# HCSS'07: The Verified Software Initiative

# The Verified Software Initiative[a]

N. Shankar

shankar@csl.sri.com

URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory

SRI International

Menlo Park, CA

[a]Supported by NSF Grant Nos.CNS-0646174.

## Outline

- What's the matter with software?
- The Verified Software Initiative (VSI)
  - The VSI Manifesto
  - The VSI Research Agenda
- Conclusions

## Overview

The construction of reliable software is one of the central intellectual challenges of the 21st century.

Verification is the rigorous demonstration of the correctness of software with respect to a specification of its intended behavior.

Automated verification technologies have been progressing rapidly in recent years.

The Verified Software Initiative is a fifteen-year research project with the goal of developing a large body (million plus source lines of code) of verified code.

The project will involve the collaboration of hundreds of researchers over tens of countries coordinated through a Verified Software Repository.

## Software Rocks!

Software is ubiquitous. It is the nerve center of the communication, entertainment, defence, transportation, power, financial, and commercial infrastructure.

According to a GAO report, 80% of the functionality of the F/A 22 is provided by software, up from 10% in the 1960s.

Software doesn't deteriorate with age.

Software is one of the few engineered artifacts that can, in principle, be analyzed with complete mathematical precision assuming correct hardware behavior.

## What's the Matter with Software?

Software now takes 40 to 50% of the development budget of the products in which it is used. Software development is 40% of the DoD R&D budget, and 40% of this is expended on fixing poor quality software.

Typically, over half of the development cost and time goes into detecting and correcting errors.

A 2002 NIST study estimated that unreliable software costs the United States 20 to 60 Billion dollars.

## The Window of Vulnerability

Software errors can be exploited as security holes.

In CACM (June 2006), FX Lindner wrote,

> *if software always worked as specified or intended by its makers, only a small subset would be vulnerable to attack, and defenses would be much easier to implement.*

The Code Red virus cost 2.6 Billion dollars, and overall computer viruses alone are estimated to cost around 17 Billion dollars in the year 2000.

## Software Complexity

A cell phone contains around 5 Million lines of code (MLOC).

The Red Har 7.1 Linux distribution contains about 30 MLOC.

| Year | Operating System | MLOC (Million) |
|---|---|---:|
| 1993 | Windows NT 3.1 | 6 |
| 1994 | Windows NT 3.5 | 10 |
| 1996 | Windows NT 4.0 | 16 |
| 2000 | Windows 2000 | 29 |
| 2002 | Windows XP | 40 |
| 2005 | Windows Vista Beta 2 | 50 |

## PITAC Reports

The 1999 Presidential Information Technology Advisory Committee (PITAC) report entitled *Information Technology Research: Investing in our Future* (`http://www.nitrd.gov/pitac/report/`) notes:

> *Software systems are now among the most complex human-engineered structures. ... The Nation needs robust systems, but the software our systems depend on is often fragile. . . . Even after large, expensive testing efforts, commercial software is shipped riddled with errors (bugs) . . . The Nation cannot afford to let the current situation continue. We must commit to developing the science, technologies, and methods needed to build robust software systems, ones that are reliable, fault-tolerant, secure, evolvable, scalable, maintainable, and cost-effective.*

## Hardware Verification

Hardware verification is far from a solved problem.

The 2005 edition of the *International Technology Roadmap for Semiconductors* (`http://public.itrs.net/`) asserts:

> *Without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry* (italics in the original).

Looking into the future, the new system-on-a-chip (SoC) hardware is . . . software.

## Software Verification

Software can, in principle, be verified with respect to a precise specification.

Practical software verification is therefore a significant intellectual challenge

1. Do we have a good enough theoretical understanding of the semantic aspects of software to fully formalize software correctness?

2. Are verification tools up to the task of demonstrating the correctness of realistically usable software?

3. Can verification technology enter the mainstream of software development?

Software verification alone does not solve unreliability, but if it does become practical, the improvement in software quality will be tangible.

## The Verified Software Initiative: Background

Tony Hoare in 2001 first proposed software verification as a grand scientific challenge for computing.

This led to a one-day workshop in April 2004, and a three-day workshop in February 2005.

A very successful IFIP Working Conference was held in October 2005 in Zurich, Switzerland.

The Verified Software Initiative was crafted at several follow-up workshops.

A second Working Conference will be held in October 2008.

## VSI: An Overview

- A fifteen-year scientific project involving hundreds of researchers spanning tens of countries.
- An inclusive research agenda focusing on long-term, collaborative work yielding proven and usable technology.
- The goal is the convincing demonstration of verification in the end-to-end development of large-scale software with
  1. Precise external specifications
  2. Complete internal specification, and
  3. Machine-checked proofs of their correspondence.
- Such software should continue to evolve in a verified state.

## VSI: The Motivation

- The high cost of unreliable and insecure software. This can only get worse in the networked, service-oriented, multi-core future.

- Verification theory is addressing real problems of object-oriented programs, safe interfaces, and secure information flow.

- Verification technology has been advancing rapidly on many fronts: static analysis, dynamic analysis, model checking, satisfiability procedures, interactive provers.

- Availability of representative benchmarks and open source specifications, code, and even IP cores as targets for experimentation.

## Some Landmark Verification Projects

The CLI Stack: ACL2 verification of FM9001, Piton Assembler, and micro-Gypsy programming language. Floating-point for AMD processors.

Several Spark ADA projects from Praxis UK.

The ESC/Java project: Assertion and error checking for Java programs using the Simplify theorem prover.

Météor Paris Metro code development with the B-tool.

The Microsoft SLAM project: C Device Driver verification using predicate abstraction and model checking.

Both Praxis and B-tool experience is that very few proof obligations need manual intervention.

## SWOT Analysis

| | |
|---|---|
| Strengths | Good foundation<br>Powerful and versatile tools<br>Growing standardization |
| Weaknesses | Scale/efficiency of tools<br>Lack of skilled manpower<br>Resistance to collaboration<br>Lack of recognition |
| Opportunities | Growing software complexity<br>High cost of unreliability/validation<br>Open source code base |
| Threats | Funding instability<br>Shrinking time-to-market<br>Lightweight bug-finding tools |

## VSI: The Technical Agenda

A *comprehensive theory* of programming that covers the features needed to build practical and reliable programs.

A *coherent toolset* that automates the theory and scales up to the analysis of large codes.

A *collection of verified programs* that replace existing unverified ones, and continue to evolve in a verified state.

*"You can't say anymore it can't be done! Here, we have done it. "*

The long-term goal is to establish that the science and practice of computing do converge in education and in industrial practice.

## VSI: A Comprehensive Theory

Theory constitutes languages, logics and calculi, semantics, models, proof techniques, and verification algorithms.

Languages include specification languages, modeling languages, foundational logics and calculi, programming languages, and assertion languages.

Theory provides abstraction techniques for separating concerns and refinement methods for linking levels of abstraction from specifications and models to implementations.

Semantics is the glue that links different languages, properties, execution, abstractions, and analysis algorithms.

## Theory Challenges

- Modeling formalisms for data, structure, behavior, and services.
- Golden models of widely used standards, protocols, and interfaces.
- Unified semantics for specification and programming languages.
- Language extensions for types, assertions, specifications, and performance.
- Tractable analysis techniques to infer or verify extensional and intensional properties.
- Compositional analyses to separate concerns along modules and aspects.
- Refinement techniques that automatically or manually bridge the gap between abstract specifications and concrete implementations.

## VSI: Tools

Dynamic analysis using testing, monitoring, and offline trace mining.

Static analysis to demonstrate the absence of runtime errors and to derive implied properties.

Model checking to check temporal properties.

Satisfiability solvers for assertion checking, bounded model checking, test case generation, and constraint solving.

Protocol analyzers for domain-specific problems like cryptographic protocols.

Interactive verifiers combining automated tools with manual guidance.

Code generators for producing correct code from high-level algorithmic descriptions.

## Static Analysis Example

```
Assume(n >= 0);
x := n;
y := 0;
z := 0;
while (x > 0)
  if (*)
     x := x - 1;
     y := y + 1;
   else
     x := x - 1;
     z := z + 1;


Assert(x = 0 AND y + z = n);
```

## SMT Solving Example

```
(not
 (forall (?i Int) (?pp Queue)(?aa Array)(?perm Array)(?ee Array)
         (?newperm Array)
         (implies (and (= ?ee (store (store (elems ?pp)
                                            (- ?i 1)
                                            (select (elems ?pp) ?i))
                                     ?i (select (elems ?pp) (- ?i 1))))
                       (= ?newperm (store (store ?perm
                                                 (- ?i 1)
                                                 (select ?perm ?i))
                                          ?i (select ?perm (- ?i 1))))
                       (forall (?i Int) (= (select ?aa (select ?perm ?i))
                                           (select (elems ?pp) ?i))))
                  (forall (?i Int) (= (select ?aa (select ?newperm ?i))
                                      (select ?ee ?i))))))
```

## SMT Progress

| | PVS 1982 | Simplify 1994 | SVC 1998 | ICS 2002 | Yices 2005 | Yices’ 2006 |
|---|---|---|---|---|---|---|
| tgc-3 | 2023.80 | 0.77 | 0.10 | 0.07 | 0.00 | 0.00 |
| tgc-8 | $> 3600$ | 1129.57 | 68.43 | 0.40 | 0.03 | 0.03 |

| | Simplify | SVC | ICS | Yices | BCLT | Yices’ |
|---|---|---|---|---|---|---|
| fischer6-10 | $> 3600$ | $> 3600$ | $> 3600$ | 38.89 | 12.32 | 3.45 |
| fischer6-20 | $> 3600$ | $> 3600$ | $> 3600$ | $> 3600$ | $> 3600$ | 140.00 |

| | SVC | UCLID 2003 | ICS | Yices | BCLT 2005 | Yices |
|---|---|---|---|---|---|---|
| cache-10 | 5.48 | 2.23 | 219.00 | 0.02 | 0.32 | 0.02 |
| cache-12 | 116.60 | 6.09 | $> 3600$ | 0.20 | 0.53 | 0.07 |
| cache-14 | $> 3600$ | 160.49 | $> 3600$ | 2.40 | 2.64 | 0.20 |

## Tools Challenges

- Challenges such as scalability, precision, extensibility, expressiveness, traceability, tunability, abstraction, and richer interfaces are common to all the tools.
- Combining tools to enhance their individual expressiveness, precision, and efficiency.
- Reduction techniques for decomposing the analysis using abstraction and decomposition.
- Tool workflows for progressing from cheap but imprecise analyses to expensive but precise ones, and defining iterative scripts.
- Evidence production in the form of error traces and proofs.

## An Evidential Tool Bus

Tool integration is central to the large-scale verification of software.

Examples of such integration include

1. Typechecking combined with decision procedures
2. Dynamic analysis to conjecture invariants combined with invariant checking
3. Static analysis to prove the absence of runtime errors combined with SMT solving to reduce false alarms.
4. Counterexample-guided abstraction refinement combining SMT solvers with model checkers.

## Evidential Tool Bus: Interchange Formats

XML-based intermediate languages will be used to exchange information.

These dialects cover propositional logic, typed and untyped quantifier-free first-order logic, and typed higher-order logic with definition principles and formalized theories.

Dialects similar to existing intermediate formats like SMT-LIB, BoogiePL, and SAL.

Labels/handles will be used to cover terms, types, declarations, transition systems, proof steps, BDD representations, and contexts.

## Evidential Tool Bus: Judgments

Judgments about the various semantic entities on the tool bus.

These include

1. $\Gamma$ is a type context representing the declarations $d_1, \ldots, d_n$.
2. $C$ is a decision procedure context representing the input atoms $\phi_1, \ldots, \phi_n$.
3. $\rho$ is a satisfying assignment for the formula $\phi$.
4. $\phi$ is a propositional formula.
5. $\hat{\tau}$ is an abstraction of the transition system $\tau$.

## Evidential Tool Bus: Scripting

The tool bus judgments are tied together using a logic programming framework.

Tools can be invoked explicitly or implicitly.

Each tool returns evidence supporting the judgment.

The tool bus registers the various syntactic entities, provides mappings between different languages, and manages the interaction.

SRI's Open Agent Architecture is being investigated as a prototype such a tool bus.

## VSI Experiments

| | |
|---|---|
| Years 1–2 | Pilot studies: Mondex, Flash file system, File synchronizer. |
| Years 3–5 | Specifications of common standards<br>Absence of runtime errors, security holes |
| Years 6–10 | Tool integration<br>Verification of medium examples: Distributed file systems, MINIX 3, medical devices<br>Libraries: GMP, STL, CUDD, OpenSSL, . . . |
| Years 11–15 | System verification: A LAMP stack |

## Conclusions

Software is woven into the fabric of society.

There is a high cost to software unreliability.

Software correctness is also a deep intellectual, scientific, and technological challenge.

The Verified Software Initiative is an ambitious 15-year program of research aimed at demonstrating the practicality of large-scale verification.

The VSI project will develop the theory, build and integrate the tools, and perform the experiments leading to the above goal.

We will have succeeded when verified software begins to drive out its unverified counterpart.