Nonetheless, the perception has led to exploration of some promising mitigating techniques that might be useful under a verified software paradigm. One approach involves inferring invariants (*e.g.,* loop invariants) either by dynamic or static analysis of code [2]. A complementary approach involves minimizing what needs to be written in mathematical language by providing special syntax for certain situations: syntax that looks more familiar and code-like to software developers. For instance, rather than demanding that the post-condition of an operation include a clause like `x' = x`, `x = old(x)` or `x = #x` to specify that the value of `x` does not change, most specification languages have tailor-made syntax for documenting this. JML [3] uses a `modifies` clause to list operation parameters whose values might be changed during the operation body. RESOLVE [4] offers (among others) a `restores` parameter "mode" to state that an operation parameter, while it might change temporarily during the body of the operation, has the same value at the end of the operation body as it had at the beginning.

The net effect of such mechanisms incrementally reduces the mathematical annotation burden for the software professional. It is not yet clear how effective the invariant-inference approach will be under a verified software paradigm; when automated verification does not succeed, it will be critical for a human to understand these invariants in order to repair the code, the annotations or both. This means that inferred invariants should be not only technically correct but also comprehensible to the software professional, who will ultimately be responsible for at least reading and likely for modifying formal mathematical descriptions of software behavior. Some human input into writing invariants and other assertions therefore seems unavoidable.

This paper describes an advance down the special-syntax path: providing language constructs to reduce the annotation burden. It focuses on relationships between abstract invariant properties of individual variables that hold during an entire code segment and loop invariants within that segment. We observe that two kinds of properties must be included in a loop invariant to verify software. The first kind arise from the desire to treat a loop as a single statement in straight-line code for verification and reasoning purposes. These properties document the behavior of the loop (by stating what it does not change); they are intimately tied to the loop and local to it. The second kind arise from the need to maintain continuity of abstract invariants on variable values. These properties are often incidental to a particular loop yet are critical pieces of the loop invariant. For example, when using memoization to avoid re-computation of a function with a Java `Map`, one abstract invariant on the `Map`'s value is that if a key is defined then the value associated with that key is the function applied to the key. This information must be written in the loop invariant for any loop involving the `Map`, because this property is true before (perhaps well before) the loop is encountered, is maintained by the loop, and might be intended to persist after (perhaps well after) the loop has terminated. This restricted set of `Map` values is known *a priori* by the software developer independently of any loops, and it can and should be documented. If the documentation is formal, its connection to the code can be verified. In other words, this documentation not only records the software developer's reasoning but—in a verified software paradigm—also can be used to check that the reasoning is correct.

This paper presents an idea for a programming language construct, restrictions, that can be used to document *abstract invariant properties of individual variables over segments of code*. The potential impact on automated verification arises partly because this construct implicitly provides guidance to the verifier by "factoring" potentially difficult verification conditions (VCs) into conceptually simpler VCs. The over-abundance of assumptions in VCs has been reported [5], [6] as a problem for back-end provers.

The paper is structured as follows. Section II presents a simple motivating example (in C++ rather than RESOLVE for brevity). Conclusion are in Section III.

II. Motivating Example

Consider code that computes x^p where x is a double and p is a positive integer; see Figure 1(a). It computes x^p by first computing x^{2^k} where k is the largest natural number that satisfies $2^k \leq p$ and then making a recursive call to finish the job. In this particular implementation, `q` always equals $2^{k'}$ where k' is some non-negative integer, and this property holds both as a loop invariant and, more generally, as an invariant on `q` throughout the code. We argue that this invariant can and should be documented.

One method a software professional can use to document the invariant on `q` is to add extra assertions in the code. At every line where the invariant holds, she `asserts` that the mathematical formula is true in the code. Frame properties allow one to limit the number of such statements needed, by using them only after a modification to a variable under consideration. This documents the invariant on `q`, but it is rather clumsy and the annotation burden is high. Restrictions are a construct to document the claims for this code more clearly and to reduce the annotation burden. Figure 1(b) shows what the code might look like in this situation. The loop invariant is simplified and the invariant on `q` is explicit. Each operation that is allowed on variables with a specific restriction is documented with both the constraints on the variables (*e.g.,* both have the `PowerOfTwo` restriction on them) and may also include additional `requires` clauses via the `also requires` section, as shown in the example, and additional `ensures` clauses in the `also ensures` section. The additional requires clauses are used to maintain the restriction (although this example uses a very restrictive `also requires` clause), while the additional ensures clauses allow for programmers to add more information about the effects of these operations while under the restriction.

```
double Power (double x, int p)
{
    double result = x;
    int q = 1;

    while(q <= p/2)
    /*!
        updates result, q
        maintains
            result = x ^ q and
            q <= p and
            there exists k : integer
                (q = 2 ^ k)
        decreases
            p - q
    !*/
    {
        q *= 2;
        result *= result;
    }

    if (p - q > 0)
    {
        result *= Power(x, p-q);
    }
    return result;
}
```

(a) Original version

```
/*!
  restriction PowerOfTwo(int n)
     is there exists k: integer (n = 2 ^ k)
    operations
         *(int n, int m)
           where PowerOfTwo(n)
           also requires m = 2
!*/
double Power (double x, int p)
{
    double result = x;
    int q = 1;
    /*! restrict q to PowerOfTwo !*/
    while(q <= p/2)
    /*!
        updates result, q
        maintains
            result = x ^ q and
            q <= p


        decreases
            p - q
    !*/
    {
        q *= 2;
        result *= result;
    }
    /*! relax PowerOfTwo on q!*/
    if (p - q > 0)
    {
        result *= Power(x, p-q);
    }
    return result;
}
```

(b) Documented with a restriction

Fig. 1. Code to compute x^p

III. Conclusion

We have presented an idea for a new programming language construct, restrictions, that may help address a limitation in current verification languages, namely the clumsiness of formally documenting client code. We argue that this approach not only can simplify VCs generated in client code, but also can result in reasoning reuse. This reuse can happen both when restrictions are reused across clients, and even when there are multiple calls to a single restriction operation by a particular client.

IV. Acknowledgments

The authors are grateful for the constructive feedback from Bruce Adcock, Derek Bronish, Paolo Bucci, Harvey M. Friedman, Wayne Heym, Bill Ogden, Aditi Tagore, Bruce Weide and Diego Zaccai. This material is based upon work supported by the National Science Foundation under Grants No. DMS-0701260 and CCF-0811737. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] C. Flanagan and S. Qadeer, "Predicate abstraction for software verification," in *POPL '02*. New York: ACM, 2002, pp. 191–202.

[2] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," *SIGPLAN Not.*, vol. 44, no. 6, pp. 223–234, 2009.

[3] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: notations and tools supporting detailed design in Java," in *OOPSLA 2000 Companion, Minneapolis, Minnesota*, 2000, pp. 105–106. [Online]. Available: citeseer.ist.psu.edu/leavens00jml.html

[4] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben, "Part I:

the RESOLVE framework and discipline: a research synopsis," *SIGSOFT Softw. Eng. Notes*, vol. 19, no. 4, pp. 23–28, 1994.

[5] K. Zee, V. Kuncak, and M. C. Rinard, "An integrated proof language for imperative programs," in *PLDI '09*. New York, NY, USA: ACM, 2009, pp. 338–351.

[6] J. Kirschenbaum, B. M. Adcock, D. Bronish, P. Bucci, and B. W. Weide, "Adapting isabelle theories to help verify code that uses abstract data types," in *SAVCBS*, November 2008, pp. 67–74.