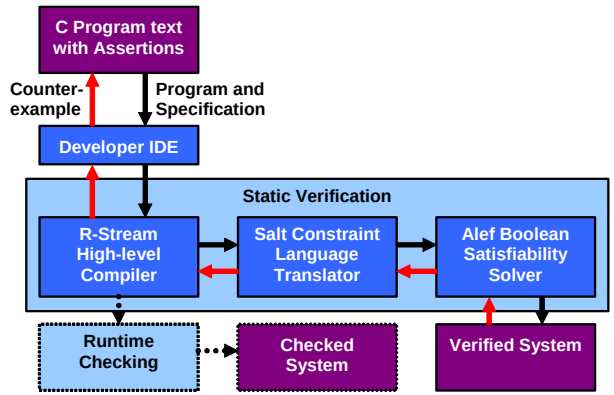
# R-Verify™: Deep Checking of Embedded Code

*Reservoir Labs, Inc.*

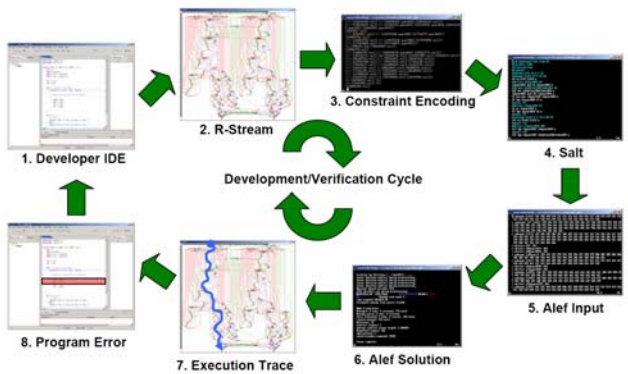
## R-Verify Design




**R-Verify** is built on three core Reservoir Labs technologies:

- **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 descriptions
- **Alef**™ – a multiprocessor Boolean satisfiability (SAT) solver which contains novel algorithms that allow it to take advantage of HPC hardware

## Development & Verification Cycle



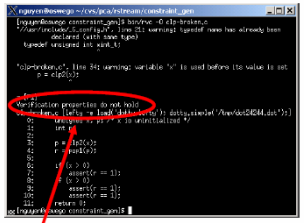

**R-Verify** plugs-in to existing IDEs such as Eclipse via an 8-step cycle:

1. **Normal IDE Source Development**
2. **R-Stream compiler builds an internal representation of the code**
3. **IR and specification are translated to Salt constraints**
4. **Salt tool translates the combined constraints to annotated CNF**
5. **Alef searches for an execution that violates the specification**
6. **Violation is returned to R-Stream**
7. **Violation is reflected as an error path in the R-Stream IR**
8. **Error path in the R-Stream IR is reflected back to the IDE**

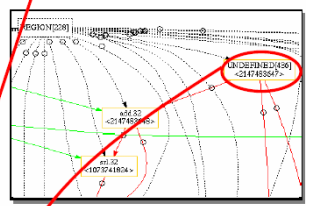
## Deep Bit-Level Checking

```
#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);

    if (x > 0)
        assert(r == 1);

    return 0;
}
```

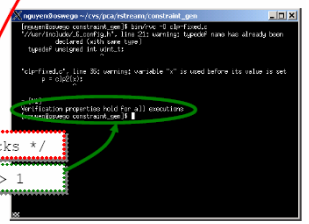
A. Example program demonstrating integer overflow bug



B. R-Verify reporting an assertion failed



C. R-Stream representation of counterexample




D. R-Verify verifying correctness of program after fix

***"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."***

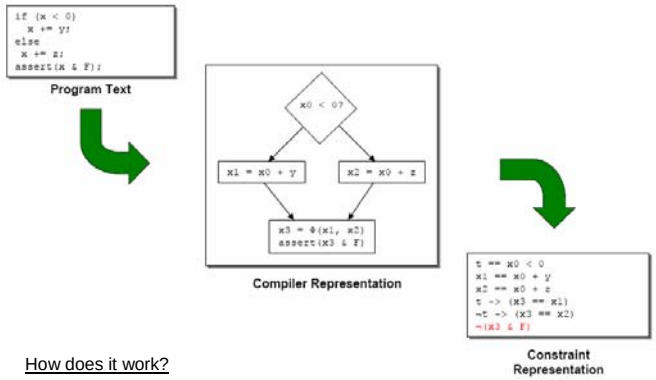
## The Satisfiability Problem

**Definition:** Given a Boolean formula $F$, decide if there is an assignment to the variables in $F$ such that $F$ evaluates to *true*.

**Example:** $F = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_3)$

**Solution:** $F$ evaluates to *true* (is satisfied) if $X_1 = 0$, $X_2 = 1$, $X_3 = 1$.

## SAT-Based Static Assertion Checking




How does it work?

1. **Model** the program (P) and the verification conditions (Q) using constraints
2. **Use Alef** SAT solver to look for a solution to: $P \wedge \neg Q$
3. **Decode** an Alef solution into a counter-example

## Application to VSIPL

**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.

We are currently applying **R-Verify** to:

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