A Runtime and Analysis Framework Support for Unit Component Testing in Distributed Systems

Jun Li, Keith Moore
Hewlett-Packard Laboratories
Palo Alto, CA 94304
Email: {jun.li, keith.moore}@hp.com

Abstract

This paper presents a test framework to support unit component testing in distributed component-based systems that are built upon component technologies like CORBA, COM/.NET, J2EE/RMI. The framework exploits automatic code instrumentation at the stubs and the skeletons of the calls in order to monitor a global call session. The calls can be cross-thread, cross-process and cross-processor. We further define certain testing-related interfaces for driver components in the component test harness and extend the IDL compiler, such that at runtime, test-related attributes can be automatically embedded in the call session identifier and propagated system-wide. As a result, various support for unit component testing can be enabled, including behavior coordination for stub components, collaborator component determination from historical execution, selective regression testing, and crash site pinpointing.

1. Introduction

Component-Based Systems (such as CORBA, COM/.NET and J2EE/RMI) have improved the construction of large-scale distributed systems. However, testing these systems remains challenging, as often they are deployed in multi-process and multi-processor environments. Furthermore, they are usually long-lived and evolve over time in order to take advantage of new hardware platforms and to provide new system functionalities according to ever-changing business requirements. The combination of complex system features introduction with shortened development cycle time has made test automation critical.

Unit component testing has proven to be critical to ensure the overall system to deliver the functionalities promised [2]. In a large-scale system, a component test harness is itself a distributed component-based application. The system involves the component under-test, and a collection of collaborator components that the component is dependent on. These dependent components may have concrete implementations or be stubbed with controlled implementations. The stub components belong to the driver components in the component test harness. In such a testing-related distributed application, common questions asked are:

- How to test a component as if it were in the full distributed system under development, without running the full system?
- Which test cases in test suites should be rerun when a component is modified?
- How can we effectively test a component under different configurations, such as with/without networking, or with different failure conditions, and how to coordinate different components across the distributed system?
- When a test case failed, how can the test management system report the failure and its root causal information automatically?

Our observation is that large component-based systems are dynamically bound, multi-threaded applications with complex component interactions. The dynamic nature of these systems defeats static analysis approaches that count on a one-to-one mapping between interface and implementations, and the concurrent component interactions and callbacks defeat simple message interception and logging.

Software testing is an infrastructure technology [17]. In this paper, we present a test framework to component testing that leverages the previous work on distributed applications monitoring. The framework automatically instruments the applications by extending the interface definition language (IDL) [11] compiler to insert monitoring code (called probes) into the generated stub and skeleton modules. The probes establish a global session across all the components involved. The data collected from these probes enables the construction of the system-wide dynamic call graph

Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07)
0-7695-2755-8/07 $20.00 © 2007 IEEE
1530-1605/07 $20.00 © 2007 IEEE

that reveals the system-wide function caller/callee relationship at the component level, regardless of reentrancy, callbacks, thread and process boundaries, and unsynchronized clocks.

Furthermore, we carefully define a set of component interfaces for driver components, such that the IDL compiler can inject test-related attributes and actions into the probes, with knowledge support from these driver components. These actions are used to log test-related attributes (such as test suite and test case identifiers), allow these attributes to be embedded into the global session identifier, and apply test-related controls during the test execution.

With the test-augmented runtime monitoring capability, the observed test execution behavior can be automatically linked with the corresponding test cases, both at runtime and offline environment. As a result, we can understand and also control in a test harness, how the component under test interacts with other components, and therefore develop the test techniques traditionally applicable only to sequential or weakly distributed applications. To answer the questions identified previously, we present in this paper several testing techniques, including behavior coordination for stub components, collaborator component determination from historical execution, selective regression testing, and crash site pinpointing.

The paper is organized as follows. Section 2 introduces the distributed systems monitoring framework. Section 3 explains our testing-related extension to the monitoring framework. Section 4 details stub components coordination and Section 5 presents various offline testing analyses. Implementations results are shown in Section 6. Related work is described in Section 7 and Section 8 provides summaries and future work.

2. Distributed Systems Monitoring

Distributed systems monitoring is important to collect runtime information to help determine system-wide component interactions. In distributed and component-based systems, the caller and callee can be located in different threads, different processes, or even different processors. A simple approach of monitoring the execution of each individual thread is not sufficient as cross-thread causality cannot always be determined in re-entrant code (regardless of timestamps).

Our solution to the above problem is to add hidden information between the caller and callee that propagates a global causality identifier [13, 7, 8]. This global causal relationship works cross-thread, cross-process and cross-processor. Figure 1 shows an example of distributed application with four application components deployed in four different machines (from M_1 to M_4). In fact, what Figure 1 illustrates is the Dynamic System Call Graph (DSCG), constructed off-line by analyzing the runtime information captured from the monitored application. This graph unveils all the component-level invocations with their call hierarchies (i.e., both sibling call relationship like GetMetaData and GetFile, and the parent/child relationship like Authenticate and RetrieveMultiMediaDocument).

With the causal identifier propagated system-wide, we can monitor functional behavior (input parameters, output parameters, and possible exceptions) and run-time resource allocation (timing latency, CPU, virtual memory, number of threads spawned, etc.), and have such captured runtime information to be annotated to the constructed DSCG. The user can inspect the captured runtime information when traversing the DSCG graph. For example, in Figure 1, the user can

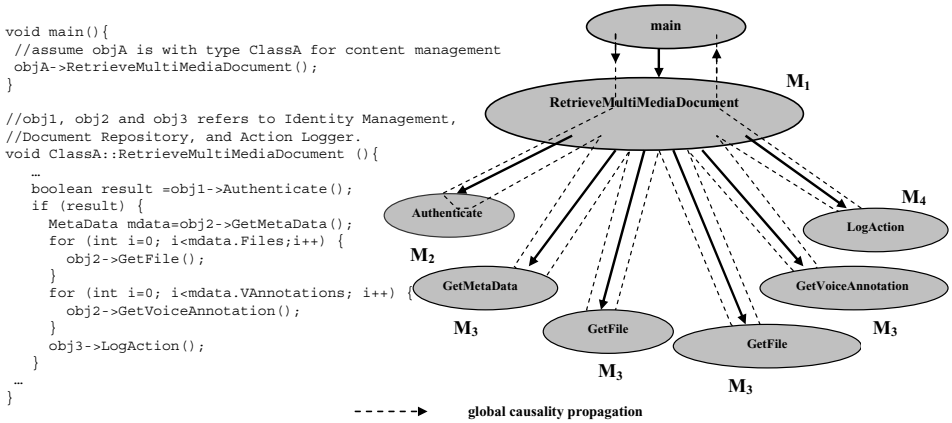

Figure 1: An example of code about remote invocations and global causality propagation

find that the method RetrieveMultiMediaDocument constantly increased virtual memory usage at M_1. Further inspection can reveal that memory leak only happened when the method GetFile (hosted by M_3) was invoked. This finding can then lead to code inspection on method implementation of RetrieveMultiMediaDocument. The root cause of the failure could be that the returned file content to M_1 was stored in M_1's memory for some post-processing, and no memory release was performed at M_1.

3. Test-Related Monitoring Extension

The test-related extension of the distributed monitoring framework covers distributed call session, a defined set of test-relating component interfaces, and the IDL compiler extension. As a result, test-related attributes can be injected, propagated, and captured/retrieved system-wide at runtime. Later at the analysis phase, these test-related attributes can be superimposed to other analysis results, such as DSCG.

3.1. Distributed Call Sessions

We introduce a *global distributed call session* across all the call instances that are with the same causality identifier. A session container is attached to the global distributed call session, to augment the information carried in the global session. The core information contained in the session is the causality identifier, which is defined in `CausalityInfo` (shown in Figure 2) and consists of a UUID to denote the global identifier of this session and an event sequence number that is incremented by one whenever a call-related probing point (belonging to one of the following: stub start, skeleton start, skeleton end, and stub end) is encountered. A session container can further include *application-specific session* information, defined by a value type of `ApplicationSession`. Each application under monitoring can have its own specific session information, with its type extended from `ApplicationSession`. A value type is an object that is passed by value, rather than by reference, whose primary purpose is to encapsulate object state information and to allow application to construct a local copy of an object from the object state [11]. The corresponding language and runtime support in COM is Customer Marshalling [3].

In this paper, the testing-related application session information at least covers:

(1) The scope of the test case currently under execution, represented as a test suite identifier and a test case identifier;

```
module GenericSession {

  struct CausalityInfo {
        UUID global_function_id;
        unsigned long event_seq_no;
  };

  valuetype ApplicationSession{

  };
};


module TestingRelatedSession {

  valuetype TestingRelatedAppSession:
        GenericSession::ApplicationSession{
    public UUID suite_id;
    public long case_id;
  };

  struct SessionContainer {
    GenericSession::CausalityInfo
                          causalityInfo;
    TestingRelatedAppSession appSession;
  };

};
```

Figure 2: IDL definition for call session container

(2) The selective instrumentation indicator. If an interface method in a component is chosen for monitoring, then in this distributed call session, starting from this selected interface method, all the downstream calls will bear this indicator to turn on the instrumentation, but not the upstream of calls in this distributed call session or the other distributed call sessions that are not involved with this particular interface method [7].

3.2. Testing Related Components

To construct a distributed component-based system, following the Component Definition Language (CDL) [5], to each component, in addition to the IDL definitions for the supported interfaces, we have the following information:

- Component Specifications: the supported interfaces, threading (reentrancy), relationship to other components (containment, aggregation, factory, etc.), unique identifier, etc.
- Packaging and Deployment Specifications: to use server thread or Dynamic-Linked-Library (DLL) to host this component.

With automatic code generation from the declarative packaging and deployment specifications, component grouping and deployment in a multi-process environment becomes simple and flexible. In CDL, the IDL compiler that deals with stub and skeleton code

```
interface TCInterface {
 void register (in TestCaseID case,
             in TestSuiteID suite);
};

interface TestInvocationInterface {
    void perform_test1();
    void perform_test2();
    ….
};

component TestComponent {
    supports TCInterface;
    supports TestInvocationInterface;
};


interface ComponentStubbing{
    UUID  suite_id();
    long  case_id();
};
```

Figure 3: Testing related specifications

automation becomes one particular running phase of the CDL compiler.

A test harness typically involves one *Component under Test*, a driver component called *Test Component*, and a collection of *Collaborator Components* that the component under test are dependent on. More specifically, the test component is responsible for creating an instance of the component under test, and exposing a collection of test case execution handlers to the test client. Examples of these handlers are *perform_test1* and *perform_test2* in `TestInvocationInterface` (shown in Figure 3). We define a special method called *register* in `TCInterface` (with concrete implementation provided). It accepts the test suite identifier and test case identifier as the inputs. This method is always invoked in the test case execution handlers. Through this method, the test suite and test case identifiers are all pushed into the distributed call session container, and then propagated in the test harness.

A collaborator component that interacts with the component under test can be replaced with a *Stub Component*, to simulate various behaviors which may occur in a final implementation of this collaborator component. Its purpose is to verify the behavior of the component under test in a controlled execution environment. A stub component has its implementation to follow the exact same interface(s) provided by a concrete collaborator component. Such controlled response can be as simple as to return a pre-defined result or throw an exception. In addition to the interfaces defined by the user-defined application component, the corresponding stub component supports the interface `ComponentStubbing`. This interface is equipped with two methods, namely, suite_id() and case_id(), to retrieve the test suite and test case identifiers respectively, from the distributed call session's container. These two methods are invoked during the execution of the stub method defined in the stub component, to query which specific test case is currently under execution.

Therefore, the methods: register(.) and suite_id()/case_id() collectively form a push/pull model to the testing-related attributes. The register(.) method takes the test suite and test case identifiers, and puts them into a Thread-Specific Storage (TSS) area in the client thread. Before a request message is sent by this calling thread, the call session container populates these two identifiers by pulling them from the TSS. At the server side, after the call request arrives, the test-related identifiers are unpacked from the container, and pushed into the TSS of the server thread. Finally, the suite_id()/case_id() pulls the respective identifier out from the TSS. The actual action about how these identifiers are pushed to or pulled from the session containers is done by the instrumented stubs/skeletons.

By interface querying of `ComponentStubbing`, we can further determine whether the component is a stub component or a real collaborator component.

Both the test component and the stub components are referred to the *driver components*. With CDL specifications and its compiler support, a component test harness can be constructed flexibly, after the component under test and driver components are identified. For example, depending on high-level testing objectives, the test harness can be deployed as a single-process system to test functional behaviors, or a multi-process system to test resource consumption, or a multi-processor system to test communication bandwidth and workload related issues.

3.3. Linking DSCGs with Test Cases

In a component test harness, the user follows the set of testing related interfaces and methods defined in Figure 3 to construct test cases. The monitoring related stubs/skeletons are generated from the IDL compiler. Other code generation involved is from the CDL compiler about component packaging and deployment. During the test execution of the entire test suite, monitoring information is captured [8]. At the analysis phase, we are able to construct the dynamic system call graph (DSCG) for the monitored distributed component-based application. Regarding testing-related attributes capturing, through the IDL

instrumentation occurred at the stubs/skeletons, we can allow test suite identifiers and test case identifiers to be captured after the register method is called. As the distributed call session propagates, at each component (test component, or component under test, or a collaborator component), these two testing-related identifiers are captured/propagated in order to keep track of the particular test case that the current component is exercised for. Further, at each component, the information on whether a component is stubbed or not is captured. As a result, once the DSCG is constructed, we can superimpose the call graph with the testing-related information that we captured and determine which sub graph in the DSCG corresponds to a particular test case. Each graph node is automatically annotated with its test suite identifier, test case identifier, and the distinction of the methods that are supported by either stub components or concrete components.

Figure 4 shows an example of an annotated dynamic call graph for a particular test case, following the example in Figure 1. If capturing of functional behaviors is turned on, for example, to capture the thrown exceptions from different components in different test cases, the DSCG's node inspection can reveal such captured runtime failure as well.

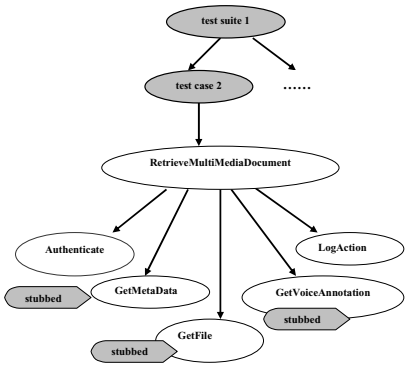

Figure 4: Automatic annotation of DSCG from test suite execution

4. Dynamic Coordination of Stub Components' Controlled Behavior

In a complex distributed application, a component under test likely involves multiple stub components, each of which provides a controlled response to the component under test or other collaborator components that are with concrete implementation. The controlled response can be varied from one test case to the other, depending on a particular behavior aspect that the test case is targeted to. Therefore, in the construction of the component test harness, the stub components need to conform to their user-defined component interfaces, but their behavior-controlled implementation is modifiable. For example, for a content-management component under testing, to the method *RetrieveMultiMediaDocument*, the stub component *IdentityManagement* requires its stubbed method *authenticate* to return true or false, or throw invalid user exceptions, depending on which test case is currently under execution. *RetrieveMultiMediaDocument* is also involved with the component D*ocumentRepository*, a stub component. Its method *GetFile* will return a large file, or no file found, or respond after a significant time delay, depending on the test case under current execution.

Overall, to respond to a test case, the stub components in the component test harness need to coordinate their behavior (their controlled response) in order to satisfy the needs of the current active test case. However, the component test harness is a distributed application, which can have the involved components deployed in different processes. To coordinate the controlled behavior of different stub components located in different processes becomes an issue.

One solution is a centralized test case coordinator that allows the test component to push the test suite and test case identifiers to the central store, and have the stub components to pull the corresponding identifiers from the store. The primary disadvantage is that it disallows concurrent execution of test cases, as a stub component might be serving two test cases simultaneously, each of which is originated from a different process.

A better solution is to take advantage of the distributed call session container to perform such coordination. More specifically, the test suite and test case identifiers are stored in the session container, propagated in the entire component test harness, and retrieved by the related stub components right at the point where it needs to provide its test-case-specific response. The further advantage of this distributed controlled behavior coordination scheme is that it can be well integrated with the test-case-aware DSCGs described in Section 3.3.

An example of incorporating test suite and test case identifiers into stub components to achieve distributed behavior coordination is shown in Figure 5. Each if/else if structure accounts for different stub behaviors required by different test cases. The stub behavior is switched, to handle failure related situations such as IO exception or authentication exception, depending on the test suite id and test case id retrieved in the stub

method. Other common failure-related situations can include long time delay, invalid message that leads to marshalling related exceptions, etc.

```
byte[] DocumentRepository::GetFile (…)  {
 if(suite_id()==Suite1.SUITE_ID) {
   if (case_id()==1) {
        throw new  FileIOException();
   }
   else if (case_id()==2) {
      //large file
      byte[] content = new byte[100000000];
      Random.create(content);
      return content;
   }
   else {
      //average-size file
      byte[] content = new byte[100000];
      Random.create(content);
      return content;
   }
 }
}

boolean IdentityManagement::Authenticate(…) {
 if(suite_id()==Suite1.SUITE_ID) {
  if (case_id()==6) {
       throw new InvalidUserID();
  }
  else if (case_id()==7)
     return false;
  else
     return true;
 }
}
```

Figure 5: Stub component coordinated response

5. Testing Related Analyses

This section presents three offline testing-related analyses to explore the runtime support about testing-related distributed call session. The analyses rely on the component-interaction models discovered from runtime monitoring, in particular, the DSCG and its automatic annotation of testing-related attributes.

5.1 Collaborators and Stubbing Boundary Determination

In an evolvable large-scale distributed application, two different consecutive releases typically only have slight modifications to a small set of components, especially when a release cycle is short. We can take advantage of the old code from release R_{i-1}, to test the new code in release R_i. One practice is to incorporate the collaborator components with their implementation from R_{i-1}, into the component test harness for component C in R_i. A version control system supports such mixing of components from two different releases. If a collaborator component is unchanged, reusing the previous component release greatly reduces development effort for component testing. For a complex application, test developers, often not component developers, likely encounter the following problems: (1) What are the collaborator components to construct the component test harness? (2) If a real collaborator component in R_{i-1} is incorporated, what other dependent components in R_{i-1} need to be included? (3) Component collaborator inclusion tends to grow due to inter-component dependencies. Manual stubbing should be employed to stop further inclusion. Given multiple stubbing choices, which choice leads to minimum stubbing effort and therefore the components from R_{i-1} can be reused at the largest degree?

The solution relies on thorough searching of the stored DSCGs collected from all the component test harnesses and the system integration test harness in R_{i-1}. Here we assume that the system integration related test suites **S** cover all the interactions happened in all unit component test harness (this assumption will be relaxed later). For a component **P** in a unit-component test suite **U** in release R_i, we initialize n=0, P_n=**P** and Q^{i-1}=∅, and carry out the following:

(S1) For every test suite $s_j^{i-1} \in S^{i-1}$ indexed with j, scan its associated $DSCG_j^{i-1}$, and identify the components Q_j^{i-1} that component P_n has ever directly interacted in s_j^{i-1}'s execution;

(S2) The aggregation of all these identified components over all test suites to achieve Q^{i-1}, i.e., $Q^{i-1} = Q^{i-1} \cup (\bigcup_j Q_j^{i-1})$. Q^{i-1} represents all the components that can directly interact with component P in Release R_{i-1}.

(S3) For each component $q \in Q^{i-1}$, if the user intends to include the real implementation of q in Release R_{i-1} to the target test suite U, we have n=n+1 and P_n=q, and revisit S1 for the next searching round.

The final Q^{i-1} is the set of collaborator components from R_{i-1} to construct the test harness for component **P** in R_i.

At S3, if the user decides not to propagate further, but complete manual stubbing for the entire component q is too expensive, partial manual stubbing to a subset of the interface methods for component q is appropriate. The implementation of the non-stubbed interface methods can still leverage the real component implementation of q via certain degree of source code reuse. Because the DSCG actually shows component interactions at the interface method level, at S1, we can determine the number of the components in R_{i-1} that a

particular interface method has ever interacted with. The criterion is that the larger the number of the components this interface method ever interacts with, the more suitable this interface method is to be manually stubbed.

In practice, system integration testing does not necessarily provide the same coverage as unit component testing and therefore cannot fully substitute unit component testing. Often unit component test suites exercise local component interactions more thoroughly in order to expose component defects as early as possible. The scanning procedures described in S1 and S2 should also include all the DSCGs from the execution of unit-component test suites. In addition, in each component test suite, an interaction is valid if the target component $q \in Q^{i-1}$ is not marked with "stubbing" and q is labeled with R_{i-1}.

5.2 Regression Test Cases Selection

It would be very inefficient and time consuming to rerun all the unit component test suites when a component is modified in the late development phase of the large-scale distributed application. Such a change can incur to component implementation with possible interface modification. It is important to determine only a subset of test cases in a subset of test suites that really needs to be rerun to ensure that the system still functions as it is supposed to be after the change. Or at least, to prioritize the selected subset of the test cases that needs to be rerun first.

In the current release R_i, the DSCGs can be uncovered automatically for all the available component test suites, by turning on runtime monitoring for all these test suites and applying offline analysis onto the captured runtime information. In each DSCG associated with a test suite, following Section 3.3, we can determine a sub-graph, which is a DSCG by itself and corresponds to a sequence of interface method invocations in a test case execution. If a change occurs to component C in release R_i, a test case needs to be rerun if a test case's DSCG contains at least one method invocation to component C that satisfies the following two conditions: (1) Component C is not stubbed. The stubbing component has the customized implementation in a specific test harness, and is different from final real component's implementation; (2) Component C is from release R_i. Components from release R_{i-1} are historical and therefore not subject to change any more.

Since DSCGs reveal component interactions at the interface method level, this change impact estimation can be actually applied to a specific method M belonging to component C with more precise estimation.

5.3 Crash Site Pinpointing

A component under test harness is a distributed component-based system that can potentially involve many components across different processes and even different processors. When the system crashes during test case execution (for example, nightly test execution after system build), it is valuable to determine under which test case the execution crashes, and which component (either the component under test, or one of the collaborator components) is the culprit for the crash. Once the culprit component is determined, such information can be further reported to the high-level test management system, from which the developer responsible for the component will be notified, along with the captured runtime information.

Our approach to automatically pinpoint the crash point is to extend the algorithm that uncovers the DSCG [8]. This algorithm is based on a simple state transition model of the events: stub start, skeleton start, skeleton end, and stub end, produced by the stub/skeleton-based probes. We assume that the crash site always happens only to the implementation of the user-defined application components, because the middleware runtime infrastructure and the IDL compiler are generally well developed and tested, and thus much more reliable.

We further assume only a single failure happened to a single component that leads to the crash. To prevent monitoring data loss due to process crashing, we need to ensure that every log collected in an active thread (participating in a distributed call session) will be recorded immediately to the persistent storage, such as a file system. Each log file is designated to one process (all log files will be collected to a central data repository offline for monitoring/test related analyses). This check-pointing mechanism allows us to have the latest system snapshot just before the crash. Once the crash occurs, through certain operating system monitoring utility support, such as the WMI management package in Windows, we can determine the crash site at the process level.

By applying the DSCG construction tool to check log information, we can determine which distributed call session is terminated prematurely, if the probing event in the session cannot follow the correct state-transition model. The checking is done as follows. According to our failure model, a chain *c* (that is, the call session) is identified to be broken, if (1) *c* is terminated with last event happened in the identified crashed process; and (2) the last event of c is a *skeleton*

start (raised by the *skeleton start* probe). This is because it is only after this probe that the thread of control reached the user-defined method implementation, and encountered crashing in this method's execution.

If *c* is unique, the interface method that leads to the crash can be identified from the last skeleton start event logged. The associated component and interface identity can be inferred from this probe's other monitoring information. The identified component is called a *crash* component. However, *c* might not be unique, if when the crash happens, some other threads in the process are in a waiting state (waiting for a lock or a synchronous IO read/write). In general, from all distributed call chains that meet Criteria 1 and 2, we can identify a set of *crash candidate* components, if no unique crash components can be determined. As a result, the test management system can report both the crash components and the involved call chains to the responsible component developers.

For the developers to pinpoint the exact location of the crash site, we can take advantage of flexible repackaging and re-deployment of component-based systems. A crash candidate component X is chosen, based on the crash analysis results. In the test harness, we modify its component configuration, such that X is packaged now in a newly declared server thread T_{new}, and this T_{new} is the only thread in a newly declared process P_{new}. We then rebuild the entire component test harness and re-execute the test suite. If P_{new} is the crash site, we can determine that X is the crash component. Otherwise, a new crash candidate component is chosen, and the re-packaging, re-deployment and re-execution of the test suite is performed in turn to determine whether the newly chosen X is the crash component. If the locking and IO related APIs are also instrumented, the user can have more knowledge to exclude those candidate components which were actually in a waiting state due to the locking or IO, and therefore speed up this crash pinpointing analysis.

6. Results and Discussion

The complete test infrastructure is still under development. Herein we show the implementation results that we have achieved in distributed systems monitoring and related testing supports, on the way to build the comprehensive test framework.

Our distributed monitoring framework, coSpy, has been developed and demonstrated on a large-scale industrial embedded application (built-upon a COM-like middleware infrastructure). The largest system run captured up to 200,000 inter-component calls in the HPUX simulator environment. The entire system totally has about 2 million lines of code, with a runtime configuration of 4 processes over 32 threads on a single-processor. The full DSCG graph is shown in Figure 6, in which each graph node represents a interface method call in a component.

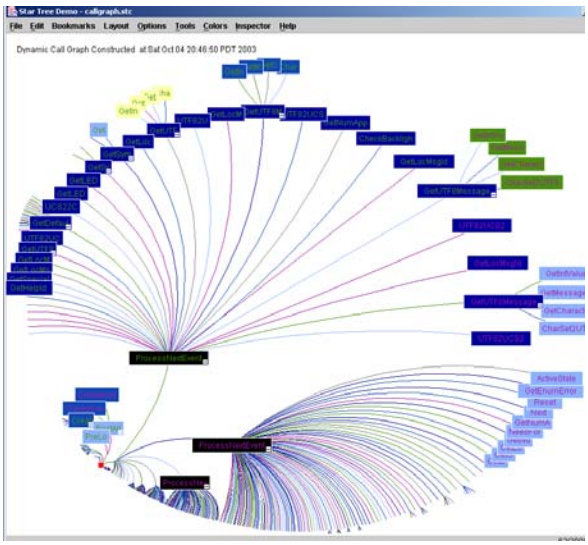

Figure 6: An example of dynamic call graph

By navigating the DSCG graph, each node can be inspected carefully with the associated names of component and interface. Timing latency distribution observed across multiple call instances can be shown as well. Once an application crashes, the DSCG will become incomplete at some tree hierarchy, which can allow users to identify the failure component, as described in Section 5.3.

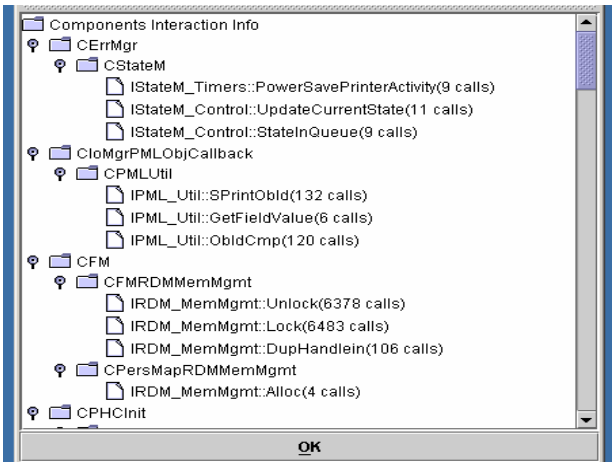

Figure 7: Interface method level test coverage

Figure 7 provides the detailed components and their interfaces/methods involved in the corresponding DSCG from a particular system run. It serves as a test

```
host name: snowhite
process id: 7832
global process id:2988e185ed8745a8af111ce3fece72f300000000
…..
test case: f594363b06a847a68cda6888cfa9135a::100004
IDL:HelloWorld:1.0 test_call_back <skel start> 1182756:388071 1182756:388082 5(thread) fca81cdbfc4a489e8e2cfc243b06272f00000000(global) 7 13 0
IDL:CORBA/Object:1.0:cb011cde-c8fc-4c2e-a211-a2377ce1538d(end point)+2f372531-b7c3-405c-9e0b-556fb3893860(object key)[1] 4[2]

test case: f594363b06a847a68cda6888cfa9135a::100004
IDL:CallBack:1.0 greeting <stub start> 1182756:390365 1182756:390372 5(thread) fca81cdbfc4a489e8e2cfc243b06272f000000000(global) 8 14 0
test case: f594363b06a847a68cda6888cfa9135a::100004
IDL:CallBack:1.0 greeting <stub end> 1182756:397837 1182756:397850 5(thread) fca81cdbfc4a489e8e2cfc243b06272f00000000(global) 8 17 0
Greeting from CallBack 1(return)
….
test case: f594363b06a847a68cda6888cfa9135a::100004
IDL:HelloWorld:1.0 test_call_back <skel end> 1182756:429437 1182756:429520 5(thread) fca81cdbfc4a489e8e2cfc243b06272f00000000(global) 7 30 0
…..
test case: f594363b06a847a68cda6888cfa9135a::100009
IDL:HelloWorld:1.0 test_raiseexception_2 <skel end> 1182756:500343 1182756:500385 4(thread) fca81cdbfc4a489e8e2cfc243b06272f00000000(global) 27 54 0
IDL:ConceptNotFoundException:1.0(user exception)
```

Figure 8: Log file with test case identifier propagation and input/output parameters logging

coverage report. From our experiments on the system integration testing, it is found that only 800 interface methods out of the total 4000 interface methods were exercised in the particular test run that produces our largest call graph, indicating only 25% coverage for this test case (with a particular type of document file as the input). Our experiments further showed that other similar test cases, each of which is different only in terms of test input (different document content for processing), do not improve the test-coverage significantly. Thus, our tool offers the assistance on identifying missing test cases for the component(s) under test.

We have successfully incorporated test suite and test case identifiers into distributed call session. Figure 8 shows a portion of log file that we obtained from a running application (with 2 processes) built upon our CORBA-based infrastructure [10]. In each log, the first line shows the current test case identifier, and the second line shows the associated probing event (stub/skeleton's start/end) and causality identifier. Optionally, the third line shows function parameters (input/output/return) captured and possible exception thrown from the method invocation. The detailed function call logging is useful for the developers to figure out the root cause of test case failure.

7. Related Work

Here we compare our framework support to other techniques focused on distributed system testing.

Model-based testing environments often provide runtime monitoring to validate user-defined models. In Rhapsody [14], code is generated from user-defined specifications. Event-based component interactions can be captured and validated, as the tool provides a thread-based object execution framework. Regarding the capability of controlling the middleware runtime and compiler support, our approach is focused on a broader range of distributed environments with multiple processes and processors, rather than just the single-processed environment.

Regression test case selection has been explored for C/C++ and Java applications by combining static programming analysis and dynamic system tracing [4, 16, 6]. This approach is effective for single-processed and multithreaded applications, but not for multi-processed applications, as their tracing is at local procedure call or basic-block level, and does not explore causality linking between the call parties located at two different processes.

Regarding component behavior checking, a component's behavior is specified as executable specifications [1]. A proxy is installed between a client and the component, which forwards the client request messages to the execution engine. The results from the server component and the execution engine are compared by the proxy for behavior equivalence between the component and its specification. However, the behavior checker seems to deal with a single component rather than multiple components.

Capture/replay technique has been explored to test server-side application. In QALoad [12], all bi-directional communication messages between a client and the application server are recorded. The captured one-client-one-server application run becomes the seed for load testing. Multiple clients hosted in different machines can be replicated to stimulate the server with previously recorded request/response messages. It is difficult to extend this technique to multiple servers, since message channel interception is not sufficient to ensure that the ordering (causality) replay of the messages from multiple interacting parties.

In distributed test support, TCBeans [18] facilitates test case implementation in Java Beans. RMI allows these test cases to be distributed for concurrent

execution. Execution parallelism is also explored in [9] by dynamically scheduling different testing suites onto different networked computers. Such frameworks only reveal test execution results like PASS/FAIL, without controlled behavior coordination and per-test-case detailed failure reports.

8. Conclusions

We presented a test framework with runtime and analysis support, to focus on unit component testing for distributed component-based systems. The test framework relies on the distributed systems monitoring tool that has been demonstrated on both COM and CORBA based applications. This test architecture allows a distributed call session container to be propagated in a component test harness, with testing related attributes to be injected into this container. As a result, we can at runtime coordinate the behavior of multiple components involved, and at offline environment to construct component-interaction model from the captured test suite execution. The linking between the test cases and the associated runtime behavior model can be revealed. From such behavior model, various testing analyses are presented, including collaborator components determination, test case selection/prioritization and crash component report and pinpointing. The unique features of our test architecture are:

- A monitoring framework facilitates an observable testing environment, without modifying application components and their interfaces;
- With the IDL compiler extension and runtime testing-attributes injection, the linking and synchronization between the test-related artifacts and implementation-related artifacts is simplified;
- The component-interaction model automatically constructed from the monitored test execution enables various systematic test supports for component-based applications' maintenance and evolution, both at runtime and offline environments. Such tool supports distinguish themselves from others in a cross-process and cross-processor environment that involves multiple implementation languages.

For future work, we can incorporate our stub component coordination into unit testing tools like JUnit and NUnit, to leverage their automatic test execution environment. Capture/replay based testing that involves more than two server components in the call session [15], is also interesting. Enforcing global session propagation with exact historical call ordering is the key. Our testing techniques reported are focused on unit component testing, certainly various issues, e.g., scalability, will arise, when we try to extend such techniques into system integration testing.

9. References

[1] M. Barnett and W. Schulte, "Spying on Components: A Runtime Verification Technique," OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems, Oct. 2001.

[2] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addision-Wesley Pub. Co., 2003.

[3] D. Box, Essential COM, Addison-Wesly, Pub. Co., 1998.

[4] Y. Chen, D. Rosenblum, and K. Vo, "TestTube: A System for Selective Regression Testing," Proceedings of the 16th International Conference on Software Engineering, pp. 211-220, 1994.

[5] P. Fulghum and K. Moore, "CDL," HP Internal Design Document, Dec. 1999.

[6] M. J. Harrold, "Regression Test Selection for Java Software," Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 312-326, Oct. 2001.

[7] Jun Li, "Monitoring of Component-Based Systems," HPL Technical Report HPL-2002-25(R.1).

[8] Jun Li, "Monitoring and Characterization of Component-Based Systems with Global Causality Capture," Proceedings of the 23rd International Conference on Distributed Computing Systems, pp. 422-31, May 2003.

[9] J. Mathews, "Distributed Automated Software Graphical User Interface (GUI) Testing," US Patent No. US 2003/0098879.

[10] K. Moore, and E. Kirshenbaum, "Building Evolvable Systems: the ORBlite Project," *Hewlett-Packard Journal*, pp. 62-72, Vol. 48, No.1, Feb. 1997.

[11] OMG, *The Common Object Request Broker: Architecture and Specification,* Oct. 2000.

[12] QALoad, http://www.compuware.com/.

[13] F. D. Reynolds, J. D. Northcutt, E. D. Jensen, R. K. Clark, S. E. Shipman, B. Dasarathy, and D. P. Maynard, "Threads: A Programming Construct for Reliable Real-Time Distributed Computing," *Internal Journal of Mini and Microcomputers*, Vol. 12, No. 3, 1990, pp. 119-27.

[14] Rhapsody, http://www.ilogix.com/rhapsody/.

[15] M. Ronsse and K. D. Bosschere, "RecPlay: A Fully Integrated Practical Record/Replay System," ACM Transactions on Computer Systems 17, 2 (May 1999), pp. 133-52.

[16] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression Test Selection for C++ Software," Journal of Software Testing, Verification and Reliability, pp. 77-109, Vol. 10, No. 2, June 2000.

[17] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," Technical Report, National Institute of Standards and Technology (NIST), May 2002.

[18] TCBeans, http://www.haifa.il.ibm.com/