Exploring Use of Symbolic Execution for Service Analysis

Yang Zhang[1], Long Wang[1,2], Zhengang Wang[3], Dongdong Shangguan[3]
[1]REASONS Lab, Institute for Network Sciences and Cyberspace, Tsinghua University, [2]Zhongguancun Laboratory, [3]Huawei Corporation
y-z21@mails.tsinghua.edu.cn, longwang@tsinghua.edu.cn, {wangzhengang, shangguandongdong1}@huawei.com

***Abstract*— Service computing is growing popular thanks to the cloud computing technology. However, the behavior of cloud services is complicated as they involve a large number of components and complex interactions between them. It is highly helpful for service users and providers to well understand the service behavior if they can exhaustively explore potential component interactions and service scenarios. This paper discusses the challenges associated with this and an approach to achieving this goal. This paper also reports our progress of exploring the use of symbolic execution for service analysis, together with our research plan.**

Keywords—distributed service, symbolic execution, system analysis, code representation

I. Introduction

Nowadays many computer applications run on computing platforms as services. People use the applications by sending them requests to invoke the services. With the growing adoption of advanced technologies such as cloud computing, more and more domains are employing this service computing paradigm.

Typically, an IT service is comprised of multiple components, and the processing of a service request involves executions among the components. When the number of components in a service is large or the interactions of the components are not easy to understand, it is challenging for the service provider to thoroughly understand the service execution scenarios, including various error-present scenarios.

We believe that exhaustive or extensive explorations of potential interactions between the service components and potential scenarios of the service execution, including error-present scenarios, are very helpful for a service provider and computing platform provider (e.g. cloud provider) to well understand and analyze the service behavior.

Currently there is no approach that can perform such exhaustive or extensive explorations of distributed service behavior. Most of current work analyzes service scenarios of component interactions based on graph. Luo et al. [5] build graphs of invocation relationships in microservices and stochastic models for interactions between microservices. GSMART [6] can automatically generate a service dependency graph to visualize and analyze the dependencies between microservices and services. They capture only coarse-granularity behavior of services, and do not allow for exhaustive explorations of execution scenarios in fine granularity, e.g. message between component processes.

Other service analysis approaches include i) traditional availability models like Reliability Block Diagrams [1], Monte Carlo methods [2][3], Bayesian networks [4], and Markov models, ii) customized models for certain systems only [7][8][9], iii) representations/specifications of systems in formats such as UML [10] and its varieties like SysML [11] and Candy [12], and iv) systematic testing tools like dBug [19] and DeMeter [20]. These existing approaches do not provide the desired capability of exhaustive explorations of service execution scenarios, or require experts to create the specific service's detailed model manually (like Markov models) based on the experts' comprehensive understandings of the service behavior.

Fine-grain exhaustive analysis exists in technologies such as static analysis of programs, symbolic execution, and model checking. Can we employ such technologies for analyzing IT services? We plan to explore the potential of this direction. As the first venture, we explore using symbolic execution for service analysis in this paper.

Symbolic execution has been applied in program analysis for exhausting program execution paths [14-18]. We can leverage the technology for exhaustive or extensive explorations of execution paths in service computing, based on an observation that there is an analogy between IT services and object-oriented programs. Here we mention one similarity: the instantiation relationship between a service component and its replicas is similar to that between an object class and its instances. This analogy is discussed in details in Section II.

We see the following challenges should be addressed for using symbolic execution to analyze services.

i) How to take advantage of the analogy between IT services and object-oriented programs? Symbolic execution is unable to be directly applied to an IT service. We should find a way to make it possible. To address this challenge, we invent an algorithm that transforms the behavior of a distributed service, in terms of component interactions, into program code representation.

ii) How to analyze service behavior using symbolic execution with the analogy exploited? After the service behavior is transformed into program code, we then need to translate a target problem or property of the service under study into a form in program code that symbolic execution is able to handle.

iii) Dealing with issues that are associated with symbolic execution. As the size of the target service or the problem under study scales up, the issues of the symbolic execution technology, e.g. state explosion, should be addressed.

In this paper we report our progress in handling these challenges and our preliminary results that show the feasibility of the proposed approach for service analysis. Specifically, we designed an algorithm to transform distributed service behavior into program code representation (for Challenge i), and conducted a case study of service analysis by applying symbolic execution onto the generated program code (for Challenge ii). Case studies with larger scale services and problems (for Challenge iii) are arranged in our research plan.

II. Analogy Between Services and Programs

We observed an analogy between a distributed service in service computing and an object-oriented program. Table 1 lists the analogy. Then the service behavior can be represented or abstracted by the program. Here we describe the similarities in the analogy.

i) In a service, one kind of component has one or multiple replicas as the instantiation of the component. In an object-oriented program, an object class has one or multiple object instances as its instantiation.

This work has been supported by the NSFC Project of China under Grant 62132009.

Table 1: Analogy between a distributed service and an object-oriented program

Distributed Service	Object-Oriented Program
component	object class
component replica	object instance
sub-component/sub-service	class member variable/field
behavior of a component	member function of a class
processing of a request	processing of a piece of input
orchestration of multiple component replicas (and their subcomponents) for request processing	programming of multiple object instances (and their member variables and functions) for input processing

ii) A replica of a service component may comprise multiple subcomponents or microservices. An object instance may comprise instances of class member objects and fields.

iii) A component replica provides certain functions for the service. An object instance provides certain functions via its member functions for the program.

iv) A service or component processes incoming requests. A program or object class processes input or function parameters.

v) A service processes a request by its components and sub-services. Components and subservices may also have their own subcomponents and subservices do operations on behalf of the request. A program processes input by its objects and their member functions and variables (which are also objects). The objects may also have their own member functions and objects do operations on behalf of the input.

In this sense, the service behavior can be regarded as the execution of a corresponding object-oriented program.

III. Transforming Services into Programs

We use a simple example service to illustrate the essential idea of the service-to-program transformation, and then present the algorithm that does the transformation generally.

A. Example Service

The example is a simplified three-tier service for the illustration purpose, as shown in Figure 1. Typically, the frontend component receives a request from the service client, performs request parsing and invokes the middleware component to perform the real service. Examples of the frontend component are a web server, a load balancer, and an API parser/handler. Application servers running the service logic like Tomcat, and applications providing specific handling like nova-compute and glance-registry in OpenStack, are examples of the middleware component. Sometimes the processing of a service request involves queries of another backend component, e.g. a configuration store or a database.

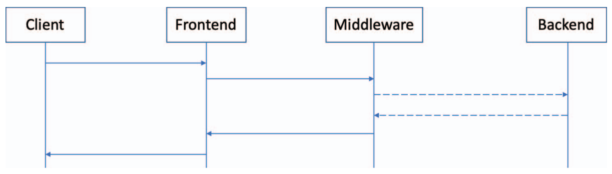


Figure 1: A simple three-tier example service

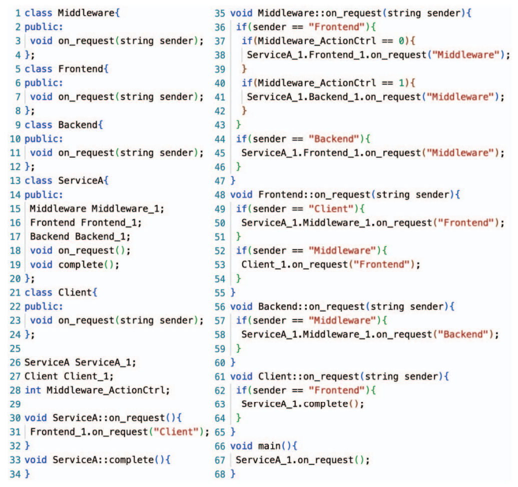
Figure 1 shows the execution paths of the example service in terms of interactions between a service client and the service components. There are two execution paths in the figure (the dash lines indicate optional path segments, i.e. they are only traversed in certain scenarios):

- *Client – Frontend – Middleware – Frontend – Client.*
- *Client – Frontend – Middleware – Backend – Middleware – Frontend – Client.*

B. Program Code Representation of the Example Service

The program code representation of the service is illustrated in Figure 2. We can select any object-oriented programming language for the representation as long as it is convenient for programs in the language to be symbolically executed. In this study we chose C++.

The example service has three types of components, which are defined as object classes in the program code (lines 1-12 in Figure 2). The service itself is defined as a class, with the instances of the three component classes defined as its member variables (lines 13-17). The client is not part of the service, but is also represented in the code as a class (lines 21-24) for capturing the entire request processing flow.



```
 1 class Middleware{
 2 public:
 3  void on_request(string sender);
 4 };
 5 class Frontend{
 6 public:
 7  void on_request(string sender);
 8 };
 9 class Backend{
10 public:
11  void on_request(string sender);
12 };
13 class ServiceA{
14 public:
15  Middleware Middleware_1;
16  Frontend Frontend_1;
17  Backend Backend_1;
18  void on_request();
19  void complete();
20 };
21 class Client{
22 public:
23  void on_request(string sender);
24 };
25
26 ServiceA ServiceA_1;
27 Client Client_1;
28 int Middleware_ActionCtrl;
29
30 void ServiceA::on_request(){
31  Frontend_1.on_request("Client");
32 }
33 void ServiceA::complete(){
34 }
35 void Middleware::on_request(string sender){
36  if(sender == "Frontend"){
37   if(Middleware_ActionCtrl == 0){
38    ServiceA_1.Frontend_1.on_request("Middleware");
39   }
40   if(Middleware_ActionCtrl == 1){
41    ServiceA_1.Backend_1.on_request("Middleware");
42   }
43  }
44  if(sender == "Backend"){
45   ServiceA_1.Frontend_1.on_request("Middleware");
46  }
47 }
48 void Frontend::on_request(string sender){
49  if(sender == "Client"){
50   ServiceA_1.Middleware_1.on_request("Frontend");
51  }
52  if(sender == "Middleware"){
53   Client_1.on_request("Frontend");
54  }
55 }
56 void Backend::on_request(string sender){
57  if(sender == "Middleware"){
58   ServiceA_1.Middleware_1.on_request("Backend");
59  }
60 }
61 void Client::on_request(string sender){
62  if(sender == "Frontend"){
63   ServiceA_1.complete();
64  }
65 }
66 void main(){
67  ServiceA_1.on_request();
68 }
```

Figure 2: Program code representation of the example service

The entire service request processing starts at the service's *on_request()* function in *main()* (line 67), and ends at the client's receiving of the service response, i.e. the service's *complete()* in the client's *on_request()* function (line 63).

The function of each component is the receiving of a request and the invocation of the next component's processing. Each of the three component classes has an *on_request()* member function, and its logic is the identification of the next component for the specific request processing scenario and the invocation of this component's *on_request()* function (lines 35-60). In this example the next component is identified according to the sender of the received message, as shown in lines 36, 44, 49, 52, 57 and 62; e.g., when the backend receives a message from the middleware (line 57), it responds to the middleware, the next component is the middleware, and *Middleware_1.on_request("Backend")* is invoked (line 58).

When the middleware receives message from the frontend, it may or may not query the backend for the processing. This path forking is represented by introducing a variable *Middleware_ActionCtrl* which controls which path to take (lines 37-42). Different values of the variable will be selected during symbolic execution to explore different paths.

C. Transformation Algorithm

We designed an algorithm, given in Figure 3, to automatically transform a service into program code representation. The service behavior we consider is the component interactions (or communications) such as the one shown in Figure 1. The input of the algorithm is service traces that capture the service behavior. Existing tracing technologies [21-25] can be employed to provide the service traces. Specifically, our algorithm uses traces collected by the REPTrace technology [22] as input (the set of traces S_events in Figure 3).

First, we extract the information of the machines and all components, including component instance (i.e. replica) information and host information from input traces, and put them in the *S_machine* and *S_component* sets as well as the *topology* hashtable, respectively (lines 12-20 in Figure 3).

Second, we obtain the information of component actions from the traces. As currently we only focus on component interactions, the action of a component ***c*** indicates which next component to invoke when ***c*** receives different messages from different components like lines 49-54 in Figure 2. We record ***c***'s actions in its *actions* hashtable (line 4 in Figure 3), which maps the sender component of the received message to a list of next component instances to invoke. Lines 22-30 in Figure 3 collect the component actions from the traces.

Finally, we generate the program code. Line 33 generates the code for the definition of the *Machine* class (the class is not shown in Figure 2, but has the host information and is also generated into the code for studying host dependency). Line 35 invokes *generate_component_code()* to generate the code for all components. When ***c*** receives message from the sender, it sends another message to one of the next component instances to invoke its function. The *actions* hashtable is exploited for generating the code of invoking the right next component instance in different scenarios, and a control variable is generated in the code to support path forking in symbolic execution if need be (lines 43-54).

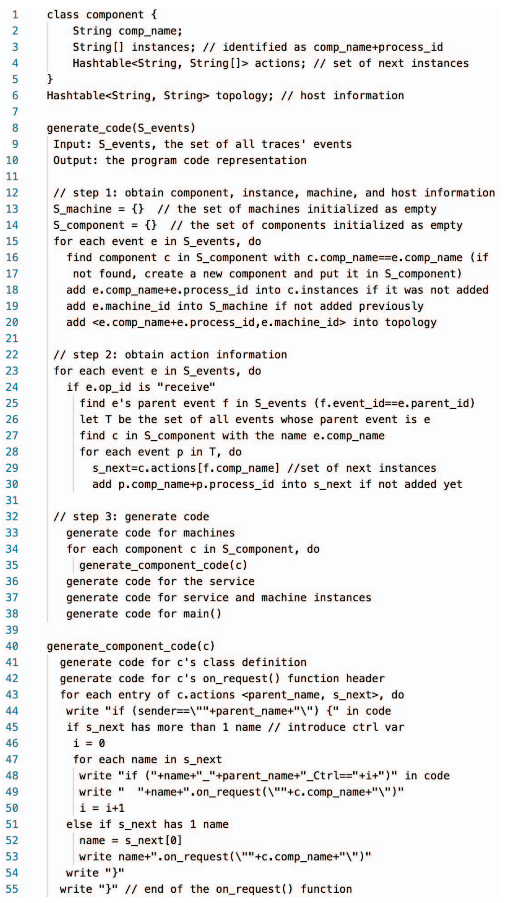
Line 36 generates code of the class definition and member functions for the service, like lines 13-20, 30-34 in Figure 2. When generating the on_request() of the service, we first find all events in S_events that do not have any parent event. If they are the same event in multiple traces, e.g. the same "Client – Frontend" in both execution paths in Section II.A, this event is the only root service invocation. We generate the code of the root service invocation, like line 31 in Figure 2, in the service's on_request(). If there are multiple different events in the multiple traces, which means this service has multiple entry points, we also generate the code of using a control variable to support the multi-entry situation.

Line 37 generates the code for instantiations of the service, its component instances and the machines, based on the information in *S_machine* and *S_component*. We also generate the code of dealing with the host information in line 37. Line 38 generates the *main()* function, the entry of the program.

IV. A Preliminary Case Study of Service Analysis

During cloud maintenance we encountered a task of analyzing cloud service dependencies on relevant service components, particularly identifying which components a service has strong or weak availability dependencies on.

This task motivated our idea of trying to analyze service execution paths extensively or exhaustively so that we are able to measure the impact of a component instance failure onto the service. Although certain other technologies, e.g. fault injection, may also deal with the task, we would rather investigate an alternative approach that exposes the service behavior as a panoramic view and studies the view in a path-exhausting or state-exhausting way. Therefore, we selected service dependencies upon its components as the target property. The preliminary case study was conducted against the example service.



```
1   class component {
2       String comp_name;
3       String[] instances; // identified as comp_name+process_id
4       Hashtable<String, String[]> actions; // set of next instances
5   }
6   Hashtable<String, String> topology; // host information
7
8   generate_code(S_events)
9    Input: S_events, the set of all traces' events
10   Output: the program code representation
11
12   // step 1: obtain component, instance, machine, and host information
13   S_machine = {}  // the set of machines initialized as empty
14   S_component = {}  // the set of components initialized as empty
15   for each event e in S_events, do
16     find component c in S_component with c.comp_name==e.comp_name (if
17      not found, create a new component and put it in S_component)
18     add e.comp_name+e.process_id into c.instances if it was not added
19     add e.machine_id into S_machine if not added previously
20     add <e.comp_name+e.process_id,e.machine_id> into topology
21
22   // step 2: obtain action information
23   for each event e in S_events, do
24     if e.op_id is "receive"
25       find e's parent event f in S_events (f.event_id==e.parent_id)
26       let T be the set of all events whose parent event is e
27       find c in S_component with the name e.comp_name
28       for each event p in T, do
29         s_next=c.actions[f.comp_name] //set of next instances
30         add p.comp_name+p.process_id into s_next if not added yet
31
32   // step 3: generate code
33     generate code for machines
34     for each component c in S_component, do
35       generate_component_code(c)
36     generate code for the service
37     generate code for service and machine instances
38     generate code for main()
39
40  generate_component_code(c)
41    generate code for c's class definition
42    generate code for c's on_request() function header
43    for each entry of c.actions <parent_name, s_next>, do
44     write "if (sender==\""+parent_name+"\") {" in code
45     if s_next has more than 1 name // introduce ctrl var
46      i = 0
47      for each name in s_next
48       write "if ("+name+"_"+parent_name+"_Ctrl=="+i+")" in code
49       write "  "+name+".on_request(\""+c.comp_name+"\")"
50       i = i+1
51     else if s_next has 1 name
52       name = s_next[0]
53       write name+".on_request(\""+c.comp_name+"\")"
54     write "}"
55    write "}" // end of the on_request() function
```

Figure 3: The code generation algorithm

A. Enhancing Generated Code

We study the example service's dependency on its components, particularly how many request processing paths are affected if a component fails. The symbolic execution tool we use in this study is Klee [13]. Symbolic execution cannot be directly applied to the generated code like that in Figure 2. We add the following enhancements to the generated code.

There are other symbolic execution tools like Symbolic PathFinder [32], Angr [33], QSYM [34], and SymCC [35]. Compared with them, Klee (a) supports two modes: symbolic execution and concrete execution, (b) processes C-language-like program syntax, and (c) is a pretty mature and stable tool. As our transformation algorithm generates C-language-like programs, and the support of both modes potentially provides flexibility to support various goal properties in future studies, we selected Klee.

Definition of symbolic variables. Symbolic variable is a type of variable provided in the Klee tool such that, during symbolic execution Klee explores values of the variable as much as possible within its value range, and hence, exhaustively explores possible paths of the program.

A member variable *status* is added to every component's class to indicate all components' availability states, and similarly, another *status* variable is added to the host class (Machine) for studying the host dependency. When the *status* is 0, the component is not functioning properly.

We enhance the generated code to designate all components' *status* variables and the *Middleware_ActionCtrl* variable of the middleware component as symbolic variables by using *klee_make_symbolic()*. Basically, all control variables and state variables (like *status*) should be designated as symbolic variables.

Dependency study and target failure model. To study the dependencies of service availability on components' *status*, we add *klee_assert()* statements at appropriate places of the generated code. When the *status* of a component instance is assigned as 0 by Klee, the assertion failure stops this execution path and marks the path as a failure. For the dependency strength analysis in the paper, crashes are the target failure model because dependency strength shows to what extent one component crash causes a service failure.

B. *Symbolic Execution*

The example service has 3 components with their interactions during the service as illustrated in Figure 1. We implemented the example service and deployed it on a single machine. In the implementation the middleware has a 15% probability of querying the backend and an 85% probability of responding to the frontend without querying the backend. The client sent 20 requests to the frontend during the experiment.

We implemented the algorithm that transforms the service into program code and enhanced the code for the service analysis. Then we compiled the enhanced program code (in C++), built LLVM bitcode, and then used Klee to symbolically execute the bitcode. There are three components' *status* variables, one *Machine*'s *status* variable, and the *Middleware_ActionCtrl* variable. It is a total of 5 symbolic variables, each of which may have the value 0 or 1. The symbolic execution explores 2^5=32 execution paths of the program in a single pass.

The notions of strong/weak dependency are defined with regard to the probability of a component or host failure causing a service failure in all requests' execution paths. When the probability is larger than a given threshold, the service is considered as bearing a strong dependency on the component; otherwise, the service has a weak dependency on the component.

Dependency analysis. We run the symbolic execution of the enhanced code to explore all possible request execution paths of the example service. In this experiment, Klee executed a total of 28917 LLVM bitcode instructions and explored all of the 32 execution paths. The truth table of these paths is given in Table 2. 3 paths completed (Result=1 in the table), and the remaining 29 paths terminated incompletely due to assertion failures.

Table 2: Symbolic execution results for the example service

	Machine	Frontend	Middleware	Backend	Ctrl Var	Result
path 1	1	1	1	1	1	1
path 2	1	1	1	1	0	1
path 3	1	1	1	0	1	0
path 4	1	1	1	0	0	1
path 5~8	1	1	0	-	-	0
path 9~16	1	0	-	-	-	0
path 17~24	0	1	-	-	-	0
path 25~32	0	0	-	-	-	0

Here we study the service dependency on the backend component. The number of distinct paths where Backend failure causes service failure (Backend=0, Result=0 and Machine, Frontend and Middleware being 1) is 1 (path 3). As 3 of the 20 requests go along path 3, the probability of the backend failure causing the service failure is 15%. The probability of the *Machine* or any other component failure causing the service failure is 100%. If the given threshold is 20%, the service has a weak dependency on the backend and a strong dependency on the machine and any of the other components.

V. Discussions and Research Plan

Scope of Service Computing in Consideration. Our main idea motivated by the observed analogy between services and programs is to do service analysis like program analysis after the transformation. So the broadest scope includes all types of services comprised of multiple components with communications among them. But the initial study in this paper focuses on dependency analysis, and here we mainly consider micro-services.

Service Properties to be Studied. Besides service dependencies, we are considering some other behaviors/properties to study using this approach, including cross-component error propagation and protocol verification. We find the observed analogy is still valid for these properties, though component source code should be considered besides component interactions.

Why Choose Symbolic Execution? Other program analysis technologies like model checking may also be applied to the generated programs for analyzing services. We select symbolic execution in this initial study because it has two modes: symbolic execution and concrete execution. So, it potentially provides us flexibility to support various goal properties in future studies.

Scalability Issue. This paper focuses on transforming service behavior into a program to enable use of symbolic execution for extensive exploration of service behaviors. We believe the scalability issue in using symbolic execution for service behavior analysis is similar to that in using it for program analysis. As certain technologies [26-31] alleviate the issue in program analysis, similar technologies can be applied (maybe with some adjustments) to alleviate the issue in service behavior analysis.

Research Plan. We have done a preliminary case study, and are taking the following plan to conduct the research work:

i) Besides studying the strength of service dependencies, we plan to identify a few more service problems or properties to study. Candidates we are considering include cross-component error propagation, verification of protocol implementations, and troubleshooting of services.

ii) We will study these service problems or properties in large-scale service platforms like OpenStack and Kubernetes.

iii) When we employ symbolic execution to study service behavior in such platforms, there may be scalability issue like state explosion. We will leverage state-of-the-art technologies of state explosion mitigation [26-31] (with required adjustments) for addressing it.

VI. Conclusion

In this paper we explored the idea of using symbolic execution for studying behavior of distributed services. Basically, we observed that there is an analogy between distributed service behavior and object-oriented program execution. Then we propose an algorithm that transforms distributed service behavior into program code representation, and applies symbolic execution onto the program code for investigating the service behavior. We carried out a case study as preliminary exploration of applying symbolic execution for service analysis. We are following a research plan to advance the proposed approach to a practical stage.

References

[1] W. Ahmed, et al.. Formalization of Reliability Block Diagrams in Higher-order Logic. Journal of Applied Logic. 2016.

[2] C. Singh and J. Mitra, Monte Carlo simulation for reliability analysis of emergency and standby power systems, IEEE Industry Applications Conference, 1995.

[3] E. Dąbrowska, Monte Carlo Simulation Approach to Reliability Analysis of Complex Systems. Journal of KONBiN. 2020.

[4] D. Lee, et al., A Nonparametric Bayesian Network Approach to Assessing System Reliability at Early Design Stages. Reliability Engineering & System Safety. 2017.

[5] S. Luo, et al., Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. ACM Symposium on Cloud Computing, 2021.

[6] S.-P. Ma, et al., Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Generation Computer Systems, 2019.

[7] M. J C, et al., Experiences with modeling network topologies at multiple levels of abstraction, USENIX Symposium on Networked Systems Design and Implementation. 2020.

[8] Y.-W. E. Sung, et al., Robotron: Top-down Network Management at Facebook Scale. ACM SIGCOMM, 2016.

[9] H. H. Liu, et al., Automatic Life Cycle Management of Network Configurations. Afternoon Workshop on Self-Driving Networks, 2018.

[10] Unified Modeling Language, http://uml.org/

[11] SysML Open Source Project: What is SysML? Who created SysML?, https://sysml.org/

[12] F. Machida, et al., Candy: Component-based Availability Modeling Framework for Cloud Service Management Using SysML, SRDS 2011.

[13] KLEE Symbolic Execution Engine, https://klee.github.io/

[14] C. S. Păsăreanu, et al., Symbolic PathFinder: symbolic execution of Java bytecode. IEEE/ACM International Conference on Automated Software Engineering, 2010.

[15] V. Chipounov, et al., S2E: a platform for in-vivo multi-path analysis of software systems. SIGPLAN No. 46, 2011.

[16] I. Yun, et al., QSYM: a practical concolic execution engine tailored for hybrid fuzzing. USENIX Security. 2018.

[17] S. Poeplau, et al., Symbolic execution with SymCC: Don't interpret, compile! USENIX Security Symposium, 2020.

[18] S. Poeplau, et al., SymQEMU: Compilation-based symbolic execution for binaries, 2021.

[19] J. Simsa, et al., DBug: systematic evaluation of distributed systems, International conference on Systems software verification, 2010.

[20] H. Guo, et al., Practical software model checking via dynamic interface reduction, ACM Symposium on Operating Systems Principles, 2011.

[21] B. Tak, et al., vPath: Precise Discovery of Request Processing Paths from Black-box Observations of Thread and Network Activi- ties, USENIX Annual technical conference. 2009.

[22] Y. Yang, et al., Capturing Request Execution Path for Understanding Service Behavior and Detecting Anomalies without Code Instrumentation, IEEE Transactions on Services Computing, 2023.

[23] J. Mace, R. Roelke, R. Fonseca, Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems, Proceedings of the 25th Symposium on Operating Systems Principles. ACM, 2015: 378-393.

[24] HTrace, http://htrace.incubator.apache.org/

[25] J. Kaldor, et. al, Canopy: an End-to-end Performance Tracing and Analysis System, ACM Symposium on Operating Systems Principles (SOSP), 2017.

[26] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, Efficient state merging in symbolic execution, ACM SIGPLAN Notices, Volume 47, Issue 6, 2012.

[27] F. Nejati, et. al, Handling State Space Explosion in Component-Based Software Verification: A Review, in IEEE Access, vol. 9, 2021.

[28] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, Martin Vechev, Learning to Explore Paths for Symbolic Execution, Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS), 2021.

[29] Saparya Krishnamoorthy, Michael S. Hsiao, Loganathan Lingappan, Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs, IEEE Asian Test Symposium, 2010.

[30] Kaki Ryan, Cynthia Sturton, Sylvia: Countering the Path Explosion Problem in the Symbolic Execution of Hardware Designs, Formal Methods in Computer-Aided Design (FMCAD), 2023.

[31] Shan Zhou, Jinbo Wang, Panpan Xue, Xiangyang Wang, Lu Kong, An Approach to the State Explosion Problem: SOPC Case Study, Special Issue of Advanced Communication and Networking Techniques for Artificial Intelligence of Things (AIoT), 2023.

[32] Păsăreanu, C. S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., & Rungta, N. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering, 20, 391-425, 2013.

[33] Angr documentation, https://docs.angr.io/en/latest/ quickstart.html

[34] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, Taesoo Kim, QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing, 27th USENIX Security Symposium, 2018.

[35] SymCC: efficient compiler-based symbolic execution, https://github.com/eurecom-s3/symcc