Efficient JavaScript Mutation Testing

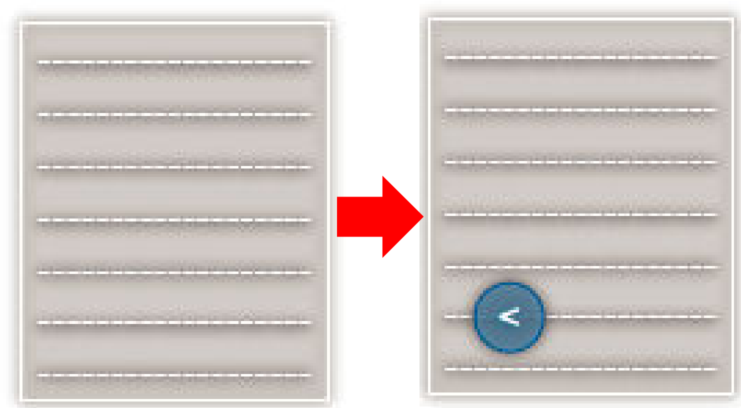
Shabnam Mirshokraie
Ali Mesbah
Karthik Pattabiraman

University of British Columbia

Mutation Testing

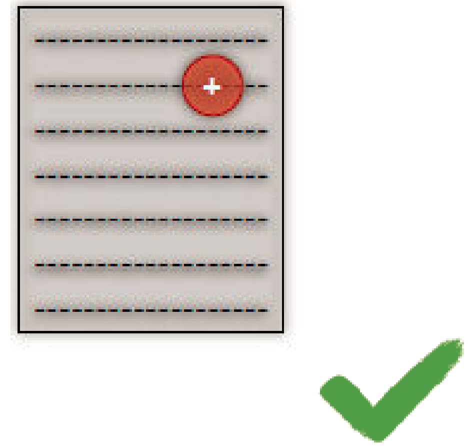
- A fault-based technique to assess and improve the quality of a test suite

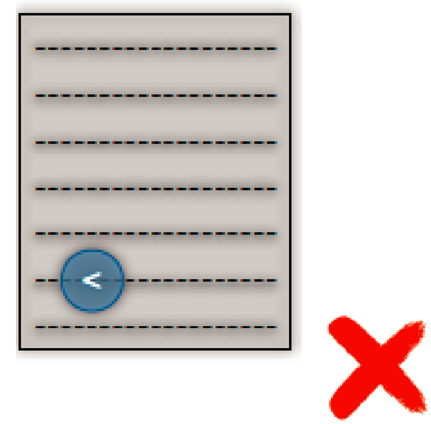
- Creates modified versions of the program



Original Program **Mutant**

The number of killed mutants by a test suite as a measure of its effectiveness





Mutation Testing Challenges

- High computational cost

- Equivalent mutants
  - Syntactically different but semantically identical to the original program
  - 10 to 40 percent of mutants are equivalent

Prior Approaches

- Evolutionary techniques (GECCO'04, IST'11)
- Impact on the application's expected behaviour (ISSTA'09, ICST'10)

- First generates mutants and then examins for equivalency
  - Computationally expensive and inefficient

Our Main Goal:

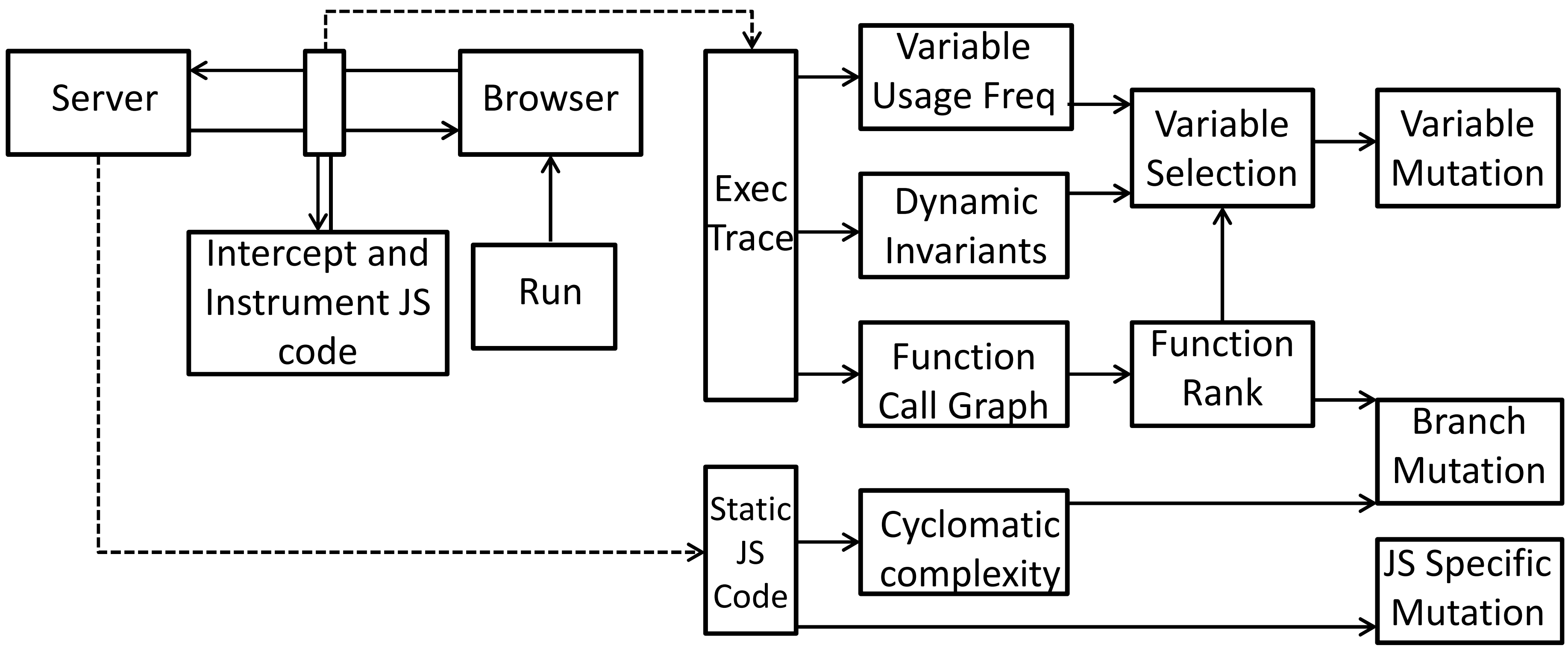
Avoid generating equivalent mutants

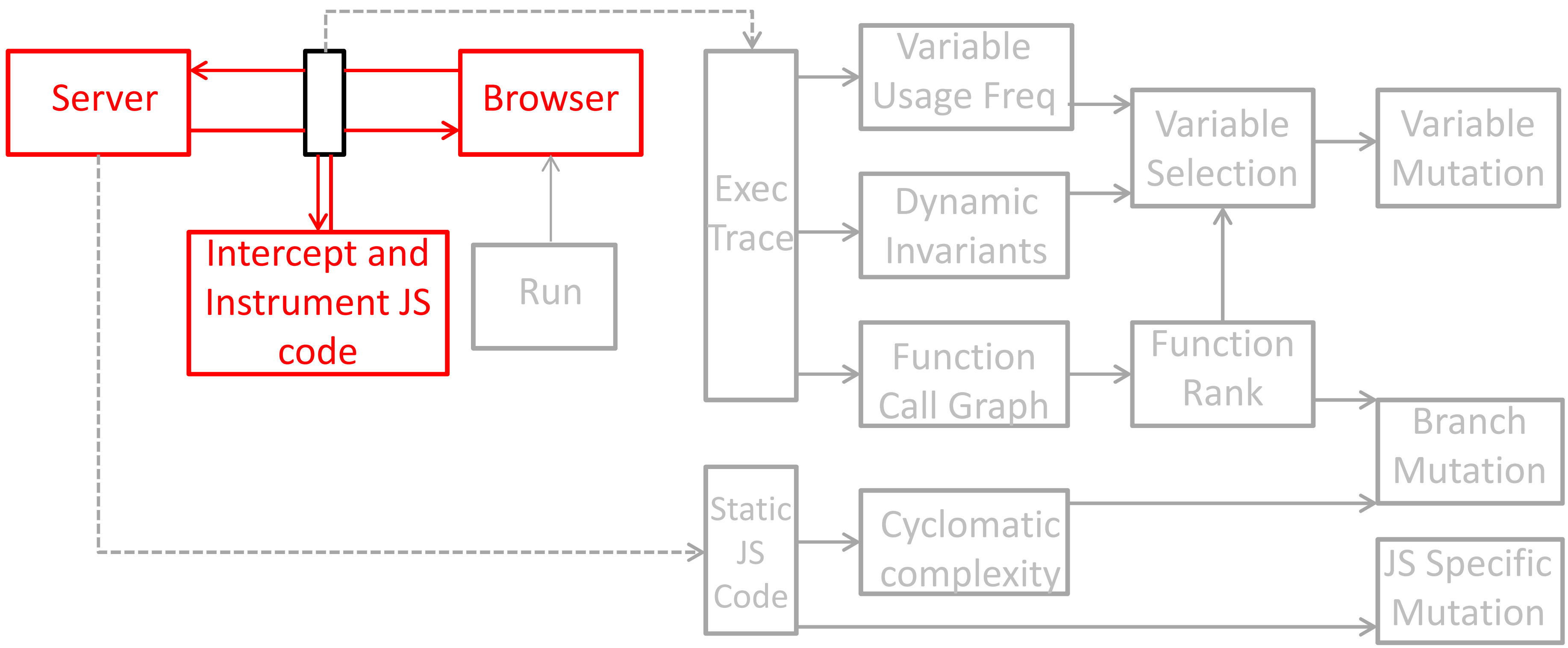
How? Narrows down the scope of the mutation process to behavioural affecting parts of the code that

Our Approach

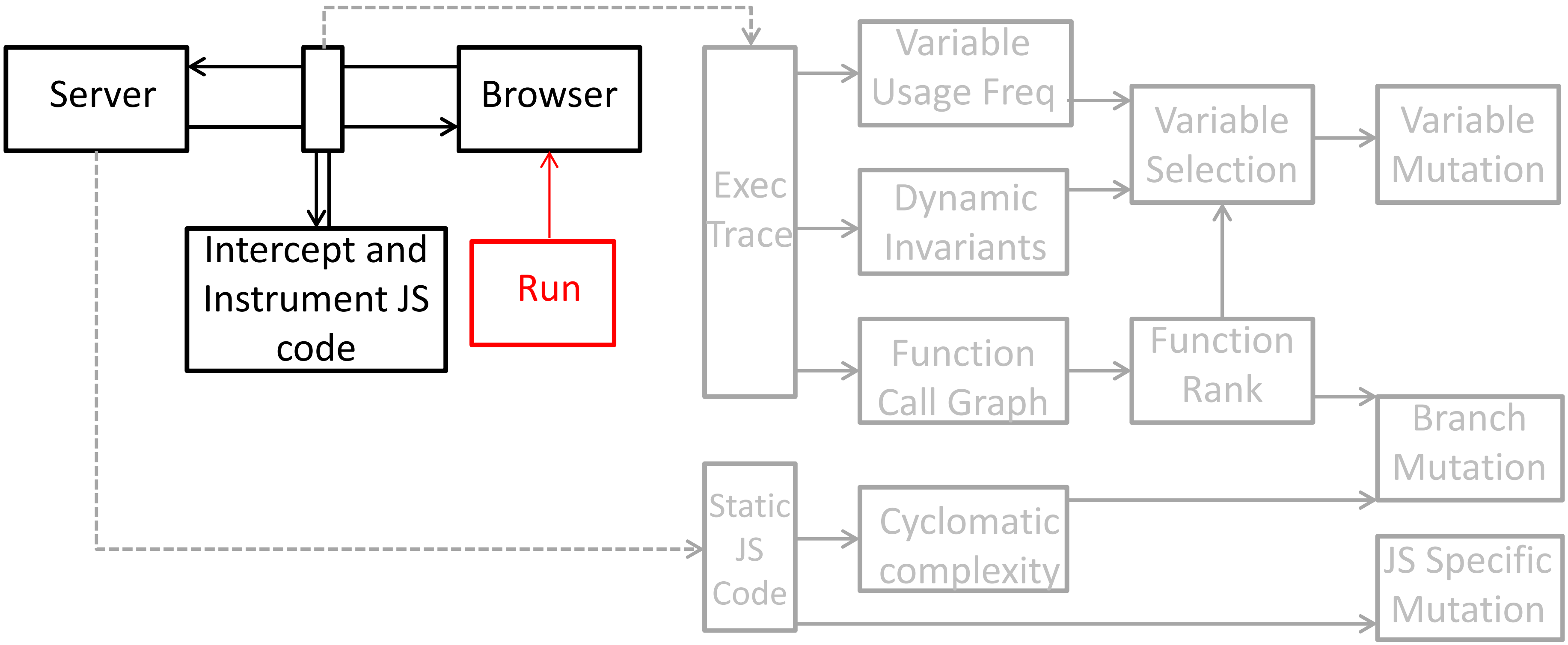
- Static and dynamic analysis

- A generic mutation testing that guides the mutation generation process:
  - Fewer but more effective mutations
  - Mutations with clear impact on the program's behaviour


Server
Browser
Intercept and Instrument JS code
Run
Exec Trace
Static JS Code
Variable Usage Freq
Dynamic Invariants
Function Call Graph
Cyclomatic complexity
Variable Selection
Function Rank
Variable Mutation
Branch Mutation
JS Specific Mutation

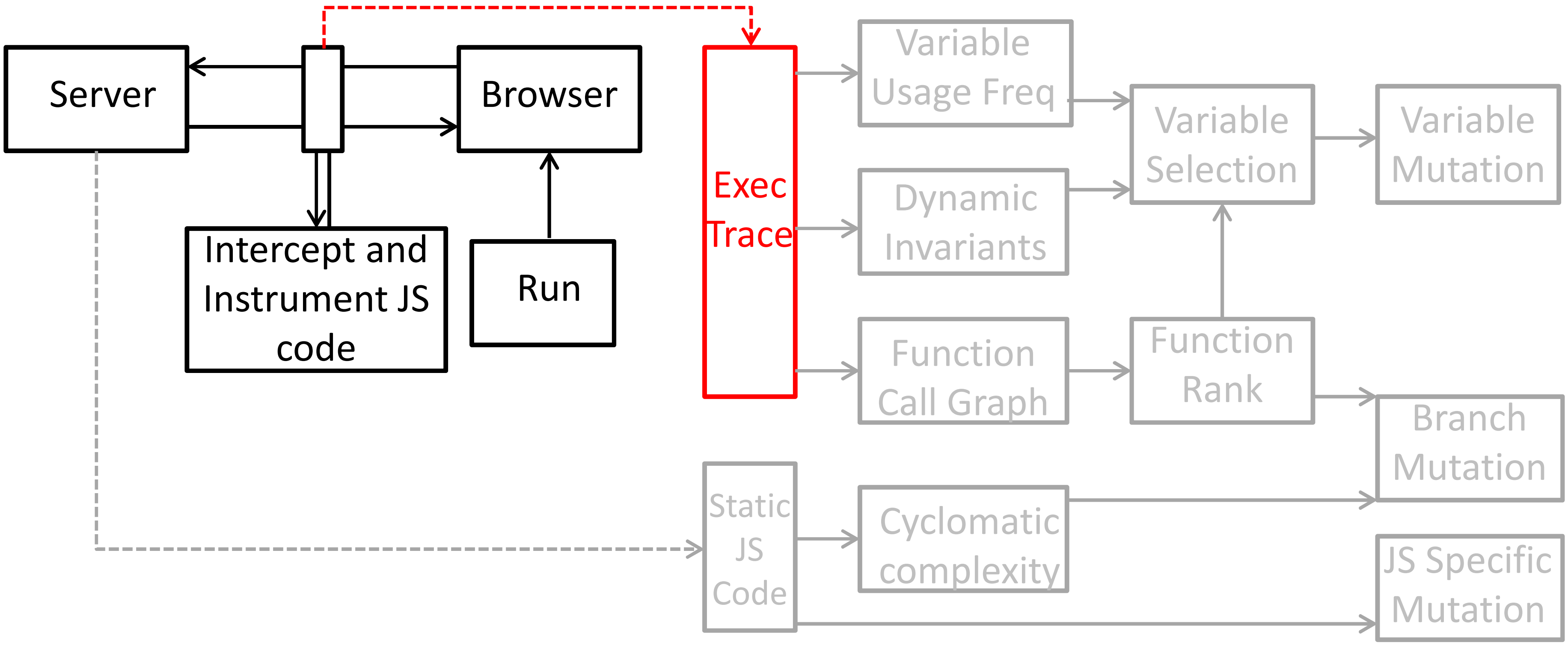


Intercepts the JavaScript code by setting up a proxy and instrumenting the code

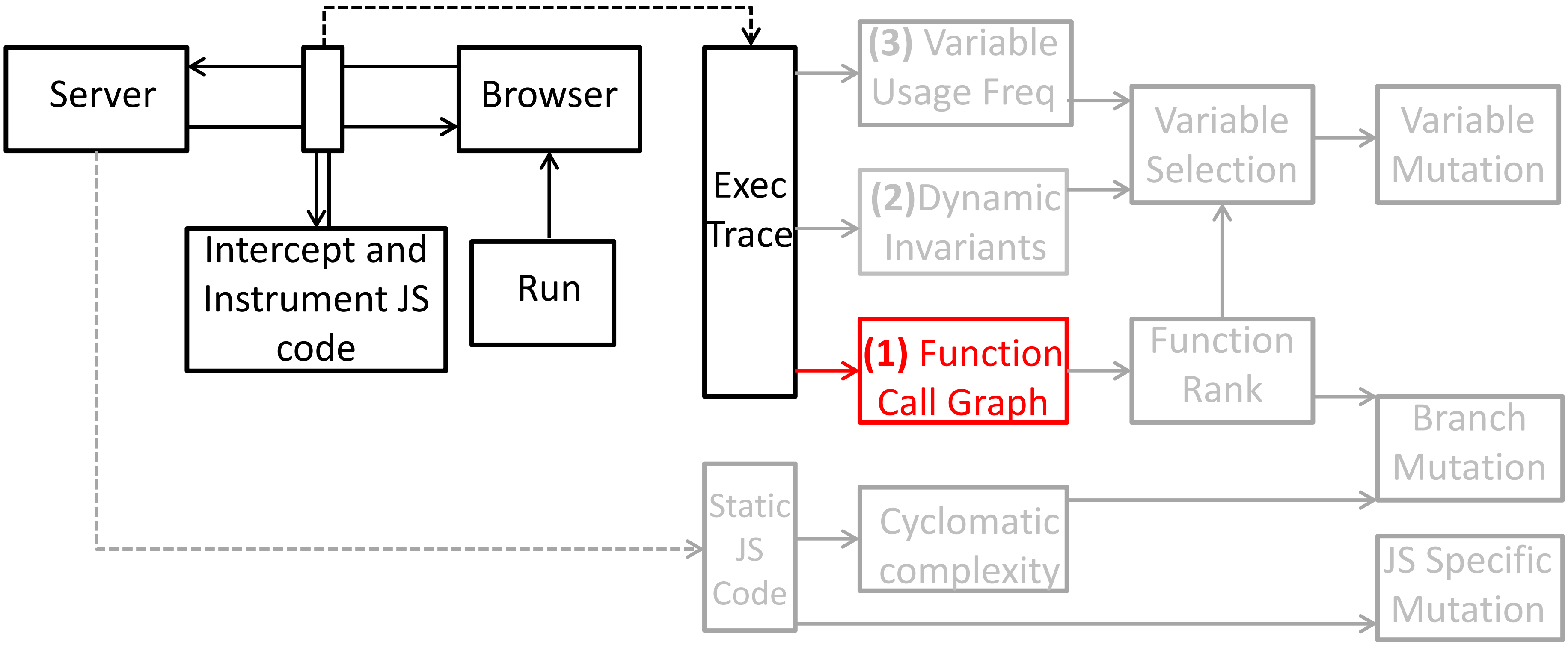


Executes the instrumented program by:

- Crawling the application automatically
- Running the existing test suite
- Combination of the two

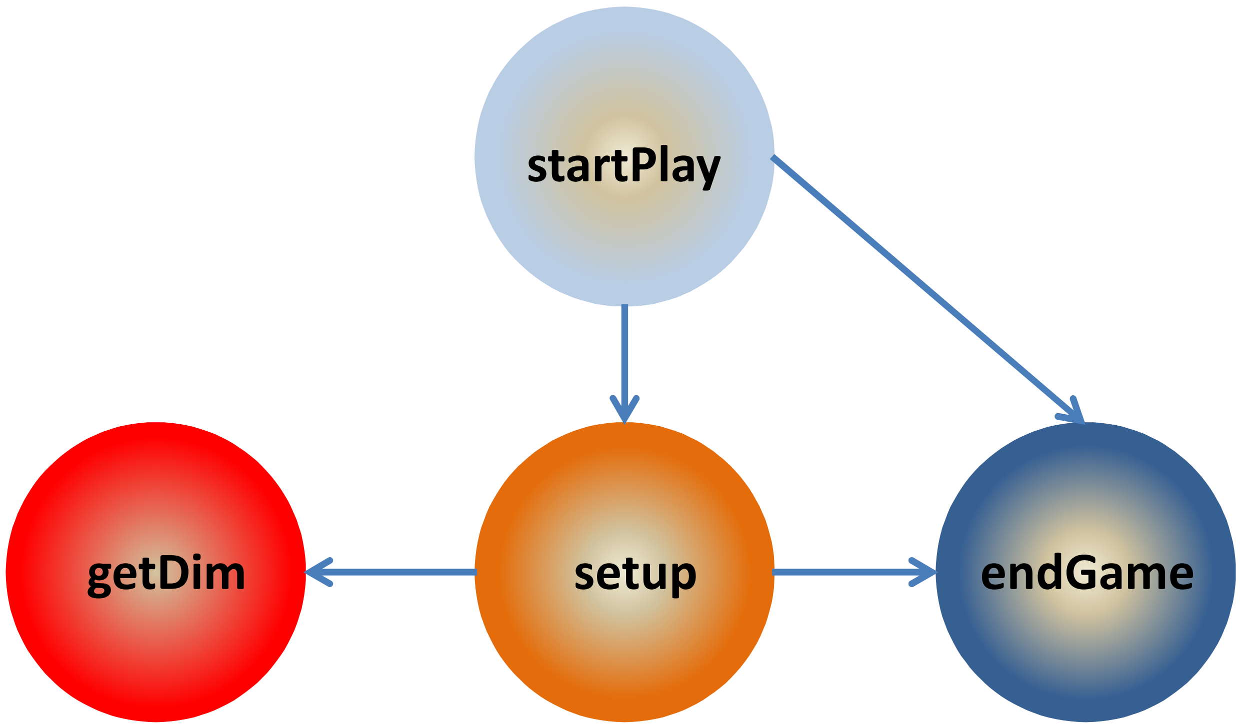


Gathers detailed execution traces of the application under test

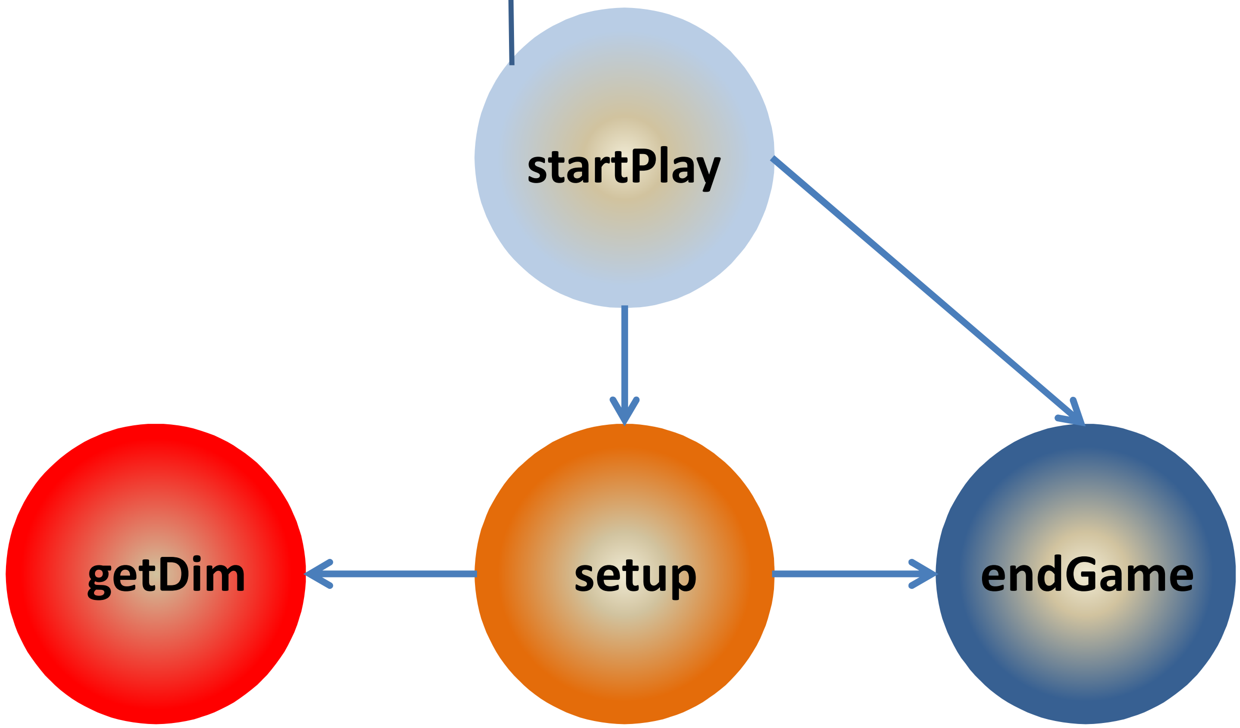


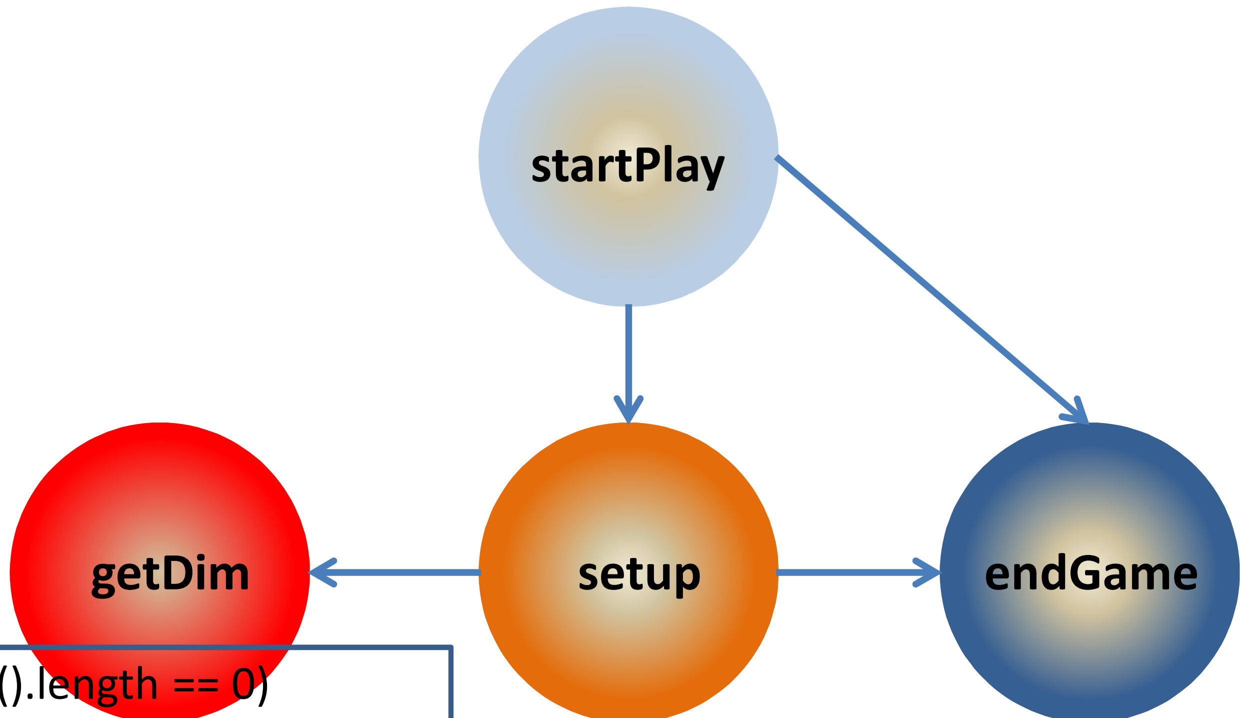
Information extraction from the execution traces:

(1) Dynamic call graph of the application

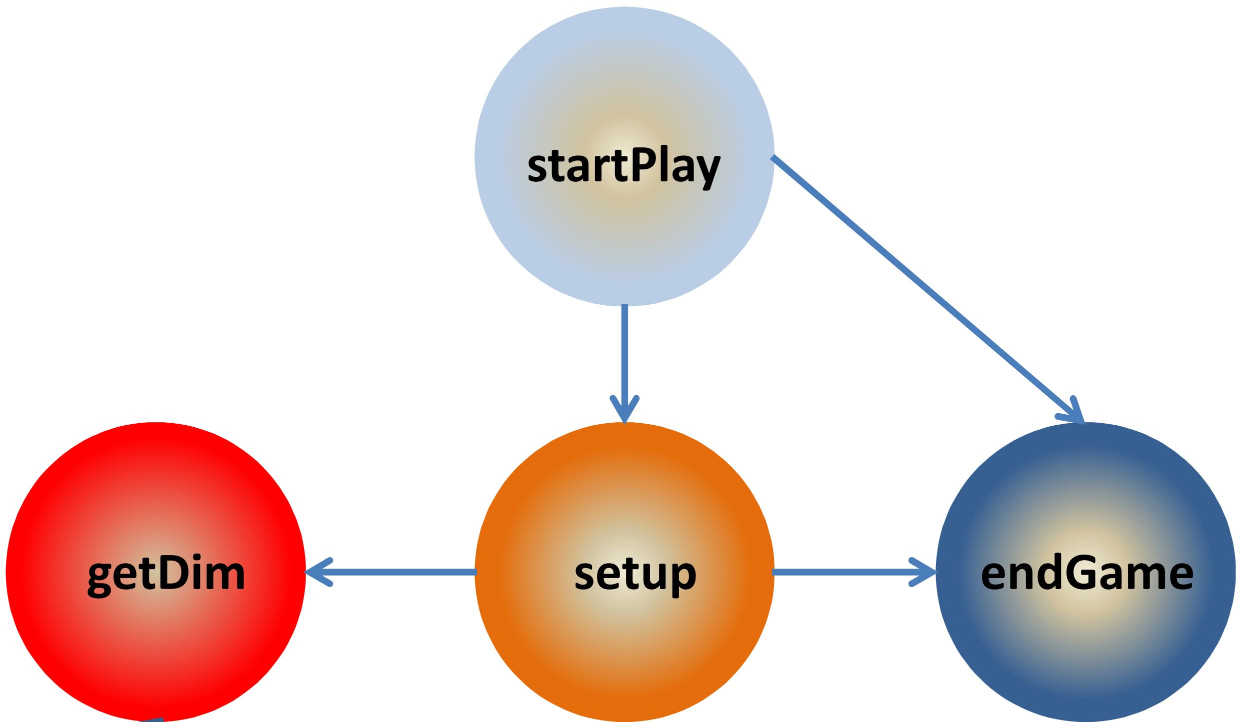

startPlay
getDim
setup
endGame

```
for(i=0; i<$(".allCells").get().length; i++)
    setup($(".allCells").get(i).prop('tagName'));
  endGame();
```

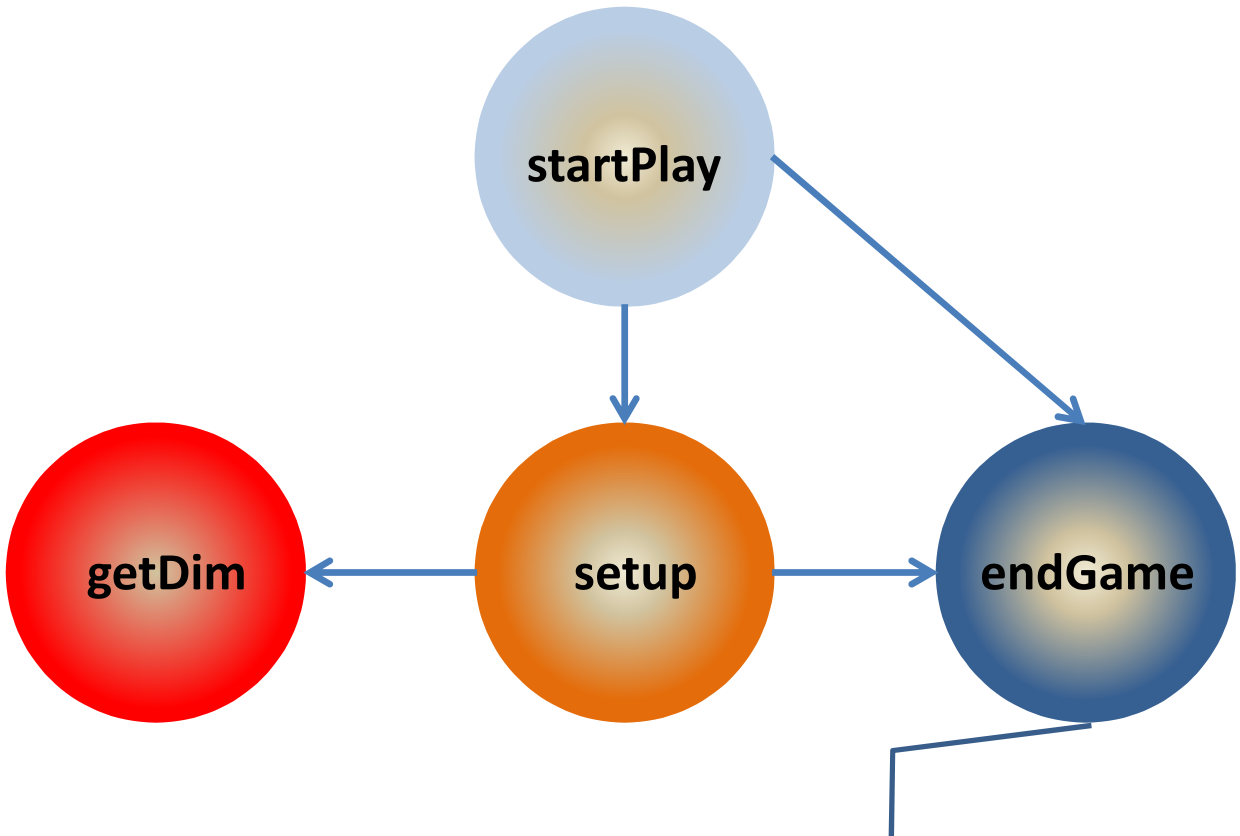




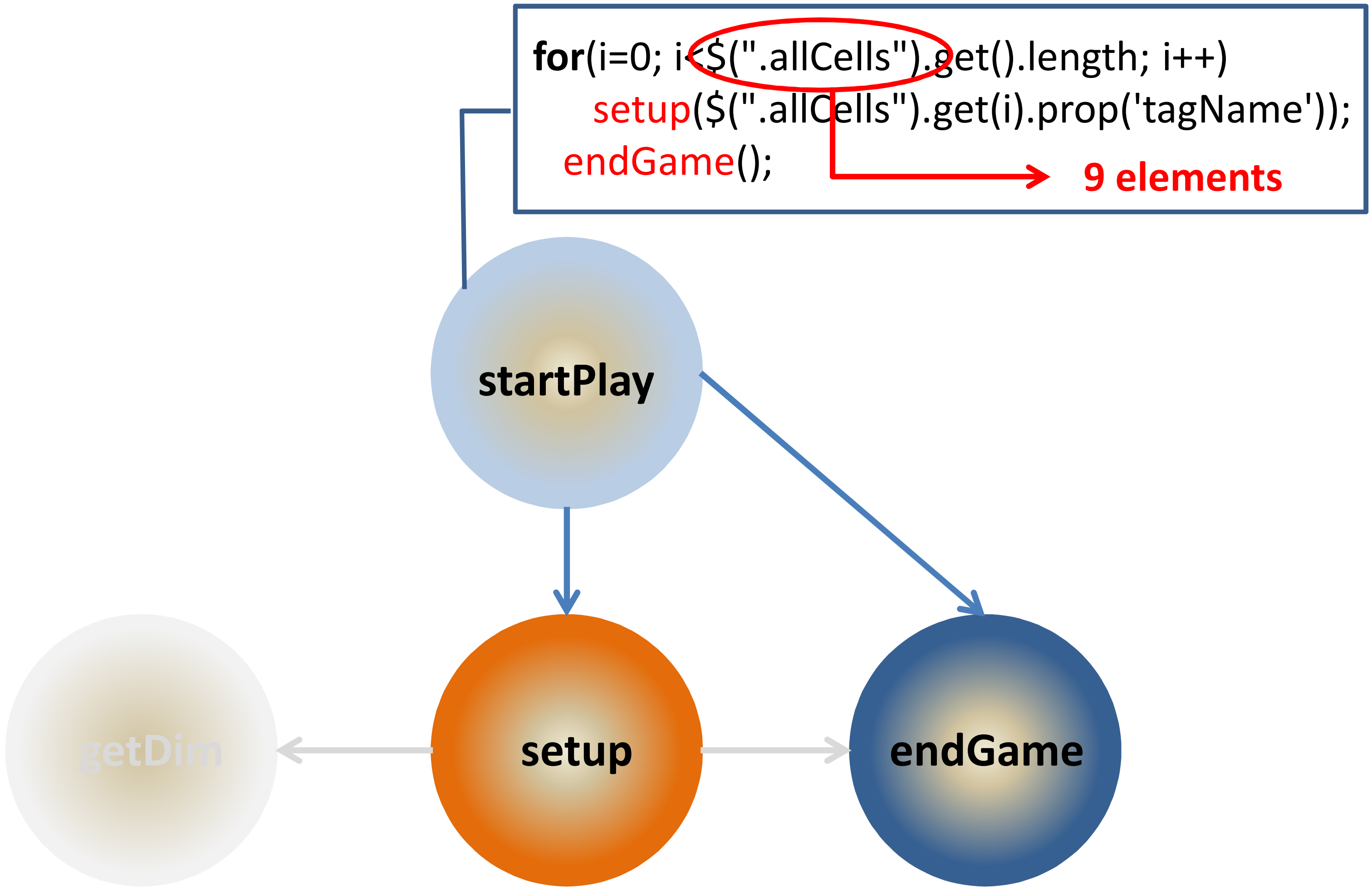

```
if($(cellTag).get().length == 0)
      endGame();
for(i=0; i<$(cellTag).get().length; i++)
       dimension=
       getDim($(cellTag).get(i).width(),
              $(cellTag).get(i).height());
              $(cellTag).get(i).css('height',
              dimension+'px');
```




```
var w = width*2, h = height*4, v = w/h;
 if(v > 1)
    return (v);
 else
    return (1/v);
```

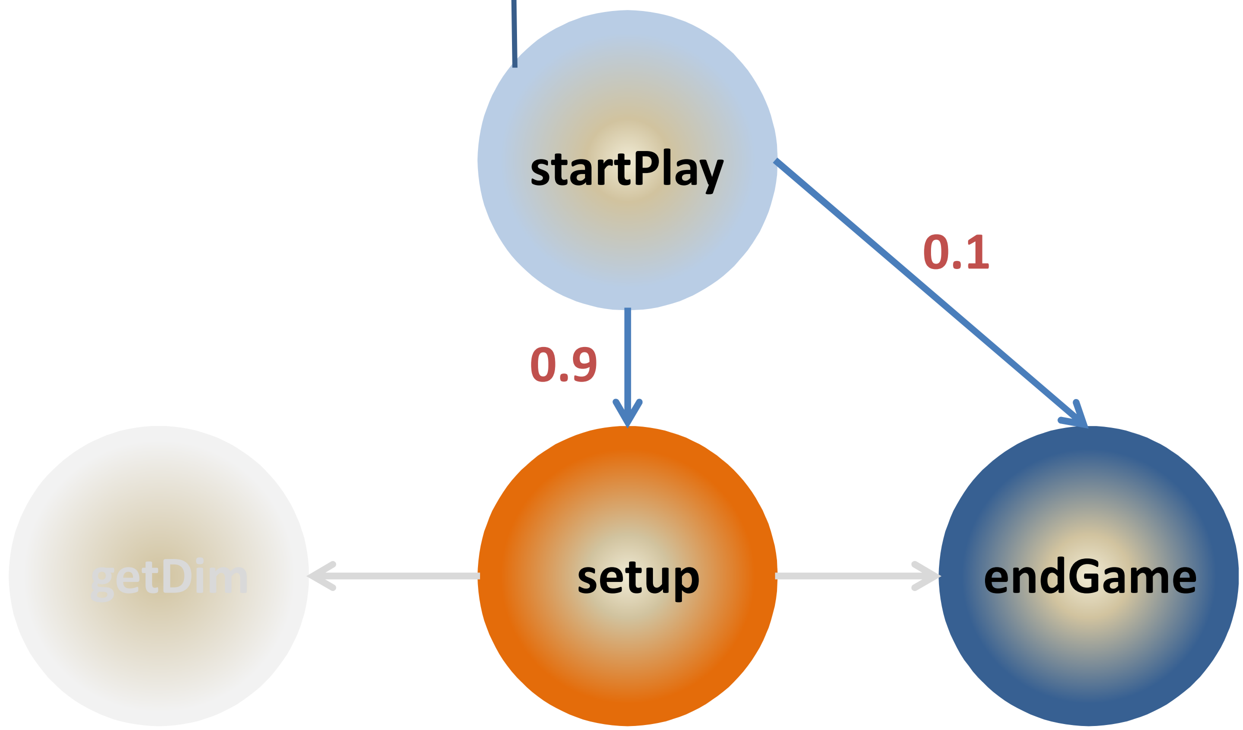


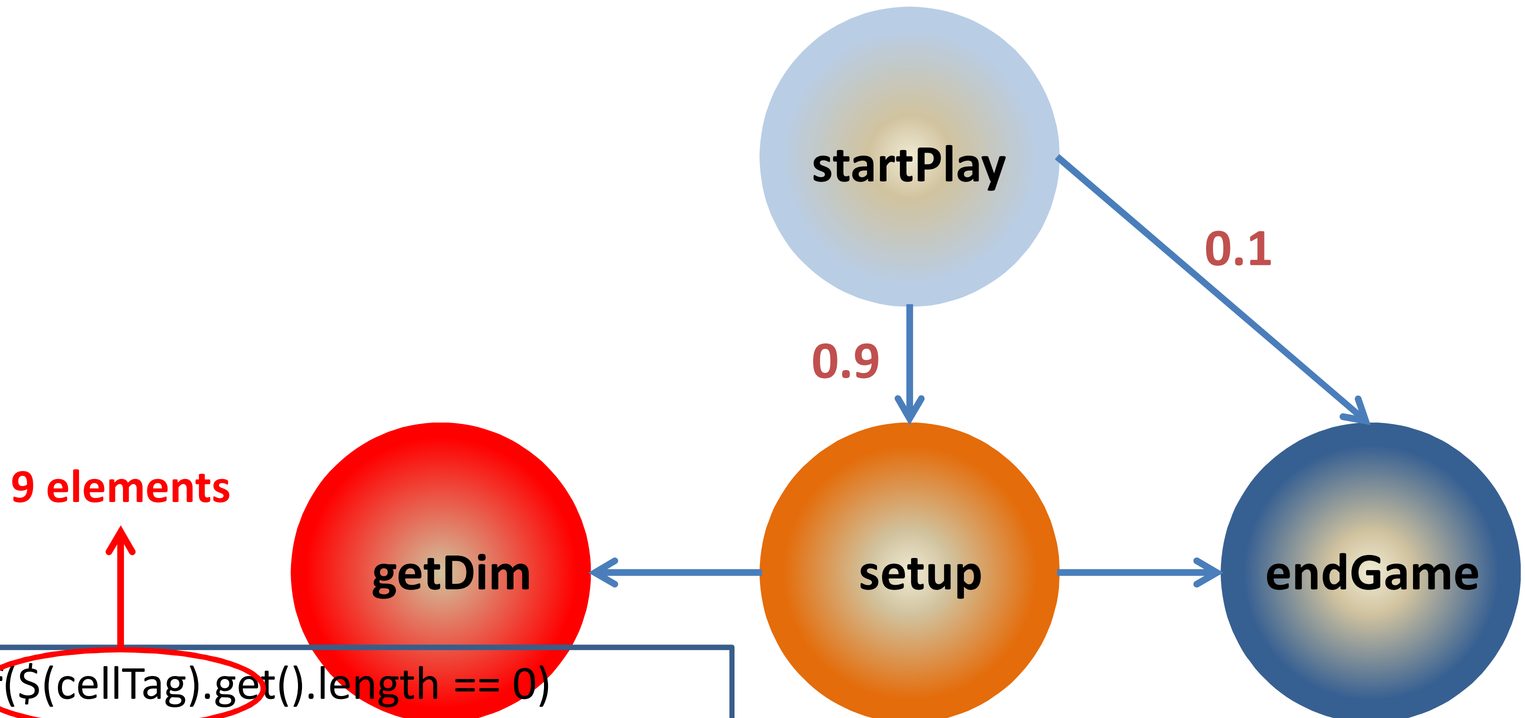

```
$(#end).css('height', getDim($('body').width(),
$('body').height())+'px');
```


for(i=0; i<$(".allCells").get().length; i++)
setup($(".allCells").get(i).prop('tagName'));
endGame();
9 elements
startPlay
getDim
setup
endGame

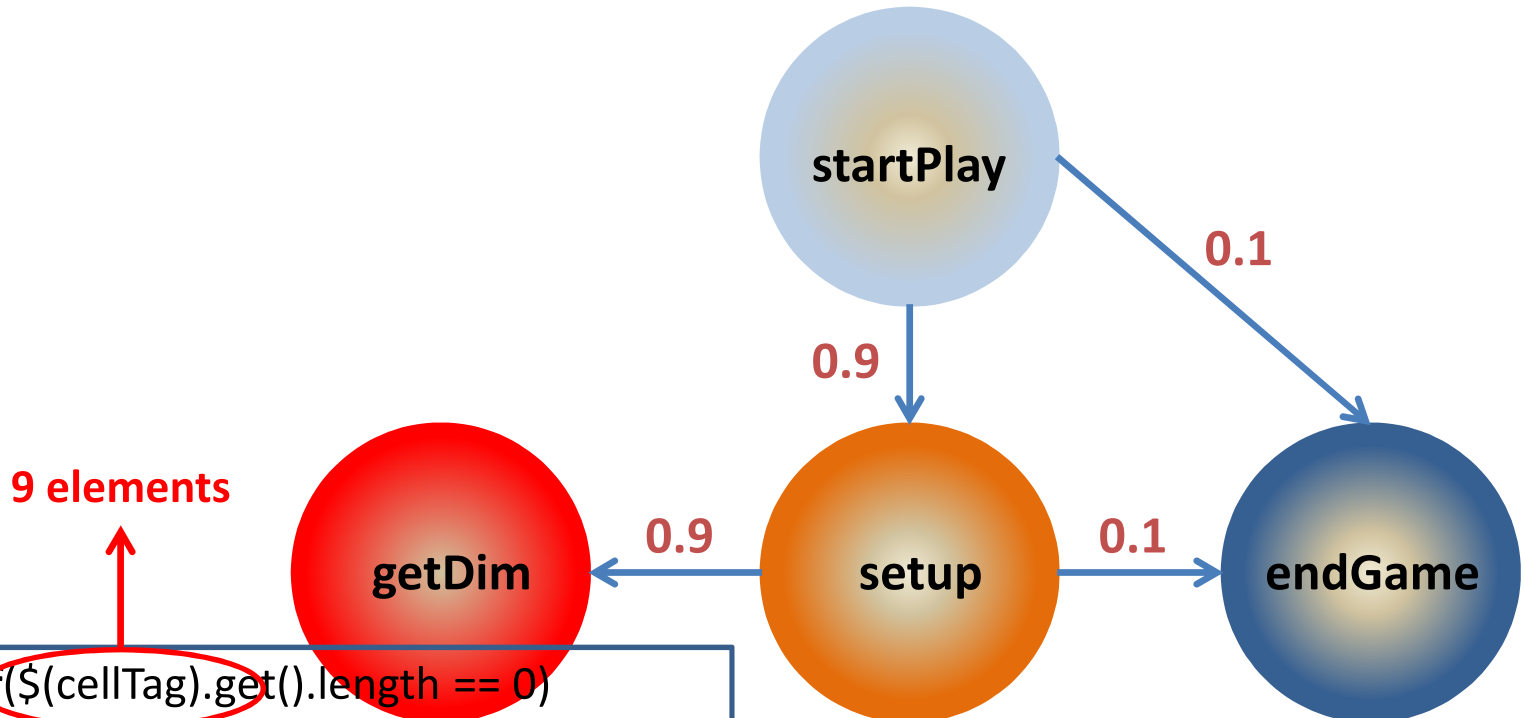
```
for(i=0; i<$(".allCells").get().length; i++)
    setup($(".allCells").get(i).prop('tagName'));
  endGame();
```

9 elements

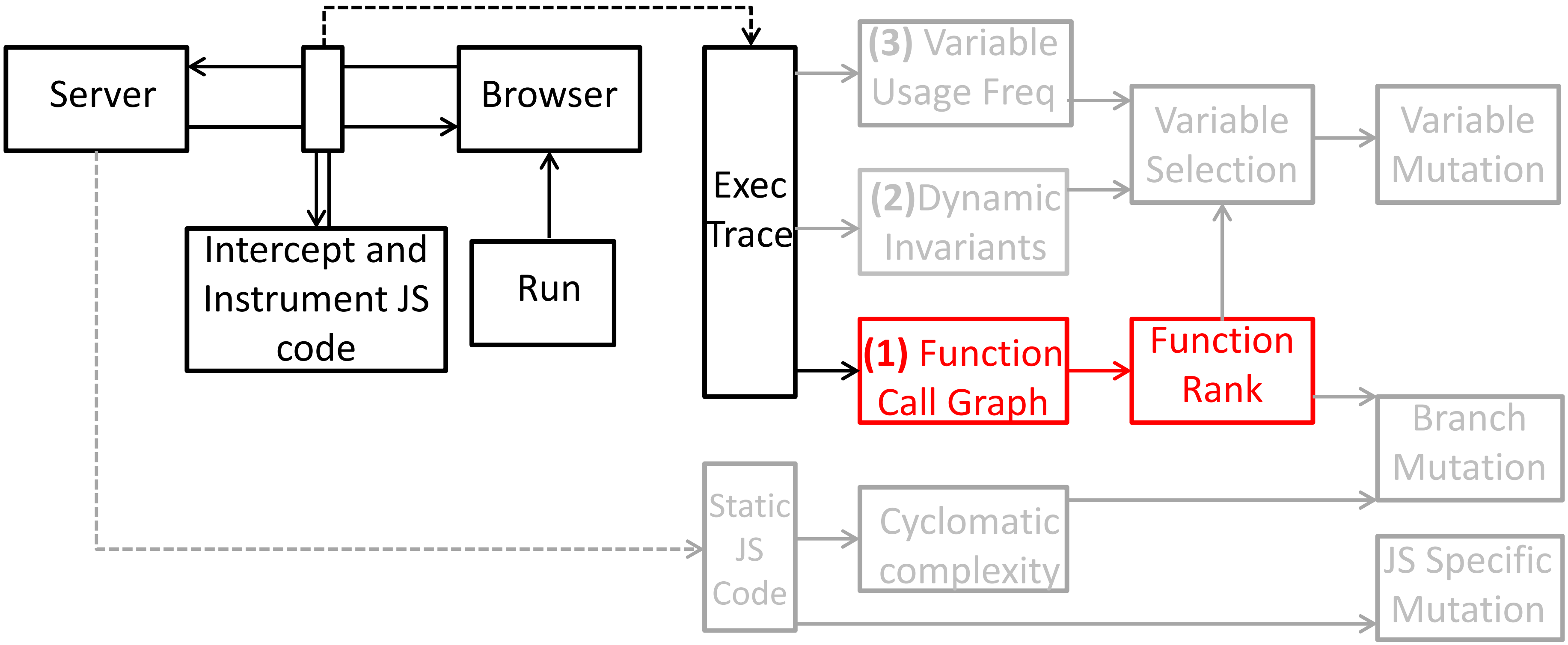





```
if($(cellTag).get().length == 0)
      endGame();
for(i=0; i<$(cellTag).get().length; i++)
      dimension=
      getDim($(cellTag).get(i).width(),
              $(cellTag).get(i).height());
              $(cellTag).get(i).css('height',
              dimension+'px');
```



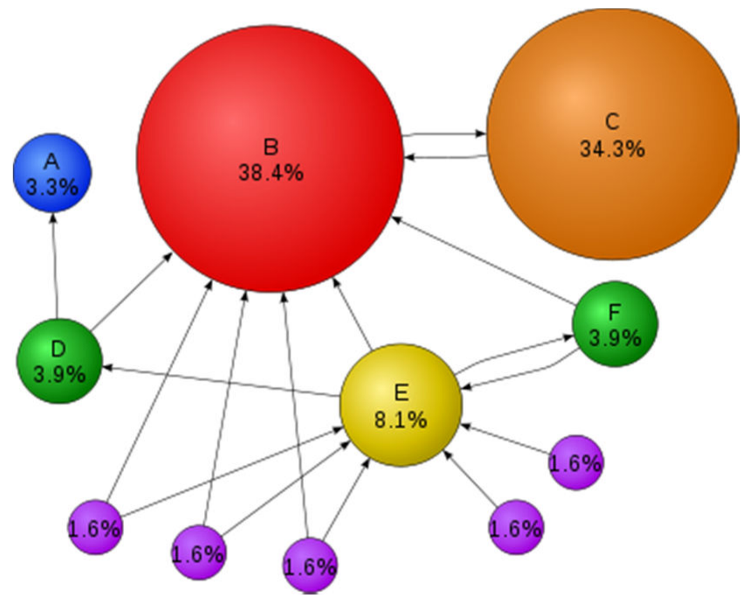

```
if($(cellTag).get().length == 0)
      endGame();
for(i=0; i<$(cellTag).get().length; i++)
      dimension=
      getDim($(cellTag).get(i).width(),
               $(cellTag).get(i).height());
               $(cellTag).get(i).css('height',
               dimension+'px');
```



Information extraction from the execution traces:

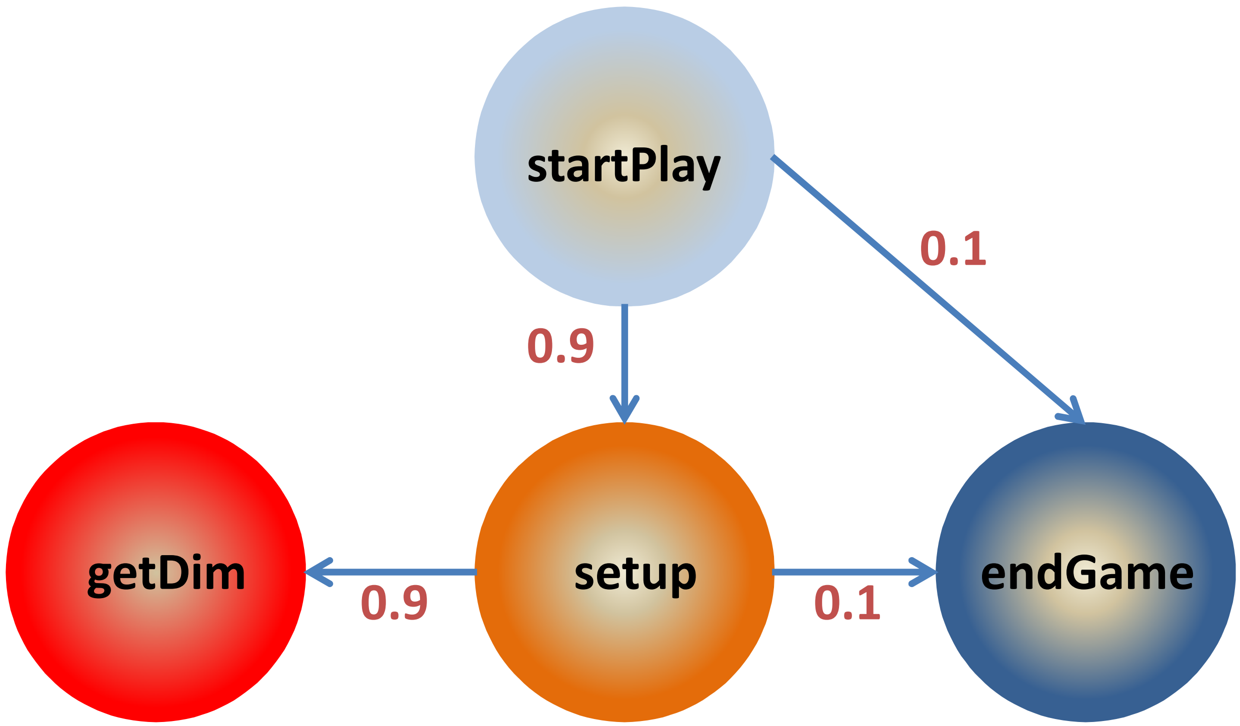
(1) Dynamic call graph of the application → FunctionRank

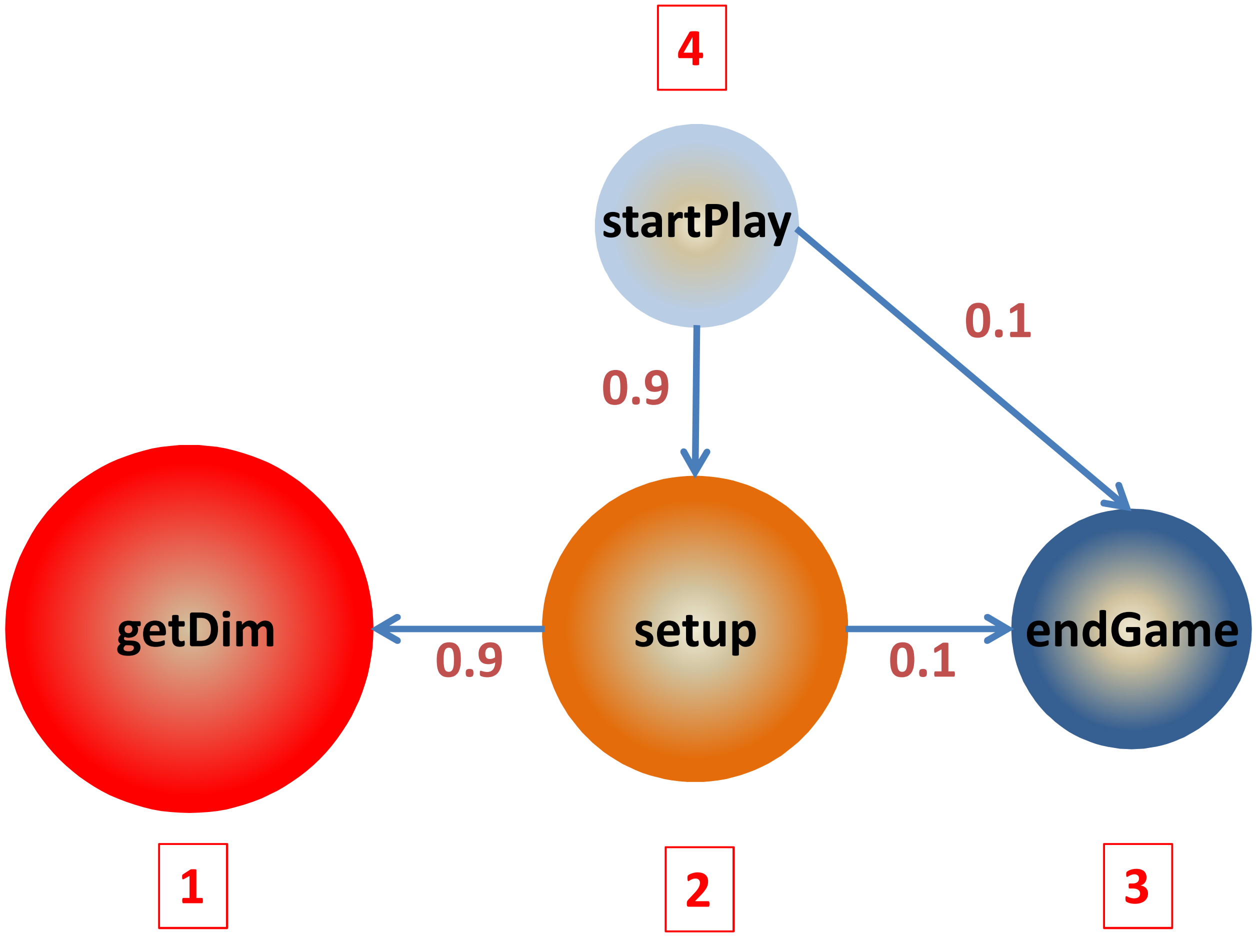
- Ranks and select functions for generating variable mutations

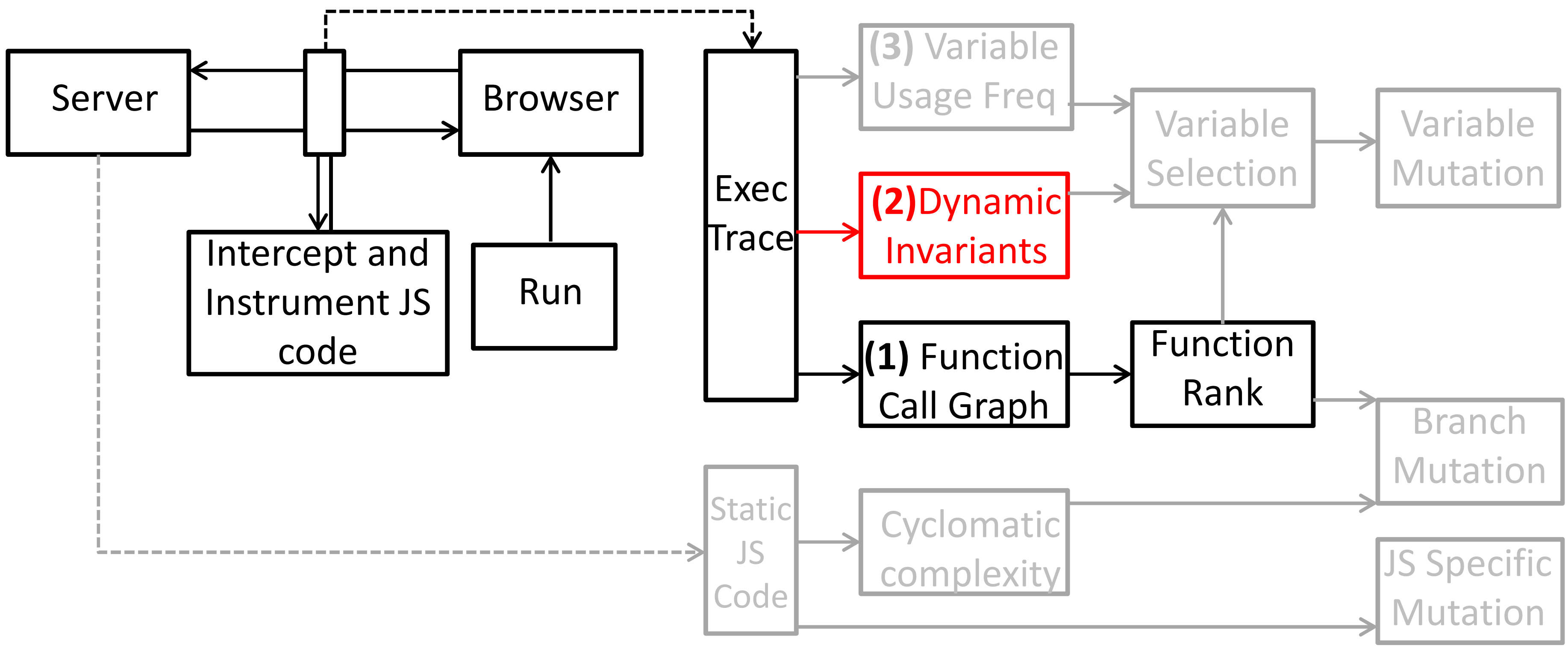


- FunctionRank
  - # dynamic calls to a given function
  - Measures the relative importance of each function at runtime

Function called by several functions with high FunctionRank and high call frequency receives a high rank itself


startPlay
0.9
0.1
getDim
0.9
setup
0.1
endGame


4
startPlay
0.9
0.1
getDim
setup
endGame
0.9
0.1
1
2
3



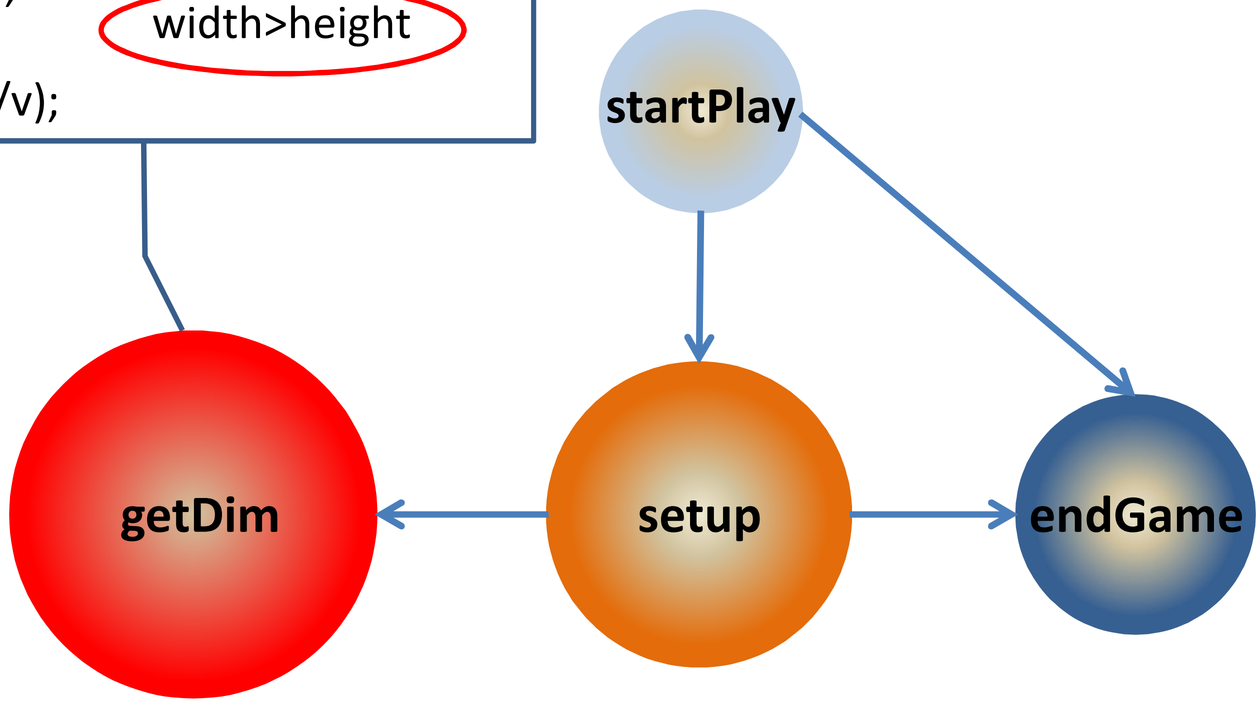
Information extraction from the execution traces:

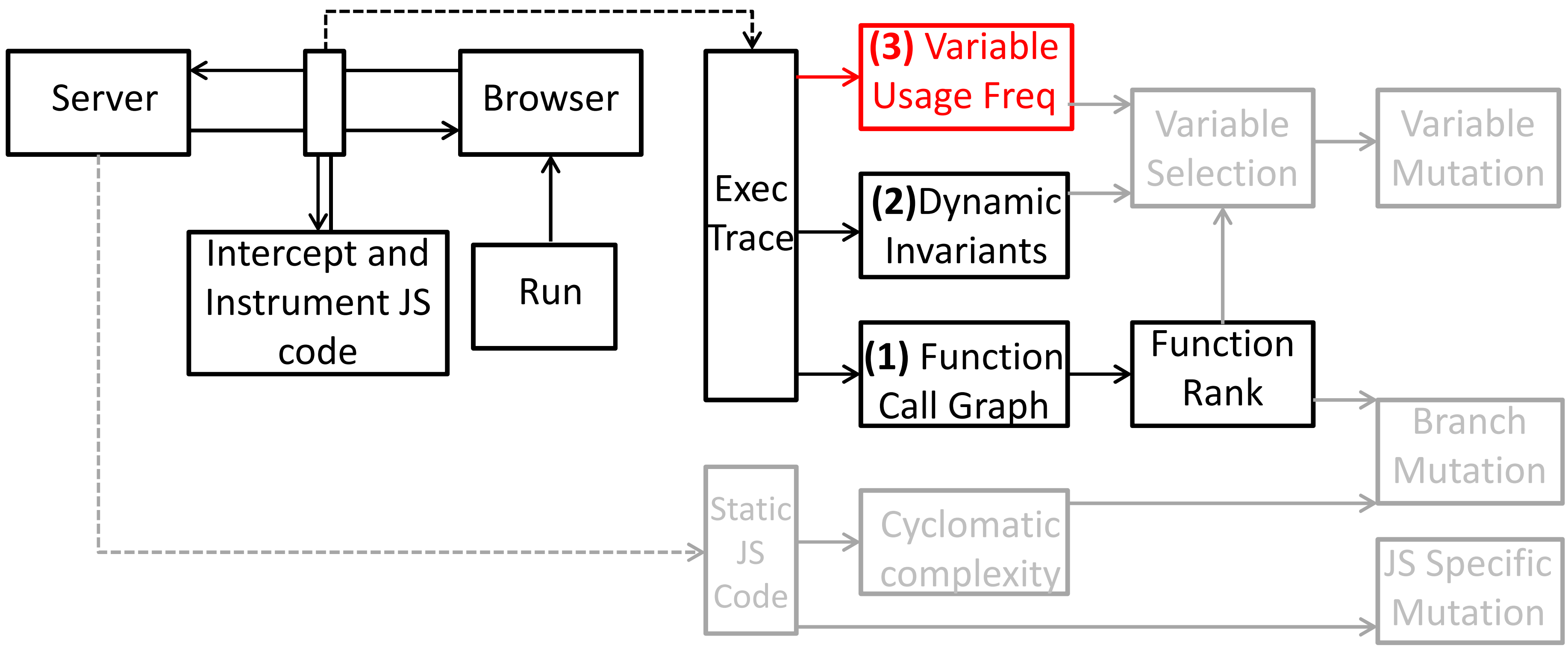
(1) Dynamic call graph of the application → FunctionRank

(2) Dynamic invariants

```
var w = width*2, h = height*4, v = w/h;
 if(v > 1)
    return (v);
 else
    return (1/v);
```

width>height



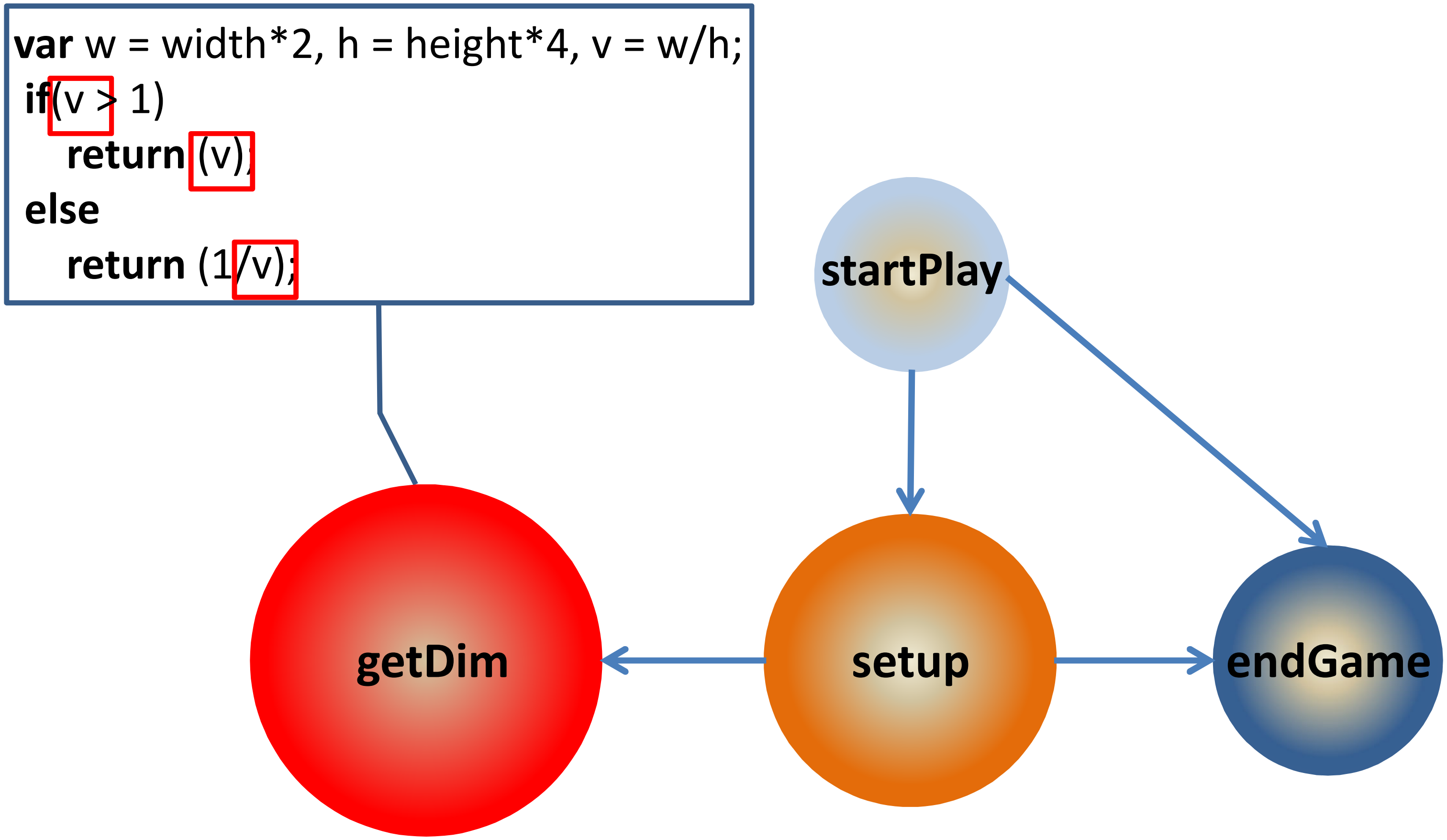


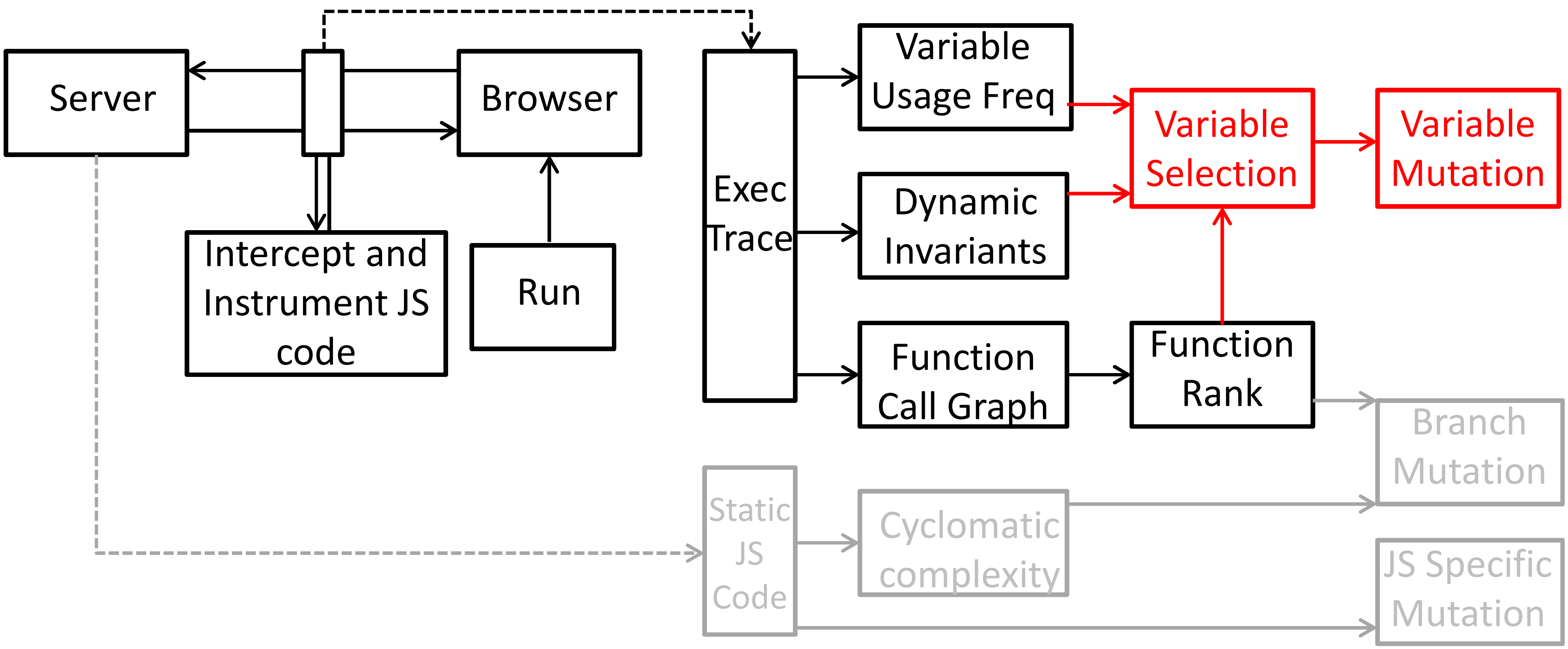
Information extraction from the execution traces:

(1) Dynamic call graph of the application → FunctionRank

(2) Dynamic invariants

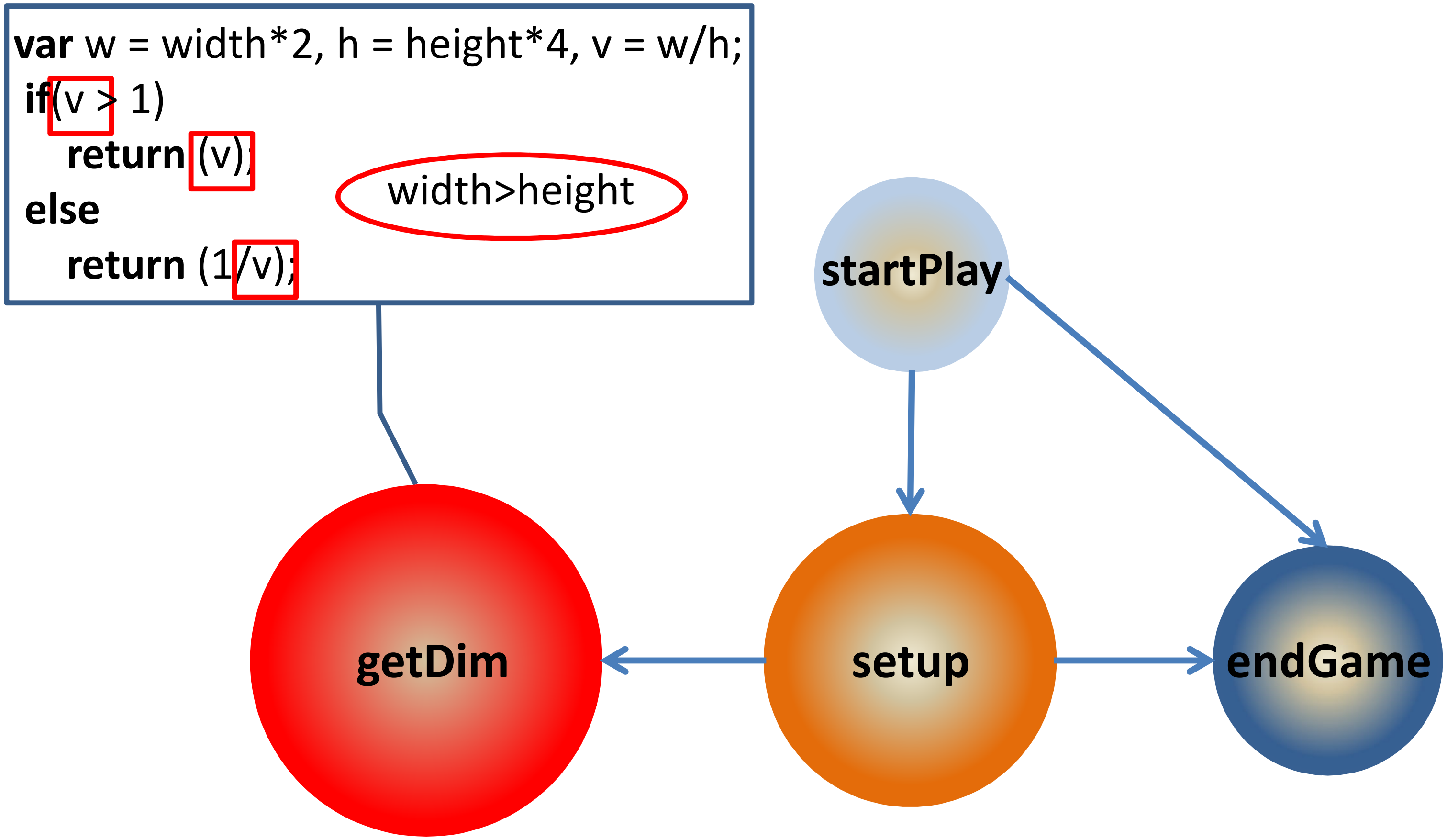
(3) Variable usage frequency


var w = width*2, h = height*4, v = w/h;
if(v > 1)
return (v);
else
return (1/v);
startPlay
getDim
setup
endGame

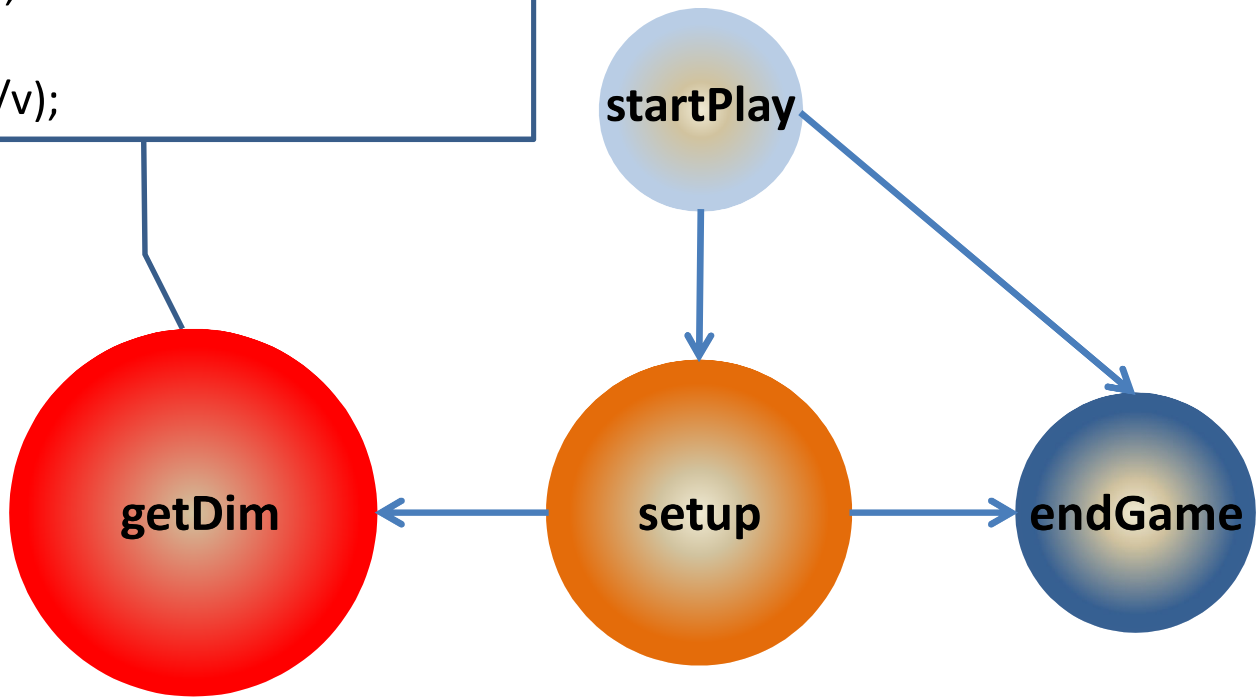


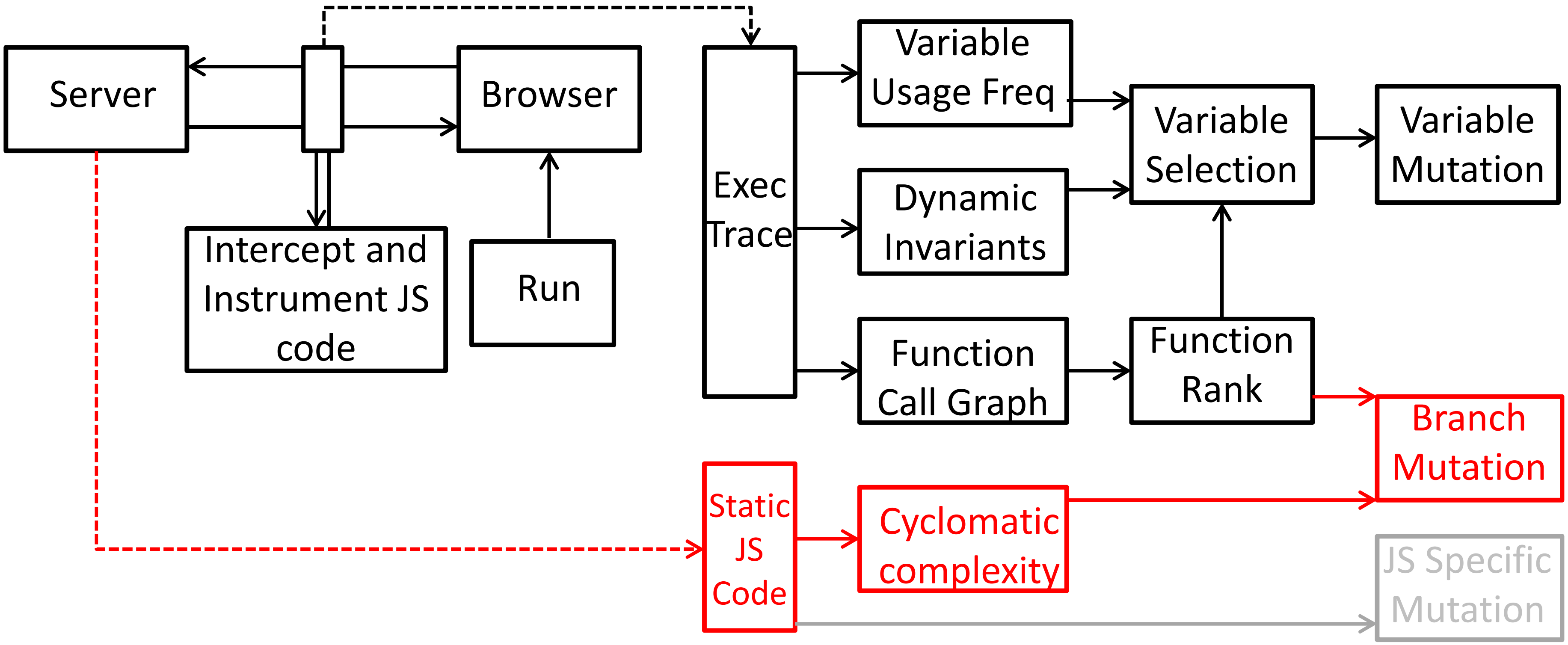
Variables with a significant impact on the function’s outcome based on:

- Usage frequency
- Dynamic invariants

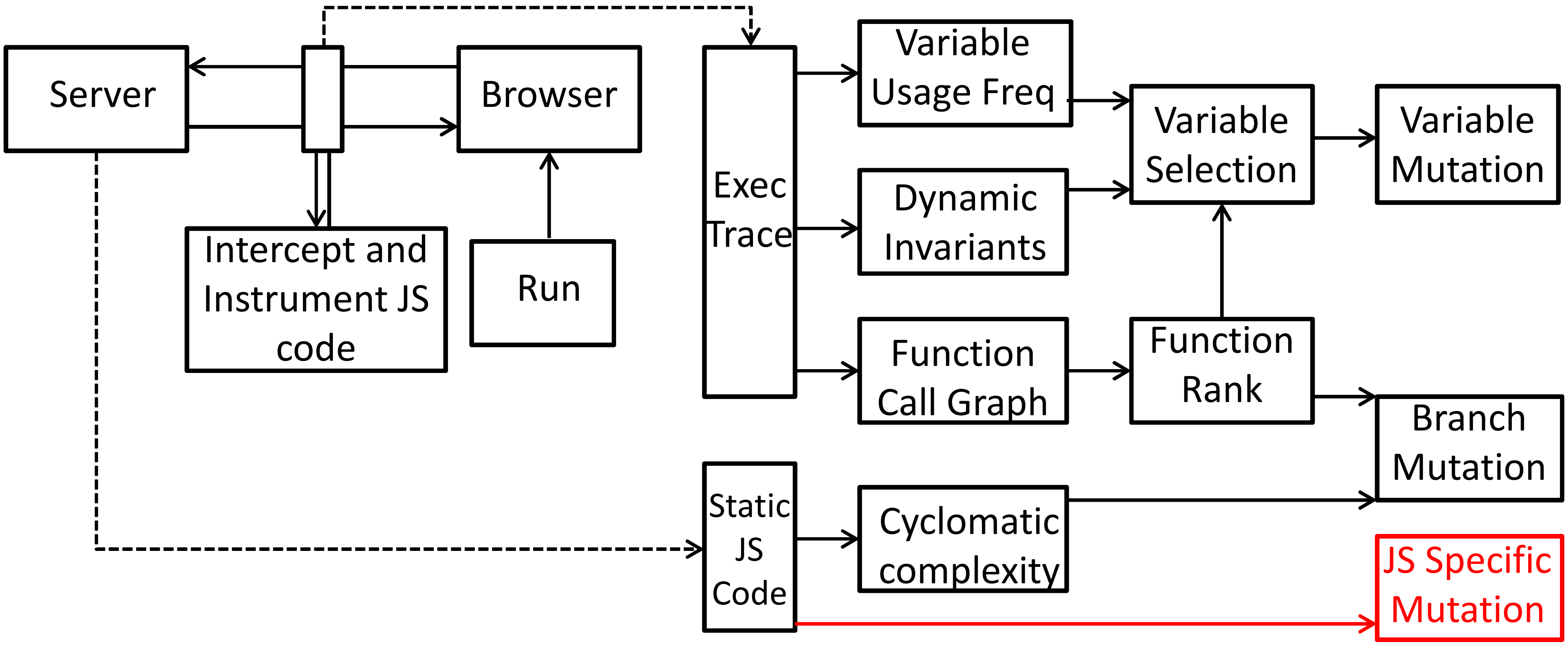

var w = width*2, h = height*4, v = w/h;
if(v > 1)
return (v);
else
return (1/v);
width>height
startPlay
getDim
setup
endGame

```
var w = width*2, h = height*4, v = w/h;
 if(v > 1)
    return (v);
 else
    return (1/v);
```





- Cyclomatic complexity: Statically analyzing the code
- Branch mutation on the highly ranked functions with high cyclomatic complexity



Consider a number of JavaScript specific operators:

- Common mistakes that JavaScript programmers make
- Collected from various resources

Tool Implementation

- Implementation of the approach in an open-source tool called Mutandis
- JavaScript dynamic invariants: JSART (ICWE'12)
- Execution trace profiler: Collecting data by exercising the application
  - Exhaustive automatic navigation: Crawljax
  - Execution of existing test cases
  - Combination of crawling and test suite execution

Evaluation

- How efficient is Mutandis in generating non-equivalent mutants?
- How effective is Mutandis in injecting critical behaviour-affecting faults?
- How useful is Mutandis in assessing existing test cases of a given web application?

Evaluation

- How efficient is Mutandis in generating non-equivalent mutants?
- How effective is Mutandis in injecting critical behaviour-affecting faults?
- How useful is Mutandis in assessing existing test cases of a given web application?

Efficiency in generating non-equivalent mutants

- Objects: 5 open-source web applications
- Inject 200 faults for the five objects
  - Automatically generating mutants using Mutandis
- Examine output for equivalent mutants

Results:

On average, the percentage of equivalent mutants generated is 7%

- On average, 10 to 40 percent of equivalent mutants → Efficiency of Mutandis in generating non-equivalent mutants

Evaluation

- How efficient is Mutandis in generating non-equivalent mutants?
- How effective is Mutandis in injecting critical behaviour-affecting faults?
- How useful is Mutandis in assessing existing test cases of a given web application?

Effectiveness in injecting critical behaviour-affecting faults

- Use the bug severity ranks used by the Bugzilla bug tracking system
- Choose non-equivalent mutants and assigning a bug score according to the ranks

Bug Severity	Description	Rank
Critical	Crashes, data loss	5
Major	Major loss of functionality	4
Normal	Some loss of functionality, regular issues	3
Minor	Minor loss of functionality	2
Trivial	Cosmetic issue	1

- The average bug severity rank across all applications is 3.6

- The injected faults cause normal to major loss of functionality.

- More than 70% of the faults with major loss of functionality, are in the top 20% of important functions in terms of FunctionRank
  - Importance of FunctionRank in the fault seeding process

Results:

Mutandis is effective in generating mutants that cause nontrivial errors

Evaluation

- How efficient is Mutandis in generating non-equivalent mutants?
- How effective is Mutandis in injecting critical behaviour-affecting faults?
- How useful is Mutandis in assessing existing test cases of a given web application?

Usefulness in assessing existing test cases

- Run Mutandis on two JavaScript libraries
  - Available Qunit test cases
- Generate 120 mutants for each library
- Determine the usefulness of our approach based on the number of:
  - Non-equivalent generated mutants
  - Non-equivalent surviving mutants

- Less than 3% of the mutants generated by Mutandis are equivalents

- All the non-equivalent mutants that are not killed by the test suites, are in the top 30% of the important functions in terms of FunctionRank
  - The importance of FunctionRank in test case generation

Results:

Guides testers towards designing test cases for important portions of the code

Conclusion

- Mutation testing technique that leverages dynamic and static characteristics
- Selectively mutates portions of the code that exhibit a high probability of being error-prone and affecting the observable behaviour
- Implementation of the approach in an open-source tool called Mutandis (https://github.com/saltlab/mutandis)
- The evaluation of Mutandis
  - Efficacy of the approach