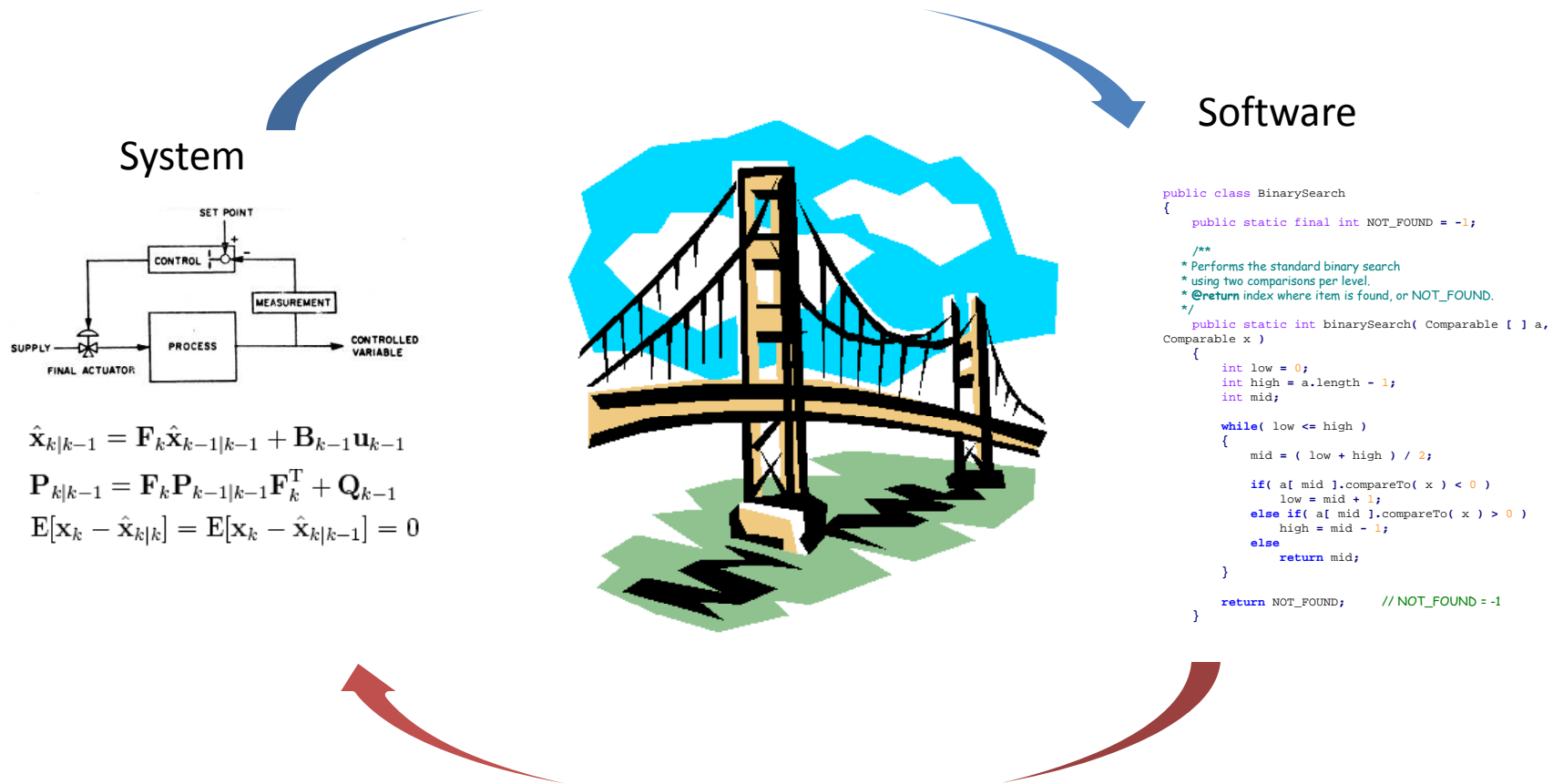


Identifying Implicitly Declared Self-Tuning Behavior through Dynamic Analysis

Hamoun Ghanbari, Marin Litoiu
Department of Computer Science
York University
ICSE 2009 Workshop
Software Engineering for
Adaptive and Self-Managing Systems (SEAMS)

Software-system bridge



A legacy software with hidden self-* capabilities

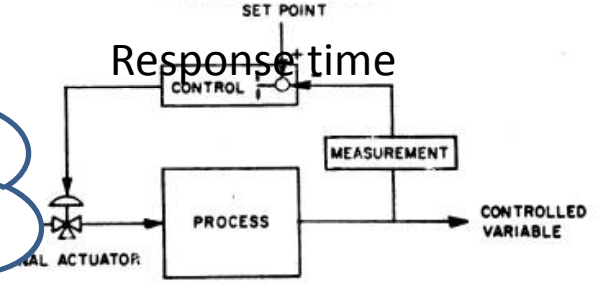
I have a software with 100000 LOC which employs some kind of autonomic self- behavior*



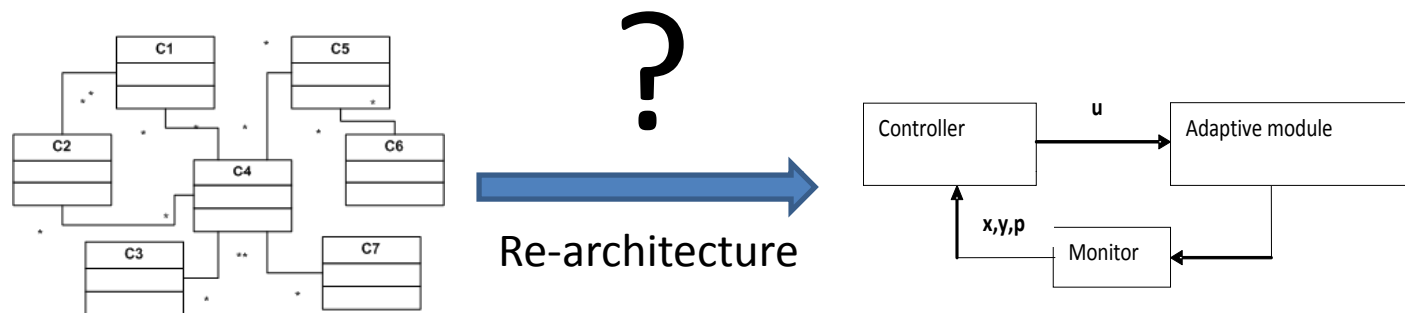
Appropriately by myself separately using:

```

...
if(responseTimeDrop >
    5% of averageResponseTime)
{
    workingThreads++;
}
...
    
```



Final goal



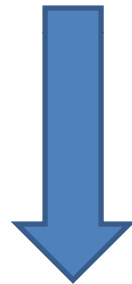
Integration of existing software into autonomic frameworks by:

1. Automating the identification of tuning parameters
2. Re-architecting the software to expose those parameters through a separate control module.

Current approach

Self-tuning behavior is roughly considered to be composed of a *reflexive behavior* connected to a *control mechanism*.

How to find
control parameters
and *reflexive*
values in
software?



Mining program's state and behavior

1. Find the relationship between system's state and behavior
2. Classification of the results into the taxonomy of state-behavior signatures.

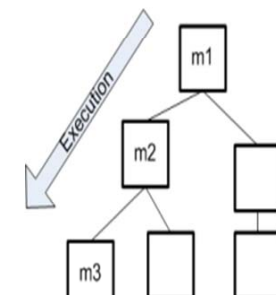
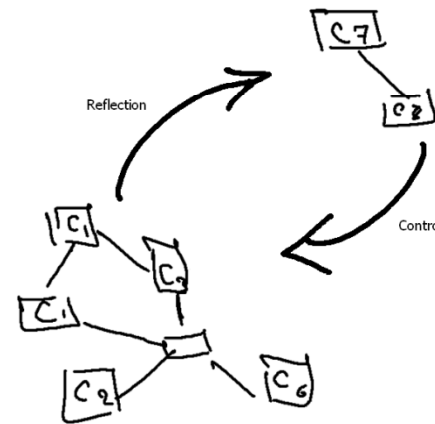
arrivalRate=4
socketTimeout=7
isSynchronized=true

Corresponds to?



yes

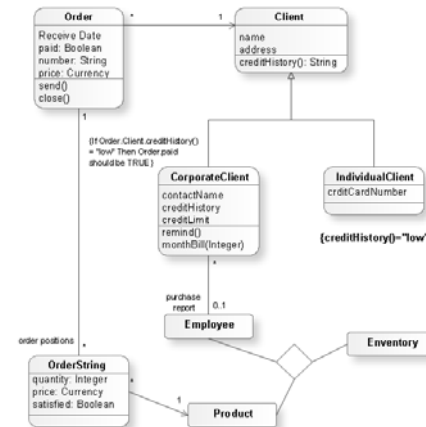
Classify the
correspondence
into the taxonomy



Related Work

1. STAC project: software tuning panels for autonomic control

1. [3] N. Brake, J.R. Cordy, E.D. y, M. Litoiu, and V.P. u, **“Automating discovery of software tuning parameters”**, *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, ACM, Leipzig, Germany, 2008, pp. 65-72.



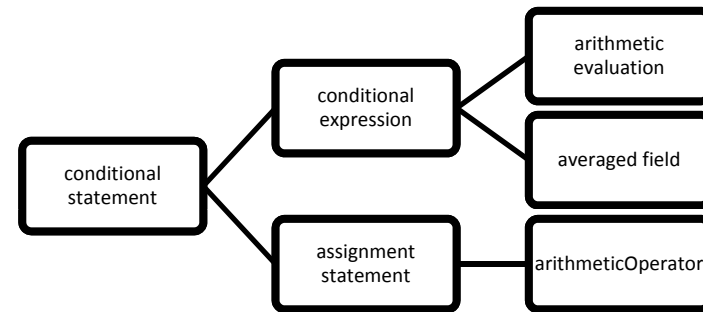
Static way

```

...
if(responseTimeDrop >
  5% of averagaeResponseTime)
{
  workingThreads++;
}
...

```

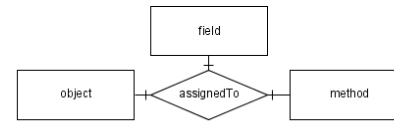
Convert to pattern



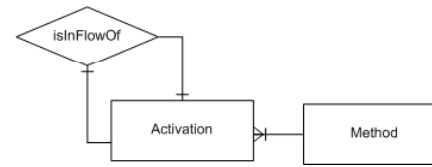
Structural pattern

Data model

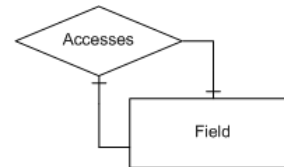
1. Aliasing



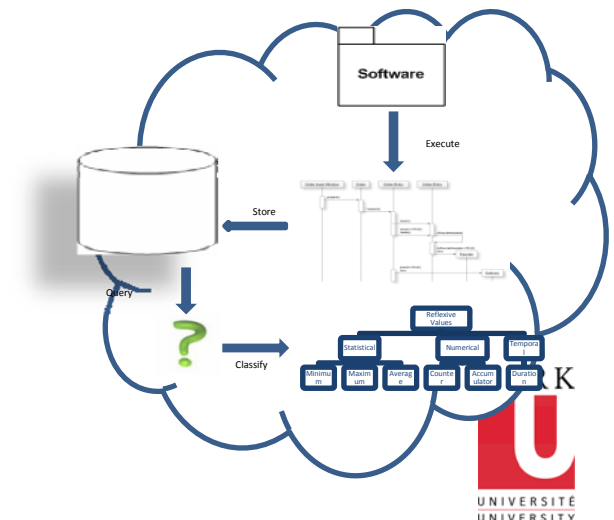
2. Activation flow



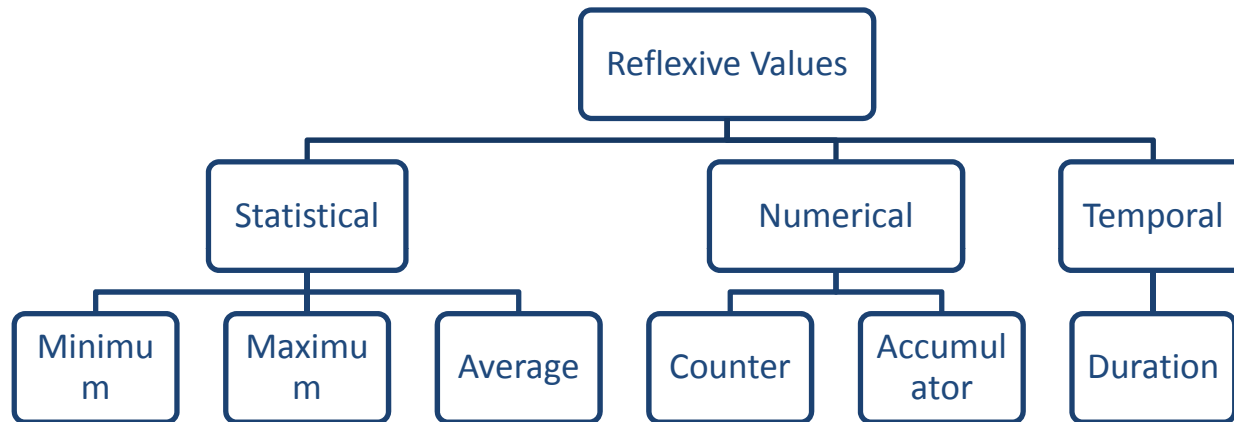
3. Static reachability



4. Distance metric



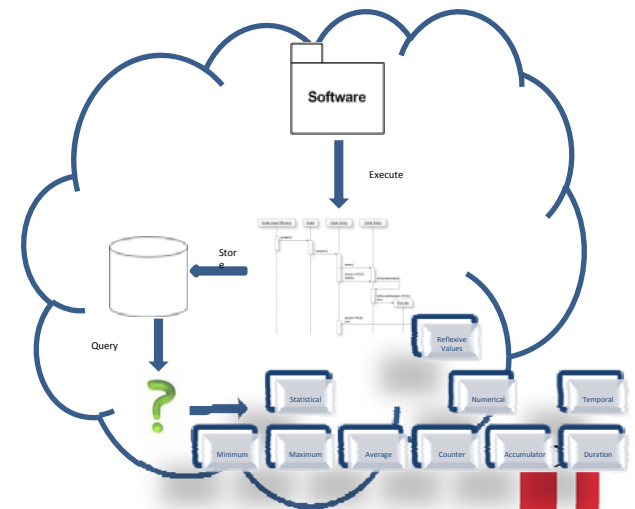
Reflexive value (sensor) patterns



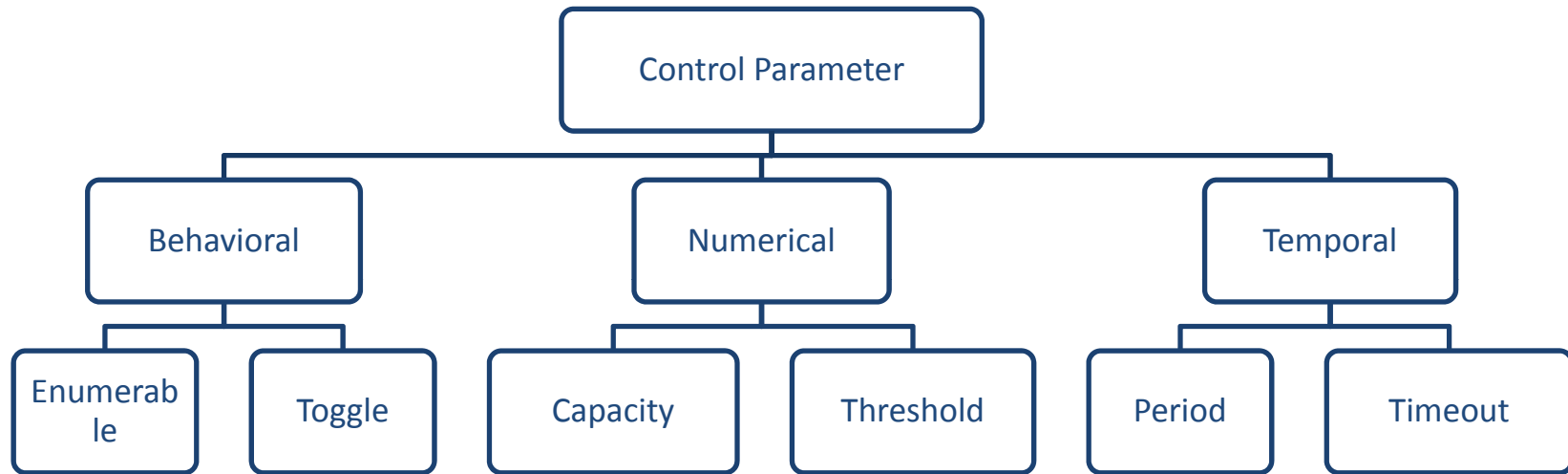
Duration: Measures the length of an event or action has occurred using discrete, constant increments (e.g., number of active connections).

Counter: Measures how many times an event or action has occurred using discrete, constant increments (e.g., number of active connections).

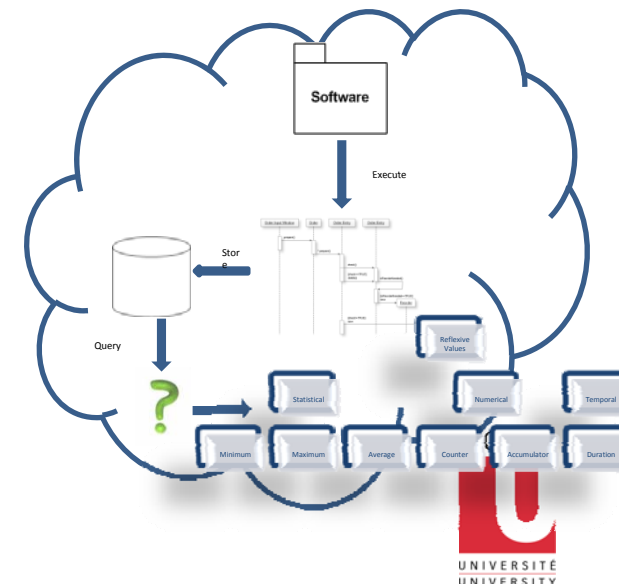
Average: Measures the statistical average of some numerical or temporal parameter.



Control (actuator) patterns



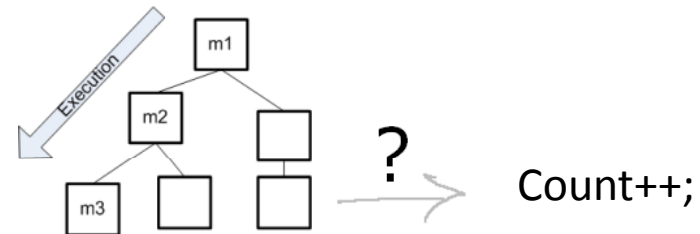
Capacity: a limit on how large (i.e. toggle: enable some length of time and the only way for it to be a resource can be used is to deactivate before it will be used (e.g., reading bytes from a network connection).
 finite set of states (e.g., thread priorities, algorithm selection).



Mining reflexive patterns

1. Identifying counter pattern:

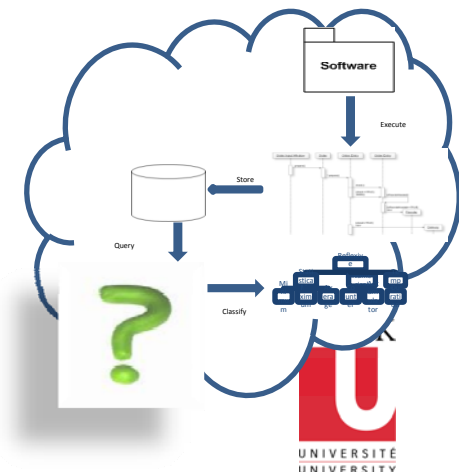
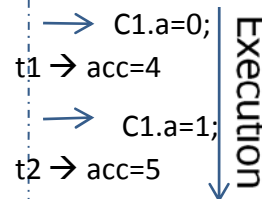
```
isCounter (fld1,counter_obj,fis):-
    isIncremented(m1, counter_obj,fld1),!,
    callStackAfterExecutionOf(m1,callStack),
    fis=frequentItemSetOf(callStack).
```



2. Identifying Accumulator pattern:

```
isAccumulator(accumulator,fld1,accumulated_obj,fld2)
:-aliasedTo(m1,accumulator,fld1,obj1),
  aliasedTo(m2,accumulator,fld1,obj2),
  d=distance(obj1,obj2),
  aliasedTo(_,accumulated_obj,fld2,accum_val)
  ,equals(accum_val,d).
```

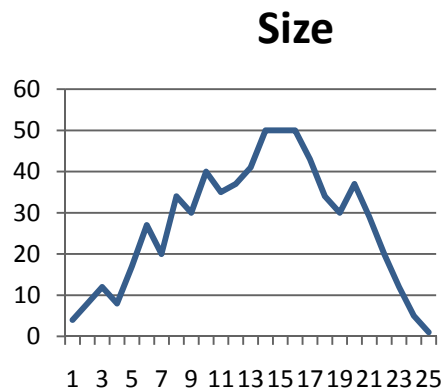
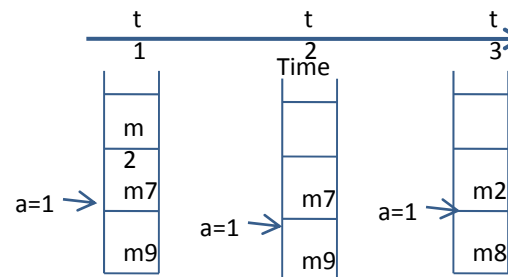
```
public class C1{
    public static int a=0;
}
```



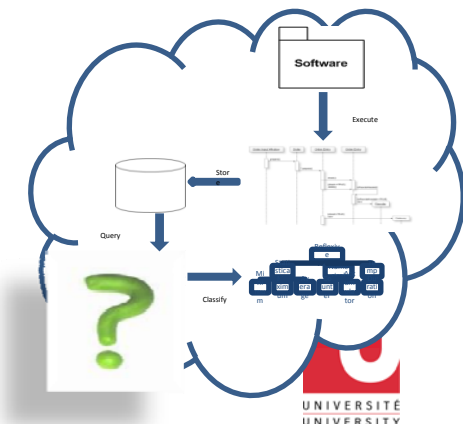
Mining control patterns

- Mining execution control patterns:
 1. Attach variables and call stack together in method activations
 2. Identify a common subset of call stack and variables values.
- Mining temporal/physical :

```
isToggling(fld1,container,fis):-
isLastValueOf(container,fld1,m1,fld_value),
callStackAfterExecutionOf(m1,callStack),
fis=frequentItemSetOf([fld_value|callStack]).
```



```
int maxSize=50;
public onArrival(Packet
packet){
    if (!bufferSize > maxSize){
        insert(packet);
    }
}
```



Mining self-tuning loop (reflection + control)

Example: a web server might increase the connection pool size if the maximum number of faults per second exceeds some amount.



```
{
  update(maxReqPerSec);
  boolean increaseSize=
    curFaultRate > maxFaultRate;
  if (increaseSize){
    maxFaultRate = curFaultRate +
      (curFaultRate - maxFaultRate)*0.1;
    conPool.setSize(curSize*1.2);
  }
}
```

We encode the full control loop:

- **reflexive parameter:** *maximum* number of faults per second.
- **controlled behavior:** increasing the size of connection pool.



```
isMaximumToggleLoop():-
  isMaximum(container1,maximum_fld),
  isToggling(boolean_fld,container2,partOfStack),
  hasCorrespondence(maximum_fld,boolean_fld).
```

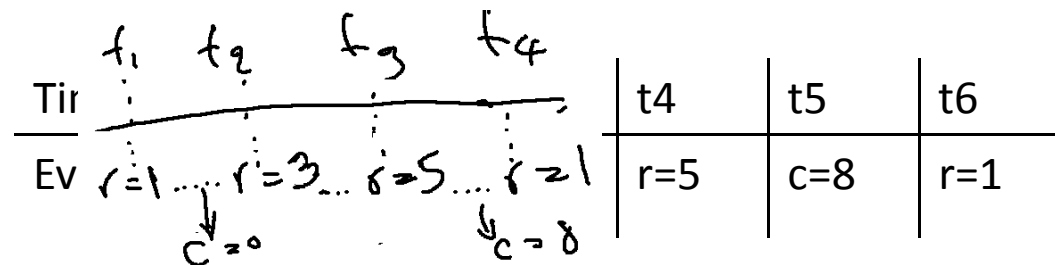
The Predicate hasCorrespondence()

Consider this static structure:

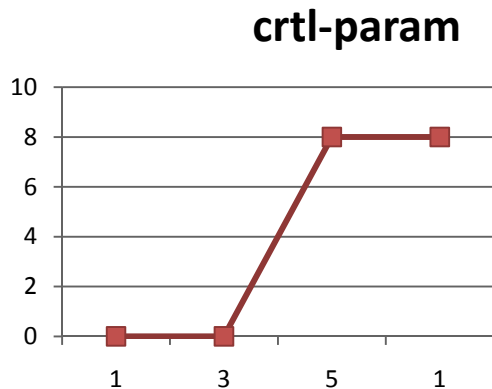
```
public class C1{
    public int reflexiveParam;
}

public class C2{
    public int controlParam;
}
```

And This execution:



Correspondence(r,c)={<1,0>, <3,0>, <5,8>, <1,8>}



Could we find a regression? → Yes.->hasCorrespondence(c,r)=true

Example:

A behavioral pattern: Toggle

- An example of a toggle named `disableUploadTimeout` is mentioned.
- This toggle controls whether timeouts are employed when reading from a socket.
- Whenever the `disableUploadTimeout` is false `setSoTimeout` is called.

```
class Http11Processor {  
  
    public boolean disableUploadTimeout = false;  
    public int keepAliveTimeout;  
    public int soTimeout;  
    public boolean keptAlive;  
    //...  
    public void process(Socket socket) throws  
        IOException {  
        //...  
        if (!disableUploadTimeout && keptAlive) {  
            if (keepAliveTimeout > 0) if false  
                socket.setSoTimeout(keepAliveTimeout);  
            }  
            else if (soTimeout > 0) {  
                socket.setSoTimeout(soTimeout);  
            }  
        }  
        //...  
        if (!disableUploadTimeout) {  
            int timeout = 0; is false  
            socket.setSoTimeout(timeout);  
        }  
    }  
}
```

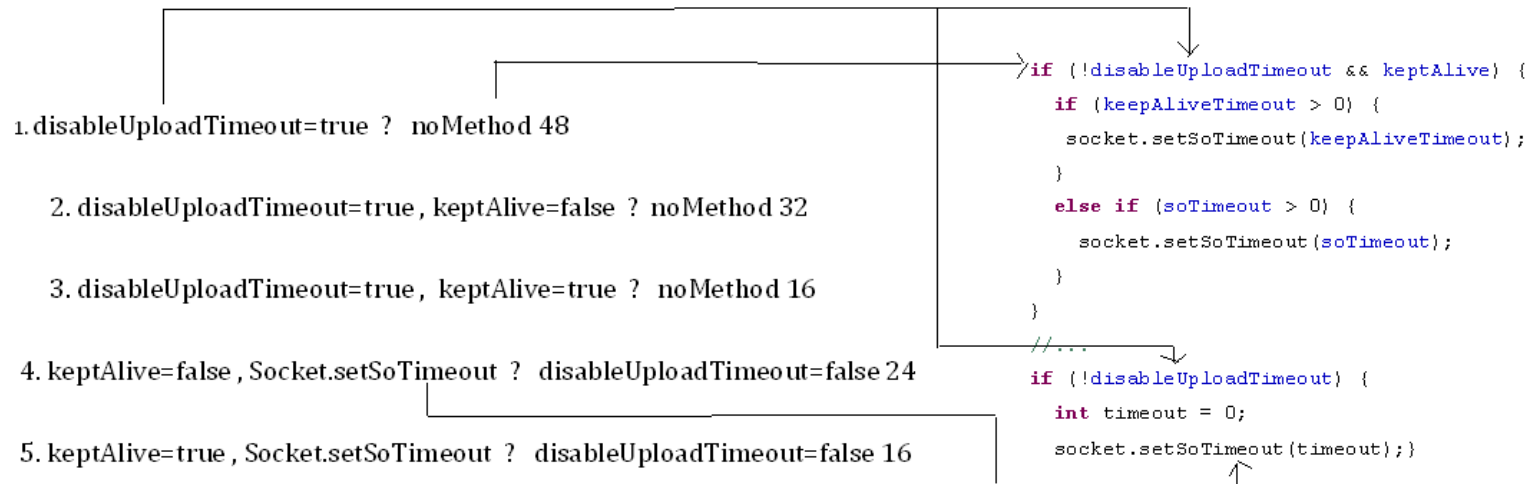
is true

is false

Example:

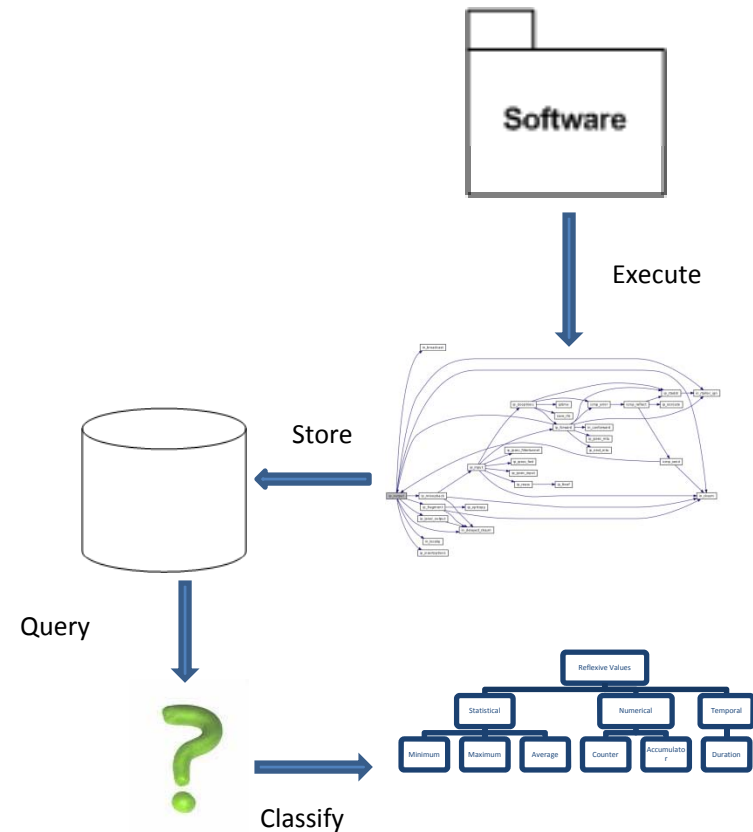
A behavioral pattern: Toggle

- applying association rule mining proves presence of toggle behavior:
 - method `Socket.setSoTimeout` *correlates* with the value of `disableUploadTimeout`.



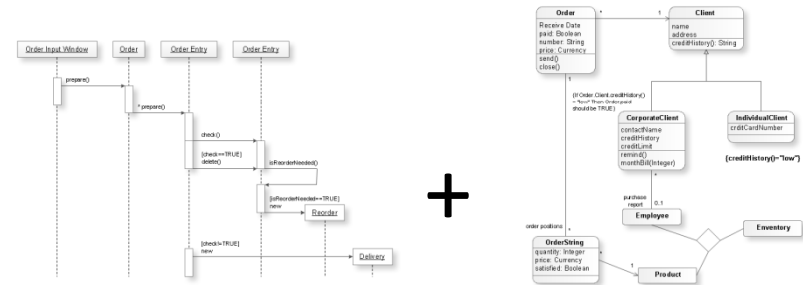
Conclusion

- we introduced a new technique to identify tuning parameters using dynamic analysis.
- Tested on different types of tuning parameters: statistical, numerical, temporal.
- Our approach can be used as a filter for static analysis.



Future Work

- Combining static and dynamic analysis could help addressing the scalability issue in larger applications
- Identifying the implicitly declared self-tuning behavior instead of just self-tuning parameters.
- Using sophisticated probabilistic approaches.



$$P(a) = \frac{\sum_{i=0}^{a-1} \left(\frac{1-p}{p}\right)^i}{\sum_{i=0}^{a+b-1} \left(\frac{1-p}{p}\right)^i}$$

