Hierarchical Abstraction of Execution Traces for Program Comprehension

Kaj Dreef

Supervisors: prof. James A. Jones & prof. Arie van Deursen

Outline

1. Introduction
2. Motivation
3. Approach
4. Hierarchical Phase Visualization
5. Evaluation
6. Discussion
7. Conclusion

Introduction

Introduction

- Software comprehension is essential while performing maintenance tasks
- Understanding runtime behavior is complex because execution traces:
    1. are long, i.e., 1.000.000's of events;
    2. contain many low-level details;
    3. contain no abstractions.
- Previous work focused on:
    - Visualizing traces
    - Discover phases in the execution

Motivation

Motivation

- Challenge 1: Information overload
- Challenge 2: Behaviors Contain Sub-Behaviors
- Challenge 3: Incomprehensible Execution Traces

Approach

-- Sage --

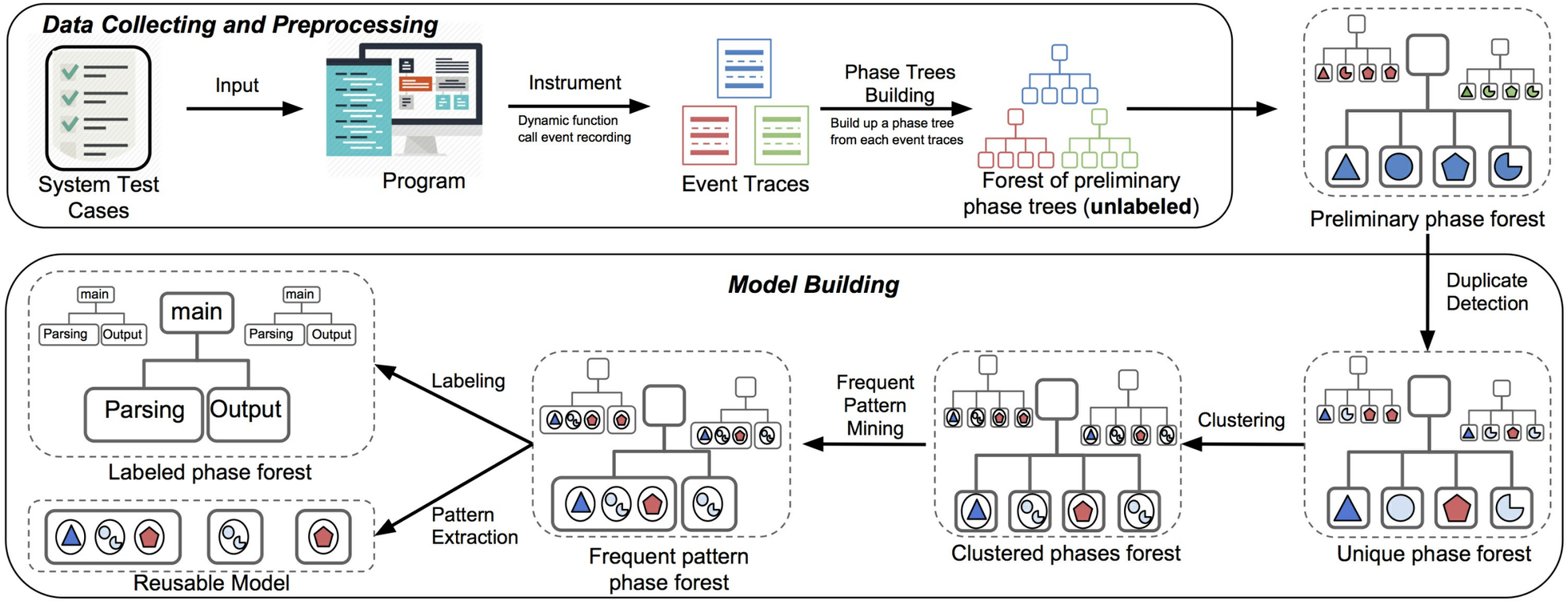
Sage

- Goal: Hierarchically abstracting an execution trace into high-level phases.

Sage

- Goal: Hierarchically abstracting an execution trace into high-level phases.
- Approach:

  1. Data Collection & Phase Detection
  2. Model Building & Execution Abstraction



Sage

Data Collection

- Dynamic Instrumenter (Blinky)
- Collect only method-invocation enter events
- Each method-invocation event is captured as a triple of:
  - Execution ID
  - Method Signature
  - Call depth

Sage

Data Collection

Phase Detection

Simplified steps of the phase detection:

1. Input of phase detection is an execution trace

Execution ID	Method Signature	Call depth
1	getDayOfMonth	1
2	getYear	2
3	getYearMillis	3
4	getYearInfo	4
5	getMonthOfYear	2
6	getYearMillis	3
7	isLeapYear	3
8	getDayOfMonth	2

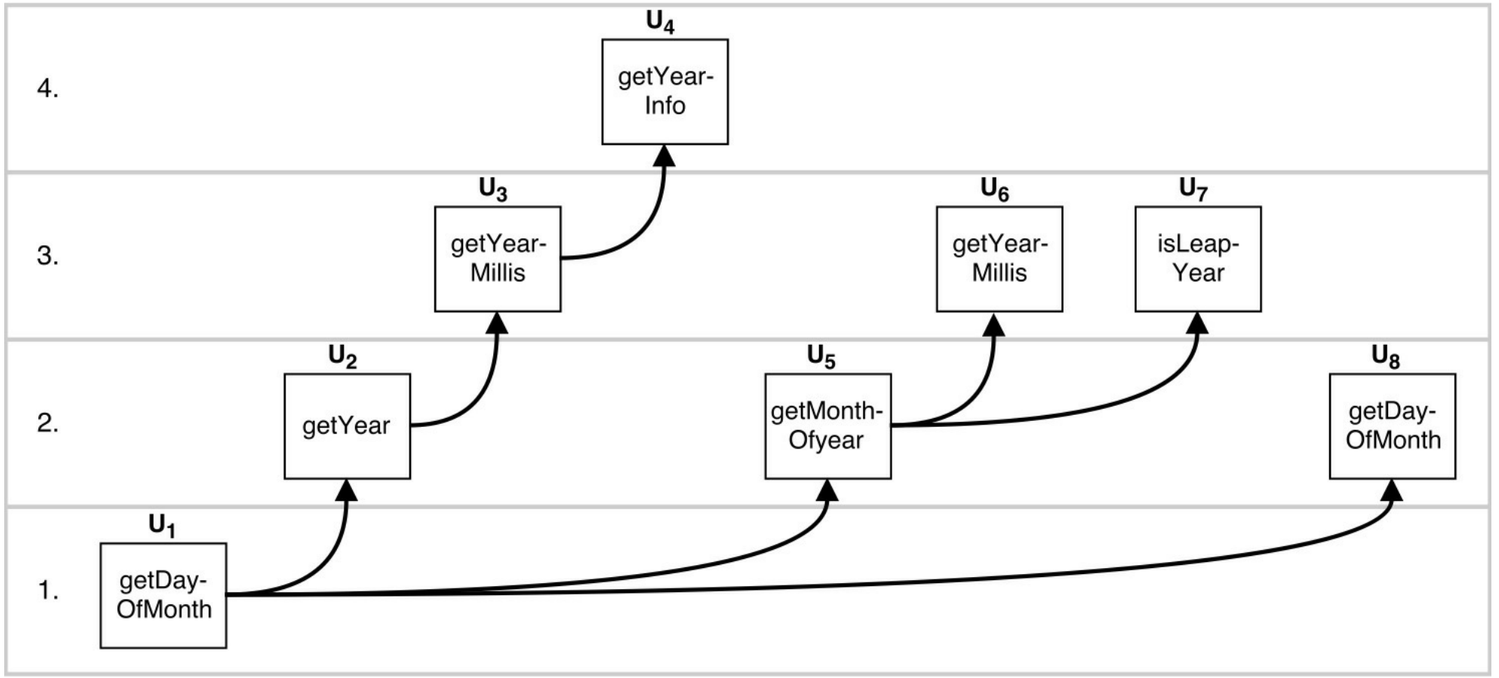
Sage

Data Collection

Phase Detection

Simplified steps of the phase detection:

1. Input of phase detection is an execution trace



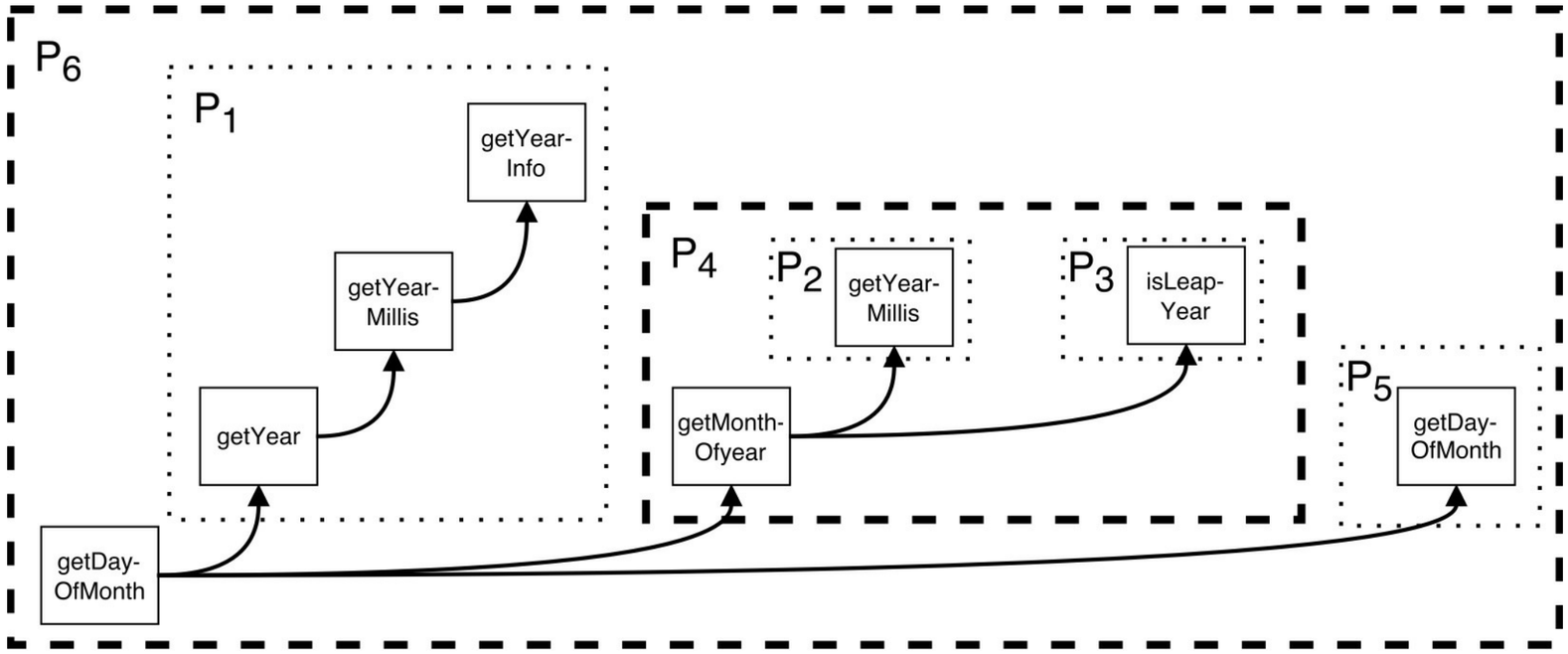
Sage

Data Collection

Phase Detection

Simplified steps of the phase detection:

1. Input of phase detection is an execution trace
2. Use the call depth to locate the phase boundaries



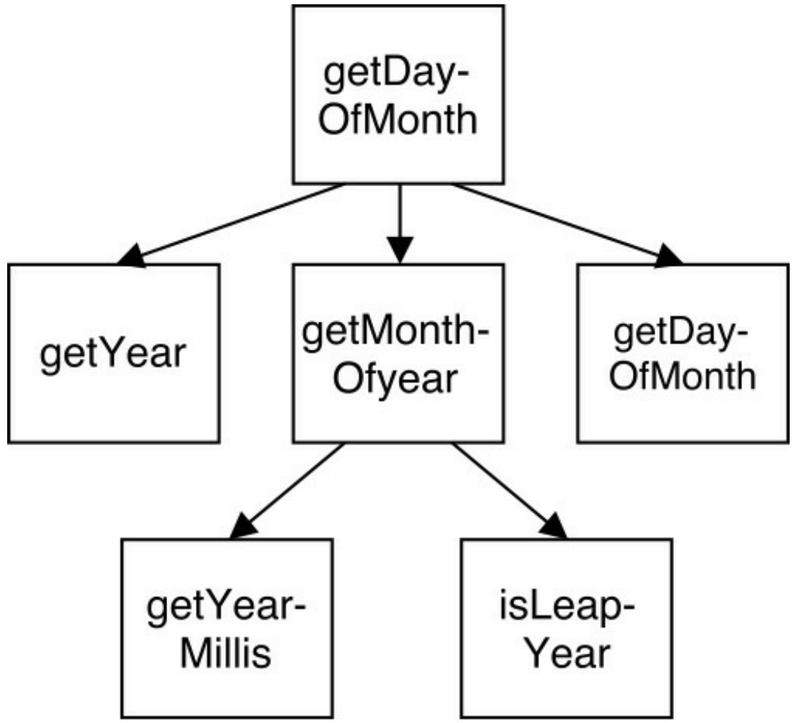
Sage

Data Collection

Phase Detection

Simplified steps of the phase detection:

1. Input of phase detection is an execution trace
2. Use the call depth to locate the phase boundaries
3. Finally, the result is a preliminary phase tree



Sage

Data Collection

Phase Detection

- Algorithm

Algorithm: Main loop

Input : *eventList*: List of execution events
Output: *root*: The root node of the phase tree
Initialization: Initialize an empty stack *historyStack* to hold previous execution events or phases.
Initialize an empty *root* node.

```
1  begin
       /* End of file event signalizes the end of the execution trace.          */
2      eventList.append( end of file event with call depth of -1 )
3      foreach event u_i in eventList do
4          if historyStack.size() == 0 then
5              historyStack.push(currEvent)
6              continue
7          end
8          prevEvent ← historyStack.peek()
9          if prevEvent.getCallDepth() < u_i.getCallDepth() then
10             historyStack.push(u_i)
11         end
12         else if prevEvent.getCallDepth() == u_i.getCallDepth() then
13             historyStack.pop()
14             node ← createNode(prevEvent)
15             historyStack.push(node)
16             historyStack.push(u_i)
17         else if prevEvent.getCallDepth() > u_i.getCallDepth() then
18             root ← findPhasesInHistory(u_i, historyStack)
19         end
20     end
21     return root
22 end
```

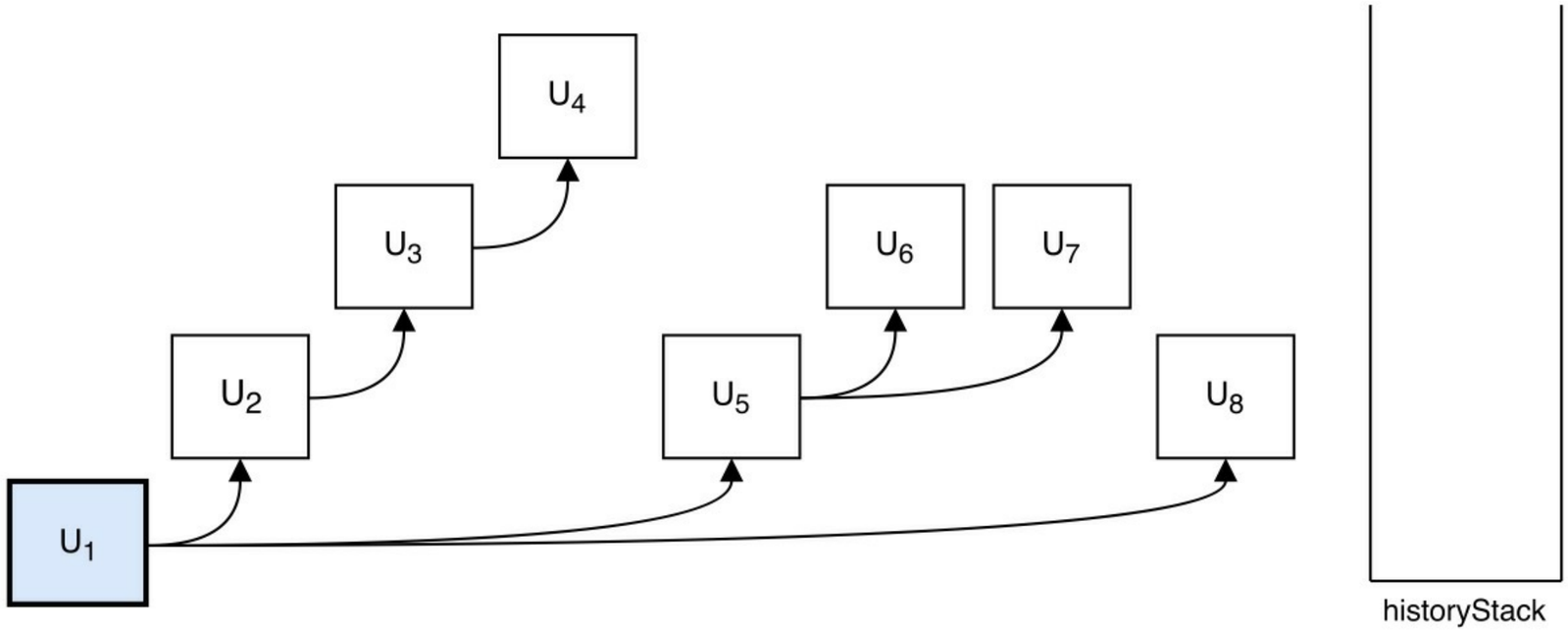
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Step 1: Initial trace - Push to history stack



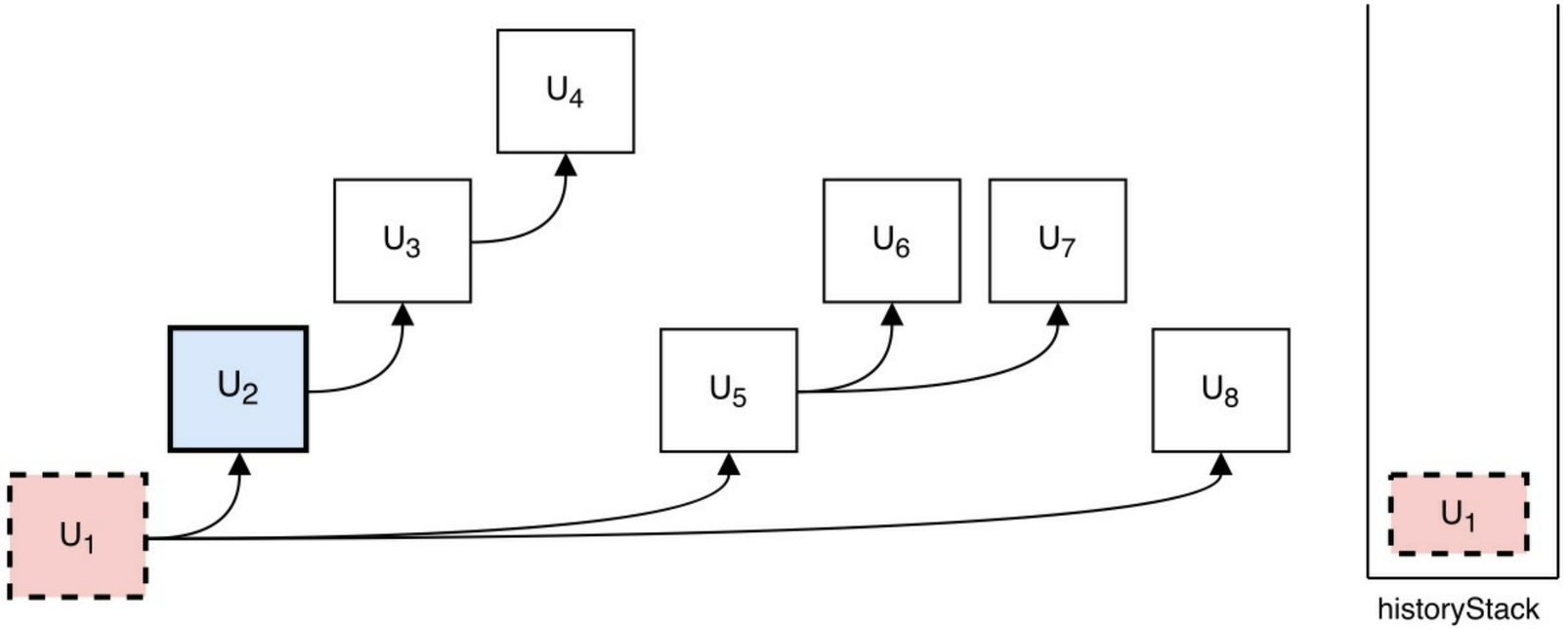
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Step 2: Push to history stack



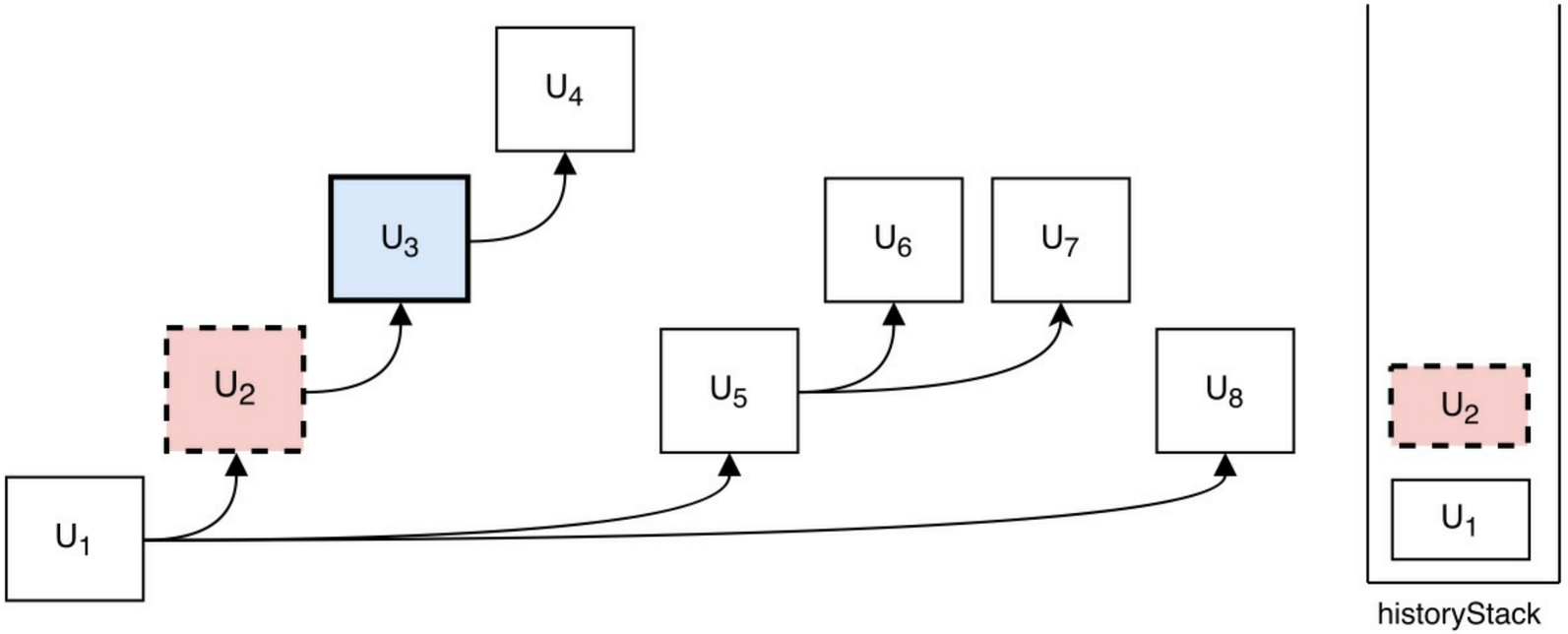
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Step 3: Push to history stack



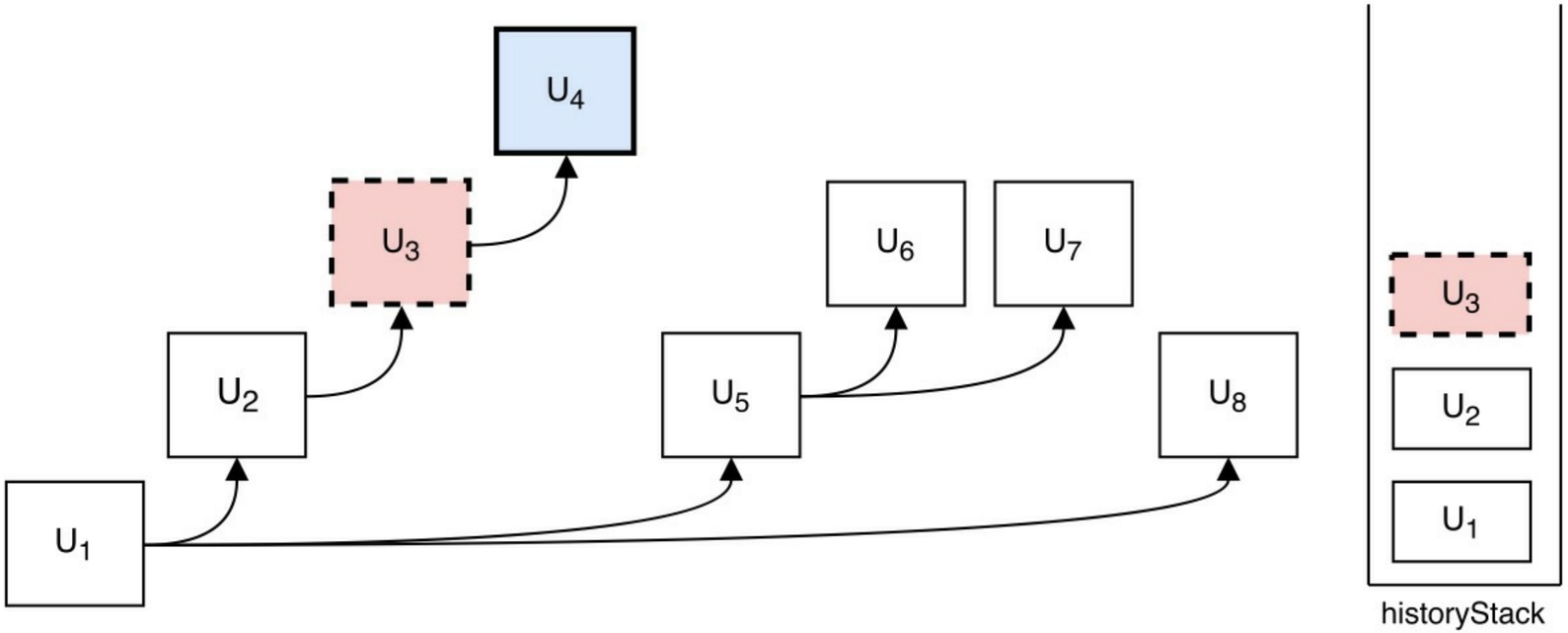
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Step 4: Push to history stack



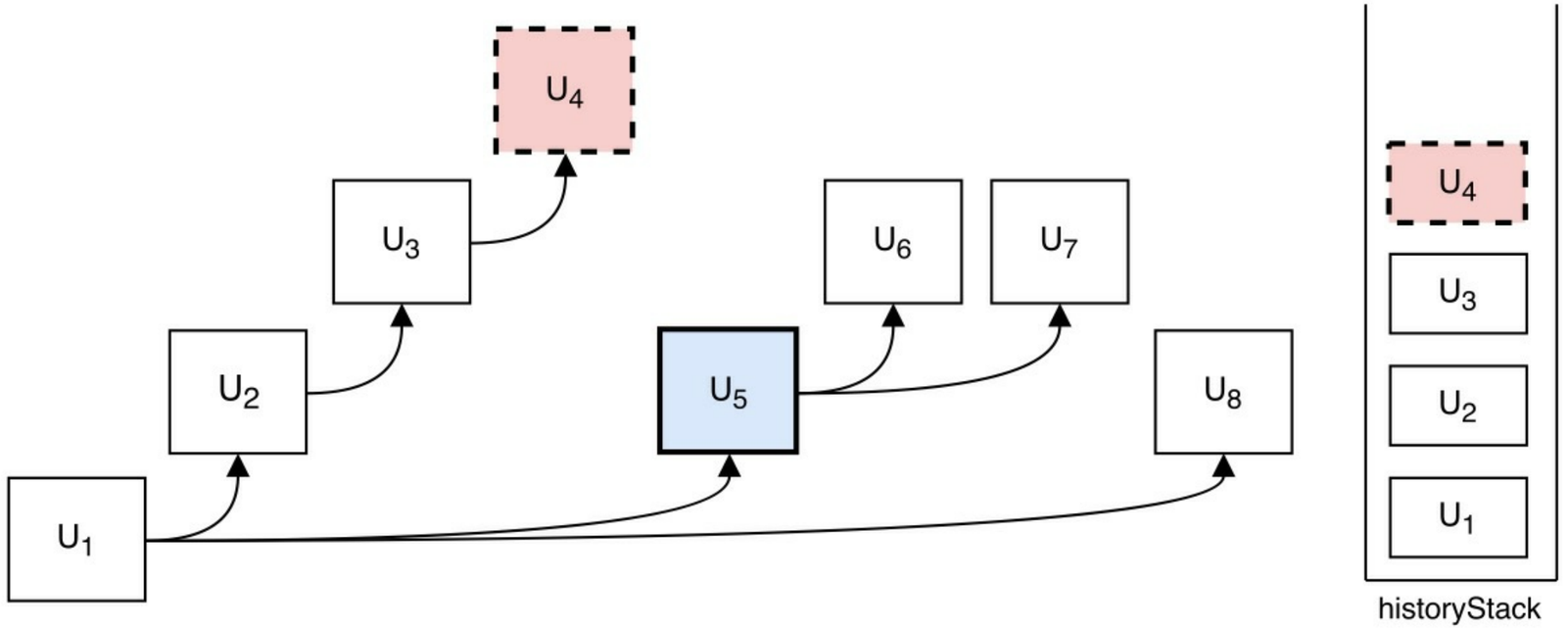
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Step 5: Stagnation!



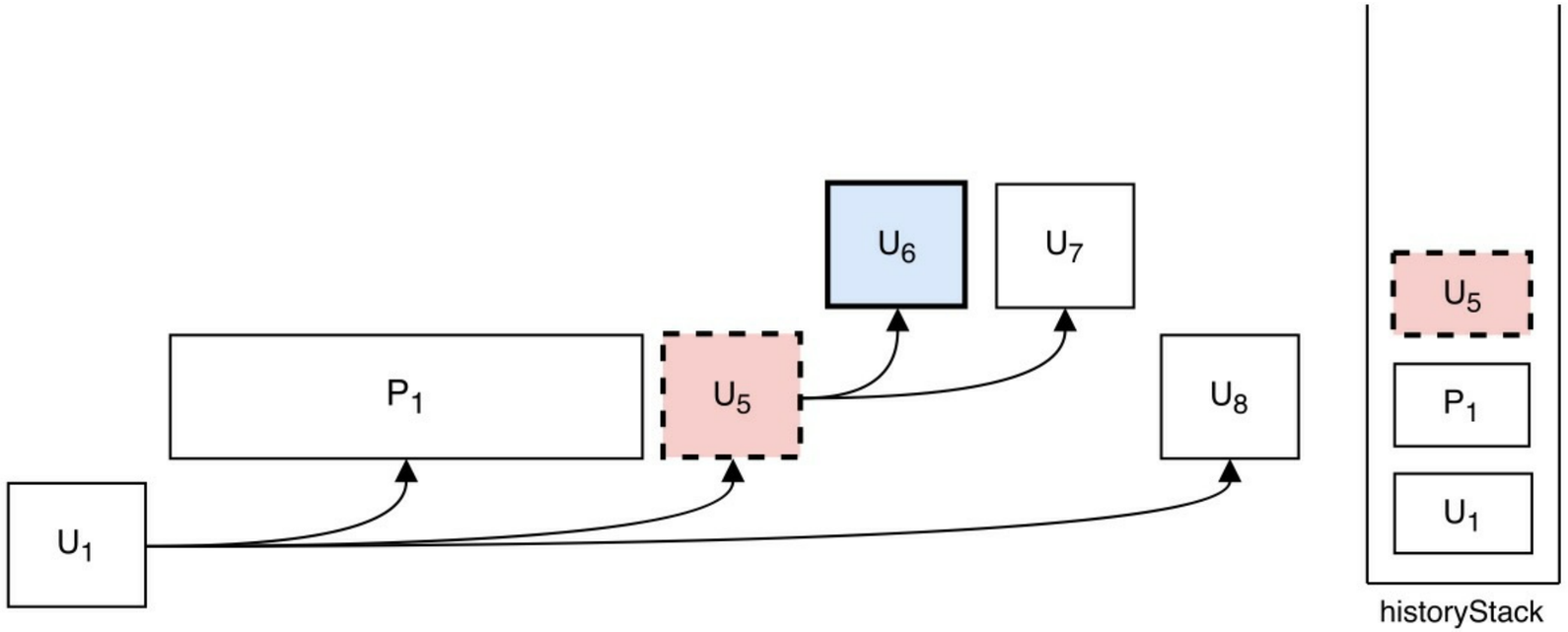
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Step 6: Push to history stack



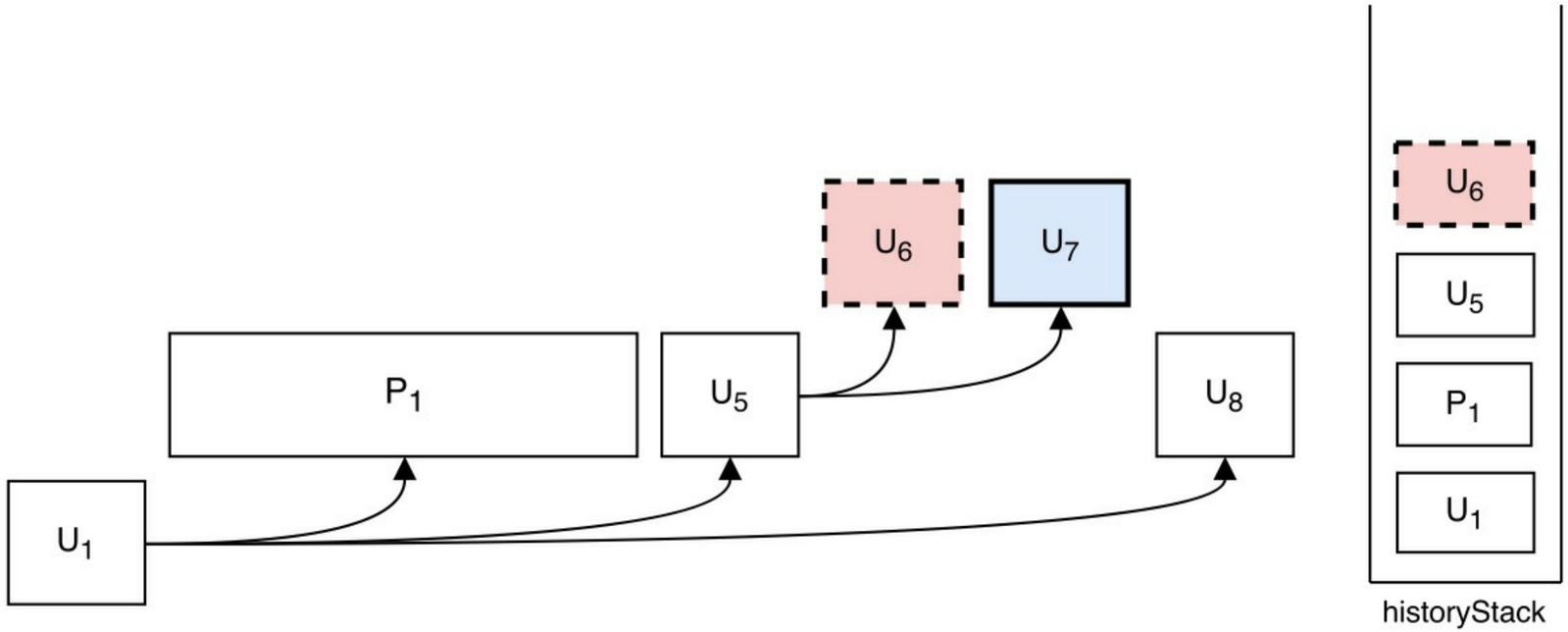
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Step 7: Stagnation!



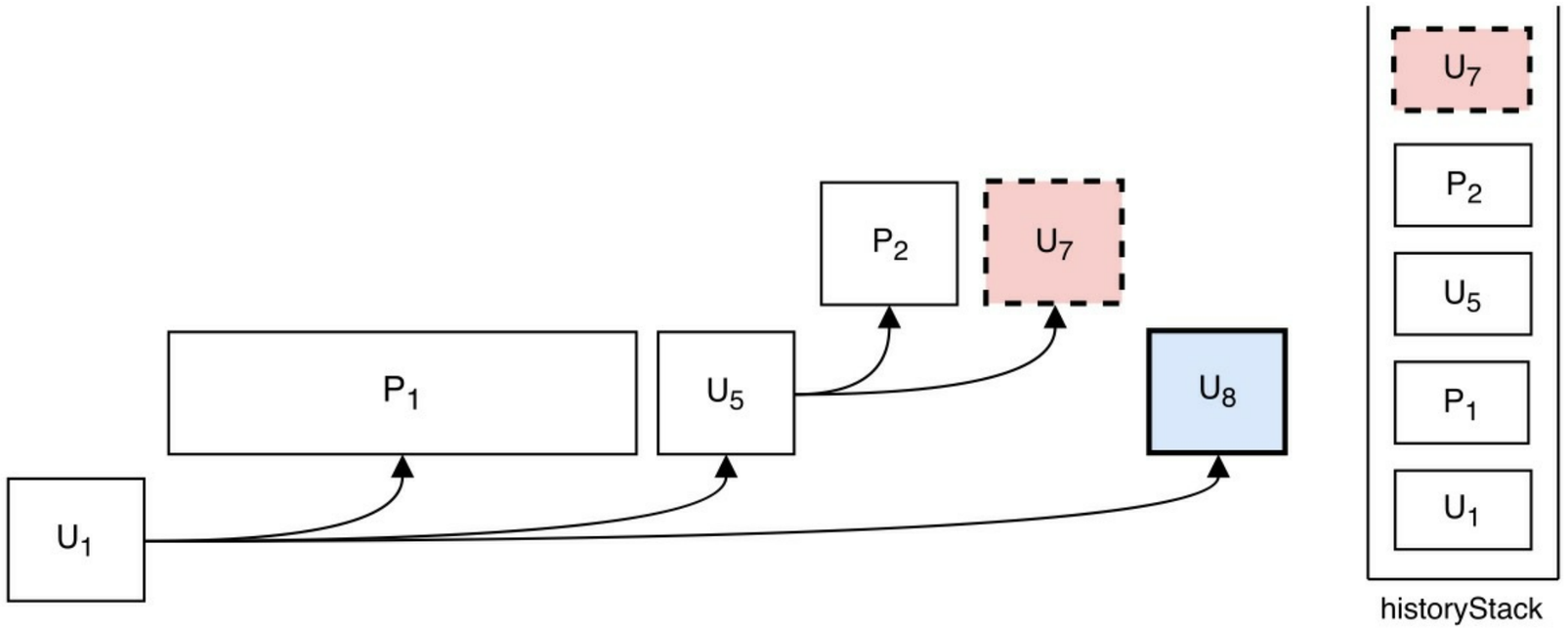
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Step 8: Stagnation!



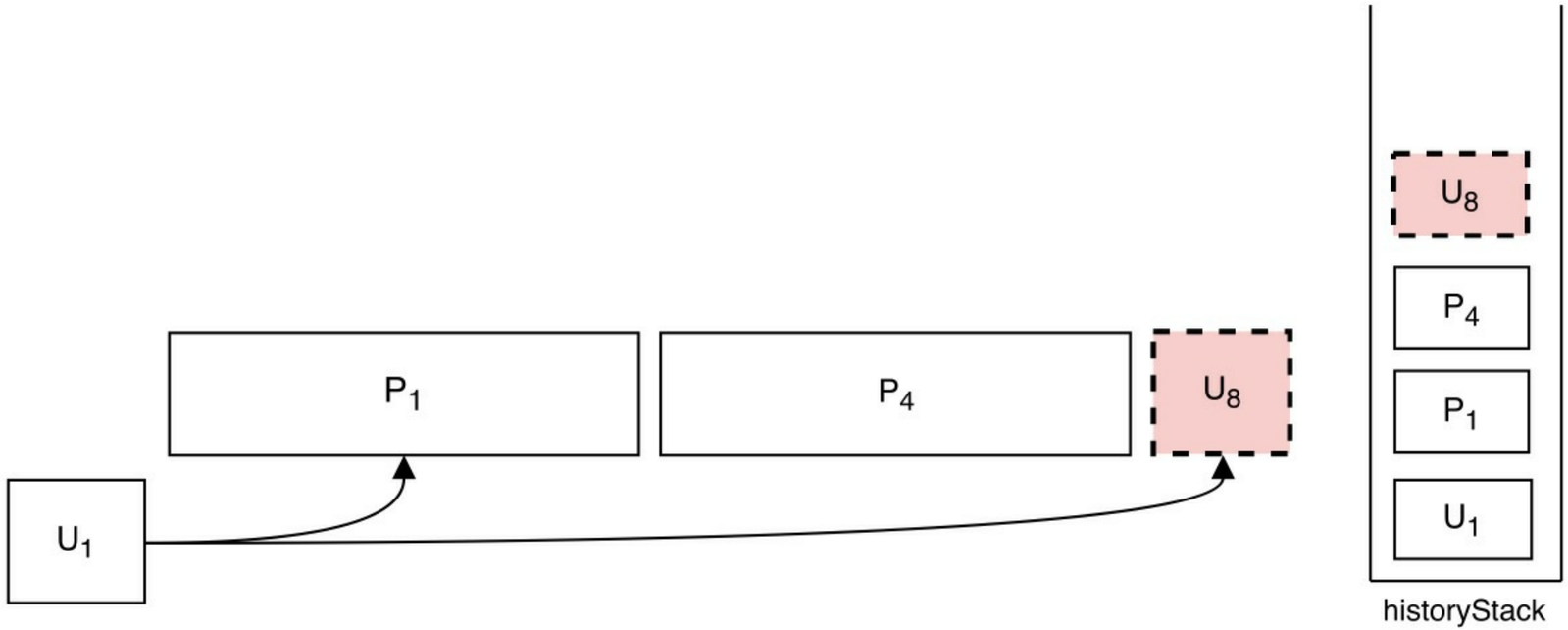
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Step 9: End of trace -> Stagnation!



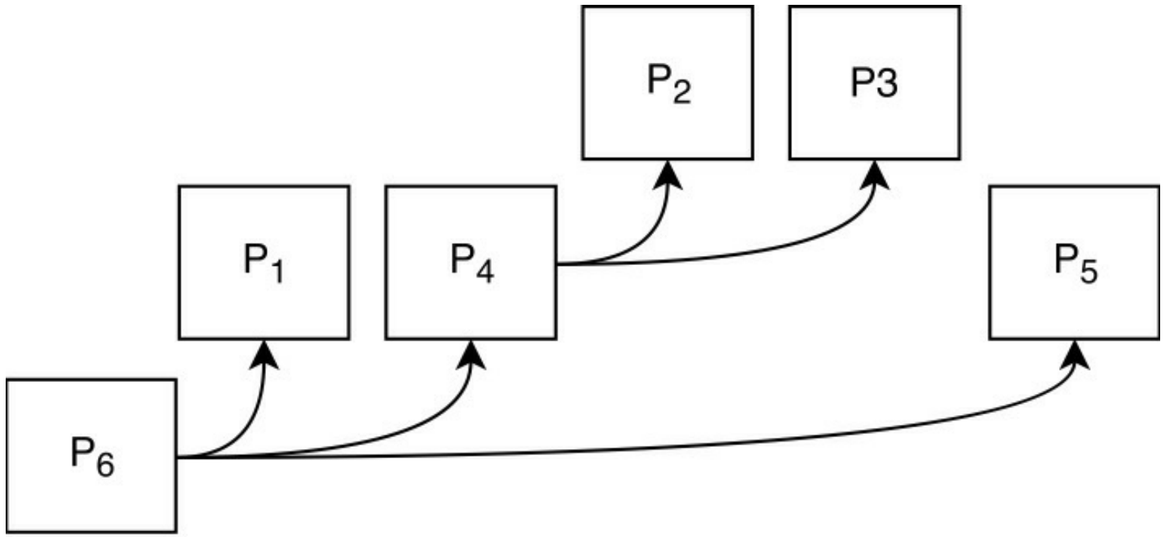
Sage

Data Collection

Phase Detection

- Algorithm
- Example

Finally: Preliminary Phase Tree



Sage

Data Collection

Phase Detection

Model Building

Goal: Train model to quickly abstract future execution traces.

- Input: A set of preliminary phase trees
- Model building consists of four steps:
  1. Duplicate detection
  2. Phase clustering (Agglomerative clustering)
  3. Frequent pattern mining
  4. Semantic labeling (using TF-IDF)
- Output: A model, which can be seen as a dictionary of recurrent phases.

Sage

Data Collection

Phase Detection

Model Building

Execution Abstraction

- At this moment we have:
  - an execution trace
  - a model containing the recurrent phases
- Abstracting an execution trace needs steps:
  1. Phase detection
  2. Duplicate detection
  3. Apply the model
  4. If necessary, reapply labeling on phases that are not in the model.

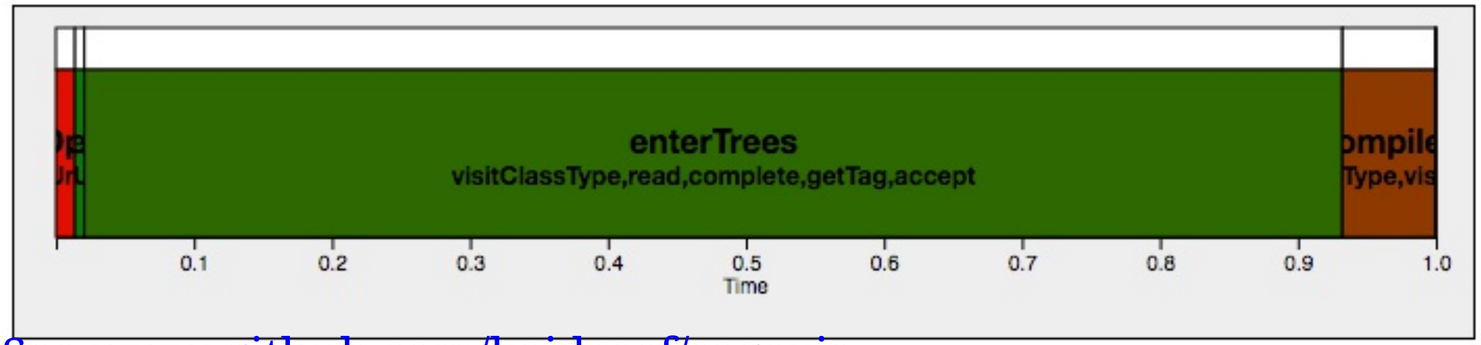
Hierarchical Phase Visualization

-- SageVis --

SageVis

Overview

- Goal: Visualize the phases in an execution trace in an intuitive way, so developers can explore them at different levels of granularity.
- Overview of the visualization:
  - Horizontal axis presents time.
  - Width of a phase represents how long the phase took.
  - Vertical axis could display multiple threads.
  - Depth: behind every timeline another timeline can be displayed with a lower level of granularity
  - Colors are used to help recognize similar phases.



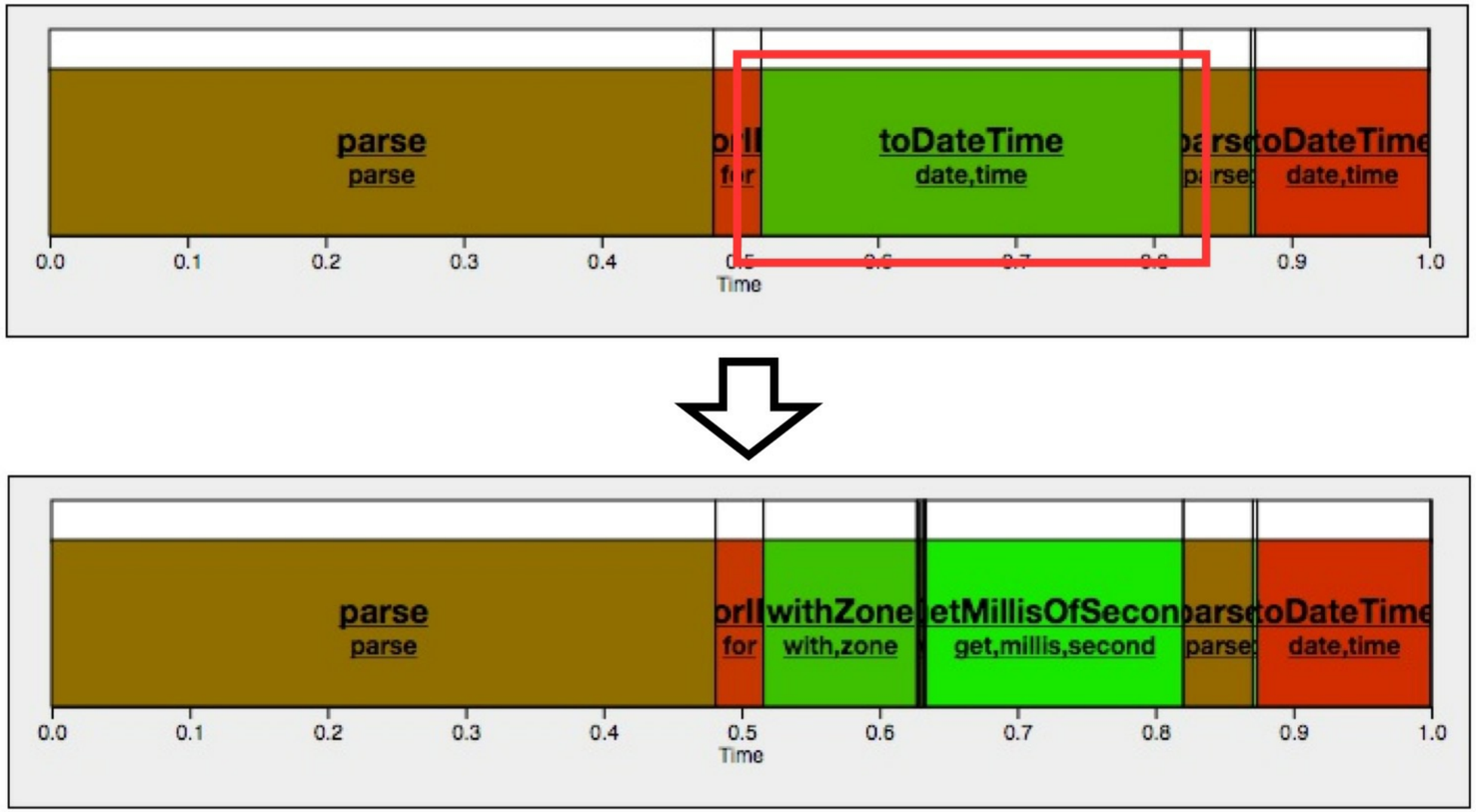
See www.github.com/kajdreef/sagevis

SageVis

Overview

Interactions

Expanding and Hiding Phases (1)



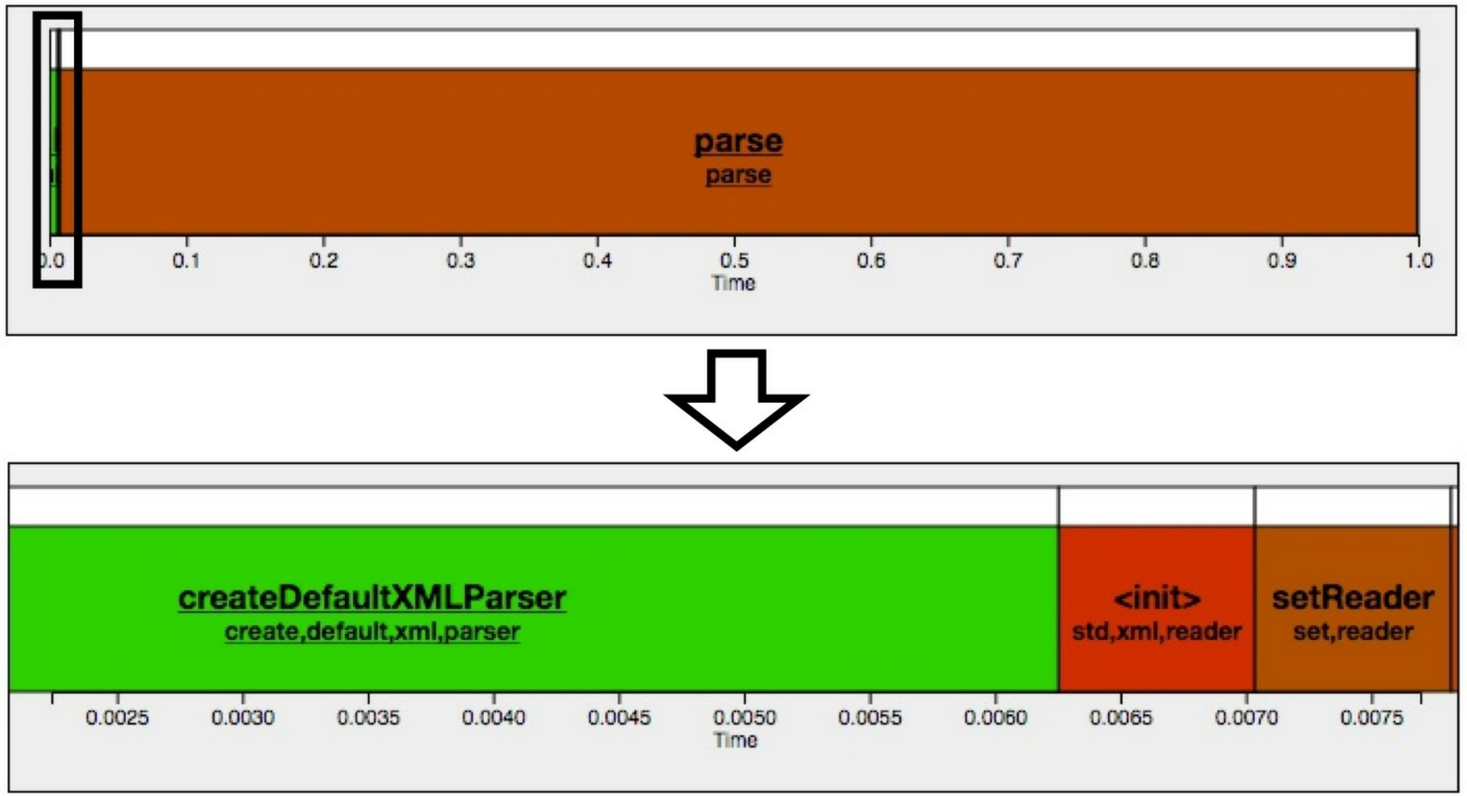
Expanding and Hiding Phases (2)

SageVis

Overview

Interactions

Zooming

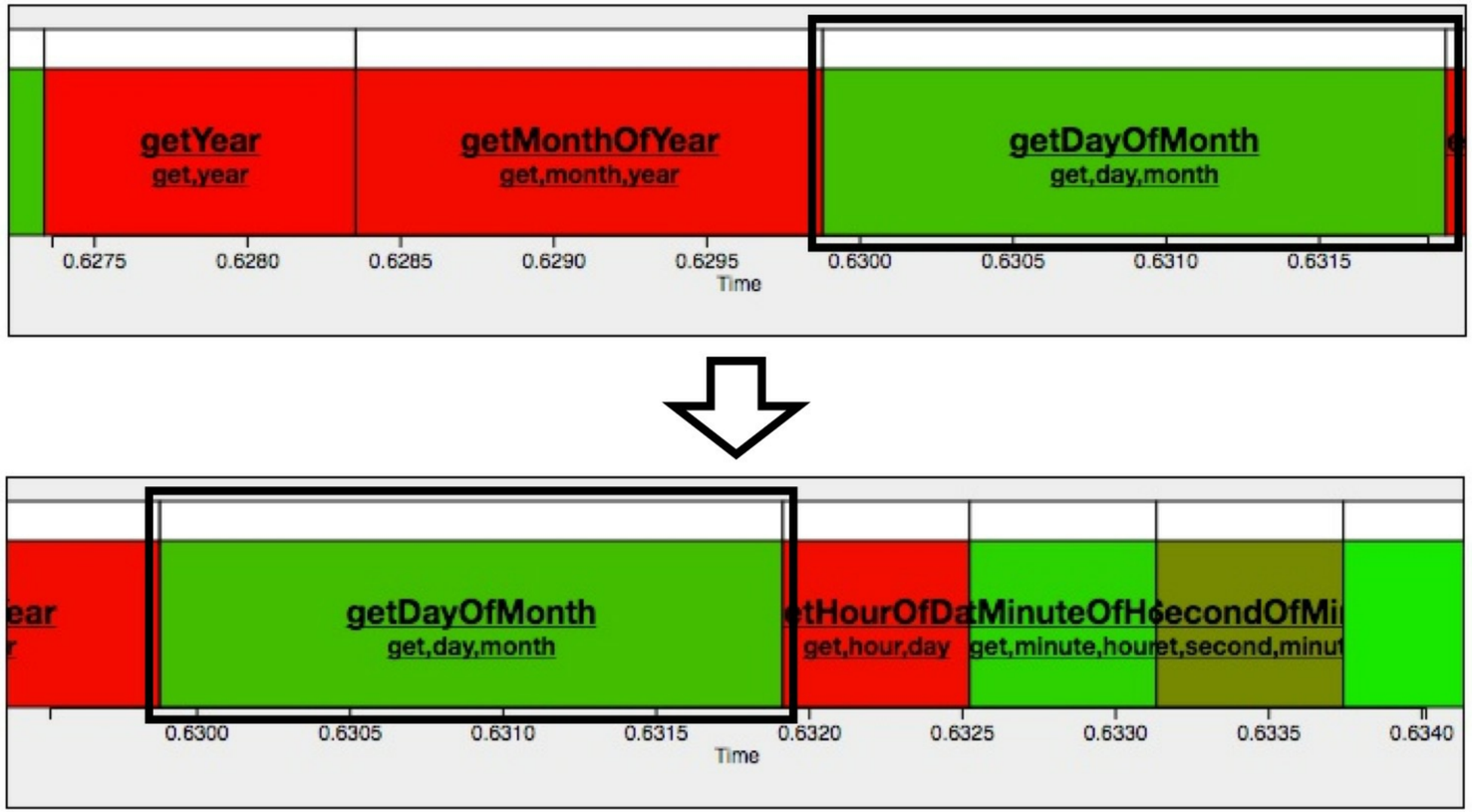


SageVis

Overview

Interactions

Panning

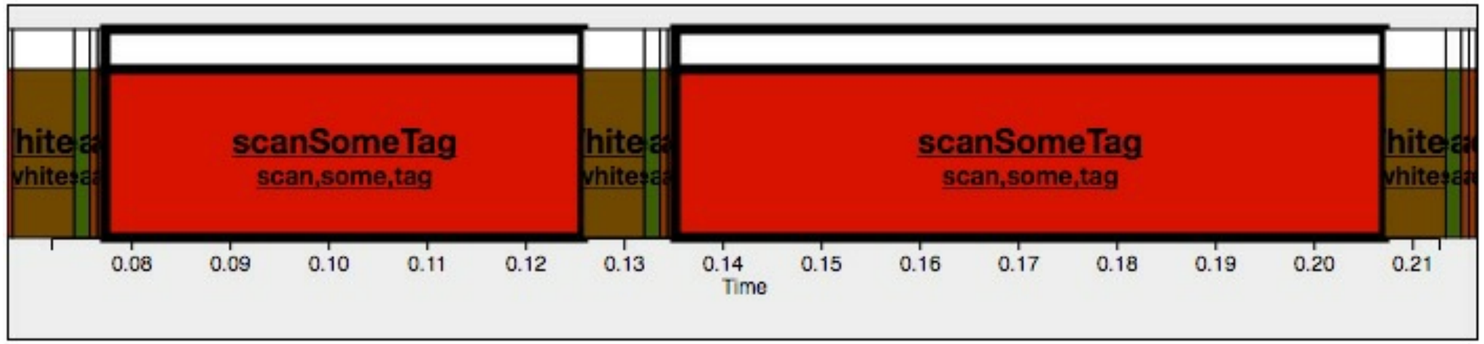


SageVis

Overview

Interactions

Phase Highlighting



Evaluation

Evaluation

Research Questions

RQ1: To what extent does SAGE alleviate the information overload challenge?

RQ2: Does the hierarchical phase abstraction provide substantially different levels of granularities of behavior?

RQ3: Are the derived labels sufficiently meaningful?

Evaluation

Research Questions

Methodology

1. Quantitative (RQ1 and RQ2):
    - Three projects: NanoXML, Javac, and Jedit.
        1. Determine the reduction in number of phases for each step of the process
        2. Determine number of phases at each level.

Evaluation

Research Questions

Methodology

1. Quantitative (RQ1 and RQ2):
    - Three projects: NanoXML, Javac, and Jedit.
        1. Determine the reduction in number of phases for each step of the process
        2. Determine number of phases at each level.
2. User study (RQ3):
    - One project: Jedit
    - Compute set of labels for 9 high-level behaviors
    - Let users determine the correct behavior based on the given labels

Evaluation

Research Questions

Methodology

1. Quantitative (RQ1 and RQ2):
    - Three projects: NanoXML, Javac, and Jedit.
        1. Determine the reduction in number of phases for each step of the process
        2. Determine number of phases at each level.
2. User study (RQ3):
    - One project: Jedit
    - Compute set of labels for 9 high-level behaviors
    - Let users determine the correct behavior based on the given labels
3. Case study (RQ1, RQ2, and RQ3):
    - Abstracted execution trace of Javac
    - Present how Sage and SageVis can help program comprehension.

Evaluation

Research Questions

Methodology

1. Quantitative

RQ1: To what extent does SAGE alleviate the information overload challenge?

Projects	|Traces|	|Events|	|Methods|	Phase Detection |phases|	Duplicate Detection |phases|	Clustering |phases|	Pattern Mining |phases|
NANOXML	235	247,471	100	225,430	786	160	47
JAVAC	19	22,439,965	3919	51,878	1605	650	57
JEDIT	18	10,129,771	5431	692,286	17,088	3838	84

Javac phase distribution:

Level	Average	Median	Max	Min	Std	Skewness	Kurtosis
1	2.0	2.0	2	2	0.0	0.0	−3.0
2	3.0	3.0	3	3	0.0	0.0	−3.0
3	3.0	3.0	3	3	0.0	0.0	−3.0
4	4.6	5.0	5	4	0.5	−0.3	−1.9
5	8.5	9.0	10	7	1.1	−0.4	−1.3
6	10.3	10.0	15	7	1.8	0.8	0.3
7	28.5	16.0	86	11	21.7	1.3	0.5
8	93.6	38.0	375	20	98.9	1.5	1.2

Evaluation

Research Questions

Methodology

1. Quantitative

RQ2: Does the hierarchical phase abstraction provide substantially different levels of granularities of behavior?

Javac phase distribution:

Level	Average	Median	Max	Min	Std	Skewness	Kurtosis
1	2.0	2.0	2	2	0.0	0.0	−3.0
2	3.0	3.0	3	3	0.0	0.0	−3.0
3	3.0	3.0	3	3	0.0	0.0	−3.0
4	4.6	5.0	5	4	0.5	−0.3	−1.9
5	8.5	9.0	10	7	1.1	−0.4	−1.3
6	10.3	10.0	15	7	1.8	0.8	0.3
7	28.5	16.0	86	11	21.7	1.3	0.5
8	93.6	38.0	375	20	98.9	1.5	1.2

Evaluation

RQ3: Are the derived labels sufficiently meaningful?

Functionality	$\|Handouts\|$	$\|Correct\|$	Accuracy
New File	56	46	0.82
Change Font	28	21	0.75
Word Count	56	28	0.50
Close&Exit	70	56	0.80
Keyword Highlight	28	11	0.39
Open File	28	22	0.79
Typing	70	52	0.74
Search	56	36	0.64
Indent	28	24	0.86
Total	420	296	0.70

Evaluation

Research Questions

Methodology

1. Quantitative

2. User Study

3. Case Study

Discussion

Discussion

1. Addressing Information Overload

- Execution traces contain millions of events
- Sage output in the tens of phases

Discussion

1. Addressing Information Overload

- Execution traces contain millions of events
- Sage output in the tens of phases

2. Addressing Behavior Subsumption

- Sage output provides multiple levels of granularity

Discussion

1. Addressing Information Overload

- Execution traces contain millions of events
- Sage output in the tens of phases

2. Addressing Behavior Subsumption

- Sage output provides multiple levels of granularity

3. Addressing Comprehensibility of Execution Traces

- Average behavior identification accuracy of 70%

Conclusion

Conclusion

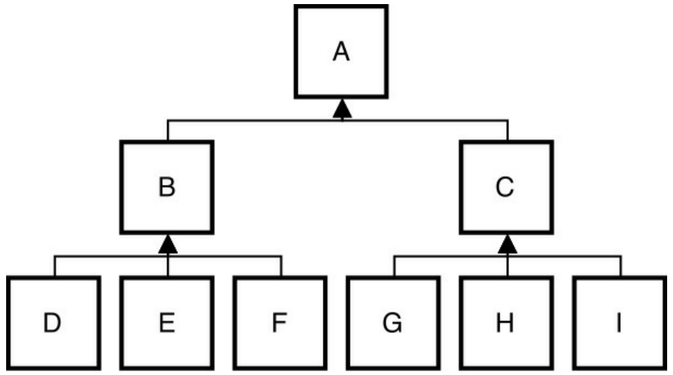
- We introduced:
  - Sage: Hierarchically abstracts an execution trace.
  - SageVis: Interactive visualization of the abstracted execution trace.
- The quantitative evaluation revealed us that Sage can successfully abstract the execution trace and reduce the amount of data to be inspected.
- The user study showed us that for most functionalities the users were able to achieve a 70% accuracy.
- The case study demonstrates Sage ability to reveal the primary behavior phases within a large execution.

Extra Slides!

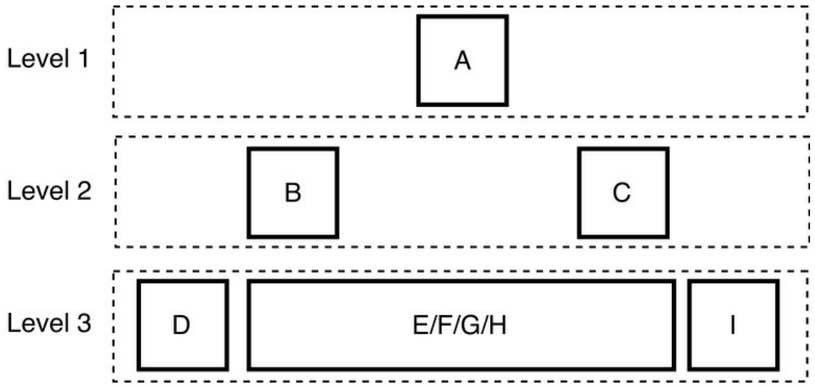
SageVis

Data Structure

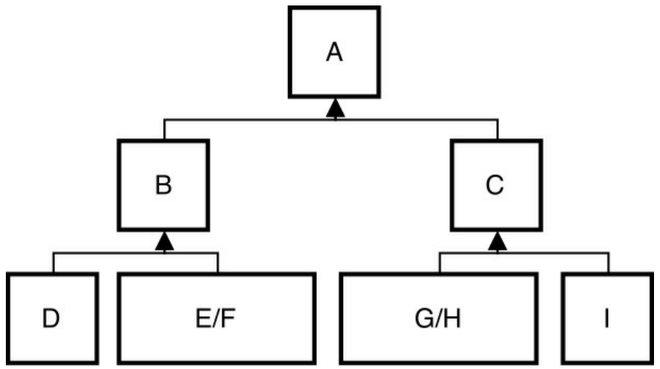
Only Phase Detection



FPM



FPM with boundaries



Evaluation

Computational Efficiency

RQ4: What is the computational efficiency of each abstraction step of Sage?

Size abstraction for each step of the training procedure in seconds

Projects	Phase Detection	Duplicate Detection	Clustering	Pattern Mining	Labeling
NANOXML	3.5	0.1	0.5	50.8	1.7
JAVAC	247.8	0.0	20.1	428.9	1.0
JEDIT	143.6	0.4	1994.3	5846.0	20.7

Abstraction time cost in seconds

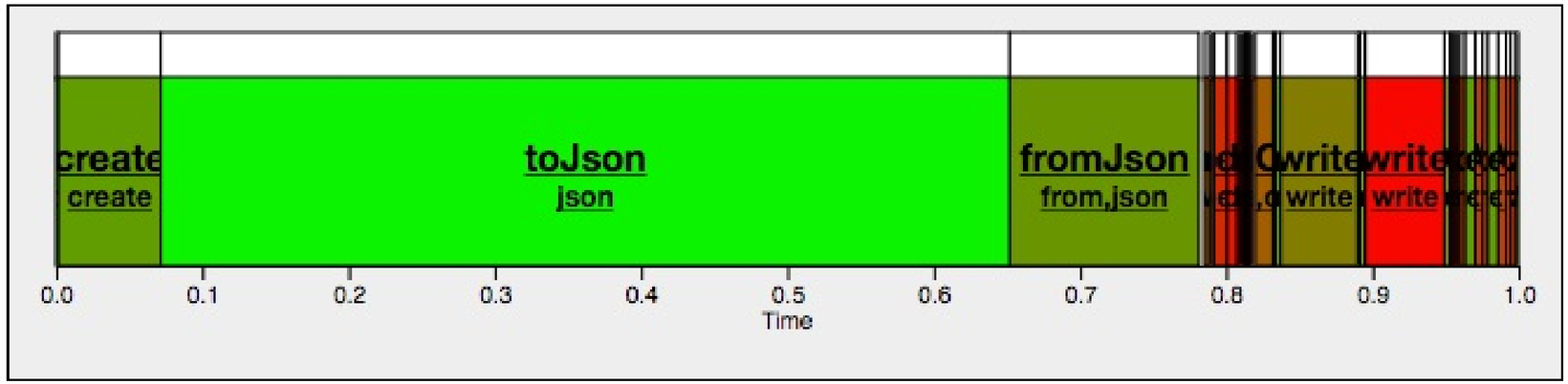
Project	time
NANOXML	0 s
JAVAC	13 s
JEDIT	9.1s

Visualization Comparison

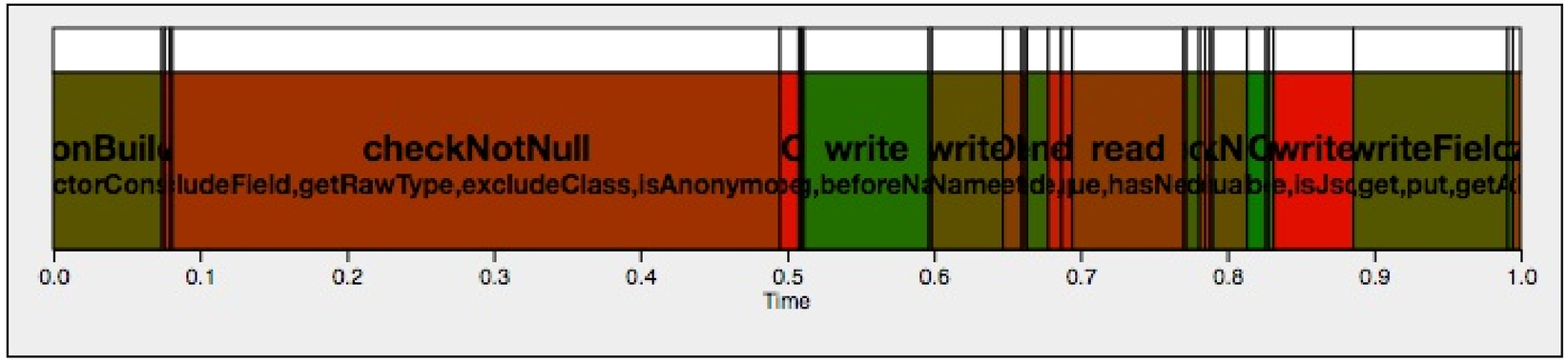
1. Navigation

RQ5: What is the impact of frequent pattern mining on the visualization?

Visualization without frequent pattern mining



Visualization with frequent pattern mining



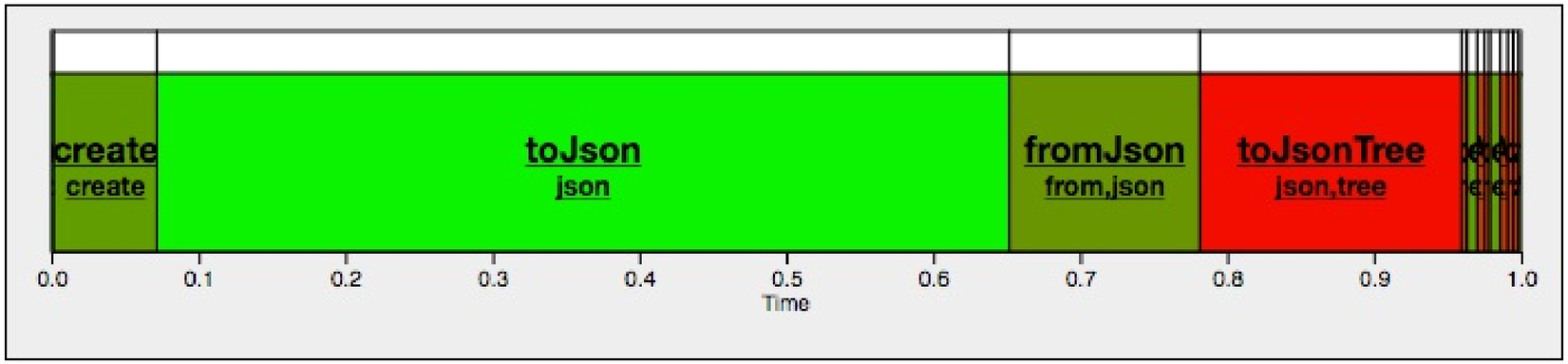
Visualization Comparison

1. Navigation

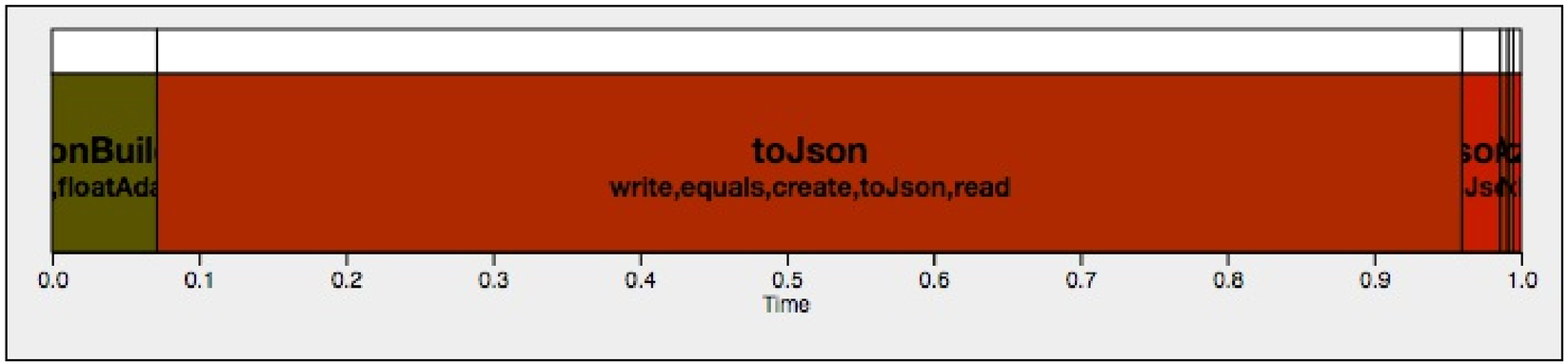
2. Behavior Detection

RQ5: What is the impact of frequent pattern mining on the visualization?

Visualization without frequent pattern mining



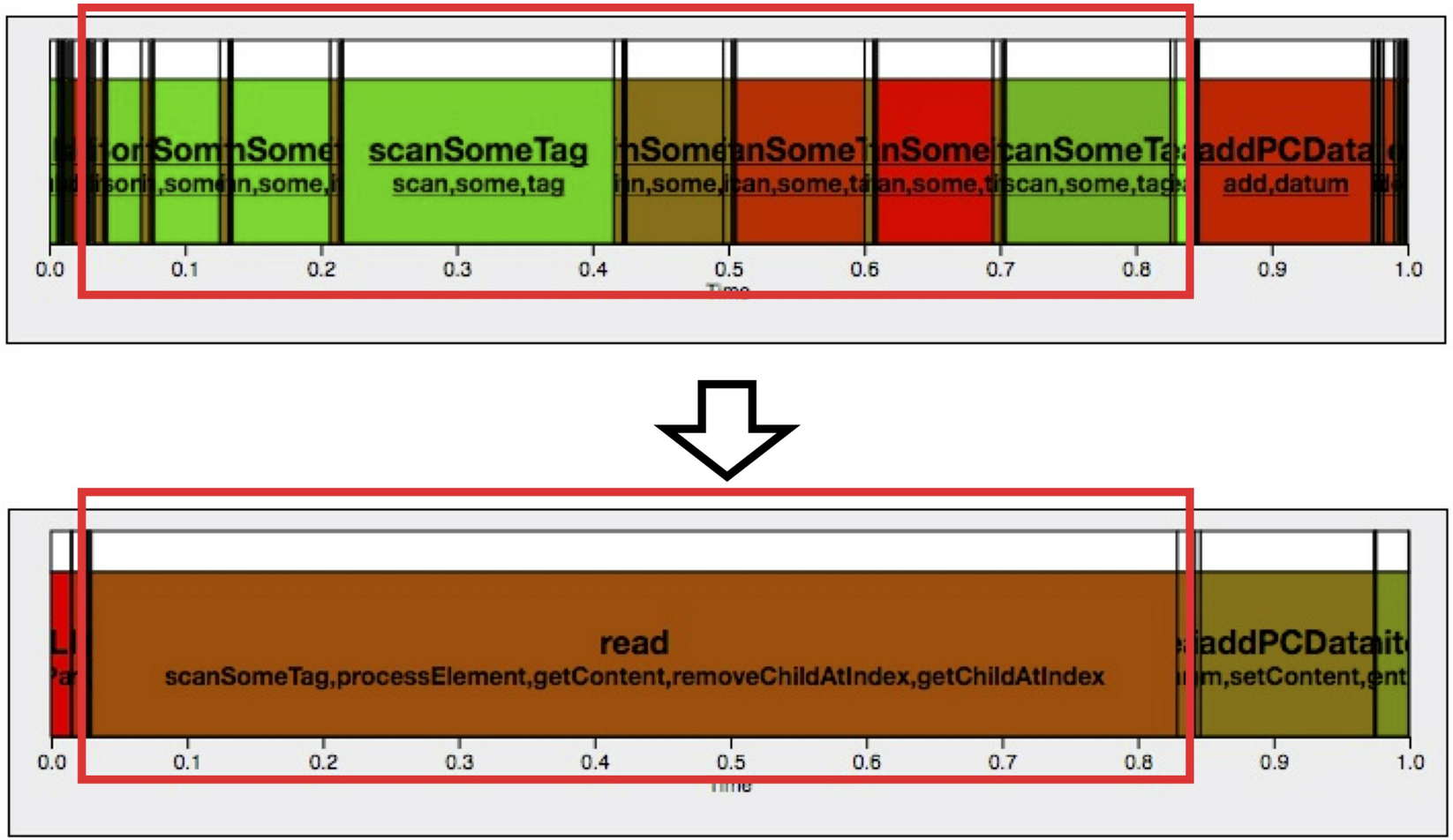
Visualization with frequent pattern mining



Visualization Comparison

1. Navigation
2. Behavior Detection
3. Visualizing Patterns

RQ5: What is the impact of frequent pattern mining on the visualization?



Future work

- Overall

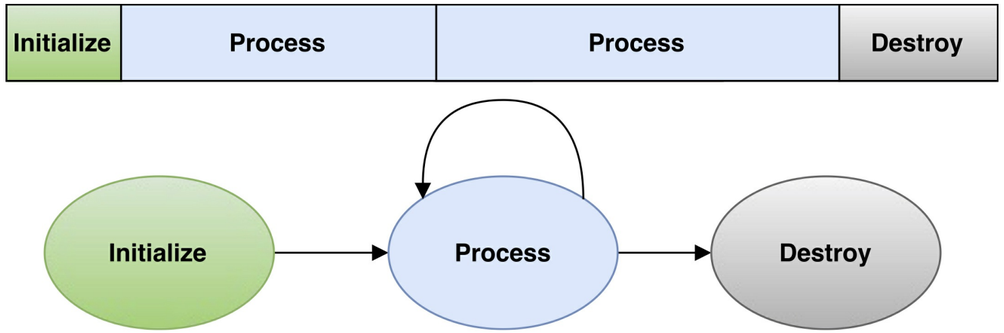
- Multithreaded support
- Evaluate Sage and SageVis on larger systems

- Visualization

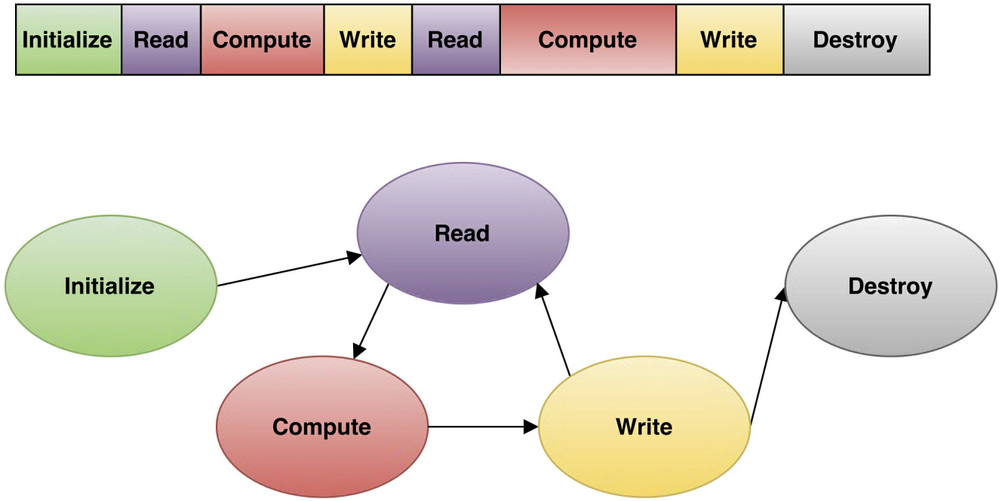
- Dynamically Inferred State Machine

Dynamically Inferred State Machine

Details hidden away

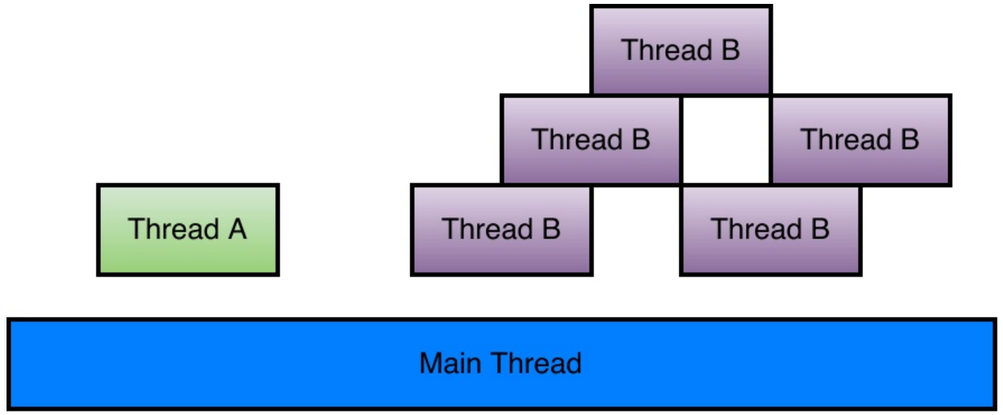


Unfold the details

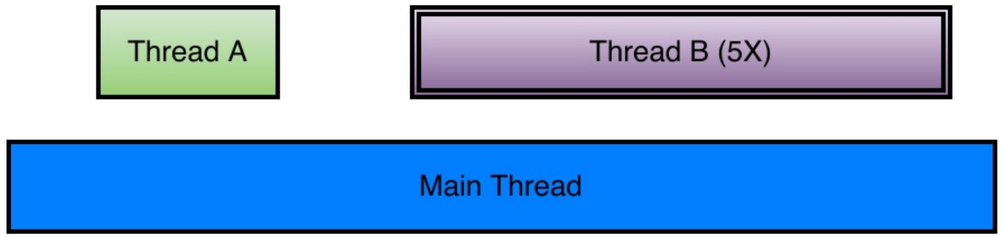


Multithreaded Support

Normal View



Fold similar concurrent threads into each other

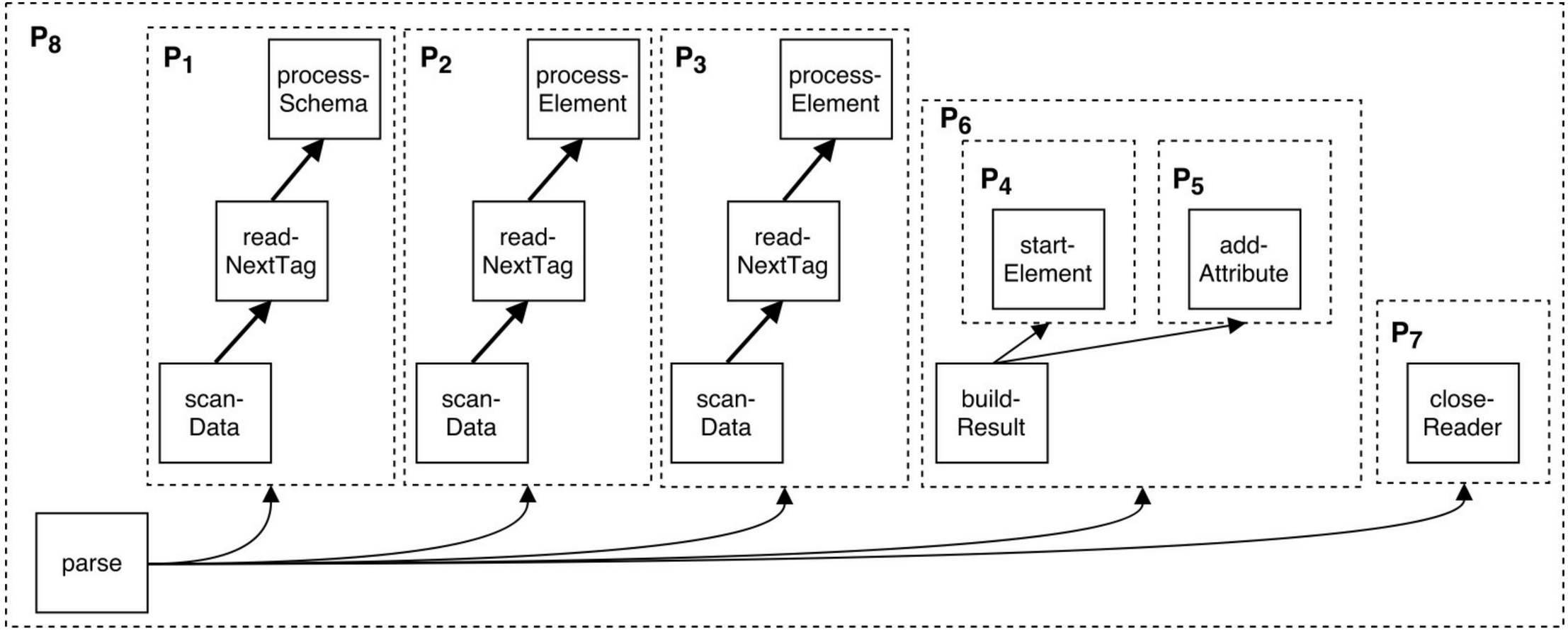


Algorithm: Find Phases in History

Input : *endEvent*: End event
historyStack: A stack which contains previously executed events and/or phases
Output: *root*: List of sub-phases
Initialization: Initialize an empty list *children*.
prevStack = *endEvent*

```
1  begin
2      while historyStack.size() > 0 do
3          topStack ← historyStack.pop()
4          if topStack.getCallDepth() == prevStack.getCallDepth() then
5              p ← createNode(prevStack, children)
6              children.append(topStack)
7              children.append(p)
8          else if topStack.getCallDepth() == endEvent.getCallDepth() then
9              children.append(topStack)
10             node ← createNode(topStack, children)
11             historyStack.push(node)
12             historyStack.push(endEvent)
13             return None
14         else
15             children.append(topStack)
16         end
17         prevStack ← topStack
18     end
19     root ← createNode(topStack, children)
20     return root
21 end
```

Example: Duplicate Detection and Clustering



NanoXML distribution

Level	Average	Median	Max	Min	Std	Skewness	Kurtosis
1	3.5	3.0	20	1	2.6	4.9	27.6
2	5.6	5.0	32	1	4.6	3.4	16.2
3	12.0	6.0	38	1	10.8	0.7	−1.0
4	12.5	6.0	42	1	11.7	0.8	−0.8
5	19.4	15.0	90	2	15.6	1.4	2.7
6	23.8	19.0	140	2	19.1	2.6	13.0
7	30.3	24.0	269	2	32.5	5.1	34.3
8	43.5	41.5	414	3	50.4	4.6	28.7

JEdit distribution

Level	Average	Median	Max	Min	Std	Skewness	Kurtosis
1	19.7	16.0	88	6	17.5	3.2	9.9
2	109.6	96.0	252	25	66.0	0.7	−0.6
3	481.2	381.0	1695	206	359.9	2.4	5.0
4	834.8	723.5	2338	424	457.1	2.1	4.2
5	1599.8	1319.5	4375	731	865.5	2.2	3.9
6	2633.3	2248.0	6302	1153	1200.8	1.6	2.4
7	4642.2	4913.5	8340	2200	1646.7	0.3	−0.5
8	6955.8	7071.5	11,804	2673	2741.9	0.1	−0.9