Automating Testing of Service-oriented Mobile Applications with Distributed Knowledge and Reasoning

James Edmondson, Aniruddha Gokhale, Sandeep Neema
Dept of EECS, Vanderbilt University
Nashville, TN 37212, USA
{*james.r.edmondson,a.gokhale,sandeep.neema*}*@vanderbilt.edu*

***Abstract*—Automated testing of distributed, service-oriented applications, particularly mobile applications, is a hard problem due to challenges testers often must deal with, such as (1) heterogeneous platforms, (2) difficulty in introducing additional resources or backups of resources that fail during testing, and (3) lack of fine-grained control over test sequencing. Depending on the testing infrastructure model, the testers may also be required to fully define all the hosts involved in testing, be forced to loosely define test execution, or may not be able to dynamically respond to application failures or changes in hosts available for testing. To address these challenges, this paper describes an approach that combines portable operating system libraries with knowledge and reasoning, which together leverage the best features of centralized and decentralized testing infrastructures to support both heterogeneous systems and distributed control. A domain-specific modeling language is provided that simplifies, visualizes, and aggregates test settings to aid developers in constructing relevant, feature-rich tests. The models of the tests are subsequently mapped onto a portable testing framework, which uses a distributed knowledge and reasoning engine to process and disseminate testing events, successes, and failures. We validate the solution with automated testing scenarios involving service-oriented smartphone-based applications.**

***Keywords*-test sequencing; distributed control in testing; portability; knowledge dissemination; domain specific modeling language;**

I. INTRODUCTION

Automated testing of distributed, service-oriented applications generally falls into two execution models: centralized and decentralized. In a centralized model, a testing control service sends testing commands to services installed on each host taking part in testing, generally in a push model. In a decentralized model, there is no centralized control service, and testing occurs in a purely distributed manner. If synchronization is required, decentralized infrastructures may use network lock files, or they may require separate daemons with intimate knowledge of test sequencing or custom messaging protocols.

To understand the two models, let us consider an example where a developer would like to test a service-oriented smartphone-based mobile application which sends a message to an application server. Ideally, the smartphone should not send a message to the server until the server is ready, and suppose this should happen after 15 seconds.

In the centralized model shown in Figure 1, a control server sends a command to launch the application server, waits for 15 seconds, and then sends a command to the smartphone or host connected to the smartphone, which runs a script or launches a unit test already on the smartphone and sends a message to the application server. If no such script or unit test exists, a tester might emulate a message that looks like it came from a smartphone.

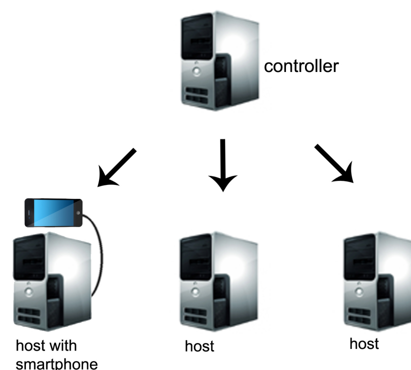

Figure 1. Centralized software testing

For simple tests like this example, the centralized model works, particularly when services or applications need to only do one task and then exit. Additionally, the centralized model can allow for fine-grained launch sequencing based on time and provides a single point of configuration when changing tests.

In a decentralized model, portable scripts (*e.g.*, in Perl, Python, or Ruby) are written that perform a series of steps appropriate for the host running each script without using a centralized controller. If coordination is needed, the scripts may use network lock files. In the context of our example, the first script might launch an application server, wait for 15 seconds, and then create a lock file called “server.ready” on a network file system. A second host runs a different script that keeps checking the network file system for a file called “server.ready”. When the file is created, the second script may launch another script or service to automate a smartphone sending a message.

The decentralized model is portable, elegant and intuitive and avoids any type of centralized messaging bottleneck or single point-of-failure. It also allows for moving scripts between hosts and not binding the testing infrastructure to a single host or port configuration, which may change as hosts go in and out of service. For clouds, LANs, and grids, decentralized testing infrastructures are highly appropriate, intuitive, and may be the only real available solution when host/port information are unknown before an experiment is swapped in (*e.g.*, in a cloud), without the cloud API exposing the information directly.

One of the major issues with the decentralized model is the usage of network file systems as lock files for testing coordination. Network file systems, however, were not created with lock files in mind. Our extensive experience conducting decentralized testing over local area networks reveals that network file systems, such as the widely-used Network File System (NFS), make no guarantees that local caches would ever be updated with the lock files. Moreover, in practice, timing delays could be thrown off by several minutes, even if the lock file appears at all. A custom communication model built on top of TCP or UDP could potentially solve this issue, but we do not observe this approach being used in traditional testing solutions.

While centralized models can avoid network file systems, they are tightly coupled to hosts, present a single point-of-failure (the testing controller), and the configuration of the controller can become cumbersome as tests become more complex, especially when services need to be launched in parallel with different timing delays. Additionally, most centralized models are not portable to all operating systems and instead cater to a specific architecture (*e.g.*, Windows).

In addition to these sequencing and portability concerns, the testers need to be able to codify and debug their test concerns with an intuitive interface. A cumbersome, unusable infrastructure is unlikely to benefit developers. From all these concerns, we form the following requirements of a distributed, automated testing infrastructure.

1) It should support heterogeneous OS platforms, networking platforms and topologies
2) It should allow for dynamic sequencing of tests based on testing events, successes, and failures
3) It should be host-agnostic and allow for moving test setups easily to other nodes (*e.g.*, in a cloud)
4) It should allow for seamless recovery from node failure, and facilitate the creation and integration of backup testing entities
5) It should be easily configurable

Our solution to address these requirements brings together the advantages of centralized and decentralized approaches without suffering from their individual drawbacks. In particular, it automates the testing of distributed service-oriented applications, illustrating the approach on case studies drawn from smartphone-based mobile applications. Our solution is realized as an open-source tool kit called KATS (KaRL Automated Testing Suite). KATS is based on an underlying framework called the Knowledge and Reasoning Language (KaRL) engine [5] available from madara.googlecode.com.

The rest of this paper is organized as follows: Section II details prior work relating to automated testing infrastructures; Section III presents an outline of our solution approach to distributed, automated testing using a DSML and distributed knowledge and reasoning; Section IV discusses examples of case studies that mimic real-world, distributed black box testing that KATS is being developed for; Section V outlines some performance related experiments; and finally Section VI presents concluding remarks.

II. Related Work

A vast majority of networked testing infrastructures [21], [3], [4], [10] utilize a centralized controller. This centralized paradigm extends to most deployment [15], [14] and testing instrumentation technologies [14], [23], [24], [8] as well.

Though these related tool suites do not support truly distributed test scheduling and automation, they often do include useful instrumentation tools [23] that allow for real time performance analysis – which we do not include in our tool suite at the moment. Other instrumentation systems place themselves in between applications and the underlying middleware or networking layer [22]. Our solution relies on distributed blackbox testing without any requirement for hooks into the operating system or services to be tested.

Model-driven testing systems exist for centralized or non-distributed systems [7], [20], [6], [11], [1]. Other systems use a generic programming approach [12] for functional testing. Both these types of systems demand intimate knowledge of internal behaviors of the application or service being tested – often requiring code insertion into the application or modeling/reading of source code.

Recent work using domain-specific modeling in testing with mobile phones appears in the literature [18], but there are many key differences between the two approaches. We do not require a hardware modification to directly send key events to the phone and instead rely on the Android Debug Bridge which is part of the Android SDK. Additionally, we offer automated and barriered test cases in contrast to interactive sessions between users and the phone in [18].

We are unaware of prior work using distributed knowledge and reasoning in automating networked testing, nor have we seen prior art using an anonymous publish/subscribe (pub/sub) paradigm to coordinate and sequence test execution among generic networked processes or programs. Scripted distributed testing using pub/sub exists [25] but it requires a networked file system and customized scripting to sequence tests and suffers from all of the NFS problems mentioned in decentralized testing in Section I.

III. Automated Testing Framework for Service-oriented Distributed Applications

This section describes our solution approach to realizing automated testing of distributed, service-oriented applications focusing on mobile services and applications. The contributions of this solution are as follows:

1) A portable process and process group specification that addresses Requirement 1 by utilizing the Adaptive Communication Environment (ACE) [19], which has been ported to most platforms including Linux, Windows, Mac, Android, and iOS (see Section III-A).
2) A portable knowledge and reasoning engine called KaRL [5] that processes testing events, successes and failures (Requirements 2, 4) and distributes knowledge across the testing network via an OpenSplice DDS transport [17]. DDS is an anonymous publish/subscribe protocol that is network architecture portable (Requirement 1) and helps us remain host-agnostic (Requirement 3) (see Section III-C).
3) A middleware solution called the KaRL Automated Testing Suite (KATS) which integrates and configures KaRL and standardizes the reasoning operations into a process lifecycle that facilitates fine-grained sequencing (Requirement 2), and conditions for starting or using backup services (Requirement 4) (see Section III-B).
4) Our solution provides a domain-specific modeling language (DSML) [16] for visualizing test sequences and parameters in an intuitive way (Requirement 5). From this DSML, we generate test configurations for the decentralized agents to follow (see Section III-D).

The resulting architecture is shown in Figure 2.

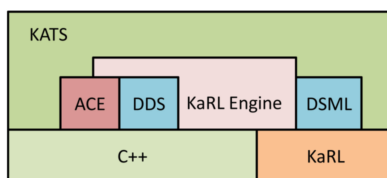

Figure 2. Overview of Solution Architecture

A. Processes and Process Groups

Processes in the KATS DSML are organized into process groups which dictate how individual processes are configured and launched. Each process has over a dozen configuration settings to affect changes in OS scheduling, file I/O, test sequencing, and logging, and we accomplish OS portability by using the Adaptive Communication Environment (ACE) middleware, which supports a wide range of operating systems–including mobile phone platforms like iOS and Android. Among the more useful settings for services supported in the current version of the KATS DSML are the following:

- Real-time scheduling at highest OS priority, which ensures minimum jitter, fewer context switches, and higher priority for tested applications.
- Executable name, working directory, and command-line arguments for fine-grained control of applications that are launched and what configurations and parameters they will be started with.
- Redirecting stdin, stderr and stdout to files for emulating user input and flexible file logging. Any redirection from or to a file is buffered by the OS, which minimizes jitter and reduces the time spent in OS calls - which are expensive.
- Kill times and signals (POSIX) for errant applications. Even on non-Unix operating systems (*e.g.*, Windows), we allow for termination of applications after a specified time delay if the application does not return first. This feature is configurable by the tester.
- Test sequencing information (*e.g.*, preconditions and postconditions).

These settings allow for flexible management of each application or service launch within a process group. Process groups have additional settings indicating whether to launch the processes in the group in parallel or in sequence. Each process group also includes the settings listed above except for executable name and command-line arguments. This expressiveness for process and process group settings is part of our solution to address Requirement 1 (test settings portable across heterogenous OS platforms).

B. Configuring Test Sequencing

Processes in the KATS infrastructure can be sequenced in two complementary ways: temporal delays and conditions. Temporal delays occur after KATS conditions have been checked. These conditions come in three supported types: barriers, preconditions, and postconditions.

The sequence of these temporal delays and conditions is shown in Figure 3. Barriers are groupings of processes or process groups that must rendezvous to the same point before moving onward with process launching. Barriers require setting a process id and the number of processes expected to participate in the barrier event.

After the barrier, preconditions are checked against the local context within that represent conditions that must be true before the process may be launched. The application is then launched and its return code is saved into the KaRL variable *.kats.return*, which allows for postconditions to feature logics that check whether or not the application succeeded or failed based on the return code (the exit value returned by the process). Postconditions may also make global state modifications, which may satisfy other application or service preconditions. Temporal delays, barriers, preconditions, and postconditions can be specified on either

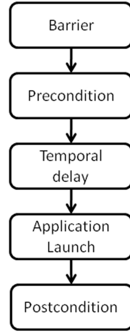

Figure 3. KATS Process Lifecycle

a process or process group level. These lifecycle elements are the foundation of our solution to meet Requirement 2 (dynamic sequencing of tests based on test progress or failure).

C. Augmenting Decentralized Testing with Knowledge and Reasoning

The process lifecycle outlined in Section III-B provides the infrastructure for meeting Requirement 2, but we still need a communication and reasoning mechanism for distributing knowledge and learning from testing events, successes, and failures. In this section, we look at the distributed knowledge and reasoning engine we developed called the Knowledge and Reasoning Language (KaRL) engine [5].

KaRL allows developers to manipulate knowledge via expressions in a programming logic which are evaluated against a local context (see Figure 4). The KaRL reasoning engine mechanisms used for evaluating logics allow for accessing or mutating knowledge variables and provide fine-grained debugging capabilities for global knowledge state in an atomic, thread-safe manner.

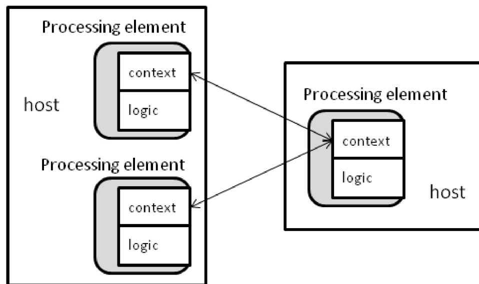

Figure 4. KaRL evaluates user logics against a local context, which is then synchronized with other contexts

The underlying reasoning engine mechanisms create knowledge events that are disseminated based on whether they are considered local (*i.e.*, the variable begins with a period) or global (*i.e.*, anything else) and whether or not there are interested software entities in the network that would like information on the published knowledge domain. Once the knowledge events in KaRL are ready to be disseminated, they can be sent to the appropriate entities using an underlying transport mechanism.

The coupling of KaRL with an anonymous publish/subscribe paradigm fits our knowledge and reasoning needs for testing in the following ways. First, the communication is host-agnostic–which means we do not have to do any configuration when changing hosts or enlarging a service cloud (Requirement 3). Second, the KaRL reasoning service is robust to application or node failures and allows us to seamlessly integrate backup services to take over in testing operations, if necessary (Requirement 4). Third, we wanted an engine that could change preconditions, postconditions, and barriers on the fly, at runtime, and most other knowledge and reasoning engines do not allow this without resetting all knowledge periodically.

The KaRL engine and its underlying transports are seamlessly incorporated into the KATS testing framework with very little interaction or configuration required by the tester. The only time a tester manually works with KaRL is when they are creating preconditions and postconditions. Though barriers use KaRL, the user only has to specify a barrier name, an id for this process, and the number of processes participating in the barrier. KATS builds the KaRL logic from this information provided by the user in the visual DSML of their testing procedures.

We now briefly outline the KaRL solution to encoding testing information into conditions for the expressed purpose of distributed reasoning. KaRL is a knowledge representation and predicate calculus engine built for real-time, continuous systems. KaRL features first class support for multi-modal knowledge values, and each knowledge variable may be one of 2^{64} possible values. KATS conditions (*i.e.*, barriers, preconditions, and postconditions) are KaRL expressions that take the following form $Predicate => Result$, which can be read as "if the Predicate is true then evaluate Result." By default, $Predicate$ is true in the KaRL language, and KaRL expressions may be chained together with the *Both* operator ;. This operator ; simply means that both the left and right expression should be evaluated.

For an example of a KaRL expression that can be used in a KATS condition, consider the following possible postcondition: $server.finished => (no.backup => all.tests.stop = 1)$. This KaRL expression will check the variable $server.finished$ and if it is not zero (*i.e.*, true), then we will check if there is a backup server available. If $no.backup$ has been set to anything but zero, we then indicate that all tests need to stop by setting $all.tests.stop$ to 1.

KaRL supports most multi-modal logic operations and conditionals including $==$, !=, &&, and $||$ and mutations on knowledge, including $+$, $-$, $*$, $/$, $\%$, where the latter three are multiplication, division, and remainder or modulus.

For a more comprehensive listing, please see the project site. See Section IV for specific, simple examples of how KaRL expressions offer powerful sequencing capabilities to a tester.

D. Domain-Specific Modeling

KATS provides a modeling front-end (DSML) to overcome mundane and erroneous tasks handled by the testers (Requirement 5). The KATS DSML comprises a UML-based meta-model developed using the open-source and freely available Generic Modeling Environment (GME) [13] to codify the process lifecycle and all reasoning operations in a highly configurable way. This DSML has four key components that users can create and configure: processes, process groups, hosts, and barriers. The DSML in turn generates an XML file that is interpreted by KATS tools. If testers do not wish to use the visual DSML, they can encode simple XML files according to the KATS schema. We show example testing XML files in Section IV.[1]

IV. Demonstrating KATS on Case Studies

In this section we present experimental scenarios that highlight typical distributed testing configurations. These configurations feature service-oriented applications using smartphones connected to hosts because these scenarios represent the types of experiments we are currently working on for our sponsored R&D projects. To concretize our case studies, we assume that the smartphones used in each of these examples are Android phones that allow for instrumentation of application launches through the Android Debug Bridge [2].

A. Sequenced Smartphone Scenario

In our first test case, five smartphones are connected to a server and testers are interested in determining if the server will receive the five smartphone requests in the correct order with no arbitrary temporal delays between them. Specifically, the testers want to start the gateway service, wait for 15 seconds, and then send a message from Phone1, then Phone2, Phone3, etc. We assume that all smartphones are connected by USB to the same host and the server resides on its own host.

In KATS, a model is created that creates a barrier named `SimplePhoneSequence`, which has six processes and IDs from 0..5 inclusive and arbitrarily assigned. The five phone processes can be setup in one of two ways and both require an initial step of launching the server with no preconditions or postconditions. Additionally, we can sequence the phones in two different ways: (1) no pre- or post-conditions and (2) with pre- and post-conditions.

The first option requires that KATS process group launch all the phone instrumentation scripts in parallel (a single element added to the process group XML configuration file). This configuration is by far the easiest to setup, and requires only setting command line arguments for the phone scripts (for the user-defined application logic) and possibly redirecting stderr and stdout to files for logging.

```
<group>
    <setup>
        <parallel />
        <domain>case_study_1</domain>
    </setup>
    <process>
        <domain>case_study_1</domain>
        <barrier name="barrier" />
        <id>0</id>
        <processes>6</processes>
        <executable>ServerApp</executable>
        <commandline>--port 55555</commandline>
    </process>
    <process>
        <domain>case_study_1</domain>
        <barrier name="barrier" />
        <id>1</id>
        <processes>6</processes>
        <delay>15</delay>
        <postcondition>phone1.done=1</postcondition>
        <executable>PhoneMessage</executable>
        <commandline>--server 55555 --phoneid 1</commandline>
    </process>
    <process>
        <domain>case_study_1</domain>
        <barrier name="barrier" />
        <id>2</id>
        <processes>6</processes>
        <precondition>phone1.done</precondition>
        <postcondition>phone2.done=1</postcondition>
        <executable>PhoneMessage</executable>
        <commandline>--server 55555 --phoneid 2</commandline>
    </process>
... phones 3-4 are omitted for brevity ...
    <process>
        <domain>case_study_1</domain>
        <barrier name="barrier" />
        <id>5</id>
        <processes>6</processes>
        <precondition>phone4.done</precondition>
        <executable>PhoneMessage</executable>
        <commandline>--server 55555 --phoneid 5</commandline>
    </process>
</group>
```

Figure 5. Generated XML from DSML for the Sequenced Smartphone Scenario

The second configuration possibility requires that Phones 1..5 have preconditions and/or postconditions set according to settings shown in Table I. Phone2 depends on Phone1 completing, Phone3 depends on Phone2, and so on. With this intuitive mapping of preconditions and postconditions, we have accomplished distributed testing execution with no operating system sleep calls and no centralized controller. The XML generated from the visual KATS DSML is shown in Figure 5.

Table I
Preconditions, Postconditions, and Temporal Delay in the Sequenced Smartphone Scenario

Name	Pre	Post	Delay
Phone1		Phone1.done=1	15
Phone2	Phone1.done	Phone2.done=1	
Phone3	Phone2.done	Phone3.done=1	
Phone4	Phone3.done	Phone4.done=1	
Phone5	Phone4.done		

[1] We do not provide details of the modeling environment due to space limitations, and since it is optional in KATS.

B. Backup Service for Smartphones Scenario

The testers have acquired a backup server host, and they want to create tests that have phones switch over to the backup server if the main server starts dropping connections. To complicate the scenario, testers bring in three more machines and attach the five phones to random machines. The three phones are blasting clients that send 100 messages a second to the gateway service. If any of these scripts fail to send their message, they exit with a return code of 1. Otherwise, they return a 0 for success.

The two other phones start only if the blasting clients fail with a return code of 1. No phone should try to contact a server until 15 seconds after the gateway service is ready, and all phones should be killed after 3 minutes regardless of whether they succeed in contacting the main server or not. We assume the backup gateway service is launched along with or sometime after the main gateway service.

With KATS, the testers can produce this more complicated sequencing scenario with minimal configuration of preconditions and postconditions shown in Table II.[2] ".kats.return" is the return value of the launched process (in this case one of the blasting clients). The postconditions here indicate that "if my return value is non-zero, set the global variable *backup* to 1." The other two phones have preconditions that say "if the global variable *backup* is non-zero, launch my process." In this way, we facilitate the testing scenario created by the testers, and we can move these process groups wherever we need to in the network or cloud.

Table II
SETTINGS FOR PHONES IN THE BACKUP SCENARIO

Name	Pre	Post	Kill
Phone1		.kats.return => backup=1	180
Phone2		.kats.return => backup=1	180
Phone3		.kats.return => backup=1	180
Phone4	backup		180
Phone5	backup		180

To implement this same solution with a centralized model would require a custom controller with push and pull capability and may require multiple versions of testing services customized to a particular operating system. In KATS, we get this cross-platform functionality included.

V. EVALUATING KATS

In this section we briefly describe performance information regarding our testing infrastructure KATS. Though very few testing infrastructures are concerned with latency and performance, decentralized service-oriented software can often have a high overhead especially in group communications like barriers. Because our solution is novel in how it disseminates data to a testing infrastructure, it is important to investigate whether KATS features come at a high performance cost.

We divide these performance metrics into two sections: condition latency (the time it takes for a precondition or postcondition to reach other interested hosts on a local area network) and barrier latency (the time it takes for a group of distributed applications to barrier together across a network).

A. Experimental Testbed Setup

Unless specified otherwise, all experiments were conducted on five IBM blades with dual core Intel Xeon processors at 2.8 GHZ each and 1 GB of RAM running Fedora Core 10 Linux. Each blade is connected across gigabit ethernet and networked together with a switch. The code was compiled with g++ with level 3 optimization, and each test featured a real-time class to elevate OS scheduling priority to minimize jitter during the test runs. This testbed is similar to the one we are using for sponsored research.

B. Measuring Condition Latency

Condition latency is the time it takes for a precondition or postcondition of one application to reach another application across the network. Recall that in testing, a precondition is a condition that must be true before an application can launch. Consequently, this latency between the setting of a variable in one part of the network and the evaluation of this data in interested testing entities is an important metric to have.

Setup. Two applications (App 1 and App 2) were launched on two separate hosts. Each application was configured with conditions indicated in Table III, and then the applications were launched repeatedly. These conditions create a roundtrip latency time on each host. We take this roundtrip latency time and divide by two to get our averaged latency numbers.

Table III
CONDITIONS USED FOR LATENCY TESTS

App 1	App 2
P0 == P1 => ++P0	P0 != P1 => P1 = P0

Results. The results of our condition latency tests are shown in Table IV. These results represent average latency for a condition to be sent and processed from one host to the next. Each column represents the number of round trips completed. The *Ping* row is the latency reported between the nodes via the Linux ping utility. *Dissem* is the average dissemination latency of KATS conditions.

Analysis. The KaRL interpreter and OpenSplice DDS transport provide very low latency for the delivery of KATS conditions. On our network, this allows for sub-millisecond accuracy when a tested application's dependencies are met. This will vary depending on network latency in the developer

[2] Generated XML is not shown due to space limitations.

Table IV
CONDITION RESULTS

	5k	25k	50k	100k	500k
Ping	114 us	114 us	114 us	114 us	114 us
Dissem	650 us	440 us	437 us	315 us	317 us

testbed. The lower the latency, the more fine-grained the test sequencing can be, and the more expressive testers can be with their distributed testing scenarios.

C. Measuring Barrier Latency

Barrier latency is the time it takes for a group of KATS processes to execute a barrier operation before launching the user-defined applications. Barrier operations are far more expensive than setting a condition, but barriers are useful when testers need all testing participants to be up and ready. From our experience, all distributed testing scenarios require a barrier, unless the individual tests do not interact with other testing participants. Consequently, the barrier time is an important metric because it is the overhead that must be performed before each distributed test can start.

Setup. Five hosts with two CPUs apiece launch applications (simple sleep statements of five seconds) after reaching barriers under different networked process stress loads. High resolution timers are used to measure the time it takes to complete a barrier. The tests are repeated 10 times and an average time is reported.

Each host in these tests launched its own barrier set of 2 to 5 applications. We only have two CPUs per host, so as the number of applications increases, no further speed increase is possible (it is pure overhead from context switching past 2 application launches). We will discuss why we did not connect the hosts in a LAN-wide barrier in the Analysis section. Each barrier grouping was executed 10 times per host and averaged. We complement the average latency numbers with minimum and maximum observed barrier time.

Results. The results of our barrier latency tests are shown in Table V. These results represent latencies experienced on a single host launching applications >= the number of CPUs on the host. The minimum latency was the smallest time spent in a barrier in the 50 tests. The maximum was the largest, and the average was also for the 50 tests.

Table V
BARRIER LATENCY RESULTS

Apps per host	Min	Avg	Max
2	0.8 ms	2.0 ms	2.7 ms
3	1.7 ms	9.4 ms	21.5 ms
4	2.4 ms	17.2 ms	17.2 ms
5	3.0 ms	17.3 ms	35.2 ms

Analysis. We use intrahost communication here to highlight the speed and precision that the KATS system is able to achieve with distributed processes. Within a single host, as is the case here, the cause of deviation between the min and max is caused by context switching. Since our experimental machine had two cores, the deviation is very low on the two apps test. Running this same series on a real-time kernel can reduce the deviation, but in a networked test, the benefits will be negligible.

We have found that barrier latencies for networked processes across many hosts are hugely dependent on when the distributed tests are started on each host, far more than anything dictated by the latency of the underlying KaRL system in disseminating knowledge shown in Section V-B.

As an example, consider the case where KATS tests will be started distributedly via cron jobs on each host at 5:00 p.m., according to the system clock which is synchronized with the network time protocol (NTP). Even under the best of circumstances, this protocol guarantees accuracy between the networked clocks in the hundreds of milliseconds – orders of magnitude larger than how long barriers, preconditions, or postconditions execute in KATS or its KaRL infrastructure outside of situations with heavy context switching between the KATS processes and the OpenSplice DDS daemon.

This may seem like a problem, but this is exactly the reason barriers are recommended among test participants. Barriers ensure that regardless of time synchronization or order of test launches in the network, the exact sequence of testing events will be launched precisely as specified by the testers.

VI. CONCLUSIONS

In this paper we have described a novel, portable approach to decentralized testing automation that accomplishes distributed test sequencing and control for service-oriented applications. Our solution provides testers with distributed barriers for processes involved in a test sequence, preconditions for process entry that can be based on knowledge, and postconditions that can export knowledge to facilitate test sequencing over the network using an anonymous publish/subscribe paradigm. Testers generate XML-based test scenarios that can be used by our solution to distributedly automate and sequence thousands of tests in the appropriate order.

We have validated our work with case studies involving smartphones, client applications, and custom application services. Additionally, we show how to use the built-in KATS features to set knowledge based on the success or failure of individual test processes based on their return values.

Future work for our system includes more tools for forensic accounting, performance metrics reporting for tested applications, knowledge observers for visualizing testing

progress, and more intuitive UI design that makes conditions easier to create, understand, and edit. All source code for this project, including examples and case studies can be found at our project site at http://madara.googlecode.com and is provided under a BSD license.

References

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *IN PROC. INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS (ISSTA*, pages 123–133. ACM Press, 2002.

[2] G. Chang, C. Tan, G. Li, and C. Zhu. Developing mobile applications on the android platform. In X. Jiang, M. Ma, and C. Chen, editors, *Mobile Multimedia Processing*, volume 5960 of *Lecture Notes in Computer Science*, pages 264–286. Springer Berlin / Heidelberg, 2010.

[3] J. C. Cunha, J. Loureno, T. R. Anto, J. A. Lourenco, and T. R. A. Ao. An experiment in tool integration: the ddbg parallel and distributed debugger. In *EUROMICRO Journal of Systems Architecture, nd Special Issue on Tools and Environments for Parallel Processing*, pages 708–717. Springer, 1999.

[4] C. Dumitrescu, I. Raicu, M. Ripeanu, and I. Foster. Diperf: An automated distributed performance testing framework. In *in 5th International Workshop in Grid Computing*, pages 289–296. IEEE Computer Society, 2004.

[5] J. Edmondson and A. Gokhale. Design of a scalable reasoning engine for distributed, real-time and embedded systems. In *Proceedings of the 5th conference on Knowledge Science, Engineering and Management, KSEM 2011*, Lecture Notes in Artificial Intelligence (LNAI). Springer, 2011.

[6] F. Fraikin and T. Leonhardt. Seditec " testing based on sequence diagrams. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, pages 261–, Washington, DC, USA, 2002. IEEE Computer Society.

[7] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

[8] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: on-line monitoring and steering of large-scale parallel programs. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers '95., Fifth Symposium on the*, pages 422 –429, feb 1995.

[9] A. Hartman, M. Katara, and S. Olvovsky. Choosing a test modeling language: a survey. In *Proceedings of the 2nd international Haifa verification conference on Hardware and software, verification and testing*, HVC'06, pages 204–218, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] M. J. Katchabow. Making distributed applications manageable through instrumentation. *Journal of Systems and Software*, 1999.

[11] S. Khurshid and D. Marinov. Testera: A novel framework for testing java programs. In *In IEEE International Conference on Automated Software Engineering (ASE*, pages 22–31, 2003.

[12] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In *The 14th International Workshop on the Implementation of Functional Languages, IFL02, Selected Papers, volume 2670 of LNCS*, pages 84–100. Springer, 2002.

[13] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[14] T. Li and T. Bollinger. T.: Distributed and parallel data mining on the grid. In *Proc. 7th Workshop Parallel Systems and Algorithms*, page 2003.

[15] A. S. McGough, A. Akram, L. Guo, M. Krznaric, L. Dickens, D. Colling, J. Martyniak, R. Powell, P. Kyberd, and C. Kotsokalis. Gridcc: real-time workflow system. In *Proceedings of the 2nd workshop on Workflows in support of large-scale science*, WORKS '07, pages 3–12, New York, NY, USA, 2007. ACM.

[16] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[17] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.2 edition, Jan. 2007.

[18] Y. Ridene, N. Belloir, F. Barbier, and N. Couture. A DSML for Mobile Phone Applications Testing. In *proceedings of 10th Workshop on Domain-Specific Modeling in SPLASH 10th Workshop on Domain-Specific Modeling in SPLASH*, page nc, France, 10 2010.

[19] D. C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the* 12th *Annual Sun Users Group Conference*, pages 214–225, San Jose, CA, Dec. 1993. SUG.

[20] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *FASE*, 2006.

[21] J. Tufarolo, J. Nielsen, S. Symington, R. Weatherly, A. Wilson, and T. C. Hyon. Automated distributed system testing: designing an rti verification system. In *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future - Volume 2*, WSC '99, pages 1094–1102, New York, NY, USA, 1999. ACM.

[22] J. S. Vetter and B. R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[23] A. Waheed and D. T. Rover. A structured approach to instrumentation system development and evaluation. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[24] A. Waheed, D. T. Rover, and J. K. Hollingsworth. Modeling, evaluation, and testing of paradyn instrumentation system. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[25] M. Xiong, J. Parsons, J. Edmondson, H. Nguyen, and D. Schmidt. Evaluating technologies for tactical information management in net-centric systems. In *Proceedings of the Defense Transformation and Net-Centric Systems conference*, 2007.