Sponsored by the U.S. Department of Defense © 2005 by Carnegie Mellon University

From Finding Bugs to Certifying their Absence

Sagar Chaki

September 21, 2006

Motivation

We are increasingly reliant on software despite continuing problems controlling software quality.

Human and financial

- Therac-25 X-Ray machine
  - At least 6 known incidents with 5 fatalities
- Ariane 5 loss (~$640 million)
- Pentium recall (~$450 million)

Recurring maintenance costs – security patches

Consumer confidence

- Airport shutdown due to malfunction in baggage X-Ray

Development costs and delayed time to market

© 2005 by Carnegie Mellon University

Finding Bugs

Experience points us to common causes of bugs

- Improper use of shared resources
- Failure to comply with API policies
- Poor coding practices
- Buffer overflow

Nevertheless, bugs remain undetected and with unknown effects despite expensive quality assurance practices

- Effects may not be observed, e.g., during testing
- Even if observed, difficult to reproduce and eradicate

© 2005 by Carnegie Mellon University

Bug-Finding Technology

For generic problems – good, but only partial solutions

- Compilers
- Commercial code analysis tools

For specific problems

- Testing – incomplete coverage, focus on effect
- Code inspection – inconsistent results

We need a means to get consistent results with more complete coverage when looking for known causes

© 2005 by Carnegie Mellon University

Software Model Checking

Key ideas:

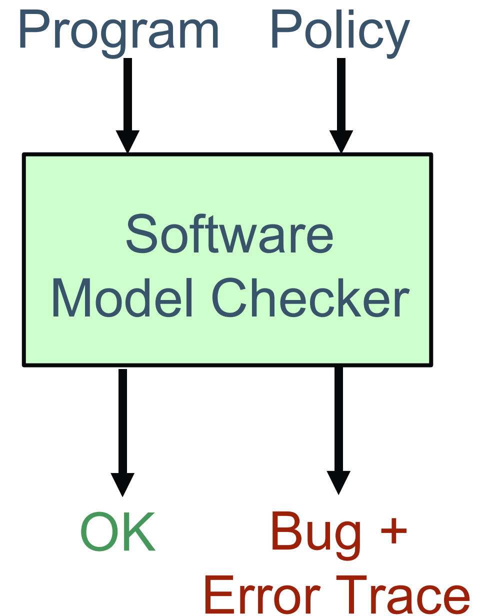
- Exhaustive analysis
- Automated abstraction of conservative finite models
- Policy independent

Strengths:

- Usable at industrial scale*
- Packages tool complexity and deep domain expertise

Limitations:

- Scale will always be a factor
- Asymmetric confidence



* http://www.microsoft.com/whdc/devtools/tools/sdv.mspx

© 2005 by Carnegie Mellon University

Model Checking Successes

Typically applied by experts for specific products

- Cache coherence protocols
  - IEEE Futurebus+
- Microprocessors
  - Intel, IBM, Motorola
- Telecommunications
  - Lucent PathStar switch

Microsoft Static Driver Verifier (SDV) is a notable exception and a good example of big impact, broad application of software model checking

© 2005 by Carnegie Mellon University

Microsoft Static Driver Verifier

Goal: reduce “blue screen of death” occurrences.

Many failures traced to improper interaction by third-party device drivers with the Windows kernel.

Solution: identify correct interaction policies and automate violation detection with software model checking.

Restriction to known policies allowed optimizations for scale and black box use by any engineer.

SDV has been used as part of driver certification since 2003 and is now publicly available

- http://www.microsoft.com/whdc/devtools/tools/sdv.mspx

© 2005 by Carnegie Mellon University

Our Experience

Robotics line communications library
- Widely understood domain
- Able to identify meaningful policies
- Found several important bugs

Embedded Operating System (6000 LOC)
- Less expertise, more general policies
- Still successful at finding bugs

Focused application of model checking for specific issues yields practical improvements
- Buffer overflow identification with fewer false positives

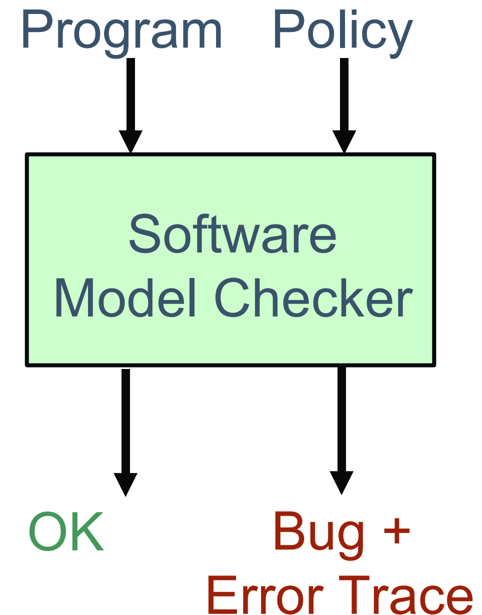
© 2005 by Carnegie Mellon University

Verification and Trust

Software model checking alone is inadequate for safety-critical systems

- Checker itself a complex system
  - Policies pass because checker is buggy
- Difficult to detect this situation
  - Checker only says "OK"
  - Have to trust the checker

Goal: Move the model checker out of the Trusted Computing Base (TCB)



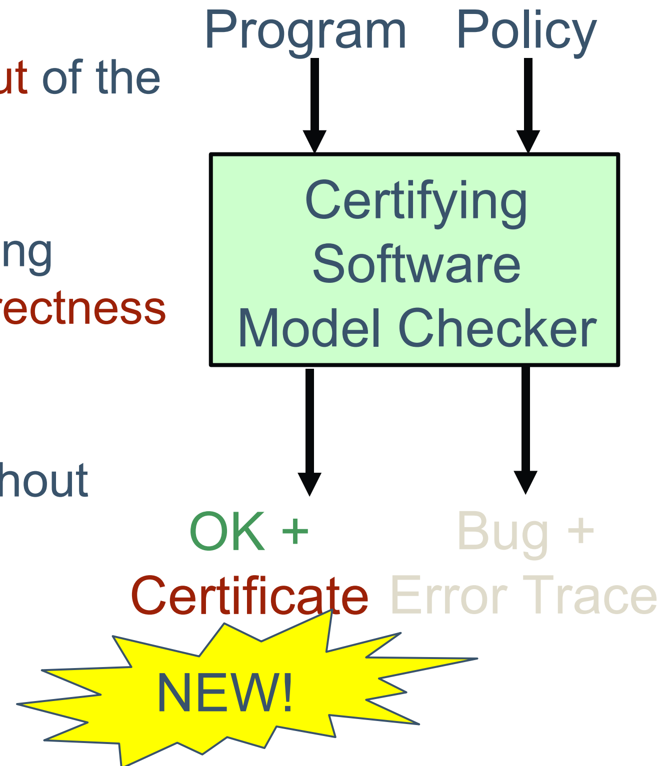
© 2005 by Carnegie Mellon University

From Verification to Certification

Have to trust model checker if “OK”

Goal: Move the model checker out of the Trusted Computing Base (TCB)

Idea: Use certifying model checking
• Model checker generates a correctness certificate with “OK”
• Certificate can be validated independently and objectively without trusting the model checker
• Symmetric confidence



© 2005 by Carnegie Mellon University

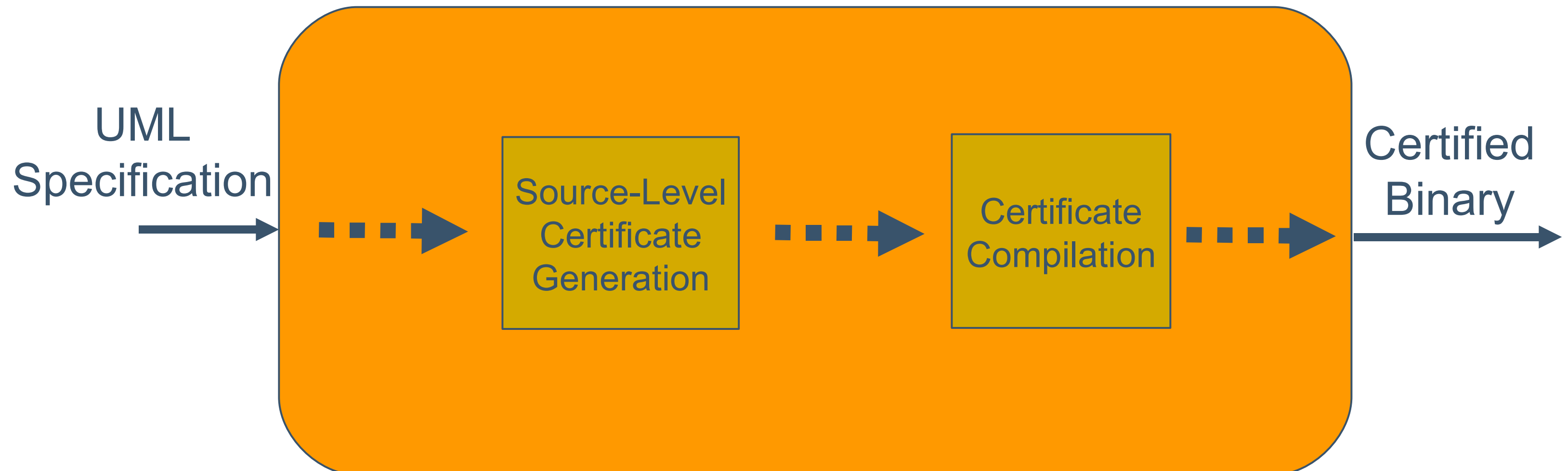
Certifying Binaries

Certifying software model checking certifies source code

- Still have to trust the compiler – another complex system

Can eliminate the compiler from the TCB

- By "compiling" the certificate generated by the model checker
- Computationally inexpensive



© 2005 by Carnegie Mellon University

Rest of the Talk

A framework to generate certified assembly code from UML specifications in a completely automated manner

- Step 1: Certify source code
- Step 2: Push certification to assembly code
- No need to trust the mechanisms (humans or tools) involved in the certification process

Implementation and experimental validation

- Demonstration to follow later on in the presentation

© 2005 by Carnegie Mellon University

Certifying a Maze

Finish

Start

Breadcrumbs
+
Arrows
=
Certificate

Certificate can be validated objectively

© 2005 by Carnegie Mellon University

Certifying a Program

The program is like a maze

- Can be certified by the following two artifacts

(1) Invariants

- Conditions that always hold during execution
- Similar to the breadcrumbs

(2) Proof of the following facts:

- Each invariant implies the next
- Invariants collectively imply the policy
- Similar to the arrows

Verification Condition (VC)

Invariants + Proof = Certificate

- Certificate can be validated objectively

© 2005 by Carnegie Mellon University

Certificate Immune to Some Changes

Finish

Start

Original Certificate

© 2005 by Carnegie Mellon University

Certificate Immune to Some Changes

Finish

Start

Maze changed

Program changed

© 2005 by Carnegie Mellon University

Certificate Immune to Some Changes

Finish

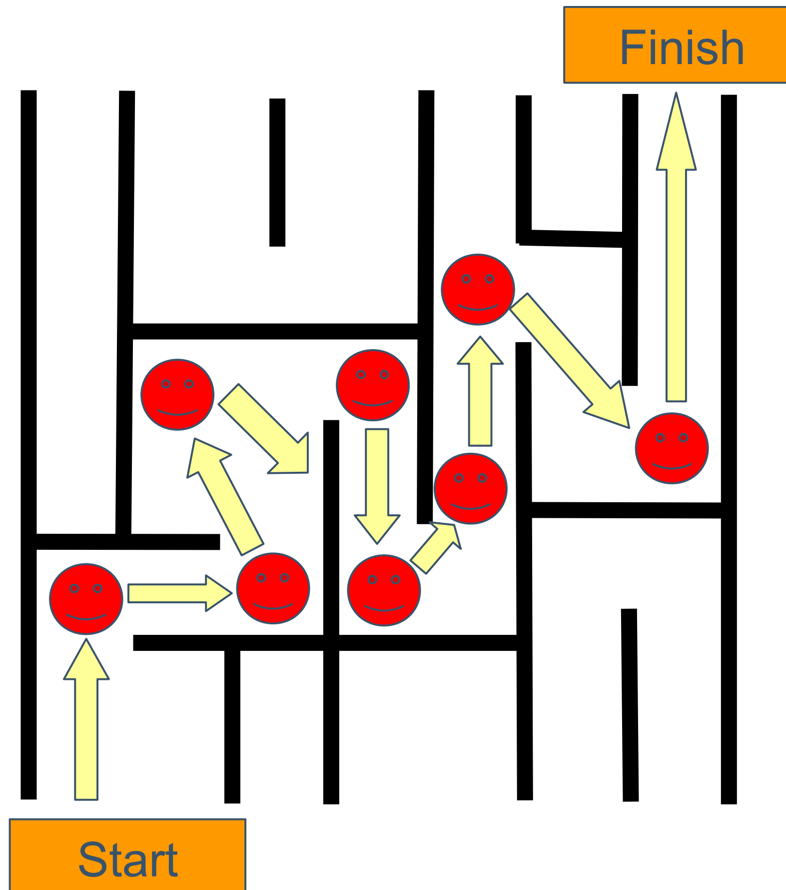
Maze and Certificate both changed

Program and Certificate both changed

Start

© 2005 by Carnegie Mellon University

Certificate Invalidated by Other Changes



Certificate changed

© 2005 by Carnegie Mellon University

Certificate Invalidated by Other Changes

Finish

Maze changed

Program changed

If the certificate is valid then there is a path from Start to Finish

If the certificate is valid then the program satisfies desired policy

Start

Soundness Theorem

© 2005 by Carnegie Mellon University

Certification Protocol

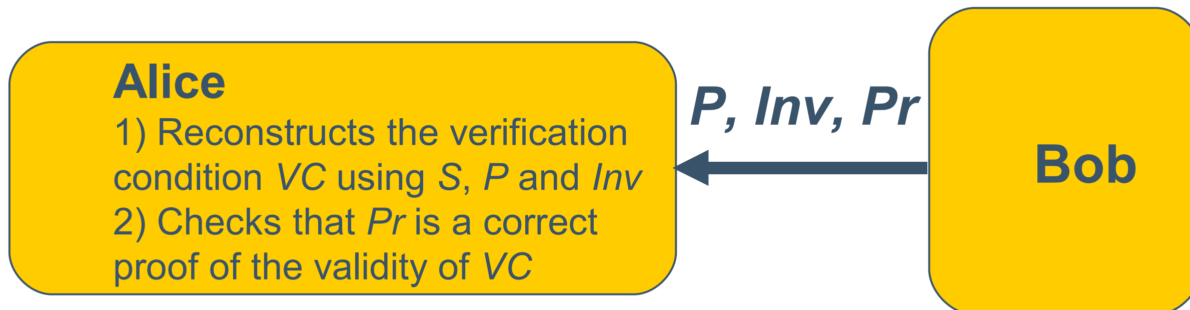
Bob wants to convince Alice that his program *P* satisfies a policy *S*

- Policy *S* may or may not be supplied by Alice

Bob supplies Certificate = Invariant (*Inv*) + Proof of VC (*Pr*) along with *P*

Alice checks that the certificate is correct

- Typically much easier than constructing the certificate



The *VC* construction procedure is public and deterministic. So Bob can construct the exact *VC* as Alice in order to produce a correct proof. Alice does not have to trust Bob.

© 2005 by Carnegie Mellon University

Basic Definitions

Two types of policy

- Safety : something bad never happens
  - a channel is never used before being secured
- Liveness : something good eventually happens
  - a channel is eventually secured

Invariant : A condition that is always true whenever the program execution reaches a specific program location

- Used to certify safety

Ranking function : A mechanism for imposing an ordering between various program states

- Used to certify liveness

© 2005 by Carnegie Mellon University

Example

```
#define CHANNEL_SECURED (n == 10)

void main()
{
        int n = 0;
        [n >= 0 && n <= 10]
        while(n < 10)
                [n >= 0 && n < 10]
                n = n + 1;
        [n == 10]
        assert(CHANNEL_SECURED);
        use_channel();
        return;
}
```

invariants

liveness: this assertion is eventually checked

safety: this assertion never fails

© 2005 by Carnegie Mellon University

Example

```
#define CHANNEL_SECURED (n == 10)

void main()
{
        int n = 0;
        [n >= 0 && n <= 10]
        while(n < 10)
                [n >= 0 && n < 10]
                n = n + 1;
        [n == 10]
        assert(CHANNEL_SECURED);
        use_channel();
        return;
}
```

invariants

safety policy below can be certified using invariants above

safety: this assertion never fails

© 2005 by Carnegie Mellon University

Certification Protocol

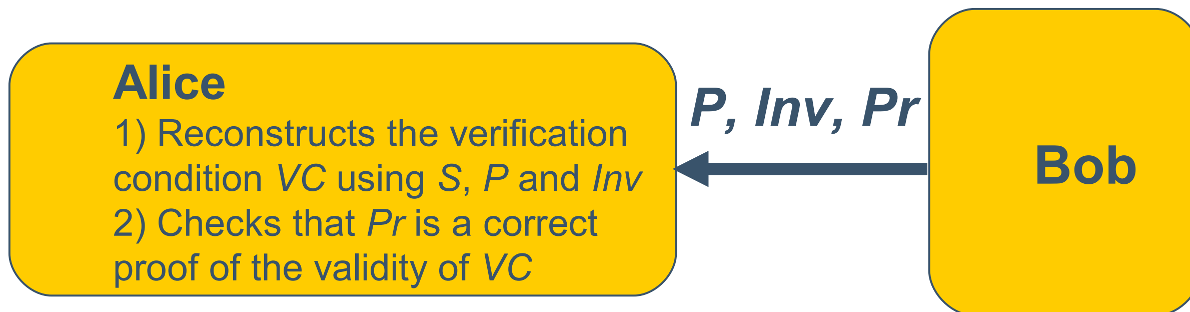
Bob wants to convince Alice that his program *P* satisfies a policy *S*

- Policy *S* may or may not be supplied by Alice

Bob supplies Certificate = Invariant (*Inv*) + Proof of VC (*Pr*) along with *P*

Alice checks that the certificate is correct

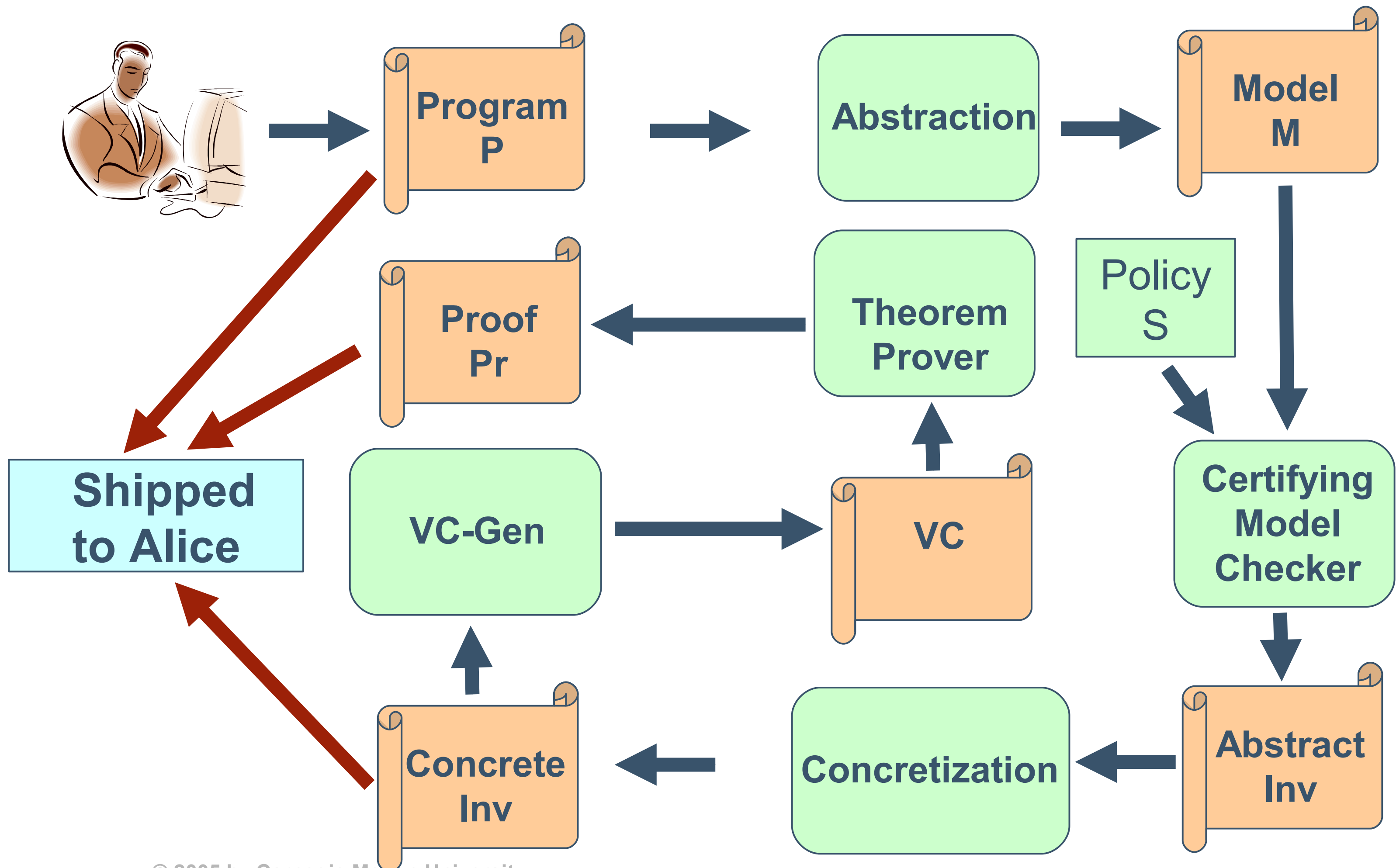
- Typically much easier than constructing the certificate



Let's look more closely at Bob

© 2005 by Carnegie Mellon University

Bob: Internal Details



© 2005 by Carnegie Mellon University

Bob: Internal Details

Program P → Abstraction → Model M

Policy S

Theorem Prover

Proof Pr

Certifying Model Checker

VC-Gen → VC

Concrete Inv ← Concretization ← Abstract Inv

Completely Automated

Shipped to Alice

© 2005 by Carnegie Mellon University

SAT-based Software Certification

Used Boolean satisfiability to generate compact proofs that certify safety and liveness policies of C programs

- were able to certify correct locking behavior in device drivers and an embedded OS
- proofs were 100 -- 100,000 times smaller

SAT-based Software Certification, S. Chaki, Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2006

© 2005 by Carnegie Mellon University

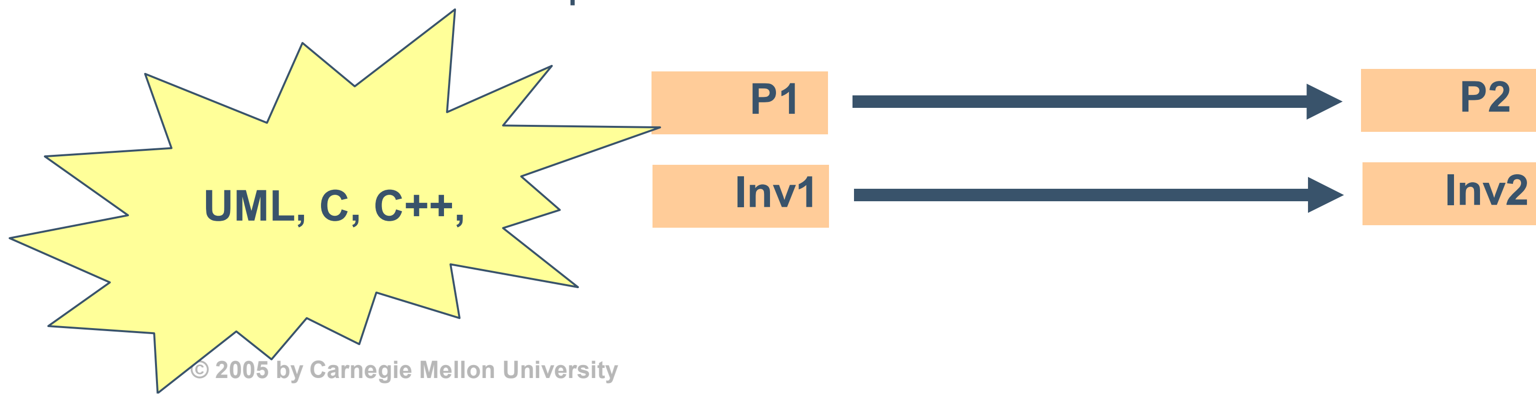
Certifying Binaries

A binary are just another kind of program. The difficulty is obtaining appropriate invariants from low-level instructions.

- We already know how to do this for high-level programs
- But binaries are obtained by compiling high-level programs
- Why don't we "compile" the invariants as well !

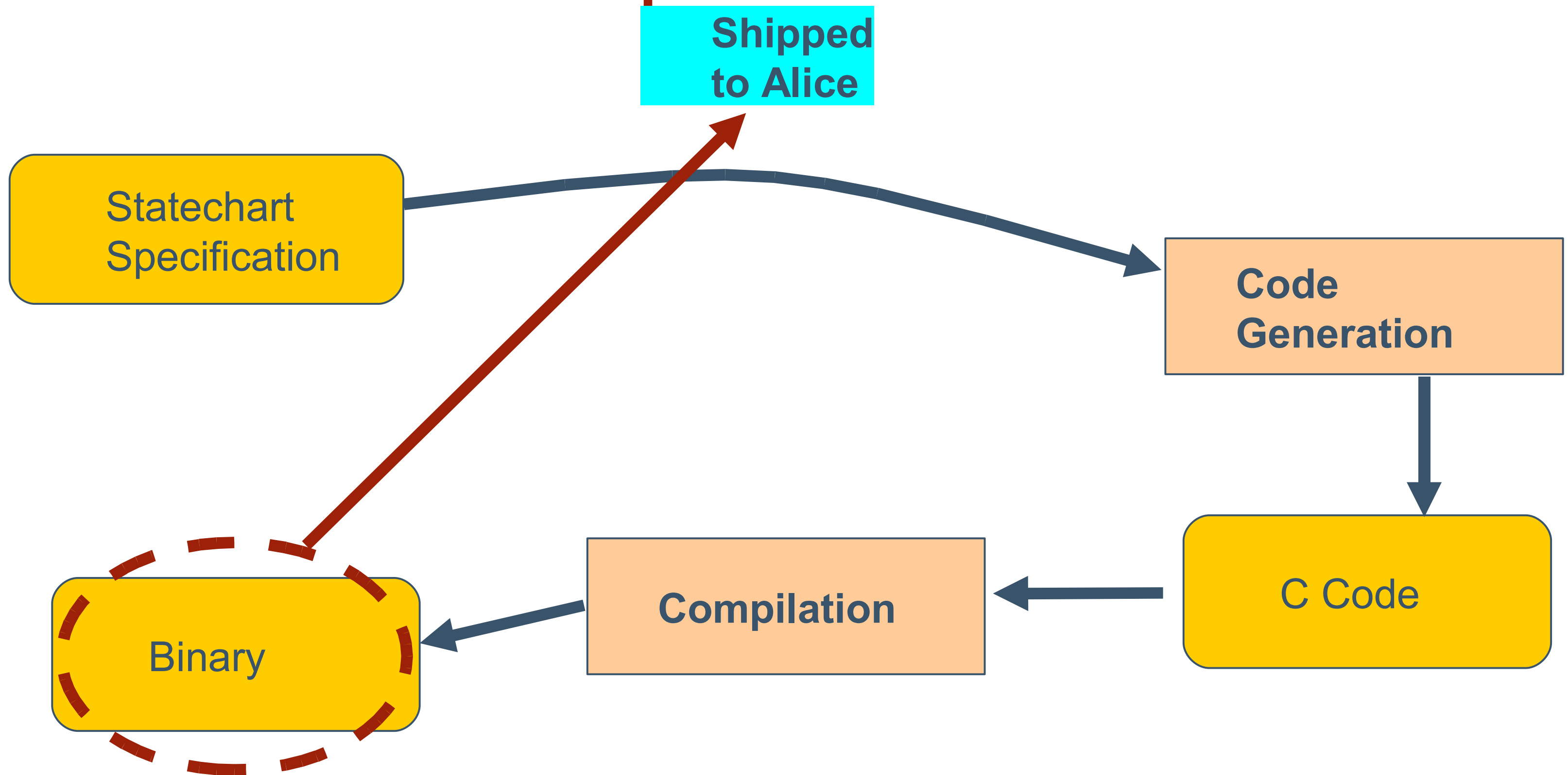
The key idea is invariant compilation

- Independent of the actual programming languages involved
- Need precise semantics for source and target languages, and the translation process



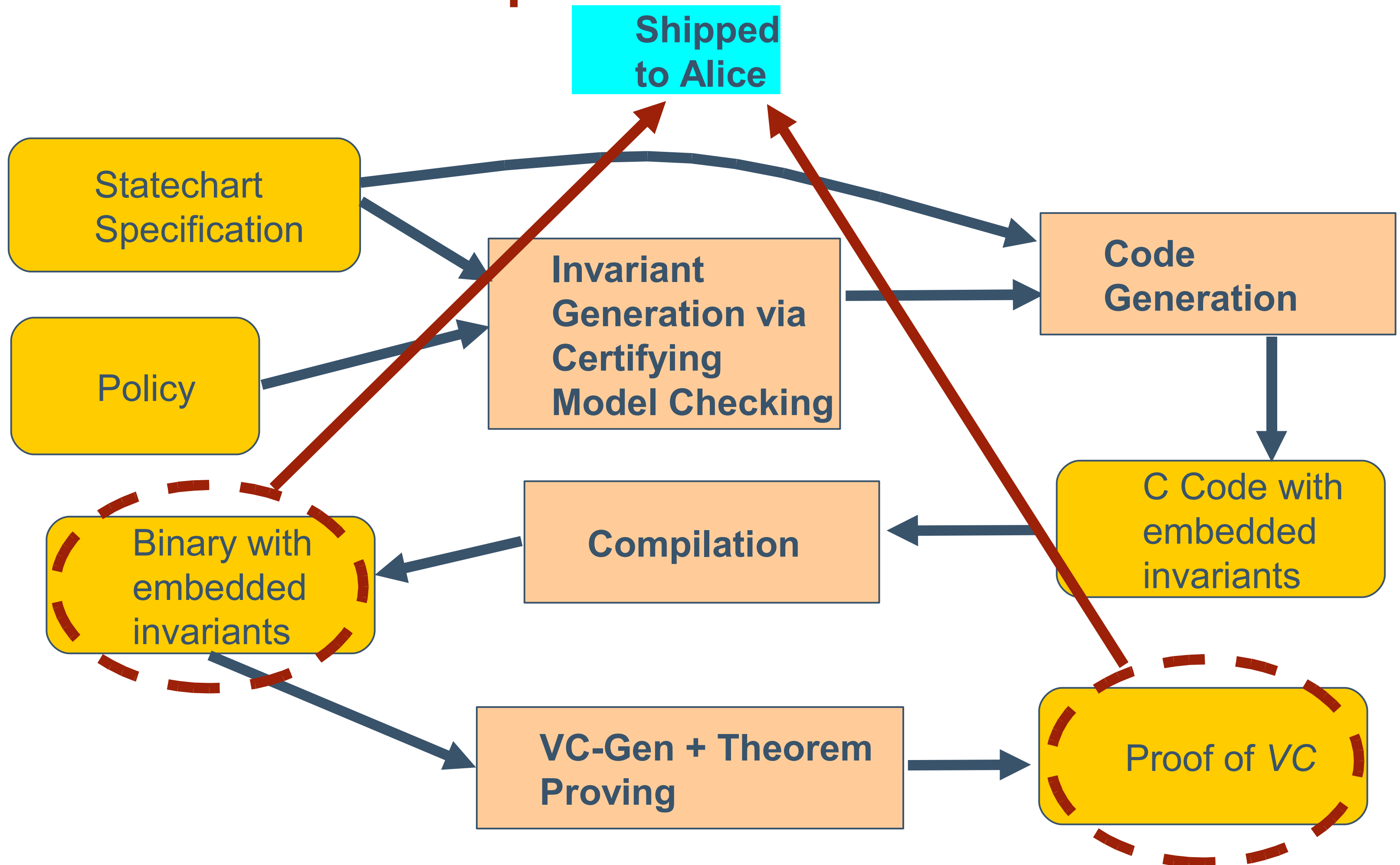
© 2005 by Carnegie Mellon University

Uncertified Component Binaries



© 2005 by Carnegie Mellon University

Certified Component Binaries



© 2005 by Carnegie Mellon University

Experience with an Embedded OS

Micro-C is an lightweight OS for embedded real-time applications

- 6000 lines of C code, supports multi-threading
- uses a “lock” to ensure mutual exclusion

We embedded (manually) appropriate invariants to certify that the OS follows a correct locking convention

Our prototype generated a binary (PowerPC assembly) with embedded invariants as well as the proof of the VC

- used a standard gcc cross compiler
- only default compiler optimizations were allowed

© 2005 by Carnegie Mellon University

Experience with Embedded OS

Certified the executable of Micro-C Operating System for correct ordering of lock and unlock operations

- Executable size = 335 KB
- Certificate size = 12 KB

Small trusted computing base (TCB) source code size

- Translator to SAT = 445 KB
- SAT proof checker = 25 KB

Certifying model checker and compiler eliminated from TCB

- About 20,000 KB source code size
- Over 40 times improvement

© 2005 by Carnegie Mellon University

Experience with “TAR”

Program for archiving files

- We used the implementation in the Plan9 OS

Certified that a certain buffer can never be overflowed when the program is executed

- Annotations were added manually
- Applications to certifying software security

Still an application of translating a certified source to a certified binary

© 2005 by Carnegie Mellon University

Demo : Generating Certified Components

A component that manipulates an integer variable “i”

Policy: The value of “i” should never become negative

The demo consists of three stages:

7. We will generate a certified binary

9. We will modify the binary such that the policy is still obeyed. We will show that the altered binary can still be certified.

11. We will modify the binary such that the policy is violated. We will show that the modified binary can no longer be certified.

© 2005 by Carnegie Mellon University

Key Results/Conclusion

Component certification involves two broad activities

- invariant generation and invariant compilation

We have a prototypical implementation that was successful in delivering a proof of concept

- Compiled invariants that certify proper locking discipline for an embedded OS and absence of buffer overflows in a commonly used program
- Generated certified binary for a simple illustrative component specification
- Our implementation also supports ranking functions for certifying liveness policies.

Invariants and ranking functions lead to a basis of objective confidence in the behavior of untrusted software

© 2005 by Carnegie Mellon University

Future Directions

The theoretical foundations have already been laid. There is an urgent need for broader transition to practitioners.

- Organizations who benefit most from this technology
- Medical device industry, auto industry
- Third-party supplied components

“Proof engineering” decisions have to be agreed upon

- This calls for concerted standardization effort
- Stakeholders who can formulate and enforce formal certification standards

© 2005 by Carnegie Mellon University

Questions?

http://www.sei.cmu.edu/staff/chaki
chaki@sei.cmu.edu

© 2005 by Carnegie Mellon University