Thanks for the introduction

I’ll be presenting our work on the automated repair of concurrent programs

This was collaborative work by myself, Markus, and my two colleagues Sepideh and Chao
Typical Day Programming

Even though I am sure we are all more than familiar, let me go over what I believe to be the typical day of a programmer.

You start your day at your desk, staring at your screen, building the next best and most world changing software.

This process mostly involves fighting with the compiler, fixing syntax errors, and reworking the design.

However, eventually all your sound ideas have been created, the compiler is happy, and you feel its time for a well deserved break.

Just for good measure, you decide to actually run your program just to admire your handy work.

At this point, your beautiful creation decides to lash out with a runtime segmentation fault.

Its only now that the real hard work begins: you need to answer two complicated questions.

First, where is the bug? Often a bug is quite difficult to localize especially complex concurrency bugs involving a multitude of threads.

Even once the bug is found, you still need to find an appropriate fix: the fix needs to not only correct the current issue but also needs to not introduce new bugs.
Even though I am sure we are all more than familiar, let me go over what I believe to be the typical day of a programmer.

You start your day at your desk, staring at your screen, building the next best and most world changing software.

This process mostly involves fighting with the compiler, fixing syntax errors, and reworking the design.

However, eventually all your sound ideas have been created, the compiler is happy, and you feel its time for a well deserved break.

Just for good measure, you decide to actually run your program just to admire your handy work.

At this point, your beautiful creation decides to lash out with a runtime segmentation fault.

It's only now that the real hard work begins: you need to answer two complicated questions.

First, where is the bug? Often a bug is quite difficult to localize especially complex concurrency bugs involving a multitude of threads.

Even once the bug is found, you still need to find an appropriate fix: the fix needs to not only correct the current issue but also needs to not introduce new bugs.
Typical Day Programming

- Even though I am sure we are all more than familiar, let me go over what I believe to be the typical day of a programmer.
- You start your day at your desk, staring at your screen, building the next best and most world changing software.
- This process mostly involves fighting with the compiler, fixing syntax errors, and reworking the design.
- However, eventually all your sound ideas have been created, the compiler is happy, and you feel it's time for a well deserved break.
- Just for good measure, you decide to actually run your program just to admire your handy work.
- At this point, your beautiful creation decides to lash out with a runtime segmentation fault.
- It's only now that the real hard work begins: you need to answer two complicated questions.
- First, where is the bug? Often a bug is quite difficult to localize especially complex concurrency bugs involving a multitude of threads.
- Even once the bug is found, you still need to find an appropriate fix: the fix needs to not only correct the current issue but also needs to not introduce new bugs.
Typical Day Programming

• Even though I am sure we are all more than familiar, let me go over what I believe to be the typical day of a programmer
• You start your day at your desk, staring at your screen, building the next best and most world changing software
• This process mostly involves fighting with the compiler, fixing syntax errors, and reworking the design
• However, eventually all your sound ideas have been created, the compiler is happy, and you feel its time for a well deserved break
• Just for good measure, you decide to actually run your program just to admire your handy work
• At this point, your beautiful creation decides to lash out with a runtime segmentation fault
• Its only now that the real hard work begins: you need to answer two complicated questions
• First, where is the bug? Often a bug is quite difficult to localize especially complex concurrency bugs involving a multitude of threads
• Even once the bug is found, you still need to find an appropriate fix: the fix needs to not only correct the current issue but also needs to not introduce new bugs
The goal of our new tool is to allow programmers to have a glorious day developing multithreaded programs.

The beginning of the story is the same, the programmer creates a new, but obviously imperfect, program.

Then, our tool, ConcBugAssist, detects and localizes faults in the concurrent program.

These localizations describe precise thread interactions leading to the bug.

After that, we go even further: we propose a modification of the original program to repair the bug.

At this point, I want to be clear that the proposed repair is correct only with respect to the verification procedure.
Glorious Day Programming

Programming...

ConcBugAssist: Bugs found!

- The goal of our new tool is to allow programmers to have a glorious day developing multithreaded programs.
- The beginning of the story is the same, the programmer creates a new, but obviously imperfect, program.
- Then, our tool, ConcBugAssist, detects and localizes faults in the concurrent program.
- These localizations describe precise thread interactions leading to the bug.
- After that, we go even further: we propose a modification of the original program to repair the bug.
- At this point, I want to be clear that the proposed repair is correct only with respect to the verification procedure.
The goal of our new tool is to allow programmers to have a glorious day developing multithreaded programs.

The beginning of the story is the same, the programmer creates a new, but obviously imperfect, program.

Then, our tool, ConcBugAssist, detects and localizes faults in the concurrent program.

These localizations describe precise thread interactions leading to the bug.

After that, we go even further: we propose a modification of the original program to repair the bug.

At this point, I want to be clear that the proposed repair is correct only with respect to the verification procedure.
The goal of our new tool is to allow programmers to have a glorious day developing multithreaded programs.

The beginning of the story is the same, the programmer creates a new, but obviously imperfect, program.

Then, our tool, ConcBugAssist, detects and localizes faults in the concurrent program.

These localizations describe precise thread interactions leading to the bug.

After that, we go even further: we propose a modification of the original program to repair the bug.

At this point, I want to be clear that the proposed repair is correct only with respect to the verification procedure.
An Example

```c
struct list {
    int arr[MAX_SIZE];
    size_t open;
} gl;

void list_add(list_t *s, int i) {
    s->arr[s->open] = i;
    s->open += 1;
}

void t1_main() {
    int val;
    val = 1;
    list_add(&gl, val);
    return;
}

void t2_main() {
    int val;
    val = 2;
    list_add(&gl, val);
    return;
}

int main() {
    thread_t t1, t2;
    thread_create(&t1, t1_main);
    thread_create(&t2, t2_main);
    thread_join(t1);
    thread_join(t2);
    assert(list_contains(&gl, 1) && list_contains(&gl, 2));
    return 0;
}
```

• To be more concrete, here is an example of our tool in action
• Our tool works to localize and repair concurrency bugs
• By concurrency bugs, I mean those caused by some particular thread schedule rather than the sequential logic within a thread
• Consider this program where two threads modify a shared list
• The list is represented as an array with a integer value holding the next open position
• Two threads are both concurrently adding items to the list
• The correctness specification states that after the threads execute, the list contains both of their items.
• However, the add operation itself is unsynchronized: the update of the array and the update of the next available index is non-atomic
• As a result, the add operation of one thread may overwrite the result of the other thread causing the specification to be violated.
• First, the result of our localization procedure presents the programmer with minimal thread schedules leading to an assertion violation
• One such schedule is thread one’s add being overwritten by thread two
• An additional buggy schedule is the converse where thread two’s result is overwritten by thread one
• By minimal thread schedule, I mean that our tool returns a schedule containing only those statements which are involved in the bug rather than an complete schedule involving all statements in the program
An Example

To be more concrete, here is an example of our tool in action

Our tool works to localize and repair concurrency bugs

By concurrency bugs, I mean those caused by some particular thread schedule rather than the sequential logic within a thread

Consider this program where two threads modify a shared list

The list is represented as an array with a integer value holding the next open position

Two threads are both concurrently adding items to the list

The correctness specification states that after the threads execute, the list contains both of their items.

However, the add operation itself is unsynchronized: the update of the array and the update of the next available index is non-atomic

As a result, the add operation of one thread may overwrite the result of the other thread causing the specification to be violated.

First, the result of our localization procedure presents the programmer with minimal thread schedules leading to an assertion violation

One such schedule is thread one’s add being overwritten by thread two

An additional buggy schedule is the converse where thread two’s result is overwritten by thread one

By minimal thread schedule, I mean that our tool returns a schedule containing only those statements which are involved in the bug rather than an complete schedule involving all statements in the program
To be more concrete, here is an example of our tool in action

Our tool works to localize and repair concurrency bugs

By concurrency bugs, I mean those caused by some particular thread schedule rather than the sequential logic within a thread

Consider this program where two threads modify a shared list

The list is represented as an array with an integer value holding the next open position

Two threads are both concurrently adding items to the list

The correctness specification states that after the threads execute, the list contains both of their items.

However, the add operation itself is unsynchronized: the update of the array and the update of the next available index is non-atomic

As a result, the add operation of one thread may overwrite the result of the other thread causing the specification to be violated.

First, the result of our localization procedure presents the programmer with minimal thread schedules leading to an assertion violation

One such schedule is thread one’s add being overwritten by thread two

An additional buggy schedule is the converse where thread two’s result is overwritten by thread one

By minimal thread schedule, I mean that our tool returns a schedule containing only those statements which are involved in the bug rather than an complete schedule involving all statements in the program
void list_add ( list_t *s, int i) {
    s-> arr [s-> open ] = i;
    s-> open += 1;
}

void t1_main () {
    int val;
    val = 1;
    list_add(&gl, val);
    return ;
}

void t2_main () {
    int val;
    val = 2;
    list_add(&gl, val);
    return ;
}

int main () {
    thread_t t1, t2;
    thread_create(&t1, t1_main);
    thread_create(&t2, t2_main);
    thread_join(t1);
    thread_join(t2);
    assert ( list_contains (&gl , 1)
        && list_contains (&gl , 2));
    return 0;
}
To be more concrete, here is an example of our tool in action:

Our tool works to localize and repair concurrency bugs.

By concurrency bugs, I mean those caused by some particular thread schedule rather than the sequential logic within a thread.

Consider this program where two threads modify a shared list.

The list is represented as an array with an integer value holding the next open position.

Two threads are both concurrently adding items to the list.

The correctness specification states that after the threads execute, the list contains both of their items.

However, the add operation itself is unsynchronized: the update of the array and the update of the next available index is non-atomic.

As a result, the add operation of one thread may overwrite the result of the other thread causing the specification to be violated.

First, the result of our localization procedure presents the programmer with minimal thread schedules leading to an assertion violation.

One such schedule is thread one’s add being overwritten by thread two.

An additional buggy schedule is the converse where thread two’s result is overwritten by thread one.

By minimal thread schedule, I mean that our tool returns a schedule containing only those statements which are involved in the bug rather than an complete schedule involving all statements in the program.
An Example

To be more concrete, here is an example of our tool in action

Our tool works to localize and repair concurrency bugs

By concurrency bugs, I mean those caused by some particular thread schedule rather than the sequential logic within a thread

Consider this program where two threads modify a shared list

The list is represented as an array with a integer value holding the next open position

Two threads are both concurrently adding items to the list

The correctness specification states that after the threads execute, the list contains both of their items.

However, the add operation itself is unsynchronized: the update of the array and the update of the next available index is non-atomic

As a result, the add operation of one thread may overwrite the result of the other thread causing the specification to be violated.

First, the result of our localization procedure presents the programmer with minimal thread schedules leading to an assertion violation

One such schedule is thread one’s add being overwritten by thread two

An additional buggy schedule is the converse where thread two’s result is overwritten by thread one

By minimal thread schedule, I mean that our tool returns a schedule containing only those statements which are involved in the bug rather than an complete schedule involving all statements in the program
An Example

To be more concrete, here is an example of our tool in action.

Our tool works to localize and repair concurrency bugs.

By concurrency bugs, I mean those caused by some particular thread schedule rather than the sequential logic within a thread.

Consider this program where two threads modify a shared list:
- The list is represented as an array with an integer value holding the next open position.
- Two threads are both concurrently adding items to the list.
- The correctness specification states that after the threads execute, the list contains both of their items.
- However, the add operation itself is unsynchronized: the update of the array and the update of the next available index is non-atomic.
- As a result, the add operation of one thread may overwrite the result of the other thread causing the specification to be violated.

First, the result of our localization procedure presents the programmer with minimal thread schedules leading to an assertion violation.
- One such schedule is thread one’s add being overwritten by thread two.
- An additional buggy schedule is the converse where thread two’s result is overwritten by thread one.

By minimal thread schedule, I mean that our tool returns a schedule containing only those statements which are involved in the bug rather than an complete schedule involving all statements in the program.
An Example

```c
struct list {
    int arr[MAX_SIZE];
    size_t open;
} gl;

void list_add(list_t *s, int i) {
    s->arr[s->open] = i;
    s->open += 1;
}

void t1_main() {
    int val;
    val = 1;
    list_add(&gl, val);
    return;
}

void t2_main() {
    int val;
    val = 2;
    list_add(&gl, val);
    return;
}

int main() {
    thread_t t1, t2;
    thread_create(&t1, t1_main);
    thread_create(&t2, t2_main);
    thread_join(t1);
    thread_join(t2);
    assert(list_contains(&gl, 1) && list_contains(&gl, 2));
    return 0;
}
```

- The second phase of our tool takes these fault localizations and uses them to propose potential repairs to the user
- One such repair is enforcing an ordering constraint between thread one’s add and thread two’s
- Conversely, thread two could be constrained to execute before thread one
- Each of these repairs is a happens-before constraints which, when added to the program, would prevent the assertion violation
- A single happens-before edge could be enforced using a signal–wait operation across threads
An Example

```c
struct list {
    int arr[MAX_SIZE];
    size_t open;
} gl;

void list_add(list_t *s, int i) {
    s->arr[s->open] = i;
    s->open += 1;
}

void t1_main() {
    int val;
    val = 1;
    list_add(&gl, val);
    return;
}

void t2_main() {
    int val;
    val = 2;
    list_add(&gl, val);
    return;
}

int main() {
    thread_t t1, t2;
    thread_create(&t1, t1_main);
    thread_create(&t2, t2_main);
    thread_join(t1);
    thread_join(t2);
    assert(list_contains(&gl, 1) && list_contains(&gl, 2));
    return 0;
}
```

- The second phase of our tool takes these fault localizations and uses them to propose potential repairs to the user
- One such repair is enforcing an ordering constraint between thread one’s add and thread two’s
- Conversely, thread two could be constrained to execute before thread one
- Each of these repairs is a happens-before constraints which, when added to the program, would prevent the assertion violation
- A single happens-before edge could be enforced using a signal–wait operation across threads
An Example

```c
struct list {
    int arr[MAX_SIZE];
    size_t open;
} gl;

void list_add(list_t *s, int i) {
    s->arr[s->open] = i;
    s->open += 1;
}

void t1_main() {
    int val;
    val = 1;
    list_add(&gl, val);
    return;
}

void t2_main() {
    int val;
    val = 2;
    list_add(&gl, val);
    return;
}

int main() {
    thread_t t1, t2;
    thread_create(&t1, t1_main);
    thread_create(&t2, t2_main);
    thread_join(t1);
    thread_join(t2);
    assert(list_contains(&gl, 1) && list_contains(&gl, 2));
    return 0;
}
```

- The second phase of our tool takes these fault localizations and uses them to propose potential repairs to the user.
- One such repair is enforcing an ordering constraint between thread one’s add and thread two’s.
- Conversely, thread two could be constrained to execute before thread one.
- Each of these repairs is a happens-before constraints which, when added to the program, would prevent the assertion violation.
- A single happens-before edge could be enforced using a signal–wait operation across threads.
An Example

```c
cond_var c1;
void t1_main() {
    int val;
    val = 1;
    list_add(&gl, val);
    signal(c1);
    return;
}
void t2_main() {
    int val;
    val = 2;
    wait(c1);
    list_add(&gl, val);
    return;
}
```

- For instance, here is one potential solution where thread one is guaranteed to execute before thread two
- The condition variable enforces the proposed happens-before constraint
- In other words, the wait operation forces thread two to always add after thread one
- However, examining this solution, while it prevents the error, it is somewhat unsatisfactory
- We lose the extra potential parallelism by preventing thread two from running before thread one
cond_var c1;
void t1_main() {
    int val;
    val = 1;
    list_add(&gl, val);
    signal(c1);
    return;
}
void t2_main() {
    int val;
    val = 2;
    wait(c1);
    list_add(&gl, val);
    return;
}

• For instance, here is one potential solution where thread one is guaranteed to execute before thread two
• The condition variable enforces the proposed happens-before constraint
• In other words, the wait operation forces thread two to always add after thread one
• However, examining this solution, while it prevents the error, it is somewhat unsatisfactory
• We lose the the extra potential parallelism by preventing thread two from running before thread one
An Example

```c
struct list {
    int arr[MAX_SIZE];
    size_t open;
} gl;

void list_add(list_t *s, int i) {
    s->arr[s->open] = i;
    s->open += 1;
}

void t1_main() {
    int val;
    val = 1;
    list_add(&gl, val);
    return;
}

void t2_main() {
    int val;
    val = 2;
    list_add(&gl, val);
    return;
}

int main() {
    thread_t t1, t2;
    thread_create(&t1, t1_main);
    thread_create(&t2, t2_main);
    thread_join(t1);
    thread_join(t2);
    assert(list_contains(&gl, 1) && list_contains(&gl, 2));
    return 0;
}
```

• However, examining the possible solutions we can see that there is a third possibility
• Notice that either thread one can execute before thread two or thread two can execute before thread one
• This either–or solution matches perfectly the design pattern of a mutex lock
mutex m;
void t1_main () {
  int val;
  val = 1;
  lock (m);
  list_add (&gl, val);
  unlock (m);
  return;
}

void t2_main () {
  int val;
  val = 2;
  wait (c1);
  lock (m);
  list_add (&gl, val);
  unlock (m);
  return;
}

• Here, we can see the automated fix of the third potential solution
• A mutex is inserted surrounding the add operations to enforce the
  either–or constraints
• Here, we can see a fairly satisfactory repair
• The solution allows for additional parallelism compared to naively
  using a signal–wait pair to enforce one of the ordering constraints
• To us, we found it interesting that our tool automatically lifted a
  sequential list into a coarse-grained concurrent list
In summary, here are our contributions

- We created a tool called ConcBugAssist
- ConcBugAssist is an extension of concurrent bounded model checking performing automated fault localization and repair
- We formulate the problem of concurrent fault localization as a solution to the partial Max-SAT problem
- We find potential repairs by solving the problem of a binate cover
- Both of these approaches are constraint based, which differs from prior work in this area using mostly heuristics and pattern-matching.
- To validate our approach, we obtained some empirical data as well as some case studies showing the results of our tool
- The remainder of this talk will present these points in detail
Our Contributions

- ConcBugAssist
- Extension to bounded model checking

- In summary, here are our contributions
- We created a tool called ConcBugAssist
- ConcBugAssist is an extension of concurrent bounded model checking performing automated fault localization and repair
- We formulate the problem of concurrent fault localization as a solution to the partial Max-SAT problem
- We find potential repairs by solving the problem of a binate cover
- Both of these approaches are constraint based, which differs from prior work in this area using mostly heuristics and pattern-matching.
- To validate our approach, we obtained some empirical data as well has some case studies showing the results of our tool
- The remainder of this talk will present these points in detail
Our Contributions

- ConcBugAssist
- Extension to bounded model checking
- Concurrent fault localization (partial Max-SAT)

- In summary, here are our contributions
- We created a tool called ConcBugAssist
- ConcBugAssist is an extension of concurrent bounded model checking performing automated fault localization and repair
- We formulate the problem of concurrent fault localization as a solution to the partial Max-SAT problem
- We find potential repairs by solving the problem of a binate cover
- Both of these approaches are constraint based, which differs from prior work in this area using mostly heuristics and pattern-matching.
- To validate our approach, we obtained some empirical data as well has some case studies showing the results of our tool
- The remainder of this talk will present these points in detail
Our Contributions

- ConcBugAssist
- Extension to bounded model checking
- Concurrent fault localization (partial Max-SAT)
- Concurrent fault repair (binate covering)

- In summary, here are our contributions
- We created a tool called ConcBugAssist
- ConcBugAssist is an extension of concurrent bounded model checking performing automated fault localization and repair
- We formulate the problem of concurrent fault localization as a solution to the partial Max-SAT problem
- We find potential repairs by solving the problem of a binate cover
- Both of these approaches are constraint based, which differs from prior work in this area using mostly heuristics and pattern-matching.
- To validate our approach, we obtained some empirical data as well has some case studies showing the results of our tool
- The remainder of this talk will present these points in detail
Our Contributions

- ConcBugAssist
- Extension to bounded model checking
- Concurrent fault localization (partial Max-SAT)
- Concurrent fault repair (binate covering)
- Constraint based

- In summary, here are our contributions
- We created a tool called ConcBugAssist
- ConcBugAssist is an extension of concurrent bounded model checking performing automated fault localization and repair
- We formulate the problem of concurrent fault localization as a solution to the partial Max-SAT problem
- We find potential repairs by solving the problem of a binate cover
- Both of these approaches are constraint based, which differs from prior work in this area using mostly heuristics and pattern-matching.
- To validate our approach, we obtained some empirical data as well has some case studies showing the results of our tool
- The remainder of this talk will present these points in detail
Our Contributions

- ConcBugAssist
- Extension to bounded model checking
- Concurrent fault localization (partial Max-SAT)
- Concurrent fault repair (binate covering)
- Constraint based
- Empirical results and case studies

- In summary, here are our contributions
- We created a tool called ConcBugAssist
- ConcBugAssist is an extension of concurrent bounded model checking performing automated fault localization and repair
- We formulate the problem of concurrent fault localization as a solution to the partial Max-SAT problem
- We find potential repairs by solving the problem of a binate cover
- Both of these approaches are constraint based, which differs from prior work in this area using mostly heuristics and pattern-matching.
- To validate our approach, we obtained some empirical data as well has some case studies showing the results of our tool
- The remainder of this talk will present these points in detail
Next, I will provide a brief background introduction to bounded model checking.
Bounded Model Checking

- Find property violations via satisfiability

- At a high level, bounded model checking converts the problem of finding a property violation into the satisfiability of a logical formula.

- The process is bounded in that loops and recursion are unrolled to some fixed depth.

- This guarantees that the program is terminating and guarantees that the logical formula representing the program will have a finite size.

- Since bounded model checking has been well studied in the past, we treat the procedure in this presentation as a black box.

- All we really need to know is that the bounded model checker as some logical representation of the program, $\phi$.

- $\phi$ represents all possible behaviors in the program, in other words, the behavior across all valid paths.

- Then, if we want to test an assertion that property $p$ holds, we check the satisfiability of $\phi$ and not $p$.

- Intuitively, this satisfiability problems asks if there is some possible behavior of the program, represented in $\phi$, such that the property $p$ does not hold.

- If there is a satisfiable assignment then it represents some valuation of the program inputs and internal variables such that the property is violated.

- If the result is unsatisfiable, then the property holds at least up to the tested bound.
Bounded Model Checking

- Find property violations via satisfiability
- Unroll loops/recursion to certain depth

- At a high level, bounded model checking converts the problem of finding a property violation into the satisfiability of a logical formula
- The process is bounded in that loops and recursion are unrolled to some fixed depth
- This guarantees that the program is terminating and guarantees that the logical formula representing the program will have a finite size
- Since bounded model checking has been well studied in the past, we treat the procedure in this presentation as a black box
- All we really need to know is that the bounded model checker as some logical representation of the program, \( \phi \)
- \( \phi \) represents all possible behaviors in the program, in other words, the behavior across all valid paths
- Then, if we want to test an assertion that property \( p \) holds, we check the satisfiability of \( \phi \) and not \( p \)
- Intuitively, this satisfiability problems asks if there is some possible behavior of the program, represented in \( \phi \), such that the property \( p \) does not hold
- If there is a satisfiable assignment then it represents some valuation of the program inputs and internal variables such that the property is violated
- If the result is unsatisfiable, then the property holds at least up to the tested bound
Bounded Model Checking

- Find property violations via satisfiability
- Unroll loops/recursion to certain depth
- $\phi$: formula modeling program behavior

- At a high level, bounded model checking converts the problem of finding a property violation into the satisfiability of a logical formula.
- The process is bounded in that loops and recursion are unrolled to some fixed depth.
- This guarantees that the program is terminating and guarantees that the logical formula representing the program will have a finite size.
- Since bounded model checking has been well studied in the past, we treat the procedure in this presentation as a black box.
- All we really need to know is that the bounded model checker as some logical representation of the program, $\phi$.
- $\phi$ represents all possible behaviors in the program, in other words, the behavior across all valid paths.
- Then, if we want to test an assertion that property $p$ holds, we check the satisfiability of $\phi$ and not $p$.
- Intuitively, this satisfiability problem asks if there is some possible behavior of the program, represented in $\phi$, such that the property $p$ does not hold.
- If there is a satisfiable assignment then it represents some valuation of the program inputs and internal variables such that the property is violated.
- If the result is unsatisfiable, then the property holds at least up to the tested bound.
Bounded Model Checking

- Find property violations via satisfiability
- Unroll loops/recursion to certain depth
- $\phi$: formula modeling program behavior
- $\text{assert}(p) : \phi \land \neg p$

- At a high level, bounded model checking converts the problem of finding a property violation into the satisfiability of a logical formula
- The process is bounded in that loops and recursion are unrolled to some fixed depth
- This guarantees that the program is terminating and guarantees that the logical formula representing the program will have a finite size
- Since bounded model checking has been well studied in the past, we treat the procedure in this presentation as a black box
- All we really need to know is that the bounded model checker as some logical representation of the program, $\phi$
- $\phi$ represents all possible behaviors in the program, in other words, the behavior across all valid paths
- Then, if we want to test an assertion that property $p$ holds, we check the satisfiability of $\phi$ and not $p$
- Intuitively, this satisfiability problem asks if there is some possible behavior of the program, represented in $\phi$, such that the property $p$ does not hold
- If there is a satisfiable assignment then it represents some valuation of the program inputs and internal variables such that the property is violated
- If the result is unsatisfiable, then the property holds at least up to the tested bound
• Bounded model checking of concurrent programs is a natural extension to the previously described method.

• Again, a single logical formula represents all possible behaviors of the program.

• The logical representation consists of two parts: first, there is a trace formula for each thread.

• The trace formula represents the possible behaviors of each thread in the program.

• Each statement in the program has a logical clock associated with it.

• The ordering formula, ORD, imposes an order on all statements in the program.

• Intuitively, we can see this formula represents the composition of the sequential behavior of each thread.

• The trace formula represents in the individual behaviors of each thread while the ordering formula imposes an order across all threads.

• If this is not so clear right now, after the background section I will go over a more concrete example.
Concurrent Bounded Model Checking

- $\phi = TF_1 \land TF_2 \land \cdots \land TF_n \land ORD$
- Trace formula: thread semantics

- Bounded model checking of concurrent programs is a natural extension to the previously described method.
- Again, a single logical formula represents all possible behaviors of the program.
- The logical representation consists of two parts: first, there is a trace formula for each thread.
- The trace formula represents the possible behaviors of each thread in the program.
- Each statement in the program has a logical clock associated with it.
- The ordering formula, ORD, imposes an order on all statements in the program.
- Intuitively, we can see this formula represents the composition of the sequential behavior of each thread.
- The trace formula represents in the individual behaviors of each thread while the ordering formula imposes an order across all threads.
- If this is not so clear right now, after the background section I will go over a more concrete example.
Bounded model checking of concurrent programs is a natural extension to the previously described method.

Again, a single logical formula represents all possible behaviors of the program.

The logical representation consists of two parts: first, there is a trace formula for each thread.

The trace formula represents the possible behaviors of each thread in the program.

Each statement in the program has a logical clock associated with it.

The ordering formula, ORD, imposes an order on all statements in the program.

Intuitively, we can see this formula represents the composition of the sequential behavior of each thread.

The trace formula represents in the individual behaviors of each thread while the ordering formula imposes an order across all threads.

If this is not so clear right now, after the background section I will go over a more concrete example.

\[ \phi = TF_1 \land TF_2 \land \cdots \land TF_n \land ORD \]

- Trace formula: thread semantics
- Ordering: ordering of statements in the program
Overview

Introduction

Background
- Bounded Model Checking
- Partial Max-SAT

ConcBugAssist
- Fault Localization
- Fault Repair

Experimental Results

Conclusion

Next, I'll go over the partial Max-SAT problem
Partial Max-SAT

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses them self are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, c1 and c2
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is a satisfied if all of the clauses can be satisfied
- The partial max-SAT problems aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable sub-formula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula F with a minimal unsatisfiable subformla M then F without M is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses

> CNF Formula: $(x_1 \lor \neg x_2) \land (x_2 \lor x_3)$
Partial Max-SAT

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses themselves are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, c1 and c2
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is satisfied if all of the clauses can be satisfied
- The partial max-SAT problem aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable subformula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula F with a minimal unsatisfiable subformula M then F without M is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses

- CNF Formula: \((x_1 \lor \neg x_2) \land (x_2 \lor x_3)\)
- \(c_1 \land c_2\)
Partial Max-SAT

- CNF Formula: \((x_1 \lor \neg x_2) \land (x_2 \lor x_3)\)
  - \(c_1 \land c_2\)
    - \(x_1 \lor \neg x_2\)

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses them self are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, \(c_1\) and \(c_2\)
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is a satisfied if all of the clauses can be satisfied
- The partial max-SAT problems aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable sub-formula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula \(F\) with a minimal unsatisfiable subformla \(M\) then \(F\) without \(M\) is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses
A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form.

In CNF, the formula is a conjunction of clauses where the clauses themselves are a disjunction of literals.

For example, consider this formula which is in CNF form. It is the conjunction of two clauses, \( c_1 \) and \( c_2 \).

Each clause in the formula is a disjunction of literals.

A clause can be satisfied by setting one of its literals as true.

Then, the entire formula is a satisfied if all of the clauses can be satisfied.

The partial max-SAT problems aim to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true.

For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied.

However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true.

A related problem is the minimally unsatisfiable sub-formula.

Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable.

In other words, if there is a set of clauses representing an unsatisfiable formula \( F \) with a minimal unsatisfiable subformula \( M \) then \( F \) without \( M \) is satisfiable.

The problem can be further extended by designating clauses as either hard or soft.

For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses.
Partial Max-SAT

- **CNF Formula:** \((x_1 \lor \neg x_2) \land (x_2 \lor x_3)\)
- \(c_1 \land c_2\)
  - \(x_1 \lor \neg x_2\)
  - \(x_2 \lor x_3\)
- Partial Max-SAT: satisfy the maximum number of clauses

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses them self are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, \(c_1\) and \(c_2\)
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is a satisfied if all of the clauses can be satisfied
- The partial max-SAT problems aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable sub-formula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula \(F\) with a minimal unsatisfiable subformla \(M\) then \(F\) without \(M\) is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses
Partial Max-SAT

- CNF Formula: \((x_1 \lor \neg x_2) \land (x_2 \lor x_3)\)

- \(c_1 \land c_2\)
  - \(x_1 \lor \neg x_2\)
  - \(x_2 \lor x_3\)

- Partial Max-SAT: satisfy the maximum number of clauses

- Minimally unsatisfiable sub-formula

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses them self are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, \(c_1\) and \(c_2\)
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is a satisfied if all of the clauses can be satisfied
- The partial max-SAT problems aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable sub-formula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula \(F\) with a minimal unsatisfiable subformula \(M\) then \(F\) without \(M\) is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses
Partial Max-SAT

- CNF Formula: \((x_1 \lor \neg x_2) \land (x_2 \lor x_3)\)
- \(c_1 \land c_2\)
  - \(x_1 \lor \neg x_2\)
  - \(x_2 \lor x_3\)
- Partial Max-SAT: satisfy the maximum number of clauses
- Minimally unsatisfiable sub-formula
- Set of clauses: \(F\)

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses them self are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, \(c_1\) and \(c_2\)
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is a satisfied if all of the clauses can be satisfied
- The partial max-SAT problems aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable sub-formula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula \(F\) with a minimal unsatisfiable subformula \(M\) then \(F\) without \(M\) is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses
Partial Max-SAT

- CNF Formula: \((x_1 \lor \neg x_2) \land (x_2 \lor x_3)\)
- \(c_1 \land c_2\)
  - \(x_1 \lor \neg x_2\)
  - \(x_2 \lor x_3\)

- Partial Max-SAT: satisfy the maximum number of clauses
- Minimally unsatisfiable sub-formula
- Set of clauses: \(F\)
- MUS: \(M\)

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses them self are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, \(c_1\) and \(c_2\)
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is a satisfied if all of the clauses can be satisfied
- The partial max-SAT problems aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable sub-formula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula \(F\) with a minimal unsatisfiable subformla \(M\) then \(F\) without \(M\) is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses
Partial Max-SAT

- **CNF Formula**: \((x_1 \lor \neg x_2) \land (x_2 \lor x_3)\)

- **c_1 \land c_2**
  - \(x_1 \lor \neg x_2\)
  - \(x_2 \lor x_3\)

- **Partial Max-SAT**: satisfy the maximum number of clauses
- **Minimally unsatisfiable sub-formula**
- **Set of clauses**: \(F\)
- **MUS**: \(M\)
- **\(F \setminus M\) is satisfiable**

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses them self are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, \(c_1\) and \(c_2\)
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is a satisfied if all of the clauses can be satisfied
- The partial max-SAT problems aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable sub-formula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula \(F\) with a minimal unsatisfiable subformula \(M\) then \(F\) without \(M\) is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses
Partial Max-SAT

- CNF Formula: \((x_1 \lor \neg x_2) \land (x_2 \lor x_3)\)
- \(c_1 \land c_2\)
  - \(x_1 \lor \neg x_2\)
  - \(x_2 \lor x_3\)
- Partial Max-SAT: satisfy the maximum number of clauses
- Minimally unsatisfiable sub-formula
- Set of clauses: \(F\)
- MUS: \(M\)
- \(F \setminus M\) is satisfiable
- Hard clauses: must be satisfied

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses them self are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, \(c_1\) and \(c_2\)
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is a satisfied if all of the clauses can be satisfied
- The partial max-SAT problems aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable sub-formula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula \(F\) with a minimal unsatisfiable subformula \(M\) then \(F\) without \(M\) is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses
Partial Max-SAT

- CNF Formula: \((x_1 \lor \neg x_2) \land (x_2 \lor x_3)\)
- \(c_1 \land c_2\)
  - \(x_1 \lor \neg x_2\)
  - \(x_2 \lor x_3\)
- Partial Max-SAT: satisfy the maximum number of clauses
- Minimally unsatisfiable sub-formula
- Set of clauses: \(F\)
- MUS: \(M\)
- \(F \setminus M\) is satisfiable
- Hard clauses: must be satisfied
- Maximize satisfied *soft* clauses

- A formula used during bounded model checking can be represented in CNF, or, conjunctive normal form
- In CNF, the formula is a conjunction of clauses where the clauses them self are a disjunction of literals
- For example, consider this formula which is in CNF form. It is the conjunction of two clauses, \(c_1\) and \(c_2\)
- Each clause in the formula is a disjunction of literals
- A clause can be satisfied by setting one of its literals as true
- Then, the entire formula is a satisfied if all of the clauses can be satisfied
- The partial max-SAT problems aims to find the valuation of all literals in the formula such that the maximum number of clauses evaluate to true
- For a satisfiable formula, the partial max-sat problem is the same as the satisfiability problem: all of the clauses will be satisfied
- However, for an UNSAT formula, the partial max-sat solution finds the maximum number of clauses which can simultaneously evaluate to true
- A related problem is the minimally unsatisfiable sub-formula.
- Instead of finding the maximum number of clauses to evaluate to true, the goal is to find the minimal set of clauses in the formula such that if they were removed from the formula it would become satisfiable
- In other words, if there is a set of clauses representing an unsatisfiable formula \(F\) with a minimal unsatisfiable sub-formula \(M\) then \(F\) without \(M\) is satisfiable
- The problem can be further extended by designating clauses as either hard or soft
- For an unsatisfiable formula, the goal is to satisfy all the hard clauses and then maximize the number of satisfied soft clauses
A high level overview of our approach can be seen in this block diagram.

- We take as input a program with an associated property to be tested
- We pass the program and property to a bounded model checker
- If the bounded model checker finds an error, then it returns an erroneous thread schedule causing the error to occur
- By nature, this thread schedule is a total order over all statements within the program
- In other words, the erroneous schedule specifies an order for every statement in the program; obviously, for even moderate sized programs, this output is very large
- We then take this erroneous schedule and localize the fault using a partial max-sat solver
- We feed the negation of the localized fault back to the bounded model checker and then re-run the procedure
- By inserting the localized fault back into the model checker, we are asking it to find a violation of the property which is different than the one just reported
- We then repeat this process until the model checker finds no more violations
- The end set of localized faults found on each iteration is the set of root causes for violations of the property
- This can be output to the user or fed to the repair procedure
- The repair procedure then takes the localized faults, computes a binate cover, and outputs a newly repaired program without any errors
Next, I’ll go over the fault localization procedure of ConcBugAssist.
• The goal of our fault localization procedure is to create what we refer to as phi-sub-delta
• This is the set containing all the minimal inter-thread ordering constraints leading to an assertion violation.
• This set of minimal constraints can be either directly presented to the user or passed to our automated repair procedure which I will describe later
• The first step of our procedure is to pass the program under test, P, with some assertion p to the bounded model checker
• The output of the bounded model checker, assuming it finds a bug, is a concrete input to the program, phi-sub-in, and a concrete thread schedule, phi-sub-sch, causing the property to be violated
• We then generate a new formula to pass to the partial max-SAT solver. The formula consists of phi, the program representation, p, the property being verified, and the concrete input and concrete thread schedule leading to the violation
• Notice this formula is definitely unsatisfiable: the formula is asserting that using the previously found buggy program input and thread schedule when used in the program causes the property p to hold.
• We set the program representation, property, and concrete input as hard constraints and the thread schedule as soft constraints
• Recall that the hard constraints must be satisfied by the partial max-SAT solver while the number of satisfied soft constraints must be maximized
• We then compute the minimally unsatisfiable sub formula within the thread schedule using the partial max-sat solver
• Intuitively, the minimally unsatisfiable sub-formula is the subset of the concrete thread-schedule causing the property to be violated; in other words, it is the minimal root-cause of the violation
• We then add this minimal root cause to phi-delta to record the localized fault
• Then, we conjoin the negation of phi-delta with phi and repeat the BMC procedure.
• Intuitively, by negating phi-delta and re-running the BMC, we asked the model checker to find a violation of the property which is not caused by any thread schedule we’ve seen so far.
• We repeat this process until no violations are found

Fault Localization

▶ \( \phi_{\Delta} \): set of minimal inter-thread orderings causing a violation
Fault Localization

- \( \phi_{\Delta} \): set of minimal inter-thread orderings causing a violation
- BMC input: \( P \) with assertion \( p \)

- The goal of our fault localization procedure is to create what we refer to as phi-sub-delta
- This is the set containing all the minimal inter-thread ordering constraints leading to an assertion violation.
- This set of minimal constraints can be either directly presented to the user or passed to our automated repair procedure which I will describe later
- The first step of our procedure is to pass the program under test, \( P \), with some assertion \( p \) to the bounded model checker
- The output of the bounded model checker, assuming it finds a bug, is a concrete input to the program, phi-sub-in, and a concrete thread schedule, phi-sub-sch, causing the property to be violated
- We then generate a new formula to pass to the partial max-SAT solver. The formula consists of phi, the program representation, \( p \), the property being verified, and the concrete input and concrete thread schedule leading to the violation
- Notice this formula is definitely unsatisfiable: the formula is asserting that using the previously found buggy program input and thread schedule when used in the program causes the property \( p \) to hold.
- We set the program representation, property, and concrete input as hard constraints and the thread schedule as soft constraints
- Recall that the hard constraints must be satisfied by the partial max-SAT solver while the number of satisfied soft constraints must be maximized
- We then compute the minimally unsatisfiable sub formula within the thread schedule using the partial max-sat solver
- Intuitively, the minimally unsatisfiable sub-formula is the subset of the concrete thread-schedule causing the property to be violated, in other words, it is the minimal root-cause of the violation
- We then add this minimal root cause to phi-delta to record the localized fault
- Then, we conjoin the negation of phi-delta with phi and repeat the BMC procedure.
- Intuitively, by negating phi-delta and re-running the BMC, we asked the model checker to find a violation of the property which is not caused by any thread schedule we’ve seen so far.
- We repeat this process until no violations are found
Fault Localization

- $\phi_\Delta$: set of minimal inter-thread orderings causing a violation
- BMC input: $P$ with assertion $p$
- BMC output: $(\phi_{in}, \phi_{sch})$

- The goal of our fault localization procedure is to create what we refer to as phi-sub-delta
- This is the set containing all the minimal inter-thread ordering constraints leading to an assertion violation.
- This set of minimal constraints can be either directly presented to the user or passed to our automated repair procedure which I will describe later
- The first step of our procedure is to pass the program under test, $P$, with some assertion $p$ to the bounded model checker
- The output of the bounded model checker, assuming it finds a bug, is a concrete input to the program, phi-sub-in, and a concrete thread schedule, phi-sub-sch, causing the property to be violated
- We then generate a new formula to pass to the partial max-SAT solver. The formula consists of phi, the program representation, property, the property being verified, and the concrete input and concrete thread schedule leading to the violation
- Notice this formula is definitely unsatisfiable: the formula is asserting that using the previously found buggy program input and thread schedule when used in the program causes the property $p$ to hold
- We set the program representation, property, and concrete input as hard constraints and the thread schedule as soft constraints
- Recall that the hard constraints must be satisfied by the partial max-SAT solver while the number of satisfied soft constraints must be maximized
- We then compute the minimally unsatisfiable sub formula within the thread schedule using the partial max-sat solver
- Intuitively, the minimally unsatisfiable sub-formula is the subset of the concrete thread-schedule causing the property to be violated, in other words, it is the minimal root-cause of the violation
- We then add this minimal root cause to phi-delta to record the localized fault
- Then, we conjoin the negation of phi-delta with phi and repeat the BMC procedure.
- Intuitively, by negating phi-delta and re-running the BMC, we asked the model checker to find a violation of the property which is not caused by any thread schedule we've seen so far.
- We repeat this process until no violations are found
The goal of our fault localization procedure is to create what we refer to as phi-delta
This is the set containing all the minimal inter-thread ordering constraints leading to an assertion violation.
This set of minimal constraints can be either directly presented to the user or passed to our automated
repair procedure which I will describe later
The first step of our procedure is to pass the program under test, P, with some assertion p to the bounded
model checker
The output of the bounded model checker, assuming it finds a bug, is a concrete input to the program,
phi-in, and a concrete thread schedule, phi-sch, causing the property to be violated
We then generate a new formula to pass to the partial max-SAT solver. The formula consists of phi, the
program representation, p, the property being verified, and the concrete input and concrete thread
schedule leading to the violation
Notice this formula is definitely unsatisfiable: the formula is asserting that using the previously found
buggy program input and thread schedule when used in the program causes the property p to hold.
We set the program representation, property, and concrete input as hard constraints and the thread
schedule as soft constraints
Recall that the hard constraints must be satisfied by the partial max-SAT solver while the number of
satisfied soft constraints must be maximized
We then compute the minimally unsatisfiable sub formula within the thread schedule using the partial
max-sat solver
Intuitively, the minimally unsatisfiable sub-formula is the subset of the concrete thread-schedule causing
the property to be violated, in other words, it is the minimal root-cause of the violation
We then add this minimal root cause to phi-delta to record the localized fault
Then, we conjoin the negation of phi-delta with phi and repeat the BMC procedure.
Intuitively, by negating phi-delta and re-running the BMC, we asked the model checker to find a violation
of the property which is not caused by any thread schedule we’ve seen so far.
We repeat this process until no violations are found

Fault Localization

- \( \phi_\Delta \): set of minimal inter-thread orderings causing a violation
- BMC input: \( P \) with assertion \( p \)
- BMC output: \( (\phi_{in}, \phi_{sch}) \)
- Partial Max-SAT constraints: \( \phi \land p \land \phi_{in} \land \phi_{sch} \)
Fault Localization

- $\phi_\Delta$: set of minimal inter-thread orderings causing a violation
- BMC input: $P$ with assertion $p$
- BMC output: $(\phi_{in}, \phi_{sch})$
- Partial Max-SAT constraints: $\phi \land p \land \phi_{in} \land \phi_{sch}$
- Hard constraints: $\phi, p, \phi_{in}$

- The goal of our fault localization procedure is to create what we refer to as $\phi$-sub-$\Delta$
- This is the set containing all the minimal inter-thread ordering constraints leading to an assertion violation.
- This set of minimal constraints can be either directly presented to the user or passed to our automated repair procedure which I will describe later.
- The first step of our procedure is to pass the program under test, $P$, with some assertion $p$ to the bounded model checker.
- The output of the bounded model checker, assuming it finds a bug, is a concrete input to the program, $\phi$-sub-$\text{in}$, and a concrete thread schedule, $\phi$-sub-$\text{sch}$, causing the property to be violated.
- We then generate a new formula to pass to the partial max-SAT solver. The formula consists of $\phi$, the program representation, $p$, the property being verified, and the concrete input and concrete thread schedule leading to the violation.
- Notice this formula is definitely unsatisfiable: the formula is asserting that using the previously found buggy program input and thread schedule when used in the program causes the property $p$ to hold.
- We set the program representation, property, and concrete input as hard constraints and the thread schedule as soft constraints.
- Recall that the hard constraints must be satisfied by the partial max-SAT solver while the number of satisfied soft constraints must be maximized.
- We then compute the minimally unsatisfiable sub formula within the thread schedule using the partial max-sat solver.
- Intuitively, the minimally unsatisfiable sub-formula is the subset of the concrete thread-schedule causing the property to be violated, in other words, it is the minimal root-cause of the violation.
- We then add this minimal root cause to $\phi$-delta to record the localized fault.
- Then, we conjoin the negation of $\phi$-delta with $\phi$ and repeat the BMC procedure.
- Intuitively, by negating $\phi$-delta and re-running the BMC, we asked the model checker to find a violation of the property which is not caused by any thread schedule we’ve seen so far.
- We repeat this process until no violations are found.
Fault Localization

- $\phi_\Delta$: set of minimal inter-thread orderings causing a violation
- BMC input: $P$ with assertion $p$
- BMC output: $(\phi_{in}, \phi_{sch})$
- Partial Max-SAT constraints: $\phi \land p \land \phi_{in} \land \phi_{sch}$
- Hard constraints: $\phi$, $p$, $\phi_{in}$
- Soft constraints: $\phi_{sch}$

- The goal of our fault localization procedure is to create what we refer to as $\phi_{-}\Delta$
- This is the set containing all the minimal inter-thread ordering constraints leading to an assertion violation.
- This set of minimal constraints can be either directly presented to the user or passed to our automated repair procedure which I will describe later
- The first step of our procedure is to pass the program under test, $P$, with some assertion $p$ to the bounded model checker
- The output of the bounded model checker, assuming it finds a bug, is a concrete input to the program, $\phi_{-}in$, and a concrete thread schedule, $\phi_{-}sch$, causing the property to be violated
- We then generate a new formula to pass to the partial max-SAT solver. The formula consists of $\phi$, the program representation, $p$, the property being verified, and the concrete input and concrete thread schedule leading to the violation
- Notice this formula is definitely unsatisfiable: the formula is asserting that using the previously found buggy program input and thread schedule when used in the program causes the property $p$ to hold.
- We set the program representation, property, and concrete input as hard constraints and the thread schedule as soft constraints
- Recall that the hard constraints must be satisfied by the partial max-SAT solver while the number of satisfied soft constraints must be maximized
- We then compute the minimally unsatisfiable sub formula within the thread schedule using the partial max-sat solver
- Intuitively, the minimally unsatisfiable sub-formula is the subset of the concrete thread-schedule causing the property to be violated, in other words, it is the minimal root-cause of the violation
- We then add this minimal root cause to $\phi_{-}\Delta$ to record the localized fault
- Then, we conjoin the negation of $\phi_{-}\Delta$ with $\phi$ and repeat the BMC procedure.
- Intuitively, by negating $\phi_{-}\Delta$ and re-running the BMC, we asked the model checker to find a violation of the property which is not caused by any thread schedule we’ve seen so far.
- We repeat this process until no violations are found
Fault Localization

- \( \phi_{\Delta} \): set of minimal inter-thread orderings causing a violation
- BMC input: \( P \) with assertion \( p \)
- BMC output: \( (\phi_{in}, \phi_{sch}) \)
- Partial Max-SAT constraints: \( \phi \land p \land \phi_{in} \land \phi_{sch} \)
- Hard constraints: \( \phi, p, \phi_{in} \)
- Soft constraints: \( \phi_{sch} \)
- \( \phi_{\Delta} = \phi_{\Delta} \cup \text{MUS}(\phi_{sch}) \)

- The goal of our fault localization procedure is to create what we refer to as \( \phi_{\Delta} \)
- This is the set containing all the minimal inter-thread ordering constraints leading to an assertion violation.
- This set of minimal constraints can be either directly presented to the user or passed to our automated repair procedure which I will describe later.
- The first step of our procedure is to pass the program under test, \( P \), with some assertion \( p \) to the bounded model checker.
- The output of the bounded model checker, assuming it finds a bug, is a concrete input to the program, \( \phi_{\text{in}} \), and a concrete thread schedule, \( \phi_{\text{sch}} \), causing the property to be violated.
- We then generate a new formula to pass to the partial max-SAT solver. The formula consists of \( \phi \), the program representation, \( p \), the property being verified, and the concrete input and concrete thread schedule leading to the violation.
- Notice this formula is definitely unsatisfiable: the formula is asserting that using the previously found buggy program input and thread schedule when used in the program causes the property \( p \) to hold.
- We set the program representation, property, and concrete input as hard constraints and the thread schedule as soft constraints.
- Recall that the hard constraints must be satisfied by the partial max-SAT solver while the number of satisfied soft constraints must be maximized.
- We then compute the minimally unsatisfiable sub-formula within the thread schedule using the partial max-sat solver.
- Intuitively, the minimally unsatisfiable sub-formula is the subset of the concrete thread-schedule causing the property to be violated, in other words, it is the minimal root-cause of the violation.
- We then add this minimal root cause to \( \phi_{\Delta} \) to record the localized fault.
- Then, we conjoin the negation of \( \phi_{\Delta} \) with \( \phi \) and repeat the BMC procedure.
- Intuitively, by negating \( \phi_{\Delta} \) and re-running the BMC, we asked the model checker to find a violation of the property which is not caused by any thread schedule we’ve seen so far.
- We repeat this process until no violations are found.
Fault Localization

- $\phi_D$: set of minimal inter-thread orderings causing a violation
- BMC input: $P$ with assertion $p$
- BMC output: $(\phi_{in}, \phi_{sch})$
- Partial Max-SAT constraints: $\phi \land p \land \phi_{in} \land \phi_{sch}$
- Hard constraints: $\phi, p, \phi_{in}$
- Soft constraints: $\phi_{sch}$
- $\phi_D = \phi_D \cup \text{MUS}(\phi_{sch})$
- $\phi = \phi \land \neg \phi_D$

- The goal of our fault localization procedure is to create what we refer to as phi-sub-delta
- This is the set containing all the minimal inter-thread ordering constraints leading to an assertion violation.
- This set of minimal constraints can be either directly presented to the user or passed to our automated repair procedure which I will describe later
- The first step of our procedure is to pass the program under test, $P$, with some assertion $p$ to the bounded model checker
- The output of the bounded model checker, assuming it finds a bug, is a concrete input to the program, phi-sub-in, and a concrete thread schedule, phi-sub-sch, causing the property to be violated
- We then generate a new formula to pass to the partial max-SAT solver. The formula consists of phi, the program representation, p, the property being verified, and the concrete input and concrete thread schedule leading to the violation
- Notice this formula is definitely unsatisfiable: the formula is asserting that using the previously found buggy program input and thread schedule when used in the program causes the property $p$ to hold.
- We set the program representation, property, and concrete input as hard constraints and the thread schedule as soft constraints
- Recall that the hard constraints must be satisfied by the partial max-SAT solver while the number of satisfied soft constraints must be maximized
- We then compute the minimally unsatisfiable sub formula within the thread schedule using the partial max-sat solver
- Intuitively, the minimally unsatisfiable sub-formula is the subset of the concrete thread-schedule causing the property to be violated, in other words, it is the minimal root-cause of the violation
- We then add this minimal root cause to phi-delta to record the localized fault
- Then, we conjoin the negation of phi-delta with phi and repeat the BMC procedure.
- Intuitively, by negating phi-delta and re-running the BMC, we asked the model checker to find a violation of the property which is not caused by any thread schedule we’ve seen so far.
- We repeat this process until no violations are found
Fault Localization

- $\phi_\Delta$: set of minimal inter-thread orderings causing a violation
- BMC input: $P$ with assertion $p$
- BMC output: $(\phi_{in}, \phi_{sch})$
- Partial Max-SAT constraints: $\phi \land p \land \phi_{in} \land \phi_{sch}$
- Hard constraints: $\phi, p, \phi_{in}$
- Soft constraints: $\phi_{sch}$
  - $\phi_\Delta = \phi_\Delta \cup \text{MUS}(\phi_{sch})$
  - $\phi = \phi \land \neg \phi_\Delta$
- Repeat until no bug is found

- The goal of our fault localization procedure is to create what we refer to as $\phi_{sub-delta}$
- This is the set containing all the minimal inter-thread ordering constraints leading to an assertion violation.
- This set of minimal constraints can be either directly presented to the user or passed to our automated repair procedure which I will describe later.
- The first step of our procedure is to pass the program under test, $P$, with some assertion $p$ to the bounded model checker.
- The output of the bounded model checker, assuming it finds a bug, is a concrete input to the program, $\phi_{sub-in}$, and a concrete thread schedule, $\phi_{sub-sch}$, causing the property to be violated.
- We then generate a new formula to pass to the partial max-SAT solver. The formula consists of $\phi$, the program representation, $p$, the property being verified, and the concrete input and concrete thread schedule leading to the violation.
- Notice this formula is definitely unsatisfiable: the formula is asserting that using the previously found buggy program input and thread schedule when used in the program causes the property $p$ to hold.
- We set the program representation, property, and concrete input as hard constraints and the thread schedule as soft constraints.
- Recall that the hard constraints must be satisfied by the partial max-SAT solver while the number of satisfied soft constraints must be maximized.
- We then compute the minimally unsatisfiable sub formula within the thread schedule using the partial max-sat solver.
- Intuitively, the minimally unsatisfiable sub-formula is the subset of the concrete thread-schedule causing the property to be violated, in other words, it is the minimal root-cause of the violation.
- We then add this minimal root cause to $\phi_{delta}$ to record the localized fault.
- Then, we conjoin the negation of $\phi_{delta}$ with $\phi$ and repeat the BMC procedure.
- Intuitively, by negating $\phi_{delta}$ and re-running the BMC, we asked the model checker to find a violation of the property which is not caused by any thread schedule we’ve seen so far.
- We repeat this process until no violations are found.
Overview

Introduction

Background
  Bounded Model Checking
  Partial Max-SAT

ConcBugAssist
  Fault Localization
  Fault Repair

Experimental Results

Conclusion

Next, I’ll go over an example of our fault localization procedure as well as our fault repair procedure
int x = 0;
int y = 0;

void f1(void) {
    x = 0;
    y = 0;
}

void f2(void) {
    x = 1;
    y = 1;
}

int main () {
    pthread_t t1, t2;
    thread_create(t1, f1);
    thread_create(t2, f2);

    thread_join(t1);
    thread_join(t2);
    assert(x == y);
    return 0;
}

- Consider the following simple example program
- The program is a simplification of a classic two variable atomicity violation
- The first thread non-atomically sets both $x$ and $y$ to zero
- The second thread non-atomically sets both $x$ and $y$ to one
- The property being checked is that the value of $x$ and $y$ are equal, or, that the updates from the two threads are atomic
• Consider the following simple example program
• The program is a simplification of a classic two variable atomicity violation
• The first thread non-atomically sets both $x$ and $y$ to zero
• The second thread non-atomically sets both $x$ and $y$ to one
• The property being checked is that the value of $x$ and $y$ are equal, or, that the updates from the two threads are atomic
Consider the following simple example program:

The program is a simplification of a classic two variable atomicity violation:

- The first thread non-atomically sets both $x$ and $y$ to zero
- The second thread non-atomically sets both $x$ and $y$ to one
- The property being checked is that the value of $x$ and $y$ are equal, or, that the updates from the two threads are atomic
int x = 0;
int y = 0;

void f1( void ) {
    x = 0;
    y = 0;
}

void f2( void ) {
    x = 1;
    y = 1;
}

int main () {
    pthread_t t1, t2;
    thread_create (t1, f1);
    thread_create (t2, f2);
    thread_join (t1);
    thread_join (t2);
    assert (x == y);
    return 0;
}

• Consider the following simple example program
• The program is a simplification of a classic two variable atomicity violation
• The first thread non-atomically sets both x and y to zero
• The second thread non-atomically sets both x and y to one
• The property being checked is that the value of x and y are equal, or, that the updates from the two threads are atomic
Here are the results of our fault-localization procedure for the previous program:

- The first buggy thread schedule results in $x$ having a value of one and $y$ having a value of zero.
- This can occur if after thread one writes zero, it is interrupted by thread two who updates $x$ and $y$, followed by thread one setting $y$ to zero.
- The other way the bug can manifest is the reverse scenario where thread two executes before thread one.
- Just to reiterate, you can see how our fault localization procedure returns concise results: other parts of the thread schedule, such as the statements in the main thread, are not included making it easy to understand.
• Here are the results of our fault-localization procedure for the previous program
• The first buggy thread schedule results in $x$ having a value of one and $y$ having a value of zero
• This can occur if after thread one writes zero, it is interrupted by thread two who updates $x$ and $y$, followed by thread one setting $y$ to zero
• The other way the bug can manifest is the reverse scenario where thread two executes before thread one
• Just to reiterate, you can see how our fault localization procedure returns concise results: other parts of the thread schedule, such as the statements in the main thread, are not included making it easy to understand
The intuition behind our automated repair analysis is to insert happens-before constraints into the program such that the buggy interleaving is avoided.

By happens-before constraint, we mean a constraint between two inter-thread statements enforcing that statement one always occurs, on all executions, before statement two.

For example, consider the first buggy interleaving from the previous slide.

One way to block the interleaving would be to insert a happens-before constraint from statement $e_6$ in thread one to statement $e_{10}$ in thread two.

Enforcing this constraint in the program would result in thread one occurring before thread two in all executions.

This constraint blocks the buggy interleaving because it introduces a cycle into the schedule. The cycle, intuitively, is like a proof-by-contradiction proving the buggy interleaving cannot occur.

Similarly, there are many such cycle-inducing happens-before constraints which can be added to this thread schedule.

Also, there are several to be introduced for the other buggy interleaving.

Given the set of constraints to block each individual buggy schedule, our repair procedure then needs to select a subset of these constraints to both block the schedules as well as not introduce cycles into the program.
The intuition behind our automated repair analysis is to insert happens-before constraints into the program such that the buggy interleaving is avoided.

By happens-before constraint, we mean a constraint between two inter-thread statements enforcing that statement one always occurs, on all executions, before statement two.

For example, consider the first buggy interleaving from the previous slide.

One way to block the interleaving would be to insert a happens-before constraint from statement e6 in thread one to statement e10 in thread two.

Enforcing this constraint in the program would result in thread one occurring before thread two in all executions.

This constraint blocks the buggy interleaving because it introduces a cycle into the schedule. The cycle, intuitively, is like a proof-by-contradiction proving the buggy interleaving cannot occur.

Similarly, there are many such cycle-inducing happens-before constraints which can be added to this thread schedule.

Also, there are several to be introduced for the other buggy interleaving.
The intuition behind our automated repair analysis is to insert happens-before constraints into the program such that the buggy interleaving is avoided.

By happens-before constraint, we mean a constraint between two inter-thread statements enforcing that statement one always occurs, on all executions, before statement two.

For example, consider the first buggy interleaving from the previous slide:

One way to block the interleaving would be to insert a happens-before constraint from statement $e_6$ in thread one to statement $e_{10}$ in thread two.

Enforcing this constraint in the program would result in thread one occurring before thread two in all executions.

This constraint blocks the buggy interleaving because it introduces a cycle into the schedule. The cycle, intuitively, is like a proof-by-contradiction proving the buggy interleaving cannot occur.

Similarly, there are many such cycle-inducing happens-before constraints which can be added to this thread schedule.

Also, there are several to be introduced for the other buggy interleaving.

Given the set of constraints to block each individual buggy schedule, our repair procedure then needs to select a subset of these constraints to both block the schedules as well as not introduce cycles into the program.
The intuition behind our automated repair analysis is to insert happens-before constraints into the program such that the buggy interleaving is avoided.

By happens-before constraint, we mean a constraint between two inter-thread statements enforcing that statement one always occurs, on all executions, before statement two.

For example, consider the first buggy interleaving from the previous slide.

One way to block the interleaving would be to insert a happens-before constraint from statement e6 in thread one to statement e10 in thread two.

Enforcing this constraint in the program would result in thread one occurring before thread two in all executions.

This constraint blocks the buggy interleaving because it introduces a cycle into the schedule. The cycle, intuitively, is like a proof-by-contradiction proving the buggy interleaving cannot occur.

Similarly, there are many such cycle-inducing happens-before constraints which can be added to this thread schedule.

Also, there are several to be introduced for the other buggy interleaving.
**Kill-sets**

- $KS_1 = (e_{10} \rightarrow_s e_5) \lor (e_{11} \rightarrow_s e_5) \lor (e_6 \rightarrow_s e_{10}) \lor (e_6 \rightarrow_s e_{11})$
- $KS_2 = (e_5 \rightarrow_s e_{10}) \lor (e_6 \rightarrow_s e_{10}) \lor (e_{11} \rightarrow_s e_5) \lor (e_{11} \rightarrow_s e_6)$

- For each thread schedule, we create what we refer to as the kill-set for the schedule
- The kill set contains all the constraints which, when enforced in the program, would block each individual erroneous schedule
- These are the dashed lines from the previous graphs
- The second part is to find compatible constraints we refer to as omega
- Omega encodes constraints which cannot simultaneously occur in the program because they would result in a cycle, or deadlock, in the program
- For example, $e_{10}$ happening before $e_5$ would satisfy kill-set one and $e_5$ before $e_{10}$ would satisfy kill-set two.
- However, together, these two constraints introduce a cycle into the program which results in a deadlock
- Finally, all the solutions to the equation $ks_1$ and $ks_2$ and omega are considered as valid repairs
- For the sake of time, I’ll skip the algorithmic details but note that this problem while NP-Hard can be efficiently solved via branch-and-bound algorithms
Kill-sets

- $KS_1 = (e_{10} \rightarrow_s e_5) \lor (e_{11} \rightarrow_s e_5) \lor (e_6 \rightarrow_s e_{10}) \lor (e_6 \rightarrow_s e_{11})$
- $KS_2 = (e_5 \rightarrow_s e_{10}) \lor (e_6 \rightarrow_s e_{10}) \lor (e_{11} \rightarrow_s e_5) \lor (e_{11} \rightarrow_s e_6)$
- $\omega$: cross kill-set compatibility

- For each thread schedule, we create what we refer to as the kill-set for the schedule
- The kill set contains all the constraints which, when enforced in the program, would block each individual erroneous schedule
- These are the dashed lines from the previous graphs
- The second part is to find compatible constraints we refer to as $\omega$
- $\omega$ encodes constraints which cannot simultaneously occur in the program because they would result in a cycle, or deadlock, in the program
- For example, $e_{10}$ happening before $e_5$ would satisfy kill-set one and $e_5$ before $e_{10}$ would satisfy kill-set two.
- However, together, these two constraints introduce a cycle into the program which results in a deadlock
- Finally, all the solutions to the equation $ks_1$ and $ks_2$ and $\omega$ are considered as valid repairs
- For the sake of time, I’ll skip the algorithmic details but note that this problem while NP-Hard can be efficiently solved via branch-and-bound algorithms

Here, we can see there are 16 possible ways to satisfy $KS_1$ and $KS_2$. 
Kill-sets

- $\text{KS}_1 = (e_{10} \rightarrow s e_5) \lor (e_{11} \rightarrow s e_5) \lor (e_6 \rightarrow s e_{10}) \lor (e_6 \rightarrow s e_{11})$

- $\text{KS}_2 = (e_5 \rightarrow s e_{10}) \lor (e_6 \rightarrow s e_{10}) \lor (e_{11} \rightarrow s e_5) \lor (e_{11} \rightarrow s e_6)$

- $\omega$: cross kill-set compatibility
  - $(e_{10} \rightarrow s e_5) \land (e_5 \rightarrow s e_{10})$

- For each thread schedule, we create what we refer to as the kill-set for the schedule
- The kill set contains all the constraints which, when enforced in the program, would block each individual erroneous schedule
- These are the dashed lines from the previous graphs
- The second part is to find compatible constraints we refer to as $\omega$
- $\omega$ encodes constraints which cannot simultaneously occur in the program because they would result in a cycle, or deadlock, in the program
- For example, $e_{10}$ happening before $e_5$ would satisfy kill-set one and $e_5$ before $e_{10}$ would satisfy kill-set two.
- However, together, these two constraints introduce a cycle into the program which results in a deadlock
- Finally, all the solutions to the equation $\text{ks}_1$ and $\text{ks}_2$ and $\omega$ are considered as valid repairs
- For the sake of time, I’ll skip the algorithmic details but note that this problem while NP-Hard can be efficiently solved via branch-and-bound algorithms
Kill-sets

\[ KS_1 = (e_{10} \rightarrow_s e_5) \lor (e_{11} \rightarrow_s e_6) \lor (e_6 \rightarrow_s e_{10}) \lor (e_6 \rightarrow_s e_{11}) \]

\[ KS_2 = (e_5 \rightarrow_s e_{10}) \lor (e_6 \rightarrow_s e_{10}) \lor (e_{11} \rightarrow_s e_5) \lor (e_{11} \rightarrow_s e_6) \]

\[ \omega: \text{ cross kill-set compatibility} \]

\[ (e_{10} \rightarrow_s e_5) \land (e_5 \rightarrow_s e_{10}) \]

Satisfy: \( KS_1 \land KS_2 \land \omega \)

- For each thread schedule, we create what we refer to as the kill-set for the schedule
- The kill set contains all the constraints which, when enforced in the program, would block each individual erroneous schedule
- These are the dashed lines from the previous graphs
- The second part is to find compatible constraints we refer to as omega
- Omega encodes constraints which cannot simultaneously occur in the program because they would result in a cycle, or deadlock, in the program
- For example, \( e_{10} \) happening before \( e_5 \) would satisfy kill-set one and \( e_5 \) before \( e_{10} \) would satisfy kill-set two.
- However, together, these two constraints introduce a cycle into the program which results in a deadlock
- Finally, all the solutions to the equation \( ks_1 \) and \( ks_2 \) and omega are considered as valid repairs
- For the sake of time, I’ll skip the algorithmic details but note that this problem while NP-Hard can be efficiently solved via branch-and-bound algorithms
After checking the satisfiability of the previous formula, we are left with four which do not violate the constraints in the program.

As I eluded to earlier, these solutions, while fixing the bug, are often too restrictive.

For example, all four of these solutions essentially enforce a happens-before constraint causing either thread one to execute before thread two or vice versa.

What we are missing is the more permissive solution where either thread one executes before thread two or thread two executes before thread one.

One intuition of our method is that given these four elementary bug repairs, we can combine them to generate what we call composite solutions.

To do this, we construct graphs from pairs of non-contradictory elementary solutions and then check if the resulting graph forms a cycle from the elementary solutions between two threads.

As a concrete example, consider the combination of solutions (a) and (b).

Here we can see the two key points: first, the two solutions do not contradict each other; they can both simultaneously be enforced.

Additionally, the results create a cycle across the elementary edges between the two threads.

Intuitively, the fact a cycle exists across the elementary solutions implies that the two threads can be serialized in either order.

This matches the design pattern of a mutex lock.

So, in the end we present the four elementary repairs and the one composite repair to the user.
After checking the satisfiability of the previous formula, we are left with four which do not violate the constraints in the program.

As I eluded to earlier, these solutions, while fixing the bug, are often too restrictive.

For example, all four of these solutions essentially enforce a happens-before constraint causing either thread one to execute before thread two or vice versa.

What we are missing is the more permissive solution where either thread one executes before thread two or thread two executes before thread one.

One intuition of our method is that given these four elementary bug repairs, we can combine them to generate what we call composite solutions.

To do this, we construct graphs from pairs of non-contradictory elementary solutions and then check if the resulting graph forms a cycle from the elementary solutions between two threads.

As a concrete example, consider the combination of solutions (a) and (b).

Here we can see the two key points: first, the two solutions do not contradict each other; they can both simultaneously be enforced.

Additionally, the results create a cycle across the elementary edges between the two threads.

Intuitively, the fact a cycle exists across the elementary solutions implies that the two threads can be serialized in either order.

This matches the design pattern of a mutex lock.

So, in the end we present the four elementary repairs and the one composite repair to the user.
After checking the satisfiability of the previous formula, we are left with four which do not violate the constraints in the program.

As I eluded to earlier, these solutions, while fixing the bug, are often too restrictive.

For example, all four of these solutions essentially enforce a happens-before constraint causing either thread one to execute before thread two or vice versa.

What we are missing is the more permissive solution where either thread one executes before thread two or thread two executes before thread one.

One intuition of our method is that given these four elementary bug repairs, we can combine them to generate what we call composite solutions.

To do this, we construct graphs from pairs of non-contradictory elementary solutions and then check if the resulting graph forms a cycle from the elementary solutions between two threads.

As a concrete example, consider the combination of solutions (a) and (b).

Here we can see the two key points: first, the two solutions do not contradict each other; they can both simultaneously be enforced.

Additionally, the results create a cycle across the elementary edges between the two threads.

Intuitively, the fact a cycle exists across the elementary solutions implies that the two threads can be serialized in either order.

This matches the design pattern of a mutex lock.

So, in the end we present the four elementary repairs and the one composite repair to the user.
Next, I’ll present our experimental results
Experimental Results

- CBMC [Kroening and Tautschnig, 2014]
- MSUnCore [J. Marques-Silva]
- Repair implemented in Haskell
- 34 buggy concurrent programs

- We implemented ConcBugAssist as an extension to the bounded model checker CBMC
- To compute minimally unsatisfiable subformulas and localize errors we used MSUnCore
- We implemented the repair calculation solver ourselves in Haskell
- We tested ConcBugAssist on 34 buggy concurrent programs to perform fault localization and then repair
On average, our repair procedure, including the time to run the bounded model checker, took on average 16 seconds.

Our repair procedure on average took less than one second.

As I mentioned previously, the bounded model checker produces an entire concertized thread schedule for a detected bug.

Our localization procedure finds the minimal subset of the entire thread schedule.

On average, our localization result is 16 percent the size of the output from the bounded model checker, making it, hopefully, easier to understand.

For more detailed results, you can see our paper.

Finally, the scalability of our approach depends on the verification procedure.

In our case, we can repair programs suitable for bounded model checking.

We believe ideal applications would be small-to-medium programs with complex concurrency control such as device drivers and concurrent data structures.

Additionally, we believe scalability could be further increased using techniques such as predicate abstraction.
On average, our repair procedure, including the time to run the bounded model checker, took on average 16 seconds.

Our repair procedure on average took less than one second.

As I mentioned previously, the bounded model checker produces an entire concertized thread schedule for a detected bug.

Our localization procedure finds the minimal subset of the entire thread schedule.

On average, our localization result is 16 percent the size of the output from the bounded model checker, making it, hopefully, easier to understand.

For more detailed results, you can see our paper.

Finally, the scalability of our approach depends on the verification procedure.

In our case, we can repair programs suitable for bounded model checking.

We believe ideal applications would be small-to-medium programs with complex concurrency control such as device drivers and concurrent data structures.

Additionally, we believe scalability could be further increased using techniques such as predicate abstraction.
Result Summary

- Average localization runtime: 16 s
- Average repair runtime: <1 s
- Average localization reduction: %16

- On average, our repair procedure, including the time to run the bounded model checker, took on average 16 seconds
- Our repair procedure on average took less than one second
- As I mentioned previously, the bounded model checker produces an entire concertized thread schedule for a detected bug
- Our localization procedure finds the minimal subset of the entire thread schedule
- On average, our localization result is 16 percent the size of the output from the bounded model checker, making it, hopefully, easier to understand
- For more detailed results, you can see our paper
- Finally, the scalability of our approach depends on the verification procedure
- In our case, we can repair programs suitable for bounded model checking
- We believe ideal applications would be small-to-medium programs with complex concurrency control such as device drivers and concurrent data structures.
- Additionally, we believe scalability could be further increased using techniques such as predicate abstraction.
Result Summary

- Average localization runtime: 16 s
- Average repair runtime: <1 s
- Average localization reduction: %16
- Scalability depends on verification procedure

- On average, our repair procedure, including the time to run the bounded model checker, took on average 16 seconds
- Our repair procedure on average took less than one second
- As I mentioned previously, the bounded model checker produces an entire concertized thread schedule for a detected bug
- Our localization procedure finds the minimal subset of the entire thread schedule
- On average, our localization result is 16 percent the size of the output from the bounded model checker, making it, hopefully, easier to understand
- For more detailed results, you can see our paper
- Finally, the scalability of our approach depends on the verification procedure
- In our case, we can repair programs suitable for bounded model checking
- We believe ideal applications would be small-to-medium programs with complex concurrency control such as device drivers and concurrent data structures.
- Additionally, we believe scalability could be further increased using techniques such as predicate abstraction.
In conclusion, we present ConcBugAssist, a constraint based approach to concurrent fault localization and repair.

We fill the gap of prior work on using constraints to localize faults in sequential programs.

Additionally, we showed how we can, given the localization results, use more constraints to create both elementary and composite repairs.

With that, I'll take any questions.
• In conclusion, we present ConcBugAssist, a constraint based approach to concurrent fault localization and repair

• We fill the gap of prior work on using constraints to localize faults in sequential programs

• Additionally, we showed how we can, given the localization results, use more constraints to create both elementary and composite repairs

• With that, I’ll take any questions
Related Work

- Constraint based repair for sequential programs [Jose and Majumdar, 2011], [Ermis, Schäf, and Wies, 2012], [Murali et al., 2014], [Qi et al., 2012]
- Synthesis from Specification [Bloem et al., 2014], [T et al., 2010]
- Deadlock repair [Wang et al., 2008]
- Pattern based repair [Krena et al., Jin et al.]

- Our tool fits within the current ecosystem of automated program repairs tools.
- We were mostly inspired by previous automated repair tools using constraint solvers.
- However, much of the existing work only handles logical errors in sequential programs, our tool fills in the gap to handle concurrent programs.
- Our approach is similar to concurrent program synthesis from specifications except instead of a specification we start with a buggy program.
- Additionally, we focus on property violations and their repair as opposed to some prior work on deadlock repairs, which is complementary.
- Finally, we take an alternative approach compared to existing pattern based repair methods for concurrent programs.
- Instead of matching on dynamic behaviors we perform a more general method without having to use heuristics.
Our repair is only as accurate as the underlying verification engine

- One final point that I'd like to make is that our repair procedure is only as accurate as the underlying verification engine.
- For example, in CBMC we only analyzed property violations and not deadlock violations.
- As such, it may be that a repair we found introduces a deadlock caused by synchronization which already exists in the program.
- However, this issue can be overcome by adding more verification procedures to detect more violations.