Design of Intelligent Agents for Collaborative Testing of Service-Based Systems

Xiaoying BAI and Bin CHEN

Dept. of CS&T, Tsinghua University,

Beijing, China, 100084

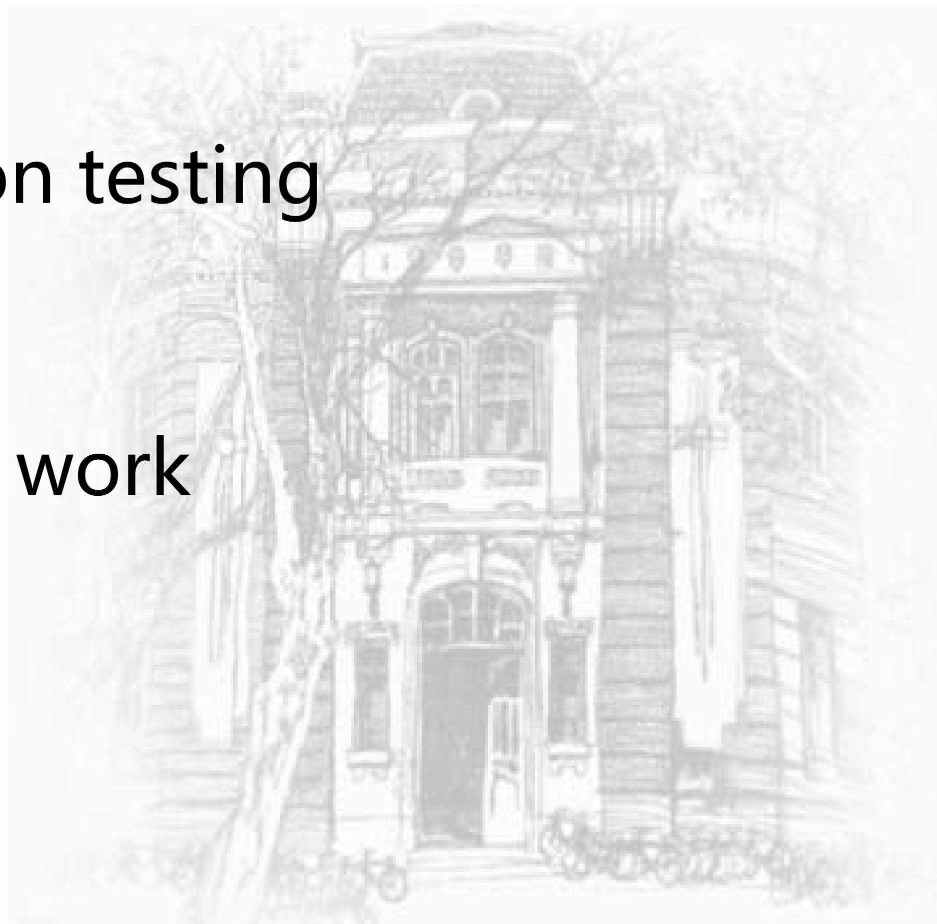
Bo MA and Yunzhan GONG

Research Institute of Networking Technology, BUPT,

Beijing, China, 100876

Outline

- Research motivation
- Test agent design
- Agent-based simulation testing
  - Performance testing
  - Coverage testing
- Conclusion and future work

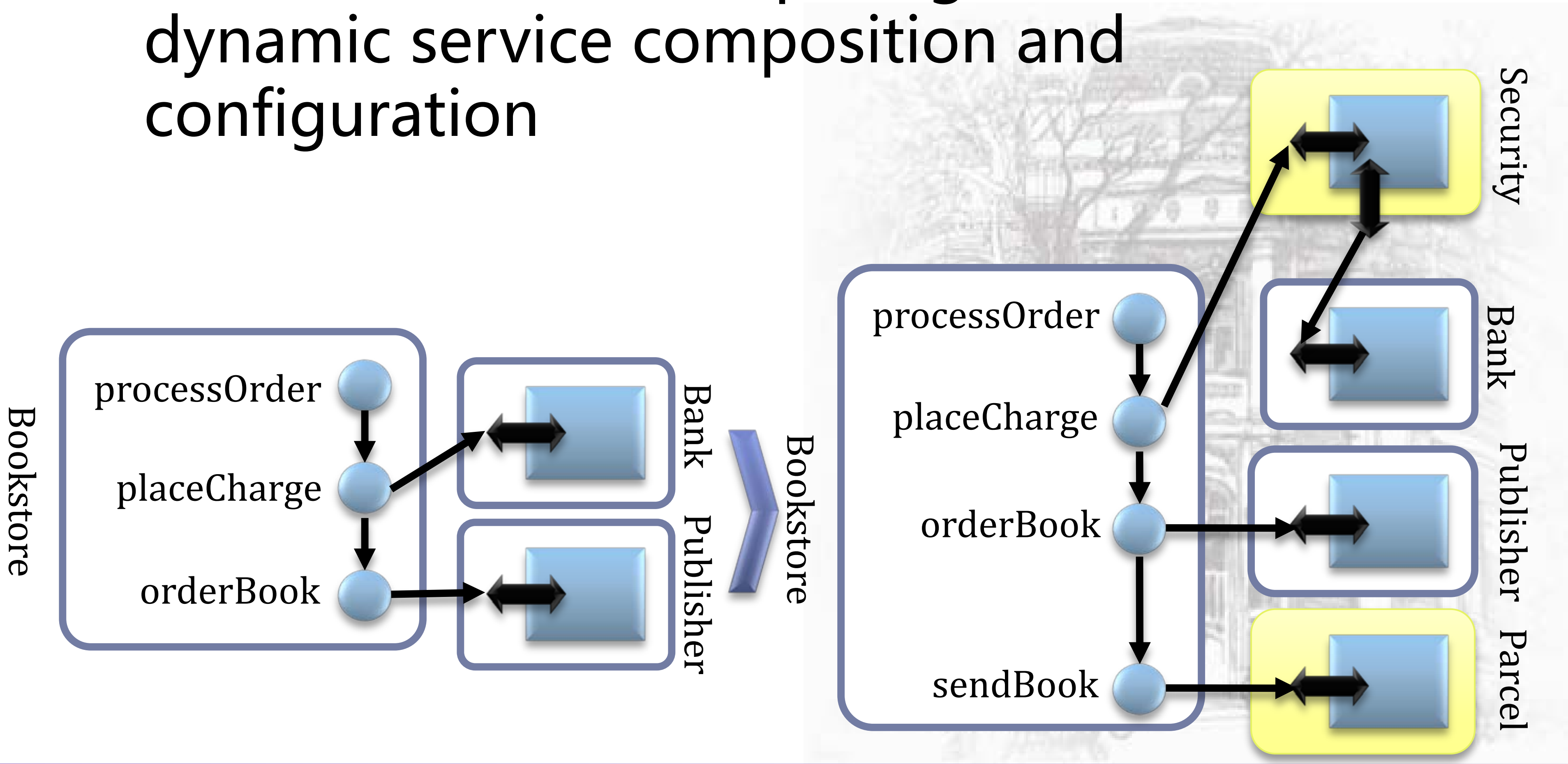


Outline

- Research motivation
- Test agent design
- Agent-based simulation testing
  - Performance testing
  - Coverage testing
- Conclusion and future work

Dynamic Architecture

- Service-oriented computing enables dynamic service composition and configuration



How to Test Dynamic Changes?

- To revalidate the re-composed and re-configured service-based systems
  - Re-select test cases
  - Re-schedule test execution
  - Re-deploy test runners
  - ….
- The challenges: changes occur **ONLINE**
  - Uncontrolled
  - Un-predictable
  - Distributed

New Testing Capabilities Required

- Adaptive testing
  - The ability to sense changes in target software systems and environment, and to adjust test accordingly.
- Dynamic testing
  - The ability to re-configure and re-compose tests, and to produce, on-demand, new test data, test cases, test plan and test deployment.
- Collaborative testing
  - The ability to coordinate test executions that are distributed dispersed.

The MAST Framework

- Multi-Agent based Service Testing Framework [Bai06, Xu06, Bai07, Ma10]
  - MAS is characterized by persistence, autonomy, social ability and reactivity
  - Test agents are defined to simulate distributed service testing
    - Test Runners simulate autonomous user behaviors
    - Runners are coordinated to simulate diversified usage scenarios

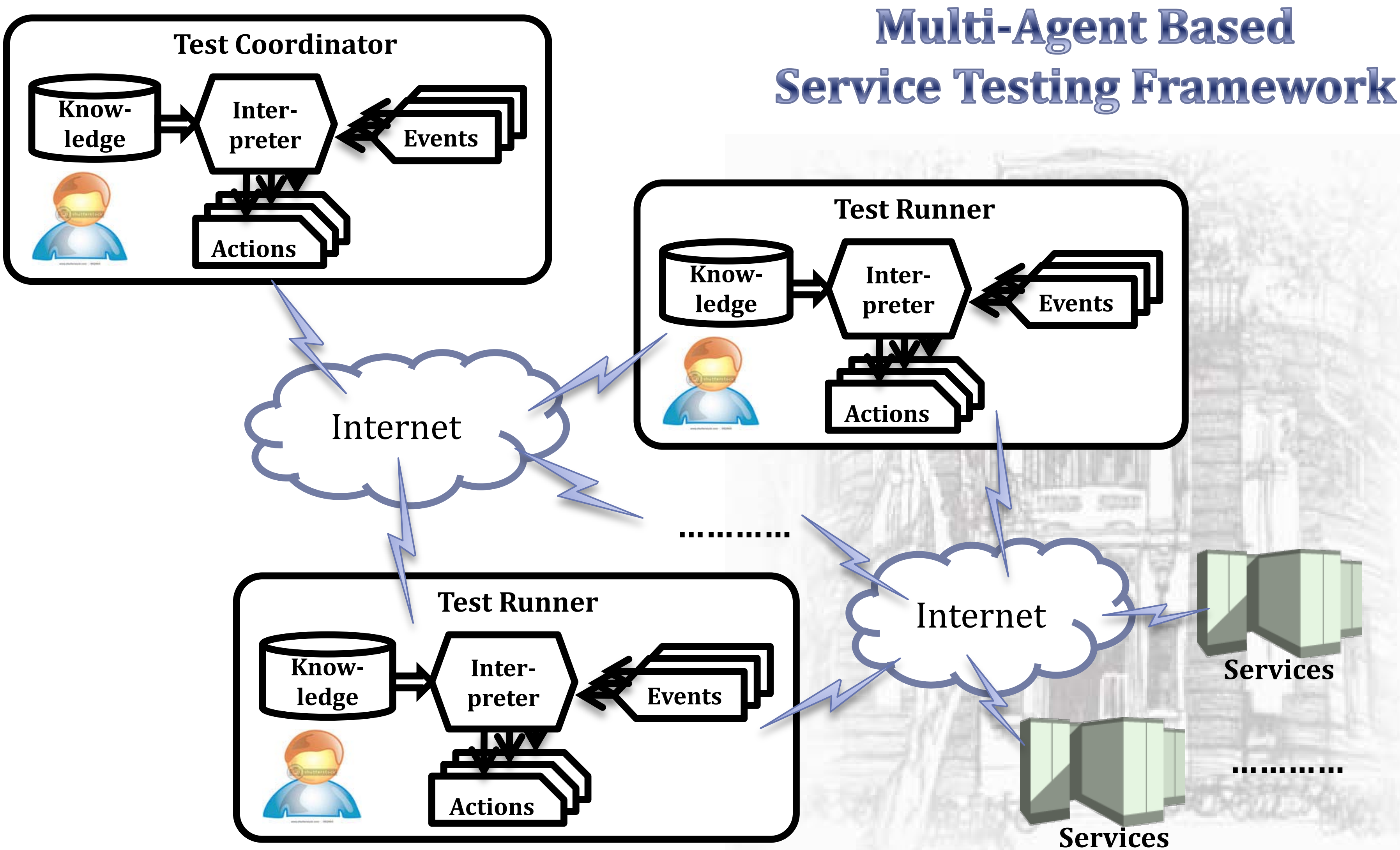
Agent Intelligence is Key to Test Effectiveness

- How to simulate users behavior?
- How to sense and react to changes?
- How to collaborate to simulate various scenarios?

The Needs

Environment Knowledge Representation
Change Events Capturing
Adaptation and Collaboration Rules

Architecture Overview



Outline

- Research motivation
- Test agent design
- Agent-based simulation testing
  - Performance testing
  - Coverage testing
- Conclusion and future work

Basic Test Agent Definition

$$TestAgent := < K, E, A, \Phi >$$

- K: the set of knowledge
- E: the set of events
- A: the set of agent actions
- Φ: the interpreter that derives an agent’s action sequences based on its knowledge and triggering events

Two Agent Types

- Test Coordinator
  - Analyze test requirements, generate test plans, create test runners, and allocate tasks to test runners.
- Test Runner
  - Accept test cases, carry test tasks to target host computers, and exercise test cases on the service under test.

Test Coordinator

- Knowledge
  - <Services, TestCases, Runners, Tasks>
    - Runners:=<ID, state, task>
    - Tasks:=<sID, tcID, result>
- Actions
  - Test Preparation
    - ParseTestScenario, GenerateRunner
  - Test Execution
    - SelectRunner, SelectTestCase, AllocateTestTask, DeployRunner
- Events
    - TEST_PARSED_OK, TEST_PARSED_ERROR
    - START_TEST
    - RUNNER_OK, RUNNER_NOT_AVAILABLE, GENERATE_RUNNER_COMPLETE
    - RUNNER_REQUEST_TASK, RUNNER_SEND_RESULT, RUNNER_UPDATE

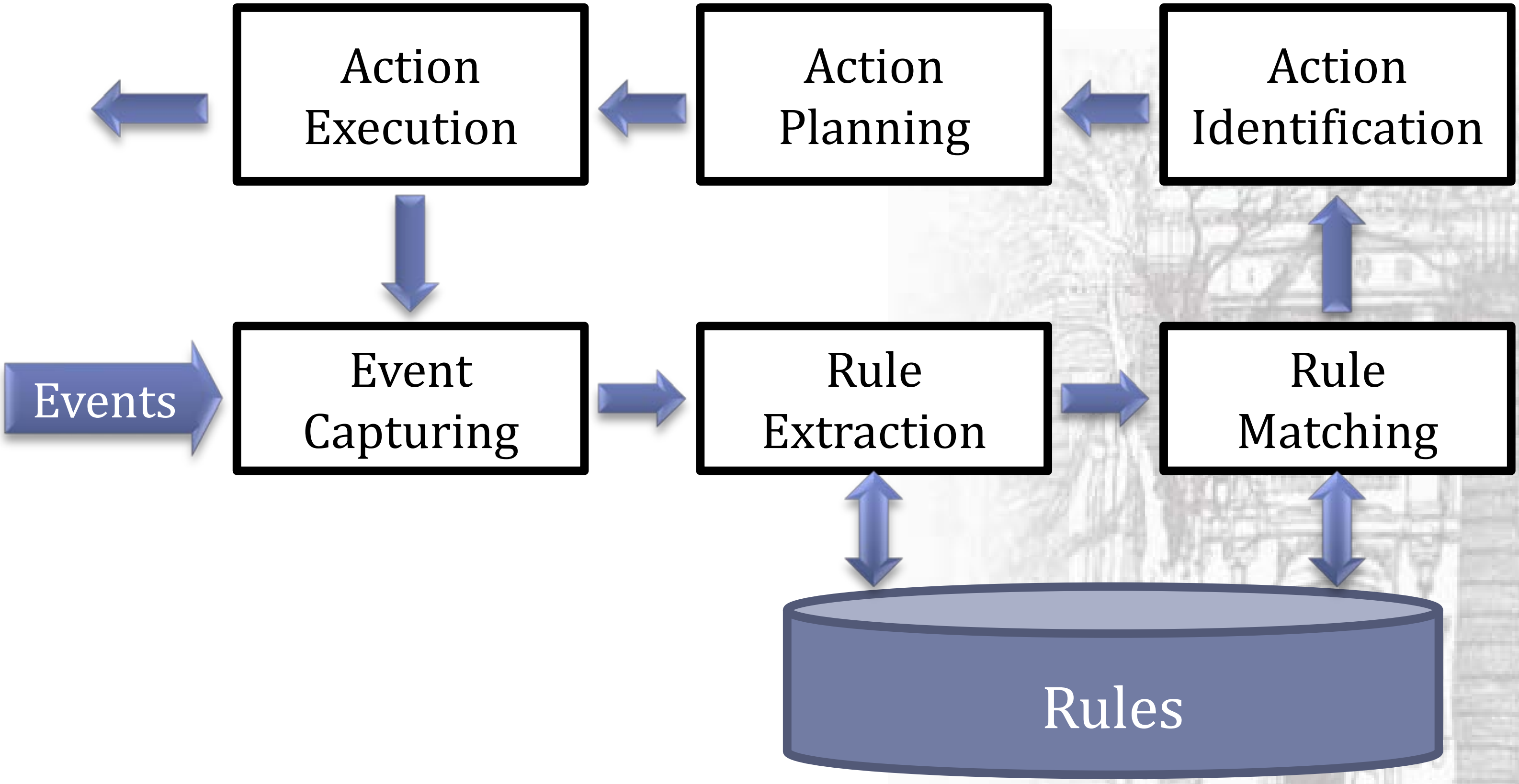
Test Runner

- Knowledge
  - <Hosts, Task, Configuration>
    - Hosts:=<URL, Resources>
    - Configuration:=<hID, tID, Parameters>
- Actions
  - Coordination
    - AcceptTask, ReturnResult, SyncState
  - Execution
    - Migrate, ExecuteTask, CollectResult
  - Decision
    - SelectHost, RequireTestTask, ConfigTest
- Events
    - Task_Arrival, Task_Finish
    - Resource_Error, Host_Error, Host_Update
    - Migration

Interpreter

- Action rules identify the actions to be triggered when certain events occur.
  - assertion → action
    - assertion: predicates of system status after event occurs
- To dynamic adjust behavior according to pre-defined rules and strategies
  - Agent decision making
  - Reactive to changes
  - Adaptive behavior

Interpreter



Outline

- Research motivation
- Test agent design
- Agent-based simulation testing
  - Performance testing
  - Coverage testing
- Conclusion and future work

Agent-Based Simulation Testing

- The generic agent design can be applied to various testing tasks with specially designed domain knowledge, events, actions, and rules.
- Test agents automatically adjust test plans and test cases to meet test objectives.

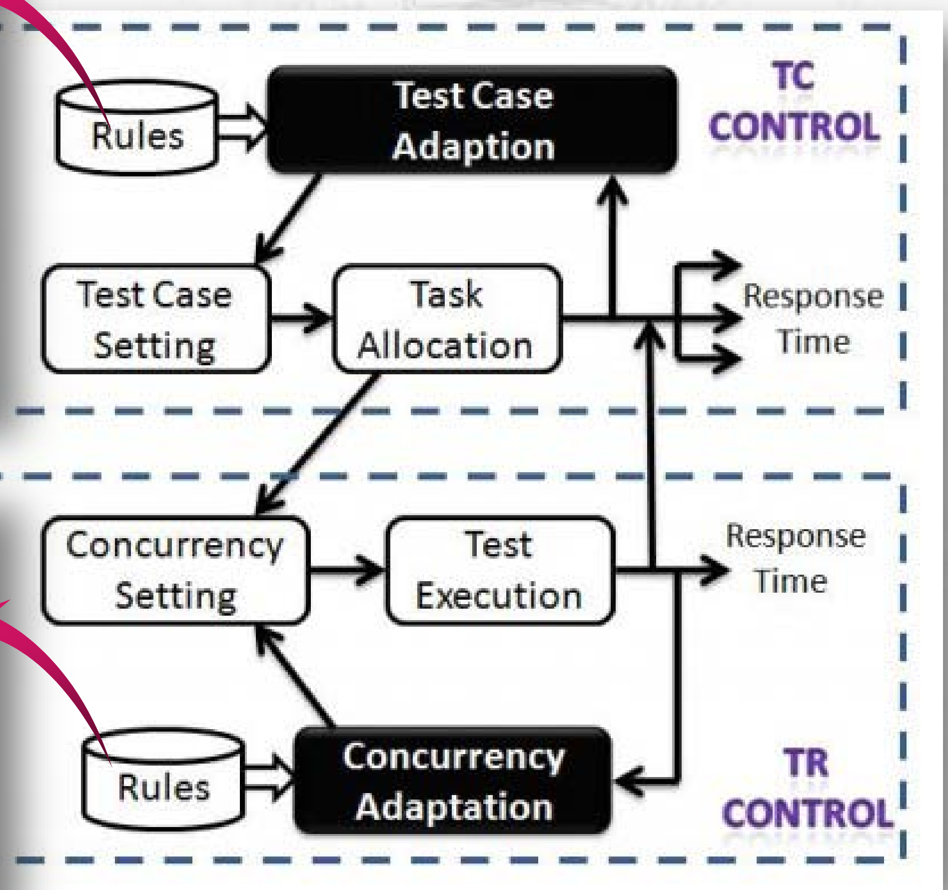
Case Study 1: Performance Testing

- Performance testing analyzes system behavior under different usage scenarios and workloads.
  - E.g. upper limit of capacity and bottlenecks under extreme load
- Two key parameters
  - Test scenarios, the complexity of test cases
  - Workloads, the number of concurrent requests
- Case study objective
  - Try-and-test manual approach → Agents autonomous decision for adaptive selection of scenarios and workloads

Case Study 1: Agent Design

Event	Rules	
	Condition	Action
Start		FindComplexity (cMin, cMax)
Over Load	greater(this.cCur, cMin)∧ null(runner.lCur)	FindComplexity (cMin, cCur)
	greater(this.cCur, cMin)∧ equals (runner.lCur, lMin)	Decrease-Complexity (cCur))
Light Load	less(this.cCur, cMax)∧ null (runner.lCur)	FindComplexity (cCur, cMax)
	less(this.cCur, cMax)∧ equals (runner.lCur, lMax)	Increase-Complexity (cCur))

$$workload = \sum f(complexity_i) \times load_i$$

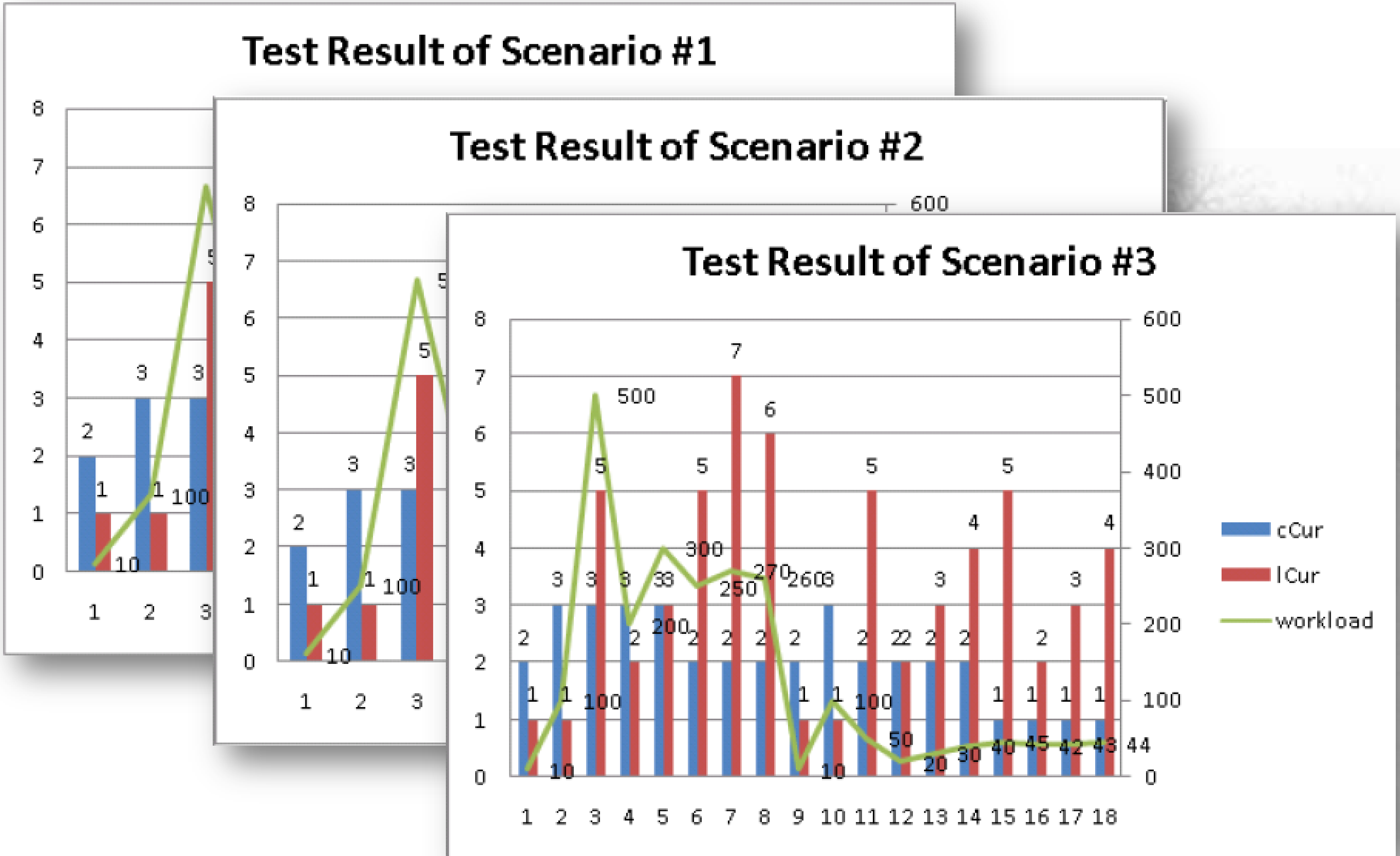


Event	Rules	
	Condition	Action
Over Load	greater(this.lCur, lMin)	FindLoad (lMin, lCur)
	equals(this.lCur, lMin)	Decrease-request()
Light Load	less(this.lCur, lMax)	FindLoad (lCur, lMax)
	equals(this.lCur, lMax)	Increase-request()

Case Study 1: Experiments

- Analyze the SUT’ s memory usage: read file and merge data in memory
  - Services deployed on Tomcat application server.
  - Scenario #1
    - Service is implemented using Java “StringBuilder” data type with little extra memory space.
  - Scenario #2
    - Service is implemented using Java “String” data type which takes up extra memory space for object construction.
  - Scenario #3
    - Simulate changes in server memory configuration of memory restrictions.

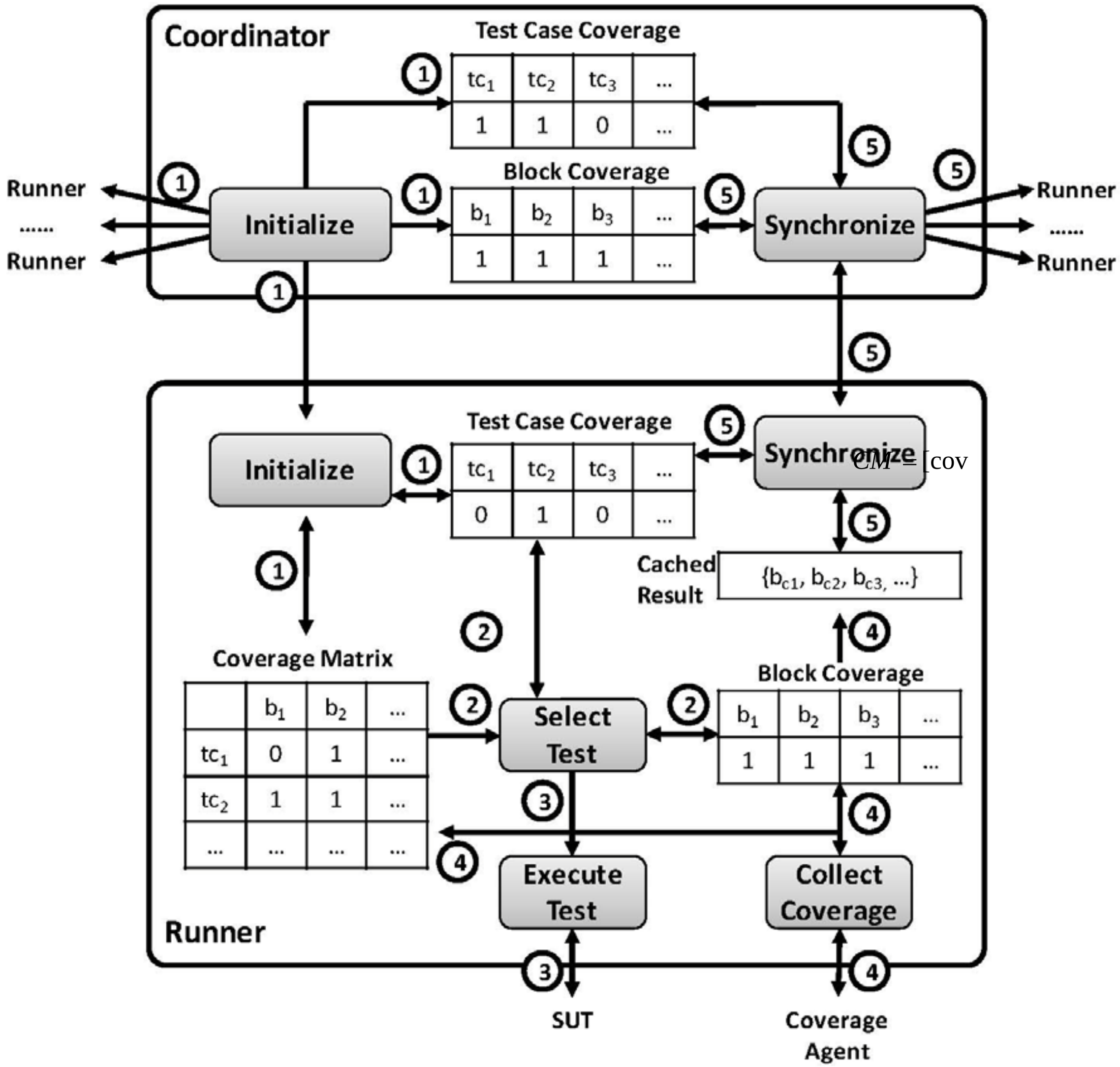
Case Study 1: Results



Case Study 2: Coverage Testing

- Coverage testing is to select a subset of test cases to cover as many as software features.
- The problem
  - TestEfficiency = number of features covered / number of test cases selected
- Case study objective
  - To coordinate test agents working in parallel with complementary coverage achievements

Case Study 2: Agent Design



- Coverage Matrix

$$CM = [\mathrm{cov}_{ij}]_{m \times n},$$

$$\mathrm{cov}_{\mathrm{ij}} = \begin{cases} 1, & \mathrm{b}_{\mathrm{j}} \in Cov(tc_i) \\ 0, & \mathrm{b}_{\mathrm{j}} \notin Cov(tc_i) \end{cases}$$

- Similarity algorithm is used to calculate the distance between any two coverage sets.

$$Dis(s_i, s_j) = 1 - \frac{|s_i \cap s_j|}{|s_i \cup s_j|}$$

Case Study 2: Experiments

- Two SUTs are exercised, each has 100 code blocks and 1000 test cases.
  - Scenario #1: test cases are sparsely overlapped, and each case has a low coverage (2%)

    $$|Cov(tc_i) \bigcap Cov(tc_j)| \le 1\%$$

  - Scenario #2: test cases are densely overlapped

    $$|Cov(tc_i) \bigcap Cov(tc_j)| \ge 20\%$$

- 10 runners are deployed for each test.
  - Initialized with a randomly selected set of test cases
  - Runner cache result threshold: 3
  - Coordinator synchronize threshold: 9

Case Study 2: Results

Scenario #1

Test Rounds	Runner1	Runner2	Runner3	Runner4	Runner5	Runner6	Runner7	Runner8	Runner9	Runner10
#1	10	10	10	10	10	10	10	10	10	10
#2	20	20	20	20	20	20	20	20	20	20
#3	30	30	30	30	30	30	30	30	30	30
Coordinator Sync	96	96	96	96	96	96	96	96	96	96
#4	98	98	98	98	98	99	98	98	99	98
#5	100	99	99	99	99	100	99	99	100	99
#6		100	100	100	100		100	100		100

Scenario #2

Test Rounds	Runner1	Runner2	Runner3	Runner4	Runner5	Runner6	Runner7	Runner8	Runner9	Runner10
#1	10	10	10	10	10	10	10	10	10	10
#2	18	18	18	18	18	18	18	18	18	18
#3	22	22	22	22	22	22	22	22	22	22
Coordinator Sync	77	77	77	77	77	77	77	77	77	77
#4	80	80	81	81	81	81	80	80	80	80
#5	83	83	84	84	84	84	83	83	83	83
#6	86	86	86	86	87	87	86	86	86	85
Coordinator Sync	100	100	100	100	100	100	100	100	100	100
#7	100	100	100	100	100	100	100	100	100	100

Case Study 2: Results

Test Rounds	Runner1	Runner2	Runner3	Runner4	Runner5	Runner6	Runner7	Runner8	Runner9	Runner10
#1	10	10	10	10	10	10	10	10	10	10
#2	19	19	20	19	18	19	19	19	20	17
#3	26	26	29	27	27	29	25	26	27	24
#4	33	31	36	35	34	35	33	33	33	33
...	...	...	...	...	...	...	...	...	...	...
#28	97	93	95	96	95	98	95	93	93	96
...	...	...	...	...	...	...	...	...	...	...
#35	99	100	99	98	96	98	98	96	98	99
...	...	...	...	...	...	...	...	...	...	...
#55	100	100	100	100	100	100	100	100	100	100

Outline

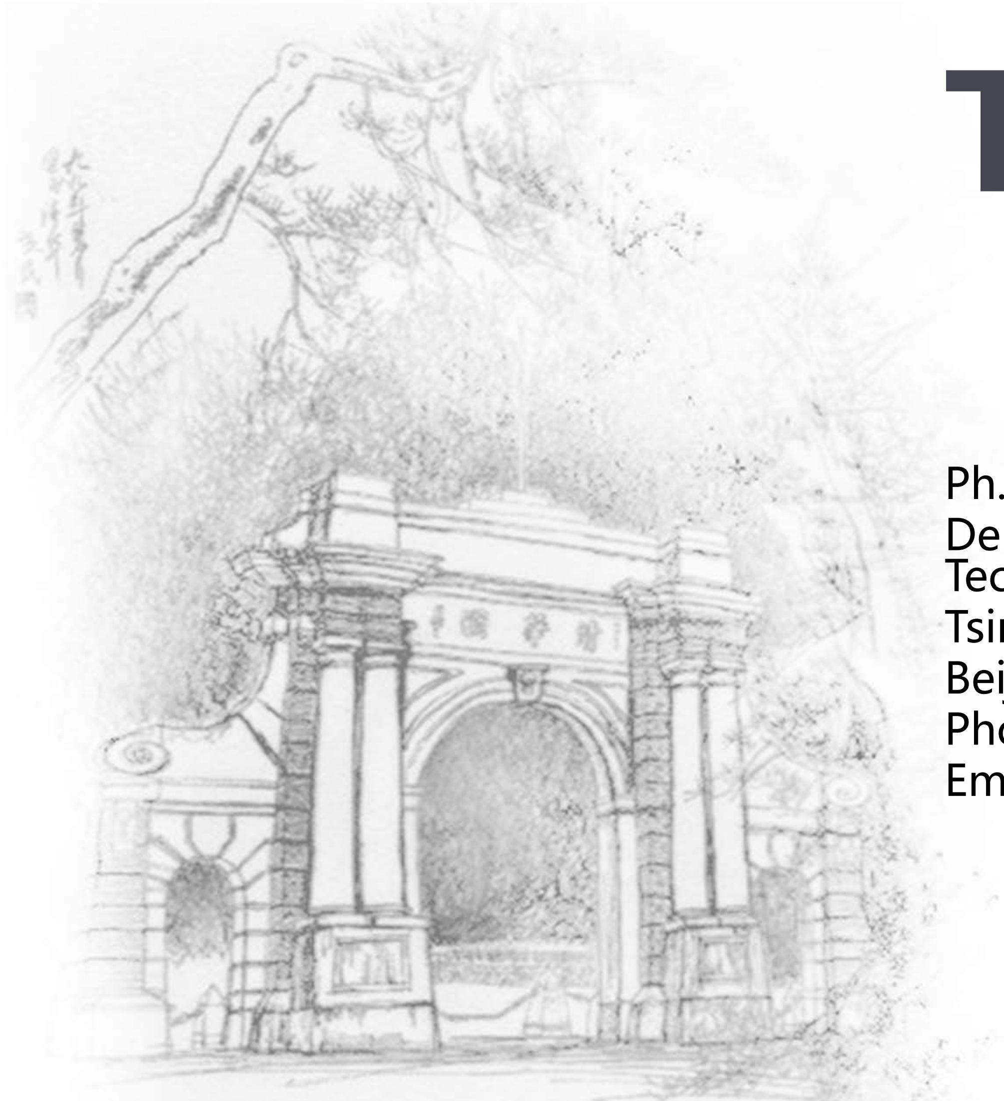
- Research motivation
- Test agent design
- Agent-based simulation testing
  - Performance testing
  - Coverage testing
- Conclusion and future work

Conclusion

- SOA systems impose new requirements of automatic and collaborative testing.
- Agent-based simulation provides a new way for SOA testing
  - Distributed deployment and dynamic migration
  - Autonomous user behavior
  - Collaborative usage scenario
  - Adaptive to environment changes
- Abstract agent model to be instantiated to address different testing tasks
- Experiments show promising improvements compared with conventional approaches

Future Work

- Agent design
  - Joint intention model for agent collaboration
- Improvement of experiments
  - Scale and complexity
- Simulation on the cloud infrastructure



Thank you!

Xiaoying Bai

Ph.D, Associate Professor
Department of Computer Science and Technology,
Tsinghua University
Beijing, China, 100084
Phone: 86-10-62794935
Email: baixy@tsinghua.edu.cn