Extracting Interactions in Component Based Systems

Trevor Parsons, Adrian Mos, Mircea Trofin, Thomas Gschwind, *Member, IEEE,* and John Murphy, *Senior Member, IEEE*

***Abstract*—Monitoring, analysing and understanding component based enterprise software systems are challenging tasks. These tasks are essential in solving and preventing performance and quality problems. Obtaining component level interactions which 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 industry. They pose specific challenges while raising interesting opportunities for component level interaction extraction tools. We present a range of representative approaches for dynamically obtaining and using component interactions. For each approach we detail the needs it addresses, and the technical requirements for building an implementation of the approach. We also take a critical look at the different available implementations of the various techniques presented. We give performance and functional considerations and contrast them against each other by outlining their relative advantages and disadvantages. Based on this data, developers and system integrators can better understand the current state of the art and the implications of choosing or implementing different dynamic interaction extraction techniques.**

***Index Terms*—Distributed objects, components, containers, tracing.**

I. Introduction

Current enteprise applications are very often large 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 run-time. This can lead to a lack of overall system understanding which in turn can manifest itself in range of different problems (e.g. incorrect performance tuning, maintainability issues etc.).

Component level interactions (CLIs), which capture component communication, can be recorded as the program executes (i.e. at run-time) 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 available to developers whereby they can record both static and dynamic traces [1].

The availability of an application's CLIs or potential interactions is in fact important in many different situations, ranging from optimizing a simple program, to reverse engineering or monitoring an entire application. For example, when performing program optimizations, we need to know at least how frequently a component is invoked and under what circumstances. For reverse engineering, it is imperative to know which component can be invoked from which other component. It is also often the case that this information would enhance data already provided by certain tools. For example, very often performance profilers collect details on the different components that make up a system (e.g. resource consumption information). However it is common that no contextual indication is given as to how the components work together to service the different user requests.

This paper covers dynamic techniques for collecting component interactions. It presents a number of different approaches for capturing CLIs, discussing when each one is most appropriate while at the same time giving the advantages and disadvantages of each. We believe the presented options cover the full range of the current state of the art for such techniques.

The remainder of the paper is structured as follows: Section II details applications of CLIs and a number of different ways in which CLIs can be represented discussing the advantages and disadvantages of each representation. A prerequisite for dynamic interaction extraction is the ability to record inter component calls at run-time. This is achieved through instrumentation of the application and is discussed in section III. The following sections IV, V and VI outline the various options available for dynamic component level interaction extraction (CIE). For each technique presented we outline the various requirements that are needed in order to implement the approach for the Java component technology, the Java Enterprise Edition (JEE), which is widely used by the industry. We also give the advantages and disadvantages of the approach, outline when its application would be most suitable, and discuss available implementations and related work. Specifically, section IV outlines an assisted interaction recording approach, section V discusses how interaction recording can be achieved automatically in a multi user environment and section VI outlines another approach for CIE that can automatically adjust its coverage to include container level information. All the techniques presented have varying degrees of portability across different vendor implementations of the JEE technology and none of them are specific to a particular vendor. Section VII

T. Parsons and J. Murphy are with the School of Computer Science and Informatics in University College Dublin, Ireland. A. Mos is with INRIA, in Rhone-Alpes, France. M. Trofin is with Microsoft, in Redmond, WA, USA and T. Gschwind is with IBM in Zurich, Switzerland.
E-mail: trevor.parsons@ucd.ie, adrian.mos@inria.fr, mtrofin@acm.org, THG@zurich.ibm.com and j.murphy@ucd.ie.

Manuscript received February 14, 2007; revised February 27, 2008 and June 12, 2008, accepted for publication June 26, 2008.

evaluates each of the different approaches in terms of the information they produce and their performance overhead. Section VIII gives our conclusions.

II. Component Interaction Representations and their Applications

In this section we discuss the different ways in which CLIs can be represented and the different levels at which this information can be recorded, i.e. application and system level. We also outline our motivation for this work and discuss why it is important to be able to collect the different component interaction representations by giving examples of the areas in which the information can be used.

CLIs can be represented in numerous ways. Typical examples include call traces [2], call graphs [3], run-time paths [4] [5] and calling context trees (CCT) [6]. The different representations contain different levels of information, have different space requirements and as such are suitable for different tasks. The authors Jerding et al. [2] have previously discussed a spectrum of representations that can be used for the purpose of tracing system execution. They discuss representations ranging from the most accurate and space inefficient, to the least accurate and most efficient. Dynamic call traces are said to be most accurate representation, while at the same time being the most space inefficient. Call graphs in contrast are the least accurate representation, but the most space efficient [2] [6].

A dynamic call trace is a recording of all calls and returns that occur in a program's execution. They capture important information such as the sequence of calls, the context in which a call was made, and repeating calls or loops. However extracting the call trace may not be feasible for long running programs since they are unbounded and their size is proportional to the number of calls in an execution [6]. A run-time path is a particular kind of dynamic call trace [5]. They are generally used in the context of multi-user/multi-client distributed applications. Run-time paths contain the same information as dynamic call traces, i.e. they contain the ordered sequence of all calls and returns that occur during execution. However they also contain related resource and performance information for each call. Furthermore run-time paths are generally associated with multi-user/multi-client systems. They group together, into a single path, the sequence of calls that correspond to a given user/client request. As such, run-time paths isolate the calls that occur in response to a given request. This is particularly important in multi-user/multi-client applications whereby many simultaneous requests may be concurrently serviced. Since run-time paths are associated with multi-user applications, they also often span a number of physically distributed system layers.

Representing CLIs as call graphs is much more space efficient than using dynamic traces or run-time paths. A call graph is a compact representation of calling behavior. While they are bounded by the size of the program, there is much interesting information about calling behavior that is dropped to gain compactness (e.g. sequencing of calls, call context). Call graphs are said to be context insensitive while call traces are context sensitive [6]. Context sensitive profiling associates a metric with a sequence of executed calls. This can be used to separate measurements from different invocations of components [6]. Profiling tools that are context insensitive can only approximate a program's context dependent behaviour.

While the paper focuses on run-time CLI extraction, it is worth mentioning that static techniques [7] can also be employed for call graph extraction. The main advantage of static call graph extraction over a run-time or dynamic approach, is that all the possible component interactions are extracted. Dynamic CLI extraction on the other hand runs the risk of not collecting all possible execution paths. This is especially the case in applications that are intensively data-driven, where some component interactions occur only under certain input conditions. Furthermore for static extraction the CLI extraction process does not require a running system which in the case of enterprise applications can be a major advantage, since setting up a live analysis testbed can be time consuming and resource intensive. The main disadvantage of static extraction is that very little additional information may be extracted (such as performance information). In terms of call graph representation, the method presented in [7] produces directed graphs from static analysis, where the nodes represent methods. A particular arc indicates that the call from the method represented by the predecessor node, to the method represented by the successor node, may take place at runtime.

A calling context tree is an intermediate representation that is more accurate than a call graph and less inefficient than the dynamic call trace in terms of space requirements. It is a more compact representation of the dynamic call trace and can represent all contexts in a given trace. The calling context tree discards any redundant paths in a trace while preserving unique contexts. Metrics from identical contexts are aggregated in order to improve space efficiency. Calling context trees are context sensitive and thus do not suffer from the same inaccuracies that can occur with call graphs. The breadth of calling context trees is also bounded by the number of methods or procedures in an application. In the absence of recursion the depth of CCTs are also bounded [6].

The component interaction representations discussed above are referred to throughout this paper. Next we discuss why it is important to be able to collect such information by giving examples of the areas in which this information can be used. The list given below, although not exhaustive, covers a wide range of applications of CLIs and serves as motivation for the remainder of the paper.

Optimization: Code and architecture optimization is a common task in software development, where certain sections of the application (e.g. hot-spots) need to be modified to reduce latency or resource consumption such that performance targets are met. To understand the different contexts in which the hot-spots are invoked developers often spend much time analysing the source code of the application. Component interactions extracted from the application however can show exactly how the hot-spot is invoked (or how it could potentially be invoked) saving much time on behalf of the developer e.g. [8] and [9] augment the performance metrics they record with calling context trees.

Reverse Engineering: Component interactions extracted from systems during development can be used to create up to date documentation that can be used by developers to help understand the system [10] [11] [12]. For example Parsons et al. [11] have shown how JEE applications can be reverse engineered using run-time paths. Similarly Briand et. al [1] have used dynamic traces to extract sequence diagrams from distributed Java systems.

Problem Determination: Component interaction representations have also been used to help assist with problem determination in large enterprise systems. For example, Chen et al. [13] use dynamic tracing (in the form of run-time paths) to capture the dynamic nature of today's distributed systems and to identify problem components in such systems automatically. Trofin et al. [7] show how, by statically extracting and analysing call graphs, run-time error-inducing incompatibilities between components can be automatically and correctly pinpointed.

Autonomic Management: Recently there has been a major initiative in developing autonomic management systems based on automated monitoring and diagnosis techniques [14] [15] [16]. A system with autonomic capabilities requires a representation of the components that make up the system, and also a representation of how these components interact to service the different requests. Component interactions extracted from the system can be used as the basis of such representations. For example, component interactions collected at run-time can be used to drive a self adapting monitoring and diagnosis approach [17].

Redundant Service Removal: Platforms for component based enterprise applications, such as those built on EJB, execute services before and after a call is dispatched to a particular component method. Depending on the actual call path through the component application, some of these services may be redundant, and not need execution. Optimizing such applications by removing redundant service calls (e.g. [18] and [19]) requires the availability of inter-component call graphs, either extracted at run-time or statically.

For component based applications all representations discussed above can contain application and system level interactions. Application level interactions omit lower level middleware calls that take place during program execution. In fact, generally they only contain calls from the application code, as designed by the system developer(s) when implementing the business functionality. System-level interactions on the other hand represent the interactions that occur between the application components and the middleware components (e.g. application components invoking container services) and the interactions that occur solely between the different middleware components (e.g. calls from one container service to another). In some cases it can be desirable to collect application level information only (e.g. for reverse engineering [11]), where as in other cases both system and application level information may be required to solve a particular problem (e.g. optimizing the use of middleware services [20]).

III. Instrumentation

To extract component interactions at run-time, the application needs to be inspected in some way. Dynamically, CIE can be achieved by instrumenting the application such that actual CLIs can be recorded. In the following subsections we outline the different options that are available for the instrumentation of JEE component based applications.

A. 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. Probes can be generated and inserted automatically by leaveraging metadata available for JEE components. During the instrumentation process the original application structure, called the Enterprise Archive in JEE (EAR) is analysed and the component meta-data is extracted (including component name, 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 meta-data in order to accommodate the insertion of probes. At run-time the probes intercept calls to and from the JEE components. The process is explained in detail in work by Parsons, Mos and Murphy [11] [17], 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 [11]. One of the main advantages of this technique is that it makes use of standard JEE mechanisms and does not require any particular JVM or middleware facilities. Thus this technique is portable across the different vendors' middleware and JVM implementations and can be used in truly heterogeneous environments [11]. The approach is also non-intrusive, insofar as no modification to the application source code or bytecode is required. This is particularly important for component based systems where components may be developed by third parties and licensing constraints may prohibit their modification. 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 [21]), however such techniques are not portable across different middleware implementations.

B. Aspect Oriented Programming

System tracing can be seen as a requirement cross-cutting the application architecture, and is a natural candidate for

Aspect Oriented Programming (AOP) [22]. With AOP, the programmer can specify in an AOP language how and where to place the instrumentation, while the AOP language compiler/run-time takes care of its deployment. Using AOP libraries developers can weave tracing aspects into their JEE applications. The tracing aspects perform the same functionality as the probes mentioned above, i.e. capturing requests to and from the JEE components. One such library for the Java platform is AspectJ [23]. AspectJ weaves aspects into Java applications by modifying the application bytecode. AspectJ allows for compile time weaving, post-compile time weaving and load time weaving of aspects into a Java application. Compile time weaving requires the source code of the application to be available. Compile-time weaving is undesirable for component based applications since source code may not always be available for all application components. In this context post-compile time weaving is a better approach. Post-compile time weaving, also known as binary weaving, allows for aspects to be woven with binary components, thus in this instance there is no need for the components' source code to be available. Load time weaving is another technique that is available with AspectJ. It allows for the already compiled code to be instrumented as the system loads and avoids the need for an extra "instrumentation" step in the build process as required by the other weaving approaches and the proxy based approach above. Load time weaving however requires JVMTI [24] support which is only available as standard in Java 1.5+ compatible JVMs, which may not be applicable for many legacy systems. The main advantages of the AOP approach is that in comparison to the proxy approach it does not require redeployment (in the case of load time weaving). Also with the AOP approach aspects can be attached to the middleware components as well as the application components. Thus application and system level information can be obtained using this approach. Govindraj et al. [25] have previously show how the InfraRed performance monitoring tool has made use of such AOP techniques to monitor JEE component based applications.

C. Bytecode Instrumentation

Bytecode Instrumentation (BCI) is a technique widely used by profiling tools to instrument Java applications. It allows for the modification of the compiled Java code (bytecode) and can be performed statically or dynamically. When working at the bytecode level, source code is not required and thus is suitable for component based systems. AOP libraries such as AspectJ discussed above, generally use BCI to weave aspects into an application. However working at the bytecode level requires a high level of expertise. One of the advantages of using an AOP approach is that it can hide the complexities of directly modifying the application bytecode. As an alternative to using AOP a number of bytecode manipulation tools exists to ease the burden of having to modify the bytecode directly (e.g. [26], [27], [28], [29], [30]). These tools hide much of the details of the (bytecode) class file format and provide APIs for bytecode manipulation.

Static BCI is the modification of bytecode before the code is executed. Its main advantage (over dynamic BCI) is that there is less run-time overhead since there is no instrumentation performed during execution. However, a drawback of this approach is that dynamically loaded or generated classes will not be exposed to the instrumentation process. In contrast dynamic BCI enables the modification of an application's bytecode as the system executes. It allows for instrumentation to be removed or modified after the initial instrumentation process and can be achieved without the need to restart the application. While this approach introduces extra run-time overhead (as a result of the run-time instrumentation process) it ensures that all components can be instrumented and it avoids the need for an extra "instrumentation step" in the build process.

Since Java 1.5 there has been standard support for dynamic BCI through the Java Virtual Machine Tools Interface (JVMTI) [24] and the java.lang.instrument package [31]. Both approaches use a profiling agent, (i.e. a pluggable library that runs embedded in a JVM) to intercept the classloading process. The agents allow for the redefinition of the class bytecode either at load time or as the system executes. The JVMTI agent is written in native code and while it is portable across different JVMs it is not portable across different platforms. The java.lang.instrument agent however is a Java agent and is thus completely portable.

As with the AOP (load-time weaving) approach, redeployment is not required when using BCI. Also using BCI for component interaction extraction allows for both application and system level information to be collected. However it also may not be applicable in situations where licensing constraints prevent code modifications. Furthermore dynamic BCI is only available as a standard feature for applications running on Java 1.5+ JVMs. The netbeans profiler [9] is an example of a profiling tool that currently makes use of dynamic BCI to instrument JEE components.

IV. Assisted Interaction Recording

The first interaction extraction technique we are presenting involves a hybrid approach consisting of both manual and automatic operations. The human operator drives the interaction extraction process by going through the front-end steps required for a business functionality and the interaction recording system monitors the component calls and constructs a representation of the resulting interaction. This representation omits lower level middleware calls that take place during program execution and contains application level component information only.

A. 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

- 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 timestamps and method IDs used in interaction reconstruction. 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 run-time path). An important advantage of this technique is that it is inherently distributed as it captures events using standard JEE technologies for distributed computing. 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 multi-user situations).

B. R1 - An Instrumentation Framework

A number of portable instrumentation techniques have been previously discussed in section III. For application-level CIE either the proxy based approach (section III-A), the AOP approach (section III-B) or the BCI approach (section III-C) 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). We refer to these as the method entry instrumentation point (IP) and the method exit IP.

C. R2 - Recording Entity

At run-time the IPs collect performance information obtained from their target application components and communicate with a centralised unit via standard Java/JEE mechanisms (such as Java Management Extensions - JMX [32]). This centralised 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. 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. During training sessions, developers record the required scenarios (such as “buy a product” in a typical e-commerce application) by going through the required steps in the system while the recording mechanism obtains and stores the generated data.

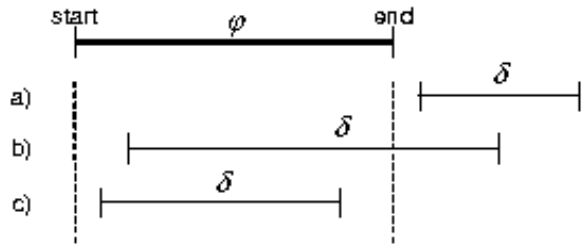

Fig. 1. Enclosing Methods

D. 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 timestamps (i.e. methods that started execution earlier are placed at the beginning of the data set).
(2) Parsing the method invocations data set for each method δ, the list of methods preceding it in the sorted data set is analysed to find a possible enclosing method. As illustrated in Figure 1, method φ encloses method δ only in situation c) where the start and end time of method δ are included in the interval created by the start and end time of method φ.
(3) If an enclosing method φ is found that satisfies the case presented in Figure 1 c), then method δ is added as a child to method φ in the interaction tree that represents the recorded interaction model.

Since this technique relies on time stamps to extract and order the recorded interactions, without knowledge of the point of entry in the interaction, it is vital that the time stamps be as accurate as possible. For methods with considerable execution times (in the order of hundreds of milliseconds or more), time measuring imprecision is not an issue as containment relationships can be observed easily. For methods with small execution times, however, the method duration might be reported as 0 ms because of the inherently imprecise Java time API [33]. When more methods report a duration of zero and an identical starting time, it is impossible to determine their sequence precisely. To overcome these clock synchronisation and precision issues in multi-node heterogeneous environments or even on locally deployed applications (for methods that appear to start at the same time or to have 0 length due to imprecise APIs or system calls), the recording process should be capable of instructing the IPs to induce a custom artificial delay into the target methods, thus guaranteeing the order of the received events.

E. Implementations and Related Work

To the best of our knowledge, the COMPAS JEE [34], an open-source monitoring framework for enterprise applications has the only implementation of this technique as part of its Interaction Recorder (IR). The COMPAS IR extracts EJB interactions from systems instrumented using this approach in a completely non-intrusive manner and generates CLIs containing performance metrics which can be saved for further processing or visualised in automatically-generated UML diagrams. The IR provides a VCR-like GUI for starting and stopping recording sessions. The GUI allows for the visualisation of the extracted interactions in either tree-format or UML. An interaction tree contains the same information as a run-time path (see section II).

Although we are not aware of any other tool implementing this technique for JEE systems, the method of recording interactions has been used in other contexts. Most notably, web-site interaction recording is used by load-generation tools that need to emulate multi-user accesses to web-sites with the purpose of stress testing them (e.g. [35] or [36]). These tools generally employ a script recorder in order to facilitate the creation of execution tests. By using an HTTP proxy server that runs on a local machine and diverting the browser traffic through this proxy server, these tools are able to capture HTTP interactions (web-page accesses, image loading, GET and POST calls) and create scripts that emulate what the user has performed in the browser. These scripts can then be enriched with other instructions and played multiple times using defined virtual user sequences. The similarity with the COMPAS Interaction Recorder stems from the way the proxy server is used and "inserted" into the user web-experience, as well as from how the information (HTTP events) is collected and ordered using time-stamps.

V. Automated Application Level Interaction Recording

An alternative approach to recording interactions for application level trace extraction is discussed in this section. This section addresses some of the disadvantages of an assisted approach (section IV) and introduces techniques that can be applied for automated application-level CIE.

A. Method Overview

In some situations, a more automated approach than that discussed in section IV is desirable. 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 multi-user systems. The problem with the assisted approach is that when simultaneous users are in the system, one user request can not be distinguished from another. An automated tracing approach for multi user systems basically works by identifying new user requests that enter the system, tagging the requests such that each request can be uniquely 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 [11]. The requirements needed to achieve this are as follows [11]:

- R1 - An instrumentation framework
- R2 - New user requests must be identified when they enter the system
- R3 - Each different user requests must be tagged with request specific information (RSI)
- R4 - The RSI must be accessible across the entire request

In the following paragraphs we discuss the different ways in which the requirements R1-R4 can be achieved in a portable manner outlining the advantages and disadvantages of each technique. The monitoring framework requirement (R1) is met in the same as for the assisted CIE approach (see section IV-B).

B. R2 - Non Intrusive Detection of New User Requests

In order to be able to tag new user requests with RSI (R3) it is important to be able to identify new requests when they first enter the system. This can be achieved intrusively by instrumenting the middleware [11] [13]. To achieve this in a non-intrusive manner, such that the approach is portable across different middleware implementations, manipulation of the middleware must be avoided. To overcome this a non-intrusive 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. This works by monitoring the depth 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 run-time 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 [13]. The call depth data is stored as part of the RSI.

C. R3 - 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 and (3) the depth of the call path is recorded for point of entry detection (discussed above). The RSI contains a unique ID assigned to each new request as well as call path depth data and call path sequencing data. The unique id is used to identify the different calls that make up each request. The function of the call path depth data is discussed in section V-B. 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 set 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 construct an ordered representation of the calls that make up the user request. The RSI can be attached to the request using the Java ThreadLocal object. Every Java thread

has a ThreadLocal object associated with it (since Java 1.2) which can be used to store an object for the lifetime of the thread. One limitation of the ThreadLocal object (discussed in section V-D) is that the information stored in the object is not propagated to remote method calls. A solution to this problem is discussed in section V-E. An alternative approach to the ThreadLocal object might be to use the Work Area Service (Java Specification Request, JSR 149 [37]). The Work Area Service also allows for the tagging of user requests with contextual information. The information is attached to each thread as with the ThreadLocal approach but has the advantage that the information is also propagated across remote method calls. However the JSR 149 was withdrawn and the Work Area Service has not been made standard. While IBM [38] have implemented this feature as part of their application server implementation, using a work area service would render any approach non-portable as it is not a standard feature.

D. R4 - Accessing the RSI Throughout the Request

Tagging threads with RSI enables the system to distinguish 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 ThreadLocal approach is that it is limited to tracing requests on systems where web and application servers are co-located. Servers that are co-located 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. 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.

E. R4 - Sharing the RSI Across JVM Boundaries

In this section, we outline how calls from a client to a server can be tracked in a distributed setting. The approach we discuss is based on piggy-backing the necessary correlation data (RSI) onto the request from a client to a remote server. In the following subsections we detail how this can been achieved in a portable manner for JEE systems.

1) Piggy-backing data: There is no standard mechanism in JEE for piggy-backing data along a remote request. Piggy backing mechanisms however are available in other technologies. For example CORBA's Portable Interceptors allow for piggy-backing of data along a request and can be used to track user requests in distributed environments. In fact, it has previously been shown that they can be used to attach additional security information onto requests [39]. We discuss how this problem can be solved for JEE systems by adding an additional parameter to the remote request and allowing the client to pass the extra request data through that additional parameter. This can be achieved without access to the application's source code as follows: A components' metadata specifies the component interfaces. The interfaces can be reconstructed and rewritten from their bytecode representation (or by analysing the XML deployment descriptor) to add a method that allows for the passing of the additional information using an additional request parameter. Next the client component is also instrumented, to add this additional parameter, and the server component instrumented, to pass it to the instrumentation infrastructure. In the next section, we give a detailed description of how this technique can be used to pass RSI across distributed component boundaries.

2) Client Side Interception: When a client component calls a distributed component in JEE, the following occurs: The client calls a stub which is a client side proxy object. The stub masks the low level networking issues from the client and forwards the request on to a server side proxy object. The server side proxy element, known as a skeleton, masks the low level networking issues from the distributed component. It also delegates the remote request to the distributed component. The distributed component then handles the request and returns control to the skeleton, which in turn returns control to the stub. The stub, in turn, hands back to the client.

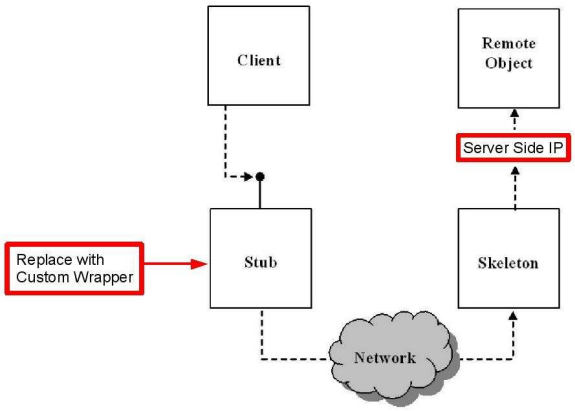

Fig. 2. Remote Method Invocation with Client and Server Side Interception

For client side interception an IP is responsible for adding an additional parameter to the remote request. Adding a client side IP can be accomplished by replacing the client stub used by the application with a custom wrapper (see figure 2). This custom wrapper provides the functionality to intercept the client's request invocation before invoking the server side EJB. The custom wrapper performs the necessary task of extracting the RSI from the ThreadLocal object (or adding it if it is the point of entry into the system) and sending this information as an extra parameter in the remote call.

In order for the custom wrapper to be able to intercept the client side calls it must replace the stubs generated by the Java Virtual Machine. With JEE, the client obtains a reference to a remote component using the Java Naming Directory Interface (JNDI) naming service and obtains the stub by casting the reference to the component's interface from a generic type to an RMI-IIOP interface type using the PortableRemoteObject.narrow method (see Listing 3).

The JVM generated stub can be replaced with the wrapper by using a custom PortableRemoteObjectDelegate object. This is supported through a standard mechanism in the JEE spec-

```
Properties props=System.getProperties();
Context ctx=new InitialContext(props);
Object obj=ctx.lookup("Sample");
SampleHome sh =(SampleHome)
  PortableRemoteObject.narrow
    (obj,SampleHome.class);

Sample s = sh.create();
System.out.println
    (s.callBusinessMethod("SomeString"));
s.remove();
```

Fig. 3. Sample Client Code

ification and can be configured as part of the Java run-time environment. Thus this approach is portable across different application server implementations.

3) Server Side Interception: On the server side the IP is in the form of a proxy layer (or wrapper) as described in section III-A. Using this approach all calls to and from the bean can be intercepted (see figure 2). 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 run-time path tracing once the data has been sent across the network it can be attached to the current thread as outlined in section V-C.

F. 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 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. 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, run-time paths, calling context trees or even as call graphs (see section II).

G. Implementations and Related Work

The pinpoint problem determination tool [13] has the ability to perform this interaction extraction for JEE systems. It makes use of the ThreadLocal approach outlined in section V-C to satisfy R3 (Tagging user requests with RSI). However its main drawback 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 run-time paths. The COMPAS JEEM monitoring tool [11] also collects application level run-time paths for JEE applications. It makes use of pinpoint libraries to satisfy R3. It satisfies the requirements R1, R2 and R4 as outlined in sections IV-B, V-B and V-E respectively. Thus it is completely portable across different middleware implementations and can not be applied to components that communicate across physically distributed JVMs. The WebMon[1] monitoring tool [40] overcomes this issue by applying the technique outlined in section V-E. Another tool which has the ability to collect application (and system) level interactions for JEE applications is the InfraRed [25] monitoring tool. Infrared instruments the application using AOP (see sectionIII-B). It also uses ThreadLocal to tag user requests with RSI (R3). Similar to the approach outlined in section V-B the InfraRed tool uses a depth counter stored within the RSI to maintain the level of calls within particular application tiers (or layers, e.g web tier, business tier, database tier). To identify a request entering the system (R2) the InfraRed developers suggest also using a ThreadLocal approach together with a specially designed aspect or a filter servlet [25]. However, the authors suggest an alternative approach to remote call tracking (R4) [25], than has been suggested in section V-E. 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 InfraRed tool produces calling context trees as output for the CLIs it records. Briand et al. have presented an approach for reverse engineering sequence diagrams from multi-threaded [41] and distributed Java applications [1]. They also record traces using an AOP monitoring approach. To distinguish between local threads they make use of a threadID identifier, however this information is not stored in a ThreadLocal object. Instead a list of threadIDs are maintained in a text file. To allow for distributed tracing they marshall data across the network using aspects. Similar to our approach they intercept remote calls such that information can be sent across the network in order to identify the client method that called a particular remote method. 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 (clientNodeID), the thread identifier (clientThreadID), the class name (clientClassName) and the object identifier (clientObjectID). However the approach essentially works in the same way since their information is used to be able to identify the client thread responsible for remote communication. Their approach has not been applied to JEE applications. The magpie project [42] also monitors CLIs across distributed systems. The initial approach [42] uses a unique identifier to identify requests and propagates this identifier from one component to the next. A later version [43] avoids the need for global identifiers and instead correlates component events based on an application specific event schema. Both versions make use of kernel, middleware and application level instrumentation. The magpie project has been implemented for Microsoft technologies and is therefore not applicable for JEE systems.

VI. Application and System Level Interaction Extraction

This section presents a CIE technique that has the ability to capture both application and system (container) level

[1]The technique described in section V-E has been implemented as part of the webmon tool

calls in multi user systems and present them in the form of calling context trees (CCT). In addition, this technique can dynamically adjust the instrumentation coverage thereby optimising its overhead to correspond to the desired level of detail for the interaction data. While more lower level details are extracted with this method, the CCTs cannot distinguish between different user identities and therefore present an aggregated view of the interactions.

A. Method Overview

This method can be used in multi-user environments and it is ideal for situations where more detail about a run-time problem is required in order to determine its cause. Since it can be used to extract interactions with container-level details associated with application method calls, it allows the separation of configuration issues (reflected in container calls) from the actual business logic. 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 component level interactions

The hotswapping capability is required to ensure that a minimal overhead is achieved even as detailed information about component calls is obtained. The class-hierarchy analysis is used in determining the instrumentation targets for specified entry points into the system and the coverage adjustment technology is required for adjusting the monitoring scope. The monitoring scope can be adjusted, depending on the system needs (high-level when no problems exist and detailed when a precise identification of causes is required). Finally, the mapping controller is responsible for matching component level method calls and lifecycle events to low-level virtual machine calls in order to connect business logic to server container logic in the monitoring stages. As this technique can obtain container-level component calls as part of the CLIs, it has an advantage 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 found out with this technique to originate 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 (similar to the previous CIE techniques) and the detailed monitoring, when this is deemed important. 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 (as discussed in section II), mainly that it cannot distinguish between different user identities. This makes it difficult to identify the causes of problems when they occur in isolated scenarios (i.e. a rare type of interaction).

B. R1 - Monitoring with Hotswapping 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 bytecode instrumentation (BCI) provides the basic capability to inject monitoring code in application bytecode during execution (see section III-C). Each method called directly or indirectly (via other methods) by a predetermined root method, will 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 light-weight (see [44] 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.

C. 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 aggregate the monitoring information in order to present meaningful data to the user. As detailed in [44], 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 (which can be easily determined by analysing the EAR for EJBs as described in section III-A or following standard notations for Servlet service methods in JEE; for JSPs, the name of the service methods depends on the target application server). 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 (see [44]). This is repeated for each of the called methods, the first time they are called.

D. R3 - Coverage Adjustment

Using the hotswapping-based instrumentation, two instrumentation levels can be used and dynamically alternated: Complete top-level instrumentation and Partial in-depth instrumentation.

Complete top-level instrumentation: 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. Developers can choose to view performance metrics associated with JEE components at different levels in the CCT hierarchy (e.g. component method, all component methods or all application components).

Partial in-depth instrumentation: When a set of hot-spots has been identified, developers can choose to initiate the recursive instrumentation process for all the methods contained in the set. For example, if the set contains an EJB component, all its business methods are selected for recursive instrumentation. 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.

E. R4 - Mapping Controller

This subsection describes the process of mapping the high-level component constructs to the level at which the dynamic bytecode instrumentation operates. Upon selection of different JEE elements for instrumentation, the tool must generate lower level instrumentation events that eventually result in the dynamic bytecode instrumentation code being injected into the appropriate classes. For instance, when instrumentation is required for Java Servlets [45], the intent is translated into an instrumentation event for the corresponding doGet or doPost handler of the Servlet. For JSPs [45], instrumentation is performed for the container-generated class which will implement the JSP code. As an example, for the Glassfish open-source application server which has been used in the evaluation section (see section VII), this class is *HttpJspBase* and the implementing method is *_jspService*. For EJBs, based on their deployment descriptor, the container generates classes implementing the two interfaces (EJB Object and EJB Home) [45]. 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.

When selecting EJBs for instrumentation at the JEE level, a tool based on this technique must map such actions to instrumentation events for the appropriate class of the EJB. The EJB Object implementation class is the appropriate location for the instrumentation bytecode because its methods wrap the bean-class implementation methods with the required services. For each method *methodX* from the bean class, there is a corresponding method *methodX* in the EJB Object implementation. The latter will contain calls to different container services in addition to the call to the bean class's *methodX*. 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 discussed in sections IV and V. 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. The connectors would consist of identifiers for sets of classes and methods corresponding to container services, in a standard format, as required by the tool.

In **top-level instrumentation**, only the EJB, Servlet and JSP methods (or container-generated artifact methods where appropriate) selected for instrumentation (the instrumentation roots) are instrumented. Let us consider the sample call-graph in figure 4 representing an EJB business method calling two other EJB business methods. Using top-level instrumentation, the only instrumented methods will be *methodX*, *methodY* and *methodZ*.

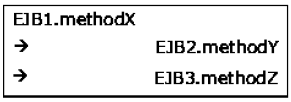

Fig. 4. Top-Level CCT

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. Therefore, observing the detailed call trees of each of the root methods can be particularly useful. These detailed call trees can contain methods that make calls to the container-generated artefacts. These artefacts have complex infrastructure logic that augments the business logic of the bean class. In this case, the **recursive instrumentation** technique presented in detail in [44] is used. Considering the top-level sample CCT in figure 4, its corresponding recursive instrumentation CCT would contain the elements presented in the CCT from figure 5 (for a simplified hypothetical scenario). In real scenarios, the CCT in figure 5 can be more complex, as each of the container services can have a complex calling tree associated. The display of such CCTs can reduce the time needed to understand the origin of a performance issue: an EJB run-time entity can perform poorly because of bad configuration choices for container services (such as security or transactions) or because of bad business logic, or a combination of both.

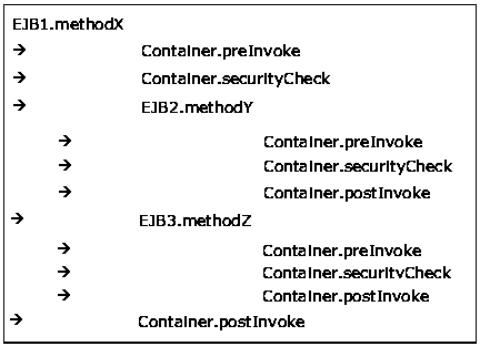

Fig. 5. In-depth CCT

F. Implementations and Related Work

This technique has been fully implemented[2] in the Sun Java Studio 7 Performance Profiler [46] which is in turn based on the less feature-rich Netbeans Profiler [9]. This profiling tool can attach to a running application server inject instrumentation code in the target components (EJBs, Servlets), 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, number of invocations, percentage of execution time) for its contained elements. An EJB for instance will show the percentage of time spent in all of its methods. A JEE application will present a percentage that includes all the percentages of is Web and EJB modules which in turn contain all the percentages of their JSPs, Servlets and EJBs. Many of the current application performance monitoring tools give a similar break down of where time is spent amongst the different JEE application components [25] [47] [48].

VII. Evaluation

In this section we evaluate the different CIE techniques that have been discussed. Firstly we present three tools that implement the techniques presented. In doing so, we show the output that is produced by the different tools when applied to a representative ecommerce application, and discuss how this output can be applied to different tasks. Subsequently, we give details on the performance considerations that developers should be aware of when applying these techniques. We present results from a number of performance tests, that show the overhead/run-time of the approaches when applied to different JEE applications. Essentially, through our evaluation we aim to show in what circumstances each approach might be suitably applied.

A. Evaluation of Tool Outputs

In this subsection we introduce three tools that implement the respective approaches. Each tool is an open source implementation or is freely available. The JEE application used for testing throughout this section is introduced in detail in section VII-A1.

1) COMPAS Interaction Recorder: The COMPAS Interaction Recorder (IR) [17] implements the assisted interaction recording approach outlined in section IV. The tool can be used by a developer to manually record CLIs. The order of the interactions are maintained and presented in the form of a calling context tree augmented with performance metrics. The IR collects both memory and method run-time related information. The tool output can be utilised to analyse system design during development (especially from a performance perspective). Next we use this tool to assist us in introducing the JEE application that is used as a running example throughout this section.

[2]Based on work performed during Adrian Mos's internship in Sun Microsystems Laboratories under Mikhail Dmitriev's supervision. As part of the author's internship work on the JFluid Project, monitoring tool was developed that used dynamic BCI to transparently inject monitoring code into JEE applications running on the Sun Java System Application Server.

The Petstore application is a sample e-commerce application developed by Sun Microsystems [49]. It has been used by Sun to showcase new technologies and design principles. The application is made up of a number of functional units including the Petstore shopping website, the Petstore administration application, order processing center application and the supplier application. For the purposes of our evaluation we make use of the Petstore shopping web site. The shopping web site is made up of over 20 web components (JSPs and servlets) and 15 business tier components (8 container managed entity beans, 3 stateful session beans and 4 stateless session beans). The web site allows customers to register, log in, and to browse or purchase the different products that are available.

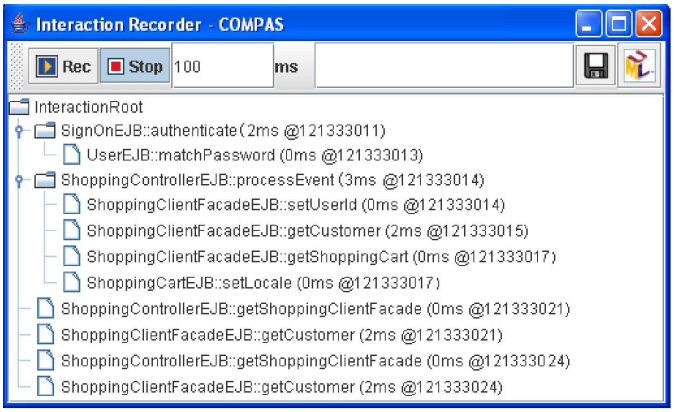

Fig. 6. Interaction Recorder CCT for Petstore "Sign On" Use Case

Figure 6 shows the output produced by the interaction recorder as a result of analysing the Petstore application's "sign on" user action. Using the interaction recorder a developer can quickly understand what components are invoked for a particular interaction and in what order without having to analyse application source code. Since the developer must sit through the recording process, he/she can quickly identify the events that correspond to particular use cases. Figure 6 shows the components that were invoked as a result of the "sign on" user action and the amount of time taken for each method call. It can be clearly seen from the IR output that the time spent in each component method is very small for this user action. It should be noted that the IR can only be used in this manner when there is one user in the system (see section IV). Nevertheless, the output can be used to identify if components are taking a particularly long time or consuming high levels of resources, even when there is only one user in the system. A developer using the tool can identify such issues early, without the need for system load tests.

To further show how the IR can be used to help with system understanding we have reverse engineered the Petstore application with the help of the tool. This was achieved using the IR to record the different application use cases. For the purpose of our evaluation we recorded 9 use cases, i.e. "sign on", "go to store front", "select category", "select item", "get item details", "add to shopping cart", "create order", "send order" and "sign out". By analysing the IR output for each use case it is possible to see the different component dependencies that occur at run-time. Furthermore it is possible to easily

produce a class diagram that gives us an overview of the architecture of the Petstore application's business tier. Figure 7 gives a class diagram that was produced by analysing the IR output for each use case. The entire process (i.e. recording the interactions and constructing the diagram) took less than 20 minutes. The output produced using the IR is particularly suited for helping developers to gain an understanding of their inter-component dependencies since it does not contain any middleware calls. Such diagrams can be particularly useful for analysing the system design in search of design issues or common antipatterns [50].

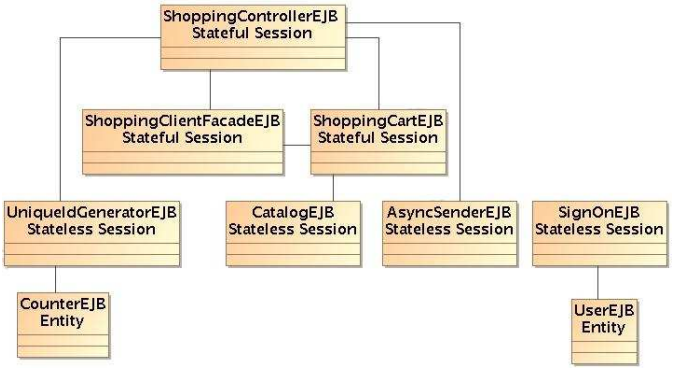

Fig. 7. Petstore EJB Layer Class Diagram

```
Run-Time Path:450
 {type=EJB, name=SignOnEJB, method=authenticate, time=2.36 ms
        {type=EJB, name=UserEJB, method=matchPassword, time= 0.03ms}}

Run-Time Path:451
 {type=EJB, name=ShoppingControllerEJB, method=processEvent, time= 2.58ms
        {type=EJB, name=ShoppingClientFacadeLocalEJB, method=setUserId, time= 0.01ms}
        {type=EJB, name=ShoppingClientFacadeLocalEJB, method=getCustomer, time= 1.36ms}
        {type=EJB, name=ShoppingClientFacadeLocalEJB, method=getShoppingCart, time= 0.01ms}
        {type=EJB, name=ShoppingCartLocalEJB, method=setLocale, time = 0.01ms}}

Run-Time Path:452
 {type=EJB, name=ShoppingControllerEJB, method=getShoppingClientFacade, time = 0.01ms}

Run-Time Path:453
 {type=EJB, name=ShoppingClientFacadeLocalEJB, method=getCustomer, time=2.62ms}

Run-Time Path:454
 {type=EJB, name=ShoppingControllerEJB, method=getShoppingClientFacade, time= 0.01ms}

Run-Time Path:455
 {type=EJB, name=ShoppingClientFacadeLocalEJB, method=getCustomer, time=1.84ms}
```

Fig. 8. Run-time Path for "Sign On" User Action (50 users)

2) COMPAS Java End to End Monitoring Tool: Having the ability to monitor a system with realistic user loads is important as it allows for developers and testers to gain insight into how the system will perform in a production environment. The COMPAS Java End to End Monitoring (JEEM) tool has this capability and is a realisation of the technique outlined in section V. Figure 8 gives an instance of a run-time path which was collected when a load of 50 users were in the Petstore application. Figure 8 shows that for this particular invocation, the time spent in the application level code when 50 users were in the system was still very low.

Because COMPAS JEEM collects application level component interactions only, similar to the IR its output can also be utilised for reverse engineering [51]. In fact the run-time paths produced correspond with UML sequence diagrams and can be easily converted to this diagrammatic format [51]. The information collected by COMPAS JEEM can also be utilised for other tasks such as problem determination [52] or automatic antipattern detection [50]. For example, the literature [5] shows how inefficient component communication can be automatically identified in instances of run-time paths.

3) Netbeans Profiler: The Netbeans profiler provides a basic implementation of the application and system level interaction extraction technique outlined in section VI. Consequently, it also has the ability to collect CLIs under load. However it requires manual configuration of root methods and filter settings to capture application and container level calls for JEE applications. The Sun Java Studio profiler [46] can in fact perform this automatically. We have made use of the Netbeans profiler for our evaluation since it is an open source implementation. Netbeans collects CLIs in the form of CCTs. The CCTs give a summary view of the component interactions and the related performance data and are thus more suitable for a general overview of system performance. Figure 9 shows a CCT that gives a summary of the calls corresponding to the Petstore "sign on" user action. The performance metrics relate to snap shots that were taken during two different 20 minute test runs, i.e. when 50 and 200 users were in the system respectively. The columns in the diagram correspond to the name of method called, the total time spent in this method for all invocations (including time spent in calls made by this method) and the total number of calls. Thus in figure 9 for the 50 users snap shot the total amount of time spent in the 73 invocations of the SignOnEJB.authenticate method is 106 milliseconds (i.e. an average of 1.45 milliseconds per invocation). The CCT contains application level calls (to the SignOnEJB and the UserEJB components) and middleware calls related to the EJB container. The information in figure 9 shows that even when 50 users are in the system the average response time of the application level calls related to the "sign on" user action is similar to that when a single user is in the system (see figure 6). Figure 6 in fact shows an invocation of this method executing in 2 milliseconds. The time spent in the application component methods when 200 users are in the system is also relatively small. For example the average time spent in SignOnEJB.authenticate method at 200 users is in fact 2.13 milliseconds (581 milliseconds/273 invocations). However according to our performance tests (see figure 13) the average response time was over 5 times higher after the test run with 200 users in comparison to after the test run with 1 user for the Petstore application instrumented with netbeans. In order to identify the components that had the biggest impact on the overall response time at 200 users we further analysed the CCT. Figure 10 shows a number of the container calls that are invoked prior to an invocation of the SignOnEJB with 200 users in the system. As can be seen from the diagram a large proportion of the time is spent mainly in the web container (74.3% in com.sun.enterprise.web.conector.grizzly.TaskBase.run) and ejb container calls (13.4% in com.sun.ejb.containers.BaseContainer.postInvoke). In fact because the CCTs contain a summary of method call information they can be easily used to identify method

Component Method Name	50 Users Method Time	50 Users No. Calls	200 Users Method Time	200 Users No. Calls
com.sun.j2ee.blueprints.signon.ejb.SignOnEJB.**authenticate** (String, String)	106 ms (0.2%)	73	581 ms (0.3%)	273
com.sun.ejb.containers.BaseContainer.**invokeTargetBeanMethod** (java.lang.refle	91.0 ms (0.2%)	73	531 ms (0.2%)	273
com.sun.ejb.containers.EJBLocalObjectInvocationHandler.**invoke** (Object, java.lang.	9.18 ms (0%)	73	31.5 ms (0%)	273
com.sun.ejb.containers.EJBLocalObjectInvocationHandler.**invoke** (Class, java.lar	9.10 ms (0%)	73	31.2 ms (0%)	273
com.sun.ejb.containers.BaseContainer.**preInvoke** (com.sun.ejb.Invocation)	6.37 ms (0%)	73	22.1 ms (0%)	273
com.sun.ejb.containers.BaseContainer.**intercept** (com.sun.ejb.Invocation)	1.28 ms (0%)	73	4.82 ms (0%)	273
com.sun.ejb.containers.BaseContainer.**invokeTargetBeanMethod** (ja	1.18 ms (0%)	73	4.57 ms (0%)	273
Self time	0.816 ms (0%)	73	3.47 ms (0%)	273
com.sun.j2ee.blueprints.signon.user.ejb.UserEJB.**matchPassword**	0.276 ms (0%)	73	0.827 ms (0%)	273
com.sun.ejb.containers.BaseContainer.**onEjbMethodStart** ()	0.054 ms (0%)	73	0.160 ms (0%)	273
com.sun.ejb.containers.BaseContainer.**onEjbMethodEnd** ()	0.040 ms (0%)	73	0.113 ms (0%)	273
Self time	0.096 ms (0%)	73	0.243 ms (0%)	273

Fig. 9. CCT Snapshot of Petstore's "sign on" User Action at 50 users and 200 users

httpSSLWorkerThread-8080-0	231746 ms (100%)	24854
com.sun.enterprise.web.connector.grizzly.TaskBase.**run** ()	231746 ms (100%)	24854
Self time	172087 ms (74.3%)	24854
com.sun.ejb.containers.EJBLocalObjectInvocationHandler.**invoke** (Object, java.lang.reflect.Method, Object[])	59460 ms (25.7%)	46709
com.sun.ejb.containers.EJBLocalObjectInvocationHandler.**invoke** (Class, java.lang.reflect.Method, Object[])	59408 ms (25.6%)	46709
com.sun.ejb.containers.BaseContainer.**postInvoke** (com.sun.ejb.Invocation)	31065 ms (13.4%)	46465
com.sun.ejb.containers.BaseContainer.**intercept** (com.sun.ejb.Invocation)	22474 ms (9.7%)	46465
com.sun.ejb.containers.BaseContainer.**invokeTargetBeanMethod** (java.lang.reflect.Method, com.	22440 ms (9.7%)	46465
com.sun.ejb.containers.EJBLocalObjectInvocationHandler.**invoke** (Object, java.lang.reflect.Meth	6275 ms (2.7%)	6510
Self time	6072 ms (2.6%)	46465
com.sun.j2ee.blueprints.customer.ejb.CustomerEJB255789288_ConcreteImpl.**getProfile** ()	3075 ms (1.3%)	2157
com.sun.j2ee.blueprints.cart.ejb.ShoppingCartLocalEJB.**getItems** ()	2056 ms (0.9%)	539
com.sun.j2ee.blueprints.cart.ejb.ShoppingCartLocalEJB.**getSubTotal** ()	1532 ms (0.7%)	539
com.sun.ejb.containers.BaseContainer.**invokeTargetBeanMethod** (java.lang.reflect.Method,	1188 ms (0.5%)	1083
com.sun.j2ee.blueprints.contactinfo.ejb.ContactInfoEJB2107668920_ConcreteImpl.**getAddress**	725 ms (0.3%)	6000
com.sun.j2ee.blueprints.customer.ejb.CustomerEJB255789288_ConcreteImpl.**getAccount** ()	719 ms (0.3%)	10000
com.sun.j2ee.blueprints.signon.ejb.SignOnEJB.**authenticate** (String, String)	581 ms (0.3%)	273

Fig. 10. Container Calls Preceding Call to SignOnEJB at 200 users

hot-spots. Because of the level of our instrumentation filter settings (we chose to instrument the ejb container in detail only) we could not investigate further where time was being spent in the web container. However by expanding the CCT (postInvoke method) it was clear that within the EJB container much time was being spent handling transactions. In fact 13.1% of the time was spent in the method com.sun.ejb.containers.BaseContainer.completeNewTx. Thus using the Netbeans profiler we could see it was important for the Petstore application that the transactional settings were implemented efficiently. It has previously been show that such transactional checks can in some cases be redundant in JEE applications [53].

B. Performance Tests

Performance considerations are particularly important when performing dynamic analysis. For example, it is important to be aware of the performance overhead that can be attributed to each CIE approach. In this section we present the results from our performance tests. We have performed load tests on three different JEE applications. Next we describe our experimental set-up and the JEE applications that were used as part of our tests.

1) Experimental Set-up: Our experimental set-up consisted of three separate machines running Windows XP. Machine A (Intel Pentium 4 2.26GHz, 1Gb RAM) was used for load generation. To simulate load on the system we used the Apache Jmeter load generation tool [36]. Machine B (Intel Core Duo 2 Processor CPU E4400 @ 2Ghz, 2Gb RAM) hosted the Sun Glassfish application server version 9.1, running on the Sun Java 1.5.0 update 13 JVM. The embedded Derby database was used in conjunction with Glassfish. A third machine (Machine C, Intel Pentium 4 2.26 Ghz, 1Gb RAM) was used to remotely collect dynamically generated CLIs. The three machines ran on a dedicated network connected through a 100Mb switch. For all tests we used the Glassfish application server default settings. Before each load test the application server was restarted and the JVM was warmed up as suggested in the literature [54]. Also for all tests there was a ramp up time of 10 seconds for users entering the system and a think time of 1 to 3 seconds between user requests.

For our load tests we made use of a JEE testbed application called AdaptiveCellsJ. We also employed the Petstore ecommerce application described in section VII-A1. AdaptiveCellsJ allows for the creation of complex artificial JEE test-beds without requiring code development [55]. Instead,

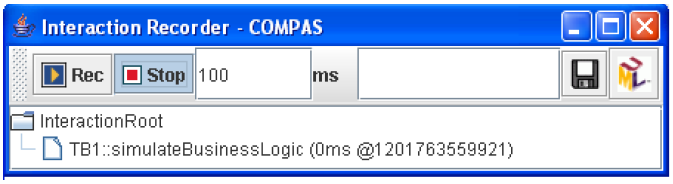

Fig. 11. CCT recorded using the IR on Simple-AdaptiveCellsJ

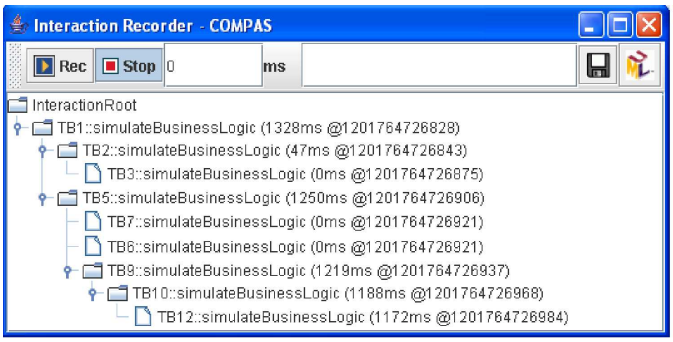

Fig. 12. CCT recorded using the IR on Complex-AdaptiveCellsJ

AdaptiveCellsJ can be used to assemble working JEE applications through simple configuration files. Using configuration files it is possible to construct complex interactions between JEE components. Furthermore AdaptiveCells allows for the emulation of CPU and memory usage through the specification of parameters in the configuration files. Thus, by selecting the appropriate configurations, testers and developers can replicate how resources such as CPU and memory are consumed by the different application components. Using the testbed we have constructed two applications for the purpose of our load tests. We have named these applications Simple-AdaptiveCellsj and Complex-AdaptiveCellsj. To give some insight into the structure of these JEE applications we have again made use of the COMPAS IR. The Simple-AdaptiveCellsj application consists of a single use case whereby a servlet calls a single EJB with a single method with no functionality (i.e no object creations or method calls are performed). It has been so designed such that the absolute time overhead of the dynamic CIE approaches can be assessed. Figure 11 shows a recording of the Simple-AdaptiveCellsj application using the IR. Complex-AdaptiveCellsj on the other hand is as the name suggests a more complex application. It consists of 6 different use cases which can be selected from a web tier (servlet) front end. Each use case results in the servlet component invoking a number of EJB components. Figure 12 shows a use case from the Complex-AdaptiveCellsj application recorded using the COMPAS IR. This particular interaction consists of 9 EJBs. For each use case a high level or resources (in particular CPU) are consumed by the various EJB components. For example figure 12 shows over 1000 milliseconds is spent in the business logic for this particular user action. This application was purposely designed in this way to assess the effect the CIE approaches had when applied to applications whereby the business logic was resource intensive (e.g. mathematical component based applications [56]). The resources consumed by the Petstore applications business logic is less than the Complex-AdaptiveCellsj application, but is naturally greater than the Simple-AdaptiveCellsj application and represents a typical ecommerce scenario.

2) Load Tests for Dynamic Extraction: Next we describe the load testing that has been carried out to assess the overhead associated with the dynamic CIE approaches. We have used the CLI extraction tools discussed above for the purpose of these tests. We have not assessed the COMPAS IR under load as this tool can not be applied in such circumstances (see section IV). For each application we have a non-instrumented version, a version that has been instrumented using the COMPAS JEEM tool, and a version that has been instrumented with the Netbeans profiler. For the Netbeans profiler we have set a filter to instrument the application components and the EJB container calls (i.e. the com.sun.ejb.* and javax.ejb.*).

The firsts tests carried out were on the Simple-AdaptiveCellsj application. After warming up the system we ran tests at 1, 10, 20, 50, 100, 200 and 500 users (corresponding to the logarithmic scale) for a period of 20 minutes. For each set of users we ran the test three times obtaining an average value for the response time of the simulated user requests. The test scripts for the Simple-AdaptiveCellsj application contained 2 user requests. An initial request for the application home page (html) followed by a request for the EJB component, through a Java servlet. For all tests carried out on the Simple-AdaptiveCellsj application the average response time did not exceed 1 millisecond. We do not show a graphical representation of this information as all response times recorded are the same (since Jmeter has millisecond precision) giving a flat graph. From these results we could see that the absolute time spent in the instrumentation added by both JEEM and Netbeans was very low.

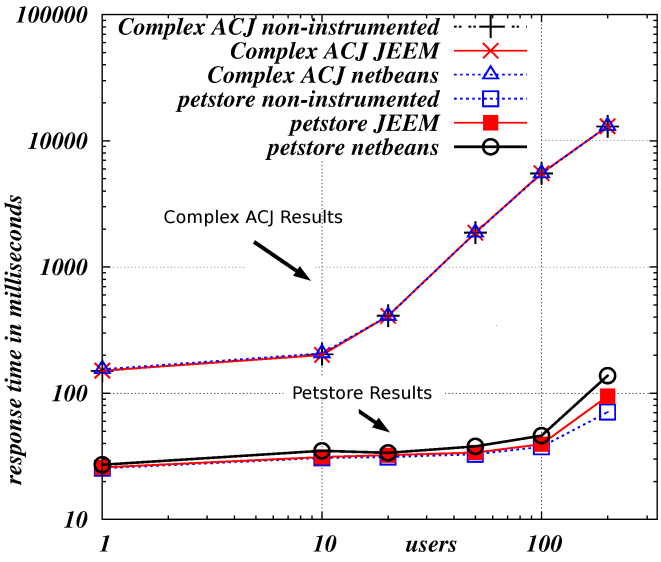

Fig. 13. Performance Test Results

The second set of tests carried out were on the Complex-AdaptiveCellsj application. The same numbers of users were simulated. However at 500 users errors were recorded in the application server logs and by the load generation tool as a consequence of system saturation. Thus these results

are not presented. The test scripts contained a call to the AdaptiveCellsj home page followed by 6 separate calls to 6 different call chains (each request consists of a call to a Java servlet followed by a chain of component calls, see figure 12 for an example). The results obtained are given in figure 13. As can be seen from the graph, there is no major deviation between the response times related to instrumented and non-instrumented ComplexAdaptivecellsj applications. We believe this has occurred since the time that can be attributed to the instrumentation is heavily outweighted by the time spent in the application business logic. For example figure 12 shows a sample use case from the Complex-Adaptivecellsj application where over 1 second is spent in the application components. Thus, even at one user the average response time of the different use cases for the non-instrumented Complex-Adaptivecellsj application was 150.33 milliseconds. This increases to 12995.33 for 200 users. The highest percentage runtime overhead was in fact recorded when 1 user was in the system. This amounted to 3.1% and can be attributed to the Netbeans profiler.

The final set of tests were carried out on the Petstore application. Again errors were reported at 500 users due to system saturation and these results are consequently not reported here. The load generation test script for the Petstore application consisted of the 9 user requests discussed in section VII-A1. A work mix was created such that 20% of users purchased products while the remaining 80% browsed. Results are shown in figure 13. As can be seen from the graph there is clearly overhead associated with both CIE tools when applied to the Petstore application. For the COMPAS JEEM tool the overhead is very low up to 100 users, ranging from 1.9% at 1 user, to 5.1% at 100 users. The increase in response time at 100 users compared to that at 1 user for the non-instrumented version of the Petstore application was 48.1%. At 200 users there is a noticeable increase in overhead to 33.6%. Furthermore when 200 users are in the system (non-instrumented) the response time had increased by almost 300%. We have previously noticed a similar trend with COMPAS JEEM tool when applied to an application of similar functionality to the Petstore application [51]. For applications of this nature when the system is heavily saturated (and response time has increased significantly) a noticeable increase in the instrumentation overhead can be observed.

The overhead associated with the Netbeans profiler on Petstore follows a similar trend. At 1 and 100 users the overhead was calculated at 7.1% and 22.8% respectively. However at 200 users a sharp rise in overhead was observed with overhead increasing to 94.3%. We believe the higher levels of overhead associated with the Netbeans profiler in comparison to COMPAS JEEM can be attributed to the fact that the Netbeans profiler obtains more detailed information collecting both system and application level data. Furthermore the Netbeans profiler (by default) also collects basic heap, garbage collection and threading information which is not obtained using COMPAS JEEM. Thus it seems that when there is an increase in the number of users in the system, the overhead related to obtaining CLIs increases (as was the case with COMPAS JEEM) and so also does that associated with collecting heap, garbage collection and thread information.

VIII. Conclusions

Component based frameworks are largely used for the development of large complex (and often physically distributed) enterprise systems. Due to their sheer scale and complexity, understanding these systems can be difficult, even for domain experts. Component level interactions (CLIs) capture component dependencies (and in some cases the context in which these dependencies occur). Thus, the availability of CLIs is essential for analysing, understanding and monitoring component based systems. In fact, CLIs have a wide range of applications including optimising application performance, understanding the origins of performance problems and reverse engineering applications. In this paper we have discussed our motivation for collecting CLIs and outlined the different ways in which CLIs can be represented. The paper also presented a range of dynamic approaches for obtaining CLIs that we believe are representative for the current state of the art. For each approach we discussed its utility, analysed its advantages and disadvantages, and described how its technical requirements can be fulfilled. In addition we presented available implementations for each approach as well as related work. Finally in the evaluation section we showed how each component interaction extraction (CIE) method can be used in realistic environments, and we reported the results of a complete range of performance measurements that were undertaken in order to determine their general impact on target systems.

The first component level interaction extraction (CIE) approach presented (assisted recording) can produce run-time values for component level interactions associated with user-defined business cases. It omits middleware calls capturing inter-component communications at the application component level, however, due to its unintrusiveness, 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 eliminates the one thread requirement, at the cost of increased intrusiveness (an identifier for each interaction needs to be associated with a thread). We describe how application level interactions can be extracted automatically for physically distributed heterogeneous systems. The performance of an enterprise systems, however, is influenced by both the application level component interactions, as well as the interactions between the application components and the middleware components. While 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 precision in detecting and solving performance problems, however, it comes at the cost of more specific, and possibly restrictive, run-time environment requirements.

Our performance evaluation of the dynamic extraction techniques shows how the performance overhead observed in terms of a percentage increase in response time is very much application and load specific. Three applications were employed for our tests. Our results show that, for a simple one

bean application, which had no real functionality, there was no increase in overhead observed for any of the CIE approaches with user loads ranging from 1 to 500 users. For our second application which was highly resource intensive, the overhead related to the dynamic interaction extraction was again very low for all user loads. Finally for the third application tested, which was a JEE sample ecommerce application, overhead for each of the dynamic extraction approaches tested increased steadily as the load on the system increased. When the system became highly saturated a large increase in system overhead was also observed.

Extracting component interactions from software systems is essential in understanding and correcting quality problems in their architectural context. The approaches presented in this paper cover the most widely used techniques for interaction extraction in enterprise Java systems, while presenting in detail their implementation, benefits, disadvantages and performance considerations. We believe that this can help system developers and integrators better understand the implications of the possible choices when building and managing component based systems.

ACKNOWLEDGMENTS

Trevor Parsons' and John Murphy's work is funded under the Enterprise Ireland Innovation Partnership in cooperation with IBM and University College Dublin.

The authors would also like to thank Adriana E. Chis for assisting with the performance tests that were carried out as part of this work.

REFERENCES

[1] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of uml sequence diagrams for distributed java software," in *IEEE Transactions on Software Engineering*, vol. 32, no. 9, 2004.

[2] D. F. Jerding, J. T. Stasko, and T. Ball, "Visualizing interactions in program execution," in *International Conference on Software Eng.*, 1997.

[3] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: a call graph execution profiler," in *Symposium on Compiler Construction*, 1982.

[4] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer, "Using runtime paths for macro analysis," in *Proc. 9th Workshop on Hot Topics in Operating Systems*, 2003.

[5] T. Parsons, "Automatic detection of performance design and deployment antipatterns in component based enterprise systems." Ph.D. dissertation, University College Dublin, 2007.

[6] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[7] M. Trofin and J. Murphy, "Static verification of component composition in contextual composition frameworks," *International Journal on Software Tools for Technology Transfer (STTT)*, January 2008.

[8] "Borland software corporation. optimizeit suite." [Online]. Available: {http://www.borland.com/optimizeit}

[9] "The netbeans profiler." [Online]. Available: http://profiler.netbeans.org/

[10] P. Tonella and A. Potrich, *Reverse Engineering of Object Oriented Code*. Springer, 2005.

[11] T. Parsons, A. Mos, and J. Murphy, "Non-intrusive end to end run-time path tracing for j2ee systems," in *IEE Proc. Software*, August 2006.

[12] T. Gschwind, J. Oberleitner, and M. Pinzger, "Using run-time data for program comprehension," in *Proceedings of the 11th International Workshop on Program Comprehension*. IEEE, May 2003.

[13] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic, internet services," in *Proc. Int. Conf. on Dependable Systems and Networks (IPDS Track)*, 2002.

[14] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, 2003.

[15] "Ibm autonomic computing." [Online]. Available: http://www.research.ibm.com/autonomic

[16] A. Diaconescu, A. Mos, and J. Murphy, "Automatic performance management in component-based software systems," in *Proceedings of the International Conference on Autonomic Computing (ICAC04)*, New York, USA, May 2004.

[17] A. Mos, "A framework for adaptive monitoring and performance management of component-based enterprise applications," Ph.D. dissertation, Dublin City University, Ireland, 2004.

[18] M. Trofin and J. Murphy, "A self-optimizing container design for enterprise java beans applications," in *Proceedings of the Second International Workshop on Dynamic Analysis*, 2004.

[19] ——, "Removing redundant boundary checks in contextual composition frameworks," in *Journal of Object Technology*, July-Augus 2006. [Online]. Available: http://www.jot.fm/issues/issues2006_07/article2

[20] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy, "Finding and removing performance bottlenecks in large systems." in *ECOOP*, 2004.

[21] M. Kunnumpurath, "Jboss administration and development third edition (3.2.xseries)," October 2003.

[22] J. D. Gradecki and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, 2003.

[23] "Aspectj." [Online]. Available: http://www.eclipse.org/aspectj

[24] "The java virtual machine tools interface." [Online]. Available: http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/

[25] K. Govindraj, S. Narayanan, B. Thomas, P. Nair, and S. Peeru., "On using aop for application performance management." in *Fifth International Conference on Aspect-Oriented Software Development (Industry Track)*, Bonn, Germany, March 2006.

[26] M. Dahm., "Bytecode engineering." in *Java-Information-Tage*, Sept. 1999.

[27] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient java bytecode translators," in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, 2003.

[28] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A code manipulation tool to implement adaptable aystems," in *Proceedings Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and Extensible Component Systems)*, Nov. 2002.

[29] R. Vallee-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing java bytecode using the soot framework: Is it feasible?" in *Compiler Construction, 9th International Conference*, 2000, pp. 18–34.

[30] "Bea. serp." [Online]. Available: http://serp.sourceforge.net/

[31] C. Austin, "J2se 5.0 in a nutshell," 2004. [Online]. Available: http://java.sun.com/developer/technicalArticles/releases/j2se15

[32] *Java Management Extensions (JMX)*, Sun Microsystems, http://java.sun.com/products/JavaManagement.

[33] "Java internet glossary." [Online]. Available: http://mindprod.com/jgloss/time.html

[34] A. Mos and J. Murphy, "Performance management in component-oriented systems using a model driven architecturetm approach," in *Proceedings of the 6th International Enterprise Distributed Object Computing Conference*. IEEE Computer Society, 2002.

[35] *Open System Testing Architecture (OpenSTA)*, http://www.opensta.org.

[36] *Apache JMeter*, http://jakarta.apache.org/jmeter/.

[37] *JSR 149, Work Area Service*. [Online]. Available: http://jcp.org/en/jsr/detail?id=149

[38] *IBM, WebSphere 6 Documentation*. [Online]. Available: http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp

[39] E. Wohlstadter, S. Jackson, and P. Devanbu, "DADO: Enhancing middleware to support crosscutting features in distributed, heterogeneous systems," in *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 2003.

[40] T. Gschwind, K. Eshghi, P. K. Garg, and K. Wurster, "WebMon: A performance profiler for web transactions," in *In Proc. IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*. IEEE, Jun. 2002.

[41] Y. L. L.C. Briand and J. Leduc, "Towards the reverse engineering of uml sequence diagrams for distributed, multithreaded java software," in *Technical Report SCE-04-04*, September, 2004. [Online]. Available: http://www.sce.carleton.ca/Squall

[42] R. M. P. Barham, R. Isaacs and D. Narayanan, "Magpie: online modelling and performance-aware systems," in *In the 9th Workshop on Hot Topics in Operating Systems*, May, 2003.

[43] R. I. P. Barham, A. Donnelly and R. Mortier, "Using magpie for request extraction and workload modelling," in *Symposium on Operating Systems Design and Implementation*, 2004.

[44] M. Dmitriev, "Profiling java applications using code hotswapping and dynamic call graph revelation," in *Proceedings of the 4th International Workshop on Software and Performance*, 2004.
[45] *The Java 2 Platform, Enterprise Edition technology (J2EE)*, Sun Microsystems, http://java.sun.com/j2ee/.
[46] "Sun java studio." [Online]. Available: http://developers.sun.com/jsenterprise/
[47] *ej-technologies*, *JProfiler*, http://www.ej-technologies.com/products/jprofiler/overview.html.
[48] "Quest, performasure." [Online]. Available: http://www.quest.com/PerformaSure
[49] Sun Microsystems, "Java Pet Store Demo," http://java.sun.com/developer/releases/petstore/, 2003.
[50] T. Parsons and J. Murphy, "Detecting performance antipatterns in component based enterprise systems," in *Journal of Object Technology*, March-April 2008. [Online]. Available: http://www.jot.fm/issues/issue_2008_03/article1/
[51] T. Parsons, A. Mos, and J. Murphy, "Non-intrusive end to end run-time path tracing for j2ee systems," in *IEE Proc. Software*, August 2006.
[52] M. C. et. al, "Pinpoint, problem determination in large, dynamic, internet services," in *In Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*, 2002.
[53] M. Trofin, "Call graph-directed boundary condition analysis in contextual composition frameworks," Ph.D. dissertation, University College Dublin, 2007.
[54] L. B. A. Buble and P. Tuma, "Corba benchmarking: A course with hidden obstacles," in *Proc. IPDPS Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems*, 2003.
[55] *AdaptiveCells/J*, http://adaptivecellsj.sourceforge.net/. [Online]. Available: http://adaptivecellsj.sourceforge.net/
[56] "Webcab components." [Online]. Available: http://www.webcabcomponents.com/

Trevor Parsons received his B.Sc. in Computer Applications at Dublin City University and his Ph.D. in Computer Science at University College Dublin. His research interests include design analysis, software antipatterns, component based software engineering, monitoirng tools and software performance engineering. He is a member of the IBM Centre for Advanced Studies and is a postdoctoral researcher at the Performance Engineering Lab in the School of Computer Science and Informatics in University College Dublin. He is currently leading a joint research initiative between University College Dublin and the IBM Dublin Software Lab in the area of expert tools for system and performance testers.

Adrian Mos is a senior research engineer at INRIA (the French National Research Institute in Computer Science and Automatics) where he is involved in technical coordination of several Service Oriented Architecture (SOA) projects. Prior to this position, he has worked in both industrial and research environments focusing on building automated management solutions for enterprise Java systems. He is an Eclipse Foundation committer and a component lead in the SOA Tools Project. He holds a PhD from Dublin City University, Ireland and a Computer Engineering degree from the Polytechnic University of Timisoara, Romania.

Mircea Trofin received his BASc in Computer Engineering at University of Waterloo, Canada, and his Ph.D. in Computer Science at University College Dublin. His research interests include the construction and analysis of component-based software applications and platforms for such applications. He is currently with the Microsoft .NET team, in Redmond, WA.

Thomas Gschwind is a researcher at IBM. His research focuses on business process models, their translation to software systems and on software engineering in general. Before joining IBM, he was assistant professor at Technische Universitt Wien where he has been working on the EASYCOMP project where he has been leading the development of new component instrumentation and composition technologies such as the Vienna Component Framework and Type-Based Adaptation.

John Murphy received his B.E. in Electronic Engineering at University College Dublin, a M.Sc. in Electrical Engineering at the California Institute of Technology and a Ph.D. in Electronic Engineering at Dublin City University. He is a SMIEEE, a Fellow and a Chartered Engineer with Engineers Ireland and holds an IBM Faculty Fellowship. He is a member of the Editorial Board for IEEE Communications Letters, IET Communications and Telecommunications System Journal. His research is in the area of performance engineering of networks and distributed systems, with almost 100 peer-reviewed journal articles or international conference full papers published and is funded by 17 competitive research grants with funding in excess of $4m. He has supervised 12 Ph.D. students to completion.