R-Verify™: Deep Checking of Embedded Code

James R. Ezick, Donald D. Nguyen, Richard A. Lethin, Rick Pancoast
{ezick, nguyen, lethin}@reservoir.com, rick.pancoast@lmco.com
Reservoir Labs, Inc. Lockheed Martin

Introduction

Software is complex and software errors are an increasing source of cost and delay in DoD system development. Static verification is an attractive approach for mitigating these factors, since it can verify precise correctness properties at development time. However, existing tools suffer from a combination of insufficient scalability, lack of smooth integration into the development process, restriction to superficial checking, and prohibitive complexity of use. With R-Verify, we have addressed these issues by designing a system that tightly integrates with a compiler, checks native source-level assertions, and uses a program abstraction geared for a more powerful and efficient SAT-based constraint solver.

R-Verify can verify deep, bit-level properties of C programs making it ideal for verification of embedded code, including VSIPL [1] arithmetic code. It takes advantage of assertion-based testing methodologies by using program assertions as the program specification. That is, assertion statements in the program are used directly by the verification system as descriptions of correct program behavior. *Because our tool functions over properties expressed in the source language of the program, it is not necessary to learn any specialized formalism.* We consider this to be one of R-Verify's key competitive advantages.

R-Verify Development & Verification Cycle

R-Verify is built on three core Reservoir technologies [2]:

- **R-Stream™**, a high-level compiler, which contains a state-of-the-art framework that permits a wide-range of program analyses
- **Salt™**, a constraint translator, which efficiently generates optimized constraint systems from high-level program constraint descriptions
- **Alef™**, a multi-processor Boolean satisfiability (SAT) solver, which contains novel algorithms that allow it to take advantage of HPC hardware

The R-Stream compiler parses the input C program using an industry-standard C parser and performs multiple transformation passes to simplify the program structure. R-Verify uses the simplified program representation to generate a system of Salt language constraints that model the execution of the program. It then uses the Salt constraint translator and the Alef satisfiability solver to solve this constraint system and consequently verify the program. R-Verify is currently exposed to developers through a plug-in to the Eclipse IDE [3]. Figure 1 illustrates the R-Verify development and verification cycle.

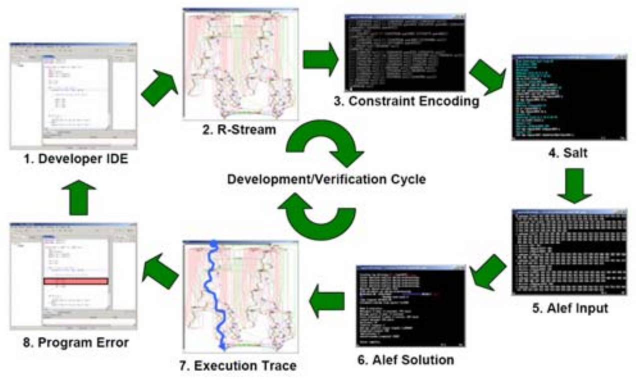


Figure 1: R-Verify development and verification process

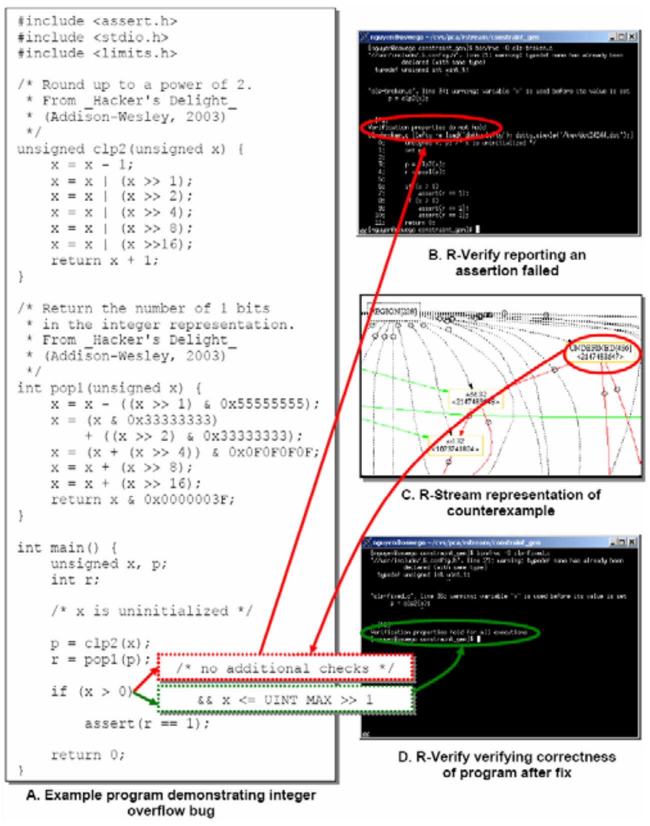
Deep Bit-Level Checking

Figure 2 illustrates how a developer can use the static assertion checking capability of R-Verify to isolate deep errors in an embedded application. The example program uses two optimized, bit-twiddling functions copied from a popular compendium of C routines to calculate (1) the next largest power of two (`clp2`) and (2) the population count or the number of non-zero bits in the binary representation of an integer (`pop1`). Using these two functions, the example program asserts that all powers of two calculated by `clp2` contain exactly one non-zero bit (see Figure 2A).

Unfortunately, the program as written contains an error derived from the fixed bit-width of computer integers. When the input argument to `clp2` becomes large enough, the arithmetic inside the function overflows and returns zero instead of the correct result (i.e., 2^{31}). R-Verify correctly identifies this error when verifying the program (see Figure 2B). The counterexample gives an instance where the assertion fails (see Figure 2C)—when the uninitialized integer `x` has the value 2,147,483,647 (i.e., 2^{31}-1). In this case, the fix for the error is to realize that the assertion does not hold for integers greater than a certain value due to the limitations of finite precision arithmetic. When the assertion is constrained to values between 0 and 2^{31}-1 (exclusive), then R-Verify correctly reports that the assertion holds for all executions (see Figure 2D).

If R-Verify identifies an error in a program, it shows a visual execution trace describing a counterexample. *R-Verify can quickly and efficiently find deep software errors that are invisible to superficial abstraction-based analysis. These types of errors are common in low-level embedded code and can easily go undetected during the normal software testing process. These undiscovered errors can be fatal in deployed software.*

This work was produced under the OSD Software Producibility Initiative, managed by ARL, under Phase I SBIR contract W911QX-06-C-0099 and DARPA ACIP project under Phase II SBIR contact W31P4Q-04-C-R257.



```
#include <assert.h>
#include <stdio.h>
#include <limits.h>

/* Round up to a power of 2.
 * From _Hacker's Delight_
 * (Addison-Wesley, 2003)
 */
unsigned clp2(unsigned x) {
     x = x - 1;
     x = x | (x >> 1);
     x = x | (x >> 2);
     x = x | (x >> 4);
     x = x | (x >> 8);
     x = x | (x >>16);
     return x + 1;
}

/* Return the number of 1 bits
 * in the integer representation.
 * From _Hacker's Delight_
 * (Addison-Wesley, 2003)
 */
int pop1(unsigned x) {
     x = x - ((x >> 1) & 0x55555555);
     x = (x & 0x33333333)
         + ((x >> 2) & 0x33333333);
     x = (x + (x >> 4)) & 0x0F0F0F0F;
     x = x + (x >> 8);
     x = x + (x >> 16);
     return x & 0x0000003F;
}

int main() {
     unsigned x, p;
     int r;

     /* x is uninitialized */

     p = clp2(x);
     r = pop1(p);    /* no additional checks */

     if (x > 0)        && x <= UINT MAX >> 1

         assert(r == 1);

     return 0;
}
```

A. Example program demonstrating integer overflow bug

```
Verification properties do not hold
```

B. R-Verify reporting an assertion failed

C. R-Stream representation of counterexample

D. R-Verify verifying correctness of program after fix

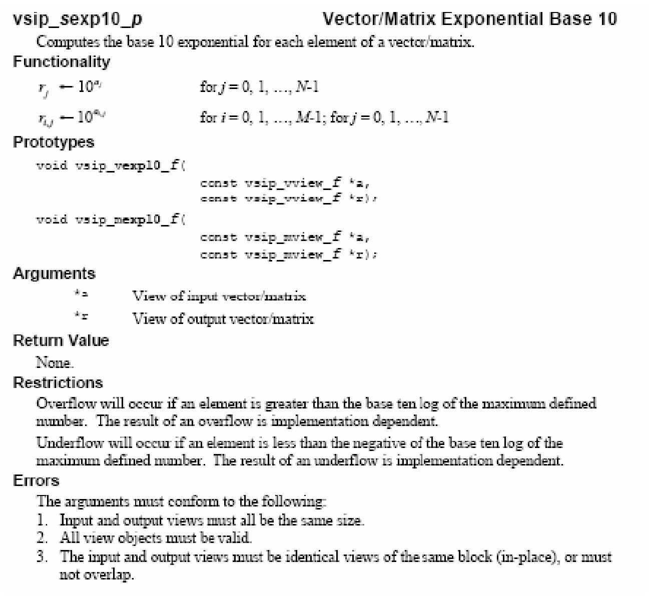
Figure 2: Example usage of R-Verify deep checking

SAT-Based Static Assertion Checking

R-Verify performs static assertion checking by constructing a single satisfiability problem that models the program semantics, and uses the Alef parallel SAT solver to determine if there is any execution that violates the program assertions. The satisfiability problem is constructed from two parts: the set of constraints that model the semantics of the input program $\boldsymbol{P}$ and the set of constraints that express the verification condition or assertions, $\boldsymbol{Q}$. The verification problem is then to find solutions to the formula: $\boldsymbol{P} \wedge \neg\boldsymbol{Q}$. That is, executions consistent with the semantics of the input program, but not with its assertions. Counterexamples to the assertions can be extracted from solutions to the verification problem. If there is no solution, then there is no execution that violates any of the program assertions. In this case, the program is correct with respect to the assertions.

The constraints generated are mostly straightforward translations of program operations into logical operations on bits. For example, the constraints that represent the addition of two numbers are similar to the Boolean circuit representation of a ripple-carry or carry look-ahead adder. *Thus, Boolean satisfiability constraints are the natural representation for bit-precise operator models and fast SAT algorithms make deep analysis in R-Verify tractable.*

Figure 3 presents a sample page from the VSIPL API specification that illustrates common types of both arithmetic and API call sequence restrictions. R-Verify, in conjunction with the static analysis capability already in R-Stream, can check each of these conditions.



vsip_sexp10_p Vector/Matrix Exponential Base 10

Computes the base 10 exponential for each element of a vector/matrix.

Functionality

$r_j \leftarrow 10^{a_j}$ for $j = 0, 1, \ldots, N\text{-}1$

$r_{i,j} \leftarrow 10^{a_{i,j}}$ for $i = 0, 1, \ldots, M\text{-}1$; for $j = 0, 1, \ldots, N\text{-}1$

Prototypes

```
void vsip_vexp10_f(
                    const vsip_vview_f *a,
                    const vsip_vview_f *r);
void vsip_mexp10_f(
                    const vsip_mview_f *a,
                    const vsip_mview_f *r);
```

Arguments

*a	View of input vector/matrix
*r	View of output vector/matrix

Return Value

None.

Restrictions

Overflow will occur if an element is greater than the base ten log of the maximum defined number. The result of an overflow is implementation dependent.

Underflow will occur if an element is less than the negative of the base ten log of the maximum defined number. The result of an underflow is implementation dependent.

Errors

The arguments must conform to the following:

1. Input and output views must all be the same size.
2. All view objects must be valid.
3. The input and output views must be identical views of the same block (in-place), or must not overlap.

Figure 3: Sample page from VSIPL specification

R-Verify renders constraint systems in the Salt constraint language for translation to the form required by Alef. The Salt translator accepts logical, set-theoretic, and arithmetic constraints in the Salt language. Salt arithmetic operators support both signed and unsigned arbitrary precision fixed-point operands and provide built-in support for arithmetic overflow constraints. Both truncation and rounding modes are supported. The Salt translator exploits the constraint semantics to perform optimizations that typically improve the search efficiency of the Alef SAT solver by 50-80%.

The Alef solver is a high-performance solver developed for HPC hardware. Alef supports parallelism both in the creation of multiple search threads and in the distribution of the computationally intense process of Boolean constraint propagation (BCP) [4]. When a violation is found, the Alef solver returns an assignment that is then trivially decoded into an execution trace using a map generated by Salt.

Conclusion

R-Verify is an integrated checking tool for embedded code designed to provide a deep analysis capability while still being easy to use. We are currently applying R-Verify to:

- Pre- and post-condition checking
- Memory safety of embedded device drivers, interrupt handlers, and VSIPL "admitted" blocks
- Numerical precision of arithmetic pipelines, with an emphasis on VSIPL pipeline implementations

References

[1] http://www.vsipl.org/.

[2] http://www.reservoir.com.

[3] http://www.eclipse.org.

[4] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *DAC'01*, Las Vegas, NV, June 2001.