Verification of Concurrent Algorithms: an Abstraction Refinement Approach

Giorgio Delzanno\textsuperscript{1} Frédéric Haziza\textsuperscript{2} Ahmed Rezine\textsuperscript{2}

\textsuperscript{1}Università di Genova
\textsuperscript{2}Uppsala Universitet

MCC Uppsala 2009
Outline

Introduction

Concurrent Algorithms

Symbolic Representation and Abstraction for Verification

Counter Example Guided Abstraction Refinement

Conclusion
Outline

Introduction

Concurrent Algorithms

Symbolic Representation and Abstraction for Verification

Counter Example Guided Abstraction Refinement

Conclusion
Introduction: parallelization

Exponential increase in processors speed has reached its technological limits

- Increase the parallelization
Introduction: parallelization

Exponential increase in processors speed has reached its technological limits

- Increase the parallelization
- Divide the workload on parallel threads
Introduction: parallelization

Exponential increase in processors speed has reached its technological limits

- Increase the parallelization
- Divide the workload on parallel threads
- Sustain the performance growth
Exponential increase in processors speed has reached its technological limits

- Increase the parallelization
- Divide the workload on parallel threads
- Sustain the performance growth
- Concurrent algorithms are already a reality, e.g. the java.util.concurrent
Parallelization comes at a price:

- Code extremely difficult to get right and to debug
Parallelization comes at a price:

- Code extremely difficult to get right and to debug
- Correctness need to cover all possible interleavings, and all possible numbers of threads
Parallelization comes at a price:

- Code extremely difficult to get right and to debug
- Correctness need to cover all possible interleavings, and all possible numbers of threads
- Requires expertise and experience
Parallelization comes at a price:

- Code extremely difficult to get right and to debug
- Correctness need to cover all possible interleavings, and all possible numbers of threads
- Requires expertise and experience
- Built on intuition, often with hand-waving correctness arguments
Parallelization comes at a price:

- Code extremely difficult to get right and to debug
- Correctness need to cover all possible interleavings, and all possible numbers of threads
- Requires expertise and experience
- Built on intuition, often with hand-waving correctness arguments

Goal: Develop automatic tools that analyze source code and check correctness for all possible interleavings and all possible numbers of threads
Tools are to implement verification techniques:
Verification Approach

Tools are to implement verification techniques:
- Testing and simulation are good for finding errors
- Model checking is good for showing correctness and for debugging

\[ \text{Model} \approx \text{Spec} \Rightarrow \text{Trace} \Rightarrow \text{Goal: Extend Model Checking to analyze concurrent algorithms} \]
Tools are to implement verification techniques:

- Testing and simulation are good for finding errors.
- Model checking is good for showing correctness and for debugging.
Verification Approach

Tools are to implement verification techniques:

- Testing and simulation are good for finding errors.
- Model checking is good for showing correctness and for debugging.

\[ \text{Model} \models \text{Spec} \]
Verification Approach

Tools are to implement verification techniques:

▶ Testing and simulation are good for finding errors

▶ Model checking is good for showing correctness and for debugging

Model $\models$ Spec
Verification Approach

Tools are to implement verification techniques:
- Testing and simulation are good for finding errors
- Model checking is good for showing correctness and for debugging

Model $\models$ Spec

Goal: Extend Model Checking to analyze concurrent algorithms
Tools are to implement verification techniques:

- Testing and simulation are good for finding errors
- Model checking is good for showing correctness and for debugging

Goal: Extend Model Checking to analyze concurrent algorithms
Outline

Introduction

Concurrent Algorithms

Symbolic Representation and Abstraction for Verification

Counter Example Guided Abstraction Refinement

Conclusion
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-blocking queue

\[ E(v_1) \]

\[ H \quad T \]

\[ \begin{array}{c}
H \quad T \\
0 \quad 1 \\
1 \quad 0 \\
\# \\
\end{array} \]
Concurrent Algorithms: A non-blocking queue

Diagram showing a non-blocking queue with nodes labeled H, T, and v2, with arrows indicating connections and directions.
Concurrent Algorithms: A non-bloquing queue
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-blocking queue

Diagram showing the structure of a non-blocking queue with nodes labeled H and T, and elements 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-bloquing queue
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: A non-blocking queue
Concurrent Algorithms: a non-blocking queue
Concurrent Algorithms: a non-blocking queue
Concurrent Algorithms: a non-blocking queue

D(v₁)

E(v₁)

E(v₂)

1 2 3 4 5 6 12

8 9 10

17

H T

0 1 1 1

0 2 2

# 0 0

node next tail

head next tail

0' 1' 2'

2'

0' 1' 2
Concurrent Algorithms: a non-blocking queue
Concurrent Algorithms: a non-blocking queue
Concurrent Algorithms: a non-blocking queue

Diagram showing nodes and connections with labels:
- $D(v_1)$
- $E(v_2)$
- $E(v_3)$

Nodes and arrows labeled with numbers and symbols such as `H`, `T`, `node`, `tail`, `next`.
Concurrent Algorithms: a non-bloquing queue
Concurrent Algorithms: a non-blocking queue
Concurrent Algorithms

Analysis needs to take into account several sources of difficulty:

▶ Arbitrary numbers of threads
▶ Infinite data domains
▶ Dynamic memory
▶ Memory model guaranteed by the machine
▶ ...

Outline

Introduction

Concurrent Algorithms

Symbolic Representation and Abstraction for Verification

Counter Example Guided Abstraction Refinement

Conclusion
Symbolic Representations for Infinite Sets of States

Violations representable by finite traces

Use a finite state observer that synchronizes with the system

Adding more threads still violates the safety property

Use upward closed sets as symbolic representations for states violating safety properties
Symbolic Representations for Infinite Sets of States

- Safety properties’
  Violations representable
  by finite traces
Symbolic Representations for Infinite Sets of States

- Safety properties’ violations representable by finite traces
- Use a finite state observer that synchronizes with the system

Adding more threads still violates the safety property

Use upward closed sets as symbolic representations for states violating safety properties
Symbolic Representations for Infinite Sets of States

- Safety properties’ violations representable by finite traces
- Use a finite state observer that synchronizes with the system
Symbolic Representations for Infinite Sets of States

- Safety properties’ violations representable by finite traces
- Use a finite state observer that synchronizes with the system
- Adding more threads still violates the safety property
Symbolic Representations for Infinite Sets of States

- Safety properties’ violations representable by finite traces
- Use a finite state observer that synchronizes with the system
- Adding more threads still violates the safety property
Symbolic Representations for Infinite Sets of States

- Safety properties’ violations representable by finite traces
- Use a finite state observer that synchronizes with the system
- Adding more threads still violates the safety property

Use upward closed sets as symbolic representations for states violating safety properties
Symbolic Representation and Backward Analysis
Symbolic Representation and Backward Analysis
Symbolic Representation and Backward Analysis
Symbolic Representation and Backward Analysis
Symbolic Representation and Backward Analysis
Symbolic Representation and Backward Analysis
Symbolic Representation and Backward Analysis
Symbolic Representation and Backward Analysis
Symbolic Representation and Backward Analysis

\[ \text{Pre}^*() \]
Symbolic Representation and Backward Analysis

\[ \text{Pre}^*(\quad) \]

\( \text{init}_a \)
\( \text{init}_b \)
\( \text{init}_c \)
Abstraction to Permit Symbolic Analysis
Abstraction to Permit Symbolic Analysis
Abstraction to Permit Symbolic Analysis

\[ c_3 \rightarrow c_4 \]

\[ c_1 \rightarrow c_2 \]
Abstraction to Permit Symbolic Analysis

\[ c_3 \rightarrow c_4 \leq c_3 \leq X \rightarrow c_2 \leq c_4 \rightarrow c_1 \]
Abstraction to Permit Symbolic Analysis
Abstraction to Permit Symbolic Analysis
Abstraction and Backward Analysis

\[
\text{Pre}^*(\quad )
\]

\[\text{init}_a\]
\[\text{init}_b\]
\[\text{init}_c\]
Outline

Introduction

Concurrent Algorithms

Symbolic Representation and Abstraction for Verification

Counter Example Guided Abstraction Refinement

Conclusion
Counter Example Guided Abstraction Refinement

φ₀ = Bad

φ₁

φ₂

φ₃

post(i₁ ∧ φ₂)

post(i₀ ∧ φ₃)

post(i₂ ∧ φ₁)
Counter Example Guided Abstraction Refinement

\[ \phi_0 = \text{Bad} \]

\[ \phi_1 = \text{post}(i_1 \land \phi_2) \]

\[ \phi_2 = \text{post}(i_0 \land \phi_3) \]

\[ \phi_3 = \text{post}(i_2 \land \phi_1) \]
Counter Example Guided Abstraction Refinement

\[ \phi_0 = \text{Bad} \]

\[ \phi_1 \]

\[ \phi_0 = \text{Bad} \]

\[ \text{post}(i_1 \land \phi_2) \]

\[ \text{post}(i_0 \land \phi_3) \]

\[ \text{post}(i_2 \land \phi_1) \]
Counter Example Guided Abstraction Refinement

\[ \phi_0 = \text{Bad} \]

\[ \phi_1 \]

\[ \phi_2 \]

\[ \text{post}(i_1 \land \phi_2) \]

\[ \text{post}(i_0 \land \phi_3) \]

\[ \text{post}(i_2 \land \phi_1) \]
Counter Example Guided Abstraction Refinement

\[ \phi_0 = \text{Bad} \]

\[ \phi_1 = \text{init} \]

\[ \text{post}(i_1 \land \phi_2) \]

\[ \text{post}(i_0 \land \phi_3) \]

\[ \text{post}(i_2 \land \phi_1) \]
Counter Example Guided Abstraction Refinement

\[ \phi_0 = \text{Bad} \]

\[ \phi_1 \]

\[ \phi_2 \]

\[ \phi_3 \]

\[ i_0 = \text{init} \]

\[ \text{post}(i_0 \land \phi_3) \]

\[ \text{post}(i_0 \land \phi_2) \]

\[ \text{post}(i_0 \land \phi_1) \]
\[ \phi_0 = \text{Bad} \]

\[ \phi_1 \]

\[ \phi_2 \]

\[ \phi_3 \]

\[ i_0 = \text{init} \]

\[ \text{post}(i_0 \land \phi_3) \]
Counter Example Guided Abstraction Refinement

\[ \phi_0 = \text{Bad} \]

\[ \phi_i \]

\[ \text{post}(i_0 \land \phi_3) \]

\[ \text{post}(i_1 \land \phi_2) \]

\[ \text{post}(i_1 \land \phi_2) \]

\[ \phi_0 = \text{Bad} \]
Counter Example Guided Abstraction Refinement

\[ \phi_0 = \text{Bad} \]

\[ \begin{align*}
\phi_1 &= \text{post}(i_1 \land \phi_2) \\
\phi_2 &= \text{post}(i_1 \land \phi_2) \\
\phi_3 &= \text{post}(i_0 \land \phi_3) \\
\phi_0 &= \text{init} \\
\end{align*} \]
Counter Example Guided Abstraction Refinement
Counter Example Guided Abstraction Refinement

$c_3 \leq c_1 \leq X \leq c_2 \leq c_4$
Counter Example Guided Abstraction Refinement

\[ c_3 \leq c_4 \leq X \leq c_2 \]
Counter Example Guided Abstraction Refinement

$c_3 \leq X \leq c_4$

$c_1 \rightarrow c_2$

$c_4$
Outline

Introduction

Concurrent Algorithms

Symbolic Representation and Abstraction for Verification

Counter Example Guided Abstraction Refinement

Conclusion
Conclusion and Future Work

- Parallelization makes heavy use of concurrent algorithms.
Conclusion and Future Work

- Parallelization makes heavy use of concurrent algorithms
- We build automatic symbolic techniques to increase confidence in concurrent algorithms
Conclusion and Future Work

- Parallelization makes heavy use of concurrent algorithms
- We build automatic symbolic techniques to increase confidence in concurrent algorithms
- Our techniques explore all interleavings for arbitrary numbers of threads

We obtained successful results and are working on the verification of code found in widespread software, like the java.util.concurrent package.
Conclusion and Future Work

- Parallelization makes heavy use of concurrent algorithms
- We build automatic symbolic techniques to increase confidence in concurrent algorithms
- Our techniques explore all interleavings for arbitrary numbers of threads
- We obtained successful results and are working on the verification of code found in widespread software, like the java.util.concurrent package