The 3rd Workshop on Automatic Service Composition -- CASCON 2011 --

Identifying Distributed Features in SOA by Mining Dynamic Call Trees
ICSM 2011

Anis Yousefi
McMaster University
yousea2@mcmaster.ca

Kamran Sartipi
Associate Professor
Faculty of Engineering and Applied Science
University of Ontario Institute of Technology (UOIT)
Kamran.Sartipi@uoit.ca
http://faculty.uoit.ca/sartipi/

November 7, 2011

UOIT

1

Introduction

Distributed nature and change over time of service computing impose new challenges on identifying the quality of services for effective service selection and composition.

We propose a technique for identifying distributed features by mining scattered dynamic call-trees

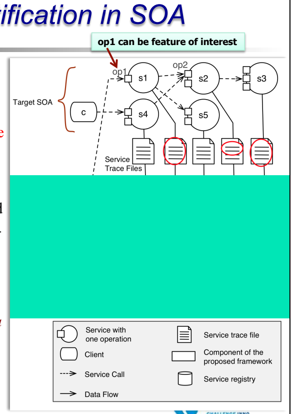
- Complexities of distributed features:
  - Feature locations may change due to change of input parameters.
  - Execution traces are scattered among different service provider platforms.
  - Trace files contain interleaving of execution traces related to different concurrent service users.
- Proposed solution:
  - Define different sets of feature-specific scenarios and execute on SOA.
  - Collect and aggregate relevant distributed execution traces.
  - Mine the resulting dynamic call trees to spot: "feature-specific", "omnipresent", and "noise" patterns.
  - Use metrics to identify the structural properties of the services.

UOIT

2

Framework for Feature Identification in SOA

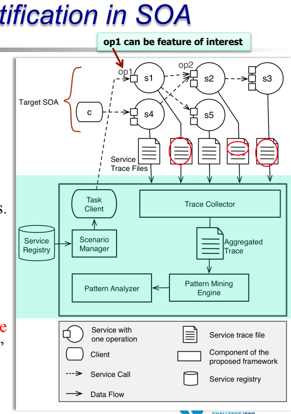
- Feature of interest: in SOA environment, an enterprise application consists of one or more service operations. A service operation can be a specific feature of interest, e.g., **op1 of s1.**
- Goal: to run feature-specific scenario sets to generate distributed execution traces, and identify the execution pattern of the feature.
- Feature-specific scenario set: a group of task scenarios that all share a specific feature. Example: in banking service "*deposit into a bank account*", the operation "*entering the amount of money*" can be the specific feature (**op1**).



3

Framework for Feature Identification in SOA

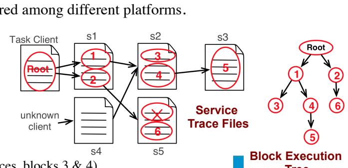
- Scenario Manager: configures "Task Clients" to execute "*feature-specific scenario sets*" on the SOA system.
- Trace Collector: collects and aggregates distributed execution traces.
- Pattern Mining Engine: discovers frequent sub-trees from dynamic call-trees of different execution traces.
- Pattern Analyzer: identifies "feature specific" patterns from "omni-present" and "noise" patterns, and applies metrics.



4

Challenges in Dynamic Analysis of SOA

- Deterministic vs. Non-deterministic features:
  - Deterministic: behavior of the feature is independent of input or state of the system. This feature always generates the same call tree.
  - Non-deterministic: produces different call trees depending on the state and input of the system. We define different Cases, where in each case some conditions are imposed such that the feature shows the same behavior. E.g., *withdraw money when account exists and balance is enough*.
- Distribution of traces: execution of a scenario may involve several services, and the traces are scattered among different platforms.
- Concurrency of events: a service is used by several concurrent users.
- Trace annotations:
  Time (before/after relation)
  Name (caller/callee relation)
  Frequency (distinguish concurrent traces, blocks 3 & 4)



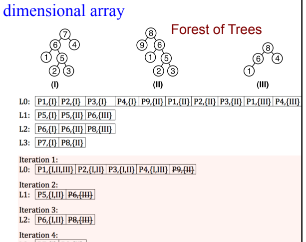
5

Mining Dynamic Call-trees:
"Mining Frequent Sub-tree in a Forest"

Running feature-specific scenario sets produces "Forest of Dynamic Call Trees"

A **frequent subtree** is a subtree T′ such that the "cardinality" of its super-trees T's (namely, "support set" of T′) is greater than or equal to a given "threshold value" (i.e., minimum support).

- Frequent sub-tree mining algorithm builds bottom-up sub-trees.
- Forest of trees is represented by a two dimensional array
- Example with four iterations of the mining process:

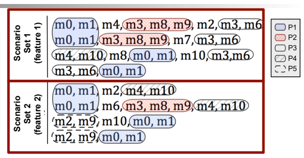


- Each array entry (e.g., P1{I}) consists of a pointer to the root of a subtree (P1) and the subtree's support set ({I}).
- Tree "I" is represented as string 761052030004.
- Minimum support threshold is two.

6

Analysis of Extracted Patterns

Traces associated with two feature-specific scenario sets and their corresponding patterns. For simplicity we have serialized each trace based on the time of method entries.



- Feature-Specific Patterns correspond to the shared feature of a scenario set: (**feature 1**: P3, deterministic), (**feature 2**: P4 & P5, non-deterministic)
- Omnipresent patterns are common among all scenario sets: (pattern P1 in both sets)
- Noise patterns are random: (pattern P2 in both sets)

		p1	p2	p3	p4	p5
Feature 1	t1	√	√	√		
	t2	√	√	√		
	t3	√		√	√	
	t4	√		√		
Feature 2	t5	√			√	
	t6	√	√		√	
	t7	√				√
	t8	√				√

Structural Analysis of SOA

- *Service Utilization*: assessing a service based on how it's operations utilize other SOA services.

  S_{op}: set of services contributing in the execution of operation "op".
  M_{op}: set of methods contributing in the execution of operation "op".
  M_{si}: $\{m_1,m_2, ..,m_o\}$ set of all methods in service "si".

$$SU(op) = \frac{1}{|S_{op}|} * \sum_{s_i \in S_{op}} \frac{|M_{s_i} \cap M_{op}|}{|M_{s_i}|}$$

- *Call Frequency*: communication overhead of a service.

  P_{op}: $\{p_1, p_2, ..., p_l\}$ set of feature-specific patterns for operation op.
  S_{pi}: $\{s_1, s_2, ..., s_m\}$ set of services contributing in execution of pattern pi.
  $OP_{si,pj}$: $\{op_1, op_2, .., op_n\}$ set of operations that are defined in the interface of service si and called in pattern pj .

$$CF(p_k) = \sum_{i=1}^{|S_{p_k}|} \sum_{j=1}^{|OP_{s_i,p_k}|} number\ of\ calls\ to\ s_i.op_j$$

$$CF_{avg}(op) = \frac{\sum_{i=1}^{l} CF(p_i)}{|P_{op}|}$$

Case Study:
Service Oriented Banking

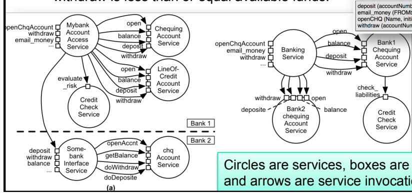
Two features: Withdraw & Email-money

- Scenarios in each feature-specific set are partitioned into a number of **cases**, where each case examines a service operation in a specific context.
- Scenarios W9 and W10 examine operation "withdraw" in the case that the account to be withdrawn from exists and the amount to withdraw is less than or equal available funds.

Feature	Case	Scenario#	Scenario
(1) Withdraw	Invalid account number	W1	Withdraw(-100,500);
		...	...
	Account does not exist	W3	Withdraw(1000,500);
		...	...
	Sufficient funds is not available	W5	n = openCHQ ("John", 0); withdraw (n, 500);
		W6	n = openCHQ ("John", 500); deposit (n, 2000); withdraw (n, 3000);
		...	...
	Sufficient funds available	W9	n = openCHQ ("John", 500); withdraw (n, 500);
		W10	n = openCHQ ("John", 500); deposit (n, 2000); withdraw (n, 1000);
		...	...
(2) Email_Money	Account to deposit from does not exist	EM1	email_money ("Bank1", 3000, "Bank2", 3000, 1500);
	Accounts exist, Sufficient funds available	EM2	n = openCHQ ("John", 1000); email_money ("Bank1", n, "Bank2", 12, 1500);
	...	...	...
...	...	...	...

deposit (accountNumber, funds)
email_money (FROMbankName, FROMaccountNumber, TObankName, TOaccountNumber, funds)
openCHQ (Name, initialFunds)
withdraw (accountNumber, funds)

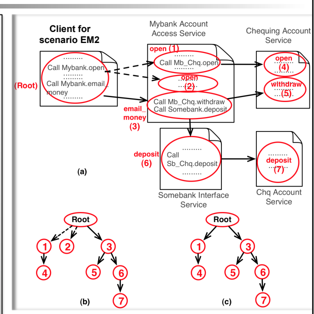
Part of the scenarios used to examine "Mybank Account Access Service".



Circles are services, boxes are operations, and arrows are service invocations.

Case Study:
Service Oriented Banking (frequent analysis)

- Blocks 1 and 2 belong to different scenarios.
- We ran scenario EM2, 5 times; we counted 5 instances of block 1 and 2 instances of block 2. Therefore, block 2 is irrelevant to scenario EM2 and must be deleted from (b).
- In (c), applying depth first traversal on the resulting block execution tree. The order of the blocks for aggregation is 1, 4, 3, 5, 6, 7.
- Trace Collector replaces each instance of a remote service call with the respective service trace block to generate an execution trace for EM2.



(a) Trace files and blocks for scenario EM2.
(b) Block execution tree before frequent analysis.
(c) After frequency analysis. Block 2 is deleted.

Feature-specific Patterns

- Feature specific patterns extracted from the execution of withdraw on the first system in page 8.
- For simplicity we serialized each pattern by noting only the sequence of method entries.
- Each pattern is a sequence of method invocations separated by",".
- Each method is specified as: *package-Name/className.methodName*.

pattern 1: mybank/accessinterfacestub.withdraw, mybank/accessinterfacestub.optimizecontent, mybank/accessinterfacestub.toenvelope, mybank/accessinterface.<init>, mybank/accessinterface.withdraw, mybank/accessinterface.accountservicefor, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.populateaxisservice, com/ibm/chqaccntimplstub.getuniquesuffix, com/ibm/chqaccntimplstub.populatefaults, com/ibm/chqaccntimplstub.withdraw, com/ibm/chqaccntimplstub.optimizecontent, com/ibm/chqaccntimplstub.toenvelope, com/ibm/chqaccntimpl.<init>, com/ibm/chqaccntimpl.withdraw, com/ibm/chqaccntimpl.ischequing, com/ibm/chqaccntimpl.validaccount, com/ibm/chqaccntimpl.balance, com/ibm/chqaccntimpl.ischequing, com/ibm/chqaccntimplstub.getenvelopenamespaces, com/ibm/chqaccntimplstub.fromom, mybank/accessinterfacestub.getenvelopenamespaces, mybank/accessinterfacestub.fromom

pattern 2: mybank/accessinterfacestub.withdraw, mybank/accessinterfacestub.optimizecontent, mybank/accessinterfacestub.toenvelope, mybank/accessinterface.<init>, mybank/accessinterface.withdraw, mybank/accessinterface.accountservicefor, mybank/accessinterfacestub.getenvelopenamespaces, mybank/accessinterfacestub.fromom

pattern 3: mybank/accessinterfacestub.withdraw, mybank/accessinterfacestub.optimizecontent, mybank/accessinterfacestub.toenvelope, mybank/accessinterface.<init>, mybank/accessinterface.withdraw, mybank/accessinterface.accountservicefor, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.<init>, com/ibm/chqaccntimplstub.populateaxisservice, com/ibm/chqaccntimplstub.getuniquesuffix, com/ibm/chqaccntimplstub.populatefaults, com/ibm/chqaccntimplstub.withdraw, com/ibm/chqaccntimplstub.optimizecontent, com/ibm/chqaccntimplstub.toenvelope, com/ibm/chqaccntimpl.<init>, com/ibm/chqaccntimpl.withdraw, com/ibm/chqaccntimpl.ischequing, com/ibm/chqaccntimpl.validaccount, mybank/accessinterfacestub.getenvelopenamespaces, mybank/accessinterfacestub.fromom

Service Utilization & Call Frequency

Execution pattern mining and analysis for operation "withdraw".

Architecture	Number of Scenarios	Number of Cases	Number of Patterns	Average Trace Size	Average Pattern Size	Number of Feature-specific Patterns	Average Feature-specific Pattern Size
Figure 6 (a)	12	4	26	51	23	3	19
Figure 6 (b)	12	4	22	55	25	4	22

Service utilization of operation "withdraw" from "Mybank Account Access Service" and "Banking Service".

Architecture	Number of Services	Service Utilization
Figure 6 (a)	2	0.41
Figure 6 (b)	3	0.49

Architecture	Feature-specific Pattern	Total Frequency of Remote Calls	Number of Feature-specific Patterns	Call Frequency
Figure 6 (a)	p1	2	3	1.7
	p2	1		
	p3	2		
Figure 6 (b)	p1	1	4	2
	p2	2		
	p3	2		
	p4	3		

Call frequency of operation "withdraw" from "Mybank Account Access Service" and "Banking Service".

Conclusion

- Identifying software features in distributed web services is much more complex than that of monolithic systems.
  - Deterministic vs. non-deterministic features
  - Distributed traces in different platforms
  - Concurrency of traces in different service platforms.
- We proposed mechanisms for: i) collecting and aggregating distributed traces; ii) mining feature-specific and omni-present patterns for non-deterministic features; assessing the structural merits of a SOA-based system using pattern-mining results.
- The proposed approach assists service selection for composition via:
  - Examining the structural quality of new web services.
  - Identifying services which contain buggy features or increase network traffic.
- The proposed approach requires close collaboration among service providers and service users.

The 3rd Workshop on Automatic Service Composition
-- CASCON 2011 --

Identifying Distributed Features in SOA by Mining Dynamic Call Trees
ICSM 2011

Anis Yousefi
McMaster University
yousea2@mcmaster.ca

Kamran Sartipi
Associate Professor
Faculty of Engineering and Applied Science
University of Ontario Institute of Technology (UOIT)
Kamran.Sartipi@uoit.ca
http://faculty.uoit.ca/sartipi/

November 7, 2011

Abstract

Distributed nature of web service computing imposes new challenges on software maintenance community for localizing different software features and maintaining proper quality of service as the services change over time. In this paper, we propose a new approach for identifying the implementation of web service features in a service oriented architecture (SOA) by mining dynamic call trees that are collected from distributed execution traces. The proposed approach addresses the complexities of SOA-based systems that arise from: **i)** features whose locations may change due to changing of input parameters; **ii)** execution traces that are scattered throughout different service provider platforms, and ; **iii)** trace files that contain interleaving of execution traces related to different concurrent service users. In this approach, we execute different groups of feature-specific scenarios and mine the resulting dynamic call trees to spot paths in the code of a service feature, which correspond to a specific user input and system state. This allows us to focus on the implementation of a specific feature in a distributed SOA-based system for different maintenance tasks such as bug localization, structure evaluation, and performance analysis. We define a set of metrics to assess structural properties of a SOA-based system. The effectiveness and applicability of our approach is demonstrated through a case study consisting of two service-oriented banking systems.