

Debugging - Omniscient Debuggers and why we need them

Steven Costiou

steven.costiou@inria.fr

EVREF / Centre Inria de l'Université de Lille

January 2024

Summary

1. Hard bugs: why we need advanced debugging
2. Omniscient debuggers: definitions
3. Examples of Omniscient debuggers
4. Implementing time-traveling debuggers
5. References

Hard bugs: Why we need advanced debugging

1

2

3

4

5

Bohr Bug [The new Hacker's Dictionary, Raymond 1996]

◆ **A Bohr Bug is a failure that is always reproducible and in a deterministic way**

▶ Conditions of its appearance are:

▶ quantifiable

▶ consistent

▶ deterministic

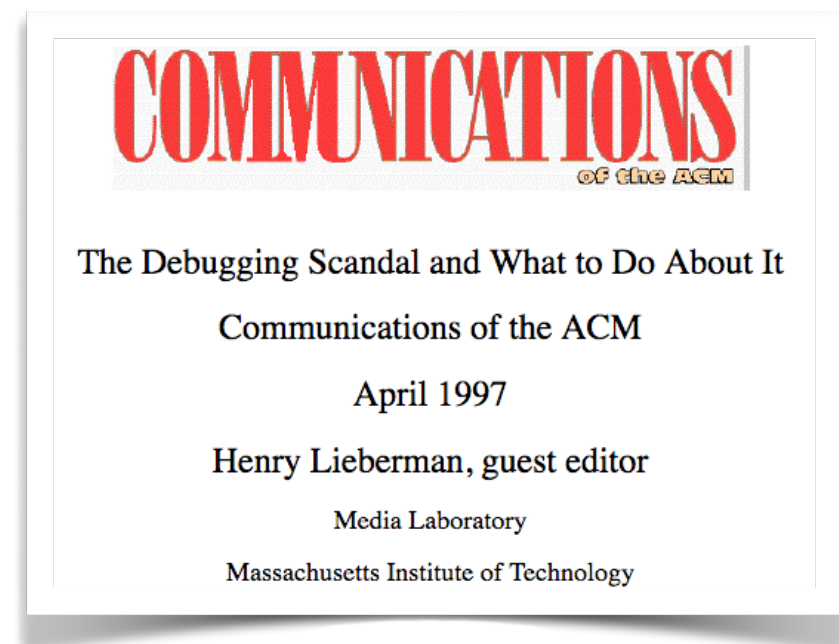
◆ Does not mean that it is easily fixable or understandable!

Hard bugs (1)

The debugging Scandal

- **Tools do not exist**
- **Tools are not adequate**
- In 1997, was referring to a situation that was already 20 years old
- It is still valid today (but we've made progress!)

1997 : still valid!



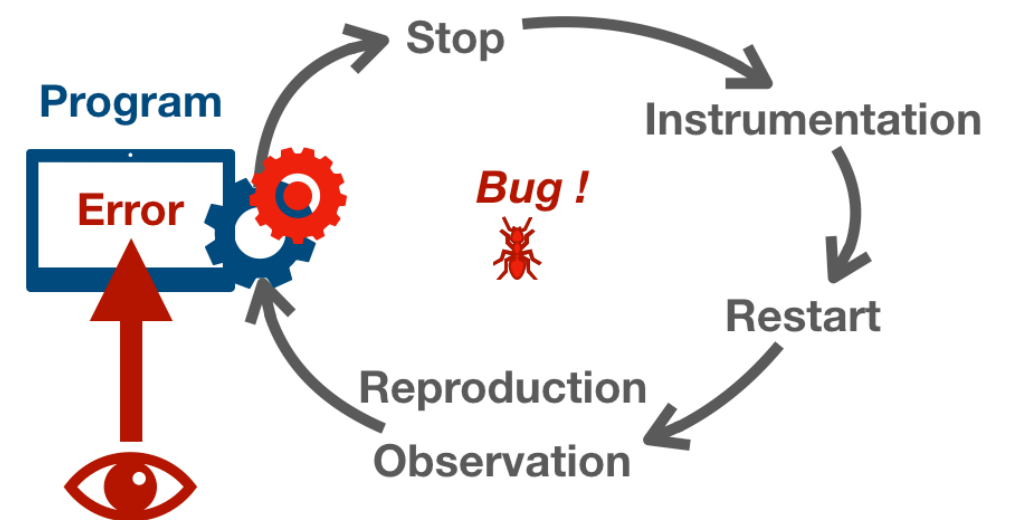
« Debugging is the dirty little secret of computer science. »

« ...embarrassment, frustration, despair... »

Hard bugs (2)

◆ Those bugs are elusive

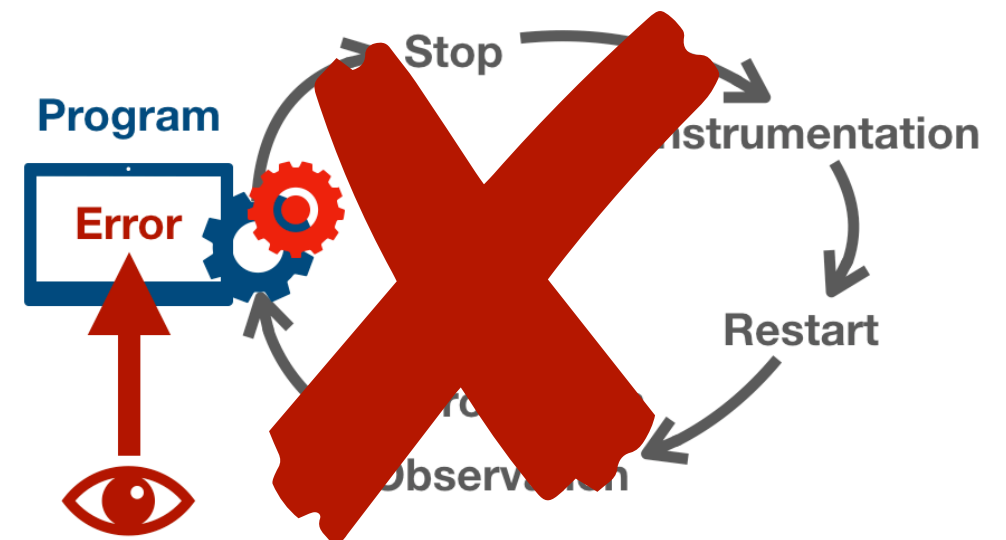
- ▶ They are hard to observe
- ▶ They are hard to reproduce
- ▶ They hide their tracks
- ▶ They are unpredictable



Hard bugs (2)

◆ Those bugs are elusive

- ▶ They are hard to observe
- ▶ They are hard to reproduce
- ▶ They hide their tracks
- ▶ They are unpredictable



“Ghost Bug” [My hairiest bug war stories, Eisenstadt 1997]

Ghost Bug, or stealth bug

This bug hides its tracks after messing with your program

- ▶ When you can observe its symptoms, it is too late!
- ▶ Information about its source or its origin is lost

“Ghost Bug” [My hairiest bug war stories, Eisenstadt 1997]

◆ Ghost Bug, or stealth bug

This bug hides its tracks after messing with your program

- ▶ When you can observe its symptoms, it is too late!
- ▶ Information about its source or its origin is lost

```
static Random rnd = new Random();  
static int randomPositive(int n) {  
    return Math.abs(rnd.nextInt()) % n;  
}
```

-2147483648

“Ghost Bug” [My hairiest bug war stories, Eisenstadt 1997]

◆ Ghost Bug, or stealth bug

This bug hides its tracks after messing with your program

- ▶ When you can observe its symptoms, it is too late!
- ▶ Information about its source or its origin is lost

```
static Random rnd = new Random();  
static int randomPositive(int n) {  
    return Math.abs(rnd.nextInt()) % n;  
}
```

?

“Surprise Scenario” [The errors of TEX, Knuth 1989]
[My hairiest bug war stories, Eisenstadt 1997]

◆ **Surprise Scenario:** you did not see it coming!

- ▶ Dormant bugs in the program
- ▶ You do not even suspect their source

“Surprise Scenario” [The errors of TEX, Knuth 1989]
[My hairiest bug war stories, Eisenstadt 1997]

◆ **Surprise Scenario:** you did not see it coming!

- ▶ Dormant bugs in the program
- ▶ You do not even suspect their source
- ▶ Often due to a misunderstanding of what the program does or should do

“Surprise Scenario”

[The errors of TEX, Knuth 1989]

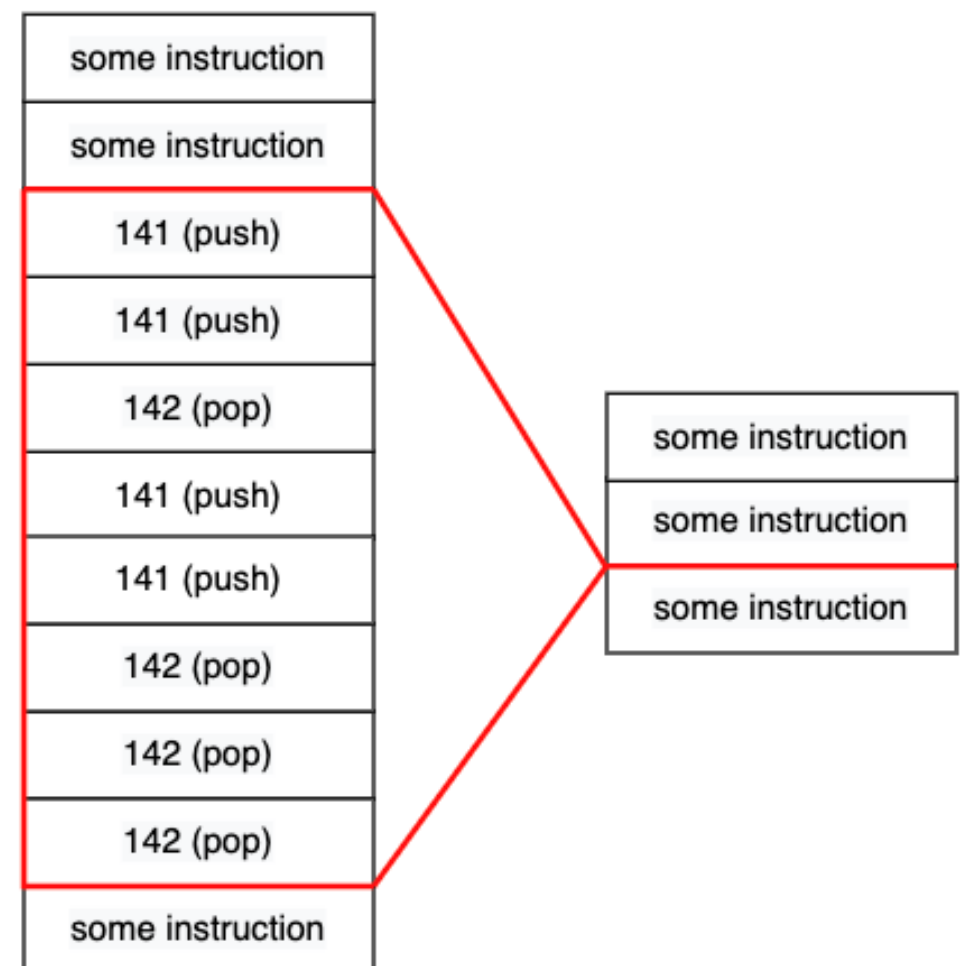
[My hairiest bug war stories, Eisenstadt 1997]

◆ Example: the final error of TEX78 (Knuth 1989)

▶ DVI language: machine-like language with 8 bits instructions followed by parameters (a sequence of bytes)

▶ Sometimes a push (141) could be immediately followed by a pop (142)

▶ Knuth optimized the output by removing instructions following this pattern



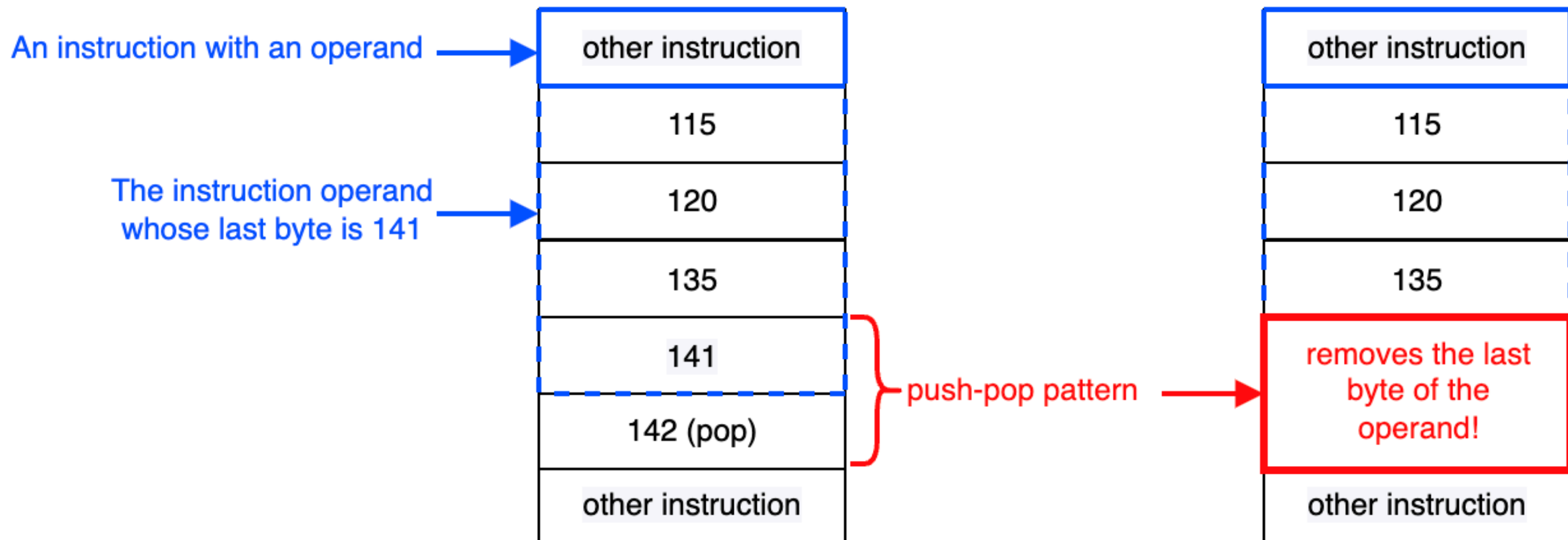
“Surprise Scenario”

[The errors of TEX, Knuth 1989]

[My hairiest bug war stories, Eisenstadt 1997]

◆ Example: the final error of TEX78 (Knuth 1989)

- ▶ Sometimes the pattern was wrongly misrecognized and removed data from parameters from other instructions!



“Heisenbug” [The new Hacker’s Dictionary, Raymond 1996]

◆ Heisenbug

Disappears when you try to observe it (or changes its behavior)

◆ Tools we use to observe and instrument programs have an influence over those programs’ behavior

◆ **Typical with concurrency:** observing a process may induce a latency in inter-process communication and hide a bug or produce a new one

“Heisenbug” [The new Hacker’s Dictionary, Raymond 1996]

◆ **Example: the Pharo language test suite**

- ◆ All unit tests are executed in the same order
 - ▶ Sometimes when you insert a test, the order is changed and then one of the tests fails
 - ▶ ...but when we run that failing test alone, now it works and even running the all suite again works!
 - ▶ Until one other test fails later in similar conditions...

“Mandelbug” [Fighting Bugs: Remove, Retry, Replicate and Rejuvenate, Grottke and Trivedi 2008]

❖ **Mandelbug**

Non-deterministic bug that only happen if specific, unpredictable and/or complex conditions are met

❖ Understanding and reproducing all conditions to reproduce a Mandelbug is difficult because of non-determinism and the lack of information

“Mandelbug” [Fighting Bugs: Remove, Retry, Replicate and Rejuvenate, Grottke and Trivedi 2008]

◆ **Mandelbug**

Non-deterministic bug that only happen if specific, unpredictable and/or complex conditions are met

◆ Understanding and reproducing all conditions to reproduce a Mandelbug is difficult because of non-determinism and the lack of information

◆ **You can hardly reproduce these bugs in development, but more easily on the deployed software**

“Mandelbug” [Fighting Bugs: Remove, Retry, Replicate and Rejuvenate, Grottke and Trivedi 2008]

◆ **Mandelbug**

Non-deterministic bug that only happen if specific, unpredictable and/or complex conditions are met

◆ Understanding and reproducing all conditions to reproduce a Mandelbug is difficult because of non-determinism and the lack of information

◆ **You can hardly reproduce these bugs in development, but more easily on the deployed software**

◆ The Patriot missile system bug can be considered as a mandelbug

“Schroedingbug” [The new Hacker’s Dictionary, Raymond 1996]

◆ **Schrøedinbug**

Implementation defect because of which the software should never have worked

“Schroedingbug” [The new Hacker’s Dictionary, Raymond 1996]

◆ **Schrøedinbug**

Implementation defect because of which the software should never have worked

◆ **It is a defect observed in the source code after which reading the software stops working until fixed**

“Schroedingbug” [The new Hacker’s Dictionary, Raymond 1996]

◆ **Schrøedinbug**

Implementation defect because of which the software should never have worked

◆ **It is a defect observed in the source code after which reading the software stops working until fixed**

- ▶ It is possible we do not realise that we actually changed something (in the configuration, in the code, hardware, etc.)
- ▶ It is also possible that after finding that in the source code, we do not use the software in the same way
- ▶ It is also possible that users (or clients) never reported it and that it was supposed to work!

Omniscient Debuggers

1

2

3

4

5

Omniscient debuggers

- ◆ These debuggers enable forward and **backward navigation** in a program execution
- ◆ We can **step forward** and **backward**, and **observe indefinitely and deterministically** the execution of the same instructions
- ◆ Based on « record/replay » or « reverse/replay » techniques

Our vocabulary

◆ Omniscient debugging/debuggers:

- ▶ Techniques and tools leveraging execution data to find information about what happened during a program execution
- ▶ Based on execution recording or/and execution reverse and replay

◆ **Back-in-time debugging/debuggers:**

- ▶ Post-mortem approach
- ▶ Records an execution, and provides means to explore that execution forward and backward

◆ **Time-traveling debugging/debuggers:**

- ▶ Interactive approach (the execution is live)
- ▶ Re-execute part or all of an execution
- ▶ Provides step-back operations

Back-in-time debugging


- ◆ **The whole execution is recorded** After run time:
 - ▶ **The recorded execution can be reloaded** in a dedicated tool (e.g., a debugger)
 - ▶ **The execution can be explored and navigated** forward and backward
- ◆ It is a post-mortem technique: the program is never re-executed

Time-Traveling Debugging


- ◆ **Recording of the program's state at key points** of the execution, named «**snapshots**»
- ◆ **During or after run time**, it is possible to **reverse the execution** to one of the snapshots
- ◆ Example of moments when snapshots are taken:
 - ▶ Breakpoints
 - ▶ Value changes in a particular variable
- ◆ It is an interactive technique: the execution is always « online », or « live »

Example

Breakpoint (snapshot)



```
rand =  
Random.nextInt(10);  
int square = rand * rand;  
float result = square / 2;
```



rand: 5

square: 25


result: 12.5



**Current pc after
step-by-step execution**

Example

Breakpoint (snapshot)



```
rand =  
Random.nextInt(10);  
int square = rand * rand;  
float result = square / 2;
```

rand: 5

square: 25

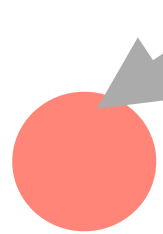
result: /



Step back

Example

Breakpoint (snapshot)



```
rand =  
Random.nextInt(10);
```

```
int square = rand * rand;
```

```
float result = square / 2;
```

rand: 5

square: /

result: /

Step back

Example

Breakpoint (snapshot)



```
rand =  
Random.nextInt(10);
```

```
int square = rand * rand;  
float result = square / 2;
```

rand: /

square: /


result: /

Step back



Example

Breakpoint (snapshot)



```
rand =  
Random.nextInt(10);  
int square = rand * rand;  
float result = square / 2;
```

rand: 5

square: /

result: /

**The recorded value
is replayed!**

Step over

Back-in-Time and Time-Traveling Debuggers Examples

1

2

3

4

5

« Interrogative » debugging

◆ **Queries over the recorded program**

- ▶ Why, at that point of the execution, is that variable in that state?
- ▶ Why is it **not** in that expected state?

◆ **Queries produces a viewable and navigable result:**

- ▶ We can visualise all entities that accessed to a variable
- ▶ We can rewind the execution flow from that point to observe the evolution of the program's state

The Whyline

The screenshot displays the Whyline IDE interface for debugging a Java Swing application. The main components are:

- Source Code Editor:** Shows the `PaintWindow.java` file. The `stateChanged()` method is highlighted, and a call to `new Color()` is circled with a '6'. A tooltip explains the call: "Called Color() on (↑) why did this execute? (1) why did getValue() return 0? (producer) (2) why did getValue() return 0? (producer) (3) why did getValue() return 0? (producer)".
- File Explorer:** Lists the project structure, including `PaintWindow$1.class`, `PaintWindow$2.class`, `PaintWindow$3.class`, and `PaintWindow.class`.
- Graphics Window:** Shows a paint application with a canvas and a toolbar. A green circle is drawn on the canvas.
- Threads Window:** Shows the execution flow, including `AWTEventQueue0-5` and `Color()`. A '5' is circled near the `Color()` call.
- Whyline Question/Answer Pane:** Displays the question "why did color = ?" and the answer "These events were responsible." with a timeline diagram showing the sequence of events from the start of the program to the `Color()` call.

At the bottom, the "Ask" pane shows the question "why did color = ?" and the "Answer" pane shows the response.

Time-Traveling Queries

◆ **Queries over the recorded program**

- ▶ Answering questions developers ask during programming activities
- ▶ 12 off-the-shelf queries available
 - ▶ Ex.: « When was this variable modified? »
- ▶ Queries allows for time-traveling to the moment where a result was found

Stack

Code pane

Object inspector

Stack

Class	Method	Package
DoubleLinkedListTest	testLinksDo	Collections-DoubleLinkedListTest
DoubleLinkedListTest	(Test performTest)	SUnit-Core
DoubleLinkedListTest	(Test [self setUp. self performTest])	SUnit-Core
FullBlockClosure	(BlockClosure ensure:)	Kernel
DoubleLinkedListTest	(Test [self setUp. self performTest])	SUnit-Core

Proceed

Into

Over

Through

Run to

Restart

Return

Where is?

Create

Advanced Step

Code pane

```
1 testLinksDo
2 | list links index |
3 list := DoubleLinkedList new.
4 links := OrderedCollection new.
5 1 to: 10 do: [ :each |
6   links add: (list add: each) ].
7 index := 1.
8 list linksDo: [ :each |
9   self assert: each equals: (links at: index).
10  self assert: each value equals: index.
11  index := index + 1 ]
```

Query input

Object inspector

Type	Variable	Value
implicit	self	DoubleLinkedListTest>>#testLinksDo
temp. var	index	nil
temp. var	list	nil
temp. var	links	nil
inst. var	testSelector	testLinksDo
inst. var	expectedFails	nil
implicit	stackTop	DoubleLinkedListTest>>#testLinksDo
implicit	thisContext	DoubleLinkedListTest>>#testLinksDo

Raw

Breakpoints

Meta

Variable	Value
Messages	All Message Sends
Messages - Object Centric	All Message Sends with selected selector
Instances Creations	All Message Sends with the selector...
Assignments - Object Centric	All Received messages
Assignments - General	
Announcements	
Microdown	
Parsers and streams	
User Queries	

Selected query

DoubleLinkedListTest>>testLinksDo

Seeker

Stepping Control

Query Scripting

Results of a query on the current execution

Step	Msg Receiver	Msg Selector	Arguments
6	274	links (OrderedCollection)	add: an Array(a DoubleLink)
7	306	list (DoubleLinkedList)	add: #(4)
8	359	links (OrderedCollection)	add: an Array(a DoubleLink)
9	391	list (DoubleLinkedList)	add: #(5)
10	444	links (OrderedCollection)	add: an Array(a DoubleLink)
11	476	list (DoubleLinkedList)	add: #(6)
12	529	links (OrderedCollection)	add: an Array(a DoubleLink)
13	561	list (DoubleLinkedList)	add: #(7)
14	614	links (OrderedCollection)	add: an Array(a DoubleLink)
15	646	list (DoubleLinkedList)	add: #(8)

Showing 20 results, fetched in: 52ms.

ExecutedBytecode: 6 (0.33% of known execution)

Predefined and user-defined time-traveling queries

Time-Traveling Queries

<https://rmod-files.lille.inria.fr/Videos/Research/2022-Time-Traveling-Queries-Demo-GDR-GPL.mp4>

How to implement time-traveling debuggers: an introduction

1

2

3

4

5

Basic implementations



◆ **The objective is to show how such debugger could be implemented**

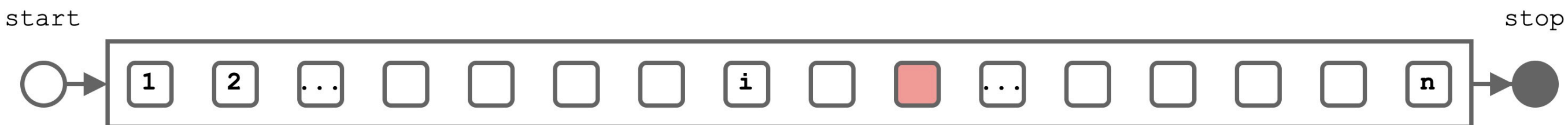
- ▶ These implementation are somewhat naive
- ▶ They are introductory to more complex materials
- ▶ They are there to experiment and learn

◆ **State-of-the-art implementations:**

- ▶ O'Callahan, Robert, et al. « Engineering record and replay for deployability: Extended technical report. » 2017.
- ▶ Maximilian Willembinck. « An interactive debugging approach based on time-traveling queries ». 2023.

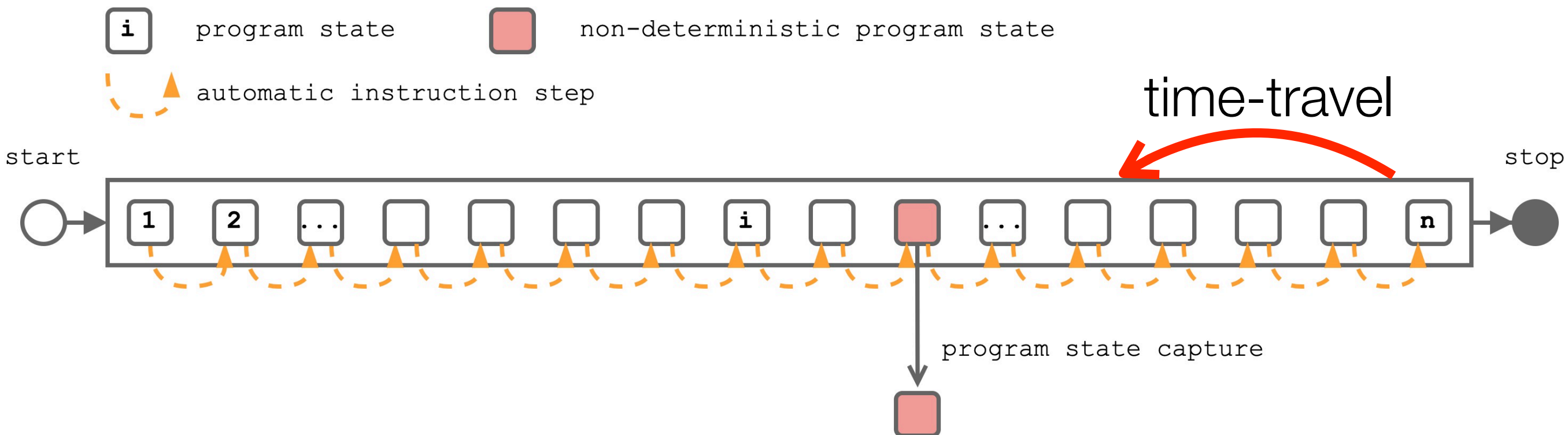
Reference example: a single-process program execution with non-deterministic values

 program state  non-deterministic program state



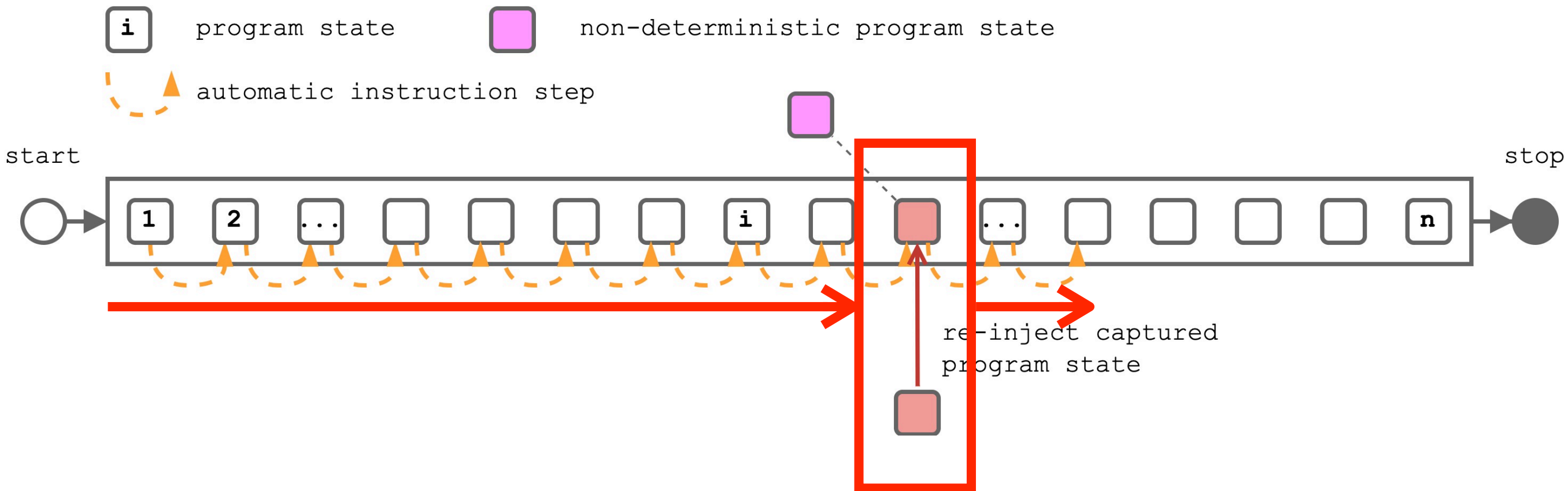
◆ **Program state:** « *the values of the program variables, as well as the current execution position* » [Zeller 09]

Step 1: control the execution



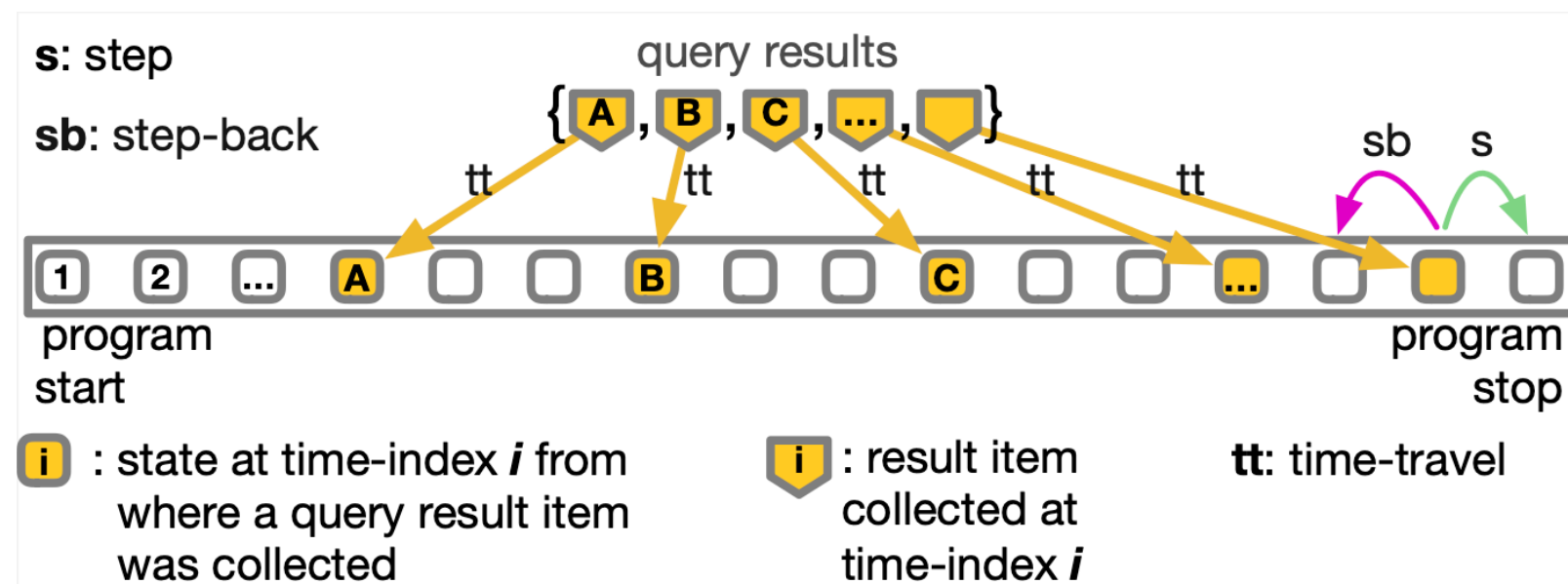
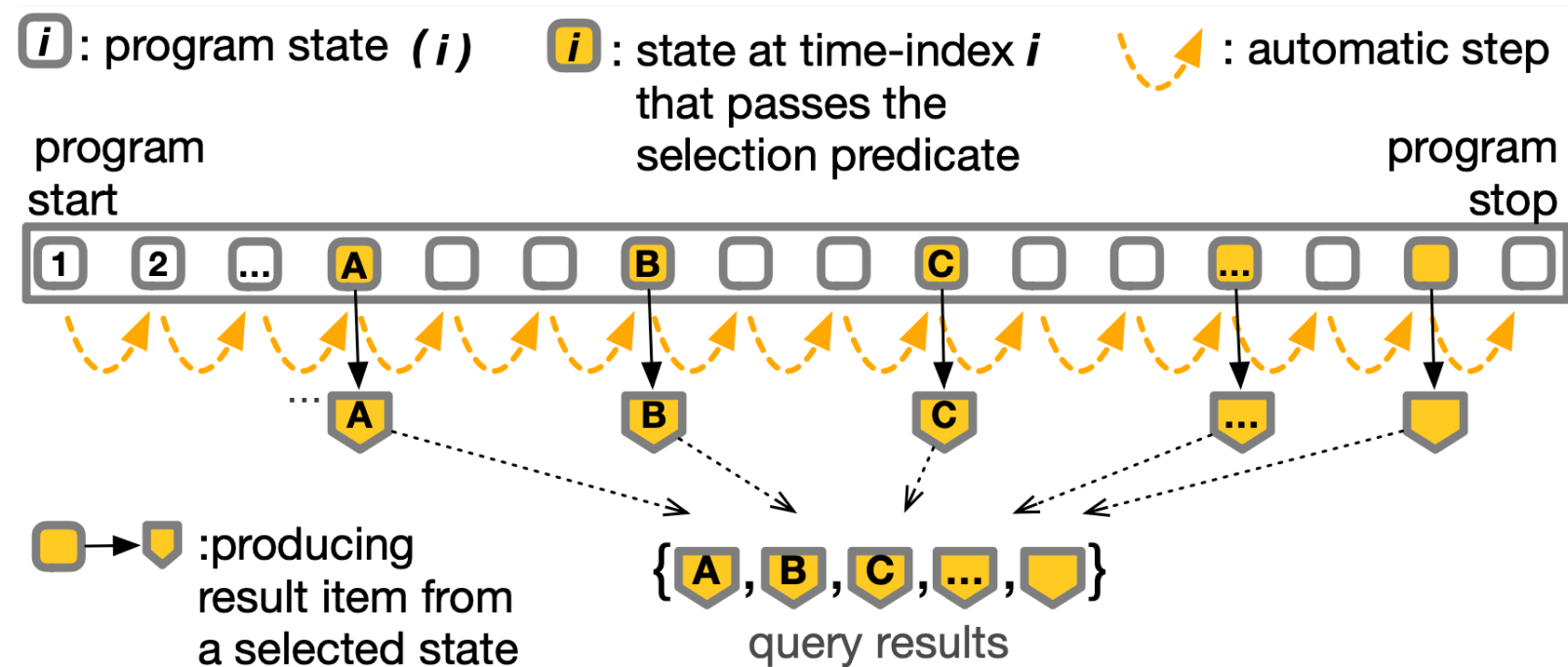
- ❖ **Cover all program states:** for example through step-by-step execution using an interpreter
- ❖ **Record non-deterministic program states** (identify them first!): primitives, system calls, etc.
- ❖ **Count the program states:** use a program counter to identify stepping order

Step 2: time-travel by replaying



- ❖ **Replay the execution from start until the target point**
- ❖ **Replace non-deterministic program state by recorded program state**
- ❖ **Stop when reaching the target program counter**

Example implementation: Time-Traveling Queries



Pros and cons

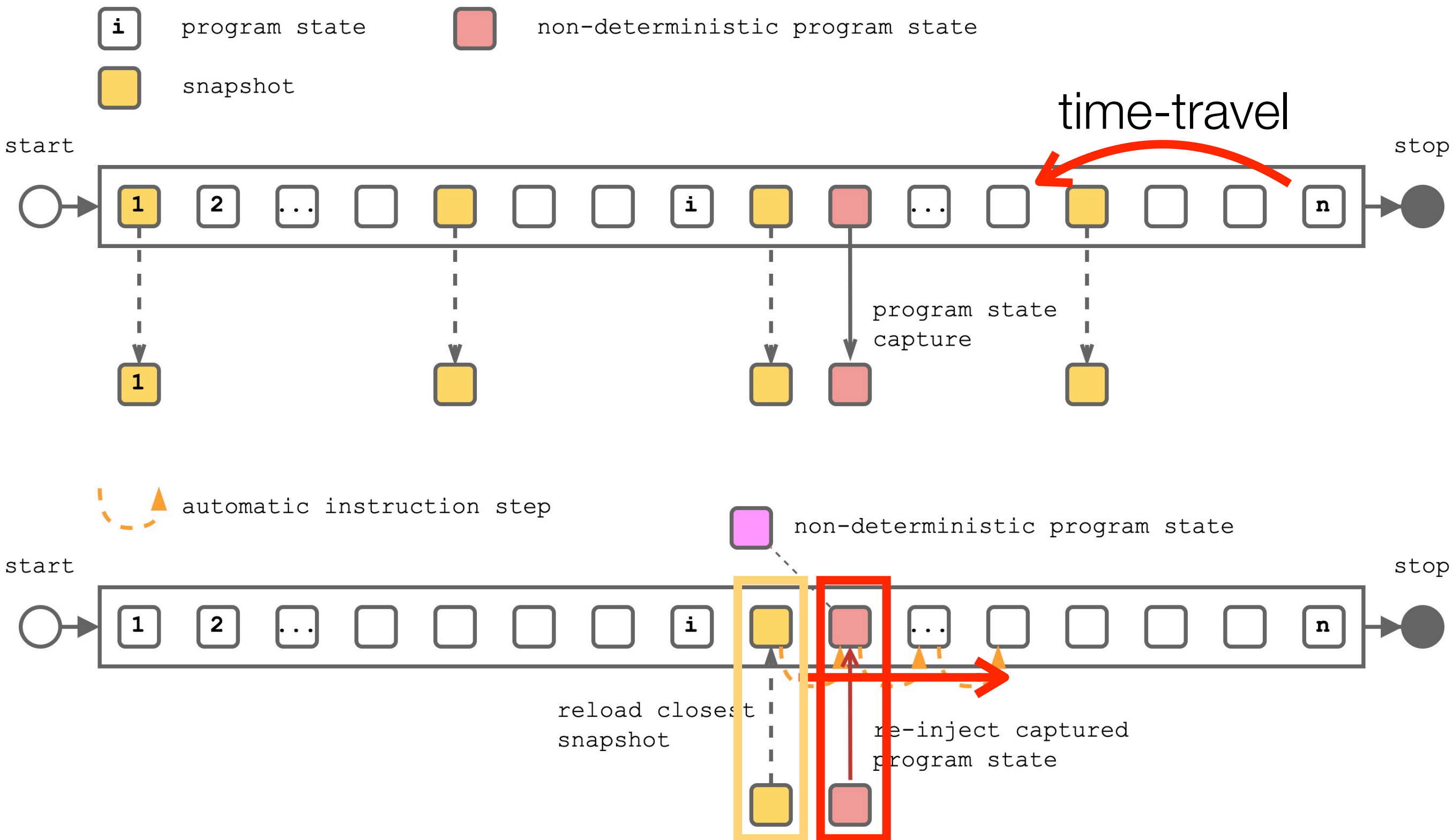
◆ **Pros**

- ▶ Easy to implement
- ▶ Good for learning how to manipulate executions
- ▶ Fine-grained execution control: adequate for tools that need to cover every program state
- ▶ Not a lot of stress on memory

◆ **Cons**

- ▶ Slow (very)
- ▶ (Does not scale easily to concurrent executions)

Better technique: snapshotting



When to snapshot?

◆ **At significant points of the execution**

- ▶ method calls,
- ▶ before/after non-deterministic computation,
- ▶ breakpoints,
- ▶ after x number of program states (interval-based),
- ▶ etc.

Pros and cons

◆ **Pros**

- ▶ Achieve good performance
- ▶ Implementation can rely on system primitives such as *fork*

◆ **Cons**

- ▶ Focus on replaying fast: limited control, does not allow to query the full execution record at each replay
- ▶ Heavy memory usage
- ▶ (Does not scale easily to concurrent executions)

Back-in-time debuggers?

◆ **Similarities, but different objective:**

- ▶ Build a recording model of your program or your language (or both)
- ▶ Instrument your program to create instance of your recording model and log this data
- ▶ Execute your program: your execution is logged according to your model
- ▶ Reconstruct your execution from your logs and visualize it in your debugger, simulating your recorded execution

◆ **The objective is to record execution data to visualize a simulation of that execution**

Example implementation: GEMOC studio debugger

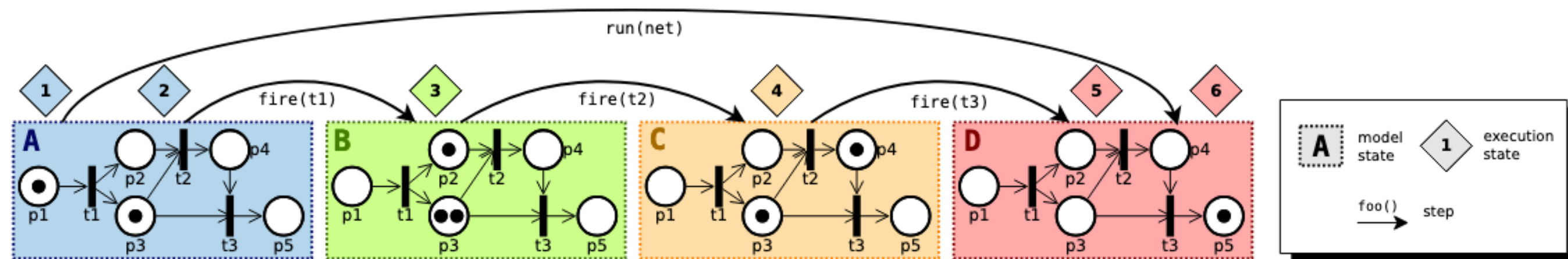


Figure 5: Example of Petri net execution trace, annotated with execution states.

Bousse, E., Leroy, D., Combemale, B., Wimmer, M. and Baudry, B. 2017. **Omniscient Debugging for Executable DSLs.** *Journal of Systems and Software (JSS)*. 137, (Nov. 2017)

Omniscient debugger: pros/cons

◆ **Pros**

- ▶ Eliminate non-determinism, as we work on recorded executions
- ▶ We can replay our execution as much as necessary until we find our bug

◆ **Cons**

- ▶ Tremendous impact on performance (some debuggers slow down the execution by more than 300 times...)
- ▶ Limited to a few minutes of execution: it is impossible to capture bugs that only appear after a long and unpredictable execution time
- ▶ Tools can be slow while loading or navigating traces (because there are a lot!)

References

1

2

3

4

5

References

1. **Out-Of-Place debugging: a debugging architecture to reduce debugging interference.** Matteo Marra, Guillermo Polito, Elisa Gonzalez Boix. The Art, Science, and Engineering of Programming, aosa, 2018.
2. **The new hacker's dictionary**, Raymond and Steele, MIT Press, 1996.
3. **Fighting bugs: Remove, retry, replicate, and rejuvenate**, Grottke and Trivedi, Computer, 2007.
4. **Bigdebug: Interactive debugger for big data analytics in apache spark**, Gulzar et. al., Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016.
5. **Debugging reinvented: asking and answering why and why not questions about program behavior**, Ko and Myers, ICSE 2008.
6. **Practical object-oriented back-in-time debugging**, Lienhard and al., ECOOP, 2008.
7. **Debugging backwards in time**, Bill Lewis, 2003.
8. **Scalable omniscient debugging**, Pothier et. al., ACM SIGPLAN Notices, 2008.
9. **A review of reverse debugging**, Jakoc Engblom, 2012.
10. **Object-centric debugging**, Ressia et. al., ICSE, 2012.
11. **Unanticipated behavior adaptation: application to the debugging of running programs**, Costiou, 2018.
12. **Squeak: Open personal computing and multimedia**, Guzdial et. al., Prentice Hall PTR, 2001.
13. **VIVA: A visual language for image processing**, Tanimoto, Journal of Visual Languages and Computing.
14. **The Handmade Hero**, Casey Muratori, 2014-2019 (see the youtube videos).
15. **Simplifying and isolating failure-inducing input**, Zeller and Hildebradt, IEEE Transactions on Software Engineering, 2002.
16. **A survey on algorithmic debugging strategies**, Josep Silva, Advances in engineering software, 2011.
17. **A survey on software fault localization**, Wong et. al., IEEE Transactions on Software Engineering, 2016.
18. **Swarm debugging: The collective intelligence on interactive debugging**, Petrillo et. al., Journal of Systems and Software, 2019