Verification of Concurrent Programs

Parker Aldric Mar

With special thanks to:
Azadeh Farzan and Zachary Kincaid

Research Field: Program Verification

- Goal is to program code that is **safe**, i.e. code that produces the correct output for every acceptable input.
- (Recall CSC236) Specify **preconditions** and **postconditions** on the initial and final states of a program, e.g. sorting an array A:
- **One Method:** Create analysis tools to **automatically verify** whether code is safe with respect to pre/postconditions (static analysis: during compile time).
- **Our context:** Verify code is safe in the presence of **program concurrency**.

Research Field: Program Verification

- Goal is to program code that is **safe**, i.e. code that produces the correct output for every acceptable input.
- (Recall CSC236) Specify **preconditions** and **postconditions** on the initial and final states of a program, e.g. sorting an array A:

{ **Precondition:** A is an array.

Postcondition: A contains the same elements but sorted in non-decreasing order.

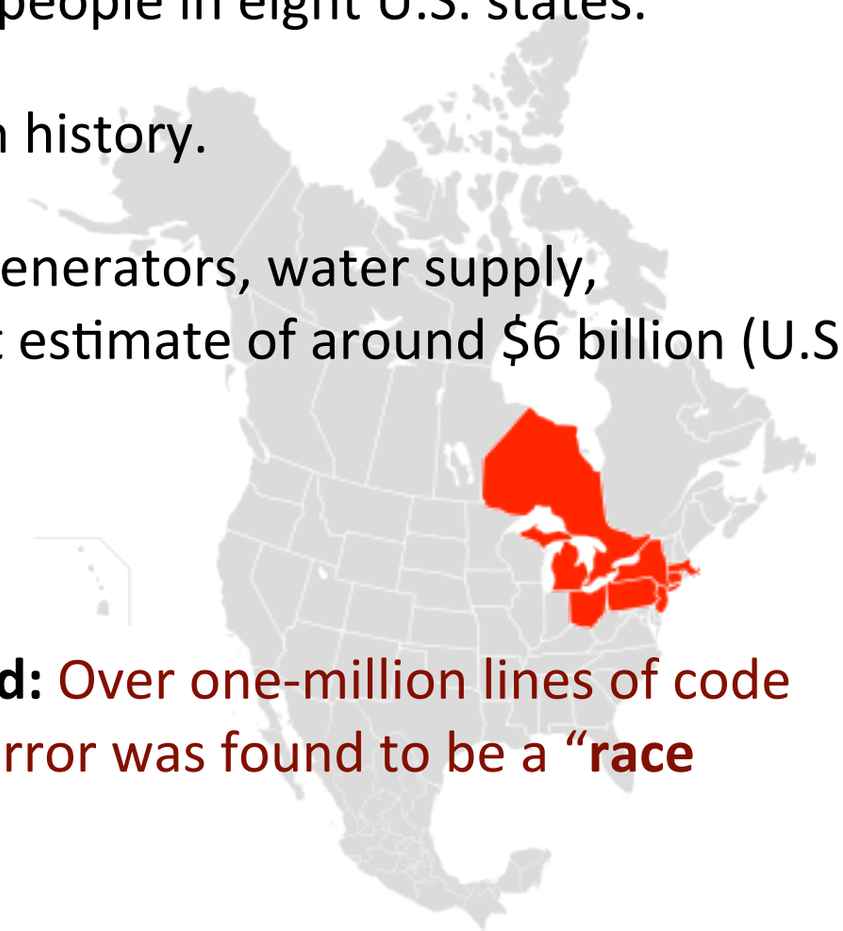
- **One Method:** Create analysis tools to **automatically verify** whether code is safe with respect to pre/postconditions (static analysis: during compile time).
- **Our context:** Verify code is safe in the presence of **program concurrency**.

The Northeast Blackout of 2003

- August 14, 2003: A **widespread power outage** affecting “an estimated 10 million people in Ontario and 45 million people in eight U.S. states.”
- The second most widespread blackout in history.
- Affected infrastructure includes power generators, water supply, transportation, and industry: a total cost estimate of around $6 billion (U.S. Department of Energy).
- What happened?

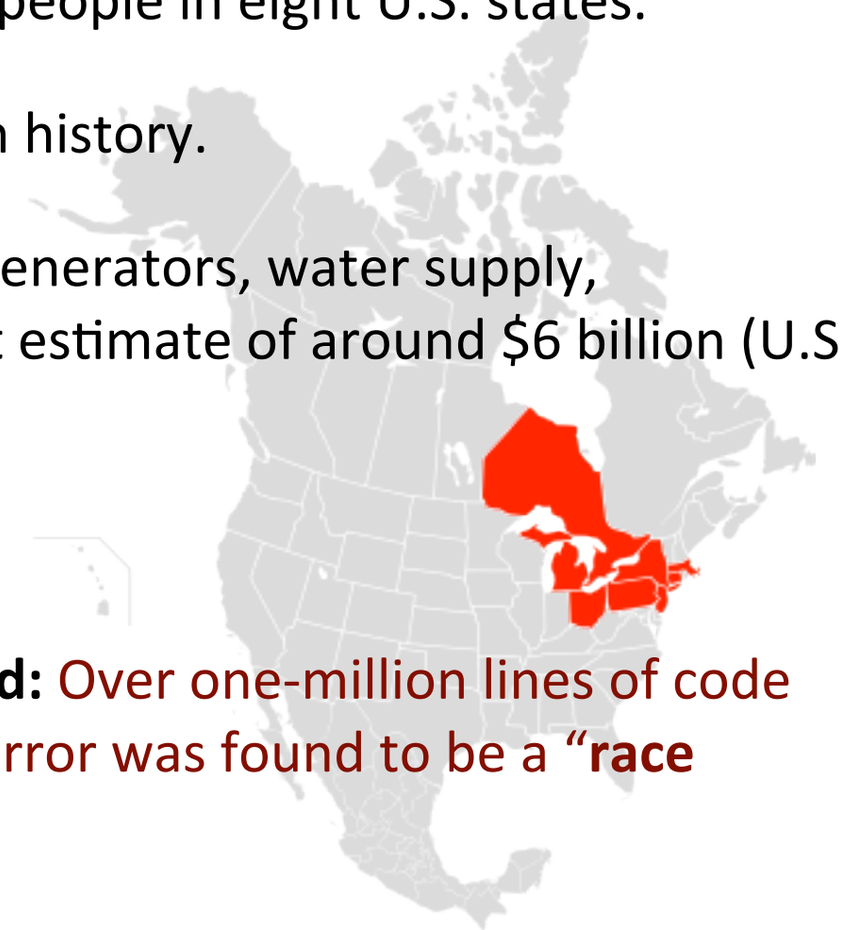
The Northeast Blackout of 2003

- August 14, 2003: A **widespread power outage** affecting "an estimated 10 million people in Ontario and 45 million people in eight U.S. states."
- The second most widespread blackout in history.
- Affected infrastructure includes power generators, water supply, transportation, and industry: a total cost estimate of around $6 billion (U.S. Department of Energy).
- What happened?
  - **A concurrent program error occurred:** Over one-million lines of code was examined for eight weeks. The error was found to be a "**race condition**".



The Northeast Blackout of 2003

- August 14, 2003: A **widespread power outage** affecting "an estimated 10 million people in Ontario and 45 million people in eight U.S. states."
- The second most widespread blackout in history.
- Affected infrastructure includes power generators, water supply, transportation, and industry: a total cost estimate of around $6 billion (U.S. Department of Energy).
- What happened?
  - **A concurrent program error occurred:** Over one-million lines of code was examined for eight weeks. The error was found to be a "**race condition**".
  - **The bug never occurred before:** "We had in excess of three million online operational hours in which nothing had ever exercised that bug. I'm not sure that more testing would have revealed it."



Lessons...

- Program concurrency can make code ***difficult to debug*** and its behavior ***difficult to predict***.
- One common concurrent program error is a **race condition**.
  - Here's an instance: One component (or thread) of a main program writes to while another one reads from the same shared variable. When the components are run concurrently, their lines of code may execute in any order (they may "interleave"). This means that the state of the program is essentially nondeterministic.
  - An example that illustrates a race condition on a sorted integer array:

Lessons...

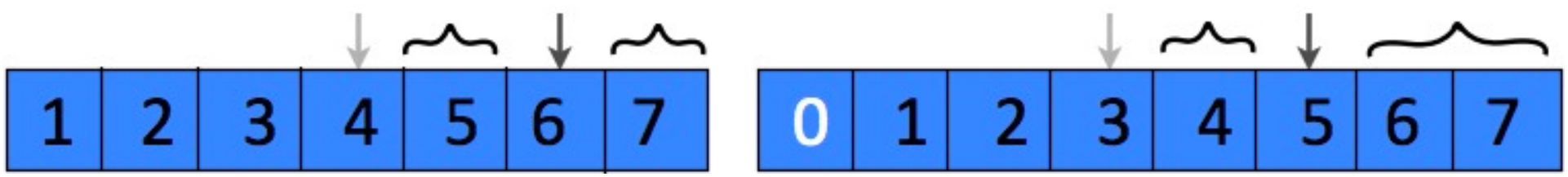
- Program concurrency can make code ***difficult to debug*** and its behavior ***difficult to predict***.
- One common concurrent program error is a **race condition**.
  - Here's an instance: One component (or thread) of a main program writes to while another one reads from the same shared variable. When the components are run concurrently, their lines of code may execute in any order (they may "interleave"). This means that the state of the program is essentially nondeterministic.
  - An example that illustrates a race condition on a sorted integer array:
    - Operation `insert(x)` inserts element `x` where it belongs in the array.
    - Operation `getIndex(x)` returns the index of an element `x` in the array, or -1 if it is not in the array. This operation is implemented using **binary search**.

Lessons...

- Program concurrency can make code ***difficult to debug*** and its behavior ***difficult to predict***.
- One common concurrent program error is a **race condition**.
  - Here's an instance: One component (or thread) of a main program writes to while another one reads from the same shared variable. When the components are run concurrently, their lines of code may execute in any order (they may "interleave"). This means that the state of the program is essentially nondeterministic.
  - An example that illustrates a race condition on a sorted integer array:
    - Operation `insert(x)` inserts element `x` where it belongs in the array.
    - Operation `getIndex(x)` returns the index of an element `x` in the array, or -1 if it is not in the array. This operation is implemented using **binary search**.
    - Run two threads **concurrently**: Thread 1 does `insert(0)`.
      Thread 2 does `getIndex(5)`.

Lessons...

- Program concurrency can make code ***difficult to debug*** and its behavior ***difficult to predict***.
- One common concurrent program error is a **race condition**.
  - Here's an instance: One component (or thread) of a main program writes to while another one reads from the same shared variable. When the components are run concurrently, their lines of code may execute in any order (they may "interleave"). This means that the state of the program is essentially nondeterministic.
  - An example that illustrates a race condition on a sorted integer array:
    - Operation `insert(x)` inserts element `x` where it belongs in the array.
    - Operation `getIndex(x)` returns the index of an element `x` in the array, or -1 if it is not in the array. This operation is implemented using **binary search**.
    - Run two threads **concurrently**: Thread 1 does `insert(0)`.
      Thread 2 does `getIndex(5)`.
    - What if `insert(0)` occurs **during** the following recursive call to `getIndex(5)`?



`getIndex(5)...` `...insert(0)...getIndex(5)?`

Concurrent Code: Increment Example

- Let's look at an example that illustrates why concurrent code is ***difficult to verify***.

Precondition: x = 0

Postcondition: x = 4

Thread A	Thread B
$\ell_{A,1}$: x++	$\ell_{B,1}$: x++
$\ell_{A,2}$: x++	$\ell_{B,2}$: x++

- An automatic program checker must verify that ***every*** interleaved thread execution of the program is correct with respect to the pre/postconditions:

Concurrent Code: Increment Example

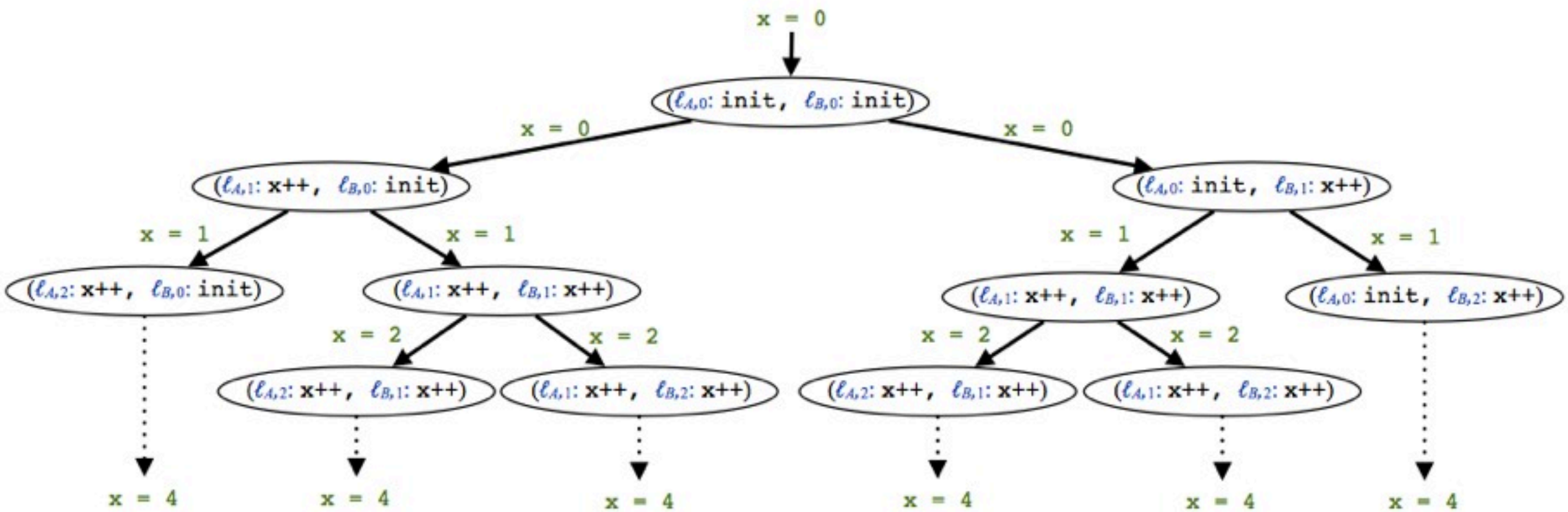
- Let's look at an example that illustrates why concurrent code is ***difficult to verify***.

Precondition: x = 0

Postcondition: x = 4

Thread A	Thread B
$\ell_{A,1}$: x++	$\ell_{B,1}$: x++
$\ell_{A,2}$: x++	$\ell_{B,2}$: x++

- An automatic program checker must verify that ***every*** interleaved thread execution of the program is correct with respect to the pre/postconditions:



Concurrent Code: Increment Example

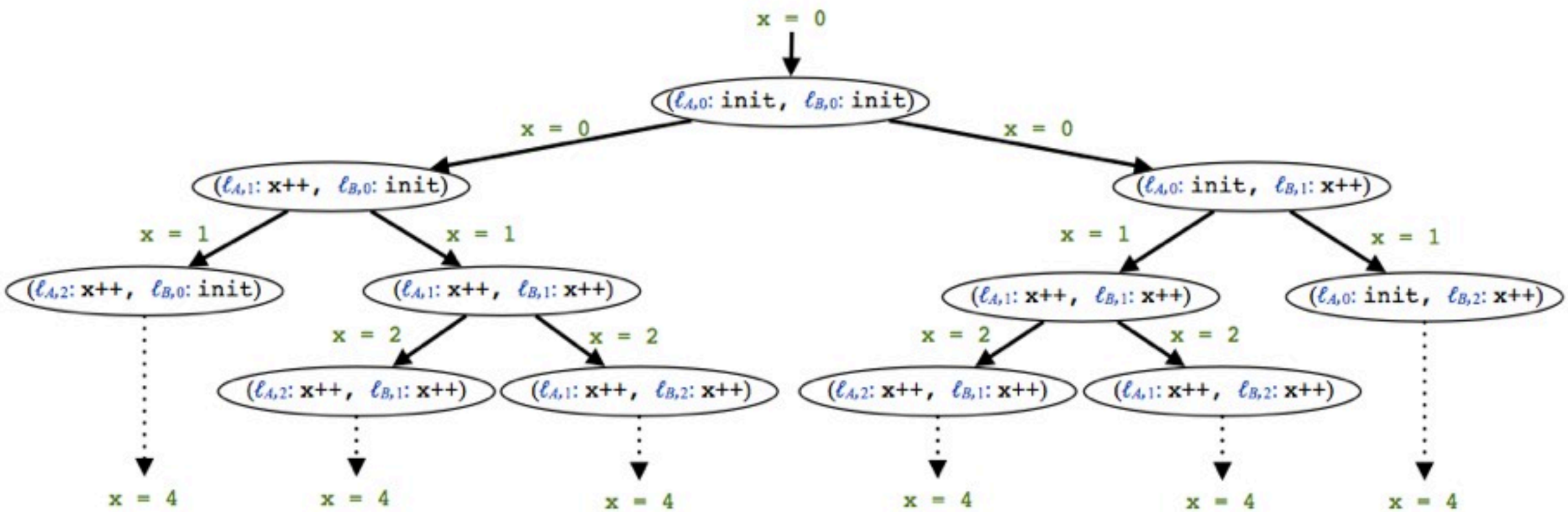
- Let's look at an example that illustrates why concurrent code is ***difficult to verify***.

Precondition: x = 0

Postcondition: x = 4

Thread A	Thread B
$\ell_{A,1}$: x++	$\ell_{B,1}$: x++
$\ell_{A,2}$: x++	$\ell_{B,2}$: x++

- An automatic program checker must verify that ***every*** interleaved thread execution of the program is correct with respect to the pre/postconditions:



The number of interleaved thread executions grows ***exponentially*** *with the number of threads!*

Tools for Verification of Concurrent Programs

- **Goal:** We are working on a tool to automatically prove the absence of defects in concurrent code during compile time. With respect to pre/postconditions, this means that we want either:
  - A **proof of correctness**, or
  - A **counterexample** in the form of an **interleaved program execution** (a "program trace") that leads to incorrect behavior.
- **Approach:** Use a **data structure** that encapsulates the language of the program to check if there exists an interleaved program execution that violates pre/postconditions.
- **Previous Efforts:** Use of a **control flow graph (CFG)** to encapsulate the flow of program P across ***every*** interleaved thread execution.

  Problem: (See the CFG from the previous slide)
  *The size of a control flow graph grows **exponentially** with the number of threads!*

Tools for Verification of Concurrent Programs

- **Goal:** We are working on a tool to automatically prove the absence of defects in concurrent code during compile time. With respect to pre/postconditions, this means that we want either:
  - A **proof of correctness**, or
  - A **counterexample** in the form of an **interleaved program execution** (a "program trace") that leads to incorrect behavior.
- **Approach:** Use a **data structure** that encapsulates the language of the program to check if there exists an interleaved program execution that violates pre/postconditions.
- **Previous Efforts:** Use of a **control flow graph (CFG)** to encapsulate the flow of program P across ***every*** interleaved thread execution.

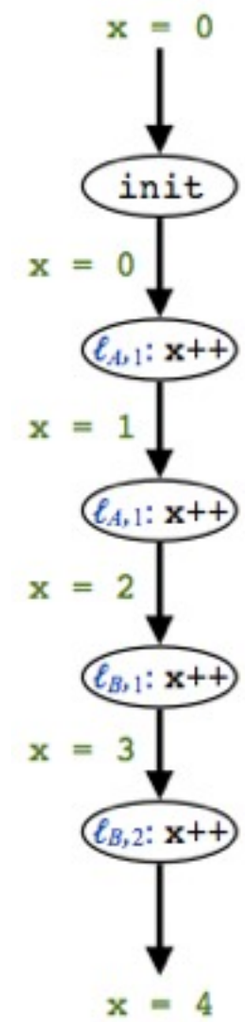
  Problem: (See the CFG from the previous slide)
  *The size of a control flow graph grows **exponentially** with the number of threads!*

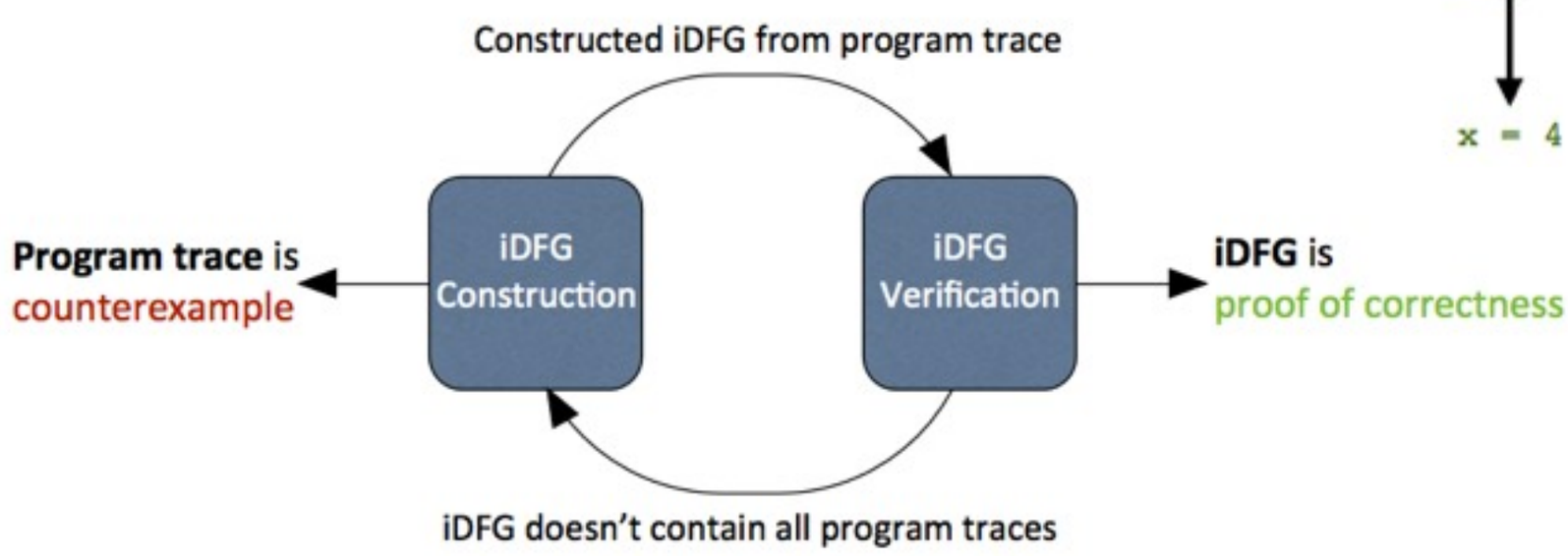
Q: *What is the State of the Art?*

Inductive Data Flow Graphs (iDFGs)

- What is an **iDFG**?
  - Proposed by Zachary Kincaid et. al.
  - An **Inductive Data Flow Graph** encapsulates relevant **data dependencies** "between program actions in interleaved thread executions."

    Resolution:
    The size of an iDFG grows polynomially with the number of threads.
  - The **iDFG verification algorithm** (below) can be used to verify the correctness of concurrent programs.



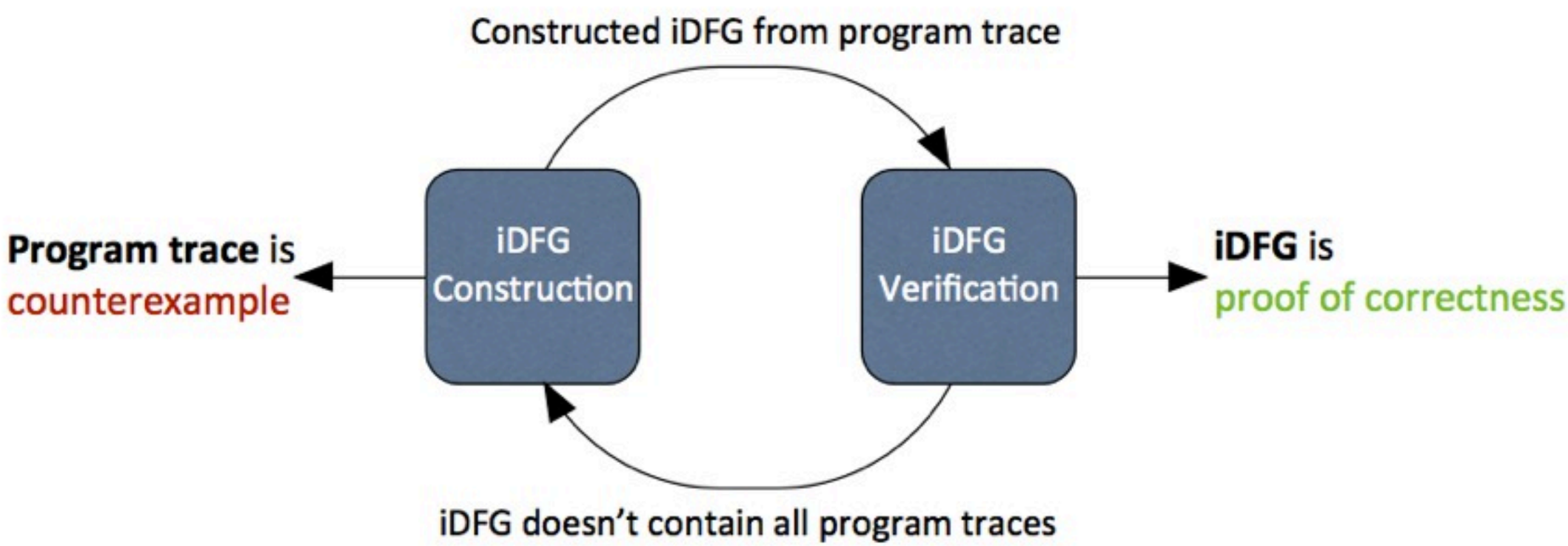


- My focus this summer is on **optimizing the iDFG verification algorithm.**

Optimizing the iDFG Verification Algorithm

- Currently, I am working on optimizing the part of the algorithm that checks the following:

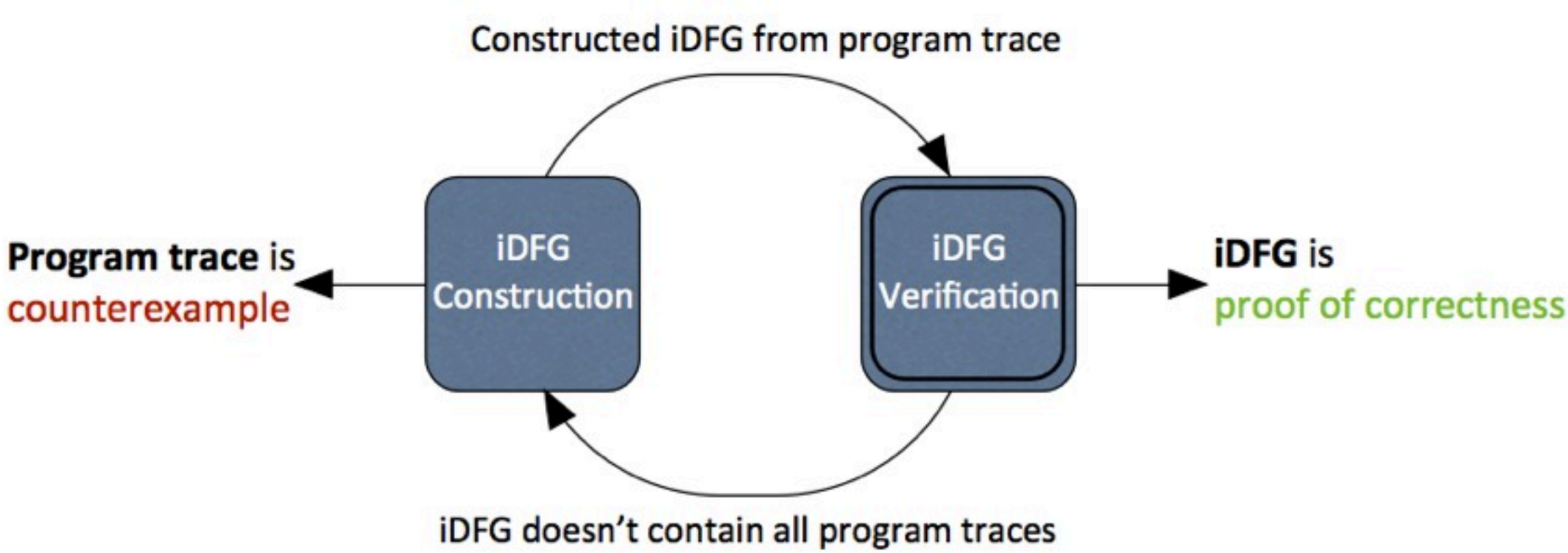
  *Does an iDFG G represent **every** interleaved thread execution of a program P?*



Optimizing the iDFG Verification Algorithm

- Currently, I am working on optimizing the part of the algorithm that checks the following:

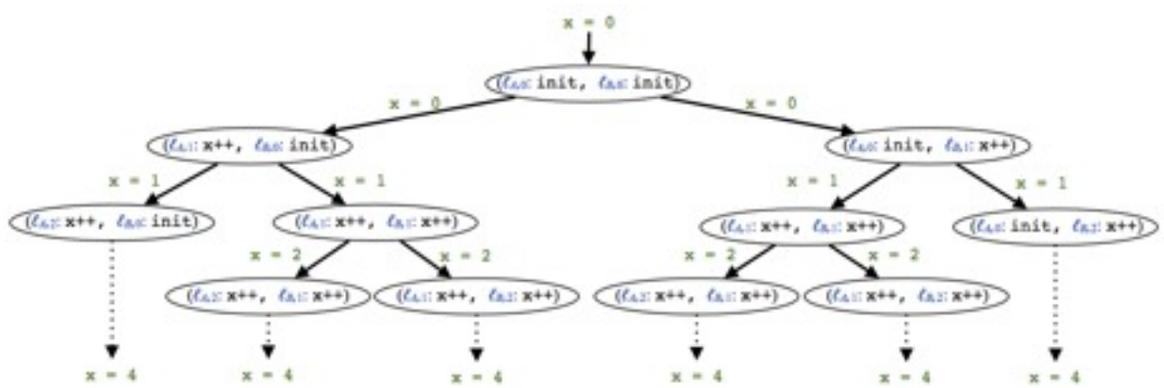
  *Does an iDFG G represent **every** interleaved thread execution of a program P?*



Optimizing the iDFG Verification Algorithm

- Currently, I am working on optimizing the part of the algorithm that checks the following:

  *Does an iDFG G represent **every** interleaved thread execution of a program P?*

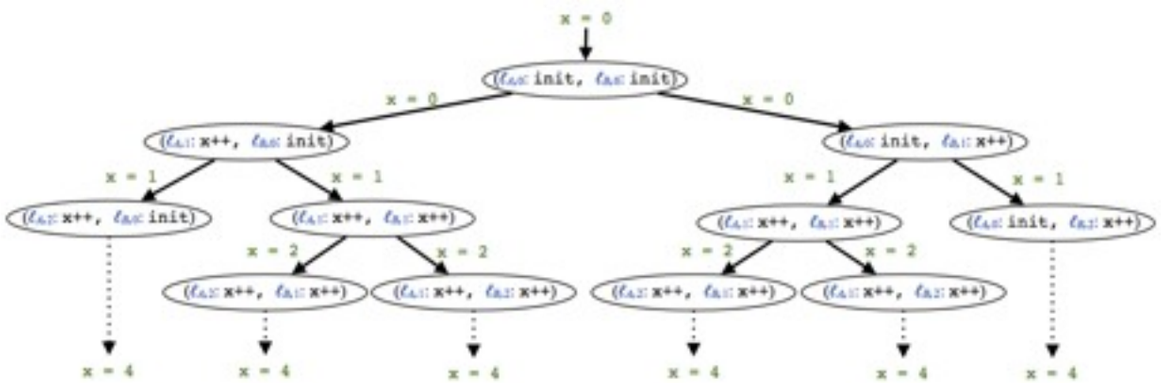


Program P

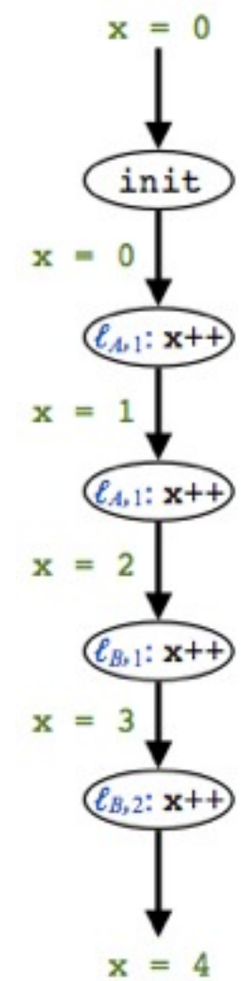
Optimizing the iDFG Verification Algorithm

- Currently, I am working on optimizing the part of the algorithm that checks the following:

*Does an iDFG G represent **every** interleaved thread execution of a program P?*



Program P

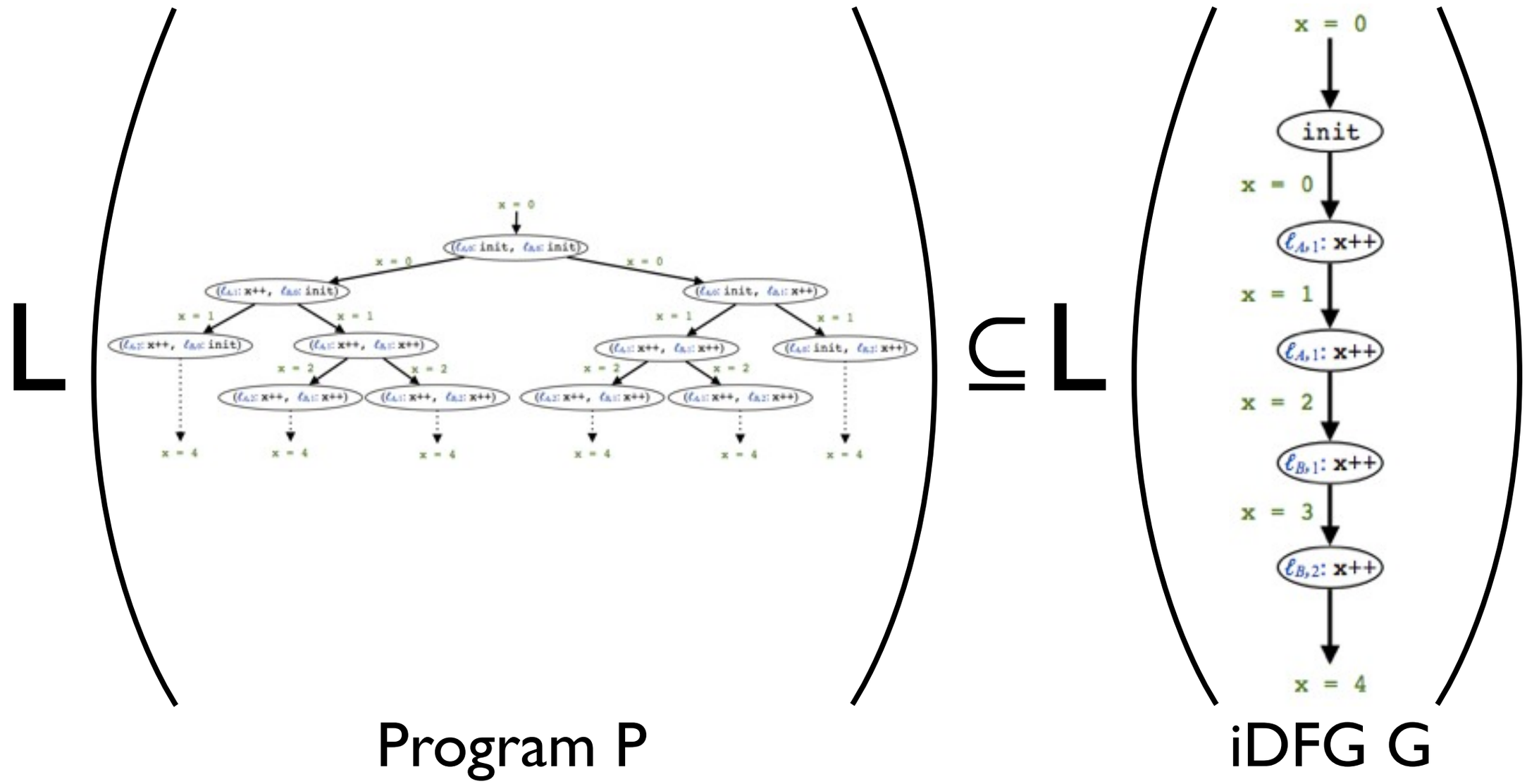


iDFG G

Optimizing the iDFG Verification Algorithm

- Currently, I am working on optimizing the part of the algorithm that checks the following:

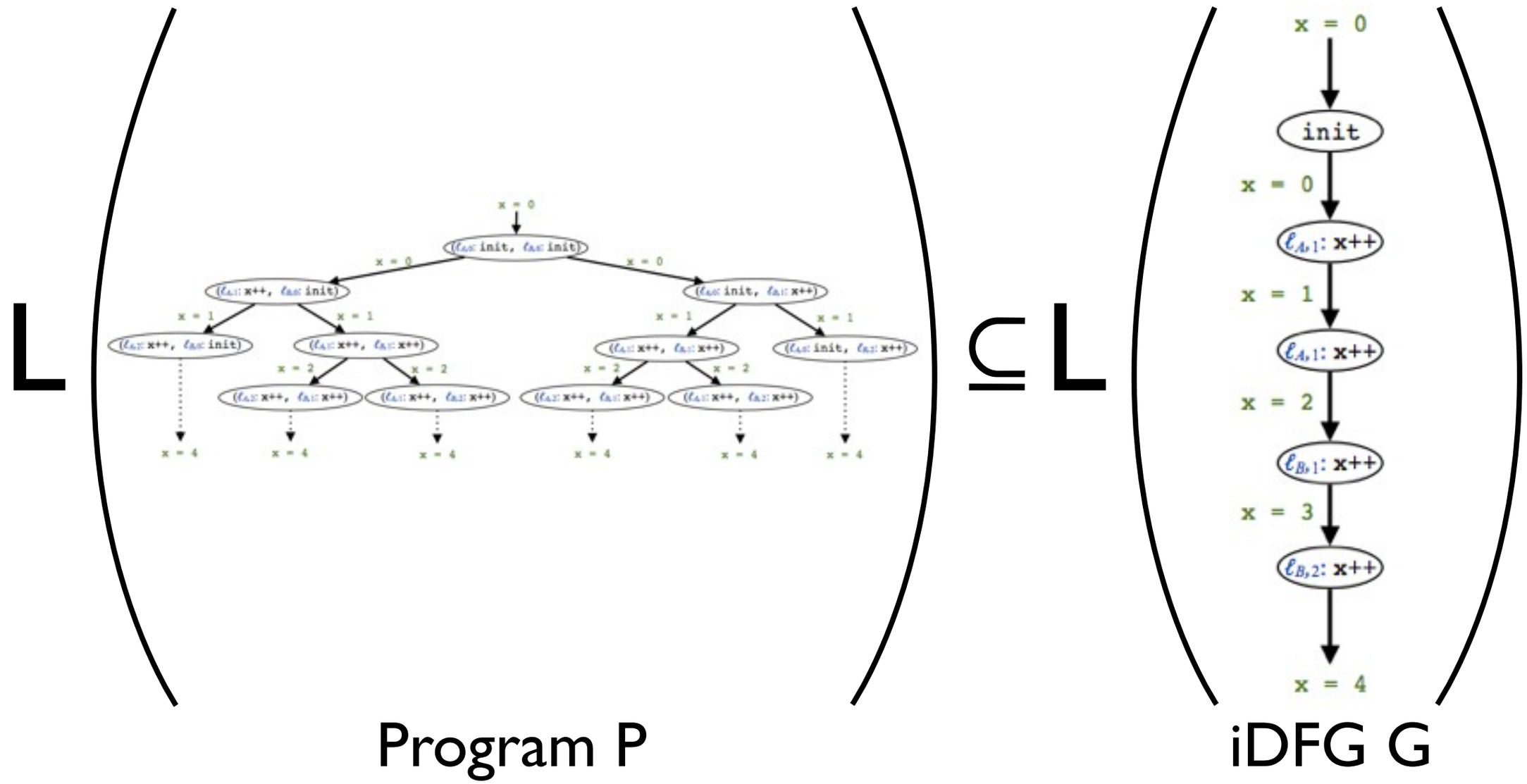
*Does an iDFG G represent **every** interleaved thread execution of a program P?*



Optimizing the iDFG Verification Algorithm

- Currently, I am working on optimizing the part of the algorithm that checks the following:

*Does an iDFG G represent **every** interleaved thread execution of a program P?*



Q: *How should we perform this optimization?*

Optimizations:

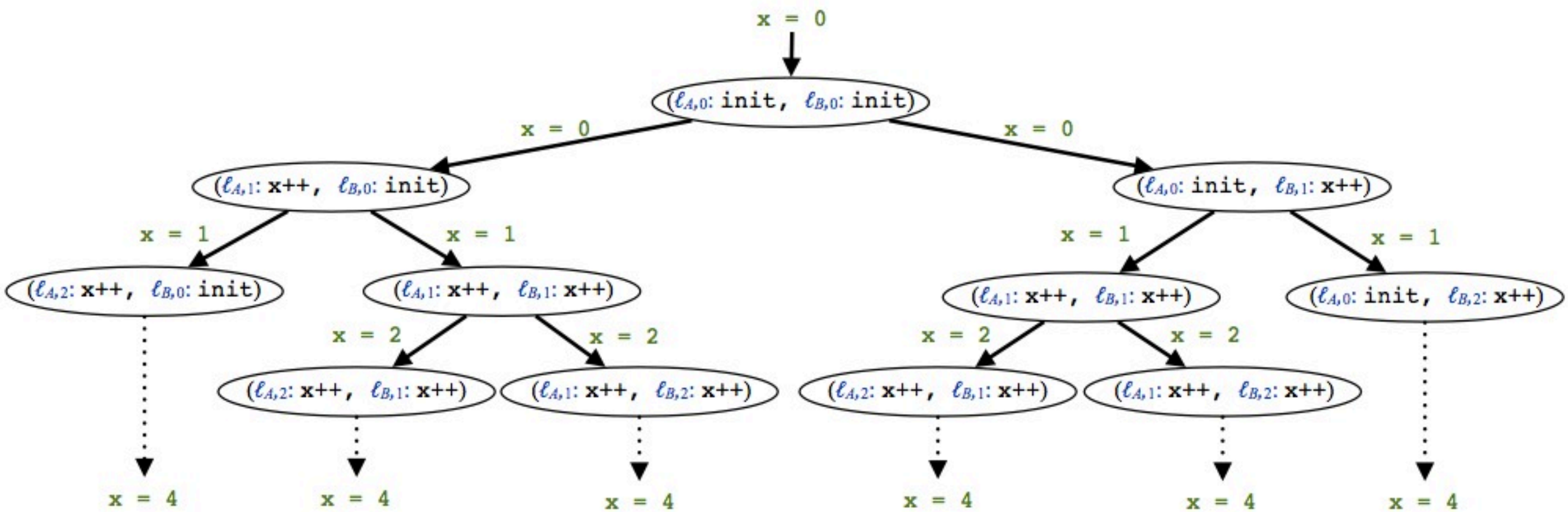
Does iDFG G represent every interleaved thread execution of program P?

I am currently working on three optimizations:

- **Optimization I: "on-the-fly"** simplifies checking whether $L(P) \subseteq L(G)$ by checking the equivalent expression $L(P) \cap \overline{L(G)} = \varnothing$ and iteratively performing the intersection construction "on-the-fly" to avoid expanding the whole intersection (which is impractical).
- **Optimization II: symmetry reduction** improves the iterative construction method of optimization I by exploiting the inherit symmetry between parallel components of concurrent code in $L(P)$.
- **Optimization III: rejecting sync state** builds on optimization I by avoiding construction when data dependencies between program actions are maintained during loops.

Optimizations:

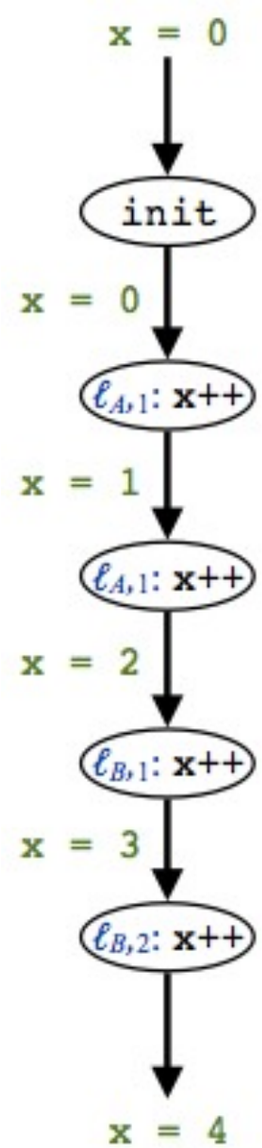
Does iDFG G represent every interleaved thread execution of program P?



Program P

Optimizations:

Does iDFG G represent every interleaved thread execution of program P?



iDFG G

Optimizations:

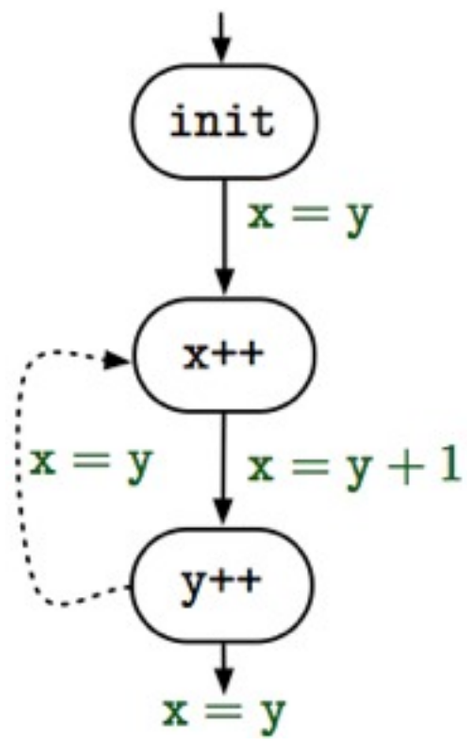
Does iDFG G represent every interleaved thread execution of program P?

Precondition: x = y

Postcondition: x = y

Program P'

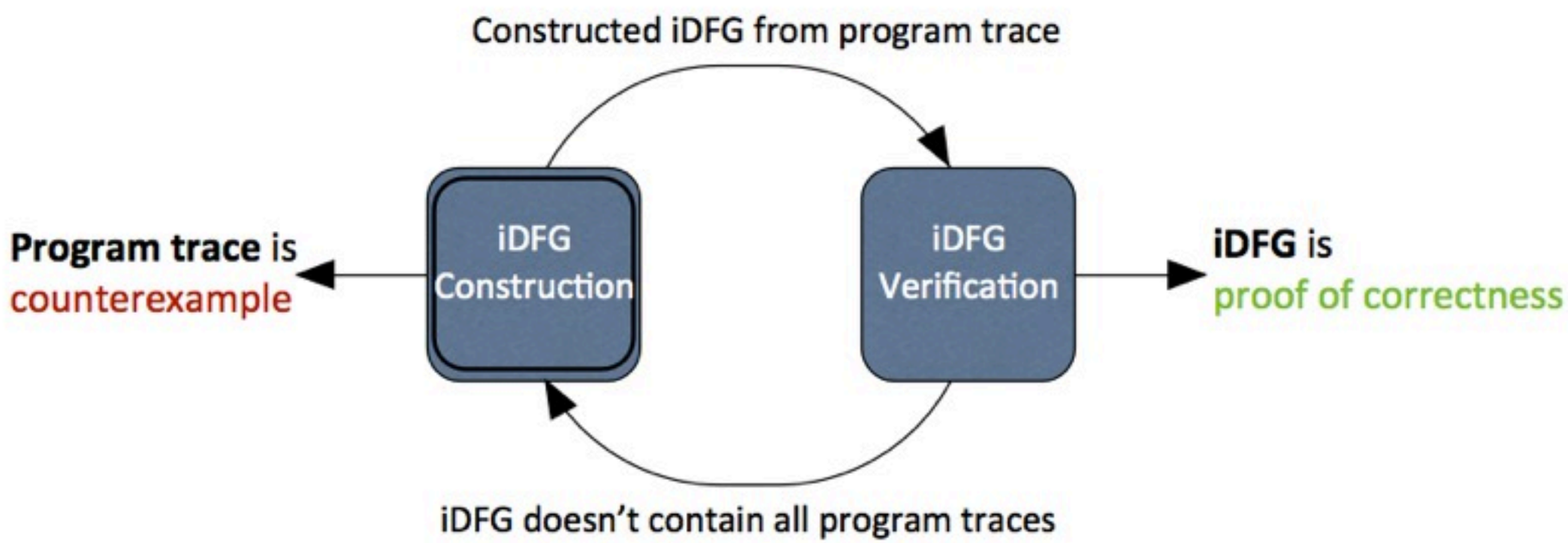
```
ℓ1: while(true):
ℓ2:     x++
ℓ3:     y++
```



iDFG G'

Progress

- Current work: Optimization II (symmetry reduction)
- Future work: Optimizations on the construction part of the iDFG verification algorithm, by selecting a program trace that will reduce the total number of traces needed for the iDFG verification algorithm to terminate.



Conclusions

- Verifying concurrent code can be challenging.
- An Inductive Data Flow Graph (iDFG) is a graph that encapsulates data dependencies between program actions. It is succinct, even in the presence of program concurrency.
- The iDFG verification algorithm is a tool for verifying concurrent code.
- Work can be done to optimize this tool.

Thank you for your attention.

Sources

- Electricity Consumers Resource Council (ELCON). *The Economic Impacts of the August 2003 Blackout*. Rep. Electricity Consumers Resource Council (ELCON), 9 Feb. 2004. Web. 17 June 2013.

- Farzan, Azadeh, Zachary Kincaid, and Andreas Podelski. *Inductive Data Flow Graphs*. *Zachary Kincaid*. Department of Computer Science University of Toronto, n.d. Web. 18 June 2013.

- Hadzilacos, Vassos. "2.1 Program Correctness." *INTRODUCTION TO THE THEORY OF COMPUTATION*. 2007 ed. N.p.: Department of Computer Science University of Toronto, n.d. N. pag. 2007. Web. 17 June 2013. https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

- Poulsen, Kevin. "Tracking the Blackout Bug." *SecurityFocus*. SecurityFocus, 4 July 2004. Web.

Backup Slide

- concurrency vs parallelism: introduces the possibility of simultaneous running and shared resources by introducing multi-tasking vs. simultaneous running.

- multi-threading vs single-threading: multitasking (wikipedia: processor switches between different threads) vs only one thread running a single task, which must finish before the next task can execute.

- multi-threading vs multi-processing: refers to the multitasked use of a single core vs. the running of multiple cores (not necessarily sharing resources).

- parallel vs. multi-threading: idea for simultaneous running vs tools to do so.

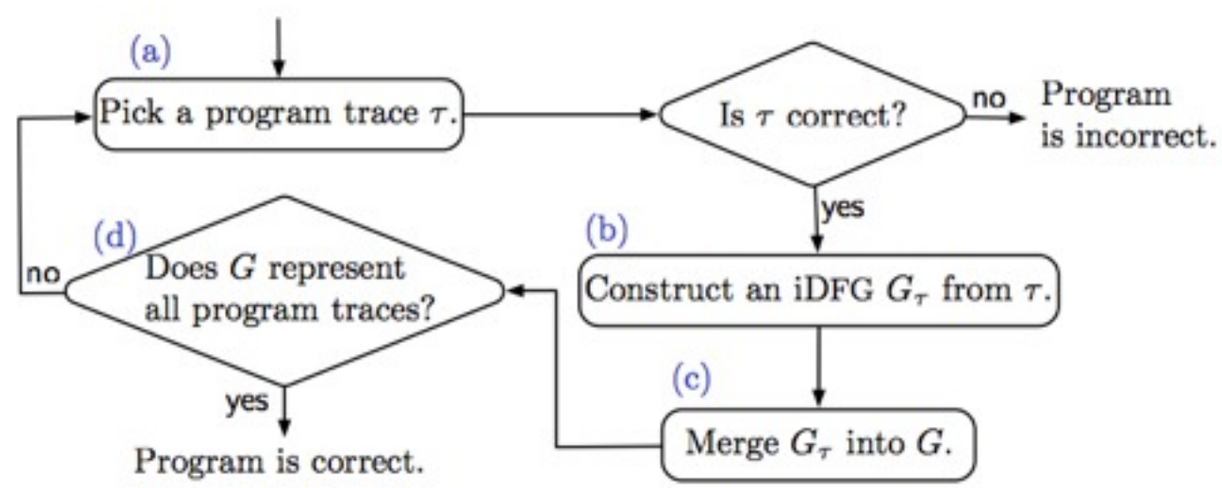


Figure 3. Verification algorithm based on inductive data flow graphs (iDFGs). Initially G is empty.