Building and Reasoning about Distributed Systems

Karl Levitt Ron Olsson Matt Bishop Biswanath Mukherjee Cui Zhang

Brian Becker Greg Benson George Fink Todd Heberlein
Mark Heckman Michael Helmke Calvin Ko Dave Peticolas Rob Shaw

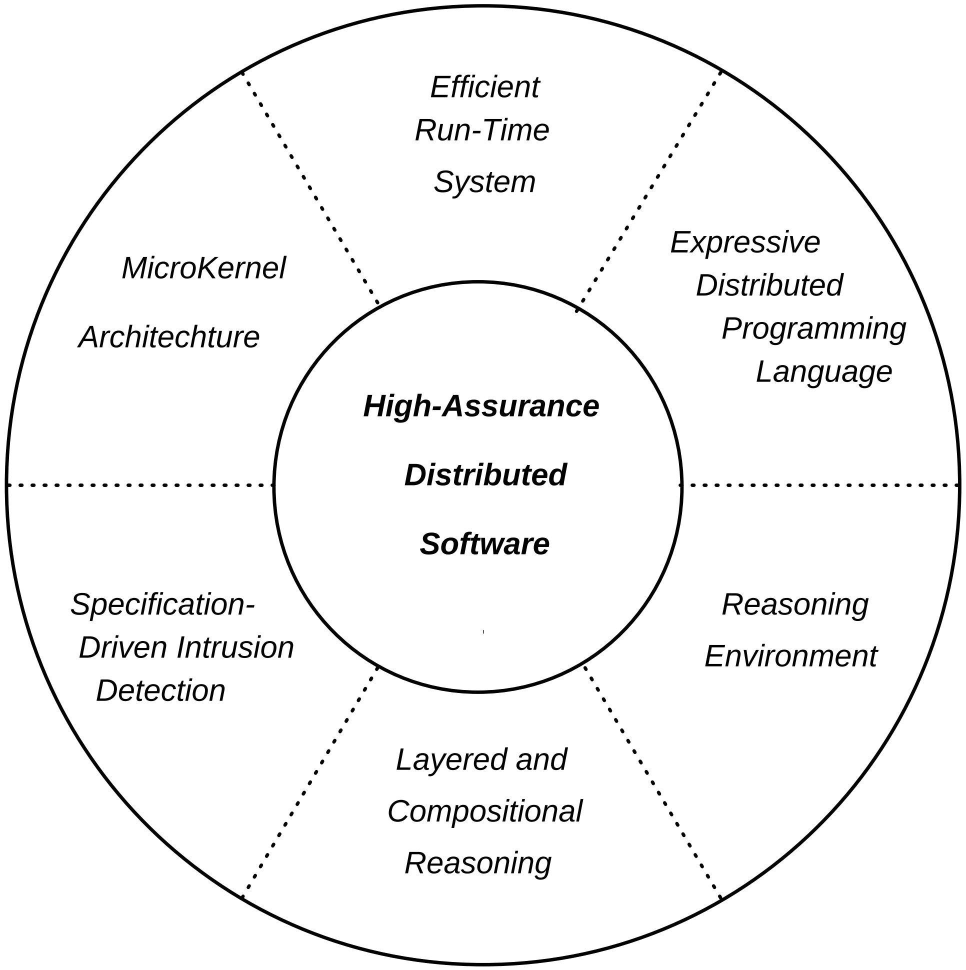
Programming Languages and Verification Lab
Department of Computer Science
University of California, Davis
July, 1995

Supported by

The Advanced Projects Research Agency

The NSA University Research Program

High-Assurance Software Research at UC Davis



Current Projects at UC Davis

- ▷ Microkernel-Based Run-Time Support for Distributed Programming Languages
- ▷ A Practical Workbench-Style Verification Environment for Reasoning about Distributed Programs
- ▷ Verification of Distributed Systems: A Methodology for Vertical Composition
- ▷ Verification of Distributed Systems: A Methodology for Microkernel/Server Composition
- ▷ Intrusion Detection for Large Networks
- ▷ Specification-Driven Testing of Programs

Advantages of Distributed Languages

- ▷ Expressiveness
- ▷ Conversation Management
- ▷ Interface Definition
- ▷ Type Checking
- ▷ Portability
- ▷ Error and Exception Handling
- ▷ Optimization
- ▷ Verification

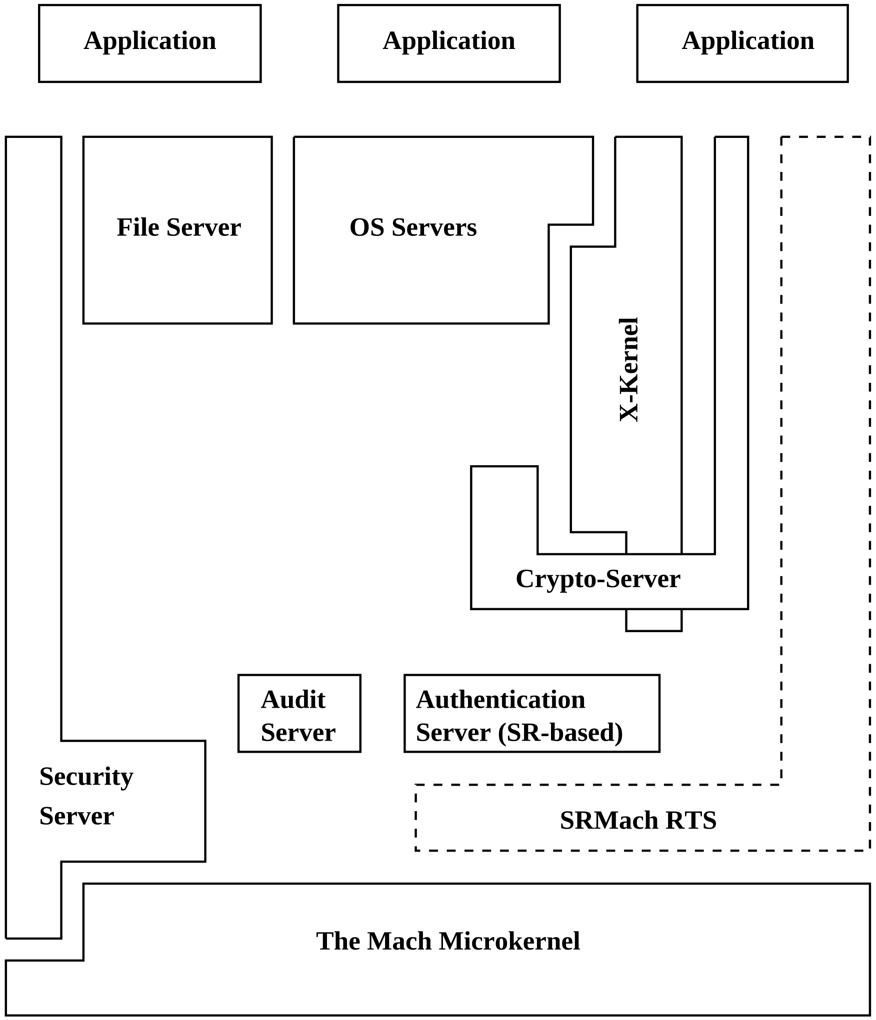
Microkernel Support for Distributed Languages

- ▷ Take advantage of microkernel technology to efficiently implement new programming models.
- ▷ SRMach - SR run-time support for the Mach microkernel
  - → The SR concurrent programming language (Arizona and UC Davis)
  - → The Mach microkernel (CMU) and Mach4 (Utah)
- ▷ Themes
  - → Making it practical to write distributed programs (especially system-level programs) in a distributed programming language.
  - → Exploring the language/kernel interface.
  - → A new approach to implementing run-time support.

Run-Time Support

- ▷ Basic Functionality (Performance Critical)
  - → Map language mechanisms to microkernel primitives
  - → Memory management
  - → I/0 (user input/output and file system access)
  - → Network Access
- ▷ Advanced Functionality (High-Level)
  - → Distributed shared memory
  - → Distributed objects
  - → Fault tolerance
  - → Deadlock detection/avoidance

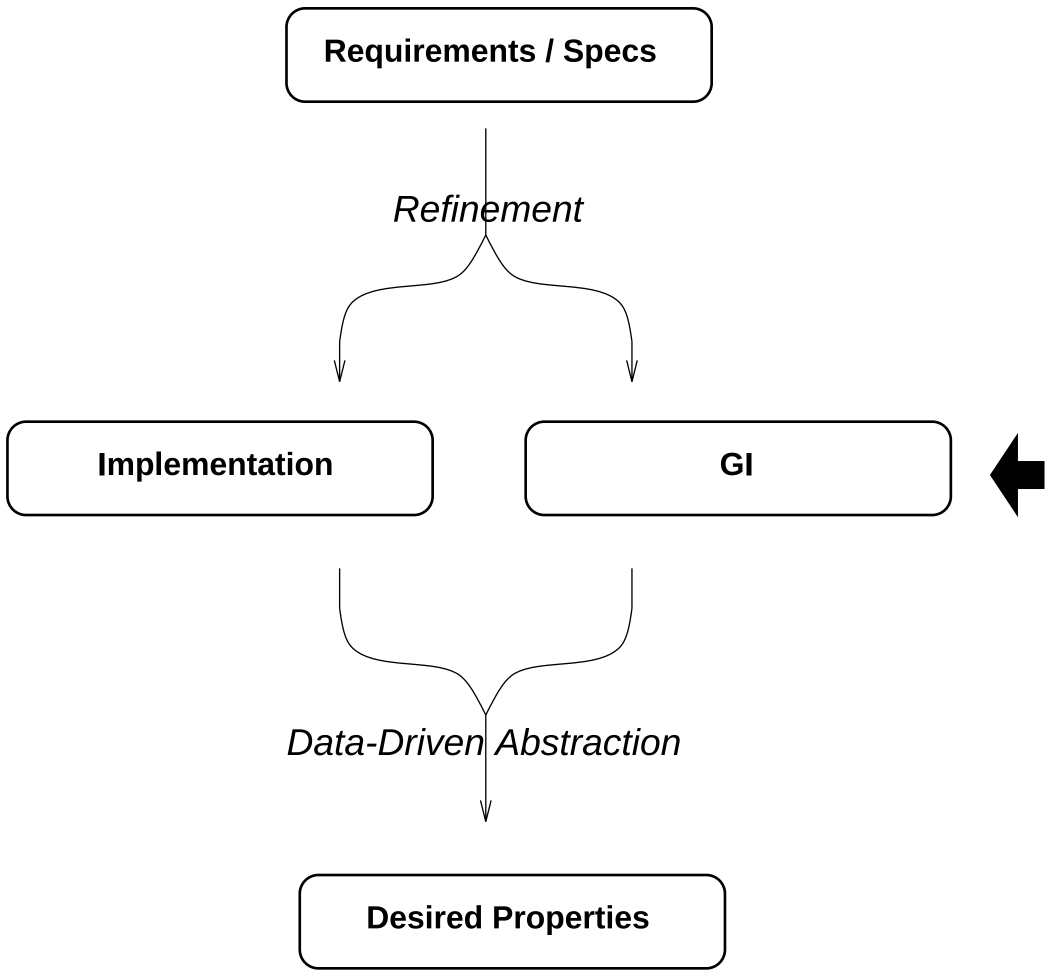
SRMach and Synergy



The Reasoning Environment Project

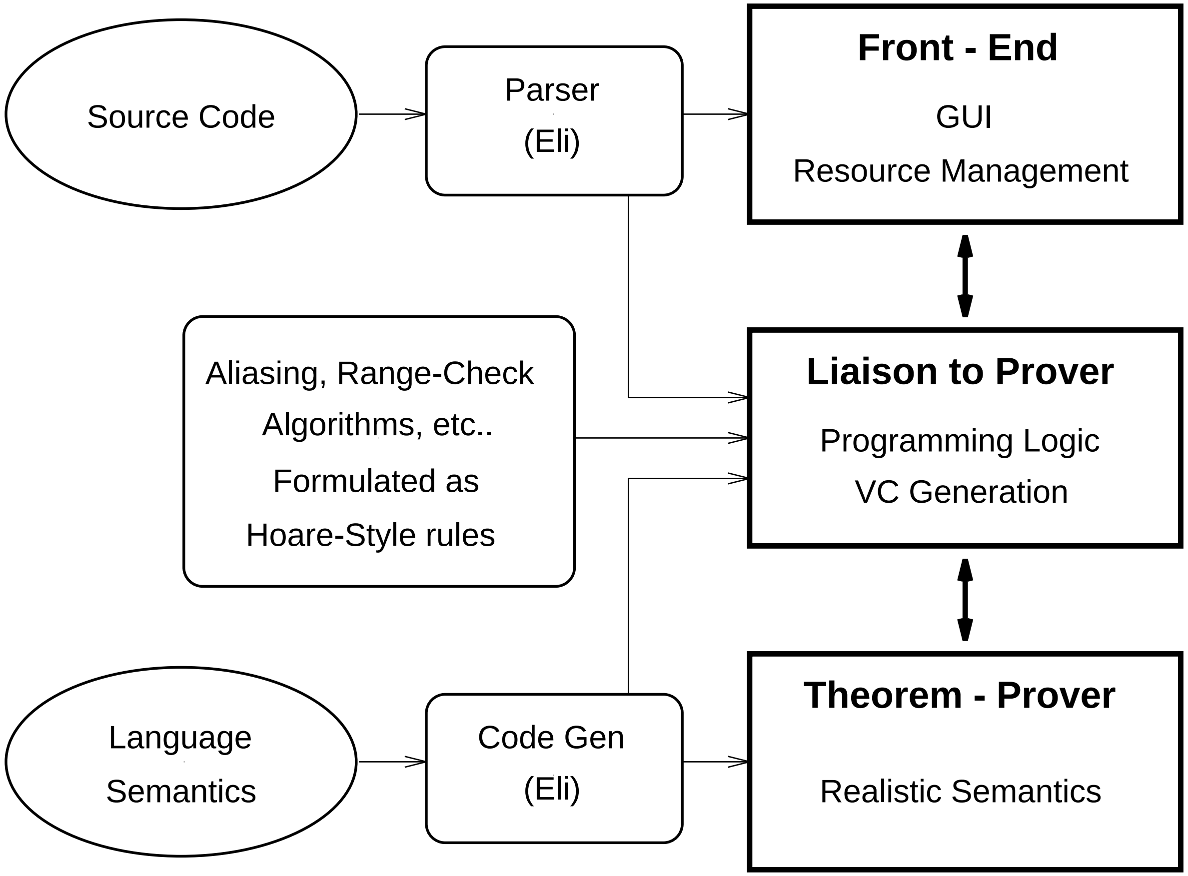
- ▷ Audience
  - → Programmers of High-Assurance Distributed Systems
- ▷ Philosophy of Code Reasoning
  - → View Theorem-Provers as the Assembly Language
  - → Recognize the Dissimilar Roles of People and Machines
    - ∗ Insight and knowledge of high-level structure
    - ∗ Extreme rigor and attention to endless details
  - → Formulate Semantic Models in line with Human's Operational View of Program Execution
- ▷ Research
  - → Workbench-Style Reasoning Environment for Distributed Programs

Reasoning Tools and the Software Lifecycle



* Handles Interference
* Reflects Programmer's View
* Allows Automation/Visualization
* Supports Human-Oriented Reasoning
* Scalable

Reasoning Environment Overview



Formulation Assistance
Adjustable Rigor

Code Assumption Discovery
Prover Script Generation
Prover Error Re-Phrasing

Soundness of VC Generation
Semantic "Correctness"

Silo Project: Formal Verification of Distributed Systems

▷ **Research Goals**

→ Develop a complete "full system" specification and verification for a distributed system from the high-level distributed programming language to the hardware by layered formal proofs in HOL.

→ Develop a formal methodology for verifying kernel/server based distributed operating systems.

→ Apply the methodology to the verification of a small but real distributed system: programming language, compiler, operating system, and network.

▷ **Challenges**

→ Concurrency and nondeterminism

→ Security

→ System scalability and composition

→ Reasoning about system changes and extensions

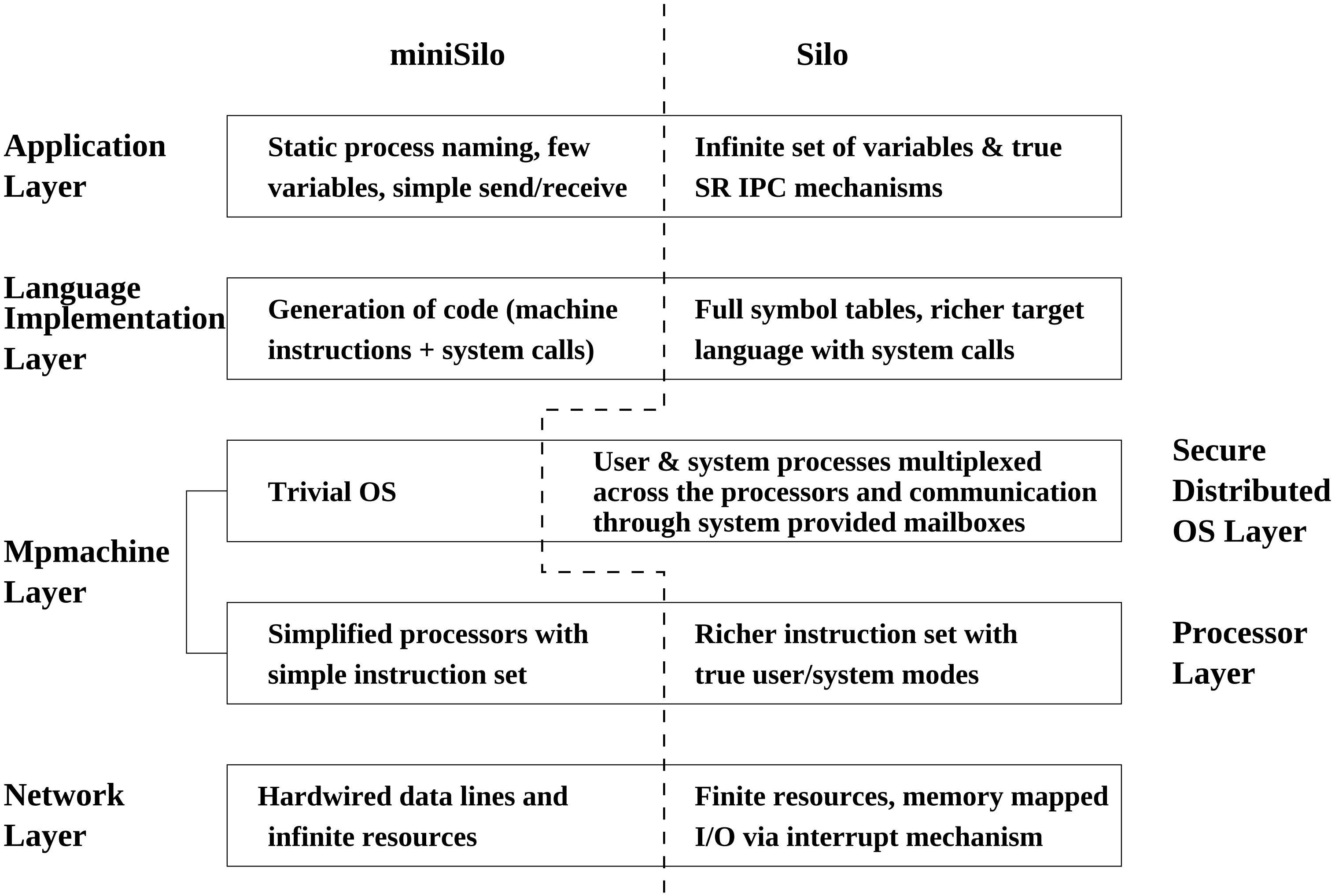
Previous Research Efforts at UCD

- **Hardware Verification**
  - → Vertical Composition
    - ∗ Formal Verification of Generic Interpreters
    - ∗ Formal Verification of Microprocessor Designs – "UCD" Viper
  - → Horizontal Composition
    - ∗ Formal Verification of a Floating Point Coprocessor and its Composition with a Central Processing Unit
- **Computer Security**
  - → Modular Verification of Security Properties Using Generic Components
- **Distributed Programming Language**
  - → Design and Development of the SR Language
  - → Formalization of SR Semantics – Partial Axiomatic Semantics for Some Statements

Overview of Silo Project

Layer			Sub-layer
Application Layer	Interface:	microSR & a Hoare Programming Logic	
	Implementation:	Soundness Proofs	
Language Implementation Layer	Interface:	Formal Semantics of microSR including IPC	
	Implementation:	Compilation Functions	
Mpmachine Layer	Interface:	IPC System Calls	Secure Distributed OS Layer
	Implementation:	OS Kernel & Servers (including security servers)	
	Interface:	Instruction Set	Processor Layer
	Implementation:	CPU & Memory	
Network Layer	Interface:	Host-host Communication Network	
	Implementation:	Network Controllers & Interconnect	

Incremental Approach: from MiniSilo to Silo



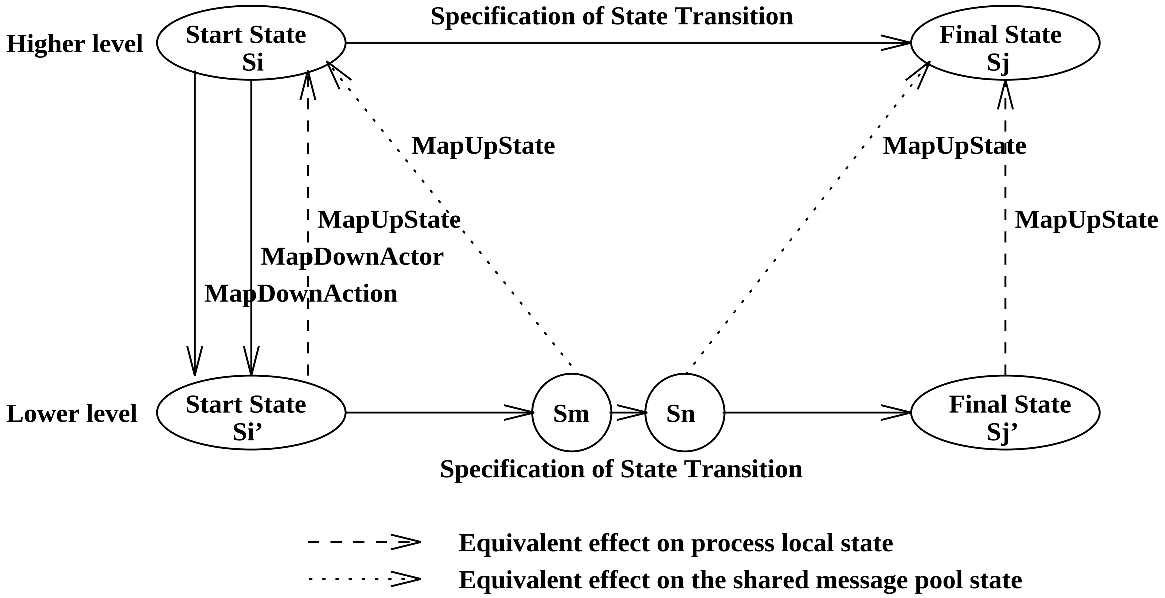
Methodology of the Layered Proof

▷ State transition models: Atomicity, concurrency, and nondeterminism

▷ Framework for semantic specifications

▷ Necessary mappings and the relationship of equivalent state transition effect between layers

▷ Generic proof obligation for the implementation correctness



Mechanization of a Programming Logic

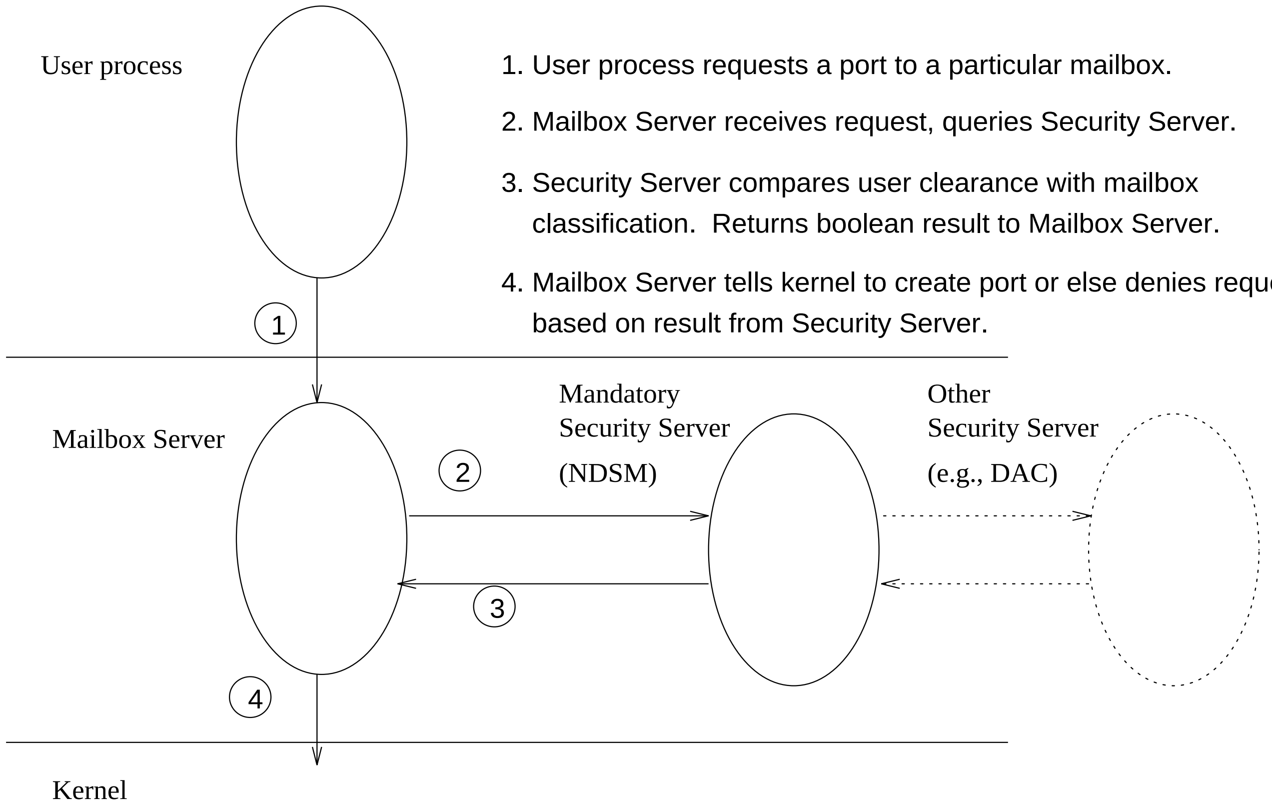
- ▷ The distributed microSR language and its assertional language are formalized in HOL.
- ▷ Axioms and Inference Rules, including the rule for the program composition (the rule for Cobegin), are HOL theorems proved as the logical implication of the semantic specification.
- ▷ Semantic consistency is guaranteed by using the same semantic specification for the soundness proof of the programming logic and the correctness proof of the language implementation. equivalent state transition effect between layers
- ▷ This programming logic supports the development of verified verification tool.

A Verified Secure, Distributed Operating System

- **Synergy and Mach-like Architecture**
  - → Kernel implements message passing, enforces process separation and schedules processes fairly.
  - → Servers enforce object management and other system policies — including access control policy — and implement distributed system.
  - → Processes invoke operating system services via message passing.
- **Designed for Verifiability**
  - → Process separation and mailboxes ensure well-defined interfaces for modular specifications.
  - → Fair scheduling allows specification of liveness properties.

Security in Silo

- ▷ Access control policy implemented by servers.
- ▷ Hierarchical security policies implemented by chaining servers.



1. User process requests a port to a particular mailbox.
2. Mailbox Server receives request, queries Security Server.
3. Security Server compares user clearance with mailbox classification. Returns boolean result to Mailbox Server.
4. Mailbox Server tells kernel to create port or else denies request based on result from Security Server.

Verification Methodology

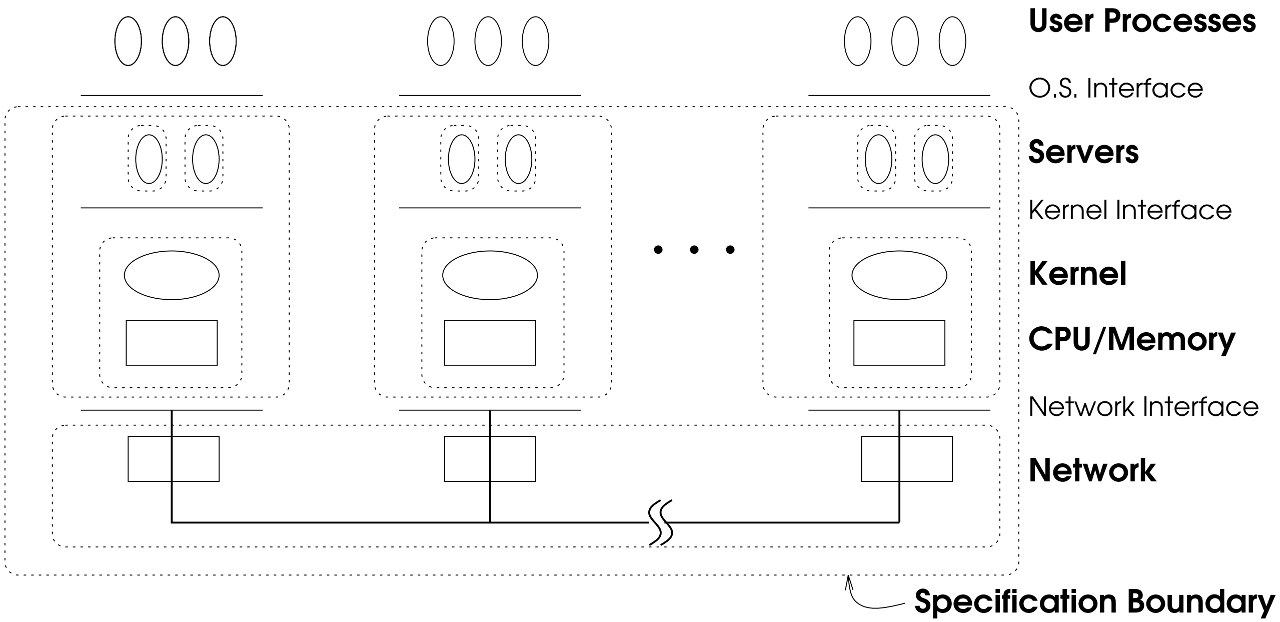
▷ Mechanizes Abadi-Lamport compositional methodology.

▷ Specifications consist of both safety and liveness properties.

▷ Kernel, servers specified and verified independently.

▷ Compose modular specifications to satisfy operating system specification.

▷ Scalable — verify subsystems separately and compose incrementally.



General Form of a Specification

- ▷ Represent system as finite or infinite sequence of atomic *state transitions*.
- ▷ State transitions occur due to *actions* (or *events*) performed by *agents*.
- ▷ Specification for a system is the set of all possible state transition sequences.
- ▷ Define set of sequences implicitly as conjunction of safety and progress properties.
  - → Safety properties: initial state and allowable state transitions.
  - → Progress properties: eventual occurrence of specific state transitions.

$$
\begin{aligned}
system\_spec = \{\ trace \mid\ & (good\_initial\_state\ trace) \\
& \wedge (good\_transitions\ trace) \\
& \wedge (progress\_is\_made\ trace) \\
\}\ &
\end{aligned}
$$

Concurrency and Composition

- ▷ In distributed system, have multiple agents who can execute non-deterministically.
- ▷ Many possible interleavings of events; many possible state-transition sequences.
  - → e.g., system with two agents, a_1 and a_2, each with a single possible state transition, e_1 and e_2, has two different possible state-transition sequences: e_1, e_2 and e_2, e_1
- ▷ Basic idea of composition is to generate complete system specification by taking conjunction of subsystem specifications.
  - → e.g., assuming a_1 and a_2 are independent, our system specification could be as follows:

$$
\begin{aligned}
system\_spec = \{\ trace \mid\ & (good\_initial\_state1\ trace) \\
& \wedge (good\_transitions1\ trace) \\
& \wedge (progress\_is\_made1\ trace) \\
& \wedge (good\_initial\_state2\ trace) \\
& \wedge (good\_transitions2\ trace) \\
& \wedge (progress\_is\_made2\ trace) \\
\}\ &
\end{aligned}
$$

System vs. Environment

- ▷ System's behavior may depend on actions by agents in the *environment*.
- ▷ System specification must consider the behavior of the environment without constraining the environment, or else the system will be impossible to build.
- ▷ System's behavior can only be expected to satisfy its specification when the environment's behavior satisfies certain assumptions.
- ▷ Can express relationship between system and environment using implication.

$$
\begin{array}{rcl}
system\_spec & = & \{\ trace \mid \begin{array}[t]{c}
\quad (good\_initial\_state\ trace) \\
\wedge\ (good\_env\_transitions\ trace) \\
\wedge\ (env\_makes\_progress\ trace)) \\
\Longrightarrow \\
(good\_sys\_transitions\ trace) \\
\wedge\ (sys\_makes\_progress\ trace)
\end{array} \\
& & \}
\end{array}
$$

Composition Principle

- ▷ When specifications include the environment, straight conjunction cannot be used to compose specifications.
- ▷ To show that a system satisfies its specification if all its components satisfy their specifications, use the following Composition Principle:

  Let Π be composition of $\Pi_1, ..., \Pi_n$, and let the following conditions hold:

  1. Every component Π_i guarantees M_i under environment assumption E_i.
  2. The environment assumption E_i of each component Π_i is satisfied if the environment of Π satisfies E and every Π_j satisfies M_j.
  3. Π guarantees M if each component Π_i guarantees M_i.

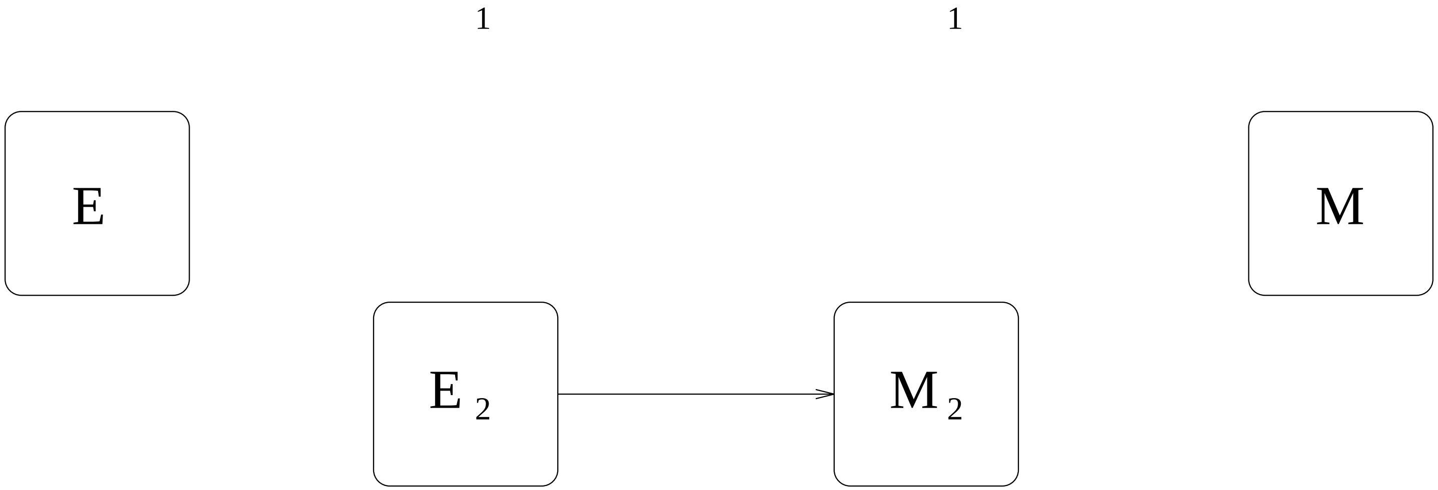
  Then Π guarantees M under environment assumption E.
- ▷ Despite the apparently circular reasoning, the principle holds when the environment assumptions are safety properties.

Composition of Specifications

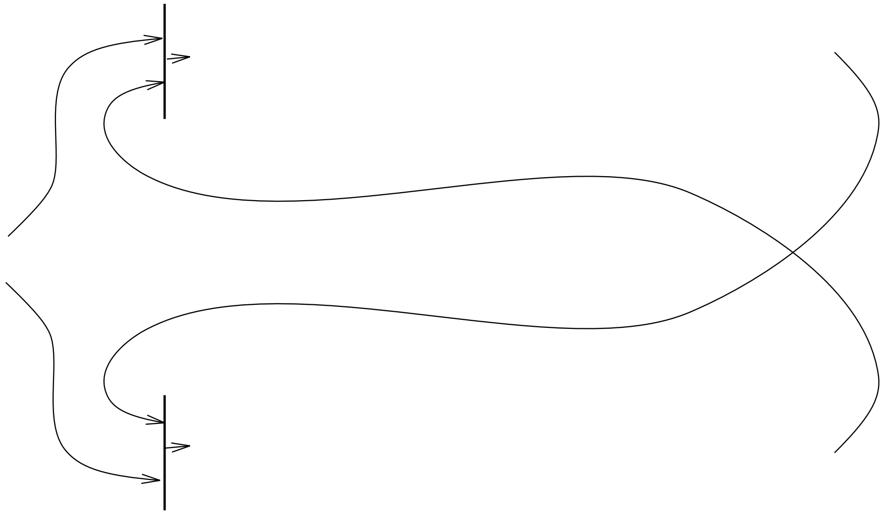
E = environment property

M = system property

$\rightarrow$ = implication

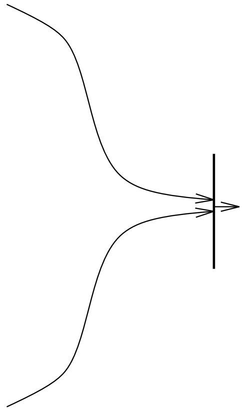


1. Definition of subsystems.



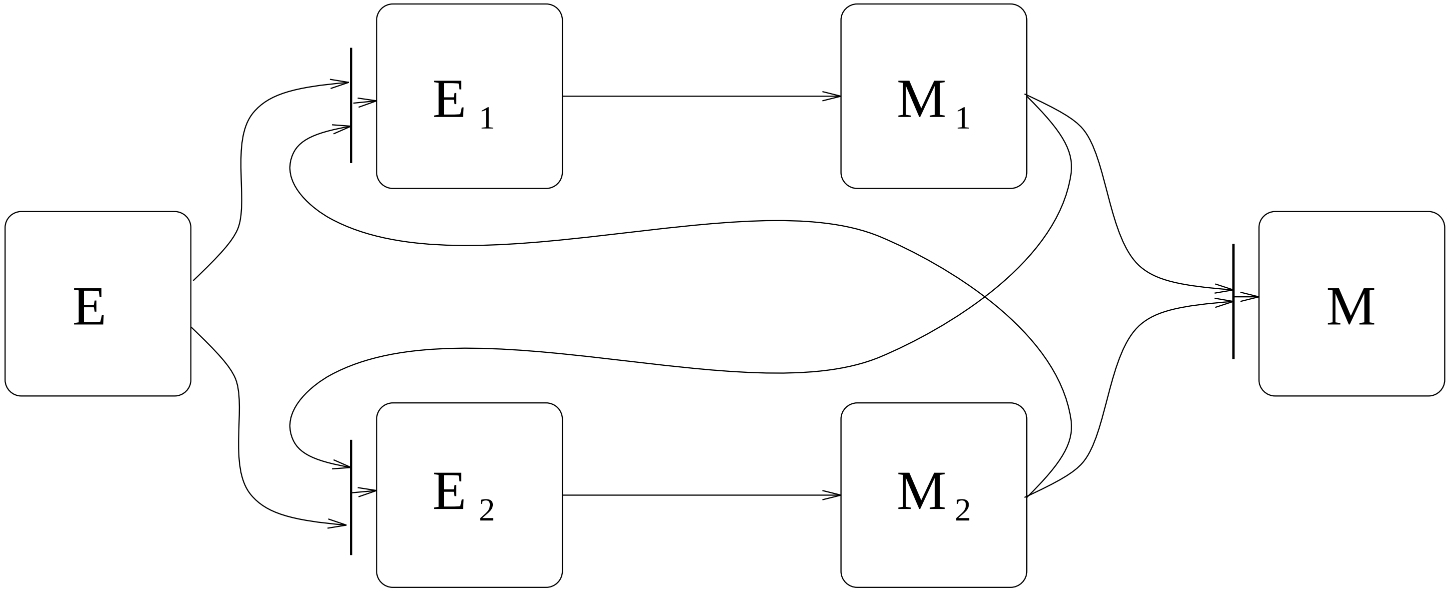
2. System environment and subsystems imply environments of other sybsystems.

| = conjunction



3. Subsystems imply complete system.

Composition of Specifications



E = environment property

M = system property

$\rightarrow$ = implication

| = conjunction

1. Definition of subsystems.

2. System environment and subsystems imply environments of other sybsystems.

3. Subsystems imply complete system.

Approaches for Retrofitting Security

▷ Static Analysis

→ Detect mistakes in system set up and configurations

→ e.g., COPS, SPI

▷ Intrusion Detection

→ Analyze audit data in real-time to detection ongoing intrusions

→ Anomaly Detection

- You know what is **NORMAL**
- Based on statistically profiled normal behavior of objects
- Detect abnormal behavior

→ Misuse Detection

- You know specifically what is **BAD**
- Based on previous attacks or obvious bad behavior
- Detect actions that match the misuse rules or signatures

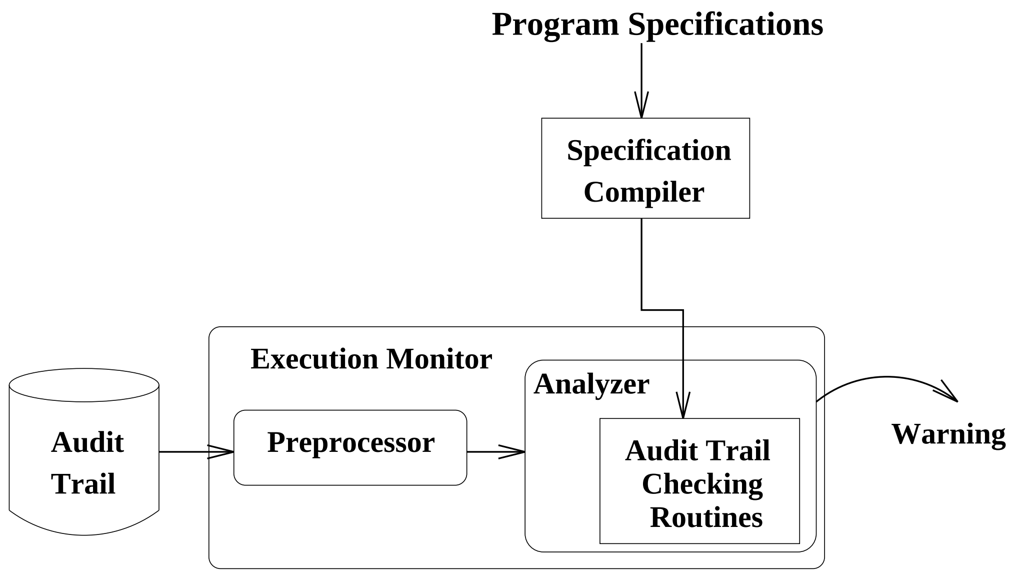
Specification-based Intrusion Detection

- ▷ You know what is **GOOD**
- ▷ Write partial specifications for objects
  - → Capture the desirable (good) behavior
  - → Security-relevant portion only
  - → Monitorable using audit data
  - → Based on the functionalities of objects
- ▷ Reason about the overall behavior of system
  - → Specifications of individual objects and their interactions
  - → With respect to a system security policy (policy-driven specification)
  - → Serve as a proof-checking process for the specifications

Implementation

- ▷ SunOS Basic Security Module (BSM) and Solaris 2.x
- ▷ Specification Compiler translates specification into Audit Trail Checking Routines (in Perl)

Implementation (cont.)



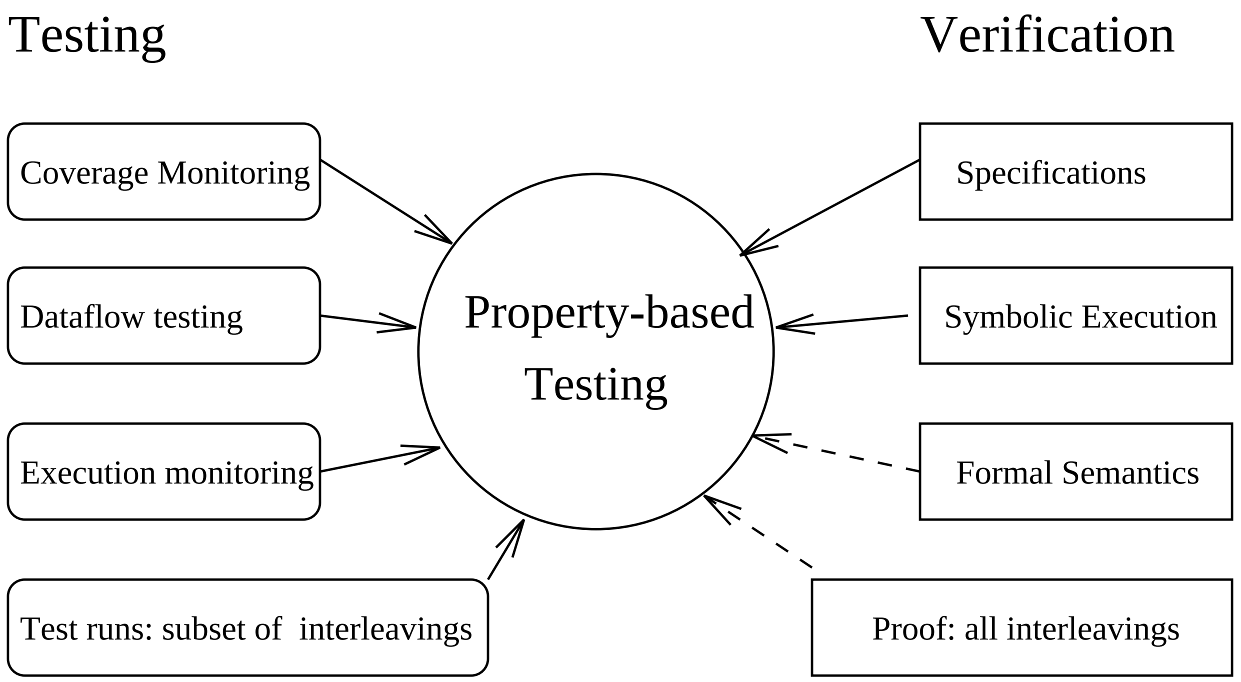
Motivation: Thesis

- ▷ Test a program with respect to only certain properties:
  - → Use properties to generate test data
  - → Use specifications of properties as test oracles
  - → Deal with only a small part of the program
    - ∗ Focusing on properties enables effective use of existing testing technologies: e.g. coverage monitoring
    - ∗ Properties can be used for "divide and conquer" testing
- ▷ Examples of properties:
  - → Improper use of a system call
  - → Undesirable functional behavior
  - → Time of Check, Time of Use generic flaw
  - → Safety properties: e.g., in flight control systems

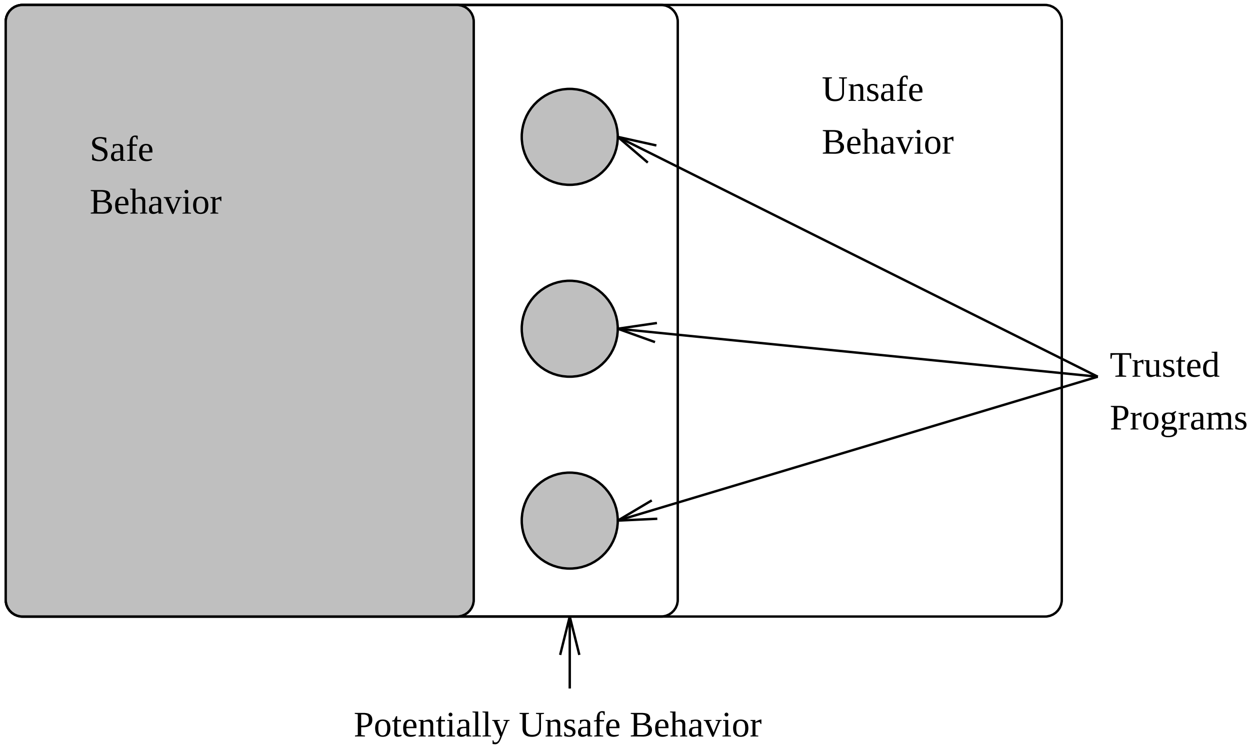
Motivation - Why security as our initial application

- ▷ Security is a rich source of programs with lots of errors that have survived extensive testing.
- ▷ Slicing is effective for security, because security properties frequently are only part of a program's full behavior.
- ▷ Security specifications for most privileged programs and servers are surprisingly concise and easy to produce.
- ▷ Many security problems arise from "generic flaws".
- ▷ Audit trails facilitate security-relevant execution monitoring

Overview


Testing
Verification
Coverage Monitoring
Dataflow testing
Execution monitoring
Test runs: subset of interleavings
Property-based Testing
Specifications
Symbolic Execution
Formal Semantics
Proof: all interleavings

Motivation: The privileged program problem

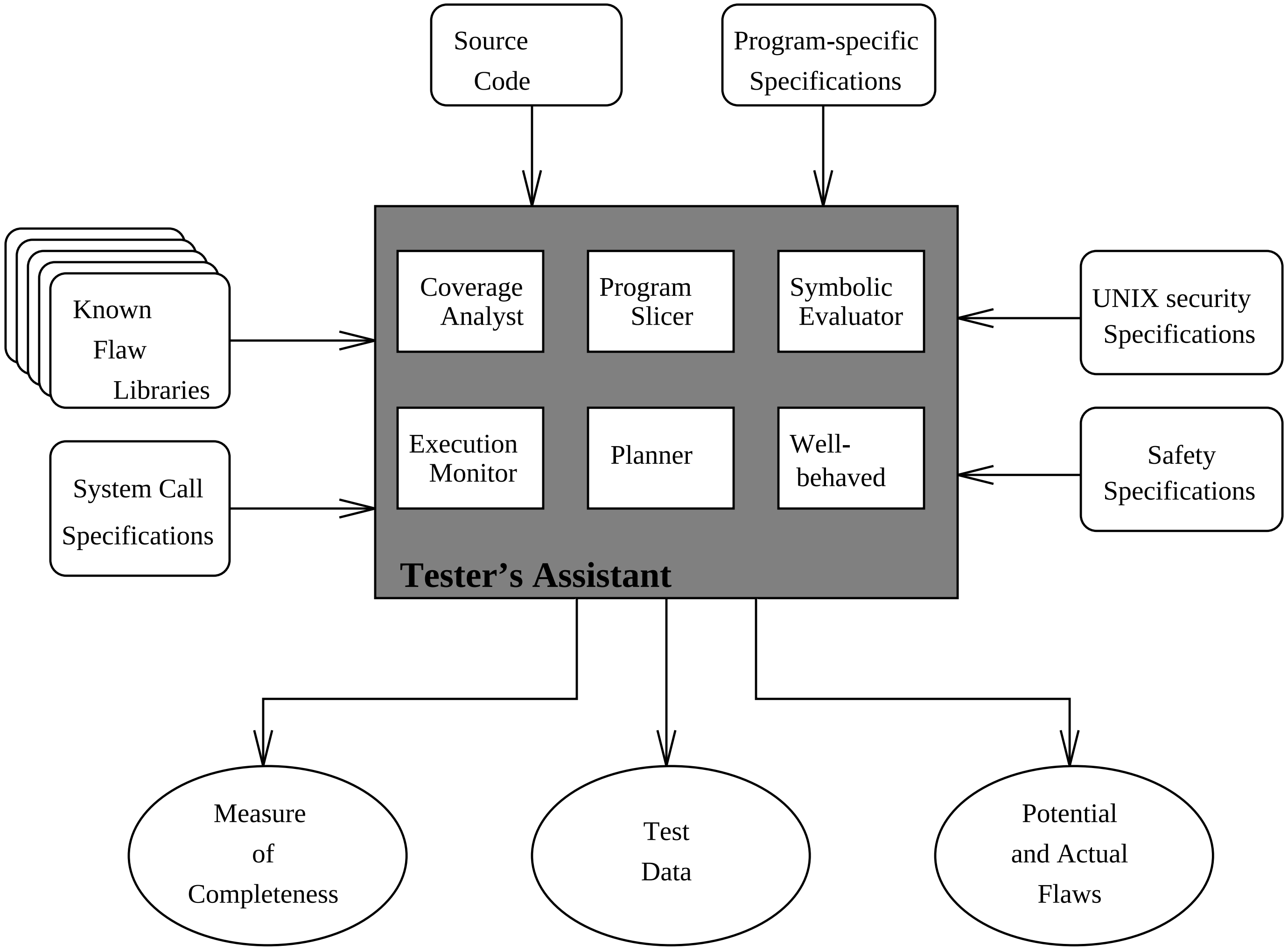


Trusted programs run as root, thus potentially can cause damage.
Therefore, they must be validated to be safe.

Approach: Overview

- ▷ Develop specifications relevant to testing for specific flaws.
- ▷ Obtain slicing criteria from specifications of properties.
- ▷ Use program slicing to reduce problem size.
  - → Check for well-behavedness to assure accurate slices.
- ▷ Use other methods to generate initial test data.
- ▷ Use execution monitor to evaluate test runs.
- ▷ Measure code coverage of test runs.
- ▷ Automate test data generation for uncovered code.

The Tester's Assistant



Summary

- ▷ Achieving high assurance for distributed systems and networks is challenging.
- ▷ A high-level distributed programming language aids in created distributed applications.
- ▷ High assurance system functions can be relegated to run-time support.
- ▷ Specialized verification support for the distributed languages is critical.
- ▷ Vertical and microkernel/server composition of systems facilitates verification and re-verification.
- ▷ Formal specifications have a role in retrofit assurance technologies (e.g., intrusion detection and testing.)