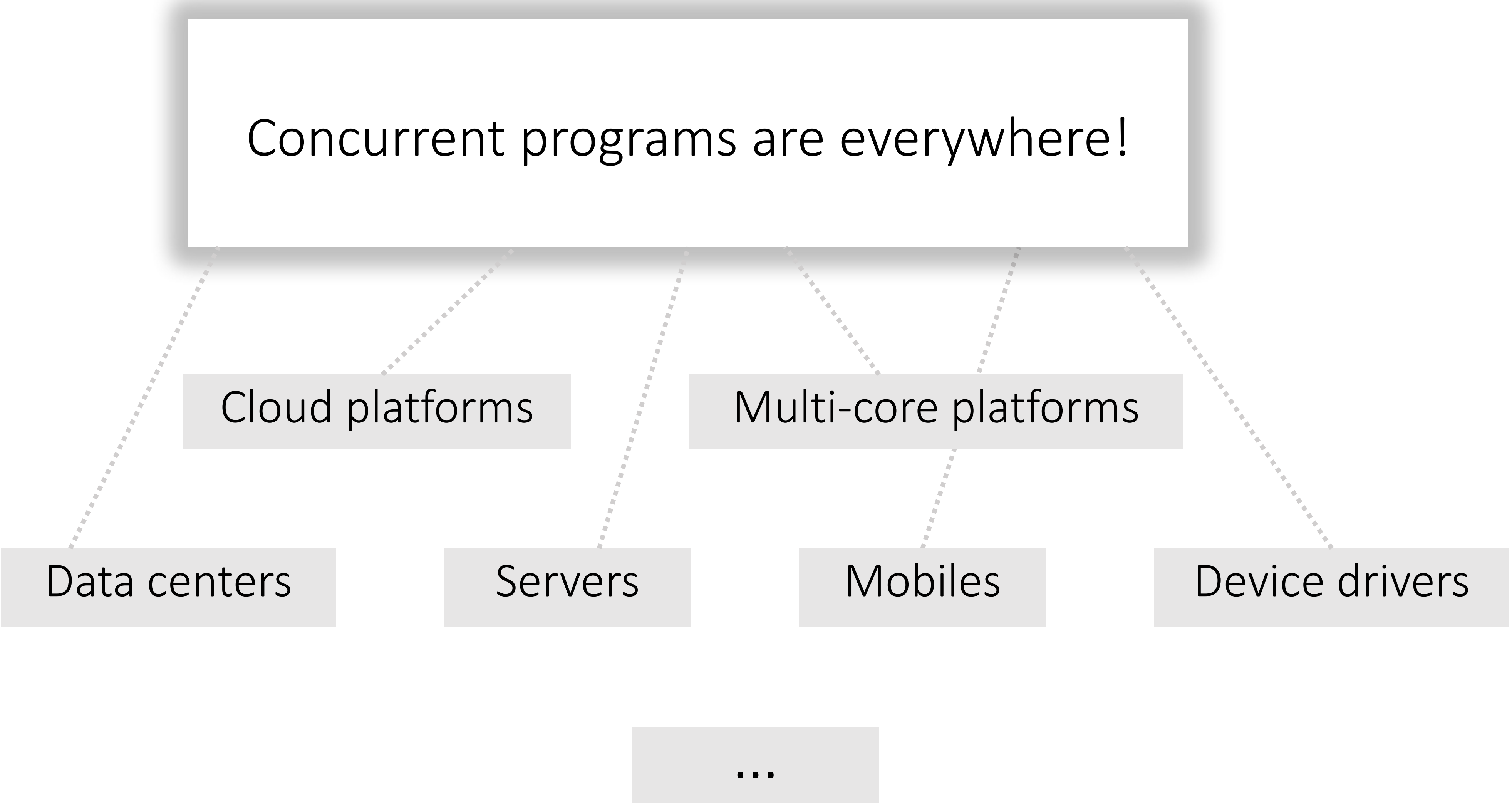
Computer-aided Concurrent Programming

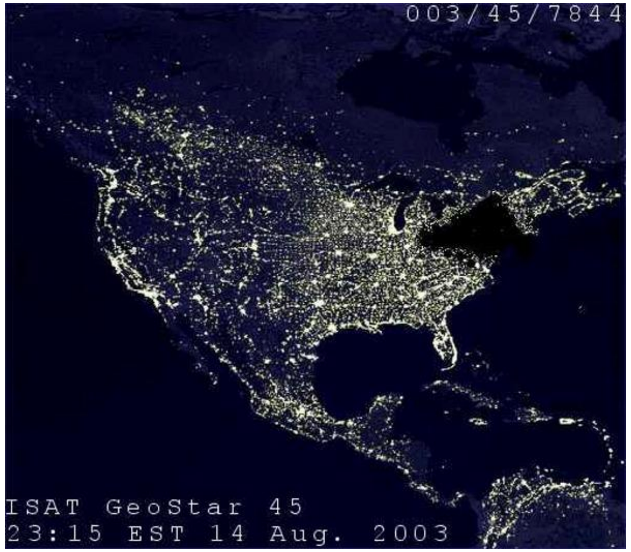
Roopsha Samanta

PURDUE
UNIVERSITY


Concurrent programs are everywhere!
Cloud platforms
Multi-core platforms
Data centers
Servers
Mobiles
Device drivers
...

Concurrency bugs are subtle and hard to debug

Therac-25 radiotherapy machine overdose
6 deaths. Race conditions, overflow error.



North American power blackout
11 deaths. $6 billion loss. Race condition.

Many concurrency bugs are due to synchronization errors

Atomicity violation

Race condition

Ordering violation

Deadlock

Livelock

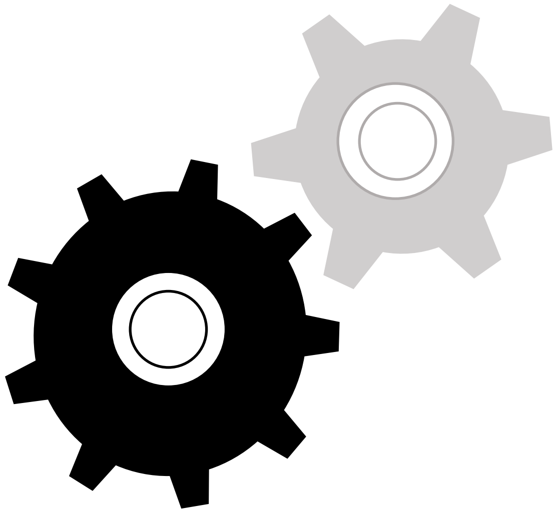
Starvation

...

Computer-aided Concurrent Programming

Unsynchronized program P →

Specification →



Synchronization Synthesizer

→ Correct program P’

Assumption: Programmer ensures P is correct when executed sequentially

A seminal paper

A cool paper

A modern approach

A seminal paper

A cool paper

A modern approach

A seminal paper

Clarke

Emerson

Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. Workshop on Logics of Programs 1981.

DESIGN AND SYNTHESIS OF SYNCHRONIZATION SKELETONS
USING BRANCHING TIME TEMPORAL LOGIC

Edmund M. Clarke
E. Allen Emerson
Aiken Computation Laboratory
Harvard University
Cambridge, Mass. 02138, USA

1. INTRODUCTION

We propose a method of constructing concurrent programs in which the *synchronization skeleton* of the program is automatically synthesized from a high-level (branching time) Temporal Logic specification. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. For example, in the synchronization skeleton for a solution to the critical section problem each process's critical section may be viewed as a single node since the internal structure of the critical section is unimportant. Most solutions to synchronization problems in the literature are in fact given as synchronization skeletons. Because synchronization skeletons are in general finite state, the propositional version of Temporal Logic can be used to specify their properties.

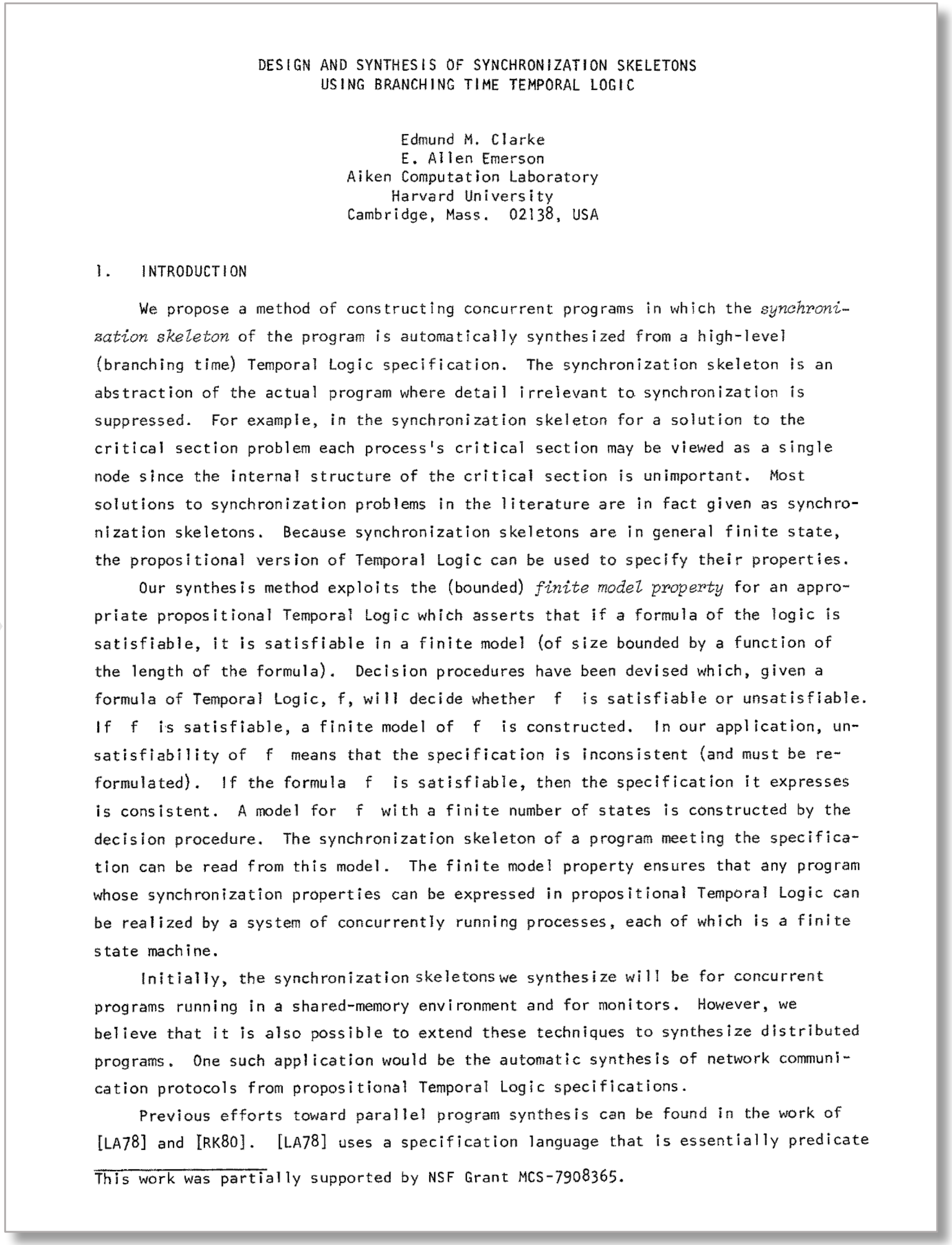
Our synthesis method exploits the (bounded) *finite model property* for an appropriate propositional Temporal Logic which asserts that if a formula of the logic is satisfiable, it is satisfiable in a finite model (of size bounded by a function of the length of the formula). Decision procedures have been devised which, given a formula of Temporal Logic, f, will decide whether f is satisfiable or unsatisfiable. If f is satisfiable, a finite model of f is constructed. In our application, unsatisfiability of f means that the specification is inconsistent (and must be reformulated). If the formula f is satisfiable, then the specification it expresses is consistent. A model for f with a finite number of states is constructed by the decision procedure. The synchronization skeleton of a program meeting the specification can be read from this model. The finite model property ensures that any program whose synchronization properties can be expressed in propositional Temporal Logic can be realized by a system of concurrently running processes, each of which is a finite state machine.

Initially, the synchronization skeletons we synthesize will be for concurrent programs running in a shared-memory environment and for monitors. However, we believe that it is also possible to extend these techniques to synthesize distributed programs. One such application would be the automatic synthesis of network communication protocols from propositional Temporal Logic specifications.

Previous efforts toward parallel program synthesis can be found in the work of [LA78] and [RK80]. [LA78] uses a specification language that is essentially predicate

This work was partially supported by NSF Grant MCS-7908365.

Algorithmic framework to check and synthesize synchronization for temporal properties of finite-state transition systems



DESIGN AND SYNTHESIS OF SYNCHRONIZATION SKELETONS
USING BRANCHING TIME TEMPORAL LOGIC

Edmund M. Clarke
E. Allen Emerson
Aiken Computation Laboratory
Harvard University
Cambridge, Mass. 02138, USA

1. INTRODUCTION

We propose a method of constructing concurrent programs in which the *synchronization skeleton* of the program is automatically synthesized from a high-level (branching time) Temporal Logic specification. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. For example, in the synchronization skeleton for a solution to the critical section problem each process's critical section may be viewed as a single node since the internal structure of the critical section is unimportant. Most solutions to synchronization problems in the literature are in fact given as synchronization skeletons. Because synchronization skeletons are in general finite state, the propositional version of Temporal Logic can be used to specify their properties.

Our synthesis method exploits the (bounded) *finite model property* for an appropriate propositional Temporal Logic which asserts that if a formula of the logic is satisfiable, it is satisfiable in a finite model (of size bounded by a function of the length of the formula). Decision procedures have been devised which, given a formula of Temporal Logic, f, will decide whether f is satisfiable or unsatisfiable. If f is satisfiable, a finite model of f is constructed. In our application, unsatisfiability of f means that the specification is inconsistent (and must be reformulated). If the formula f is satisfiable, then the specification it expresses is consistent. A model for f with a finite number of states is constructed by the decision procedure. The synchronization skeleton of a program meeting the specification can be read from this model. The finite model property ensures that any program whose synchronization properties can be expressed in propositional Temporal Logic can be realized by a system of concurrently running processes, each of which is a finite state machine.

Initially, the synchronization skeletons we synthesize will be for concurrent programs running in a shared-memory environment and for monitors. However, we believe that it is also possible to extend these techniques to synthesize distributed programs. One such application would be the automatic synthesis of network communication protocols from propositional Temporal Logic specifications.

Previous efforts toward parallel program synthesis can be found in the work of [LA78] and [RK80]. [LA78] uses a specification language that is essentially predicate

This work was partially supported by NSF Grant MCS-7908365.

- **Process:**
  Finite-state synchronization skeleton
- **Communication Model:**
  Shared-memory, interleaving-based
- **Specification:**
  Temporal logic, complete
- **Synchronization:**
  Guarded commands
- **Procedure:**
  Tableau-based decision procedure

DESIGN AND SYNTHESIS OF SYNCHRONIZATION SKELETONS USING BRANCHING TIME TEMPORAL LOGIC

Edmund M. Clarke
E. Allen Emerson
Aiken Computation Laboratory
Harvard University
Cambridge, Mass. 02138, USA

1. INTRODUCTION

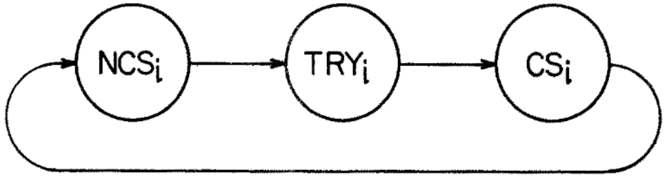
We propose a method of constructing concurrent programs in which the *synchronization skeleton* of the program is automatically synthesized from a high-level (branching time) Temporal Logic specification. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. For example, in the synchronization skeleton for a solution to the critical section problem each process's critical section may be viewed as a single node since the internal structure of the critical section is unimportant. Most solutions to synchronization problems in the literature are in fact given as synchronization skeletons. Because synchronization skeletons are in general finite state, the propositional version of Temporal Logic can be used to specify their properties.

Our synthesis method exploits the (bounded) *finite model property* for an appropriate propositional Temporal Logic which asserts that if a formula of the logic is satisfiable, it is satisfiable in a finite model (of size bounded by a function of the length of the formula). Decision procedures have been devised which, given a formula of Temporal Logic, f, will decide whether f is satisfiable or unsatisfiable. If f is satisfiable, a finite model of f is constructed. In our application, unsatisfiability of f means that the specification is inconsistent (and must be reformulated). If the formula f is satisfiable, then the specification it expresses is consistent. A model for f with a finite number of states is constructed by the decision procedure. The synchronization skeleton of a program meeting the specification can be read from this model. The finite model property ensures that any program whose synchronization properties can be expressed in propositional Temporal Logic can be realized by a system of concurrently running processes, each of which is a finite state machine.

Initially, the synchronization skeletons we synthesize will be for concurrent programs running in a shared-memory environment and for monitors. However, we believe that it is also possible to extend these techniques to synthesize distributed programs. One such application would be the automatic synthesis of network communication protocols from propositional Temporal Logic specifications.

Previous efforts toward parallel program synthesis can be found in the work of [LA78] and [RK80]. [LA78] uses a specification language that is essentially predicate

This work was partially supported by NSF Grant MCS-7908365.



Mutual exclusion:

AG $\neg(\mathrm{CS}_1 \wedge \mathrm{CS}_2)$

Absence of starvation:

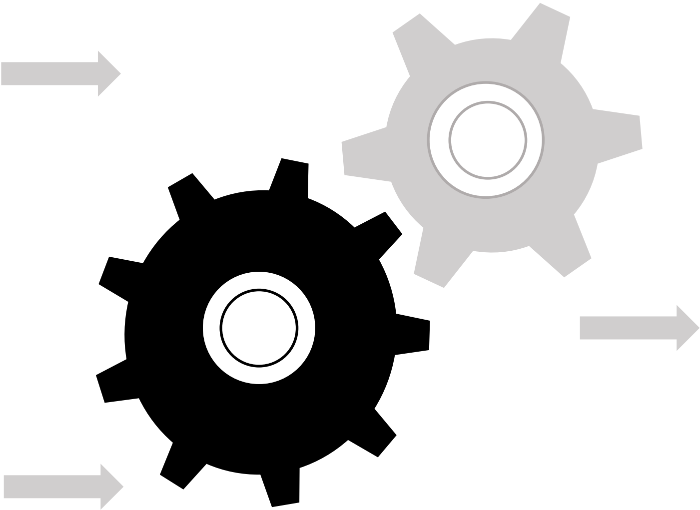
AG $\mathrm{TRY}_i \rightarrow$ **AF** CS_i

Process specification:

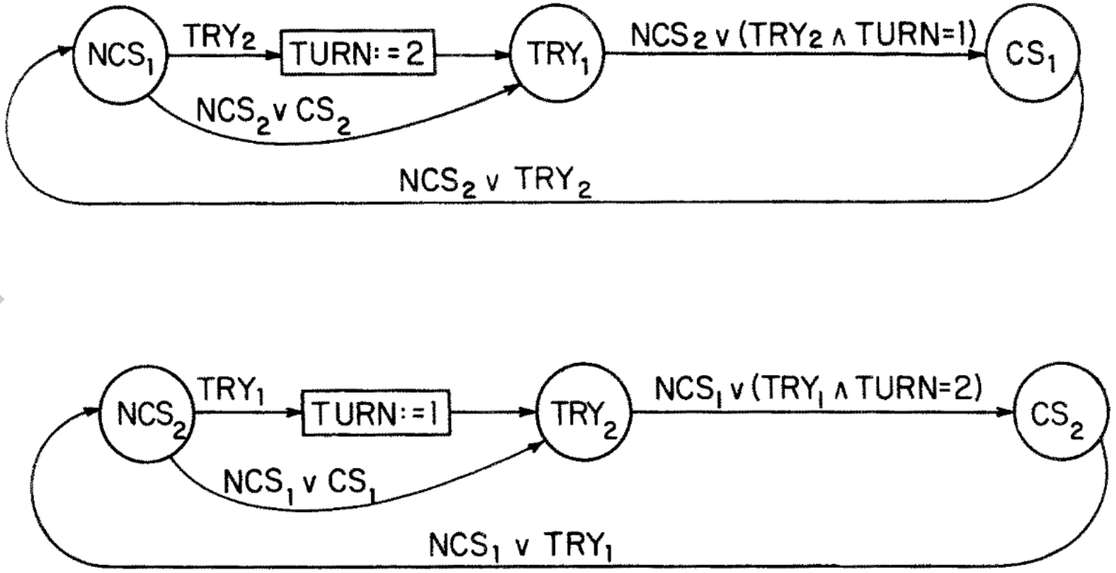
AG $\mathrm{NCS}_i \vee \mathrm{TRY}_i \vee \mathrm{CS}_i$

AG $\mathrm{NCS}_i \rightarrow \neg(\mathrm{TRY}_i \vee \mathrm{CS}_i)$

...



Synchronization Synthesizer



- **Process:**
  Finite-state synchronization skeleton
- **Communication Model:**
  Shared-memory, interleaving-based
- **Specification:**
  Temporal logic, complete
- **Synchronization:**
  Guarded commands
- **Procedure:**
  Tableau-based decision procedure

- Computation Tree Logic (CTL)
- Model Checking for CTL
- CTL Synthesis

DESIGN AND SYNTHESIS OF SYNCHRONIZATION SKELETONS
USING BRANCHING TIME TEMPORAL LOGIC

Edmund M. Clarke
E. Allen Emerson
Aiken Computation Laboratory
Harvard University
Cambridge, Mass. 02138, USA

1. INTRODUCTION

We propose a method of constructing concurrent programs in which the *synchronization skeleton* of the program is automatically synthesized from a high-level (branching time) Temporal Logic specification. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. For example, in the synchronization skeleton for a solution to the critical section problem each process's critical section may be viewed as a single node since the internal structure of the critical section is unimportant. Most solutions to synchronization problems in the literature are in fact given as synchronization skeletons. Because synchronization skeletons are in general finite state, the propositional version of Temporal Logic can be used to specify their properties.

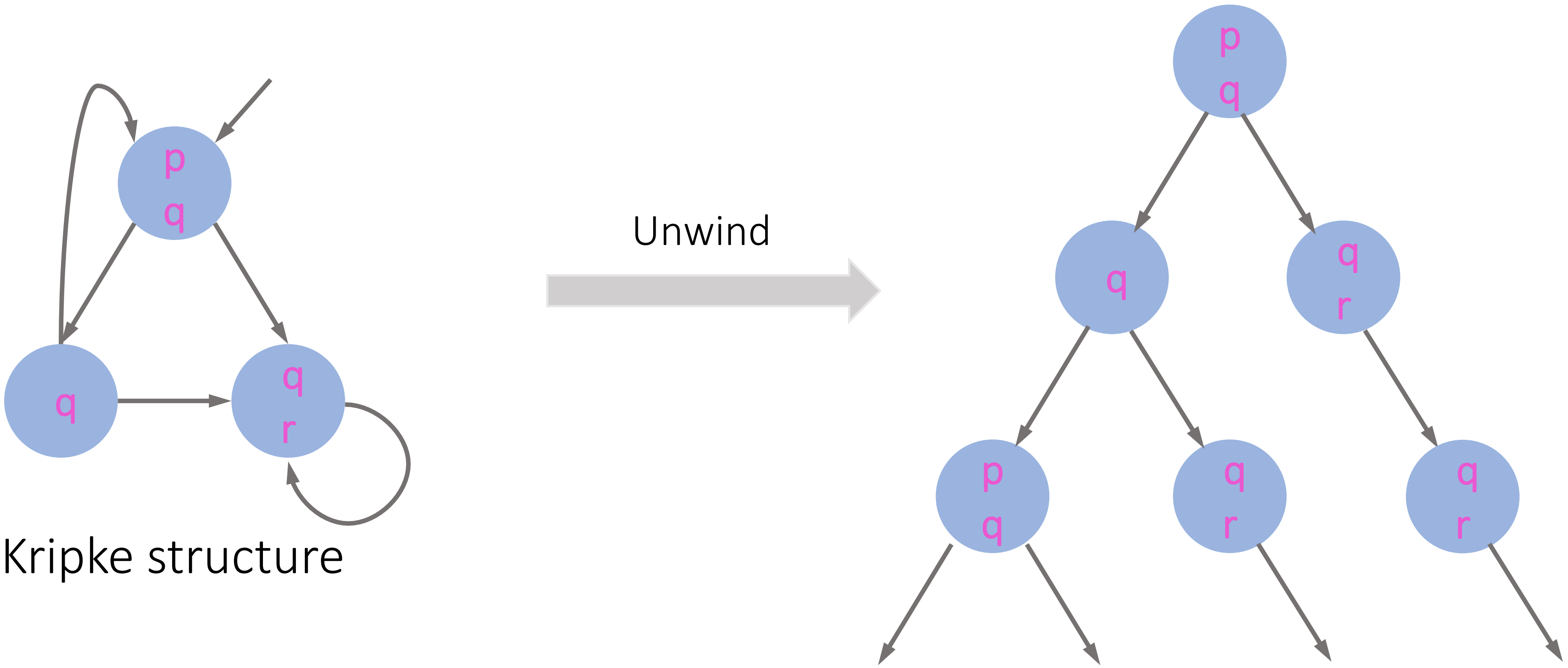
Our synthesis method exploits the (bounded) *finite model property* for an appropriate propositional Temporal Logic which asserts that if a formula of the logic is satisfiable, it is satisfiable in a finite model (of size bounded by a function of the length of the formula). Decision procedures have been devised which, given a formula of Temporal Logic, f, will decide whether f is satisfiable or unsatisfiable. If f is satisfiable, a finite model of f is constructed. In our application, unsatisfiability of f means that the specification is inconsistent (and must be reformulated). If the formula f is satisfiable, then the specification it expresses is consistent. A model for f with a finite number of states is constructed by the decision procedure. The synchronization skeleton of a program meeting the specification can be read from this model. The finite model property ensures that any program whose synchronization properties can be expressed in propositional Temporal Logic can be realized by a system of concurrently running processes, each of which is a finite state machine.

Initially, the synchronization skeletons we synthesize will be for concurrent programs running in a shared-memory environment and for monitors. However, we believe that it is also possible to extend these techniques to synthesize distributed programs. One such application would be the automatic synthesis of network communication protocols from propositional Temporal Logic specifications.

Previous efforts toward parallel program synthesis can be found in the work of [LA78] and [RK80]. [LA78] uses a specification language that is essentially predicate

This work was partially supported by NSF Grant MCS-7908365.

Temporal logic primer



Kripke structure

Computation tree

Temporal logics describe properties of infinite computation trees

Syntax of CTL

CTL /State formula

$g ::= p \mid \neg g \mid g_1 \vee g_2 \mid g_1 \wedge g_2 \mid \mathbf{A}\, f \mid \mathbf{E}\, f$

Path formula:

$f ::= \mathbf{X}\, g \mid \mathbf{F}\, g \mid \mathbf{G}\, g \mid g_1\, \mathbf{U}\, g_2$

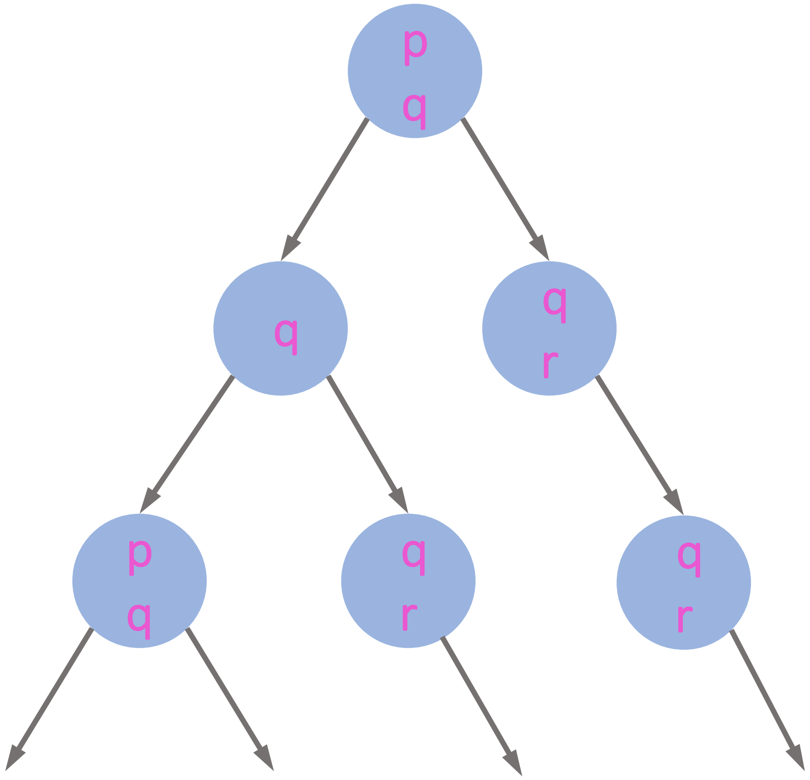
Path quantifiers

Always | Exists

Temporal operators

Nexttime | Eventually

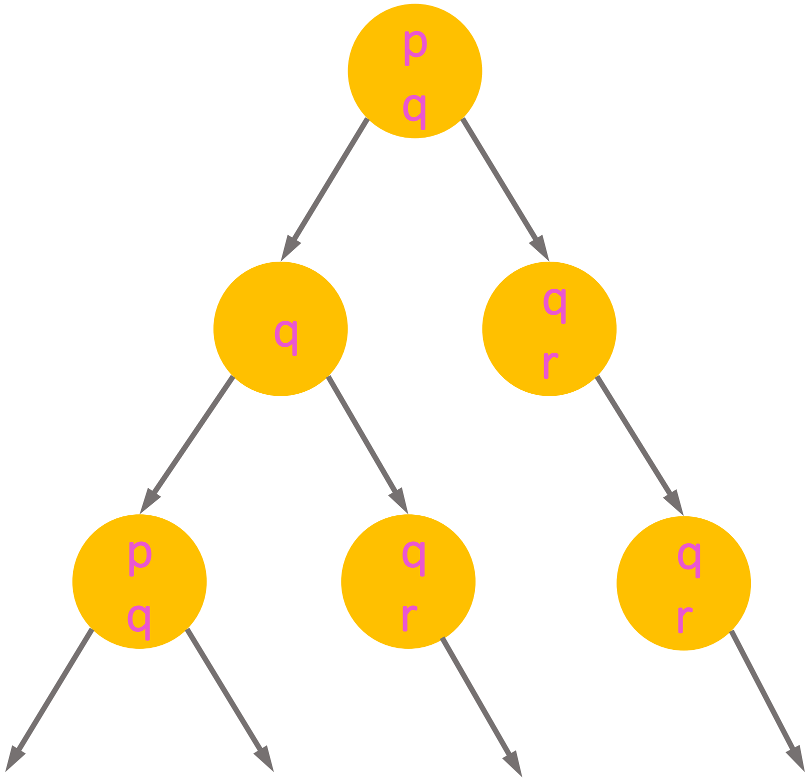
Globally | Until



Computation tree

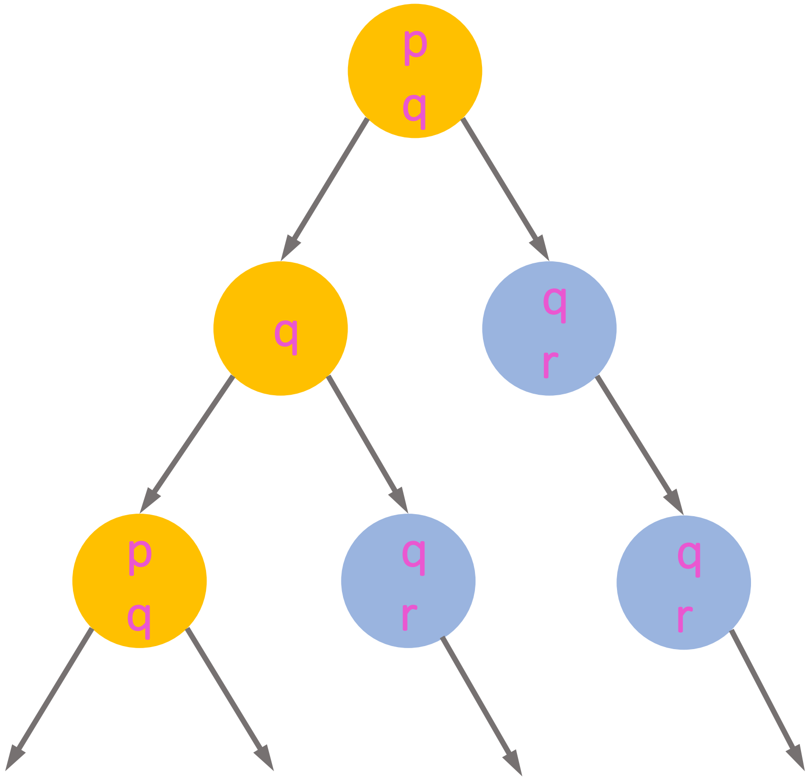
AG q

Along all paths, q holds in every state



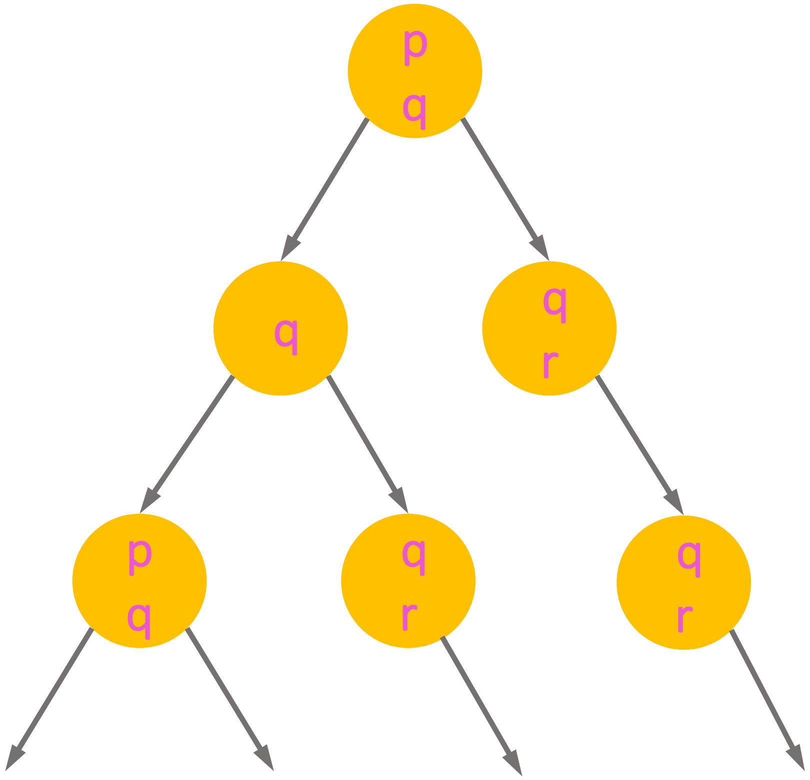
EF p

Exists a path, p holds eventually



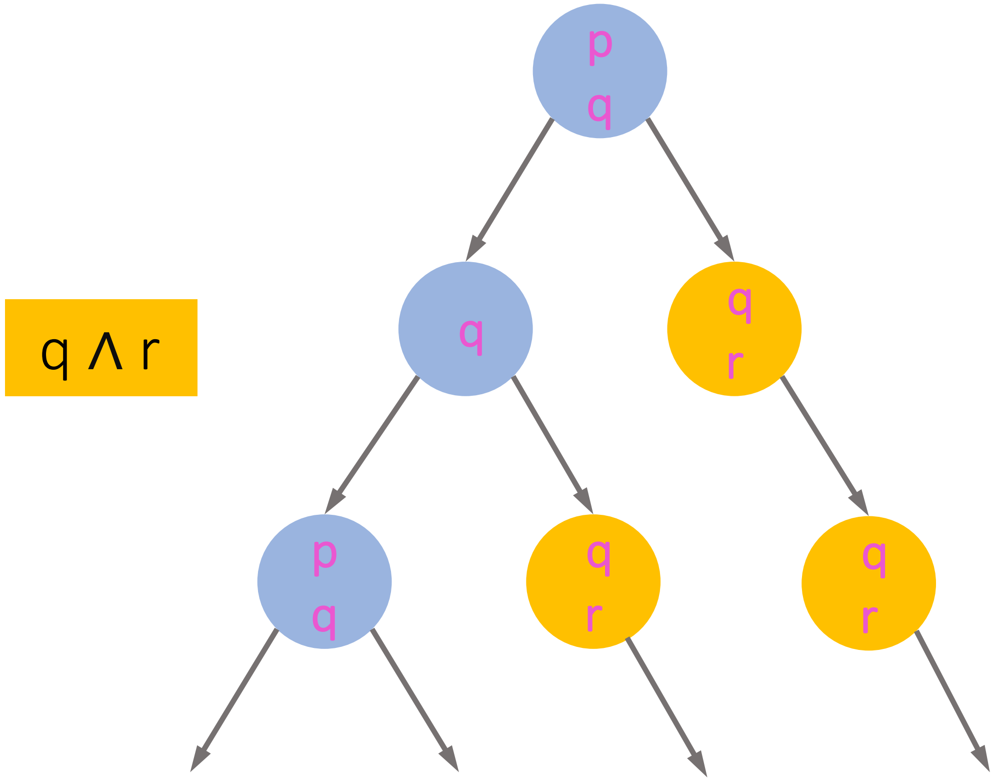
EF AG q ∧ r

Exists a path, **AG** q ∧ r holds eventually



EF AG $q \wedge r$

Exists a path, **AG** $q \wedge r$ holds eventually

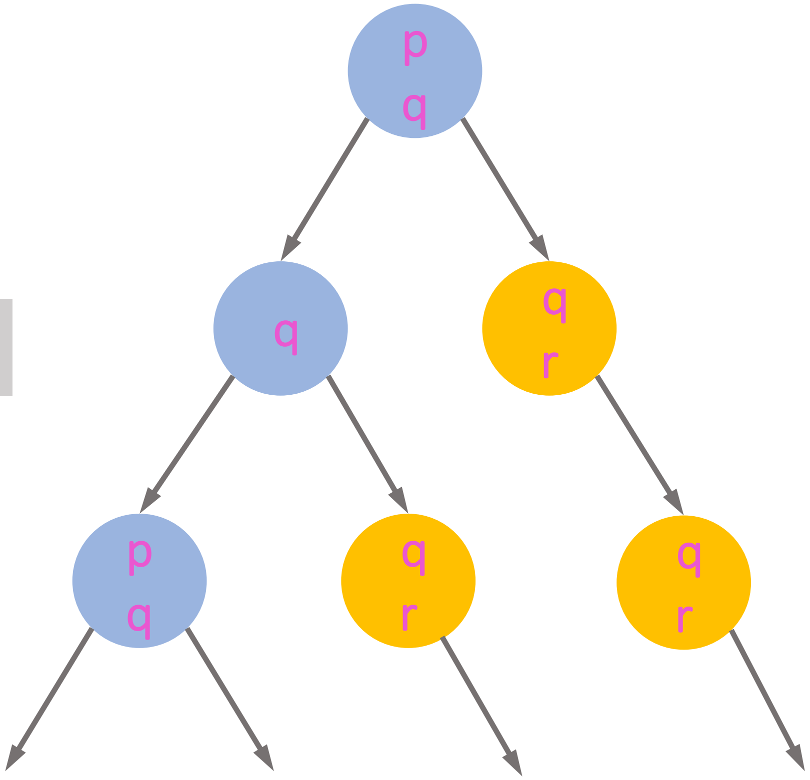


EF AG q ∧ r

AG q ∧ r

q ∧ r

Exists a path, **AG** q ∧ r holds eventually

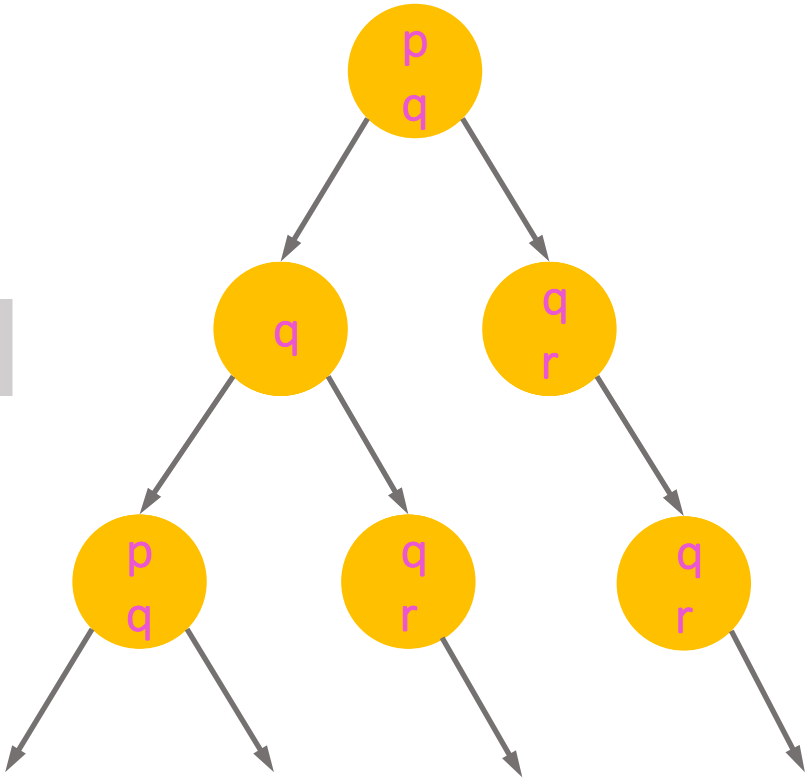


EF AG q ∧ r

AG q ∧ r

q ∧ r

Exists a path, **AG** q ∧ r holds eventually



CTL synthesis decision procedure

Input: CTL formula f
Output: SAT + a finite model of f, or, UNSAT

- Build a tableau encoding potential models of f
- Delete inconsistent portions
- If root node is deleted, return UNSAT
- Extract model of f from tableau. Return SAT + model

Tableau for **EF** p ∧ **EF** ¬p

node ⊨ f for all f ∈ label(node)

OR node

AND node

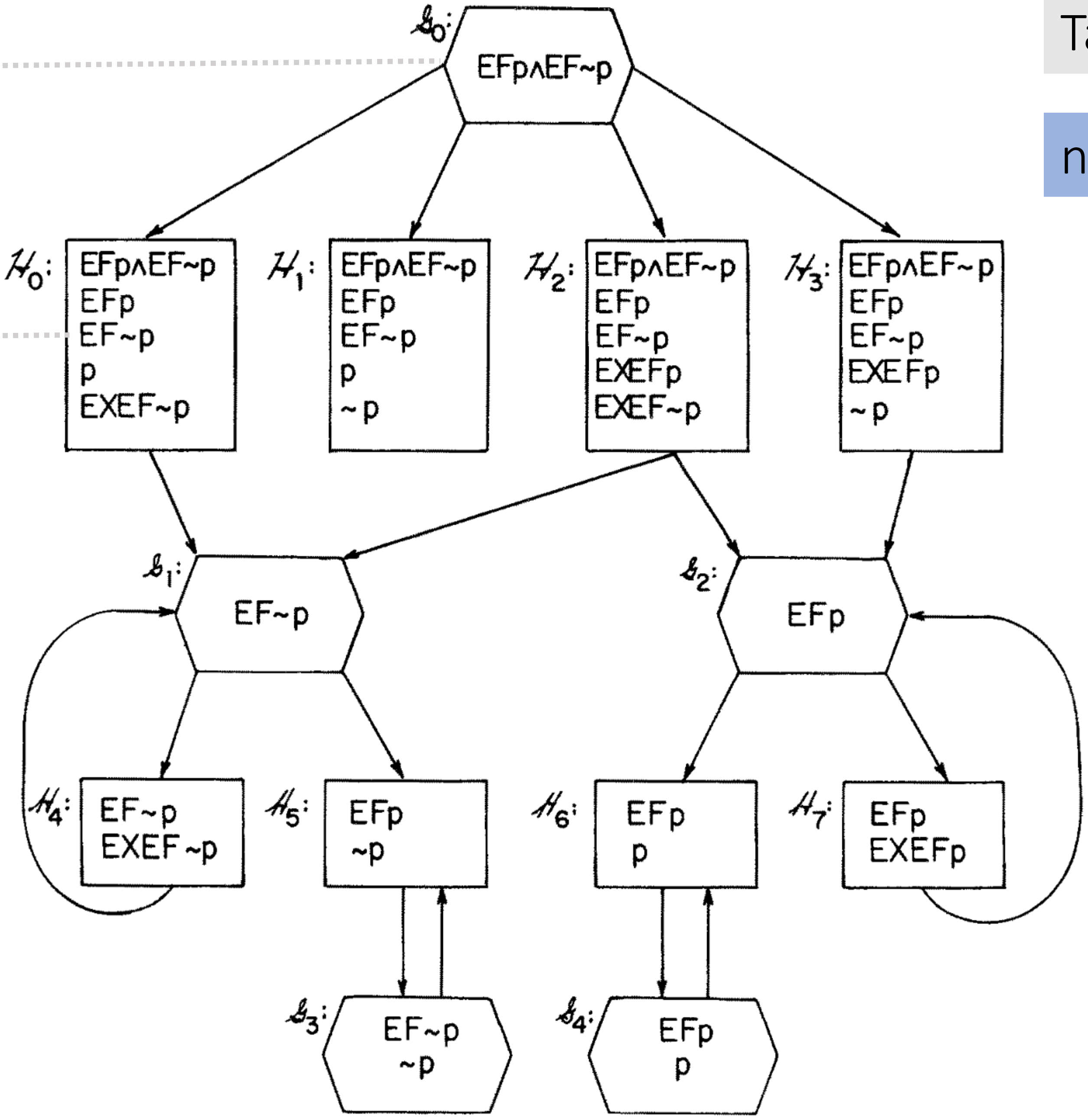
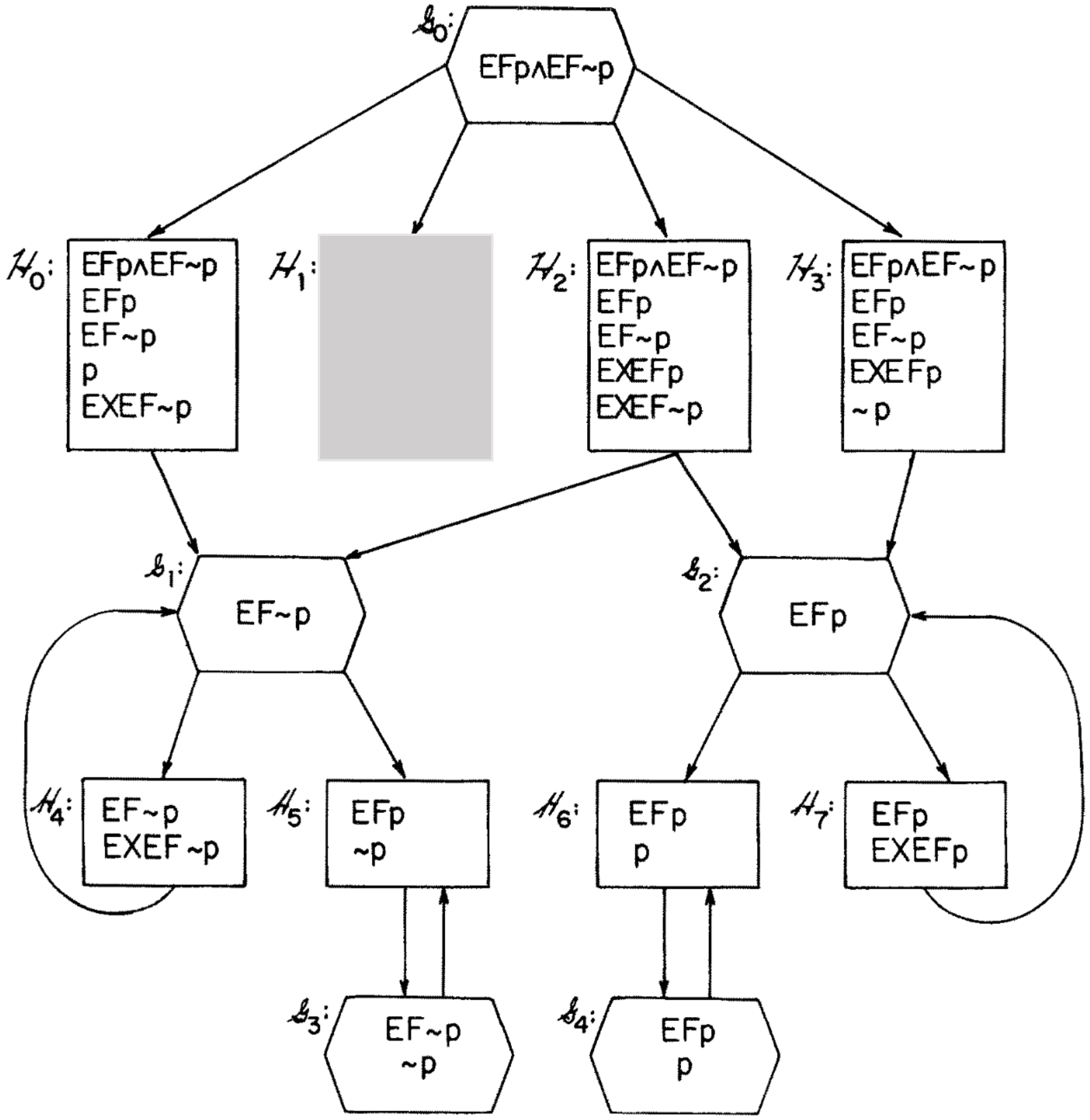
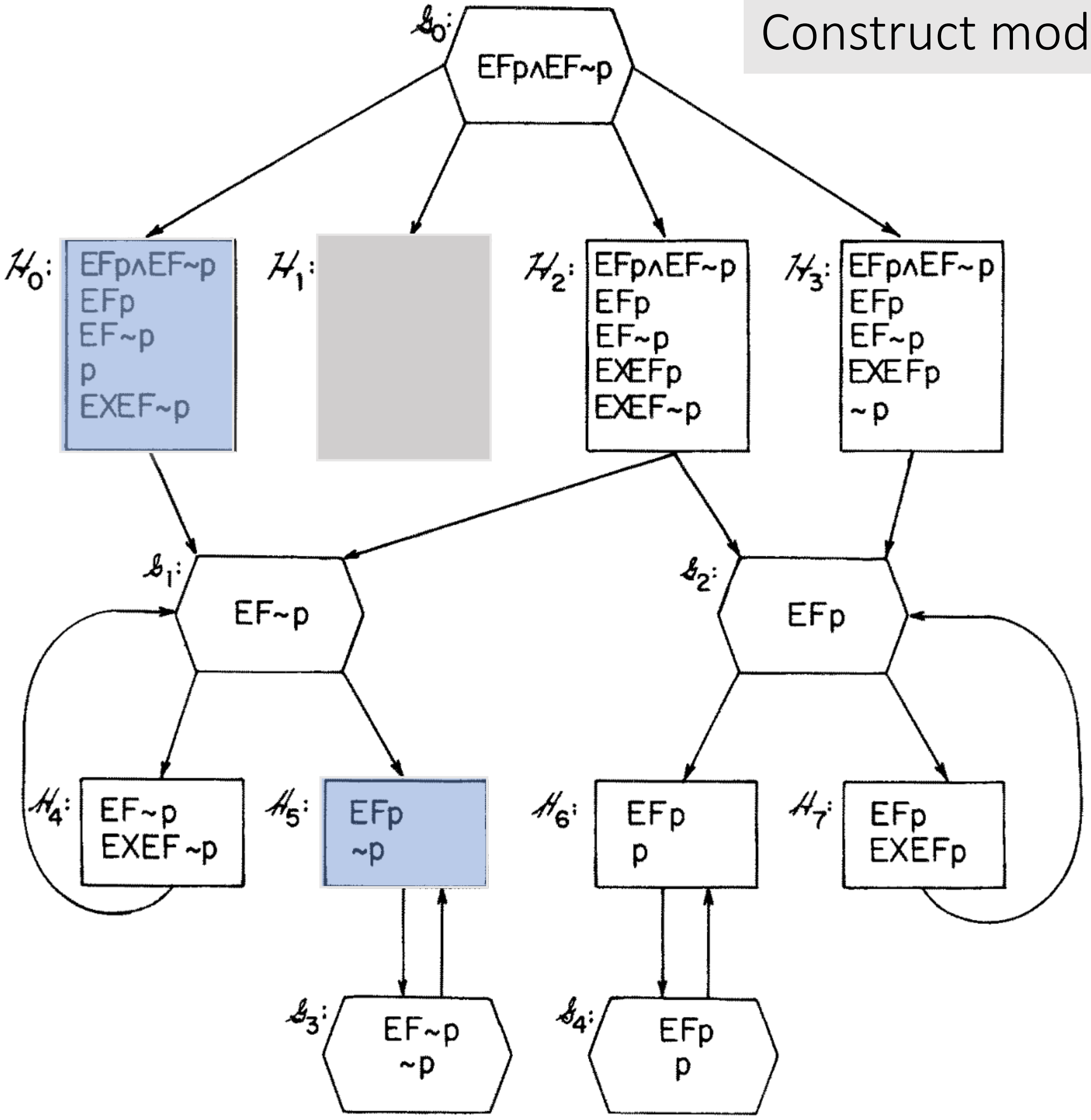


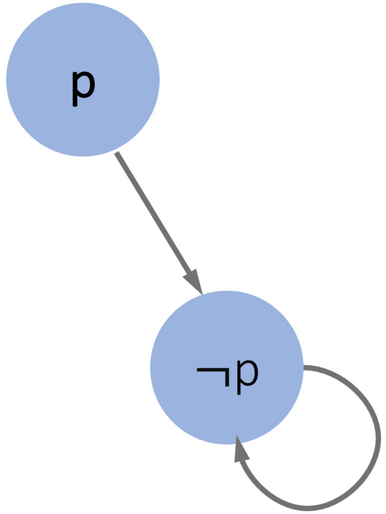
Figure from [CE81]

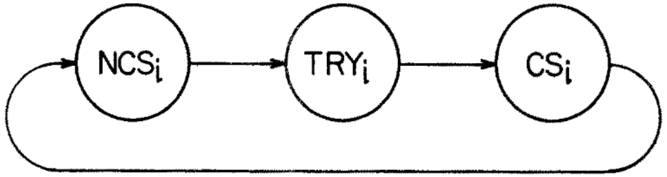
Delete inconsistent portions



Construct model from AND-nodes of tableau







Mutual exclusion:

$\mathbf{AG}\ \neg(\mathrm{CS}_1 \wedge \mathrm{CS}_2)$

Absence of starvation:

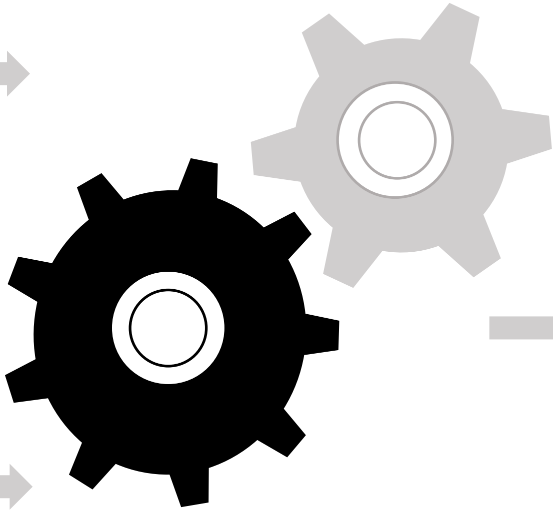
$\mathbf{AG}\ \mathrm{TRY}_i \rightarrow \mathbf{AF}\ \mathrm{CS}_i$

Process specification:

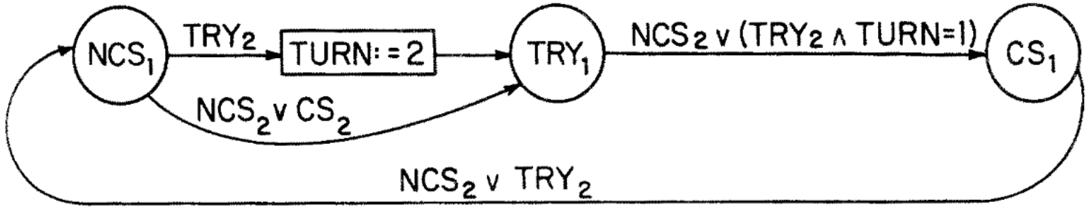
$\mathbf{AG}\ \mathrm{NCS}_i \vee \mathrm{TRY}_i \vee \mathrm{CS}_i$

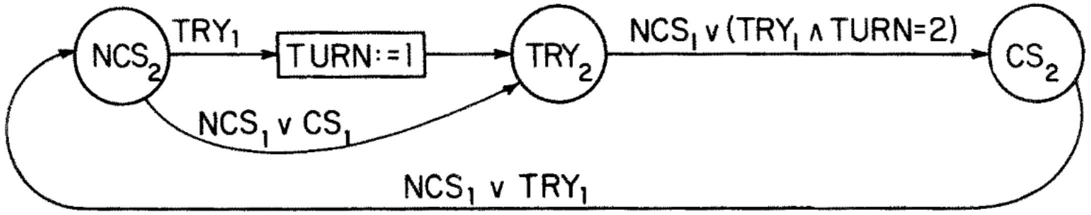
$\mathbf{AG}\ \mathrm{NCS}_i \rightarrow \neg(\mathrm{TRY}_i \vee \mathrm{CS}_i)$

...



Synchronization Synthesizer





- **Process:**
  Finite-state synchronization skeleton
- **Communication Model:**
  Shared-memory, interleaving-based
- **Specification:**
  Temporal logic, complete
- **Synchronization:**
  Guarded commands
- **Procedure:**
  Tableau-based decision procedure

- Computation Tree Logic (CTL)
- Model Checking for CTL
- CTL Synthesis

DESIGN AND SYNTHESIS OF SYNCHRONIZATION SKELETONS
USING BRANCHING TIME TEMPORAL LOGIC

Edmund M. Clarke
E. Allen Emerson
Aiken Computation Laboratory
Harvard University
Cambridge, Mass. 02138, USA

1. INTRODUCTION

We propose a method of constructing concurrent programs in which the *synchronization skeleton* of the program is automatically synthesized from a high-level (branching time) Temporal Logic specification. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. For example, in the synchronization skeleton for a solution to the critical section problem each process's critical section may be viewed as a single node since the internal structure of the critical section is unimportant. Most solutions to synchronization problems in the literature are in fact given as synchronization skeletons. Because synchronization skeletons are in general finite state, the propositional version of Temporal Logic can be used to specify their properties.

Our synthesis method exploits the (bounded) *finite model property* for an appropriate propositional Temporal Logic which asserts that if a formula of the logic is satisfiable, it is satisfiable in a finite model (of size bounded by a function of the length of the formula). Decision procedures have been devised which, given a formula of Temporal Logic, f, will decide whether f is satisfiable or unsatisfiable. If f is satisfiable, a finite model of f is constructed. In our application, unsatisfiability of f means that the specification is inconsistent (and must be reformulated). If the formula f is satisfiable, then the specification it expresses is consistent. A model for f with a finite number of states is constructed by the decision procedure. The synchronization skeleton of a program meeting the specification can be read from this model. The finite model property ensures that any program whose synchronization properties can be expressed in propositional Temporal Logic can be realized by a system of concurrently running processes, each of which is a finite state machine.

Initially, the synchronization skeletons we synthesize will be for concurrent programs running in a shared-memory environment and for monitors. However, we believe that it is also possible to extend these techniques to synthesize distributed programs. One such application would be the automatic synthesis of network communication protocols from propositional Temporal Logic specifications.

Previous efforts toward parallel program synthesis can be found in the work of [LA78] and [RK80]. [LA78] uses a specification language that is essentially predicate

This work was partially supported by NSF Grant MCS-7908365.

- Needs complete specification
- Finite-state processes
- Interleaving explosion

DESIGN AND SYNTHESIS OF SYNCHRONIZATION SKELETONS
USING BRANCHING TIME TEMPORAL LOGIC

Edmund M. Clarke
E. Allen Emerson
Aiken Computation Laboratory
Harvard University
Cambridge, Mass. 02138, USA

1. INTRODUCTION

We propose a method of constructing concurrent programs in which the *synchronization skeleton* of the program is automatically synthesized from a high-level (branching time) Temporal Logic specification. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. For example, in the synchronization skeleton for a solution to the critical section problem each process's critical section may be viewed as a single node since the internal structure of the critical section is unimportant. Most solutions to synchronization problems in the literature are in fact given as synchronization skeletons. Because synchronization skeletons are in general finite state, the propositional version of Temporal Logic can be used to specify their properties.

Our synthesis method exploits the (bounded) *finite model property* for an appropriate propositional Temporal Logic which asserts that if a formula of the logic is satisfiable, it is satisfiable in a finite model (of size bounded by a function of the length of the formula). Decision procedures have been devised which, given a formula of Temporal Logic, f, will decide whether f is satisfiable or unsatisfiable. If f is satisfiable, a finite model of f is constructed. In our application, unsatisfiability of f means that the specification is inconsistent (and must be reformulated). If the formula f is satisfiable, then the specification it expresses is consistent. A model for f with a finite number of states is constructed by the decision procedure. The synchronization skeleton of a program meeting the specification can be read from this model. The finite model property ensures that any program whose synchronization properties can be expressed in propositional Temporal Logic can be realized by a system of concurrently running processes, each of which is a finite state machine.

Initially, the synchronization skeletons we synthesize will be for concurrent programs running in a shared-memory environment and for monitors. However, we believe that it is also possible to extend these techniques to synthesize distributed programs. One such application would be the automatic synthesis of network communication protocols from propositional Temporal Logic specifications.

Previous efforts toward parallel program synthesis can be found in the work of [LA78] and [RK80]. [LA78] uses a specification language that is essentially predicate

This work was partially supported by NSF Grant MCS-7908365.

A seminal paper

A cool paper

A modern approach

A cool paper

Vechev

Yahav

Yorsh

Abstraction-Guided Synthesis of Synchronization. POPL 2010.

Abstraction-Guided Synthesis of Synchronization

Martin Vechev
IBM Research

Eran Yahav
IBM Research

Greta Yorsh
IBM Research

Abstract

We present a novel framework for automatic inference of efficient synchronization in concurrent programs, a task known to be difficult and error-prone when done manually.

Our framework is based on abstract interpretation and can infer synchronization for infinite state programs. Given a program, a specification, and an abstraction, we infer synchronization that avoids all (abstract) interleavings that may violate the specification, but permits as many valid interleavings as possible.

Combined with abstraction refinement, our framework can be viewed as a new approach for verification where both the program and the abstraction can be modified on-the-fly during the verification process. The ability to modify the program, and not only the abstraction, allows us to remove program interleavings not only when they are known to be invalid, but also when they cannot be verified using the given abstraction.

We implemented a prototype of our approach using numerical abstractions and applied it to verify several interesting programs.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]
General Terms Algorithms, Verification
Keywords concurrency, synthesis, abstract interpretation

1. Introduction

We present *abstraction-guided synthesis*, a novel approach for synthesizing efficient synchronization in concurrent programs. Our approach turns the one dimensional problem of verification under abstraction, in which only the abstraction can be modified (typically via abstraction refinement), into a two-dimensional problem, in which *both the program and the abstraction can be modified* until the abstraction is precise enough to verify the program.

Based on abstract interpretation [10], our technique synthesizes a symbolic characterization of *safe schedules* for concurrent infinite-state programs. Safe schedules can be realized by modifying the program or the scheduler:

- Concurrent programming: by automatically inferring minimal atomic sections that prevent unsafe schedules, we assist the programmer in building correct and efficient concurrent software, a task known to be difficult and error-prone.
- Benevolent runtime: a scheduler that always keeps the program execution on a safe schedule makes the runtime system more reliable and adaptive to ever-changing environment and safety requirements, without the need to modify the program.

Given a program P, a specification S, and an abstraction function α, verification determines whether $P \models_\alpha S$, that is, whether P satisfies the specification S under the abstraction α. When the answer to this question is negative, it may be the case that the program violates the specification, or that the abstraction α is not precise enough to show that the program satisfies it.

When $P \not\models_\alpha S$, abstraction refinement approaches (e.g., [3, 8]) share the common goal of trying to find a finer abstraction α' such that $P \models_{\alpha'} S$. In this paper, we investigate a complementary approach, of finding a program P' such that $P' \models_\alpha S$ under the original abstraction α and P' admits a subset of the behaviors of P. Furthermore, we combine the two directions — refining the abstraction, and restricting program behaviors, to yield a novel abstraction-guided synthesis algorithm.

One of the main challenges in our approach is to devise an algorithm for obtaining such P' from the initial program P. In this paper, we focus on *concurrent programs*, and consider changes to P that correspond to restricting interleavings by adding synchronization.

Although it is possible to apply our techniques to other settings, concurrent programs are a natural fit. Concurrent programs are often correct on most interleavings and only miss synchronization in a few corner cases, which can be then avoided by synthesizing additional synchronization. Furthermore, in many cases, constraining the permitted interleavings reduces the set of reachable (abstract) states, possibly enabling verification via a coarser abstraction and avoiding state-space explosion.

The AGS algorithm, presented in Section 4, iteratively eliminates invalid interleavings until the abstraction is precise enough to verify the program. Some of the (abstract) invalid interleavings it observes may correspond to concrete invalid interleavings, while others may be artifacts of the abstraction. Whenever the algorithm observes an (abstract) invalid interleaving, the algorithm tries to eliminate it by either (i) modifying the program, or (ii) refining the abstraction.

To refine the abstraction, the algorithm can use any standard technique (e.g.,[3, 8]). These include moving through a predetermined series of domains with increasing precision (and typically increasing cost), or refining within the same abstract domain by changing its parameters (e.g., [4]).

To modify the program, we provide a novel algorithm that generates and solves *atomicity constraints*. Atomicity constraints define which statements have to be executed atomically, without an intermediate context switch, to eliminate the invalid interleavings. This corresponds to limiting the non-deterministic choices available to the scheduler. A solution of the atomicity constraints can be implemented by adding atomic sections to the program.

Our approach separates the process of identifying the space of solutions (generating the atomicity constraints) from the process of choosing between the possible solutions, which can be based on a quantitative criterion. As we discuss in Section 6, our approach provides a solution to a *quantitative synthesis* problem [5], as it

POPL'10, January 17–23, 2009, Madrid, Spain.

Abstraction-based approach to infer synchronization to ensure safety properties of infinite-state concurrent programs

Abstraction-Guided Synthesis of Synchronization

Martin Vechev
IBM Research

Eran Yahav
IBM Research

Greta Yorsh
IBM Research

Abstract

We present a novel framework for automatic inference of efficient synchronization in concurrent programs, a task known to be difficult and error-prone when done manually.

Our framework is based on abstract interpretation and can infer synchronization for infinite state programs. Given a program, a specification, and an abstraction, we infer synchronization that avoids all (abstract) interleavings that may violate the specification, but permits as many valid interleavings as possible.

Combined with abstraction refinement, our framework can be viewed as a new approach for verification where both the program and the abstraction can be modified on-the-fly during the verification process. The ability to modify the program, and not only the abstraction, allows us to remove program interleavings not only when they are known to be invalid, but also when they cannot be verified using the given abstraction.

We implemented a prototype of our approach using numerical abstractions and applied it to verify several interesting programs.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]
General Terms Algorithms, Verification
Keywords concurrency, synthesis, abstract interpretation

1. Introduction

We present *abstraction-guided synthesis*, a novel approach for synthesizing efficient synchronization in concurrent programs. Our approach turns the one dimensional problem of verification under abstraction, in which only the abstraction can be modified (typically via abstraction refinement), into a two-dimensional problem, in which *both the program and the abstraction can be modified* until the abstraction is precise enough to verify the program.

Based on abstract interpretation [10], our technique synthesizes a symbolic characterization of *safe schedules* for concurrent infinite-state programs. Safe schedules can be realized by modifying the program or the scheduler:

- Concurrent programming: by automatically inferring minimal atomic sections that prevent unsafe schedules, we assist the programmer in building correct and efficient concurrent software, a task known to be difficult and error-prone.
- Benevolent runtime: a scheduler that always keeps the program execution on a safe schedule makes the runtime system more reliable and adaptive to ever-changing environment and safety requirements, without the need to modify the program.

Given a program P, a specification S, and an abstraction function α, verification determines whether $P \models_\alpha S$, that is, whether P satisfies the specification S under the abstraction α. When the answer to this question is negative, it may be the case that the program violates the specification, or that the abstraction α is not precise enough to show that the program satisfies it.

When $P \not\models_\alpha S$, abstraction refinement approaches (e.g., [3, 8]) share the common goal of trying to find a finer abstraction α' such that $P \models_{\alpha'} S$. In this paper, we investigate a complementary approach, of finding a program P' such that $P' \models_\alpha S$ under the original abstraction α and P' admits a subset of the behaviors of P. Furthermore, we combine the two directions — refining the abstraction, and restricting program behaviors, to yield a novel abstraction-guided synthesis algorithm.

One of the main challenges in our approach is to devise an algorithm for obtaining such P' from the initial program P. In this paper, we focus on *concurrent programs*, and consider changes to P that correspond to restricting interleavings by adding synchronization.

Although it is possible to apply our techniques to other settings, concurrent programs are a natural fit. Concurrent programs are often correct on most interleavings and only miss synchronization in a few corner cases, which can be then avoided by synthesizing additional synchronization. Furthermore, in many cases, constraining the permitted interleavings reduces the set of reachable (abstract) states, possibly enabling verification via a coarser abstraction and avoiding state-space explosion.

The AGS algorithm, presented in Section 4, iteratively eliminates invalid interleavings until the abstraction is precise enough to verify the program. Some of the (abstract) invalid interleavings it observes may correspond to concrete invalid interleavings, while others may be artifacts of the abstraction. Whenever the algorithm observes an (abstract) invalid interleaving, the algorithm tries to eliminate it by either (i) modifying the program, or (ii) refining the abstraction.

To refine the abstraction, the algorithm can use any standard technique (e.g.,[3, 8]). These include moving through a predetermined series of domains with increasing precision (and typically increasing cost), or refining within the same abstract domain by changing its parameters (e.g., [4]).

To modify the program, we provide a novel algorithm that generates and solves *atomicity constraints*. Atomicity constraints define which statements have to be executed atomically, without an intermediate context switch, to eliminate the invalid interleavings. This corresponds to limiting the non-deterministic choices available to the scheduler. A solution of the atomicity constraints can be implemented by adding atomic sections to the program.

Our approach separates the process of identifying the space of solutions (generating the atomicity constraints) from the process of choosing between the possible solutions, which can be based on a quantitative criterion. As we discuss in Section 6, our approach provides a solution to a *quantitative synthesis* problem [5], as it

POPL'10, January 17–23, 2009, Madrid, Spain.

- **Process:**
  Infinite-state program
- **Communication Model:**
  Shared-memory, interleaving-based
- **Specification:**
  Safety property
- **Synchronization:**
  Atomic section
- **Procedure:**
  Abstraction-refinement & counterexample-based

Abstraction-Guided Synthesis of Synchronization

Martin Vechev, IBM Research
Eran Yahav, IBM Research
Greta Yorsh, IBM Research

Abstract

We present a novel framework for automatic inference of efficient synchronization in concurrent programs, a task known to be difficult and error-prone when done manually.

Our framework is based on abstract interpretation and can infer synchronization for infinite state programs. Given a program, a specification, and an abstraction, we infer synchronization that avoids all (abstract) interleavings that may violate the specification, but permits as many valid interleavings as possible.

Combined with abstraction refinement, our framework can be viewed as a new approach for verification where both the program and the abstraction can be modified on-the-fly during the verification process. The ability to modify the program, and not only the abstraction, allows us to remove program interleavings not only when they are known to be invalid, but also when they cannot be verified using the given abstraction.

We implemented a prototype of our approach using numerical abstractions and applied it to verify several interesting programs.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]

General Terms Algorithms, Verification

Keywords concurrency, synthesis, abstract interpretation

1. Introduction

We present *abstraction-guided synthesis*, a novel approach for synthesizing efficient synchronization in concurrent programs. Our approach turns the one dimensional problem of verification under abstraction, in which only the abstraction can be modified (typically via abstraction refinement), into a two-dimensional problem, in which *both the program and the abstraction can be modified* until the abstraction is precise enough to verify the program.

Based on abstract interpretation [10], our technique synthesizes a symbolic characterization of *safe schedules* for concurrent infinite-state programs. Safe schedules can be realized by modifying the program or the scheduler:

- Concurrent programming: by automatically inferring minimal atomic sections that prevent unsafe schedules, we assist the programmer in building correct and efficient concurrent software, a task known to be difficult and error-prone.
- Benevolent runtime: a scheduler that always keeps the program execution on a safe schedule makes the runtime system more reliable and adaptive to ever-changing environment and safety requirements, without the need to modify the program.

POPL'10, January 17–23, 2009, Madrid, Spain.

Given a program P, a specification S, and an abstraction function α, verification determines whether $P \models_\alpha S$, that is, whether P satisfies the specification S under the abstraction α. When the answer to this question is negative, it may be the case that the program violates the specification, or that the abstraction α is not precise enough to show that the program satisfies it.

When $P \not\models_\alpha S$, abstraction refinement approaches (e.g., [3, 8]) share the common goal of trying to find a finer abstraction α' such that $P \models_{\alpha'} S$. In this paper, we investigate a complementary approach, of finding a program P' such that $P' \models_\alpha S$ under the original abstraction α and P' admits a subset of the behaviors of P. Furthermore, we combine the two directions — refining the abstraction, and restricting program behaviors, to yield a novel abstraction-guided synthesis algorithm.

One of the main challenges in our approach is to devise an algorithm for obtaining such P' from the initial program P. In this paper, we focus on *concurrent programs*, and consider changes to P that correspond to restricting interleavings by adding synchronization.

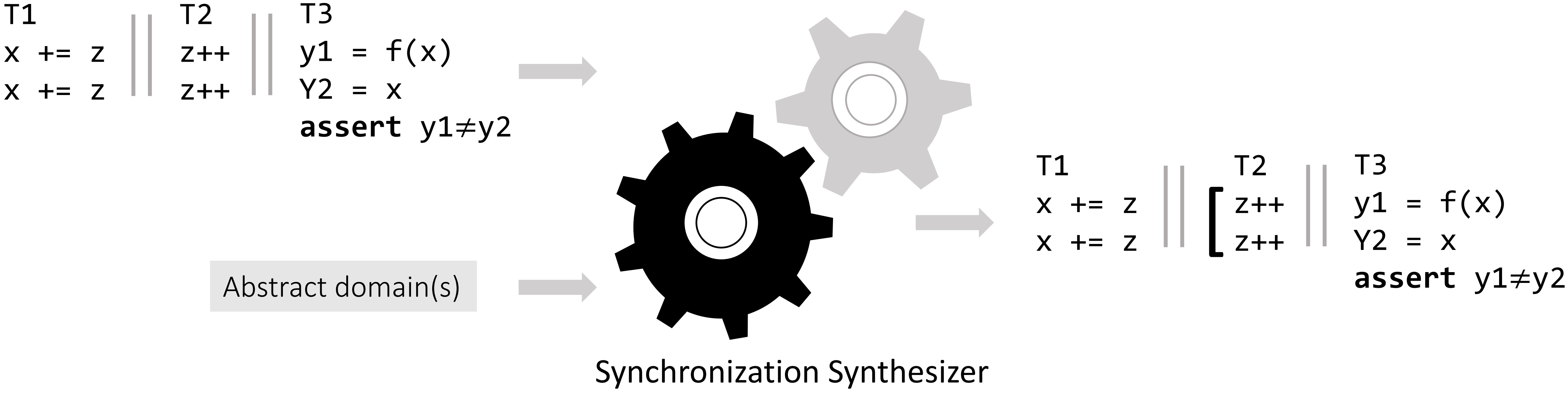
Although it is possible to apply our techniques to other settings, concurrent programs are a natural fit. Concurrent programs are often correct on most interleavings and only miss synchronization in a few corner cases, which can be then avoided by synthesizing additional synchronization. Furthermore, in many cases, constraining the permitted interleavings reduces the set of reachable (abstract) states, possibly enabling verification via a coarser abstraction and avoiding state-space explosion.

The AGS algorithm, presented in Section 4, iteratively eliminates invalid interleavings until the abstraction is precise enough to verify the program. Some of the (abstract) invalid interleavings it observes may correspond to concrete invalid interleavings, while others may be artifacts of the abstraction. Whenever the algorithm observes an (abstract) invalid interleaving, the algorithm tries to eliminate it by either (i) modifying the program, or (ii) refining the abstraction.

To refine the abstraction, the algorithm can use any standard technique (e.g.,[3, 8]). These include moving through a predetermined series of domains with increasing precision (and typically increasing cost), or refining within the same abstract domain by changing its parameters (e.g., [4]).

To modify the program, we provide a novel algorithm that generates and solves *atomicity constraints*. Atomicity constraints define which statements have to be executed atomically, without an intermediate context switch, to eliminate the invalid interleavings. This corresponds to limiting the non-deterministic choices available to the scheduler. A solution of the atomicity constraints can be implemented by adding atomic sections to the program.

Our approach separates the process of identifying the space of solutions (generating the atomicity constraints) from the process of choosing between the possible solutions, which can be based on a quantitative criterion. As we discuss in Section 6, our approach provides a solution to a *quantitative synthesis* problem [5], as it


T1
x += z
x += z
T2
z++
z++
T3
y1 = f(x)
Y2 = x
assert y1≠y2
Abstract domain(s)
Synchronization Synthesizer
T1
x += z
x += z
T2
z++
z++
T3
y1 = f(x)
Y2 = x
assert y1≠y2

- **Process:**
  Infinite-state program
- **Communication Model:**
  Shared-memory, interleaving-based
- **Specification:**
  Safety property
- **Synchronization:**
  Atomic section
- **Procedure:**
  Abstraction-refinement & counterexample-based

- Abstraction-guided synthesis
- Synthesis as repair
- Quantitative synthesis

Abstraction-Guided Synthesis of Synchronization

Martin Vechev
IBM Research

Eran Yahav
IBM Research

Greta Yorsh
IBM Research

Abstract

We present a novel framework for automatic inference of efficient synchronization in concurrent programs, a task known to be difficult and error-prone when done manually.

Our framework is based on abstract interpretation and can infer synchronization for infinite state programs. Given a program, a specification, and an abstraction, we infer synchronization that avoids all (abstract) interleavings that may violate the specification, but permits as many valid interleavings as possible.

Combined with abstraction refinement, our framework can be viewed as a new approach for verification where both the program and the abstraction can be modified on-the-fly during the verification process. The ability to modify the program, and not only the abstraction, allows us to remove program interleavings not only when they are known to be invalid, but also when they cannot be verified using the given abstraction.

We implemented a prototype of our approach using numerical abstractions and applied it to verify several interesting programs.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]
General Terms Algorithms, Verification
Keywords concurrency, synthesis, abstract interpretation

1. Introduction

We present *abstraction-guided synthesis*, a novel approach for synthesizing efficient synchronization in concurrent programs. Our approach turns the one dimensional problem of verification under abstraction, in which only the abstraction can be modified (typically via abstraction refinement), into a two-dimensional problem, in which *both the program and the abstraction can be modified* until the abstraction is precise enough to verify the program.

Based on abstract interpretation [10], our technique synthesizes a symbolic characterization of *safe schedules* for concurrent infinite-state programs. Safe schedules can be realized by modifying the program or the scheduler:

- Concurrent programming: by automatically inferring minimal atomic sections that prevent unsafe schedules, we assist the programmer in building correct and efficient concurrent software, a task known to be difficult and error-prone.
- Benevolent runtime: a scheduler that always keeps the program execution on a safe schedule makes the runtime system more reliable and adaptive to ever-changing environment and safety requirements, without the need to modify the program.

Given a program P, a specification S, and an abstraction function α, verification determines whether $P \models_\alpha S$, that is, whether P satisfies the specification S under the abstraction α. When the answer to this question is negative, it may be the case that the program violates the specification, or that the abstraction α is not precise enough to show that the program satisfies it.

When $P \not\models_\alpha S$, abstraction refinement approaches (e.g., [3, 8]) share the common goal of trying to find a finer abstraction α' such that $P \models_{\alpha'} S$. In this paper, we investigate a complementary approach, of finding a program P' such that $P' \models_\alpha S$ under the original abstraction α and P' admits a subset of the behaviors of P. Furthermore, we combine the two directions — refining the abstraction, and restricting program behaviors, to yield a novel abstraction-guided synthesis algorithm.

One of the main challenges in our approach is to devise an algorithm for obtaining such P' from the initial program P. In this paper, we focus on *concurrent programs*, and consider changes to P that correspond to restricting interleavings by adding synchronization.

Although it is possible to apply our techniques to other settings, concurrent programs are a natural fit. Concurrent programs are often correct on most interleavings and only miss synchronization in a few corner cases, which can be then avoided by synthesizing additional synchronization. Furthermore, in many cases, constraining the permitted interleavings reduces the set of reachable (abstract) states, possibly enabling verification via a coarser abstraction and avoiding state-space explosion.

The AGS algorithm, presented in Section 4, iteratively eliminates invalid interleavings until the abstraction is precise enough to verify the program. Some of the (abstract) invalid interleavings it observes may correspond to concrete invalid interleavings, while others may be artifacts of the abstraction. Whenever the algorithm observes an (abstract) invalid interleaving, the algorithm tries to eliminate it by either (i) modifying the program, or (ii) refining the abstraction.

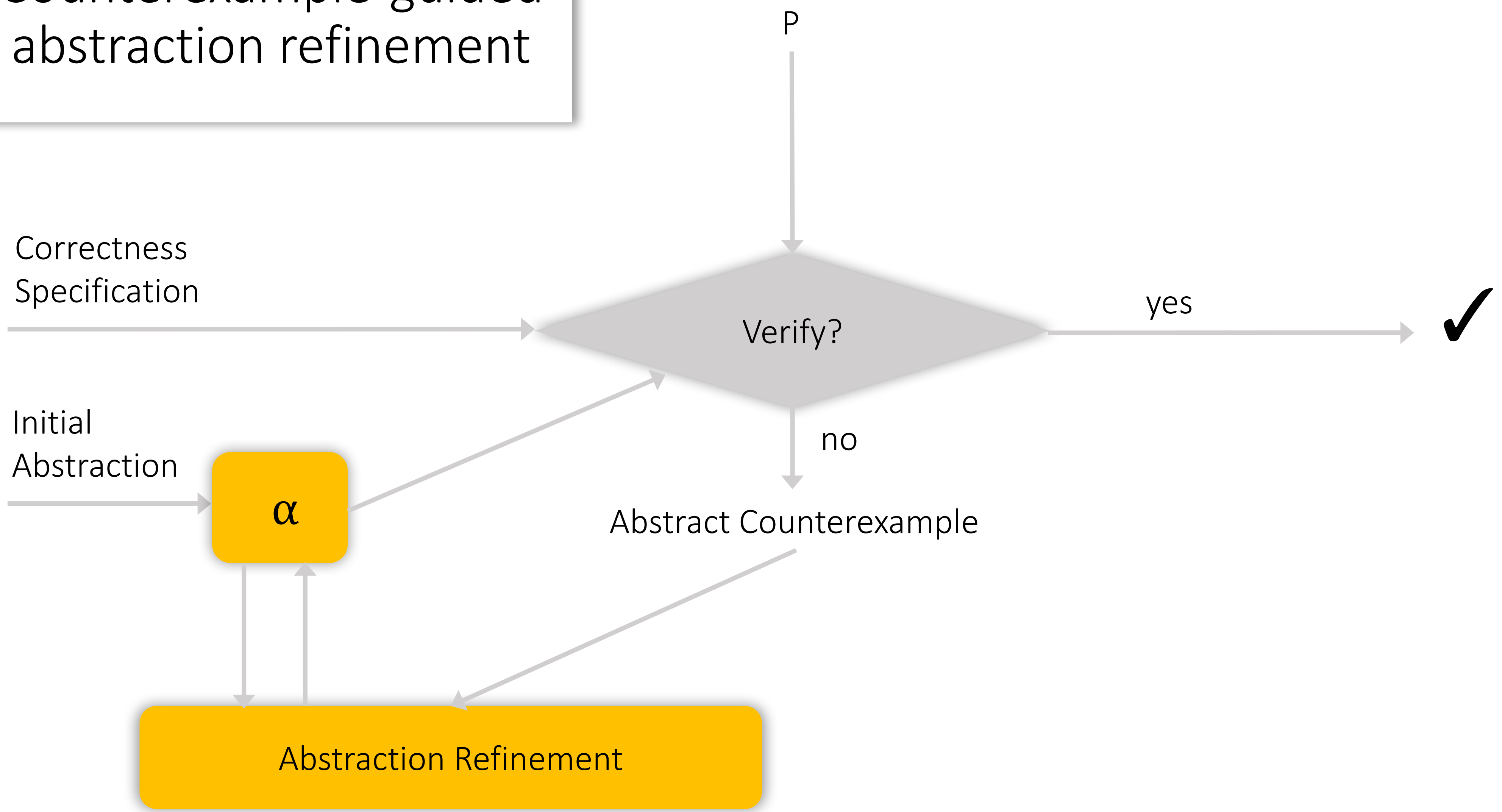
To refine the abstraction, the algorithm can use any standard technique (e.g.,[3, 8]). These include moving through a predetermined series of domains with increasing precision (and typically increasing cost), or refining within the same abstract domain by changing its parameters (e.g., [4]).

To modify the program, we provide a novel algorithm that generates and solves *atomicity constraints*. Atomicity constraints define which statements have to be executed atomically, without an intermediate context switch, to eliminate the invalid interleavings. This corresponds to limiting the non-deterministic choices available to the scheduler. A solution of the atomicity constraints can be implemented by adding atomic sections to the program.

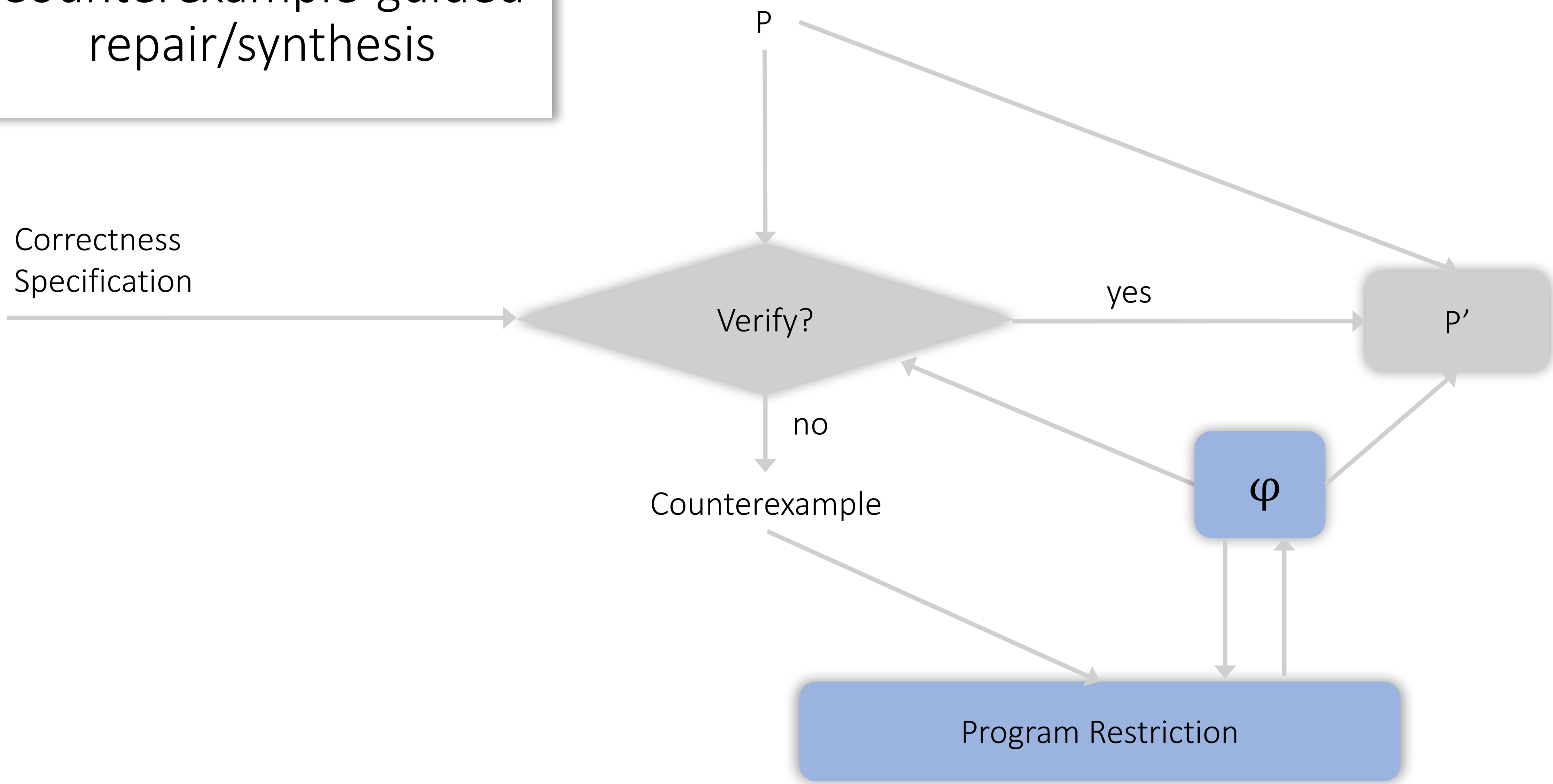
Our approach separates the process of identifying the space of solutions (generating the atomicity constraints) from the process of choosing between the possible solutions, which can be based on a quantitative criterion. As we discuss in Section 6, our approach provides a solution to a *quantitative synthesis* problem [5], as it

POPL'10, January 17–23, 2009, Madrid, Spain.

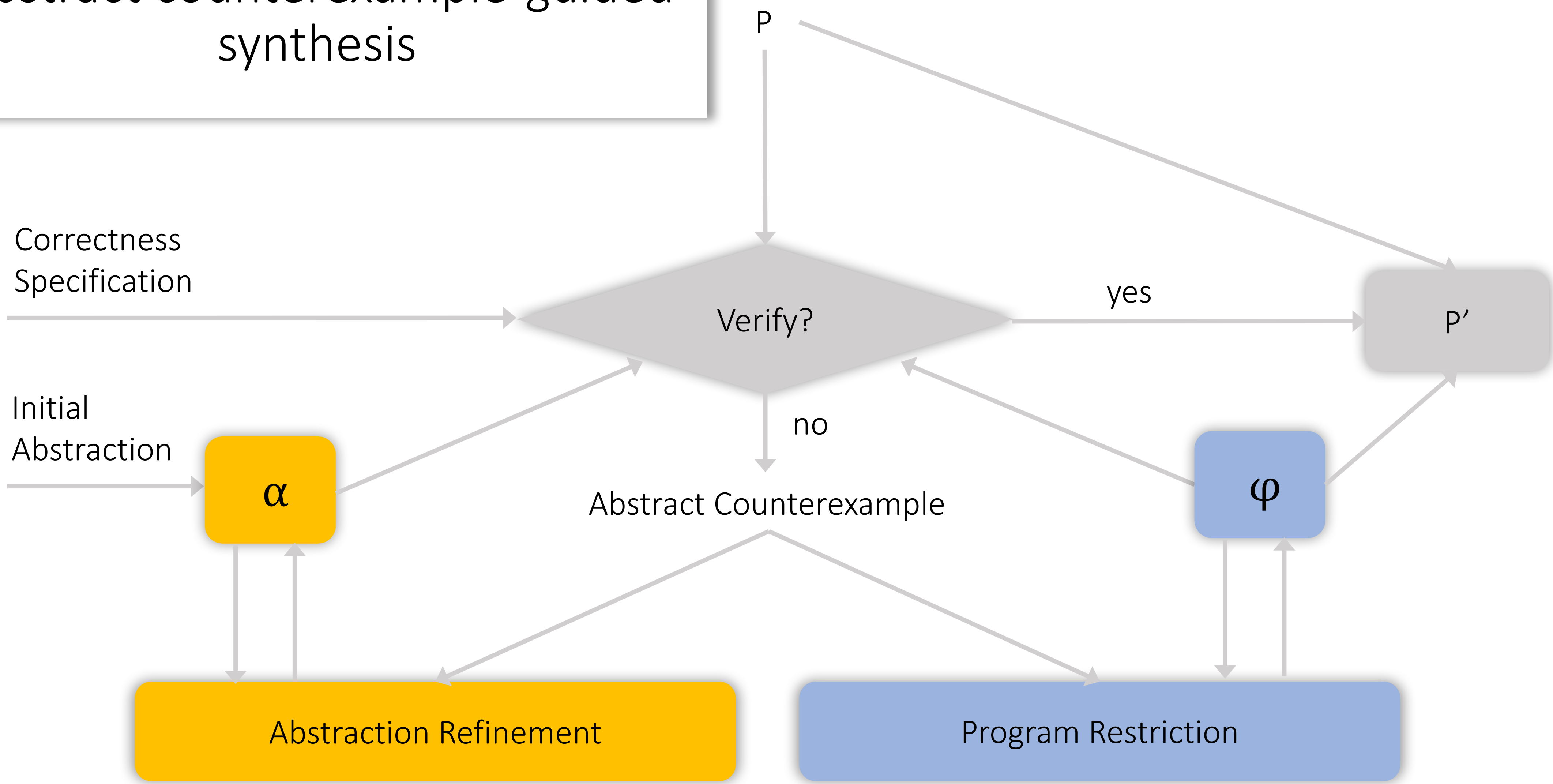
Counterexample-guided abstraction refinement



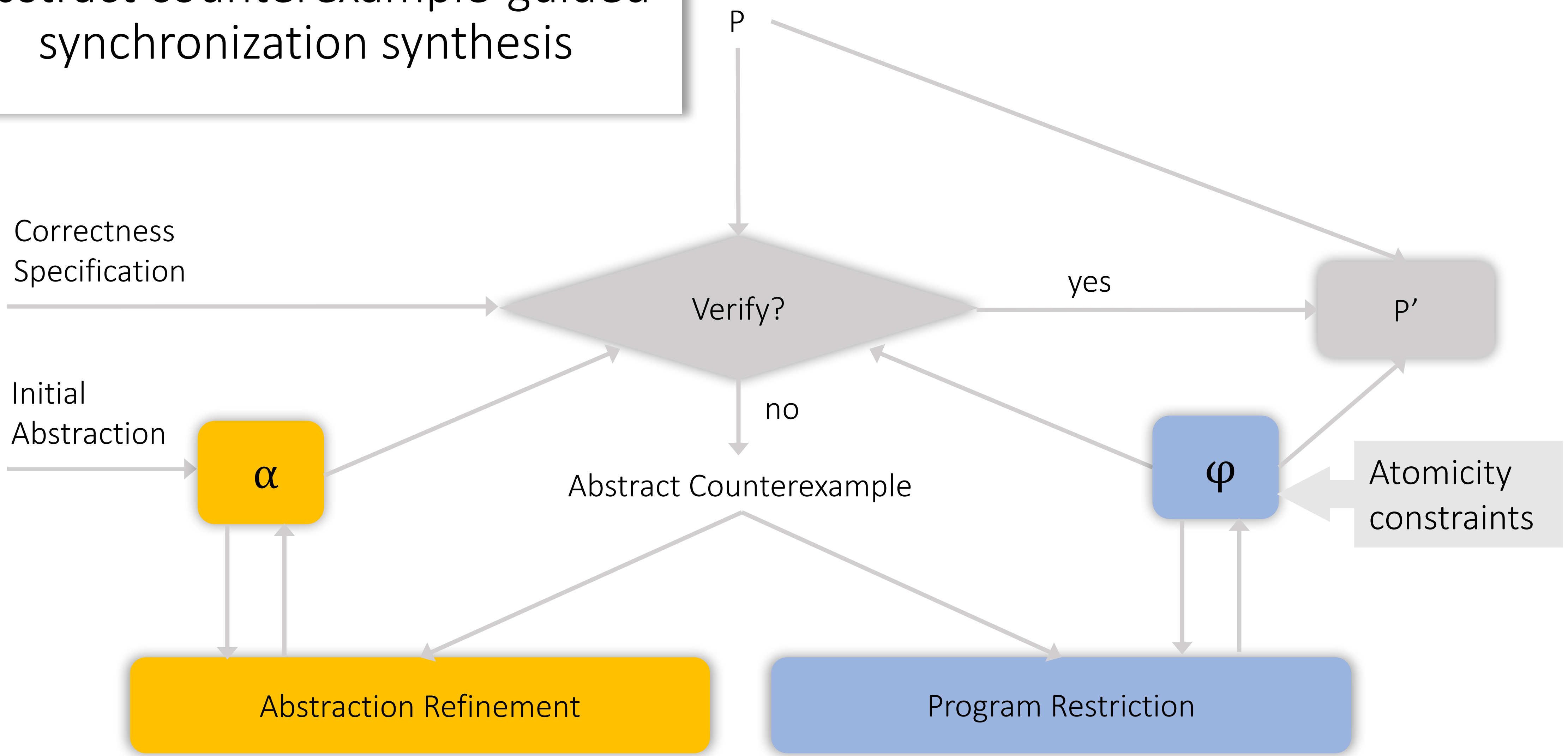
Counterexample-guided repair/synthesis

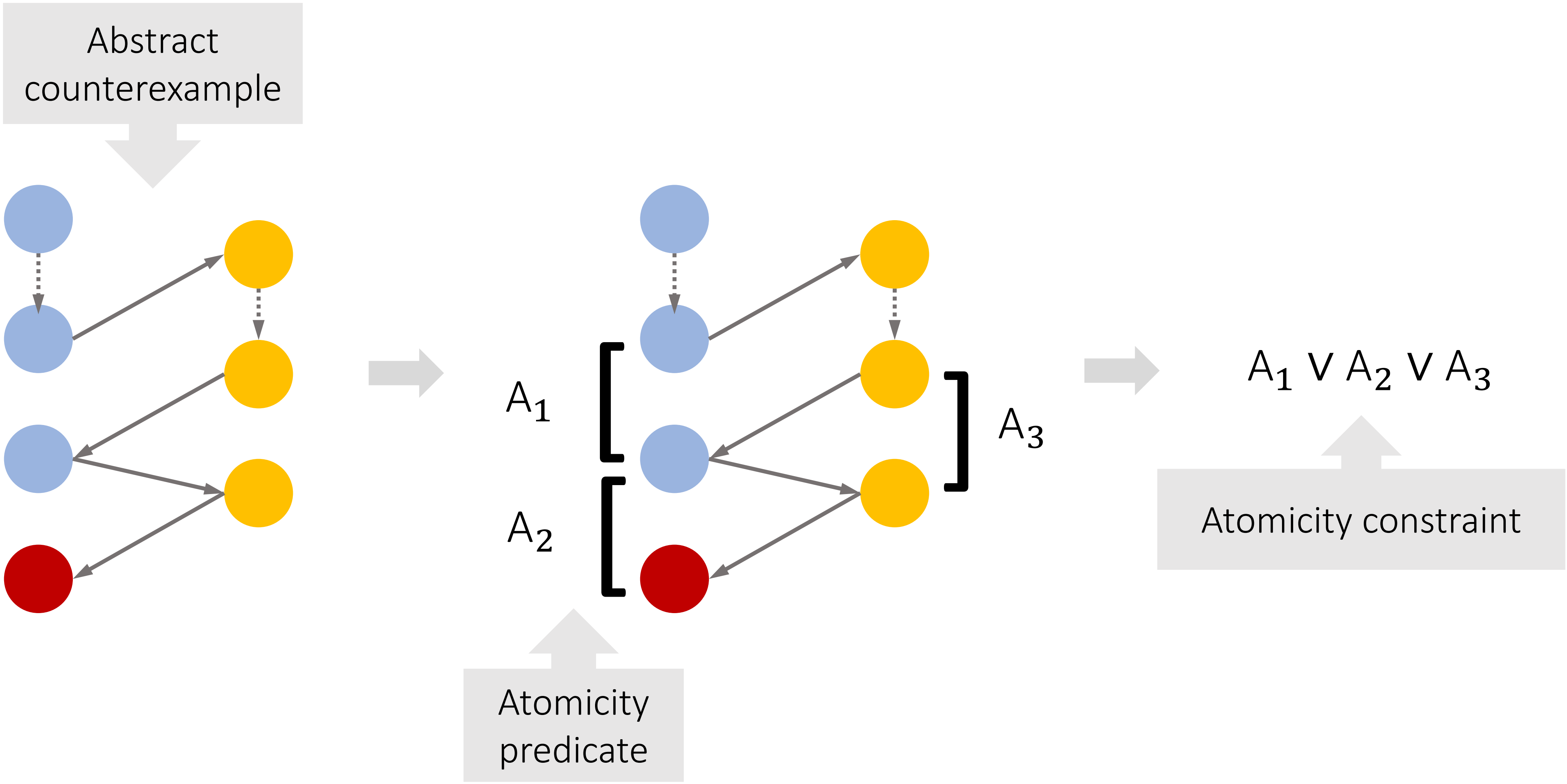


Abstract counterexample-guided synthesis



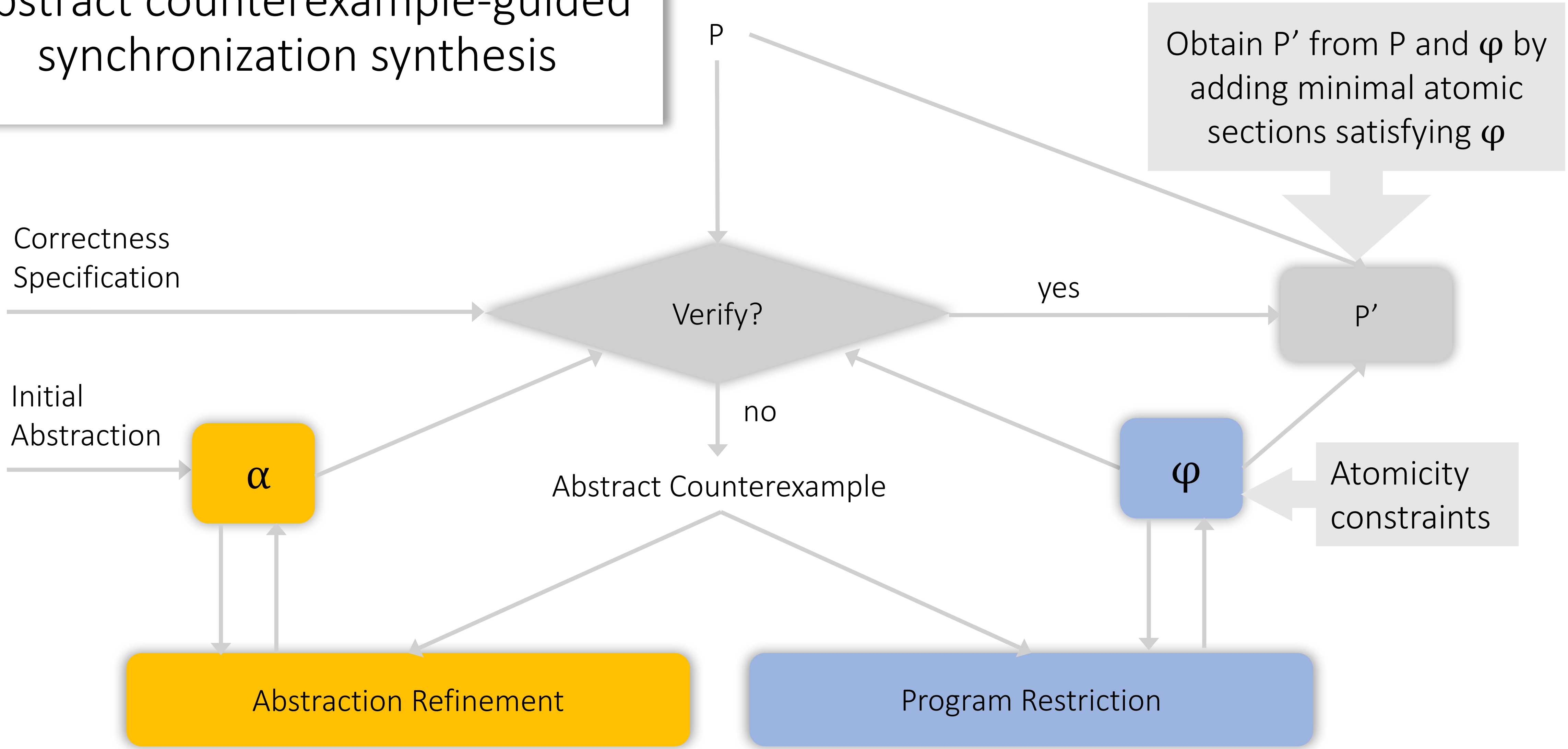
Abstract counterexample-guided synchronization synthesis

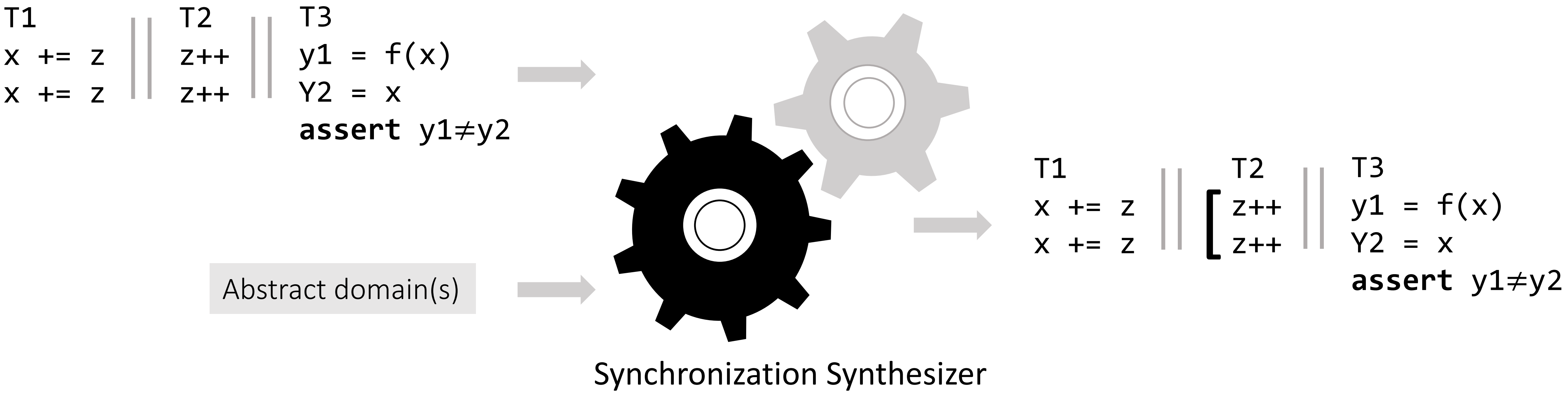




Abstract counterexample
A_1
A_2
A_3
$A_1 \vee A_2 \vee A_3$
Atomicity constraint
Atomicity predicate

Abstract counterexample-guided synchronization synthesis





T1
x += z
x += z
T2
z++
z++
T3
y1 = f(x)
Y2 = x
assert y1≠y2
Abstract domain(s)
Synchronization Synthesizer
T1
x += z
x += z
T2
z++
z++
T3
y1 = f(x)
Y2 = x
assert y1≠y2

- **Process:**
  Infinite-state program
- **Communication Model:**
  Shared-memory, interleaving-based
- **Specification:**
  Safety property
- **Synchronization:**
  Atomic section
- **Procedure:**
  Abstraction-refinement & counterexample-based

- Abstraction-guided synthesis
- Synthesis as repair
- Quantitative synthesis

Abstraction-Guided Synthesis of Synchronization

Martin Vechev
IBM Research

Eran Yahav
IBM Research

Greta Yorsh
IBM Research

Abstract

We present a novel framework for automatic inference of efficient synchronization in concurrent programs, a task known to be difficult and error-prone when done manually.

Our framework is based on abstract interpretation and can infer synchronization for infinite state programs. Given a program, a specification, and an abstraction, we infer synchronization that avoids all (abstract) interleavings that may violate the specification, but permits as many valid interleavings as possible.

Combined with abstraction refinement, our framework can be viewed as a new approach for verification where both the program and the abstraction can be modified on-the-fly during the verification process. The ability to modify the program, and not only the abstraction, allows us to remove program interleavings not only when they are known to be invalid, but also when they cannot be verified using the given abstraction.

We implemented a prototype of our approach using numerical abstractions and applied it to verify several interesting programs.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]
General Terms Algorithms, Verification
Keywords concurrency, synthesis, abstract interpretation

1. Introduction

We present *abstraction-guided synthesis*, a novel approach for synthesizing efficient synchronization in concurrent programs. Our approach turns the one dimensional problem of verification under abstraction, in which only the abstraction can be modified (typically via abstraction refinement), into a two-dimensional problem, in which *both the program and the abstraction can be modified* until the abstraction is precise enough to verify the program.

Based on abstract interpretation [10], our technique synthesizes a symbolic characterization of *safe schedules* for concurrent infinite-state programs. Safe schedules can be realized by modifying the program or the scheduler:

- Concurrent programming: by automatically inferring minimal atomic sections that prevent unsafe schedules, we assist the programmer in building correct and efficient concurrent software, a task known to be difficult and error-prone.
- Benevolent runtime: a scheduler that always keeps the program execution on a safe schedule makes the runtime system more reliable and adaptive to ever-changing environment and safety requirements, without the need to modify the program.

POPL'10, January 17–23, 2009, Madrid, Spain.

Given a program P, a specification S, and an abstraction function α, verification determines whether $P \models_\alpha S$, that is, whether P satisfies the specification S under the abstraction α. When the answer to this question is negative, it may be the case that the program violates the specification, or that the abstraction α is not precise enough to show that the program satisfies it.

When $P \not\models_\alpha S$, abstraction refinement approaches (e.g., [3, 8]) share the common goal of trying to find a finer abstraction α' such that $P \models_{\alpha'} S$. In this paper, we investigate a complementary approach, of finding a program P' such that $P' \models_\alpha S$ under the original abstraction α and P' admits a subset of the behaviors of P. Furthermore, we combine the two directions — refining the abstraction, and restricting program behaviors, to yield a novel abstraction-guided synthesis algorithm.

One of the main challenges in our approach is to devise an algorithm for obtaining such P' from the initial program P. In this paper, we focus on *concurrent programs*, and consider changes to P that correspond to restricting interleavings by adding synchronization.

Although it is possible to apply our techniques to other settings, concurrent programs are a natural fit. Concurrent programs are often correct on most interleavings and only miss synchronization in a few corner cases, which can be then avoided by synthesizing additional synchronization. Furthermore, in many cases, constraining the permitted interleavings reduces the set of reachable (abstract) states, possibly enabling verification via a coarser abstraction and avoiding state-space explosion.

The AGS algorithm, presented in Section 4, iteratively eliminates invalid interleavings until the abstraction is precise enough to verify the program. Some of the (abstract) invalid interleavings it observes may correspond to concrete invalid interleavings, while others may be artifacts of the abstraction. Whenever the algorithm observes an (abstract) invalid interleaving, the algorithm tries to eliminate it by either (i) modifying the program, or (ii) refining the abstraction.

To refine the abstraction, the algorithm can use any standard technique (e.g.,[3, 8]). These include moving through a predetermined series of domains with increasing precision (and typically increasing cost), or refining within the same abstract domain by changing its parameters (e.g., [4]).

To modify the program, we provide a novel algorithm that generates and solves *atomicity constraints*. Atomicity constraints define which statements have to be executed atomically, without an intermediate context switch, to eliminate the invalid interleavings. This corresponds to limiting the non-deterministic choices available to the scheduler. A solution of the atomicity constraints can be implemented by adding atomic sections to the program.

Our approach separates the process of identifying the space of solutions (generating the atomicity constraints) from the process of choosing between the possible solutions, which can be based on a quantitative criterion. As we discuss in Section 6, our approach provides a solution to a *quantitative synthesis* problem [5], as it

- Interleaving explosion
- Someone needs to write a specification
- Atomic sections are not very permissive

Abstraction-Guided Synthesis of Synchronization

Martin Vechev
IBM Research

Eran Yahav
IBM Research

Greta Yorsh
IBM Research

Abstract

We present a novel framework for automatic inference of efficient synchronization in concurrent programs, a task known to be difficult and error-prone when done manually.

Our framework is based on abstract interpretation and can infer synchronization for infinite state programs. Given a program, a specification, and an abstraction, we infer synchronization that avoids all (abstract) interleavings that may violate the specification, but permits as many valid interleavings as possible.

Combined with abstraction refinement, our framework can be viewed as a new approach for verification where both the program and the abstraction can be modified on-the-fly during the verification process. The ability to modify the program, and not only the abstraction, allows us to remove program interleavings not only when they are known to be invalid, but also when they cannot be verified using the given abstraction.

We implemented a prototype of our approach using numerical abstractions and applied it to verify several interesting programs.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]
General Terms Algorithms, Verification
Keywords concurrency, synthesis, abstract interpretation

1. Introduction

We present *abstraction-guided synthesis*, a novel approach for synthesizing efficient synchronization in concurrent programs. Our approach turns the one dimensional problem of verification under abstraction, in which only the abstraction can be modified (typically via abstraction refinement), into a two-dimensional problem, in which *both the program and the abstraction can be modified* until the abstraction is precise enough to verify the program.

Based on abstract interpretation [10], our technique synthesizes a symbolic characterization of *safe schedules* for concurrent infinite-state programs. Safe schedules can be realized by modifying the program or the scheduler:

- Concurrent programming: by automatically inferring minimal atomic sections that prevent unsafe schedules, we assist the programmer in building correct and efficient concurrent software, a task known to be difficult and error-prone.
- Benevolent runtime: a scheduler that always keeps the program execution on a safe schedule makes the runtime system more reliable and adaptive to ever-changing environment and safety requirements, without the need to modify the program.

POPL'10, January 17–23, 2009, Madrid, Spain.

Given a program P, a specification S, and an abstraction function α, verification determines whether $P \models_\alpha S$, that is, whether P satisfies the specification S under the abstraction α. When the answer to this question is negative, it may be the case that the program violates the specification, or that the abstraction α is not precise enough to show that the program satisfies it.

When $P \not\models_\alpha S$, abstraction refinement approaches (e.g., [3, 8]) share the common goal of trying to find a finer abstraction α' such that $P \models_{\alpha'} S$. In this paper, we investigate a complementary approach, of finding a program P' such that $P' \models_\alpha S$ under the original abstraction α and P' admits a subset of the behaviors of P. Furthermore, we combine the two directions — refining the abstraction, and restricting program behaviors, to yield a novel abstraction-guided synthesis algorithm.

One of the main challenges in our approach is to devise an algorithm for obtaining such P' from the initial program P. In this paper, we focus on *concurrent programs*, and consider changes to P that correspond to restricting interleavings by adding synchronization.

Although it is possible to apply our techniques to other settings, concurrent programs are a natural fit. Concurrent programs are often correct on most interleavings and only miss synchronization in a few corner cases, which can be then avoided by synthesizing additional synchronization. Furthermore, in many cases, constraining the permitted interleavings reduces the set of reachable (abstract) states, possibly enabling verification via a coarser abstraction and avoiding state-space explosion.

The AGS algorithm, presented in Section 4, iteratively eliminates invalid interleavings until the abstraction is precise enough to verify the program. Some of the (abstract) invalid interleavings it observes may correspond to concrete invalid interleavings, while others may be artifacts of the abstraction. Whenever the algorithm observes an (abstract) invalid interleaving, the algorithm tries to eliminate it by either (i) modifying the program, or (ii) refining the abstraction.

To refine the abstraction, the algorithm can use any standard technique (e.g.,[3, 8]). These include moving through a predetermined series of domains with increasing precision (and typically increasing cost), or refining within the same abstract domain by changing its parameters (e.g., [4]).

To modify the program, we provide a novel algorithm that generates and solves *atomicity constraints*. Atomicity constraints define which statements have to be executed atomically, without an intermediate context switch, to eliminate the invalid interleavings. This corresponds to limiting the non-deterministic choices available to the scheduler. A solution of the atomicity constraints can be implemented by adding atomic sections to the program.

Our approach separates the process of identifying the space of solutions (generating the atomicity constraints) from the process of choosing between the possible solutions, which can be based on a quantitative criterion. As we discuss in Section 6, our approach provides a solution to a *quantitative synthesis* problem [5], as it

A seminal paper

A cool paper

A modern approach

A modern approach

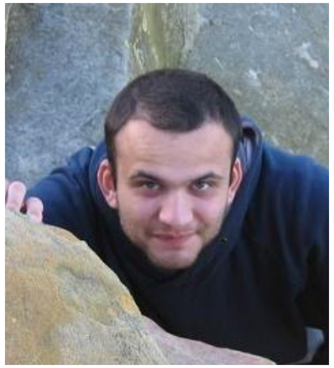
Černý

Clarke

Henzinger

Radhakrishna

Ryzhyk



Samanta

Tarrach

From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis. CAV 2016.

From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis *

Pavol Černý[1], Edmund M. Clarke[2], Thomas A. Henzinger[3], Arjun Radhakrishna[4], Leonid Ryzhyk[2], Roopsha Samanta[3], and Thorsten Tarrach[3]

[1] University of Colorado Boulder
[2] Carnegie Mellon University
[3] IST Austria
[4] University of Pennsylvania

Abstract. We present a computer-aided programming approach to concurrency. The approach allows programmers to program assuming a friendly, non-preemptive scheduler, and our synthesis procedure inserts synchronization to ensure that the final program works even with a preemptive scheduler. The correctness specification is implicit, inferred from the non-preemptive behavior. Let us consider sequences of calls that the program makes to an external interface. The specification requires that any such sequence produced under a preemptive scheduler should be included in the set of such sequences produced under a non-preemptive scheduler. The solution is based on a finitary abstraction, an algorithm for bounded language inclusion modulo an independence relation, and rules for inserting synchronization. We apply the approach to device-driver programming, where the driver threads call the software interface of the device and the API provided by the operating system. Our experiments demonstrate that our synthesis method is precise and efficient, and, since it does not require explicit specifications, is more practical than the conventional approach based on user-provided assertions.

1 Introduction

Concurrent shared-memory programming is notoriously difficult and error-prone. Program synthesis for concurrency aims to mitigate this complexity by synthesizing synchronization code automatically [4, 5, 8, 11]. However, specifying the programmer's intent may be a challenge in itself. Declarative mechanisms, such as assertions, suffer from the drawback that it is difficult to ensure that the specification is complete and fully captures the programmer's intent.

We propose a solution where the specification is *implicit*. We observe that a core difficulty in concurrent programming originates from the fact that the scheduler can *preempt* the execution of a thread at any time. We therefore give

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), by NSF under award CCF 1421752 and the Expeditions award CCF 1138996, by the Simons Foundation, and by a gift from the Intel Corporation.

Succinct Representation of Concurrent Trace Sets *

Ashutosh Gupta
IST Austria
agupta@ist.ac.at

Thomas A. Henzinger
IST Austria
tah@ist.ac.at

Arjun Radhakrishna
IST Austria, University of Pennsylvania
arjunrad@seas.upenn.edu

Roopsha Samanta
IST Austria
rsamanta@ist.ac.at

Thorsten Tarrach
IST Austria
ttarrach@ist.ac.at

Abstract

We present a method and a tool for generating succinct representations of sets of concurrent traces. We focus on trace sets that contain all correct or all incorrect permutations of events from a given trace. We represent trace sets as *HB-formulas* that are Boolean combinations of *happens-before* constraints between events. To generate a representation of incorrect interleavings, our method iteratively explores interleavings that violate the specification and gathers generalizations of the discovered interleavings into an HB-formula; its complement yields a representation of correct interleavings.

We claim that our trace set representations can drive diverse verification, fault localization, repair, and synthesis techniques for concurrent programs. We demonstrate this by using our tool in three case studies involving synchronization synthesis, bug summarization, and abstraction refinement based verification. In each case study, our initial experimental results have been promising.

In the first case study, we present an algorithm for inferring missing synchronization from an HB-formula representing correct interleavings of a given trace. The algorithm applies rules to rewrite specific patterns in the HB-formula into locks, barriers, and wait-notify constructs. In the second case study, we use an HB-formula representing incorrect interleavings for bug summarization. While the HB-formula itself is a concise counterexample summary, we present additional inference rules to help identify specific concurrency bugs such as data races, define-use order violations, and two-stage access bugs. In the final case study, we present a novel predicate learning procedure that uses HB-formulas representing abstract counterexamples to accelerate counterexample-guided abstraction refinement (CEGAR). In each iteration of the CEGAR loop, the procedure refines the abstraction to eliminate multiple spurious abstract counterexamples drawn from the HB-formula.

Categories and Subject Descriptors D [2]: 4—Formal methods

Keywords Trace Generalization; Concurrent Programs; Synchronization Synthesis; Bug Summarization; CEGAR

POPL '15, January 15–17, 2015, Mumbai, India.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3300-9/15/01...$15.00.
http://dx.doi.org/10.1145/2676726.2677008

1. Introduction

Sets of concurrent traces containing permutations of events from a given concurrent trace are useful for predictive analysis (e.g., [24, 34, 35, 41]) and synchronization synthesis (e.g., [8, 9]) of shared-memory concurrent programs. Most approaches using such trace sets are restricted to specific aspects of reasoning about concurrent programs such as data race detection [24, 34], detection of safety violations [35, 41] and fixing assertion failures [8, 9]. Moreover, the representations of trace sets and exploration strategies used in some of these approaches [8, 9, 35]) *underapproximate* the target trace sets. In this paper, we present a succinct, *complete* representation of such concurrent trace sets, which can drive diverse verification, fault localization, repair, and synthesis techniques for concurrent programs. The representation is complete in the sense that it encodes every trace in the trace set of interest.

Concurrent trace sets. First, we fix some terminology. An *execution* π of a concurrent program $\mathcal{P}$ is an alternating sequence of variable valuations and events corresponding to a feasible interleaving of instructions from the threads of $\mathcal{P}$. An execution is *good* if it satisfies a given specification, and *bad* otherwise. A *trace* is a sequence of events corresponding to an interleaving of instructions from the threads of $\mathcal{P}$. The trace of an execution π is the sequence of events within π. The language $\mathcal{L}(\tau)$ of a trace τ is the set of all executions with trace τ. A trace τ is feasible if $\mathcal{L}(\tau)$ is non-empty, and infeasible otherwise. A feasible trace τ is good if all executions in $\mathcal{L}(\tau)$ are good, and bad otherwise.

We group traces into *neighbourhoods*. The neighbourhood $\mathcal{N}_\tau$ of a trace τ contains all permutations of τ that preserve τ's intra-thread event order. The *good neighbourhood* $\mathcal{N}_\tau^g$ of a trace τ is the set containing all the good traces in $\mathcal{N}_\tau$. The *bad neighbourhood* $\mathcal{N}_\tau^b$ of a trace τ is a set containing all the bad traces in $\mathcal{N}_\tau$. The languages $\mathcal{L}(\mathcal{N}_\tau)$, $\mathcal{L}(\mathcal{N}_\tau^g)$ and $\mathcal{L}(\mathcal{N}_\tau^b)$ are the unions of the languages of all traces in $\mathcal{N}_\tau$, $\mathcal{N}_\tau^g$ and $\mathcal{N}_\tau^b$, respectively.

Representation of concurrent trace sets. There are multiple ways to represent trace sets. Some representations may be more expressive or useful for reasoning about concurrent programs than others. A candidate representation that has been used for certain trace sets is a partial order over events [8, 9, 41]. The neighbourhood of a trace, as defined above, can also be represented as a partial order. However, the good neighbourhood or the bad neighbourhood of a trace is, in general, not a partial order. For instance, for the

*This research was supported in part by the European Research Council (ERC) under grant agreement 267989 (QUAREM), by the Austrian Science Fund (FWF) NFN project S11402-N23 (RiSE), and the NSF Expeditions award CCF 1138996.

POPL 2016

FMSD 2017

Form Methods Syst Des (2017) 50:97–139
DOI 10.1007/s10703-016-0256-5

CrossMark

From non-preemptive to preemptive scheduling using synchronization synthesis

Pavol Černý[1] · Edmund M. Clarke[2] · Thomas A. Henzinger[3] · Arjun Radhakrishna[4] · Leonid Ryzhyk[5] · Roopsha Samanta[6] · Thorsten Tarrach[3]

Published online: 27 September 2016

Abstract We present a computer-aided programming approach to concurrency. The approach allows programmers to program assuming a friendly, non-preemptive scheduler, and our synthesis procedure inserts synchronization to ensure that the final program works even with a preemptive scheduler. The correctness specification is implicit, inferred from the non-preemptive behavior. Let us consider sequences of calls that the program makes to an external interface. The specification requires that any such sequence produced under

This work was published, in part, in Computer Aided Verification (CAV) 2015 [4].

✉ Thorsten Tarrach
ttarrach@ist.ac.at

Pavol Černý
pavol.cerny@colorado.edu

Edmund M. Clarke
emc@cs.cmu.edu

Thomas A. Henzinger
tah@ist.ac.at

Arjun Radhakrishna
arjunrad@cis.upenn.edu

Leonid Ryzhyk
l.ryzhyk@samsung.com

Roopsha Samanta
roopsha@cs.purdue.edu

1 University of Colorado Boulder, 425 UCB, Boulder, CO 80309, USA
2 Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
3 IST Austria, Am Campus 1, 3400 Klosterneuburg, Austria
4 University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104, USA
5 Samsung Research America, 665 Clyde Avenue, Mountain View, CA 94043, USA
6 University of Purdue, 610 Purdue Mall, West Lafayette, IN 47907, USA

Springer

Trace generalization-based framework to infer synchronization for an implicit specification of infinite-state concurrent programs

From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis *

Pavol Černý[1], Edmund M. Clarke[2], Thomas A. Henzinger[3], Arjun Radhakrishna[4], Leonid Ryzhyk[2], Roopsha Samanta[3], and Thorsten Tarrach[3]

[1] University of Colorado Boulder
[2] Carnegie Mellon University
[3] IST Austria
[4] University of Pennsylvania

Abstract. We present a computer-aided programming approach to concurrency. The approach allows programmers to program assuming a friendly, non-preemptive scheduler, and our synthesis procedure inserts synchronization to ensure that the final program works even with a preemptive scheduler. The correctness specification is implicit, inferred from the non-preemptive behavior. Let us consider sequences of calls that the program makes to an external interface. The specification requires that any such sequence produced under a preemptive scheduler should be included in the set of such sequences produced under a non-preemptive scheduler. The solution is based on a finitary abstraction, an algorithm for bounded language inclusion modulo an independence relation, and rules for inserting synchronization. We apply the approach to device-driver programming, where the driver threads call the software interface of the device and the API provided by the operating system. Our experiments demonstrate that our synthesis method is precise and efficient, and, since it does not require explicit specifications, is more practical than the conventional approach based on user-provided assertions.

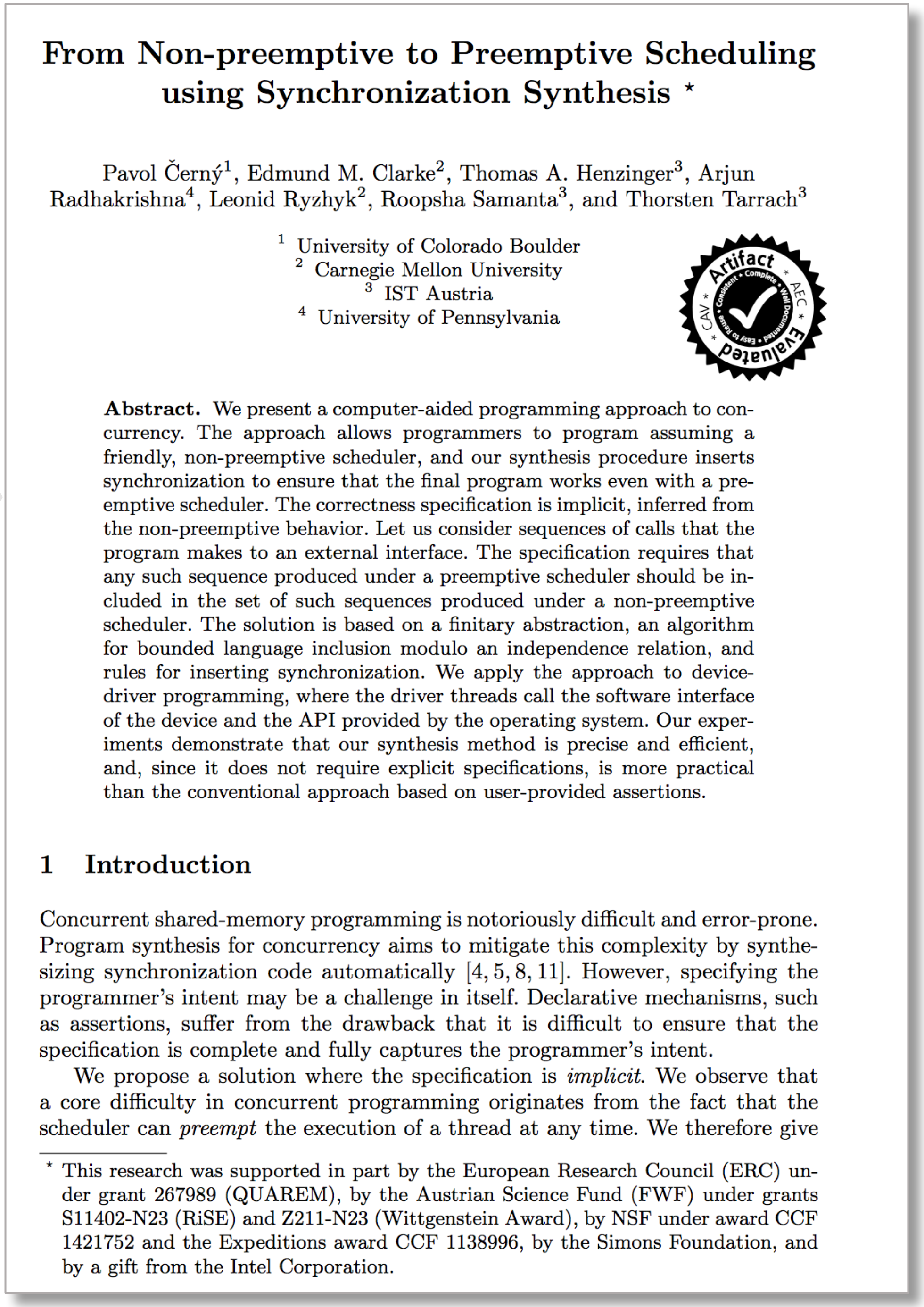
1 Introduction

Concurrent shared-memory programming is notoriously difficult and error-prone. Program synthesis for concurrency aims to mitigate this complexity by synthesizing synchronization code automatically [4, 5, 8, 11]. However, specifying the programmer's intent may be a challenge in itself. Declarative mechanisms, such as assertions, suffer from the drawback that it is difficult to ensure that the specification is complete and fully captures the programmer's intent.

We propose a solution where the specification is *implicit*. We observe that a core difficulty in concurrent programming originates from the fact that the scheduler can *preempt* the execution of a thread at any time. We therefore give

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), by NSF under award CCF 1421752 and the Expeditions award CCF 1138996, by the Simons Foundation, and by a gift from the Intel Corporation.

- **Process:**
  Infinite-state program
- **Communication Model:**
  Shared-memory, interleaving-based
- **Specification:**
  Implicit (behavior under non-preemptive scheduler), safety property
- **Synchronization:**
  Locks, wait-notify etc.
- **Procedure:**
  Counterexample generalization



From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis *

Pavol Černý[1], Edmund M. Clarke[2], Thomas A. Henzinger[3], Arjun Radhakrishna[4], Leonid Ryzhyk[2], Roopsha Samanta[3], and Thorsten Tarrach[3]

[1] University of Colorado Boulder
[2] Carnegie Mellon University
[3] IST Austria
[4] University of Pennsylvania

Abstract. We present a computer-aided programming approach to concurrency. The approach allows programmers to program assuming a friendly, non-preemptive scheduler, and our synthesis procedure inserts synchronization to ensure that the final program works even with a preemptive scheduler. The correctness specification is implicit, inferred from the non-preemptive behavior. Let us consider sequences of calls that the program makes to an external interface. The specification requires that any such sequence produced under a preemptive scheduler should be included in the set of such sequences produced under a non-preemptive scheduler. The solution is based on a finitary abstraction, an algorithm for bounded language inclusion modulo an independence relation, and rules for inserting synchronization. We apply the approach to device-driver programming, where the driver threads call the software interface of the device and the API provided by the operating system. Our experiments demonstrate that our synthesis method is precise and efficient, and, since it does not require explicit specifications, is more practical than the conventional approach based on user-provided assertions.

1 Introduction

Concurrent shared-memory programming is notoriously difficult and error-prone. Program synthesis for concurrency aims to mitigate this complexity by synthesizing synchronization code automatically [4, 5, 8, 11]. However, specifying the programmer's intent may be a challenge in itself. Declarative mechanisms, such as assertions, suffer from the drawback that it is difficult to ensure that the specification is complete and fully captures the programmer's intent.

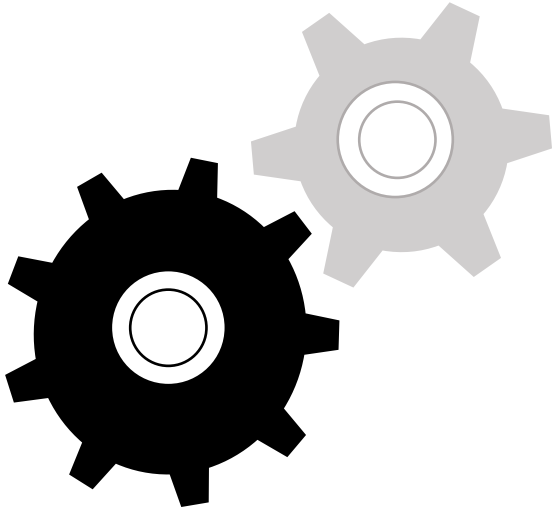
We propose a solution where the specification is *implicit*. We observe that a core difficulty in concurrent programming originates from the fact that the scheduler can *preempt* the execution of a thread at any time. We therefore give

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), by NSF under award CCF 1421752 and the Expeditions award CCF 1138996, by the Simons Foundation, and by a gift from the Intel Corporation.

P:

```
void open_dev()
  if (open==0)
     power_up();
  open := open+1;
  yield;

void close_dev()
  if (open>0)
     open := open-1;
     if (open==0)
        power_down();
  yield;
```



Synchronization Synthesizer

P':

```
void open_dev()
 lock(l)
  if (open==0)
     power_up();
  open := open+1;
 unlock(l)
  yield;

void close_dev()
 lock(l)
  if (open>0)
     open := open-1;
     if (open==0)
        power_down();
 unlock(l)
  yield;
```

$$[\![P']\!]^{preempt} \subseteq [\![P]\!]^{nonpreempt}$$

Preemption-safety

Assumption: Programmer ensures P is correct for a non-preemptive scheduler

- **Process:**
  Infinite-state program
- **Communication Model:**
  Shared-memory, , interleaving-based
- **Specification:**
  Implicit (behavior under non-preemptive scheduler), safety property
- **Synchronization:**
  Locks, wait-notify etc.
- **Procedure:**
  Counterexample generalization

- Counterexample generalization
- Specification-free synthesis
- Language inclusion verification procedure

From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis *

Pavol Černý[1], Edmund M. Clarke[2], Thomas A. Henzinger[3], Arjun Radhakrishna[4], Leonid Ryzhyk[2], Roopsha Samanta[3], and Thorsten Tarrach[3]

[1] University of Colorado Boulder
[2] Carnegie Mellon University
[3] IST Austria
[4] University of Pennsylvania

Abstract. We present a computer-aided programming approach to concurrency. The approach allows programmers to program assuming a friendly, non-preemptive scheduler, and our synthesis procedure inserts synchronization to ensure that the final program works even with a preemptive scheduler. The correctness specification is implicit, inferred from the non-preemptive behavior. Let us consider sequences of calls that the program makes to an external interface. The specification requires that any such sequence produced under a preemptive scheduler should be included in the set of such sequences produced under a non-preemptive scheduler. The solution is based on a finitary abstraction, an algorithm for bounded language inclusion modulo an independence relation, and rules for inserting synchronization. We apply the approach to device-driver programming, where the driver threads call the software interface of the device and the API provided by the operating system. Our experiments demonstrate that our synthesis method is precise and efficient, and, since it does not require explicit specifications, is more practical than the conventional approach based on user-provided assertions.

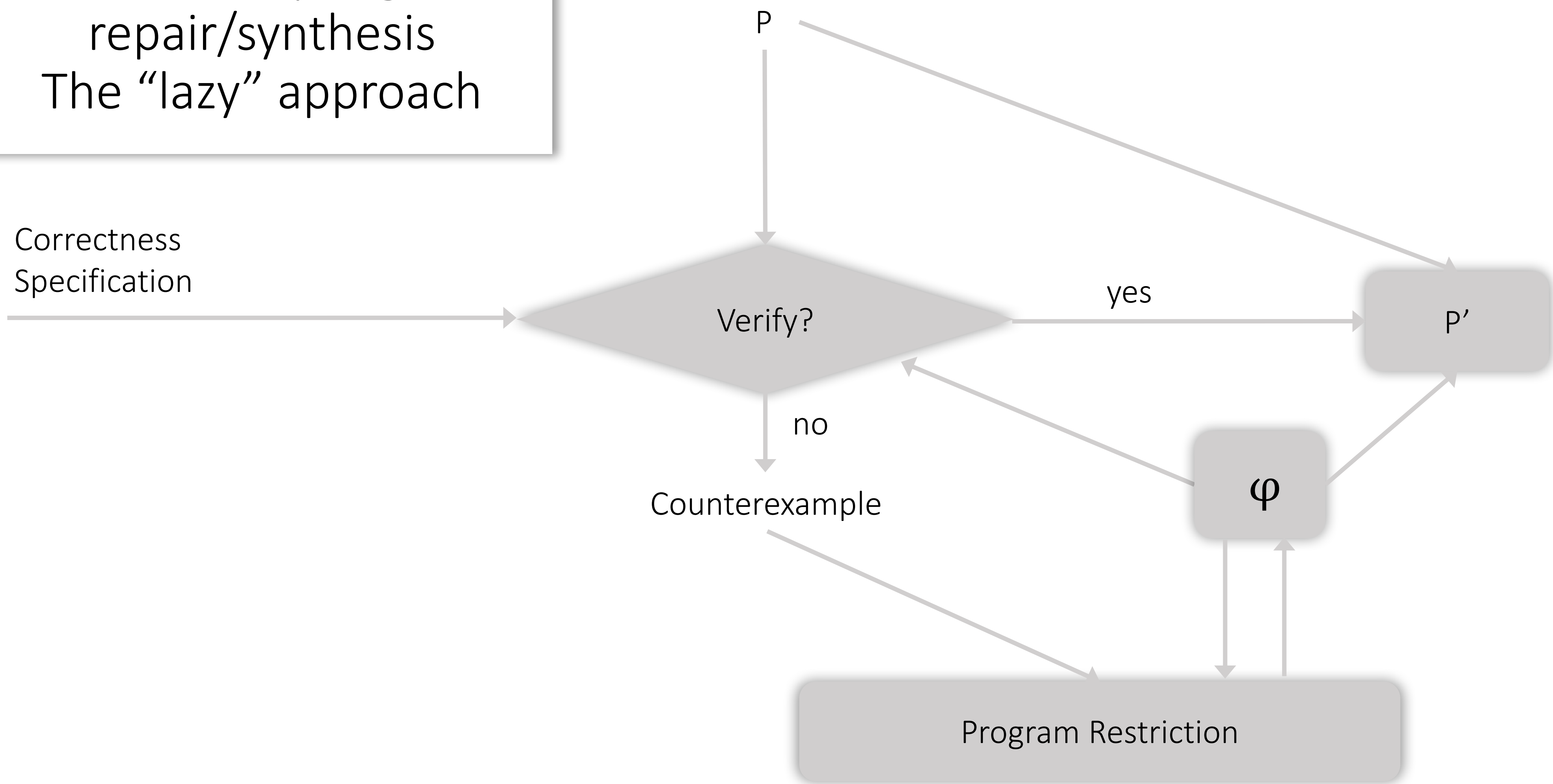
1 Introduction

Concurrent shared-memory programming is notoriously difficult and error-prone. Program synthesis for concurrency aims to mitigate this complexity by synthesizing synchronization code automatically [4, 5, 8, 11]. However, specifying the programmer's intent may be a challenge in itself. Declarative mechanisms, such as assertions, suffer from the drawback that it is difficult to ensure that the specification is complete and fully captures the programmer's intent.

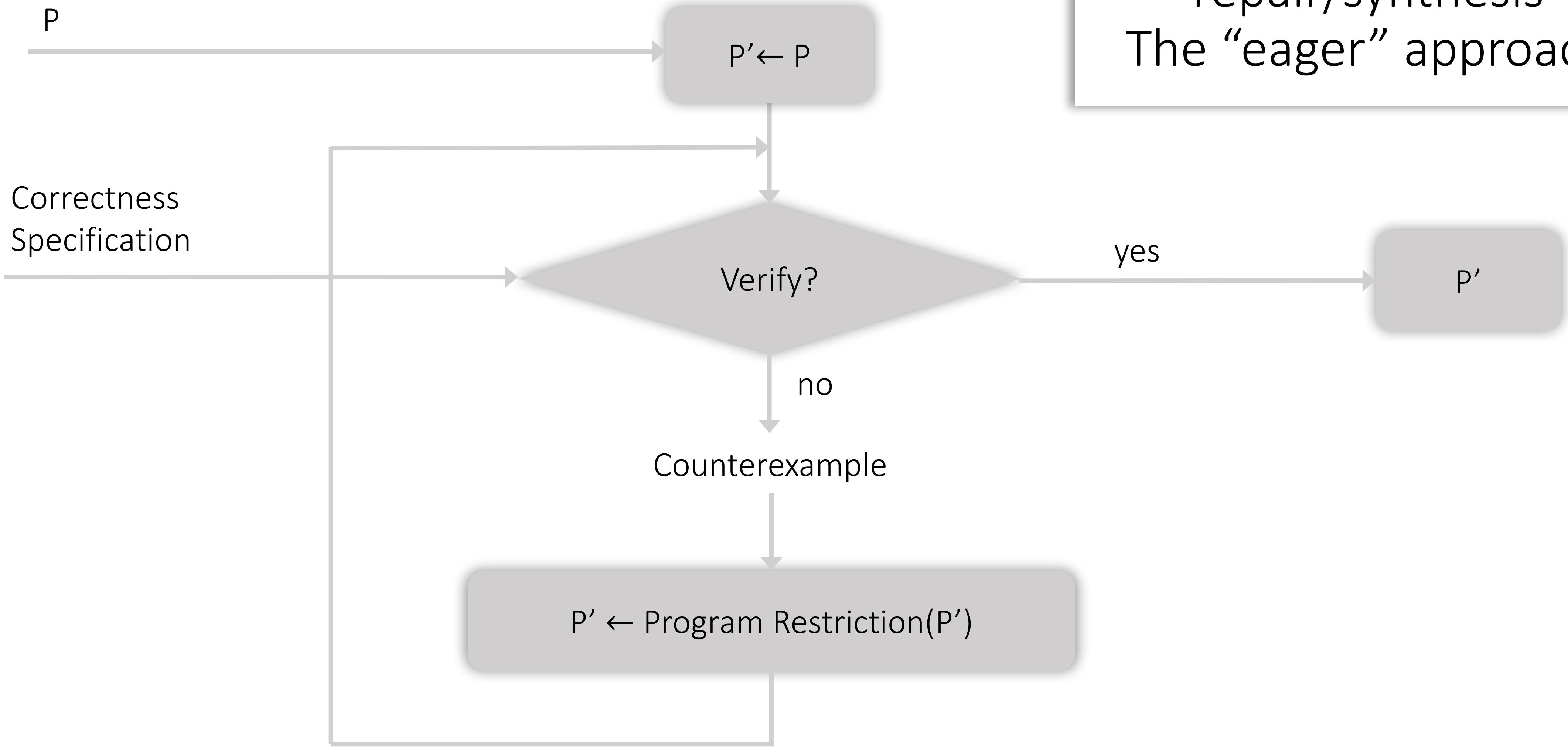
We propose a solution where the specification is *implicit*. We observe that a core difficulty in concurrent programming originates from the fact that the scheduler can *preempt* the execution of a thread at any time. We therefore give

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), by NSF under award CCF 1421752 and the Expeditions award CCF 1138996, by the Simons Foundation, and by a gift from the Intel Corporation.

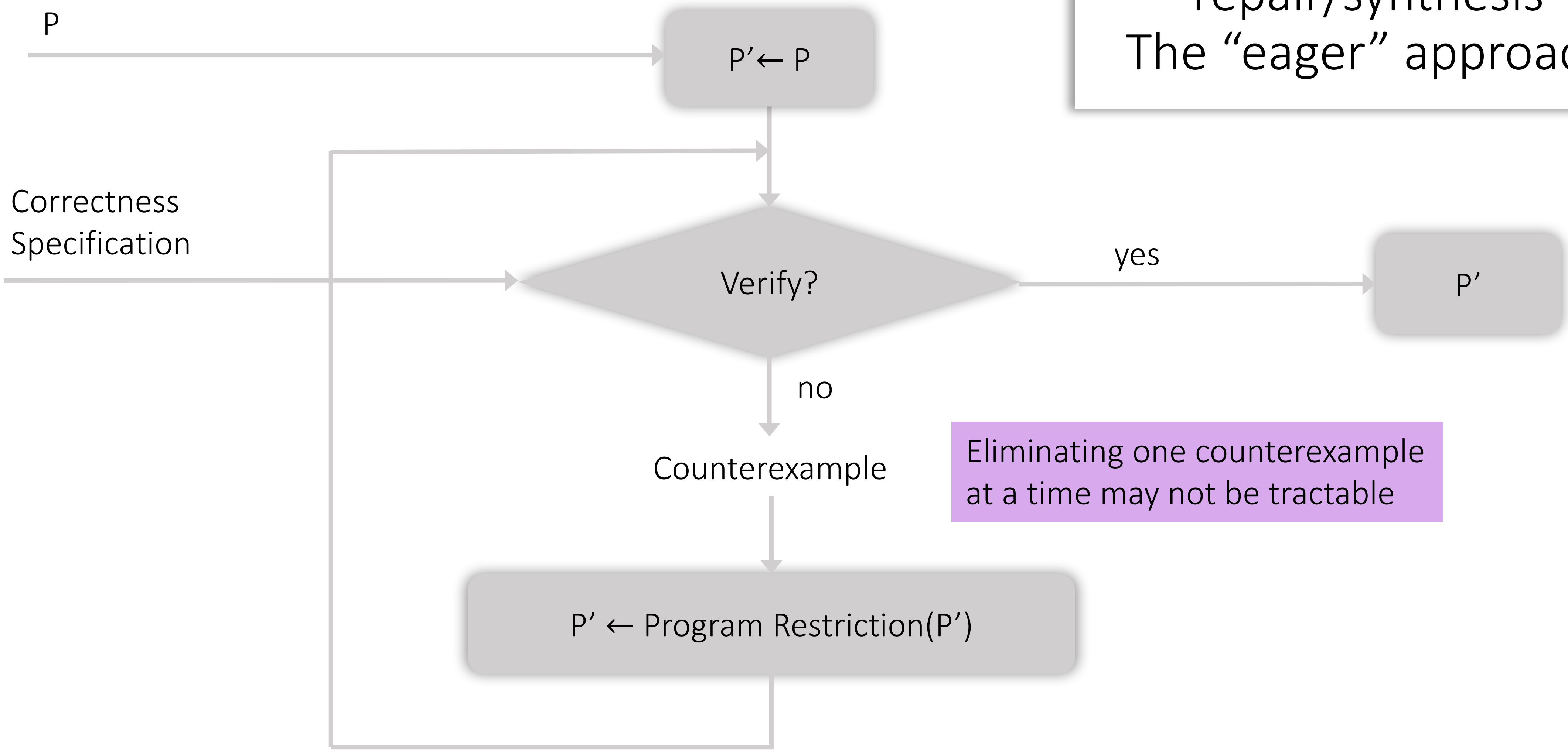
Counterexample-guided repair/synthesis
The “lazy” approach



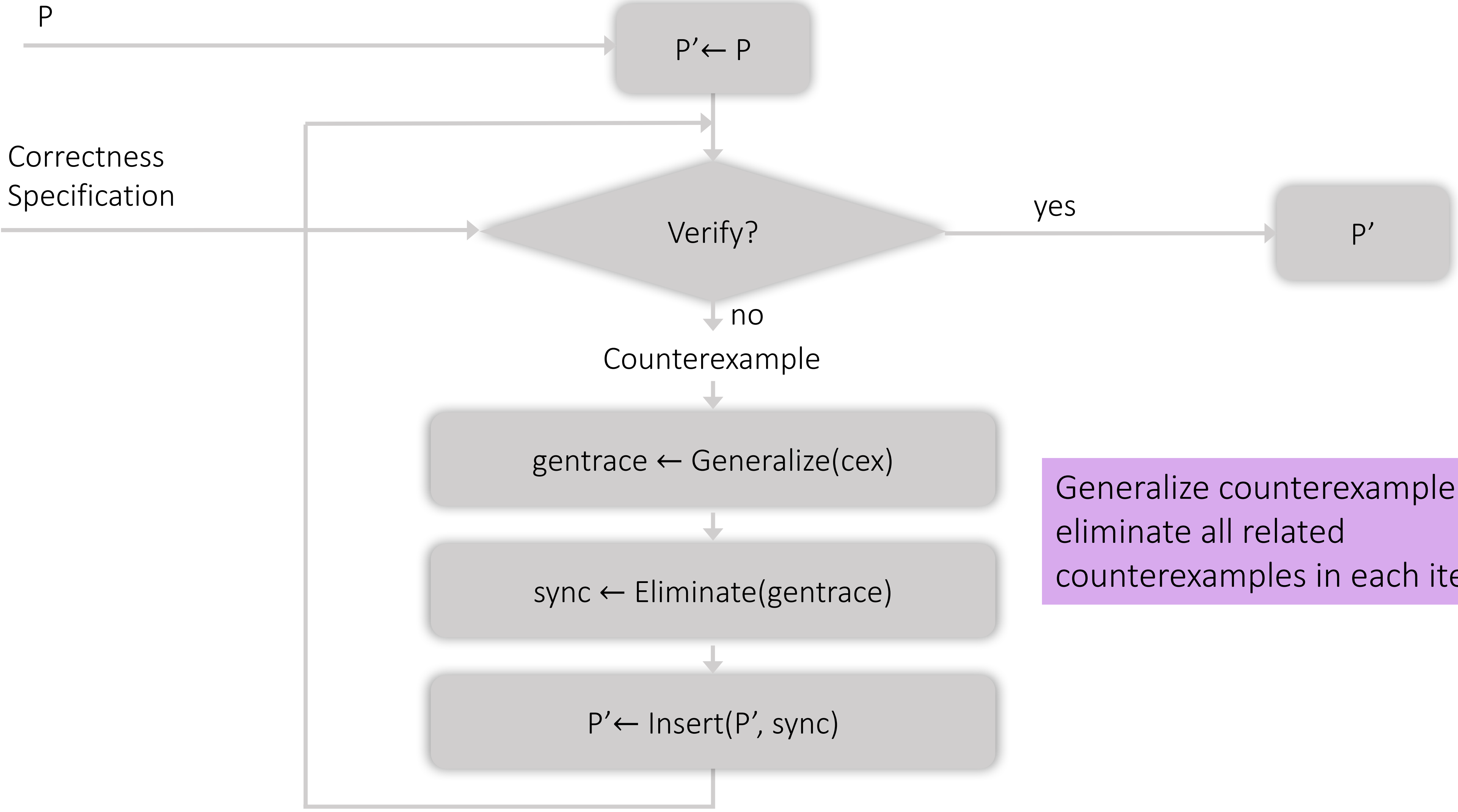
Counterexample-guided repair/synthesis
The "eager" approach



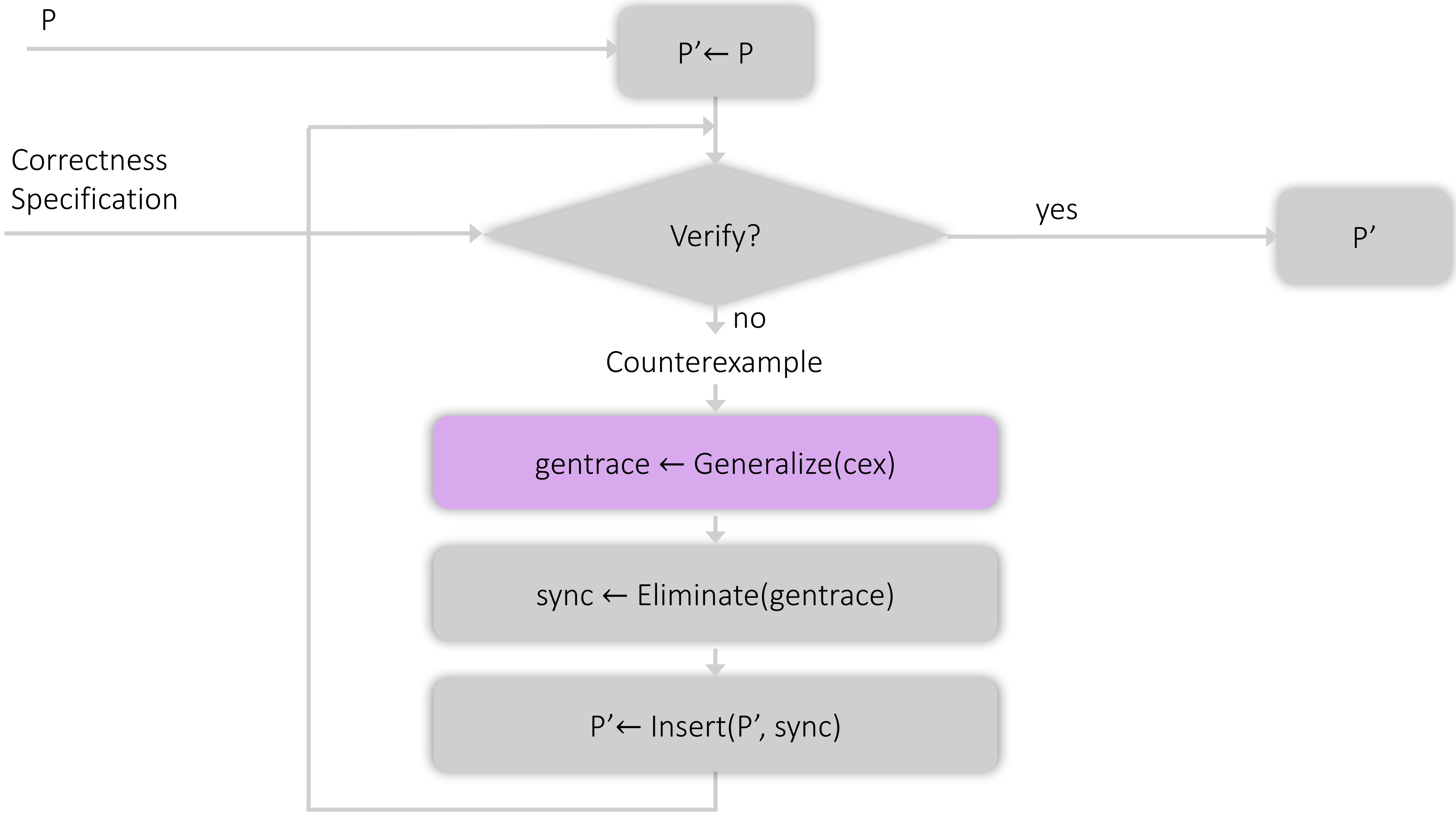
Counterexample-guided repair/synthesis
The "eager" approach



Eliminating one counterexample at a time may not be tractable



Generalize counterexample & eliminate all related counterexamples in each iteration!

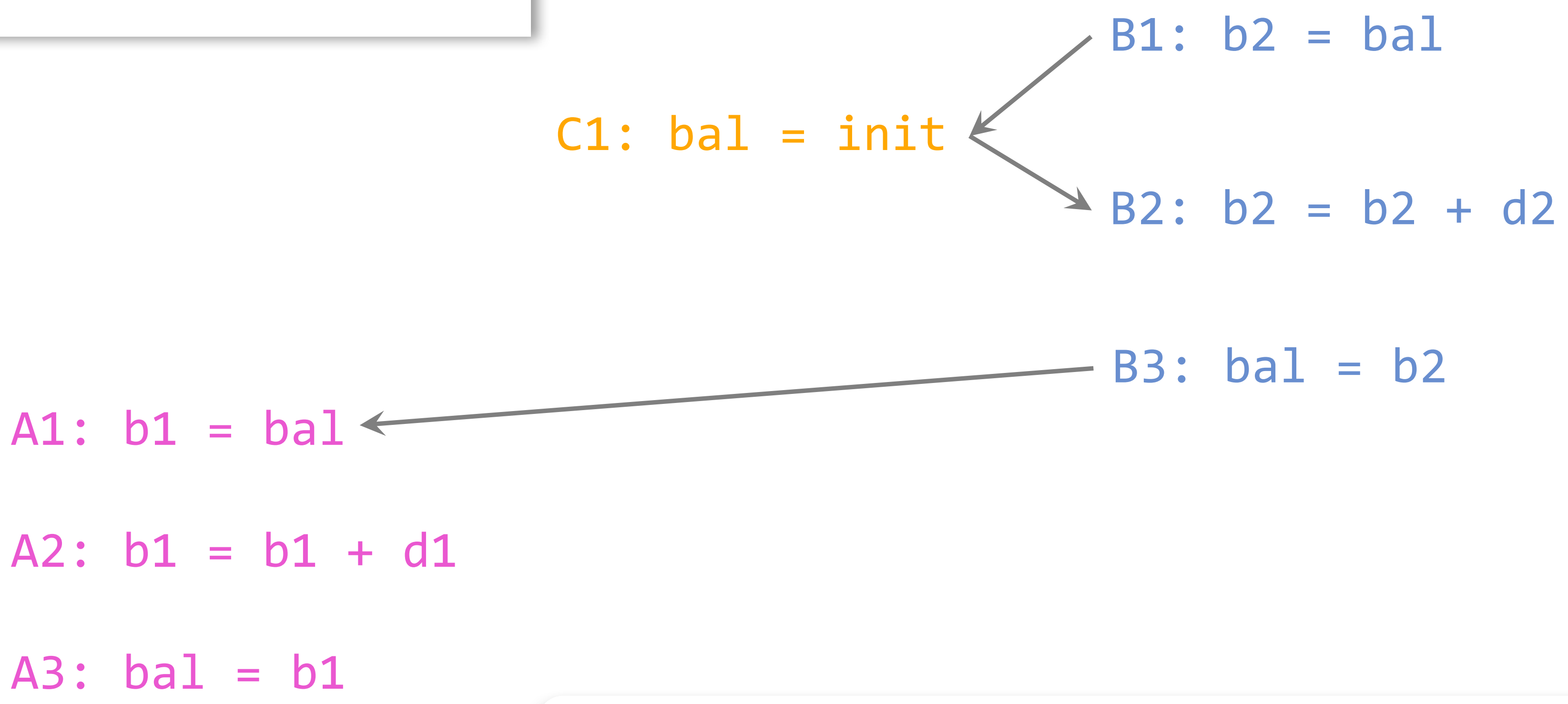

P
P’← P
Correctness
Specification
Verify?
yes
P’
no
Counterexample
gentrace ← Generalize(cex)
sync ← Eliminate(gentrace)
P’← Insert(P’, sync)

trace ➡ **H**appens **B**efore-formula

```
A1: b1 = bal          ||  C1: bal = init  ||  B1: b2 = bal
A2: b1 = b1 + 10      ||                  ||  B2: b2 = b2 + 20
A3: bal = b1          ||                  ||  B3: bal = b2
```

bal_new ≡ init + 30

Trace generalization



hb(B1,C1) ∧ hb(C1,B2) ∧ hb(B3,A1)

Trace generalization

```
B1: b2 = bal

C1: bal = init

B2: b2 = b2 + d2

B3: bal = b2

A1: b1 = bal

A2: b1 = b1 + d1

A3: bal = b1
```

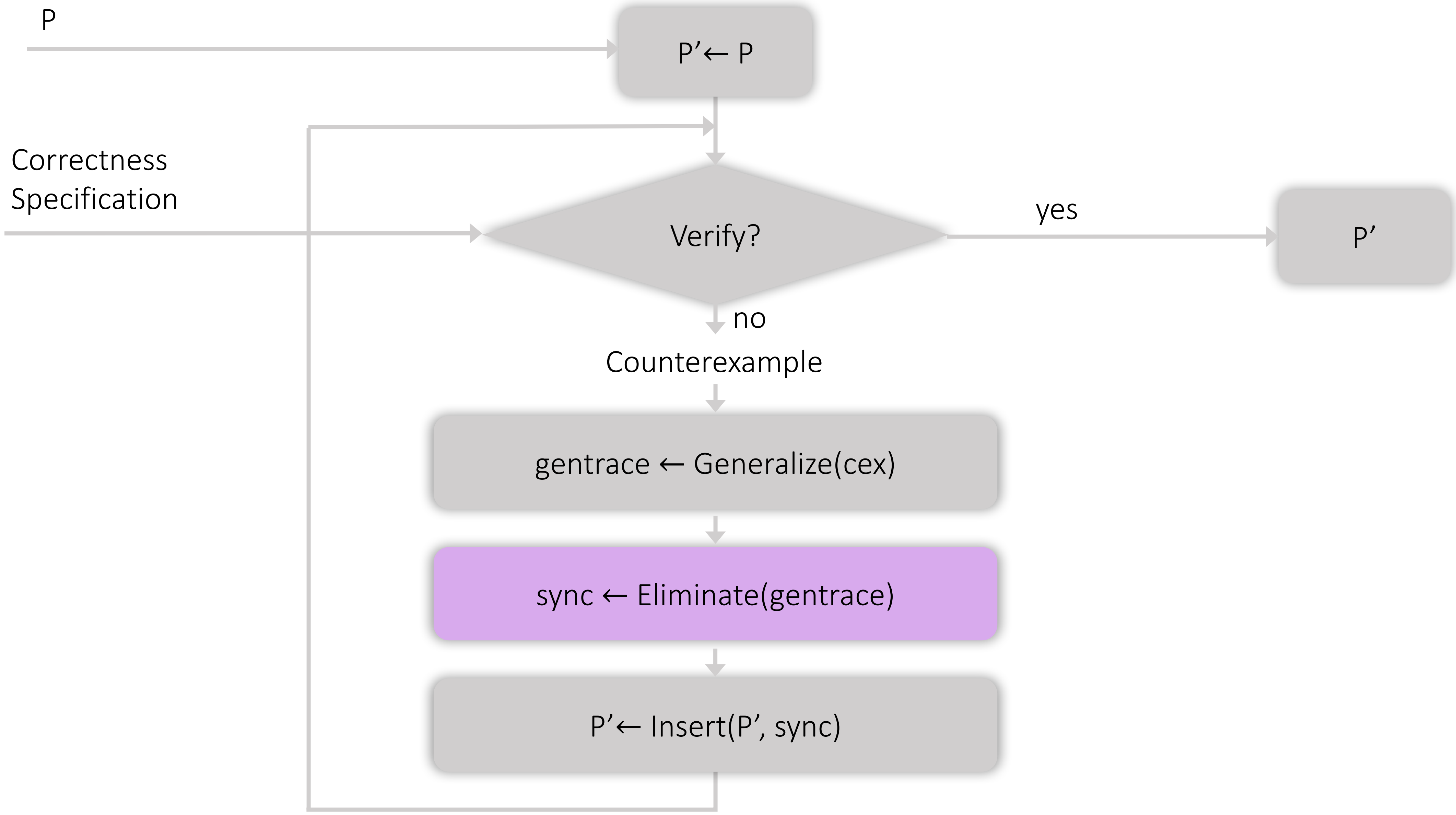
```
hb(B1,C1)
```

```
A1: b1 = bal          ||  C1: bal = init  ||  B1: b2 = bal
A2: b1 = b1 + 10      ||                  ||  B2: b2 = b2 + 20
A3: bal = b1          ||                  ||  B3: bal = b2
```

bal_new ≡ init + 30

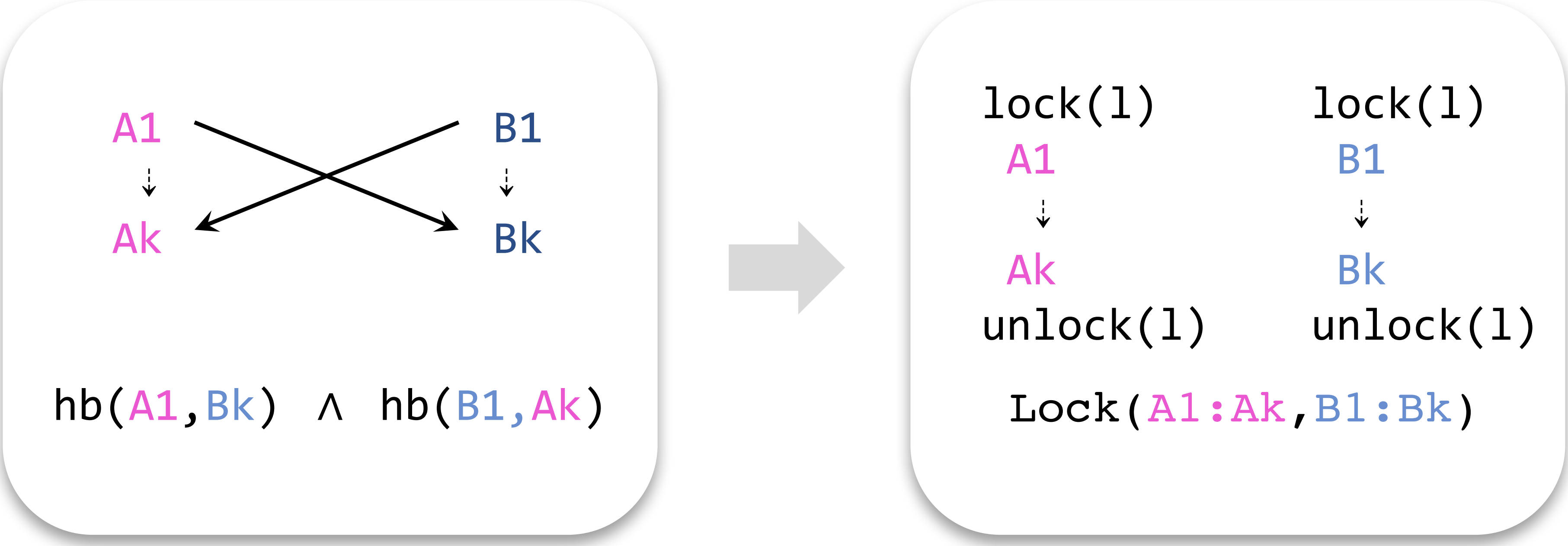
All incorrect related traces,
no correct related traces

hb(B1,C1)
∨
hb(A1,C1)
∨
hb(A1,B3) ∧ hb(B1,A3)

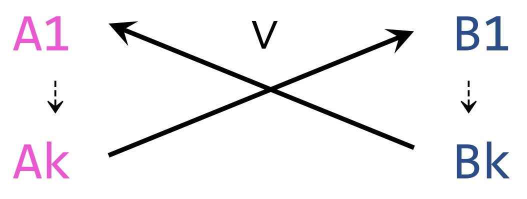

P
P'← P
Correctness
Specification
Verify?
yes
P'
no
Counterexample
gentrace ← Generalize(cex)
sync ← Eliminate(gentrace)
P'← Insert(P', sync)

HB-formula pattern ➡ Synchronization primitive

The **Lock** rewrite rule



The **Lock** rewrite rule

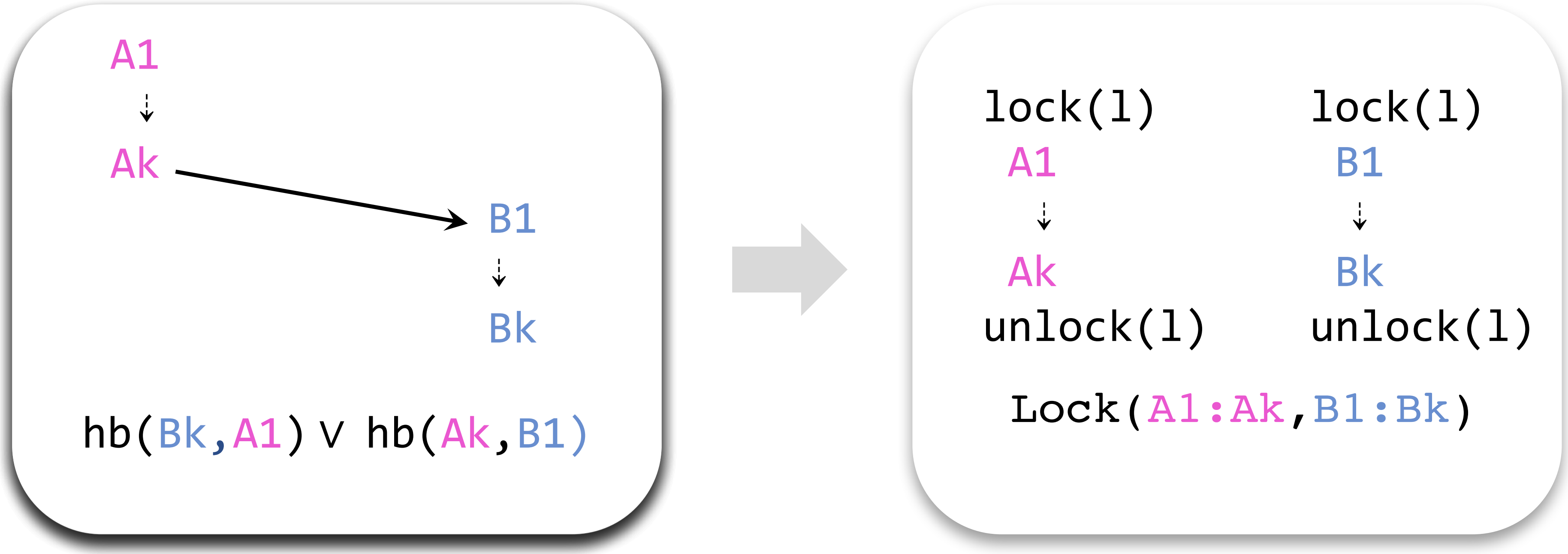



```
hb(Bk,A1) ∨ hb(Ak,B1)
```

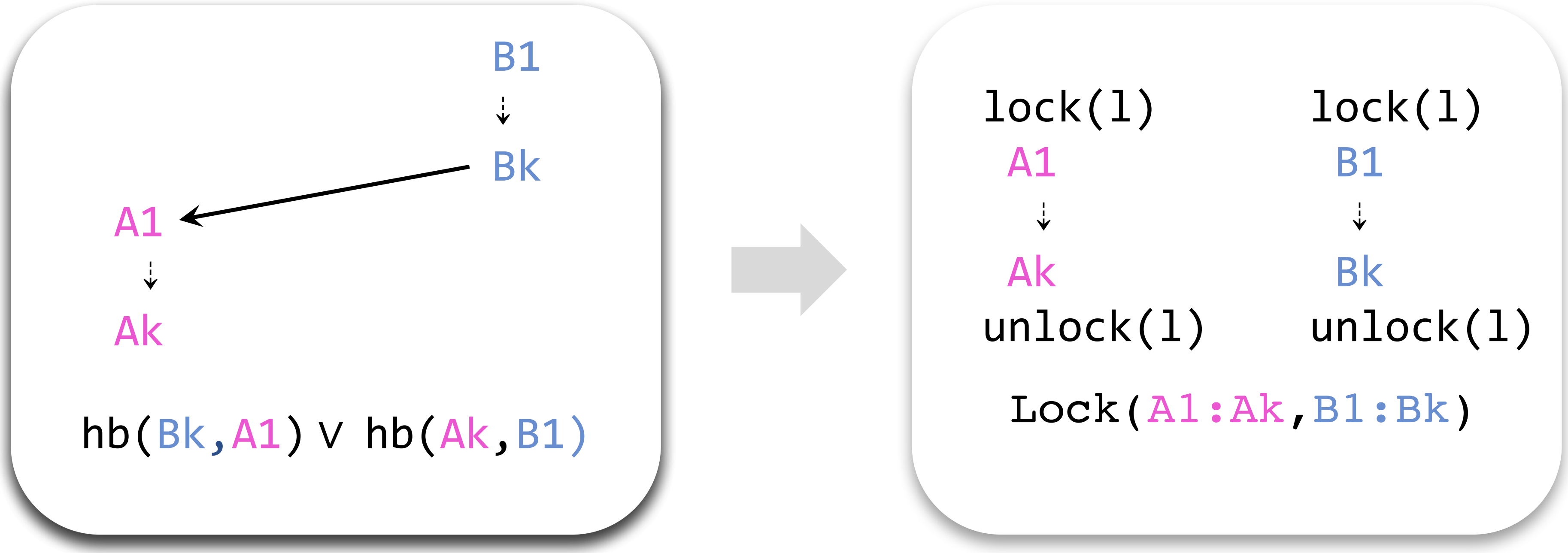
```
lock(l)        lock(l)
 A1             B1
 ⇣              ⇣
 Ak             Bk
unlock(l)      unlock(l)

 Lock(A1:Ak,B1:Bk)
```

The **Lock** rewrite rule



The **Lock** rewrite rule



$$hb(A1,C1) \lor hb(B1,C1) \lor hb(A1,B3) \land hb(B1,A3)$$

$$\Rightarrow$$

$$WaitNotify(A1,C1) \land WaitNotify(B1,C1) \land Lock(A1{:}A3,B1{:}B3)$$

```
wait(c)
lock(l)
A1: b1 = bal
A2: b1 = b1 + 10
A3: bal = b1
unlock(l)
```

||

```
C1: bal = init
notify(c)
```

||

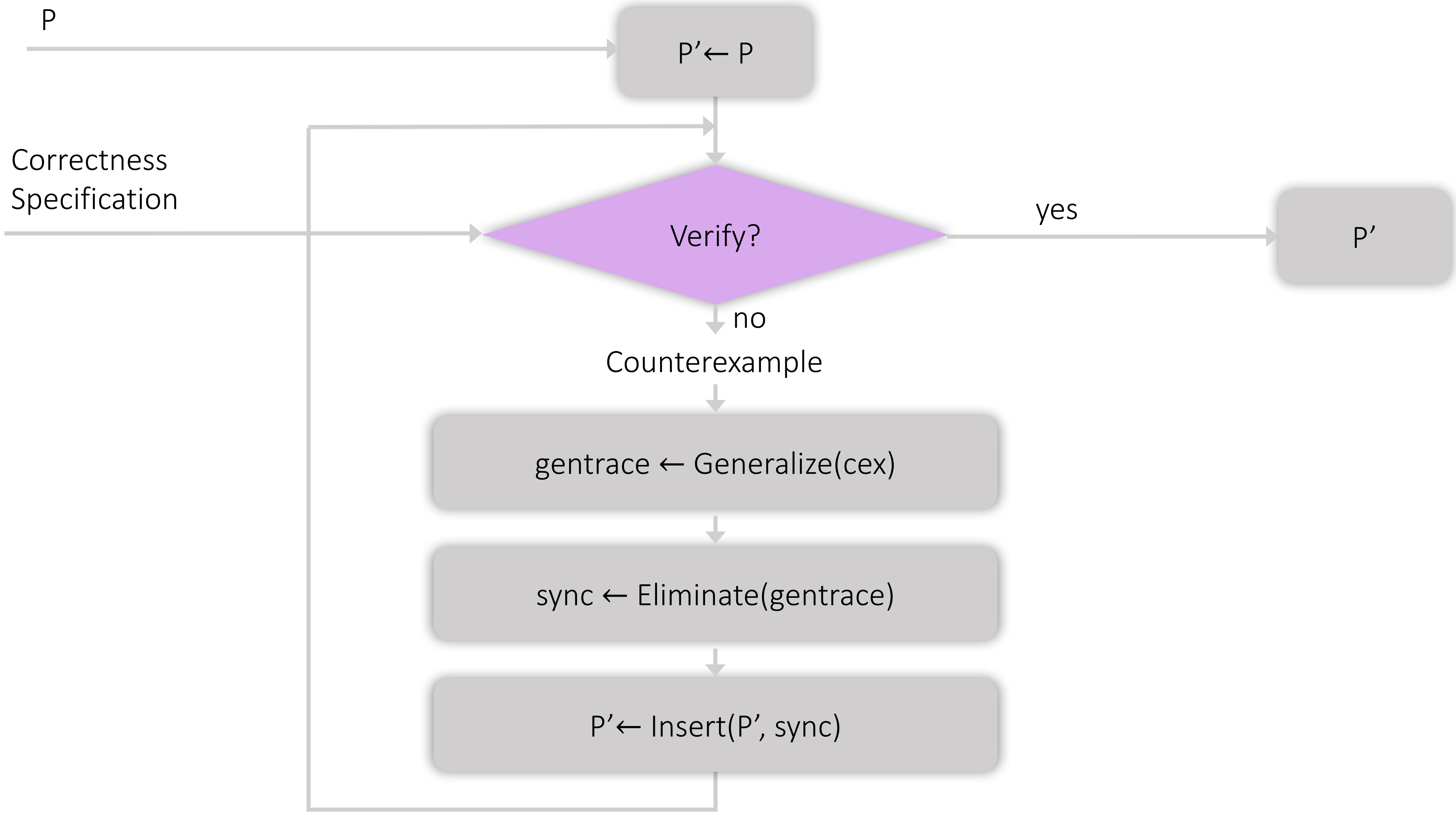
```
wait(c)
lock(l)
B1: b2 = bal
B2: b2 = b2 + 20
B3: bal = b2
unlock(l)
```

```
bal_new ≡ init + 30
```

```
     wait(c)              C1: bal = init            wait(c)
     lock(l)                 notify(c)              lock(l)
A1: b1 = bal          ||                  ||    B1: b2 = bal
A2: b1 = b1 + 10      ||                  ||    B2: b2 = b2 + 20
A3: bal = b1                                    B3: bal = b2
     unlock(l)                                      unlock(l)

                 bal_new ≡ init + 30
```

Guaranteed to eliminate all incorrect related traces


P
P'← P
Correctness
Specification
Verify?
yes
P'
no
Counterexample
gentrace ← Generalize(cex)
sync ← Eliminate(gentrace)
P'← Insert(P', sync)

- **Process:**
  Infinite-state program
- **Communication Model:**
  Shared-memory, , interleaving-based
- **Specification:**
  Implicit (behavior under non-preemptive scheduler), safety property
- **Synchronization:**
  Locks, wait-notify etc.
- **Procedure:**
  Counterexample generalization

- Counterexample generalization
- Specification-free synthesis
- Language inclusion verification procedure

From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis *

Pavol Černý[1], Edmund M. Clarke[2], Thomas A. Henzinger[3], Arjun Radhakrishna[4], Leonid Ryzhyk[2], Roopsha Samanta[3], and Thorsten Tarrach[3]

[1] University of Colorado Boulder
[2] Carnegie Mellon University
[3] IST Austria
[4] University of Pennsylvania

Abstract. We present a computer-aided programming approach to concurrency. The approach allows programmers to program assuming a friendly, non-preemptive scheduler, and our synthesis procedure inserts synchronization to ensure that the final program works even with a preemptive scheduler. The correctness specification is implicit, inferred from the non-preemptive behavior. Let us consider sequences of calls that the program makes to an external interface. The specification requires that any such sequence produced under a preemptive scheduler should be included in the set of such sequences produced under a non-preemptive scheduler. The solution is based on a finitary abstraction, an algorithm for bounded language inclusion modulo an independence relation, and rules for inserting synchronization. We apply the approach to device-driver programming, where the driver threads call the software interface of the device and the API provided by the operating system. Our experiments demonstrate that our synthesis method is precise and efficient, and, since it does not require explicit specifications, is more practical than the conventional approach based on user-provided assertions.

1 Introduction

Concurrent shared-memory programming is notoriously difficult and error-prone. Program synthesis for concurrency aims to mitigate this complexity by synthesizing synchronization code automatically [4, 5, 8, 11]. However, specifying the programmer's intent may be a challenge in itself. Declarative mechanisms, such as assertions, suffer from the drawback that it is difficult to ensure that the specification is complete and fully captures the programmer's intent.

We propose a solution where the specification is *implicit*. We observe that a core difficulty in concurrent programming originates from the fact that the scheduler can *preempt* the execution of a thread at any time. We therefore give

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), by NSF under award CCF 1421752 and the Expeditions award CCF 1138996, by the Simons Foundation, and by a gift from the Intel Corporation.

- Implicit specification is not universal
- Verification is computationally expensive

From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis *

Pavol Černý[1], Edmund M. Clarke[2], Thomas A. Henzinger[3], Arjun Radhakrishna[4], Leonid Ryzhyk[2], Roopsha Samanta[3], and Thorsten Tarrach[3]

[1] University of Colorado Boulder
[2] Carnegie Mellon University
[3] IST Austria
[4] University of Pennsylvania

Abstract. We present a computer-aided programming approach to concurrency. The approach allows programmers to program assuming a friendly, non-preemptive scheduler, and our synthesis procedure inserts synchronization to ensure that the final program works even with a preemptive scheduler. The correctness specification is implicit, inferred from the non-preemptive behavior. Let us consider sequences of calls that the program makes to an external interface. The specification requires that any such sequence produced under a preemptive scheduler should be included in the set of such sequences produced under a non-preemptive scheduler. The solution is based on a finitary abstraction, an algorithm for bounded language inclusion modulo an independence relation, and rules for inserting synchronization. We apply the approach to device-driver programming, where the driver threads call the software interface of the device and the API provided by the operating system. Our experiments demonstrate that our synthesis method is precise and efficient, and, since it does not require explicit specifications, is more practical than the conventional approach based on user-provided assertions.

1 Introduction

Concurrent shared-memory programming is notoriously difficult and error-prone. Program synthesis for concurrency aims to mitigate this complexity by synthesizing synchronization code automatically [4, 5, 8, 11]. However, specifying the programmer's intent may be a challenge in itself. Declarative mechanisms, such as assertions, suffer from the drawback that it is difficult to ensure that the specification is complete and fully captures the programmer's intent.

We propose a solution where the specification is *implicit*. We observe that a core difficulty in concurrent programming originates from the fact that the scheduler can *preempt* the execution of a thread at any time. We therefore give

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), by NSF under award CCF 1421752 and the Expeditions award CCF 1138996, by the Simons Foundation, and by a gift from the Intel Corporation.

A seminal paper

A modern approach

A trace-based approach

We have come a long way …

- Diverse specifications
- Infinite-state programs
- Diverse synchronization primitives
- Pushed scalability
- Performance-aware synthesis
- …

A seminal paper

A modern approach

A trace-based approach

… but we have miles to go.

- Assume sequential consistency
- Simple program models
- Simple performance models
- No optimistic concurrency control
- Scalability remains a challenge
- Fixed number of threads
- …

A seminal paper

A modern approach

A trace-based approach

Ongoing work

Jaber

Jacobs

Kulkarni

Samanta

Parameterized Synthesis for Distributed Applications with Consensus.

- **Process:** Finite-state synchronization skeleton
- **Communication Model:** Message-passing, partially asynchronous
- **Specification:** Temporal logic
- **Synchronization:** Guarded commands
- **Procedure:** Counterexample-based

- Parameterized verification
- Parameterized synthesis
- Abstract primitive for consensus protocols