Model Checking for Embedded Systems

Edmund Clarke, CMU

High-Confidence Embedded Systems Workshop, May 1st

Embedded Software verification projects

1. Bridging the gap between legacy code and formal specification
2. Verification of a real-time operating system
3. Verifying concurrent embedded C programs
4. Certifying compilation with proof-carrying code

More Efficient Model Checking Algorithms

5. Counterexample-Guided Abstraction Refinement for Hybrid Systems
6. Making Bounded Model Checking complete

1. Bridging the Gap between Legacy Code and Formal Specification

Daniel Kroening

Legacy Software

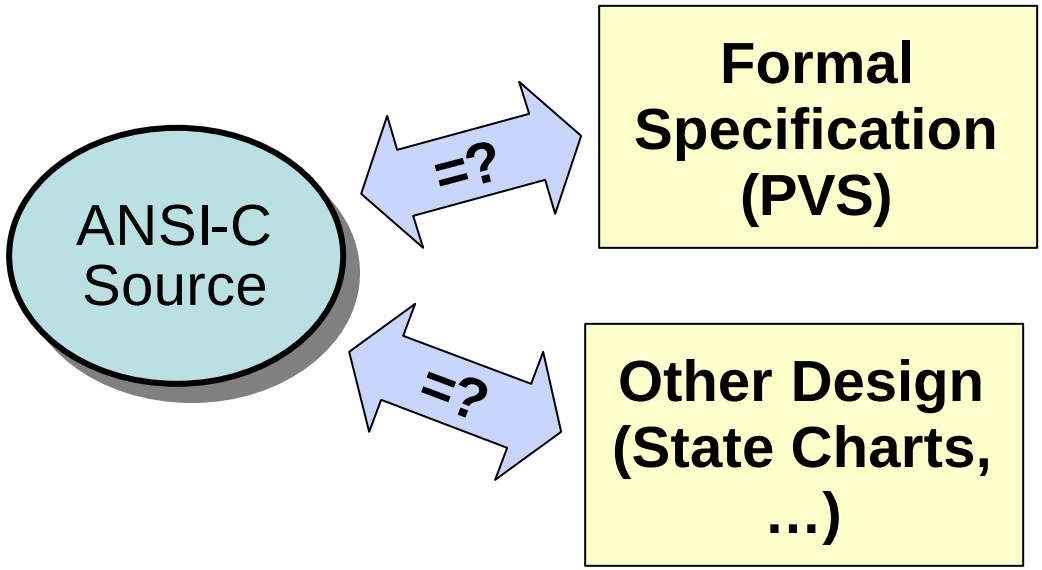
- Software is the most complex part of today's safety critical embedded systems
- Most embedded systems are legacy designs
- Written in a low level language such as ANSI-C or even assembly language
- Existing tools do not address the verification problem
- Goal: Verify legacy code with respect to
  - a formal specification
  - A high level design
  - Safety properties

Verification of Legacy Code

Bug Hunting for Security & Safety

- Safety problems because of pointers and arrays
- Run time guarantees (WCET)
- Program bugs (exceptions)

Functional Verification



CBMC

CBMC

CPROVER

ANSI-C Bounded Model Checking

- Problem: Fixpoint computation is too expensive for software
- Idea:
  - Unwind program into equation
  - Check equation using SAT
- Advantages:
  - Completely automated
  - Allows full set of ANSI-C, including full treatment of pointers and dynamic memory
- Properties:
  - Simple assertions
  - Security (Pointers/Arrays)
  - Run time guarantees (WECT)

Current Project

- Verify Safety Properties of a part of a train controller provided by GE
  - Termination / WCET
  - Correctness of pointer constructs
  - The code uses two channels for redundancy: Check that they compute the same result
  - Arithmetic consistency checks, involving multiplication and division
- The code contains x86 assembly language

Future Work

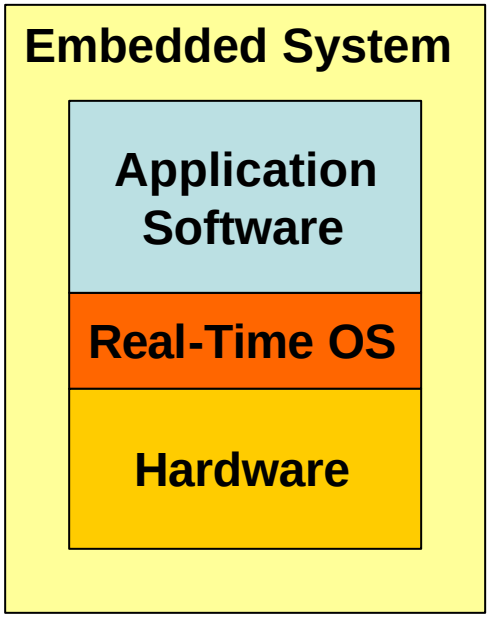
- Interval abstraction for floating point aritmetic
- Concurrent ANSI-C programs (SpecC)
- Object oriented languages (C++, Java)
- Statechart-like specification language

2. Verification of a Real-Time Operating Systems

Flavio Lerda

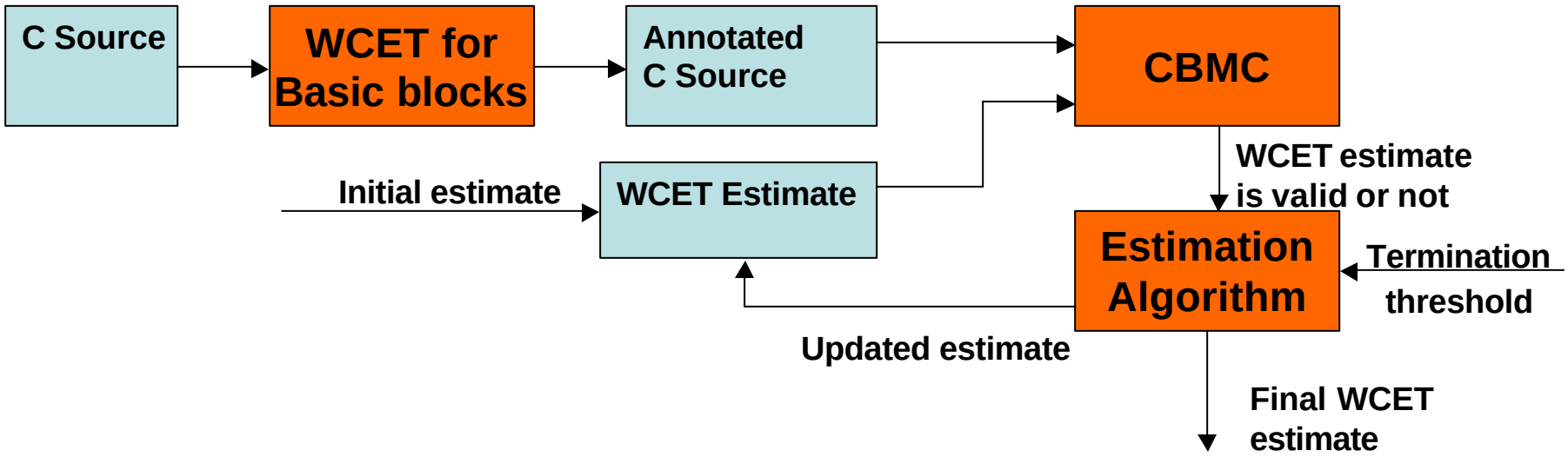
Real-Time Operating System

- Present in many embedded systems
  - Handles main functionalities
- Correctness is crucial
  - Affects the safety of the whole system
- Two type of properties
  - Timing properties
  - Functional properties



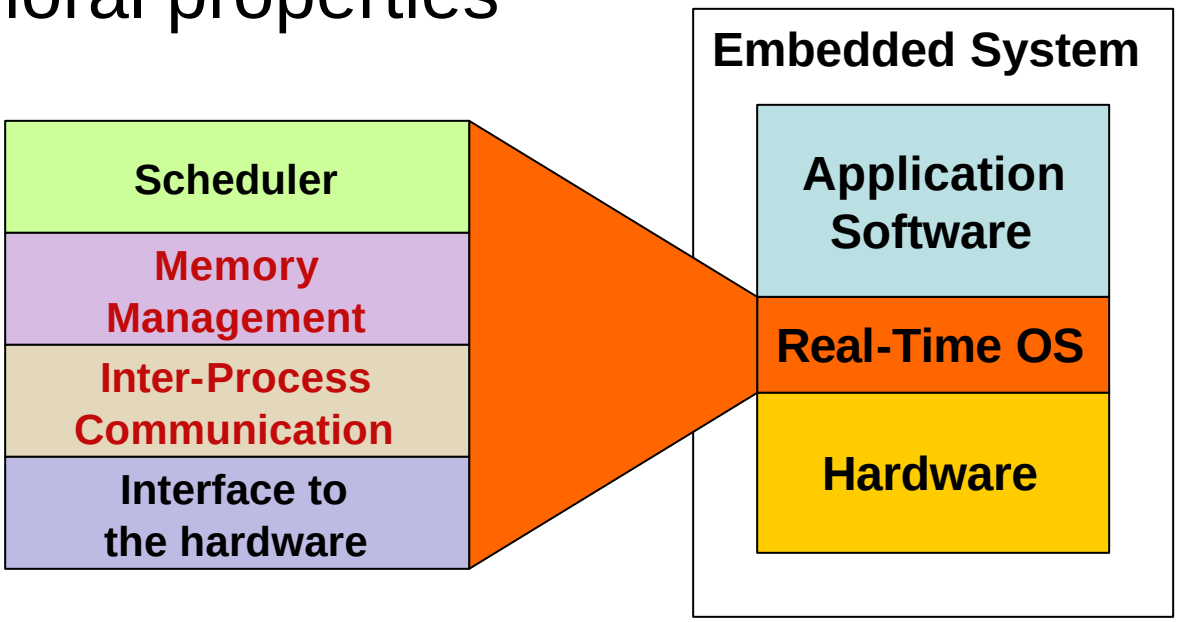
Timing Properties

- Worst Case Execution Time Analysis
  - For ANSI-C
  - Based on existing low-level analysis
  - Uses bounded model checking



Functional Properties

- Does the system behave as expected?
- Apply to different components of the OS
- Check behavioral properties



Results

- Case study: MicroC/OS
  - Real-Time Operating System
  - Freely available
  - Written in ANSI-C
- Prototype tools
  - For both type of properties
- Preliminary Results
  - On small subsystems

3. Verifying Embedded Concurrent C programs

Sagar Chaki

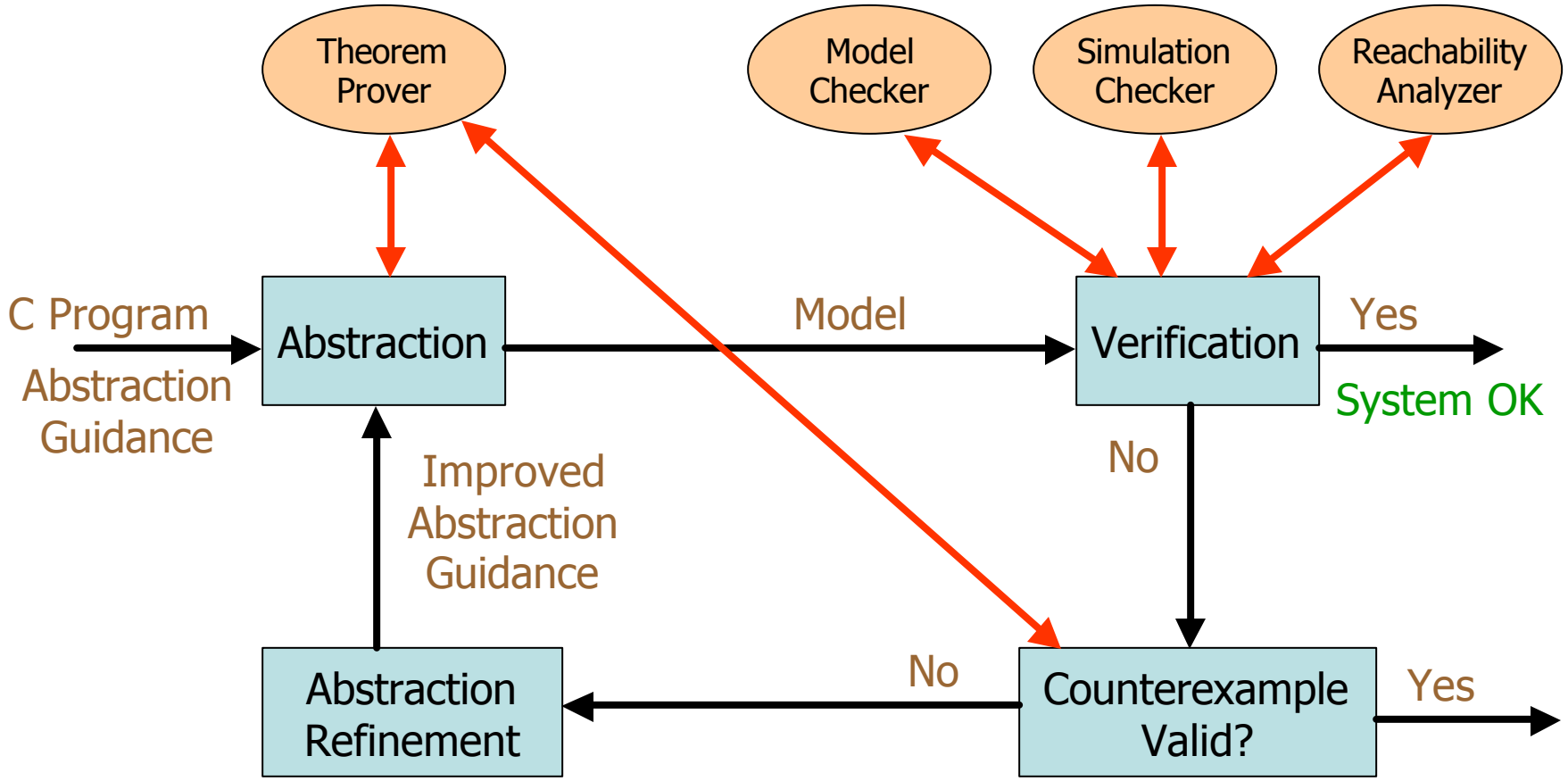
Joel Ouaknine

Karen Yorav

Overview

- Based on the counterexample guided abstraction refinement paradigm
  - Completely automated
- State explosion avoid by abstraction techniques like predicate abstraction
  - Predicate minimization
- Can handle concurrency
  - Two-level abstraction refinement

Schematic


Theorem Prover
Model Checker
Simulation Checker
Reachability Analyzer
C Program
Abstraction Guidance
Abstraction
Model
Verification
Yes
System OK
No
Improved Abstraction Guidance
Abstraction Refinement
No
Counterexample Valid?
Yes

Case Study

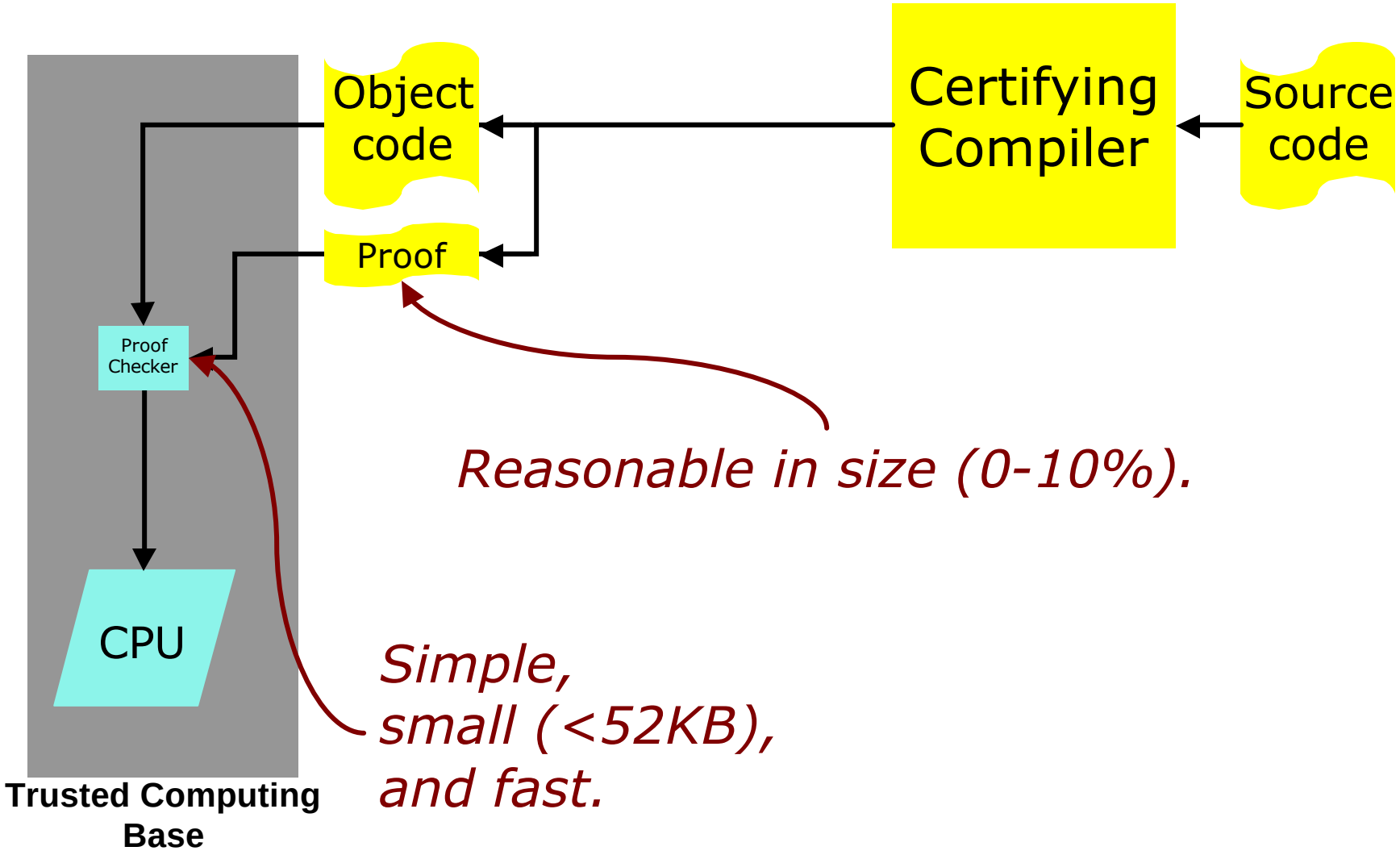
- Metal casting controller
- 30,000 lines of C
  - No recursion or dynamic memory allocation
- Verified sequential properties
  - 10 minutes CPU time, 300 MB memory
- Attempting concurrent properties
  - About 25 threads
  - No dynamic thread creation

4. Certifying Compilation with Proof-Carrying Code

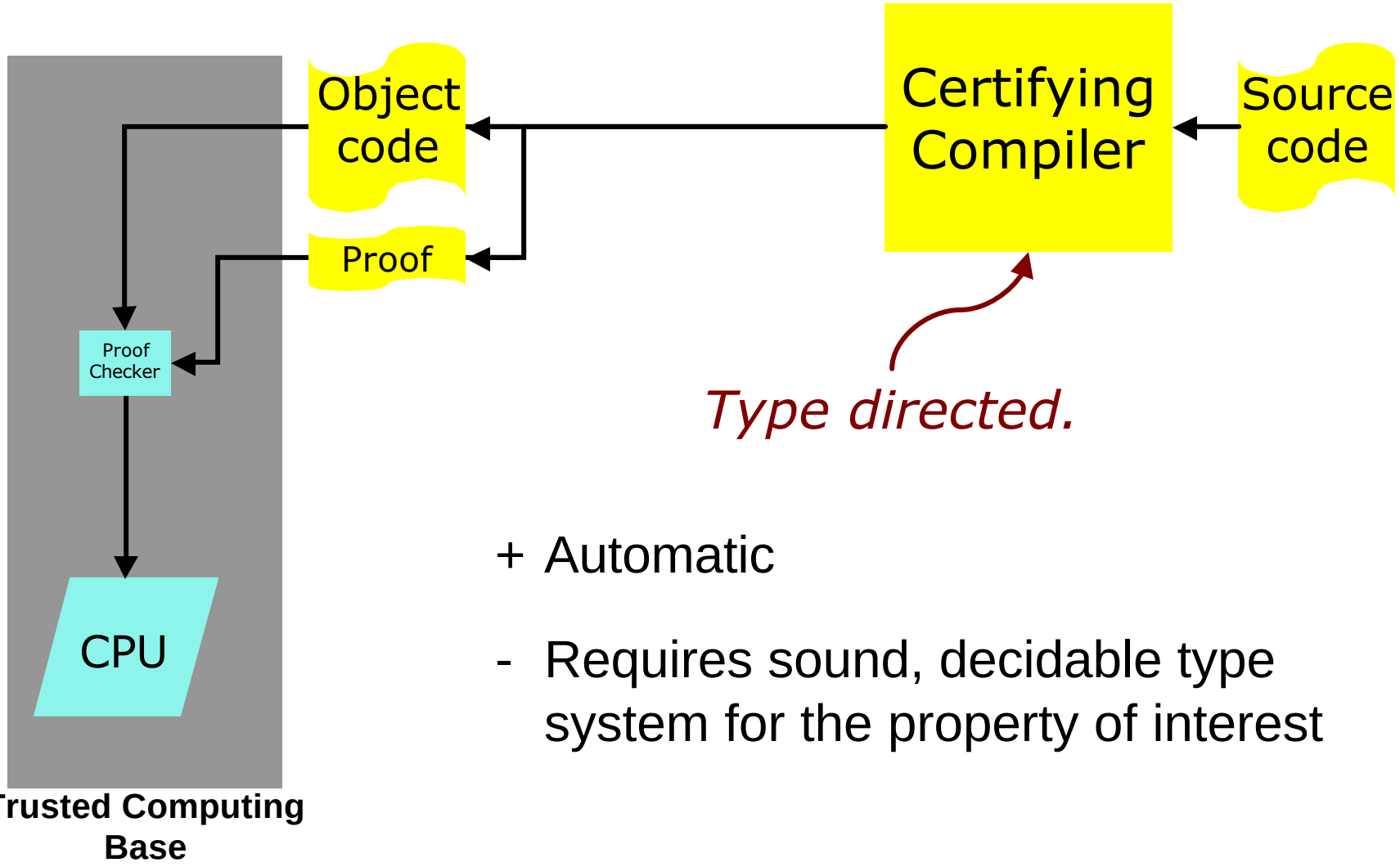
Joint work Clarke/Lee

Student: Stephen Magill

Certifying Compilation



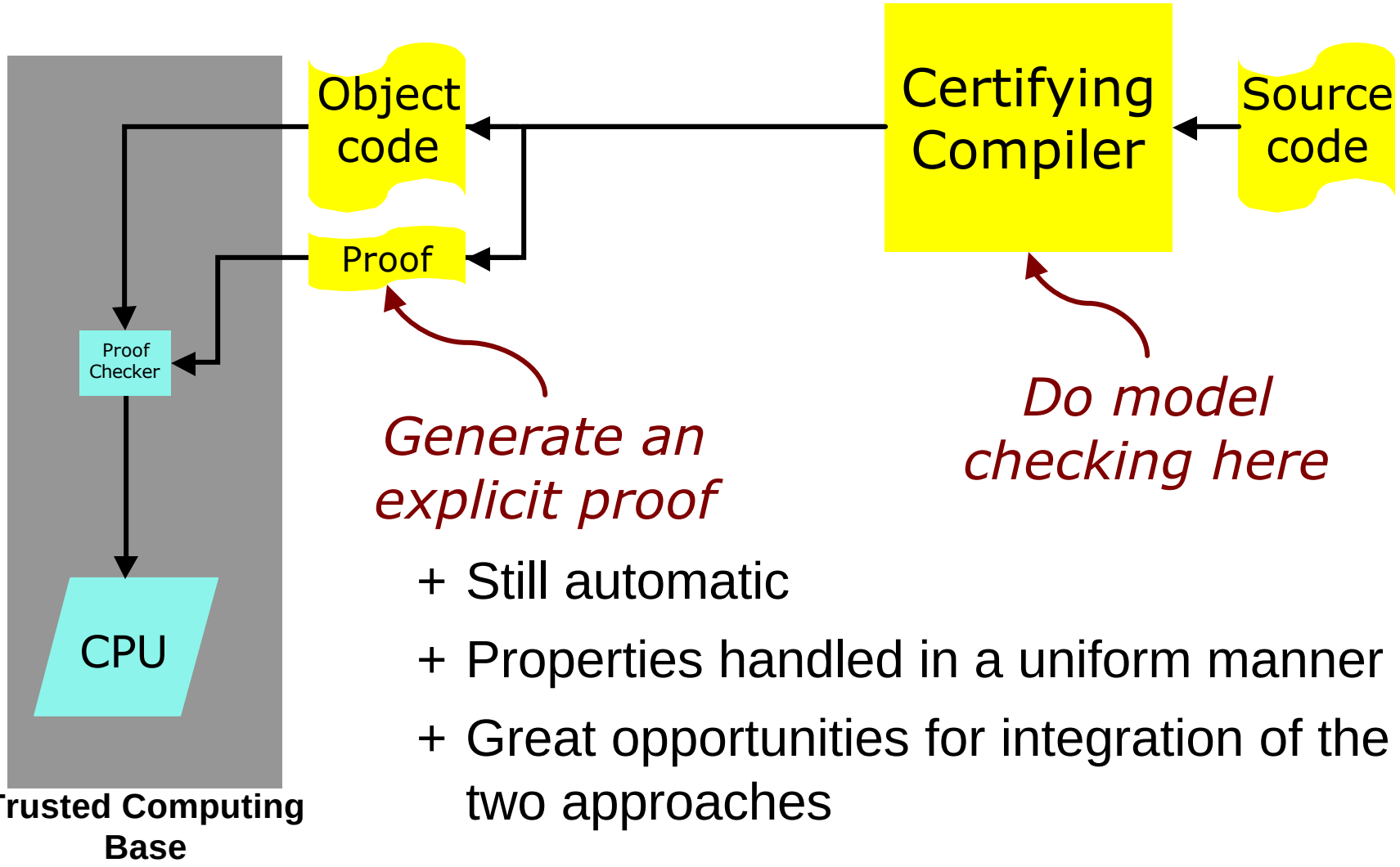
Certifying Compilation



+ Automatic

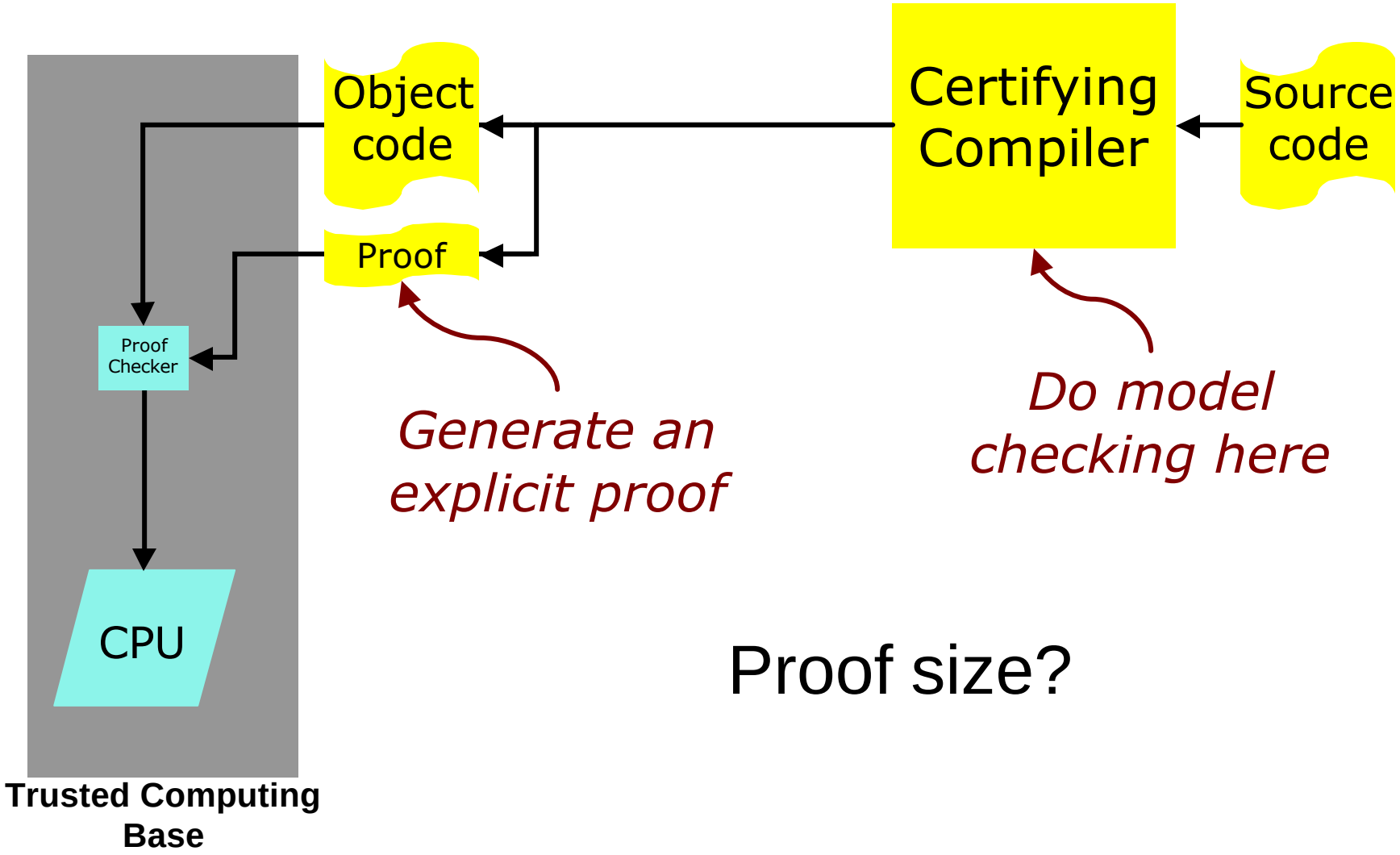
- Requires sound, decidable type system for the property of interest

Adding Model Checking



+ Still automatic
+ Properties handled in a uniform manner
+ Great opportunities for integration of the two approaches

Adding Model Checking



Proof size?

5. Counterexample-Guided Abstraction Refinement for Hybrid Systems

Joint project Clarke/Krogh:

Students:

A. Fehnker, Z. Han, J. Ouaknine,

O. Stursberg, M. Theobald

- Hybrid Systems:
  - Include continuous and discrete state variables
  - Model embedded systems
  - Applications: cars, robots, chemical plants,...
- Key Challenge:
  - Verification of properties of Hybrid System models
- Common Approach:
  - Employ abstraction to reduce complexity
  - Finding a good abstraction is difficult …
- Our Approach:
  - Automated search for a good abstraction
  - Based on Counterexample-Guided Abstraction Refinement

CEGAR: Abstraction, Validation and Refinement

Verification Problem: **Given:** **initial location + set** and **bad location**
Verify: **bad location** can never be reached

STEP1: Build initial abstraction

STEP2: Model check abstraction
no **CE** ⇒ **DONE (safe)**

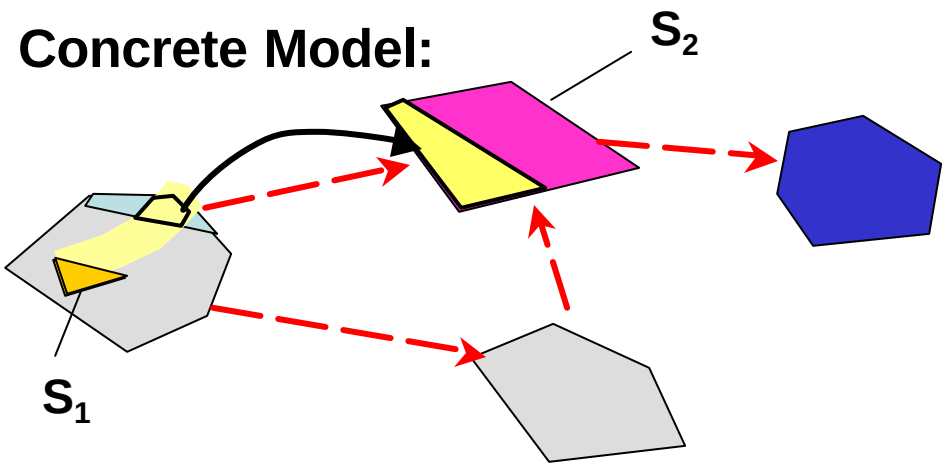
STEP3: For each transition in **CE**:
- validate transition
- refine abstraction

Case 1: S_2 is not reachable
Case 2: **Part** of S_2 not reachable
Case 3: All of S_2 reachable

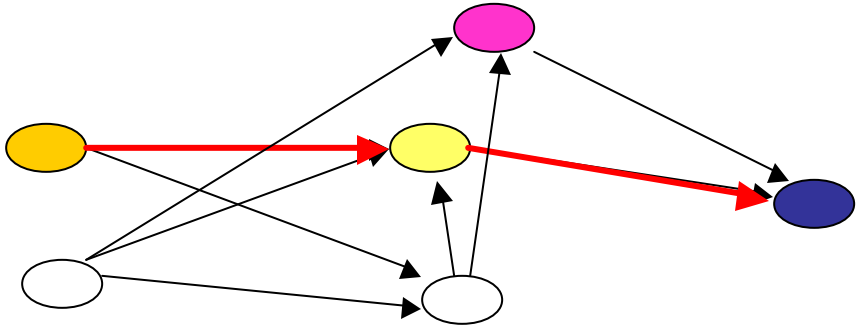
Case 2,3 ⇒ next transition
Case 1 ⇒ GOTO STEP 2

STEP4: DONE (unsafe)

Concrete Model:



Abstract Model:



Summary and Results

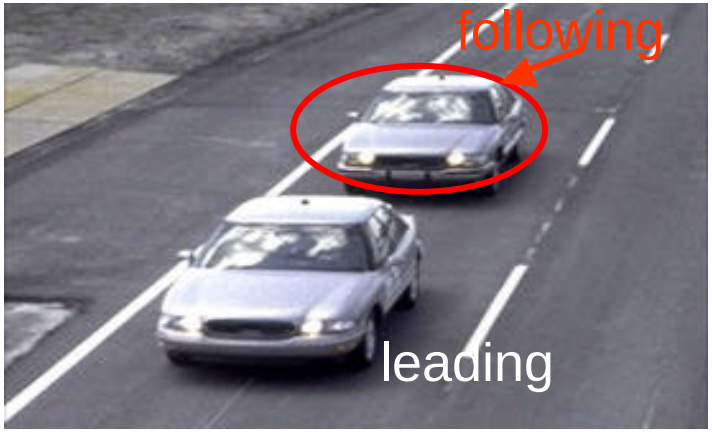
Main Ideas of CEGAR for Hybrid Systems

- explore dynamics only in part of the system relevant to the property
- local refinement within a location
- framework for different over-approximation methods
- consider fragments of counterexamples

Car Steering Example

Common reachability analysis:	185 sec
CEGAR with multiple over-approximations:	69 sec
As before, with considering fragments:	20 sec

Adaptive Cruise Control



Common reachability analysis:	728 sec
CEGAR with multiple over-approximations:	495 sec
As before, with considering fragments:	43 sec

6. Making Bounded Model Checking Complete

Daniel Kroening

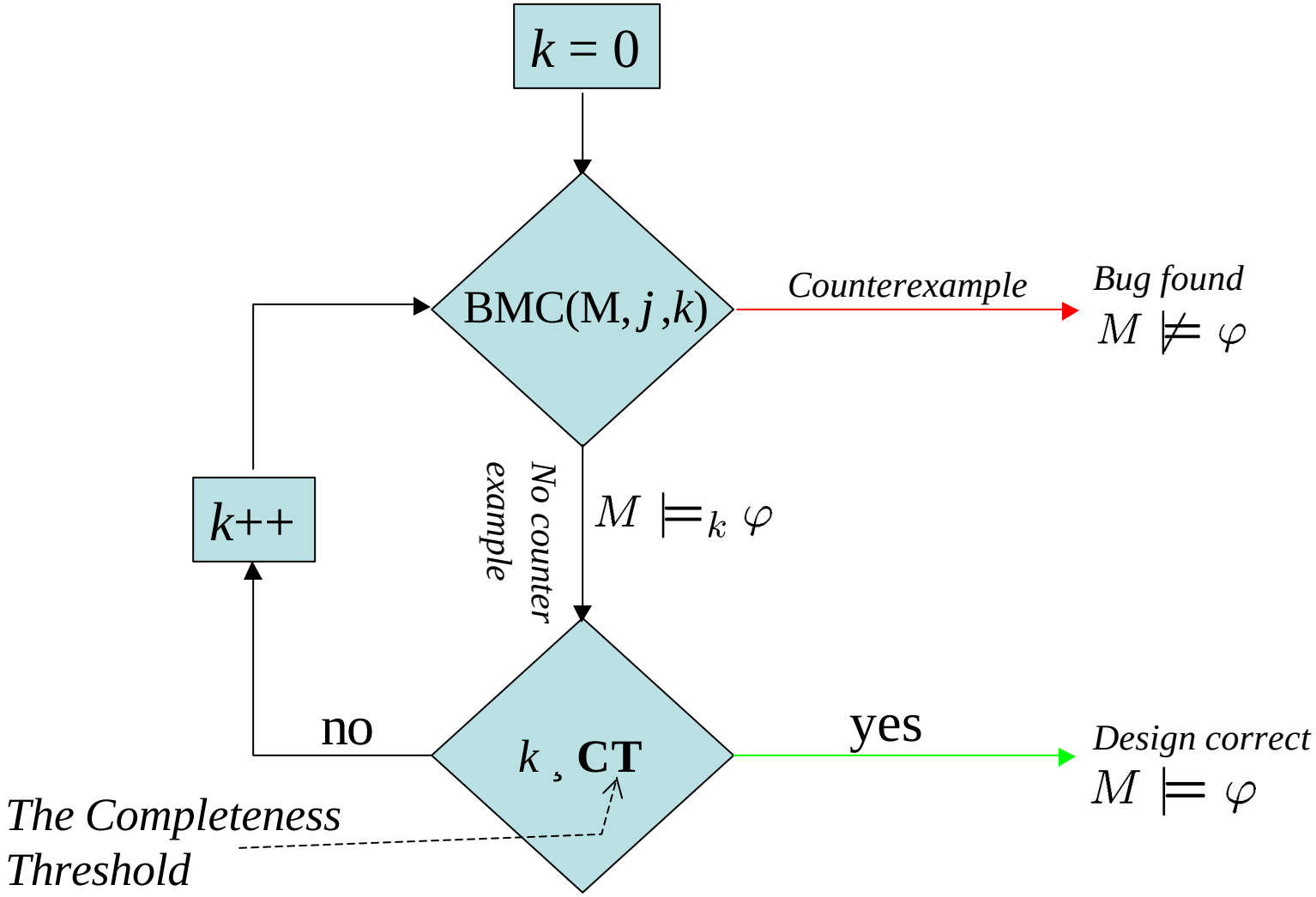
Joel Ouaknine

Ofer Strichman

Bounded Model Checking

- A technique for incremental search for logical design errors
- Competes with traditional Model Checkers
- Often finds errors orders of magnitude faster than traditional model checkers.
- Widely adopted by the chip-design industry in the last 3 years.
- Applicable to high confidence embedded systems such as embedded controllers and communication protocols.

The Bounded Model Checking loop



How deep should we look for bugs ?

- In order to prove systems correct, a pre-computed "completeness threshold" must be known.
- The completeness threshold is a number *CT* such that, if no bug is found of length *CT* or less, then there is no bug at all
- The value of *CT* depends on the model M and the property φ
- Computing *CT* for general Linear Temporal Logic formulas has so far been an open problem

Computing the completeness threshold

- Our solution is based on a graph-theoretic analysis of the automaton $M \times B_{\neg\varphi}$, where
  - *M* is the automaton representing the verified system
  - $B_{\neg\varphi}$ is the (Buchi) automaton representing the negation of the property
- The completeness threshold (*CT*) is a function of the *diameter* and the *Recurrence-diameter* of the product automaton