Computer-aided Concurrent Programming

Roopsha Samanta
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

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

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, will decide whether it is satisfiable or unsatisfiable.

If it is satisfiable, a finite model of it 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 [L78] and [K80]. [L78] uses a specification language that is essentially predicate.

This work was partially supported by NSF Grant MCS-7908565.
Algorithmic framework to check and synthesize synchronization for temporal properties of finite-state transition systems
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, if the synchronization skeleton is a solution to the critical section problem, each process 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 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, will decide whether it is satisfiable or unsatisfiable. If a is satisfiable, a finite model of will be constructed. In our application, unsatisfiability of means that the specification is inconsistent (and must be reformulated). If the formula is satisfiable, then the specification is consistent. A model for 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 [LM86] and [KMD90]. [LM86] uses a specification language that is essentially predicated.
Mutual exclusion:
\( \mathbf{AG} \neg (CS_1 \land CS_2) \)

Absence of starvation:
\( \mathbf{AG} \ TRY_i \rightarrow \mathbf{AF} CS_i \)

Process specification:
\( \mathbf{AG} \ NCS_i \lor TRY_i \lor CS_i \)
\( \mathbf{AG} \ NCS_i \rightarrow \neg (TRY_i \lor CS_i) \)

...
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
Temporal logic primer

Kripke structure

Unwind

Computation tree

Temporal logics describe properties of infinite computation trees
Syntax of CTL

CTL /State formula

\[ g ::= p \mid \neg g \mid g_1 \lor g_2 \mid g_1 \land g_2 \mid A f \mid E f \]

Path formula:

\[ f ::= X g \mid F g \mid G g \mid g_1 \ U g_2 \]

Path quantifiers
- Always
- Exists

Temporal operators
- Nexttime
- Eventually
- Globally
- Until
Computation tree
Along all paths, q holds in every state
EF p

Exists a path, p holds eventually
EF AG \( q \land r \)

Exists a path, \( AG \ q \land r \) holds eventually
EF AG $q \land r$

Exists a path, AG $q \land r$ holds eventually
Exists a path, $\text{AG } q \land r$ holds eventually
\( \text{EF AG } q \land r \quad \text{AG } q \land r \quad q \land r \)

Exists a path, \( \text{AG } q \land r \) 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 $\text{EF } p \land \text{EF } \neg p$

node $\models f$ for all $f \in \text{label(node)}$

Figure from [CE81]
Delete inconsistent portions
Construct model from AND-nodes of tableau
Synchronization Synthesizer

Mutual exclusion:
\[ \text{AG} \neg (\text{CS}_1 \land \text{CS}_2) \]

Absence of starvation:
\[ \text{AG TRY}_i \rightarrow \text{AF CS}_i \]

Process specification:
\[ \text{AG NCS}_i \lor \text{TRY}_i \lor \text{CS}_i \]
\[ \text{AG NCS}_i \rightarrow \neg (\text{TRY}_i \lor \text{CS}_i) \]
...
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

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 [LAM8] and [IBM89]. [LAM8] uses a specification language that is essentially predicate

This work was partially supported by NSF Grant MCY-7908365.
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 [Kes80]. [LA78] uses a specification language that is essentially predicate.

This work was partially supported by NSF Grant MCS-7905165.
A seminal paper

A cool paper

A modern approach
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 Descriptions: 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.
Abstraction-based approach to infer synchronization to ensure safety properties of infinite-state concurrent programs
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.3.1 [Concurrent Programming]; D.2.4 [Programming Languages]; D.3.4 [Formal Techniques]; F.1.2 [Models and Principles]; F.1.3 [Logics and Meanings of Programs]; G.2 [Mathematics of Computing]; I.2.2 [Computer Applications]; J.6.1 [Theory of Computation]: I.6.2 [Logic in Computer Science]; I.6.3 [Software]: I.6.9 [Operating System]; I.6.9 [Program Verification]; J.6.9 [Theory of Computation]: I.6.4 [Programming Languages]

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.
- Beneficial 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM Permission Policy

© 2009 ACM/IEEE 1-60595-478-8/09/01...

10.100...
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

Abstract-guided synthesis

Synthesis as repair

Quantitative synthesis
Counterexample-guided abstraction refinement
Counterexample-guided repair/synthesis

Correctness Specification → Verify?

Verify?

P

no

Counterexample

φ

yes

P'

Program Restriction
Abstract counterexample-guided synthesis

- Correctness Specification
- Initial Abstraction

Verify?

- yes
- no

Abstract Counterexample

- \( \alpha \)
- \( \varphi \)

Abstraction Refinement

Program Restriction
Abstract counterexample-guided synchronization synthesis

Correctness Specification

Initial Abstraction

α

Abstraction Refinement

Abstraction Counterexample

Verify?

φ

Program Restriction

Atomicity constraints

P

yes

no

P′
Abstract counterexample

Atomicity constraint

A_1 \lor A_2 \lor A_3

Atomicity predicate
Obtain $P'$ from $P$ and $\varphi$ by adding minimal atomic sections satisfying $\varphi$.

Abstract counterexample-guided synchronization synthesis

Correctness Specification

Initial Abstraction

$\alpha$

Verify?

Abstract Counterexample

Abstraction Refinement

Program Restriction

Yes

No

$\varphi$

Atomicity constraints

$P'$
Abstract domain(s)

Synchronization Synthesizer

T1
x += z
x += z

T2
z++
z++

T3
y1 = f(x)
Y2 = x
assert y1≠y2
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 $P$ and a specification $S$, 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 a novel framework for automatic inference of efficient synchronization in concurrent programs. Our approach takes the two-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.
- Incremental 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM 2008.

Abstraction-guided synthesis

Synthesis as repair

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

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) interleaveings 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 schedulers for concurrent infinite-state programs. Safe schedulers can be realized by modifying the program or the scheduler:
- Concurrent programming: by automatically inferring minimal atomic sections that prevent unsafe schedulers, 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM Reference Format
Martin Vechev IBM Research
Etan Yuhav IBM Research
Greta Yorsh IBM Research

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

When $P \models \alpha S$, abstraction refinement approaches (e.g., [1, 3]) share the common goal of trying to find a finer abstraction $\alpha'$ 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 $\alpha$ and $P'$ admits a subset of the behavior 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 synchronizability.

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 ACOR 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, 5]). These include moving through a pre-determined series of domains with increasing precision (and typically increasing cost), or refining within the same abstract domain by changing its parameters (e.g., [8]).

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 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 [3], as it
A seminal paper

A cool paper

A modern approach
A modern approach

From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis

Černý, Clarke, Henzinger, Radhakrishna

Ryzhyk, Samanta, Tarrach

From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis. CAV 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 pre-emptive 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
Succinct Representation of Concurrent Trace Sets

Ashutosh Gupta
IST Austria
agupta@ist.ac.at

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

Arjun Radhakrishna
IST Austria
arjun@cs.cmu.edu

Roopsha Samanta
IST Austria
rsamanta@ist.ac.at

Thorsten Tarrach
IST Austria
tarrach@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 BHH-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 BHH-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 BHH-formula representing correct interleavings of a given trace. The algorithm applies rules to rewrite specific patterns in the BHH-formula into locks, barriers, and wait statements. In the second case study, we use a BHH-formula representing incorrect interleavings for bug summarization. While the BHH-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 BHH-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 BHH-formula.

Categories and Subject Descriptors D.2.4 [Formal methods]: Trace Generalization; Concurrent Programs; Synchronization Synthesis; Bug Summarization; CEGAR

1. Introduction
Sets of concurrent traces containing permutations of events from a given concurrent trace are useful for predictive analysis (e.g., [24, 34, 45, 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 P is an alternating sequence of variable valuations and events corresponding to a feasible interleaving of P. An execution is good if it satisfies a given specification, and bad otherwise. A trace π is a sequence of events corresponding to an interleaving of instructions from the threads of P. The trace of an execution π is the sequence of events within π. The language L(π) of a trace π is the set of all executions with trace π. A trace π is feasible if L(π) is non-empty, and infasible otherwise. A feasible trace π is good if all executions in L(π) are good, and bad otherwise.

We group traces into neighborhoods. The neighborhood Nπ of a trace π contains all permutations of π that preserve π’s interleaving order. The good neighborhood Nπ of a trace π is the set containing all the good traces in Nπ. The bad neighborhood Nπ of a trace π is the set containing all the bad traces in Nπ...
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 Černý1, Edmund M. Clarke2, Thomas A. Henzinger2, Arjun Radhakrishna4, Leonid Ryzhyk2, Roopsha Samanta2, and Thorsten Tarrach3

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 267089 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), by NSF under award CCF 1421762 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
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 Černý¹, Edmund M. Clarke², Thomas A. Henzinger³, Arjun Radhakrishna⁴, Leonid Ryzhyk⁵, Roopsha Samanta¹, and Thorsten Tarrach³

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 pre-emptive 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
Counterexample-guided repair/synthesis
The “lazy” approach
Counterexample-guided repair/synthesis
The “eager” approach

Correctness Specification → Verify?

Verify? yes → P′

Verify? no → Counterexample

Counterexample → P′ ← Program Restriction(P′)
Counterexample-guided repair/synthesis
The "eager" approach

Correctness Specification

Verify?

P' ← P

P' ← Program Restriction(P')

Counterexample

Eliminating one counterexample at a time may not be tractable
P → P’

Correctness
Specification

Verify?

no → Counterexample

gentrace ← Generalize(cex)

sync ← Eliminate(gentrace)

P’ ← Insert(P’, sync)

yes → P’

Generalize counterexample & eliminate all related counterexamples in each iteration!
Counterexample

gentrace ← Generalize(cex)

sync ← Eliminate(gentrace)

P' ← Insert(P', sync)
trace $\rightarrow$ Happens Before-formula
A1: b1 = bal
A2: b1 = b1 + 10
A3: bal = b1

C1: bal = init

B1: b2 = bal
B2: b2 = b2 + 20
B3: bal = b2

bal_new ≡ init + 30
A1: \( b_1 = \text{bal} \)

A2: \( b_1 = b_1 + d_1 \)

A3: \( \text{bal} = b_1 \)

B1: \( b_2 = \text{bal} \)

B2: \( b_2 = b_2 + d_2 \)

B3: \( \text{bal} = b_2 \)

C1: \( \text{bal} = \text{init} \)

\( \text{hb}(B1, C1) \land \text{hb}(C1, B2) \land \text{hb}(B3, A1) \)
A1: b1 = bal

A2: b1 = b1 + d1

A3: bal = b1

B1: b2 = bal

B2: b2 = b2 + d2

B3: bal = b2

C1: bal = init

hb(B1, C1)
A1: \( b_1 = \text{bal} \)
A2: \( b_1 = b_1 + 10 \)
A3: \( \text{bal} = b_1 \)

B1: \( b_2 = \text{bal} \)
B2: \( b_2 = b_2 + 20 \)
B3: \( \text{bal} = b_2 \)

\[ \text{bal\_new} \equiv \text{init} + 30 \]

All incorrect related traces, no correct related traces
\textbf{Correctness Specification}:

\begin{itemize}
  \item \textbf{P} \rightarrow \textbf{P'} \leftarrow \textbf{P}
  \item \textbf{Verify?}
  \begin{itemize}
    \item yes \rightarrow \textbf{P'}
    \item no \rightarrow \textbf{Counterexample}
  \end{itemize}
  \item \textbf{Counterexample}:
    \begin{itemize}
      \item \textbf{gentrace} \leftarrow \textbf{Generalize(cex)}
      \item \textbf{sync} \leftarrow \textbf{Eliminate(gentrace)}
      \item \textbf{P'} \leftarrow \textbf{Insert(P', sync)}
    \end{itemize}
\end{itemize}
HB-formula pattern $\rightarrow$ Synchronization primitive
The Lock rewrite rule

\[ \text{hb}(A_1, B_k) \land \text{hb}(B_1, A_k) \]

\[ \text{lock}(l) \]

\[ \text{unlock}(l) \]

\[ \text{Lock}(A_1: A_k, B_1: B_k) \]
The **Lock** rewrite rule

\[
\text{hb}(Bk,A1) \lor \text{hb}(Ak,B1) \rightarrow \text{lock}(1) \lor \text{unlock}(1) \rightarrow \text{Lock}(A1:Ak,B1:Bk)
\]
The Lock rewrite rule

\[ \text{hb}(B_k, A_1) \lor \text{hb}(A_k, B_1) \]

\[ \text{lock}(1) \]

\[ \text{unlock}(1) \]
The Lock rewrite rule

\[ \text{hb}(B_k, A_1) \lor \text{hb}(A_k, B_1) \]

\[ \text{lock}(1) \]

\[ \text{unlock}(1) \]

\[ \text{Lock}(A_1:A_k, B_1:B_k) \]
\( \text{hb}(A_1,C_1) \lor \text{hb}(B_1,C_1) \lor \text{hb}(A_1,B_3) \land \text{hb}(B_1,A_3) \)
wait(c) \hspace{1cm} C1: bal = init \hspace{1cm} wait(c)

lock(l) \hspace{1cm} notify(c) \hspace{1cm} lock(l)

A1: b1 = bal
A2: b1 = b1 + 10
A3: bal = b1

unlock(l)

B1: b2 = bal
B2: b2 = b2 + 20
B3: bal = b2

unlock(l)

bal_new \equiv init + 30
wait(c) \hspace{1cm} C1: bal = init \hspace{1cm} wait(c)

lock(l) \hspace{1cm} notify(c) \hspace{1cm} lock(l)

A1: b1 = bal
A2: b1 = b1 + 10
A3: bal = b1

B1: b2 = bal
B2: b2 = b2 + 20
B3: bal = b2

unlock(l) \hspace{1cm} unlock(l)

\[
\text{bal}_{\text{new}} \equiv \text{init} + 30
\]

Guaranteed to eliminate all incorrect related traces
P

Correctness Specification

Verify?

no

Counterexample

gentrace ← Generalize(cex)

sync ← Eliminate(gentrace)

P' ← Insert(P', sync)

P' ← P

yes

P'
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 Černý1, Edmund M. Clarke2, Thomas A. Henzinger3, Arjun Radhakrishna4, Leonid Ryzhyk5, Rooopsha Samanta6, and Thorsten Tarrach7

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 pre-emptive 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 267399 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), by NSF under award CCF 1421762 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 Černý1, Edmund M. Clarke2, Thomas A. Henzinger3, Arjun Radhakrishna4, Leonid Ryzhyk5, Roopsha Samanta6, and Thorsten Tarrach7

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 267069 (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

Parameterized Synthesis for Distributed Applications with Consensus.

Jaber     Jacobs

Kulkarni   Samanta

- **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**