Volume 2, Issue 10, October 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Monitoring Interactions in Component-Based Systems

Srinivas Reddy*
Assistant Professor
Mahatma Gandhi of Institute of Technology
Hyderabad,A.P,INDIA

Prof.T.Venkat Narayana Rao
Professor
Department of Computer Science and Engineering
H.I.T.A.M,Hyderabad,A.P,INDIA,

Abstract – Monitoring, analyzing, and understanding component-based enterprise software systems are challenging tasks. These tasks are essential to solve and preventing performance and quality problems. Obtaining component-level interactions that show the relationships between different software entities is a necessary prerequisite for such efforts. This paper focuses on component-based Java applications, currently widely used by the industry. They cause specific challenges while raising interesting opportunities for component-level interaction extraction tools. This paper present a range of representative approaches for dynamically obtaining and using component interactions. For each approach, it addresses the technical requirements for building an implementation of the approach. We have also looked at different available implementations of various techniques existing . It gives performance and functional considerations and contrasts them against every other by outlining their relative advantages and disadvantages. Based on this study, developers and system integrators can better understand the current state of the art and the implications of choosing and implementing different dynamic interaction extraction techniques in diverse software environment.

Keywords – Software System, Component based development, Dynamic Interaction, System Integrators, sub-system.

I. INTRODUCTION

The current enterprise applications are very often huge and complex systems, made up of a multitude of different software components that communicate to service client requests. With such systems (which are commonly built using enterprise component frameworks), it can be difficult to understand how exactly particular components interact at runtime. This can lead to a lack of overall system understanding, which, in turn, can manifest itself in a range of different problems (e.g., incorrect performance tuning, maintainability issues, etc.). Component-level interactions (CLIs), which capture component communication and can be recorded as the program executes (i.e., at runtime) or beforehand (statically). While dynamic traces can be limited by the input data, they have the advantage of recording the actual interactions that occur during execution. Static traces, on the other hand, can be used to determine all potential paths through the system. Thus, it is important that we have techniques offered to the developers, whereby they can record both static and dynamic traces. The usage of a component in a software system includes using it to replace an out of date component to upgrade the system or a failed component to repair the system, adding it to the system to widen the system services, or composing it into the system while the system itself is still being built. Some researchers insist on a component being reusable during dynamic reconfiguration [3]. The implications of properties are different when a component is used in different applications, for different purposes or in different kinds of systems. This is the main reason why some people give more tough definitions than a component is defined by the following three axioms:

- A1- A component is capable of performing a task in isolation; i.e. without being composed with other components.
- A2- Components may be developed independently from each other.
- A3-The purpose of composition is to enable cooperation between the constituent components.

These properties are in fact those required for a "sub-system" . This paper argues that the three axioms further imply a number of more properties, called corollaries of components i.e. :

- C1- A component is capable of acquiring input from its environment and/or of presenting output to its environment.
- C2 -A component should be independent from its environment.
- C3 The addition or removal of a component should not require modification of other components in the composition.
- C4-Timeliness of output of a component should be independent from timeliness of input.
- C5-The functioning of a component should be independent of its location in a composition.
- C6-The change of location of a component should not require modifications to other components in the composition.
- C7-A component should be a unit of fault-containment.

The implication of the corollaries from the axioms is only argued informally. Property C2 implies that a component has no state. This is now generally understood to be only required in some limited circumstances, such as for dynamic reconfiguration. Property C4 only applies to real-time systems and properties C5 and C6 are only relevant to distributed mobile systems. We do not see why C7 is needed at all unless a component is to be used to replace another during the

runtime of the system. In fact, in many applications coordinators or managers can be used to coordinate fault-prone components to achieve fault-tolerance [1].

A. Interfaces

Although there is no accord on what components are, all definitions agree on the importance of interfaces of components, and interfaces are for composition without the access to source code of components. This indicates that the differences are largely reflected in decisions on what information should be included in the interface of a component.
We further argue that interfaces for different usages and different applications in different environments may contain different information, and have different properties:

1) An interface for a component in a sequential system is obviously different from one in a communicating concurrent system. The later requires the interface to include a description of the communicating protocol while the former does not.
2) An interface for a component in a real-time application will need to provide the real-time constraints of services, but an untimed application does not.
3) Components in distributed, mobile or internet-based systems require their interfaces to include information about their locations or addresses.
4) An interface (component) should be stateless when the component is required to be used dynamically and independently from other components.
5) A service component has features different from a middleware component.

Therefore, it is the interface that determines the external behaviour and features of the component and allows the component to be used as a black box. Based on the above description, our framework defines the notion of an interface for a component as a description of what is needed for the component be used in building and maintaining software systems. The description of an interface must contain information about all the viewpoints among, for example functionality, behaviour, protocols, safety, reliability, real-time, power, bandwidth, memory consumption and communication mechanisms, that are needed for composing the component in the given architecture for the application of the system. However, this description can be incremental in the sense that newly required properties or viewpoints can be added when needed according to the application [2].

B. Automated Application Level Interaction Recording Overview

An alternative approach to recording interactions for application-level trace extraction is discussed. This section addresses some of the disadvantages of an assisted approach and introduces techniques that can be applied for automated application-level CIE. For example, during system tests, traces annotated with performance metrics can be useful for identifying performance issues in an application under load. However, there are a number of issues that need to be addressed to apply an automated approach to multiuser systems. The problem with the assisted approach is that when simultaneous users are in the system, one user request cannot be distinguished from another. An automated tracing approach for multiuser systems essentially works by identifying new user requests that enter the system, tagging the requests such that each request can be exclusively identified for the duration of the request, and intercepting the requests at different points such that the calls to the different components that make up the request can be logged and their order maintained. The requirements needed to attain this are listed as follows:

- Ra—an instrumentation framework,
- Rb—new user requests must be identified when they enter the system,
- Rc—each different user requests must be tagged with request-specific information (RSI), and
- Rd—the RSI must be accessible across the entire request.

In the following paragraphs, we discuss the different ways in which the requirements Ra-Rd can be achieved in a convenient manner outlining the advantages and disadvantages of each technique. The monitoring framework requirement (Ra) is met in the same way as for the assisted CIE approach.

Rb—Nonintrusive Detection of New User Requests : In order to be able to tag new user requests with RSI (Rc), it is important to be able to recognize new requests when they first enter the system. This can be achieved intrusively by incrementing the middleware .To achieve this in a nonintrusive manner, such that the approach is portable across different middleware implementations, manipulation of the middleware must be avoided. To overcome this, a nonintrusive point-of-entry detection mechanism can be added to the IPs. The point-of-entry detection mechanism contains logic that is used to determine the point of entry of each new request that enters the system [7]. This works by monitoring the deepness of the calls made on the current thread. The depth value is incremented when entering a method and decremented upon method exit. A depth of zero indicates that the current method is the first method called in the runtime path and is thus the point of entry into the system. A major advantage of this point-of-entry mechanism is that it can be added to the monitoring IPs in each of the different application tiers, and thus, new user requests do not have to be restricted to a single tier like with previous intrusive approaches. The call depth data is stored as part of the RSI.

Rc-Tagging User Requests with Request Specific Information: When a new request enters the system, it needs to be tagged with RSI such that :
1) all calls made during the request can be identified as being members of that particular request.
2) the correct order of the calls are maintained.
3) the depth of the call path is recorded for point-of-entry detection .
The RSI contains a unique ID assigned to each fresh request, as well as call path depth data and call path sequencing data[4]. The unique ID is used to identify the dissimilar calls that make up each request. The call path sequencing data is used to maintain the order of calls that make up a user request. When a new user enters the system, the sequence data is placed to a value of zero. At each method entry IP, the sequence number is incremented. This information can be used at a later stage to build an ordered representation of the calls that make up the user request [6]. The RSI can be attached to the request using the Java Thread Local object. Every Java thread has a Thread Local Object associated with it , which can be used to store an object for the lifetime of the thread. One limitation of the Thread Local Object is that the information stored in the object is not propagated to remote method calls. An alternative approach to the Thread Local object might be to use the Work Area service (Java Specification Request, JSR).

Rd—Accessing the RSI throughout the Request: Tagging threads with RSI enables the system to discriminate between different user requests. In order to be able to trace CLIs across the entire request, this RSI must remain attached to the request for its entire lifetime and across all tiers of the JEE application. An issue with using the thread Local approach is that it is limited to tracing requests on systems where Web and application servers are collocated. Servers that are collocated run on the same JVM. Thus, for a particular user request, the same thread is used across the Web server, EJB server, and database driver. A problem arises, however, if calls are made across distributed JVMs [2]. In this situation, a new thread is spawned on the remote JVM to handle any remote calls. The new thread does not contain the RSI, and thus, calls made to the remote component cannot be traced.

C. Server Side Interception

On the server side, the IP is in the form of a proxy layer (or wrapper), Using this approach, all calls to and from the bean can be intercepted. The extra parameter (RSI) sent by the client is handled in the wrapper, which overloads each method such that it contains an extra parameter. This extra parameter is used to piggyback the data across the network. In the context of runtime path tracing, once the data has been sent across the network, it can be attached to the current thread.

D.Dynamic Trace Reconstruction

As each user request passes through the system, calls to each component that make up the system are intercepted at various IPs, and the calls are logged. A representation of the user requests as they passed through the system can easily be constructed from the logged information. To allow for easy reconstruction of the correct event sequence, the order of the calls that make up a request is maintained and logged with each logged call [4]. Each user request can thus be easily reconstructed by arranging the logged events into their correct sequence. The user requests can be represented as dynamic traces, runtime paths, CCTs, or even as call graphs.

II. PROPOSED ARCITECTURE

The main concerns about programming in this context are the flow of control and the data structure. The specifications, design and verification all focus on the algorithm and the data structure of the program [13]. For programming in the large, the major concerns are components and their consistent integration in an architectural context. The architectural design becomes a critical issue because of the significant roles it plays in communication among different stakeholders, system analysis and large-scale reuse. There are numerous definitions of software architecture the common basis of all of them is that an architecture describes a system as structural decomposition of the system into subsystems and their connections. Architecture Description Languages (ADLs) are proposed for architecture description. The basic elements of ADLs are components and connectors. An ADL also provides rules for putting (composing) components together with connectors [9]. They suffer from the disadvantage that they can only be understood by language experts – they are inaccessible to domain and application specialists. Informal and graphical notations, such as UML, are now also widely used by practical software developers for architecture specification .However; the semantic foundation for these UML-based models has not yet been firmly established. A mere structural description of a system is not adequate in supporting further system analysis, design, implementation, verification, and reconfiguration. More expressive power is needed for an ADL .In particular; an ADL should also support the following kinds of views.

A. Models of architectures

Most of the early theories focus on modelling system architectures. All these models of architectures deal with coordination's among components, in an event-based approach. They can also be used for specification of connectors and coordinators. However, they do not go to the level of component design, implementation and deployment. This might be the cause why ADLs still do not play any major role in practical software engineering. Recently, more fragile models are proposed for

describing behaviour of components and their coordination's, a channel-based model with synchronous communication. The composition of components is defined in terms of a few operators. The model is defined operationally and thus algebraic reasoning and simulation are supported for analysis. The disadvantage of this approach is that it is not clear how it can be extended to deal with other viewpoints, such as timing and resources. Also, being event-based, the model considers a layered architecture for composition, provided by connectors (gluing operations) [10]. It considers real-time constraints and scheduling analysis. The behavior of a component is defined in a form of a timed automaton. This provides a good low level model of execution of a component. However, the use of local clocks for modeling delays can hardly be said to be component-based. We need talk about a component at a higher level of granularity. The main disadvantage of message/event based approaches is that changes of the data states of a component are not specified directly. While they are good at modeling behavior of electronic devices and communicating protocols, they are not inclined to the software engineering terminology and techniques. The relation of these models to program implementations is not clear and practical software design techniques, such as design patterns, is not well supported. These lead to difficulties in understanding the consistency between the interaction protocol and the functionality.

B. Adding a Proxy Layer Using Standard JEE Mechanisms

Using standard JEE mechanisms, monitoring probes can be directly attached to the JEE components such that any requests to and responses from the components can be captured. The probes in this technique are essentially component wrappers that add proxy elements corresponding on a one-to-one basis with the application components [5]. Probes can be generated and inserted automatically by leveraging metadata available for JEE components. During the instrumentation process, the original application structure, called the Enterprise Archive in JEE (EAR) is analyzed, and the component metadata is extracted (including the component name, the component type, and the classes that implement it). Based on this information, an instrumentation process can generate the appropriate probes and can create a new EAR application archive with modified metadata in order to accommodate the insertion of probes. At runtime, the probes intercept calls to and from the JEE components. The process is explained in detail in the work of Parsons et al. whereby the authors explain how probes can be attached to Web tier components (JSPs and servlets), business tier (EJB) components, and database tier (JDBC) components. The technique has been used as part of the COMPAS Java End-to-End Monitoring (JEEM) tool. This is particularly important for component-based systems where components may be developed by third parties and licensing constraints may prohibit their modification [6]. The main drawback of this approach, however, is that it requires redeployment of the application under test. While this may be a trivial task for small-scale applications, it can be a long and complicated process for enterprise systems. Another feature of this approach is that it can only be applied to application-level components, i.e., it omits lower level middleware calls. Middleware vendors have also provided vendor-specific ways to achieve a proxy layer for monitoring purposes (e.g., the JBoss interceptor-based component architecture); however, such techniques are not portable across different middleware implementations.

C. Method Overview

This method involves recording component calls from a JEE system in order to construct the component interactions. It requires that only one user be active in the system for the duration of the recording activity. For the recording process to work, this technique requires the following elements:

- R1—an instrumentation framework,
- R2—a recording entity that can capture and store component-emitted events, and
- R3—a sequencing mechanism capable of ordering time stamps and of compensating for measurement imprecision.

The instrumentation framework is used to provide the raw data, in particular the time stamps and method IDs used in interaction reconstruction as shown in figure 1. The recording entity is required to capture all the collected data and store it for processing by the sequencing mechanism. Using this approach, it is Possible to generate the completed interaction containing the ordered method calls (i.e., a untimed path). An important advantage of this technique is that it is inherently distributed as it captures events using standard JEE technologies for distributed computing [12]. Its main disadvantage is that it can only allow one user interaction to execute in the system during recording. Therefore, this technique is suitable in situations where it is necessary to obtain the CLIs corresponding to different business scenarios. It ensures that users have complete control over which scenarios are executed for the duration of the recording process, therefore guaranteeing that the resulted CLI precisely matches the business functionality tested. The results can be used to reverse-engineer an application or to get a general idea of performance metrics associated with a business use case (however, the obtained values cannot generally be used for performance problem identification as they are not representative for multiuser situations).

R1—An Instrumentation Framework : A number of portable instrumentation techniques have been previously discussed in Section 3. For application-level CIE, the proxy-based approach, the AOP approach, or the BCI approach can be used. The proxy-based approach is particularly suited to application-level extraction since it instruments the application- level components only. Both the AOP and BCI approaches require a filtering mechanism during the instrumentation process to confine instrumentation to the application components only. To extract component interactions, instrumentation is added to

each component method. More precisely, each component method is instrumented with two method calls: one at the beginning of the method (or directly before the method) and one at the end of the method (or directly after the method call) [11]. We refer to these as the method entry instrumentation point (IP) and the method exit IP.

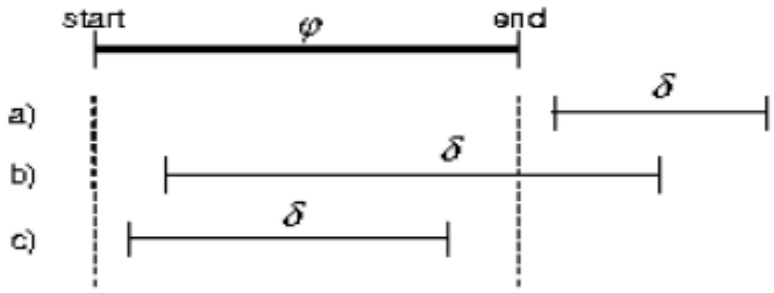

Fig. 1 Enclosing Methods

R2—Recording Entity:At runtime, the IPs collect performance information obtained from their target application components and communicate with a centralized unit via standard Java/JEE mechanisms (such as Java Management Extensions—JMX). This centralized entity is responsible for recording the Application interactions and can be used to provide monitoring and control functionality for the IPs such that recording can be initiated and terminated. The recording entity uses the IPs to extract method execution events from the running application [14]. It orders the events collected into complete interactions by using time stamps collected by the IPs. With this approach, developers are required to manually record the interactions they are interested in during a training session and instruct the system when to start and stop recording via a user interface

R3-Sequencing Mechanism :The sequencing process commences when the user decides to terminate recording mode through the recording entity. The following steps are executed as part of the sequencing process:

1) The data set containing the stored method invocation events is ordered in the ascending order of the method invocation start time stamps (i.e., methods that started executions earlier are placed at the beginning of the data set).
2) Parsing the method invocations data set for each method _, the list of methods preceding it in the sorted data set is analyzed to find a possible enclosing method. as illustrated in Fig. 1. Where in the start and end times of method are included in the interval created by the start and end times of method ’.
3) If an enclosing method’ is found that satisfies the case presented in Fig. 1, then method is added as a child to method’ in the interaction tree that represents the recorded interaction model.

III. IMPLEMENTATION

The implementation makes use of the ThreadLocal approach outlined to satisfy R3 (tagging user requests with RSI). However, its main weakness is that it instruments the middleware to satisfy the remaining requirements, and thus, it is not portable across different middleware implementations. Pinpoint collects application-level component interactions in the form of runtime paths. The COMPAS JEEM monitoring tool also collects application- level runtime paths for JEE applications. It makes use of pinpoint libraries to satisfy R3. It satisfies the requirements R1, R2, and R4. Thus, it is completely portable across different middleware implementations and cannot be applied to components that communicate across physically distributed JVMs.

The WebMon1 monitoring tool overcomes this issue by applying the technique. Another tool that has the ability to collect application-level (and system-level) interactions for JEE applications is the InfraRed monitoring tool. Infrared instruments the application using AOP. It also uses Thread Local to tag user requests with RSI (R3). Similar to the approach, the Infra Red tool uses a depth counter stored within the RSI to maintain the level of calls within particular application tiers . To make out a request entering the system (R2), the Infra Red developers suggest also using a ThreadLocal approach together with a specially designed aspect or a filter servlet. They do not marshal data across the network. Instead, they instrument the caller and callee and simply relate the calls via the method name. However, using this approach, it is not clear how the order of calls can be maintained across remote calls when the system is under load. The information they send across the network differs from the information sent in the approach outlined above. They make use of the identifier of the client node (client Node ID), the thread identifier (client Thread ID), the class name (client Class Name), and the object identifier (client Object ID). However, the approach basically works in the same way since their information is used to be able to identify the client thread responsible for remote communication. In addition, the instrumentation code is inserted into the running JEE system, without the need for a server restart. This is a completely transparent operation, making it particularly useful in production environments. This technique has the following requirements:

- R1—an instrumentation framework with hot swapping capability,
- R2—a class-hierarchy analysis technology,
- R3—a coverage adjustment technology to switch between high-level and detailed monitoring,
- R4—a mapping controller for driving low-level instrumentation to correspond to CLIs.

As this technique can obtain container level component calls as part of the CLIs, it has an benefit over the previous CIE techniques as it can present the user with the configuration context of potential quality and performance problems. Thus, problems that, with previous techniques, may appear to stem in the logic of the application can in fact be established out with this technique to create in poor configuration parameters for the application server. By identifying which container services are involved in a hot spot, the user can be directed to the server configuration responsible for altering these services. This extra information can come at the cost of additional overhead; however, this technique can switch between high-level monitoring and the detailed monitoring. This coverage adjustment technology makes the technique particularly suitable for production environments. It must be noted, however, that the method presented in this section suffers from the drawbacks of the CCT representation, mainly that it cannot distinguish between different user identities as shown in figure 2 and 3. This makes it difficult to identify the causes of problems when they occur in isolated scenarios.

R1—Monitoring with Hot swapping Instrumentation: This technique is based on instrumentation capabilities that allow the dynamic instrumentation of methods that are automatically discovered starting from a set of given root methods. Dynamic BCI provides the basic capability to insert monitoring code in application bytecode during execution. Each method called directly or indirectly (via other methods) by a predetermined root method and would be instrumented automatically by this technology by changing its bytecode. The new bytecode adds monitoring hooks to the original method and can be removed when information is no longer required. Because this hotswapping capability is lightweight (see for detailed performance measurements), this technique can affordably obtain detailed server information when needed and revert to a low-overhead mode as required by removing the instrumentation bytecode on the fly.

R2—Class-Hierarchy Analysis: A tool based on this technique must start the instrumentation from a set of top-level root methods, corresponding to the component-level methods available in the system. Then, it must collect and total the monitoring information in order to present meaningful data to the user. As detailed in instrumentation starts from the root methods, which in the case of JEE are the methods implementing the business methods in EJBs or the methods serving requests in JSPs and Servlets . Before the root methods call other methods as part of their functionality, instrumentation is injected automatically in all the methods that can be potentially called from the root methods. This is determined using Class-Hierarchy Analysis. This is repetitive for each of the called methods the first time they are called [4].

R3—Coverage Adjustment : Using the hot swapping-based instrumentation, two instrumentation levels can be used and dynamically alternated.

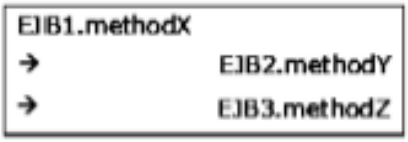

Fig. 1 Top-level CCT.

This type is a high-level low-overhead instrumentation operation across the entire target JEE system. The top-level profiling mode has a significantly lower overhead than the recursive instrumentation mode described below since it offers a system wide shallow profile of the application (in the form of a component CCT). This capability can be used when developers choose to perform a complete top-level profiling operation of the target system and can help in quickly identifying the potential performance hot spots at a coarse grained level. If the set contains an entire JEE application, all the methods corresponding to all components (the EJBs and all the HTTP handlers corresponding to all the Servlets and JSPs) in the application are selected for recursive instrumentation.

R4—Mapping Controller : This section describes the process of mapping the high-level component constructs to the level at which the dynamic BCI operates. Upon selection of different JEE elements for instrumentation, the tool must generate lower level instrumentation events that eventually result in the dynamic BCI code being injected into the appropriate classes. For EJBs, based on their deployment descriptor, the container generates classes implementing the two interfaces (EJB Object and EJB Home) .Clients of an EJB component will work with references of these container-generated objects. After creating or finding an EJB instance using the EJB Home object, clients will call methods on the EJB Object implementation, which ensures that the required services are provided for the calling context, before dispatching the call to the actual bean class instance.

Location for the instrumentation bytecode because its methods wrap the bean-class implementation methods with the required services. For each method method X from the bean class, there is a corresponding method method X in the EJB Object implementation. The latter will contain calls to different container services in addition to the call to the bean class's method X. This applies in general to most JEE application servers. If the technique is used in an environment where the container-generated classes are not known (i.e., with an unsupported application server), the only classes that can be instrumented are the Servlets and the EJB bean classes. This results in the JSPs and the EJB (system level) container services not being instrumented, which is similar to the default portable (application-level) instrumentation available in tools. To address this issue, instrumentation tools based on this technique should expose an external API to allow third parties to develop connectors for other application servers. Using top level instrumentation, the only instrumented methods will be method X, method Y, and method Z. EJB runtime entity can perform poorly because of bad configuration choices for container services (such as security or transactions), bad business logic, or a combination of both.

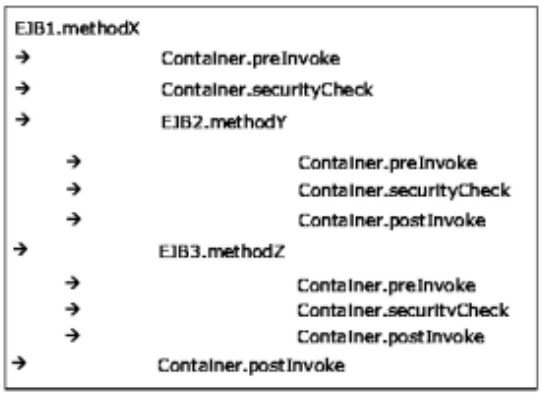

Fig. 3 In-depth CCT.

This technique has been fully implemented in the Sun Java Studio 7 Performance Profiler which is in turn based on the less feature-rich Net beans Profiler. This profiling tool can attach to a running application server, inject instrumentation code in the target components (EJBs and Servlets), and collect and aggregate the JVM performance data corresponding to these entities. Each JEE element has an associated performance data structure, which aggregates the performance results (average execution time, the number of invocations, and percentage of execution time) for its contained elements. An EJB, for instance, will show the percentage of time exhausted in all of its methods. When searching for the root cause of a performance problem observed at the component level, a deeper understanding of the call patterns that comprise the context of the performance problem may be useful. Considering the top-level sample CCT in Fig. 4, its corresponding recursive instrumentation CCT would contain the elements presented in the CCT in Fig. 5 (for a simplified hypothetical scenario). In real scenarios, the CCT in Fig.5 and 6, can be more complex, as each of the container services can have a complex calling tree associated. Gdownloader is a sample application which can used to download any application from the internet and that application can be analyzed later as summarized in the figures 7, 8 and 9.

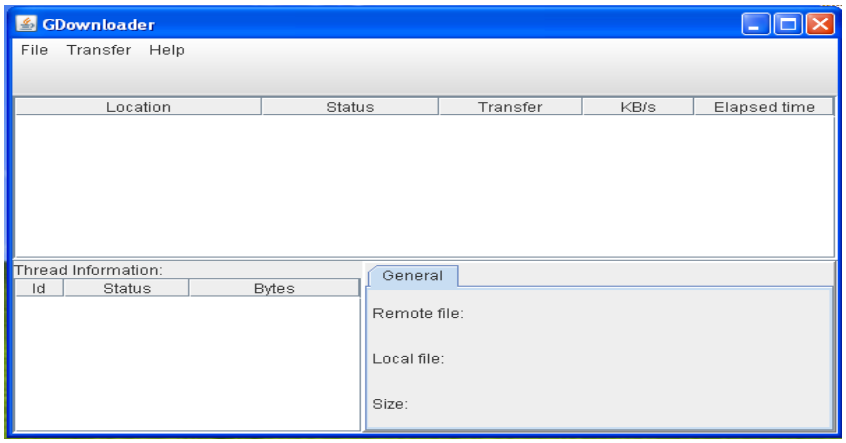

Fig. 4 A Sample application to download any java application.

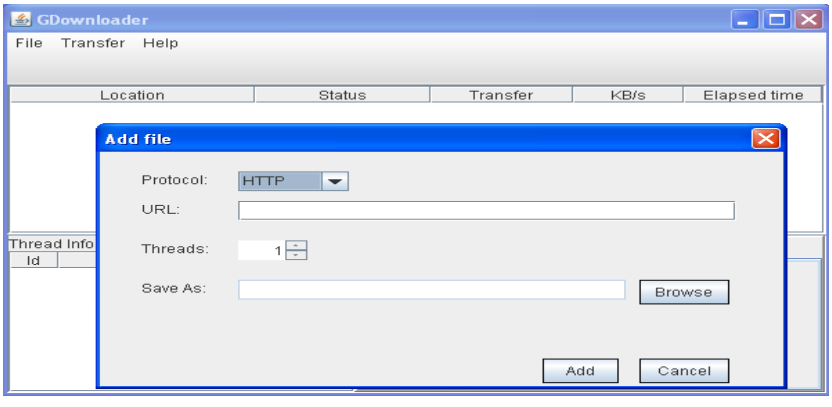

Fig. 5 Application downloading with no.of threads.

The Figure 5 dipicts the process of downloading any application from the internet. Here we should specify the Protocol, URL of the application, no.of threads to split the application for easy downloading and we can store the application at our desired place in our machine.

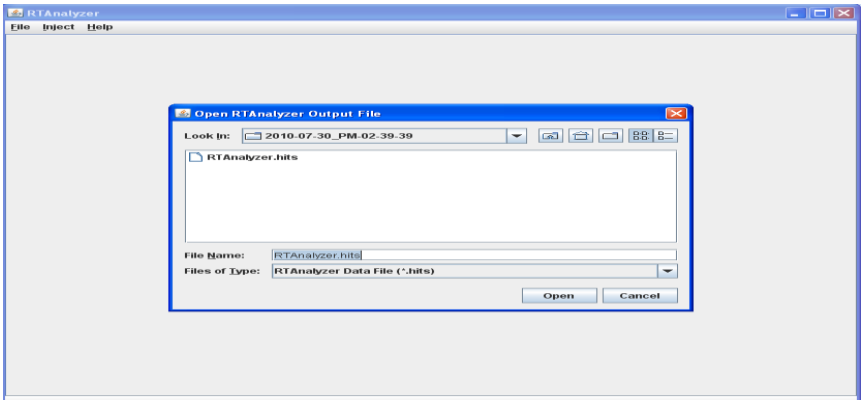

Fig. 6 Selection of output report to analyze the data.

After downloading the required application using gdownloader, we can select that particular application to analyse the component interactions.

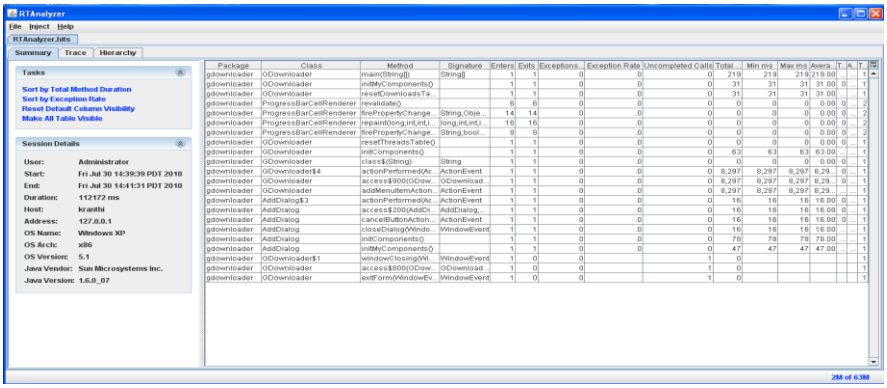

Fig. 7 Summary of the components in the gdownloader application.

The figure 7 shows the summary of the components which are available in the gdownloader application like packages, classes methods. Signatures etc.

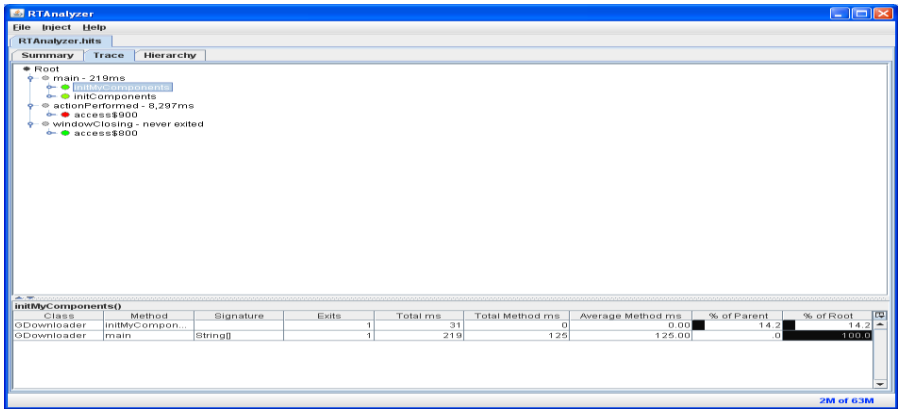

Fig. 8 Tracing of a particular component.

The figure 8 show the RT Analyzer which can show all the components in a root and tree manner so that it can easy to trace a particular component in the application.

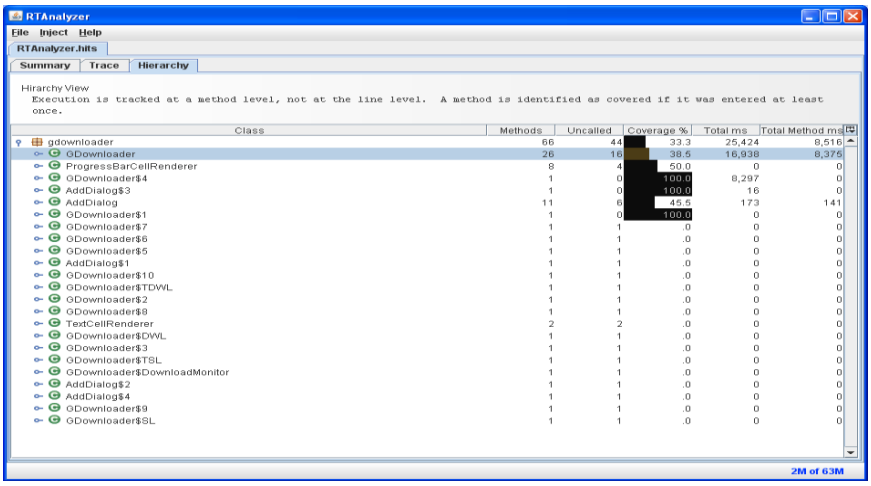

Fig. 9 Hierarchy of the components in gdownloader.

IV. CONCLUSION

Component-based frameworks are largely used for the development of large complex enterprise systems. Due to their complexity, understanding these systems can be difficult. Thus, the availability of CLIs is essential for analyzing, understanding, and monitoring component-based systems. The first CIE approach presented (assisted recording) can produce runtime values for CLIs associated with user defined business cases. It omits middleware calls capturing inter-component communications at the application component level; however, due to its uni-intrusiveness, it requires that only one thread be active at a single time. Overcoming the shortcoming of the assisted recording approach, the automated interaction extraction approach abolish the one-thread requirement, at the cost of increased intrusiveness (an identifier for each interaction needs to be associated with a thread). It is found based on this study that the performance of enterprise systems however, is influenced by both the application-level component interactions and the interactions among the application components and the middleware components. This paper has analysed through implementation the assisted and automated application-level CIE approaches collect performance data as an aggregate of component and platform contributions, the application- and system-level interaction extraction approach can distinguish between the two. This adds accuracy in detecting and solving performance problems; however, it comes at the cost of more specific and possibly restrictive runtime environment requirements.

REFERENCES

[1] L.C. Briand,Y. Labiche,and J. Leduc,"Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," IEEE Trans. Software Eng., vol. 32, no.9, Sept.2004.
[2] D. F. Jerding, J. T. Stasko, and T. Ball, "Visualizing Interactions in Program Execution," Proc.19th Int'l Conf. Software Eng., 1997.
[3] S. L. Graham, P. B. Kessler, and M. K. Mc Kusick, "GPROF: A Call Graph Execution Profiler," Proc. SIGPLAN Symp. Compiler Construction, 1982.
[4] M. Chen, E. Kicimasn, A. Accardi, A. Fox, and E. Brewer, "Using Runtime Paths for Macro Analysis," Proc. Ninth Workshop Hot Topics in Operating Systems, 2003.
[5]T. Parsons, "Automatic Detection of Performance Design and Deployment Anti-patterns in Component Based Enterprise Systems," PhD dissertation, Univ. College Dublin, 2007.
[6] G. Ammons, T.Ball, and J.R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," Proc. ACMSIGPLAN Conf. Programming Language Designand Implementation, 1997.
[7] M. Trofinand J. Murphy, "Static Verification of Component Composition in Contextual Composition Frameworks," Int' lJ. Software Tools for Technology Transfer, Jan.2008.
[8] The Netbeans Profiler, http://profiler.netbeans.org/,2008.
[9] P.Tonella and A. Potrich, Reverse Engineering of Object Oriented Code. Springer, 2005.
[10] T.Parsons, A. Mos, and J. Murphy, "Non-Intrusive End to End Run Time Path Tracing for J2EE Systems," IEEProc. Software, Aug. 2006.
[11]T. Gschwind, J. Oberleitner, and M. Pinzger, "Using Run-Time Data for Program Comprehension," Proc.11th Int ' Workshop Program Comprehension, May2003.
[12] M.Chen,E.Kiciman,E.Fratkin,A.Fox,andE.Brewer,"Pinpoint:Problem Determination in Large, Dynamic, Internet Services," Proc.Int'lConf.DependableSystemsandNetworks(IPDSTrack), 2002.
[13] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," Computer, vol.36, no.1, Jan.2003.
[14] IBM Autonomic Computing, http://www.research.ibm.com autonomic,2007.

