DYNAMIC ANALYSES FOR UNDERSTANDING AND OPTIMIZATION OF ENTERPRISE JAVA APPLICATIONS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Aleksandar Pantaleev, B.A., M.S.

* * * * *

The Ohio State University

2008

Dissertation Committee:

Atanas Rountev, Adviser

Bruce Weide

Neelam Soundarajan

Rajiv Ramnath

Approved by

Adviser

Graduate Program in
Computer Science and
Engineering

ABSTRACT

Enterprise Java (J2EE) applications present significant challenges for horizontal scaling due to their large code and dataset sizes, as well as the clustering models of application servers. Our work targets a potential reimplementation of the way Enterprise Java applications are clustered. We propose three dynamic analyses, motivated by a new design for a clustered enterprise application framework, which allow better understanding of existing applications and provide support for migrating such applications to a possible future implementation of this design. The first dynamic analysis traces the flow of primary keys, which uniquely identify certain objects within the business tier of a J2EE application. It identifies the entry points of such data to the business tier, which provides crucial information for the migration of the application towards the new clustered design. We propose an algorithm to achieve this identification and implement the approach through bytecode instrumentation. Our experimental results indicate that the analysis has practical overhead and achieves excellent precision. The second dynamic analysis identifies the entry points of query parameters to the business tier of a J2EE application, as well as certain relationships in which this data participates. This is achieved through a number of enhancements to the algorithm for the first dynamic analysis. The analysis implementation exhibits reasonable overhead and achieves high-precision identification of the entry points of

query parameters, as shown by our experimental studies. The third dynamic analysis identifies instances of the Data Transfer Object (DTO) design pattern within a J2EE application. It provides helpful information for the first dynamic analysis, and could be used as a preprocessing step to improve its precision and overhead. In addition, DTO identification is useful for a number of other tasks related to program comprehension, performance optimization, and software evolution. The algorithm investigates the lifecycle of certain objects within the business tier of a J2EE application, and matches that lifecycle against a state transition diagram that describes the behavior of a DTO. The algorithm was implemented through the Java Virtual Machine Tool Interface (JVMTI). Experimental results indicate that the analysis can identify the majority of DTO instances while exhibiting manageable overhead.

This work is a step towards a new design for clustered enterprise applications. The proposed dynamic analyses solve some key problems for a potential future implementation of services that provide replication at the *object* level in enterprise application servers. Such services will solve the significant problem of horizontal scalability in Enterprise Java applications.

To my Lynna

ACKNOWLEDGMENTS

I would like to express my extreme gratitude towards my academic adviser, Prof. Nasko Rountev. He not only provided support and guidance throughout my academic career, but also taught me how to be successful, and served as a firm protective barrier guarding me from bureaucratic obstacles to my goals. Without his help and constant feedback I would have never achieved my current stage in life.

I am grateful to Prof. Rajiv Ramnath, who is a rare business-oriented person to see another point of view. Under his patient guidance I made sense of much of the business part of Software Engineering, and was able to compare it to the Free Software movement in an informed way.

I am grateful to Prof. Neelam Soundarajan for teaching reasoning about object-oriented systems in such interesting lectures. I am also grateful to him and Prof. Bruce Weide for providing valuable feedback to this thesis.

Thanks to my parents and grandparents for supporting me remotely through these five years, and for patiently enduring my inability to visit them as often as I wanted. Finally, I would like to thank my wife, Lynna, for everything that she is.

VITA

December 2007	M.S. Computer Science & Engineering, The Ohio State University
May 2003	B.A. Computer Science, American University in Bulgaria
September 2004 – present	Graduate Research/Teaching Associate, The Ohio State University
September 2003 – August 2004	Graduate Fellow, The Ohio State University
January 19, 1981	Born – Ruse, Bulgaria

PUBLICATIONS

Research Publications

A. Pantaleev and A. Rountev. Identifying Data Transfer Objects in EJB Applications. In *Fifth International Workshop on Dynamic Analysis*, May 2007.

A. Pantaleev and J. Josephson. Prospects for Dynamic ISR Tasking and Interpretation Based on Standing Orders to Sensor Networks. In *Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications*, April 2007.

A. Pantaleev and J. Josephson. Higher-level Fusion for Military Operations Based on Abductive Inference: Proof of Principle. In *Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications*, April 2006.

C. Cekova, B. Chandrasekaran, J. Josephson and A. Pantaleev. Simulation-Based Planning for Peacekeeping Operations: Selection of Robust Plans. In *Modeling and Simulation for Military Applications*, April 2006.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Software Engineering	Prof. Atanas Rountev
Artificial Intelligence	Prof. John Josephson
Database Systems	Prof. Hakan Ferhatosmanoglu

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Modern enterprise object-oriented applications present significant challenges for both horizontal and vertical scaling due to their large code and dataset sizes, as well as their clustering and distribution models. *Horizontal scaling* is the ability to connect multiple hardware or software entities, such as servers, so that they work as a single logical unit. *Vertical scaling* is the ability to increase capacity by adding resources to an existing entity (e.g., by replacing the CPU with a faster one). The enterprise clustering models do not necessarily utilize the horizontal scaling capabilities of redundant physical or virtual machines; rather, they are devised to target other desirable properties of enterprise applications such as high availability and decreased probability of data loss.

Our work targets a potential reimplementation of the way enterprise applications are clustered. Its main goal is the introduction of horizontal scalability at the object level without sacrificing other desirable properties that are already enabled by current designs (e.g., high availability and redundancy). To that end, we outline a new design of a clustered enterprise application framework. The main focus of our work are three dynamic analyses motivated by this design. These analyses allow better understanding of existing enterprise applications and provide support for migrating

such applications to a possible future implementation of the design, e.g., as a service within a Java Enterprise application server.

1.1 Enterprise Java and Scalability

The Java platform, Enterprise Edition (J2EE), is the current leader among enterprise computing platforms. It sets the standard against which all others are compared and represents the current state of the art. The memory footprint of J2EE applications typically consists of a large number of sparsely-connected object clusters, where each cluster consists of a few hundreds of densely-connected application-specific objects intermixed with application server helper objects.

J2EE applications are implemented through the use of isolated functional areas, called *tiers*. Typical tiers are the client tier, the middle (business) tier, and the data tier. The middle tier is the one of most significance, since the business logic of the application is situated there, together with the memory footprint of the object graph; therefore, it holds the most potential benefits if optimized correctly.

J2EE applications are deployed on top of a J2EE server. J2EE servers, or *application servers*, provide standard services to the middle tier in a multi-tiered enterprise application, as well as the underlying functionality to enable such services to work together with enterprise applications. One service that is provided at this level is clustering.

Clustering allows administrators to run a single enterprise application on several parallel J2EE servers. The processing load is distributed across the cluster nodes according to a pre-defined policy, while the memory footprint is shared across all nodes. The memory footprint of an application deployed on an application server consists of

a mix of application-specific objects and supporting server objects. In a typical clustered environment the application-specific objects are *replicated* on all cluster nodes. This cluster design allows for various crucial services that an application server must provide, such as reliability, hot swapping of physical machines, reduction of response time, etc., but also results in an impossibility to provide for *horizontal scalability at the memory level*, which sets hard limits for the total memory consumption of an enterprise application. The mechanics of the clustering service also typically cause network congestion within the cluster, especially with large clusters. These problems are presented in more detail in Chapter 2. This chapter also provides a description of our proposal to solve them through a new clustering design.

All current clustering designs utilize partitioning at the *class* level. Our proposal relies on the fact that it is possible to partition the memory footprint of a typical J2EE application efficiently at the *object* level. Such partitioning allows distributing the object graph of an enterprise application over multiple physical or virtual machines, alleviating the issues mentioned previously. Each object belongs to one machine and communicates with other objects either locally, if those objects reside on the same machine, or over the network.

An additional service is essential for the implementation of this new design: one that serves as a bridge between client tiers and the middle tier of a J2EE application, examining client requests and forwarding them to the middle tier machines holding the appropriate data. This service requires intimate knowledge of the inner workings of the currently hosted J2EE application to forward requests properly, and is in effect an *object-level lookup service*. Building such a service presents a number of challenges. Specifically, implementations of this service will rely on comprehending

certain aspects of the data flow within enterprise applications. As a step towards building such a service, we propose three dynamic analyses for understanding critical aspects of J2EE applications. The output of these analyses will be used by a J2EE server administrator to configure the additional service with respect to a specific J2EE application.

1.2 Primary Key Identification and Flow

The first dynamic analysis traces the flow of primary keys within an application's middle tier. It identifies the entry points of primary keys or parts of primary keys to that tier, which provides crucial information for the migration of the application towards the conceptual clustered design briefly outlined in the previous section. The next paragraphs summarize this analysis at a high level, while Chapter 3 describes it in greater detail.

A primary key is a piece of data that uniquely identifies an object in the middle tier of a J2EE application. Primary keys are often passed from the client tier to the middle tier, either directly or, in case of composite keys, as separate parts to be assembled within the middle tier. Primary keys are then typically passed from the middle tier to the data tier. The object related to the primary key in question is either fetched from the data tier, or a reference to it in the local memory space (local to the middle tier) is returned if the object is already present there. This object is then used to execute the business logic of the application and service the original request from the client tier.

To function correctly, the object-level lookup service must be aware of the flow of primary keys between the client and the middle tier. A primary key is a crucial piece

of information in that context, because it uniquely identifies the middle tier physical machine where the object is present, and respectively where the client tier request must be redirected.

The analysis investigates the data flow of certain objects and primitive values within the middle tier of an application, and finds the data that are used as primary keys, as well as their entry points to the middle tier. As described previously, this information will be used by an administrator to configure the object-level lookup service for the particular application.

Chapter 3 describes several versions of the analysis algorithm. Its simplest version proceeds as follows:

- Whenever a remote call, coming from a client tier, is made against an entry object to the middle tier, intercept the input parameters to the call and remember information that uniquely identifies them in a data structure that we will refer to as a *value flow graph* (VFG).

- Whenever an assignment with the participation of the value of such a parameter happens within the middle tier, intercept the left-hand side of that assignment, and create an association between the memory location standing in that left-hand side and the origin of the value in the VFG.

- Whenever a call is made from the middle tier to the data tier with the participation of a primary key, look up the memory location holding the primary key value in the VFG. Follow the association link and output the origin of the primary key, which is its entry point to the middle tier.

In addition, another version of the analysis was designed. This version outputs the propagation path of a primary key within the middle tier of an application. This information improves program comprehension and can be used for debugging, testing, and maintenance purposes.

The analysis was implemented in Java through the use of bytecode instrumentation with the help of the Soot framework [65]. It was evaluated on three separate J2EE applications, running on the JBoss application server [31]. The largest and most complicated one was a real-world Enterprise Java application — the J2EE Certificate Authority (EJBCA) [22]. The two smaller applications were Duke's Bank [21] and Pet Store [30]. The entire experimental setup was deployed on Sun's Java 6 Java Virtual Machine (JVM).

We implemented two separate versions of the algorithm. The first version instruments an enterprise application to call hooks to the VFG in an online mode (i.e., while the application is running). The second version is offline: it instruments an enterprise application to output all relevant data to a file, which is then read and processed after the application has completed its test run.

Our experimental evaluation measured the total number of primary keys passed from the middle tier to the data tier, and the number of such keys that our analysis was able to match with requests incoming from a client tier. The analysis matched approximately 93% of all primary keys passed from the middle tier to the data tier when run with EJBCA's own test suite, and 100% with the two smaller applications.

We measured the run-time overhead of the analysis (both online version and offline version) and the time to process the execution trace (offline version only). The run-time overhead was 279% for the online version, and 150% for the offline version when

run on EJBCA. The overheads for the two smaller applications were similar. The time to process the trace for the offline analysis was 15 minutes and 33 seconds for EJBCA, and approximately 1 minute per trace for the smaller applications. Note that this analysis, as well as the other two analyses described below, will typically have to be executed only once per J2EE application because of the specifics of the intended use (see Section 2.4), as opposed to being continuously run in the background for the lifetime of an application. Thus, the analysis cost of 2.8× slowdown is practical and quite reasonable.

In summary, we have designed and implemented a dynamic analysis that identifies the entry points of primary keys to the middle tier of an J2EE application. Our experimental evaluation indicates that the analysis has practical overhead and achieves excellent precision. These results contribute towards the object-level lookup service described previously by identifying a major portion of the data that must be looked up so that client requests are routed to the correct middle tier machine.

1.3 Query Data Relationship Identification

The remaining portion of the data that must be looked up consists of parameters to queries, which the middle tier executes against the data tier. The second dynamic analysis we propose identifies such data as well as certain frequently-occurring relationships in which this data participates. Chapter 4 describes the analysis in detail, while the next paragraphs outline it at a high level.

The middle tier of a J2EE application has the capability of retrieving objects from the database not only on the basis of their primary keys, but also based on other parameters they contain (e.g., retrieve a Student object with a particular name, as

opposed to retrieving a Student object with a particular social security number). This functionality is made available by a query language built specifically for J2EE. Queries in that language resemble standard Structured Query Language (SQL) queries, and typically need specific parameters to fetch objects.

The object-level lookup service must be aware of such parameters that pass between the client and the middle tier. Another crucial piece of information the object-level lookup service needs in this scenario is the possible relationships in which those parameters participate, so that it can recreate the queries and return the addresses of all machines containing related objects.

The dynamic analysis investigates the flow of certain query-related objects and primitive values within the middle tier of an enterprise application, similarly to the analysis of primary keys. An important difference is a pre-processing step to this analysis, which statically parses all queries present in an enterprise application's source code, extracts certain relationships among the parameters present in those queries, and provides that information to the dynamic analysis. As with the previous analysis, an extension was designed to identify the propagation path of a query parameter.

The analysis was implemented with the same tools as the previous one, and was run on the same benchmarks. The implementation language was Java, and the bytecode instrumentation framework was Soot. The three J2EE applications (EJBCA, Duke's Bank, and Pet Store) were run on JBoss using Sun's Java 6 JVM.

We measured the total number of queries invoked, and the number of such queries our analysis was able to fully match with requests incoming from a client tier. The analysis matched 96% of all queries with EJBCA's test suite, and 100% when tested

on the two smaller applications. The approach also handled the most common relationships present in J2EE queries. Those accounted for 87% of all queries in EJBCA and 100% of all queries in the two smaller applications. The run-time overhead of the analysis was 293%.

In summary, we have designed and implemented a dynamic analysis that identifies the entry points of query parameters to the middle tier of an J2EE application, as well as commonly-used relationships in which those parameters participate within the queries. These results add to those of the previous dynamic analysis, and contribute information necessary for the implementation of an object-level lookup service.

1.4 Data Transfer Object Identification

The third dynamic analysis we propose identifies instances of the Data Transfer Object (DTO) design pattern at the boundary between the client and the middle tiers of an enterprise application. It provides helpful information for the first dynamic analysis described above, and may be used as a pre-processing step to it to improve its precision and overhead. In addition, DTO identification is useful for a number of other tasks related to program comprehension, performance optimization, and software evolution.

DTO is a design pattern common in enterprise systems. An object that is an instance of the DTO pattern encapsulates a set of values, allowing remote clients to request and receive an entire value set with a single remote call. It is generally used to alleviate network overhead within a distributed system. In the context of J2EE applications, DTOs are commonly used as *composite primary keys*, which are primary keys of complex, application-defined types. Identifying such objects and their types

is a helpful pre-processing step to the dynamic analysis of primary keys, which may utilize that information to track only certain non-standard types of parameters.

The analysis investigates the lifecycle of certain objects living at the boundary between the middle and the client tier of a J2EE application, and matches it against a state transition diagram that describes the behavior of a DTO. It was implemented in C, and interfaces with the JVM that runs the J2EE server and application through the Java 6 version of the JVM Tool Interface (JVMTI). The implementation was evaluated on EJBCA, running on JBoss, on the Java 6 JVM. It identified 10 out of a total of 11 DTO instances, with one false positive and one false negative. The run-time overhead of the analysis was 263%.

In summary, this dynamic analysis identifies instances of the DTO pattern within a J2EE application. The overhead of the analysis is practical, and the achieved precision is high. This analysis can be used as a pre-processing step to the first dynamic analysis described earlier. This work first appeared in [50].

1.5 Outline

The rest of this dissertation is organized as follows. Chapter 2 provides background information about J2EE and the scalability problems it poses, as well as a proposed alternative distributed design that contributes towards solving these problems. Chapter 3 presents the first dynamic analysis, which identifies the entry points of primary keys to the middle tier. A description of the second dynamic analysis, which identifies query parameters and some of their relationships, is introduced in Chapter 4. Chapter 5 explains the third dynamic analysis, which identifies instances of the DTO design pattern within a J2EE application. Related work is discussed

in Chapter 6. Finally, Chapter 7 concludes by summarizing the dissertation and discussing possible directions for future research.

CHAPTER 2

BACKGROUND AND PROBLEM OVERVIEW

2.1 Overview of Enterprise Java Applications

The Java platform, Enterprise Edition (J2EE), provides an API and run-time environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications. A shorthand name for such applications is "enterprise applications," so called because they are designed to solve the problems usually encountered by large enterprises. Enterprise applications are not only useful for large corporations, agencies, and governments. The benefits of an enterprise application are helpful, even essential, for individual developers and small organizations in an increasingly networked world.

The features that make enterprise applications powerful, such as security and reliability, often make them complex. The J2EE platform is designed to reduce the complexity of enterprise application development by providing an architectural model, an API, and a run-time environment that allows developers to concentrate on functionality. J2EE is a multi-tiered, component-based architecture.

2.1.1 J2EE Tiers

In a multi-tiered application, the functionality of the application is separated into isolated functional areas, called tiers. Typically, multi-tiered applications have a *client* tier, a *middle* tier, and a *data* tier. The client tier consists of a client program that makes requests to the middle tier. The middle tier's components handle client requests and process application data, storing it in a permanent data store in the data tier. J2EE application development concentrates on the middle tier.

The client tier consists of application clients that access a J2EE server. The server processes the requests that a client makes, and returns a response back to the client. Clients can be web browsers, stand-alone applications, or other servers, and they usually run on a different physical machine from the J2EE server.

The middle tier contains a *business* tier, and often also contains a *web* tier. The web tier consists of components that handle the interaction between HTTP-based clients and the business tier. It contains control flow information related to web pages. Typical clients that utilize an enterprise application's web tier, if it exists, are web browsers. The business tier consists of components that provide the business logic for an application. In a properly designed enterprise system, the core functionality exists in the business tier components. This is the most important tier in a J2EE application, and it presents the most interesting research problems to solve. Crucial components here are Enterprise JavaBeans (EJBs).

The enterprise information systems (EIS) tier consists of database servers, enterprise resource planning systems, and other legacy data sources, such as mainframes. These resources typically are located on a separate physical machine than the J2EE server, and are accessed by components on the business tier.

2.1.2 Application Servers and Containers

A J2EE server implements the J2EE platform APIs and provides the standard J2EE services. J2EE servers are sometimes called application servers. They host several application component types that correspond to the middle tier in a multi-tiered application. The J2EE server provides services to these components in the form of a *container*.

A J2EE container is the interface between a component and the lower-level functionality provided by the J2EE platform to support that component. The functionality of the container is defined by the J2EE platform, and is different for each component type. The J2EE server allows the different component types to work together to provide functionality in an enterprise application.

Two important structural segments of an application server are a *web container*, corresponding to the web tier, and an *EJB container*, corresponding to the business tier. The web container is the interface between web components and the web server. A web component can be a servlet, a JSP page, or a JavaServer Faces page. The container manages the component's lifecycle, dispatches requests to application components, and provides interfaces to context data, such as information about the current request. The EJB container is the interface between *enterprise beans*, which provide the business logic in a J2EE application, and the J2EE server. The EJB container runs on the J2EE server and manages the execution of an application's enterprise beans.

The business tier of a J2EE application is built out of several EJB component types. Stateless and stateful *session beans* are carriers of most of the application business logic, e.g., updating a shopping cart. These are the components to which

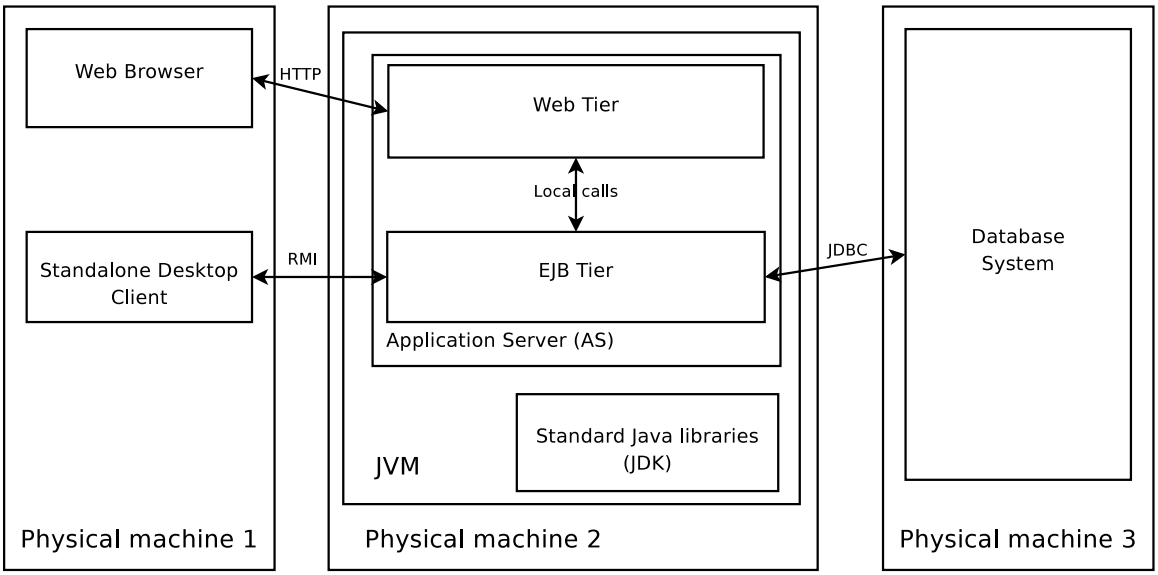

Figure 2.1: A simple J2EE deployment scenario

clients (e.g., web tier objects) connect. *Entity beans*, on the other hand, carry most of the state of the business tier, and usually are mirror images of database rows. In simplistic terms, session beans can be viewed as operators, while entity beans are the operands to those operators. The database schema is created (usually automatically) based on a well-defined *object-relational mapping* extracted from the entity beans. Thus an application programmer may focus on the business logic of the application and its object design, as opposed to dealing with extraneous activities such as persisting the object data. Other EJB types exist as well, but they are of less importance.

2.1.3 A Simple Deployment Scenario and Use Case

Figure 2.1 represents a very simplistic design for the deployment of a J2EE architectural stack. The core of the system runs on physical machine 2. A single Java

Virtual Machine (JVM) is running in an OS process. The J2EE application server is running within that JVM, together with the general utility libraries that JDK provides. The application server hosts the middle layer of the enterprise application, which in this case has a web tier as well as the obligatory business tier. Consequently, both the web container and the EJB container of the application server are in use. The web tier and the EJB tier communicate with one another through local calls, as they are running in the same JVM.

Physical machine 3 holds the data tier. It is running a relational database management system, which provides persistent storage for the data of the enterprise application. Components from the business tier transfer information to and from the data tier through a driver compliant with the Java DataBase Connectivity (JDBC) standard.

Physical machine 1 executes the client tier; there are usually many of these client machines. It is running a web browser, which connects to the web tier of the enterprise application through HTTP and presents information to the user. It could also be running a standalone desktop client, which connects directly to the business tier through RMI (Remote Method Invocation) calls.

A typical use case utilizing all tiers of the enterprise application, as presented in Figure 2.1, would be the following:

- The user directs her web browser, running on her client machine (physical machine 1), to the address of the web server that is a part of the web container of the enterprise application. The browser connects to the web tier through HTTP and retrieves a login page that requires a username and a password.

- The user enters her username and password, and clicks the "Submit" button. The browser sends the data to the web tier.

- The web tier locates a component in the EJB tier that provides user authentication. (As discussed later, this EJB component is typically a *session bean*.) The web tier makes a local call to that component, and sends it the user data.

- The EJB component retrieves the user data for the username it has just been provided from the database running on physical machine 3. As discussed later, this user data is typically represented by an *entity bean* which is a Java image of the database data, constructed on the basis of object-relational mapping. The EJB component then compares the password that the web tier provided to the one that the database contains.

- Assuming the password is correct, the EJB component returns a corresponding value to the caller from the web tier. The EJB component also sets a logged-in flag for the user by modifying the state of the entity bean.

- Now that the web tier knows the user is authenticated, it forwards the user request to a welcome page within the same web tier.

- The welcome page needs information about any items saved in the shopping cart of the user in previous sessions. The welcome page locates the corresponding EJB component (e.g., another session bean) and calls it.

- The EJB component retrieves the user's shopping cart from the database and presents the items it contains to the welcome page. This shopping cart data

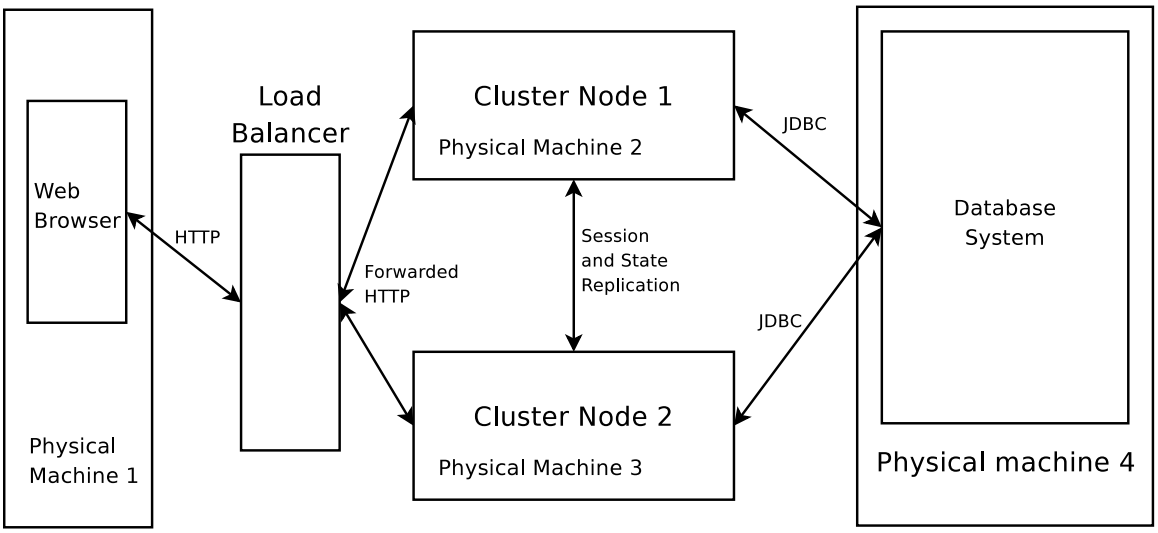

Figure 2.2: A clustered J2EE deployment scenario

itself is represented as an entity bean in the EJB container; this bean is different from, but related to the entity bean representing the user data.

- The welcome page populates itself with the data it just retrieved, and is sent to the user's browser as a response to the login request it sent initially.

- The web browser presents the welcome page to the user.

2.1.4 Deployment on a Simple Cluster

A more realistic deployment scenario is depicted in Figure 2.2. Physical machine 1 represents the client tier, with a web browser running. Physical machine 4 stands for the data tier, and contains a relational database management system. Physical machines 2 and 3 are deployed in a very simple J2EE cluster. The two cluster nodes are identical to each other, with each of them running a copy of the enterprise application in an application server. Each cluster node is running both the web tier and

the business tier of the application, and is effectively identical to physical machine 2 in Figure 2.1.

Clustering allows administrators to run an enterprise application on several parallel J2EE servers. Each of the J2EE servers is called a cluster node. The load is distributed across the nodes, so that even if one fails, the application is still accessible via the others. Clustering is crucial for J2EE, as it allows for *high availability*, *redundancy*, and *horizontal scalability* in terms of processing power.

In the scenario in Figure 2.2, both cluster nodes run a complete version of the middle tier of the enterprise application. Thus all calls between the web tier and the business tier are still local. A simple hardware load balancer is deployed between the client tier and the cluster hosting the middle tier. The load balancer is aware of the availability and the load of all cluster nodes, and routes client requests according to a pre-defined policy. A typical policy is to route requests to the cluster node that is least loaded and still available. Other policies, such as round-robin, are also common.

In a clustered server environment, *distributed state management* is a key service the cluster must provide. For instance, the session state of the web tier must be synchronized among all instances of session objects across all nodes, so that the client application reaches the same session state no matter which node serves the HTTP request. If a user puts an item in her shopping cart by pressing a button on a web page in the presentation layer, the same shopping cart containing that item should be available for the user to see and manipulate at any later point. State is automatically replicated by the clustering services, which are a part of each application server running in the cluster, according to a replication policy set by the administrators.

State must be synchronized at two levels: web tier and business tier. The state of the business tier is a complex mesh of EJB objects, and its replication among cluster nodes will be described later. The state of the web tier is contained in a few objects corresponding to a simple HTTP session, and its replication is relatively simple to execute. The information identifying a user (a user ID of some sort), as well as the user's current position in the control flow of the web tier, are propagated to all cluster nodes. They in turn locate and populate their corresponding session objects with it, instantiating new objects as necessary. Some administrators even deploy HTTP session-aware hardware load balancers, which route client requests to the web tier of the cluster node associated with that client. In that case, web tier state replication is unnecessary, because client sessions at the web tier level are typically self-contained and unrelated to each other. State replication of the business tier, however, is unavoidable because of the complexity typical of the business logic of enterprise applications.

An obvious problem with session-aware load balancers is the loss of high availability and redundancy at the web tier level. If a cluster node fails, user sessions associated with that node are lost. The business state of the objects associated with those users still exists and is persisted, but users have no way of accessing it unless they re-authenticate and start a new session. For these reasons, typically session-aware load balancers are deployed in conjunction with master-slave replication schemes for the web tier, where session state is synchronized with a backup server for each application server. If the front-end application server fails, the backup one is ready to take over its responsibilities.

Here is a short use case for the scenario presented in Figure 2.2:

- The user directs her browser to the address of the application. That address is resolved to the IP address of the load balancer.
- The load balancer forwards the request to one of the two cluster nodes. Suppose that is Node 1.
- Node 1 starts a new web session for the user. The generated unique user identifier is immediately propagated to Node 2, which starts a new web session corresponding to that identifier.
- The web tier of Node 1 is able to respond to the user request without invoking business tier functionality. It presents a login screen to the user's browser.
- The user fills in her username and password, and clicks the "Submit" button.
- The user's browser starts a new HTTP request, containing the username and password. The load balancer takes that request and routes it to one of the two cluster nodes. Suppose that is Node 2.
- The web tier of Node 2 locates the correct user session for that user. It then connects locally to the business tier, and forwards the received data for further processing.
- The business tier object responsible for user authentication takes the username and password, retrieves an EJB object (i.e., entity bean) that corresponds to the user data from the database, and matches the password. Suppose the password is correct.

- Now that the business tier of Node 2 contains an EJB object unique to a user (i.e., it carries state), this object is replicated to Node 1. The details of this replication will be described later.

- The business tier of Node 2 returns from the local call the web tier made against it, announcing successful authentication.

- The web tier sets a flag in the user session that states the user has logged in. The session state changes, and that change is propagated to the user session object in Node 1.

- The web tier returns a welcome page to the user's browser.

2.1.5 Completely Distributed Deployment

Figure 2.3 presents a completely distributed deployment scenario. The web tier and the business (EJB) tier are no longer co-located. Instead, each of them is deployed in its own cluster of physical machines – cluster 1 and cluster 2, respectively. Physical machine 1 is the client machine, running a web browser that connects to the web tier of the enterprise application. Physical machines 2 and 3 are the participants of cluster 1. Each of the two physical machines is running an application server on top of a JVM, but the EJB containers of those servers are inactive. Therefore, each of the two machines in cluster 1 is only hosting a copy of the web tier of the enterprise application. Physical machines 4 and 5 participate in cluster 2. Each of them is running an application server on top of a JVM, with only the EJB containers being active, together with their corresponding services. Physical machine 6 represents the data tier, and is running a relational database management system.

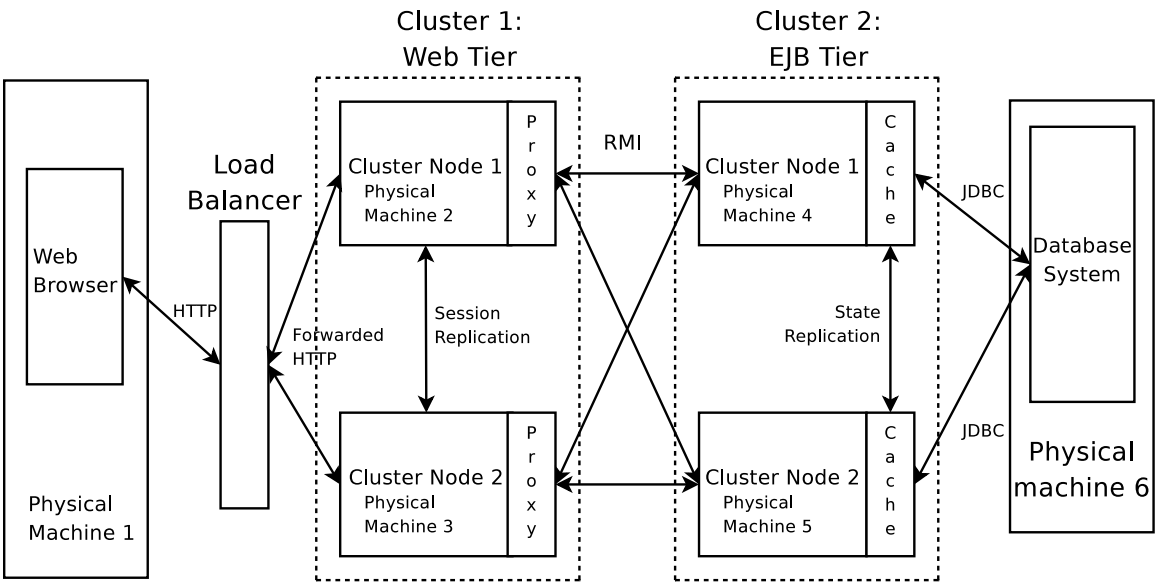

Figure 2.3: A complex J2EE deployment scenario

In this scenario the web tier and the business tier are no longer co-located. This leads to the additional complication that the web tier can only make remote calls to the business tier. Besides, there must exist a software or hardware system between the two clusters that routes those remote calls as necessary. In this case, the administrator has decided to utilize a client service that the application server provides, called *intelligent proxy*, instead of deploying yet another hardware load balancer. The web tier machines are natural choices for intelligent proxy deployment, as they effectively are clients to the business tier, and the administrator has complete control over them. What is more, RMI-aware hardware load balancers are exotic and expensive, while ordinary IP-based load balancers at this level would require yet additional services, for example distributed JNDI (Java Naming and Directory Interface).

The intelligent proxy is also called a client-side interceptor. It is a stub object, which is generated by the server, that implements the business interface of the service. The client obtains (i.e., looks up and downloads) such a stub from the server, and then makes local method calls against it in the RMI tradition. The call is automatically routed across the network and invoked against service objects managed in the server, i.e., the business tier. In a clustering environment such as the one described in Figure 2.3, the server-generated stub object is also an interceptor that understands how to route calls to nodes in cluster 2. The stub object figures out how to find the appropriate server node, marshal call parameters, unmarshal call results, and return the results to the caller client.

The stub interceptors have updated knowledge about the cluster. For instance, they know the IP addresses of all available server nodes, the algorithm to distribute load across nodes, and how to failover the request if the target node is not available. With every service request, the server node updates the stub interceptor with the latest changes in the cluster. For example, if a node drops out of the cluster, each of the client stub interceptors is updated with the new configuration the next time it connects to any active node in the cluster. All manipulations on the service stub are transparent to the client, i.e., the web tier.

Distributed state management at the level of the web tier, i.e., distributed session management, is handled by cluster 1 as described previously. Distributed state management at the level of the business tier is realized within cluster 2. State-carrying EJB objects are persisted to the database whenever necessary (this is a feature that is not exclusive to the sophisticated scenario being described, but is common among J2EE applications, regardless of the deployment scenario). The clustering services of

the EJB container, which every application server in cluster 2 is running, automatically synchronize the state of the business tier by announcing changes to local objects and persisting those objects, thus allowing all other nodes within the cluster to load the updated objects from the database and replace their stale local copies.

In a clustered business tier environment, the persistence service run by all application servers is typically upgraded to a *distributed caching environment*. Each application server has a local cache that is transparent to the application. The cache contains local copies of all state-carrying objects within the business tier, including entity beans and stateful session beans. The local cache also persists the objects to the database whenever necessary, usually according to some policy or flag. For example, when a session expires, the cache persists the entity bean that represents the shopping cart of that session, and then removes the entity bean from memory. Whenever the application changes its state, the local cache announces that change to the other members of the distributed cache (i.e., the other local caches in the cluster), which then update their state. Thus the cache serves two purposes simultaneously:

- It alleviates the network load by keeping local copies of objects, and making round trips to the database only when necessary.
- It replicates the state of its objects by understanding the object-relational mapping and communicating state changes to the other local caches without making round trips to the database.

A typical use case for the deployment scenario in Figure 2.3 is the following:

- The user directs her browser to the address of the application, which is resolved to the IP address of the load balancer. The load balancer routes the request to a node in the web tier cluster. Suppose that is Node 1, i.e., physical machine 2.

- Node 1 starts a new web session for the user. That session is propagated to Node 2. Node 1 returns the call, presenting a login screen to the user's browser.

- The user fills in her username and password, and clicks the "Submit" button.

- The browser makes a new HTTP request to the application. The load balancer forwards that request to Node 2 in Cluster 1. The web tier copy, running on physical machine 3, locates the user session.

- The web tier within physical machine 3 makes a local call to a session bean responsible for user authentication. That local call is intercepted by the intelligent proxy running in the web component of the application server.

- The intelligent proxy chooses an available and lightly loaded node within Cluster 2. Suppose that is Node 1 on physical machine 4.

- The intelligent proxy forwards the request, together with its parameters (username and password), to a remote session bean running as part of the business tier of the application on physical machine 4.

- The session bean makes a local call against the persistence mechanism, locating an entity bean object matching the username. That entity bean does not yet exist, because this is the first call for that username. The persistence mechanism, which is merged with the distributed cache, automatically connects to the database, retrieves the user data, creates an entity bean matching that user

data, and returns a reference to that entity bean to the session bean. The distributed cache keeps the local copy of the entity bean.

- The local cache announces the change of its state to the cluster. All other members of the distributed cache (the local cache on physical machine 5) update their state to match the change by contacting the local cache of physical machine 4, and retrieving the change.

- The session bean on physical machine 4 matches the password that came as a request parameter to the password contained in the entity bean object for the username. Suppose the match is exact.

- The session bean returns from the call, announcing that the authentication is successful.

- The web tier sets a flag in the user session that states the user has logged in. The session state changes, and the change is propagated to all other nodes in Cluster 1.

- The web tier returns a welcome page to the user's browser.

2.2 Challenges for Scalability

J2EE is undoubtedly a major contributor to the quick contemporary pace of enterprise application development. The J2EE specification describes various services in painstaking detail, and application servers implement those services as efficiently as possible. As a result, there is strict division of labor between application developers and server administrators. Developers are allowed to focus on the business logic of the

application and leave "glue" logic and unrelated concerns such as logging, persistence, security, caching, etc. to the services run by the application server. Server administrators, on the other hand, are charged with finding the optimal deployment scenario for a particular application, and configuring the respective application servers.

Despite the obvious advantages that J2EE provides, however, there is still significant room for improvement, especially in terms of horizontal scalability.

2.2.1 Horizontal Scalability

Horizontal scalability is one of the key goals underlying the progress of J2EE. Horizontal scalability is the ability to connect multiple hardware or software entities, such as servers, so that they work as a single logical unit. On the other hand, vertical scalability is the ability to increase capacity by adding resources to an existing entity. When discussing the benefits of scalability, usually *near-linear* scalability is implied, where the increase in beneficial metrics is ideally linearly dependent upon the number of additional entities. For example, to achieve linear vertical scalability in terms of processing power, an application would run twice as fast if the server CPU is upgraded to one that is twice as fast as the old one.

Vertical scalability is generally easy to achieve. Unfortunately, there are strict limits to its usefulness, determined by the available hardware. Horizontal scalability has a number of advantages to vertical scalability. First and foremost, it is potentially unbounded by hardware limits, because an administrator theoretically can always introduce additional physical machines in a cluster. Also, a horizontally scaled system has redundancy built in, allowing for high-availability server solutions. Therefore,

horizontal scalability in a stateful environment is an ideal, yet elusive, goal that many distributed architectures have tried to solve with varying success.

2.2.2 J2EE Horizontal Scalability

J2EE provides horizontal scalability by means of its clustering services. The distributed caching service provided by the EJB container, in particular, is key to its popularity as a distributed environment. There are scenarios, however, when J2EE does not scale very well horizontally, if at all.

Memory Scalability

From the examples discussed previously it has become clear that J2EE resembles a *shared memory architecture* built at the software (middleware) level. The state of a distributed enterprise application is propagated to all nodes within a cluster, no matter whether it is web tier state or business tier state. The memory footprint of all application servers within a cluster is effectively identical, barring slight configuration and hardware discrepancies (good administrators typically deploy physically identical machines as nodes in a cluster). It can be concluded that J2EE scales horizontally in terms of processing power only. *Memory in a J2EE application is not horizontally scalable at all, linearly or otherwise.*

Java Virtual Machines are notorious for their large memory consumption. J2EE application servers add several levels of indirection, typically implemented through the use of Java reflection and Java 5 annotations, between the JVM and the enterprise application. What is more, application servers usually run tens of services that are transparent to an application developer, but take their toll on the hardware. The

memory footprint of an application server running by itself, without any applications deployed, can easily exceed 512MB in a clustered environment.

There are also hardware-related memory limits. The obvious example is the 4GB limit of 32-bit architectures, which are still predominantly deployed in cluster configurations because of their low cost and general availability. The operating system reserves a portion of these 4GB for its kernel, and uses another portion for its services. That exacerbates the memory issue for enterprise applications. In total, the memory available to a Java enterprise application very rarely exceeds 3GB in contemporary deployment scenarios. If an application requires a larger amount of memory, no J2EE deployment can alleviate the issue as of the current state of art.

Network Scalability

Another significant area for improvement is network congestion in J2EE clusters. Because of the hard requirements for consistency typical for distributed environments and the corresponding algorithms (e.g., two- or three-phase locking), the number of maintenance RMI calls within a J2EE cluster generally exceeds the number of RMI calls that actually transfer data.

An additional problem pertinent to J2EE cluster deployments is the topology of the cluster. Many administrators use fully-connected clusters as the simplest and best choice, achieving state consistency throughout the cluster in the fastest possible way, as opposed to a hierarchical topology. This choice introduces an additional issue, however. In a fully-connected cluster the number of network messages rises quadratically with the number of nodes, which introduces a limit on the size of a cluster. In fact such a limit exists with any cluster topology, but with hierarchical topologies (e.g., star topology) it is reached more slowly. Consequently, J2EE applications do

not achieve near-linear horizontal scalability, especially when the cluster size increases beyond a practical limit.

2.3 Object-Level Lookup Service

J2EE applications cannot feasibly achieve horizontal scalability, especially in terms of memory. One solution to that problem that is relatively simple to understand, but very complex to design and implement, is to *horizontally partition the memory footprint of an enterprise application at the object level*.

Figure 2.4 represents a new design for distributing enterprise applications. The EJB tier is no longer a cluster. Rather, it consists of independent physical machines, each of them connected to an independent relational database management system. Each machine in the EJB tier is running the full enterprise application in terms of code, however it is only able to manipulate a *subset* of the total enterprise application data. Respectively, the database to which an EJB tier machine is connected contains the same subset. This design requires the deployment of several novel services that do not exist in contemporary systems.

The object lookup service bridges the gap between the web tier and the EJB tier. The current state of the art uses local or clustered Java Naming and Directory Interface (JNDI) services to bridge that gap. However, JNDI is only able to serve requests at the *class level*. If a client is to target different EJB tier machines, running the same application code, but manipulating different data, a JNDI tree is insufficient for lookups.

A possible use case for the scenario in Figure 2.4 is the following:

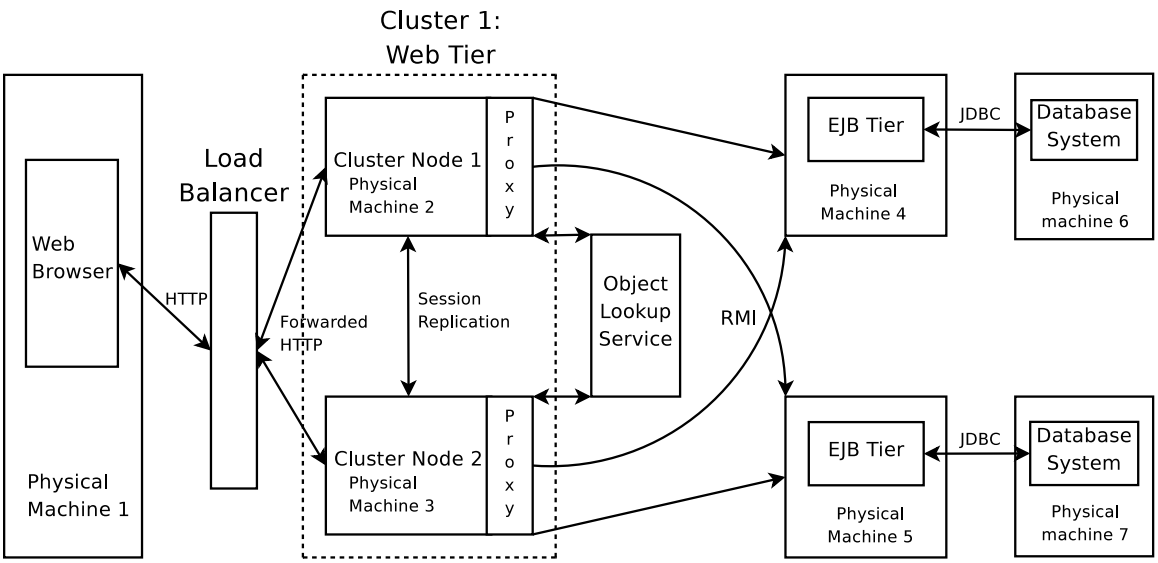

Figure 2.4: Proposed application architecture

- A user directs her browser to the web interface of the application. The load balancer routes the client request to an available machine in the web tier cluster. Suppose that is physical machine 2. That machine starts a new session for the user, propagates the session to all nodes within the web tier cluster, and returns a login page to the user.

- The user types in her username and password and clicks the "Submit" button. The corresponding client request is routed by the load balancer to an available machine in the web tier cluster. Suppose that is physical machine 3. That machine propagates the client request to a remote point within the EJB tier via the web tier proxy.

- The proxy investigates the client request and the data it carries. It then makes a lookup request against the object-level lookup service. That request carries

information regarding the session bean method the client request is targeted at, as well as the values of the input parameters to that method.

- The object-level lookup service returns the address of the EJB tier machine that contains the state corresponding to the client request. That state corresponds to a user account and other information, specific to the application. Suppose that machine is physical machine 4.

- The web tier proxy of physical machine 3 makes a remote call to the session bean running on physical machine 4, and propagates the client request.

- The session bean running on physical machine 4 executes login functionality against the data it received. It locates the user account, validates the username and password it received from the client tier, and responds positively to the login request. Note that if the request was incorrectly routed to physical machine 5, it would have been impossible to authenticate the user, because physical machine 5 contains no state regarding that user.

- Physical machine 3 sets a logged-in flag in the user session. That state is propagated to every machine in the web tier cluster. Physical machine 3 then presents the user with a welcome page.

- Any future requests relevant to that user's web tier session will be automatically routed by the proxy to physical machine 4, without any further interactions with the object lookup service.

The scenario depicted in Figure 2.4 looks complicated from a configuration point of view, and one might assume that deploying multiple applications on such an architecture simultaneously might pose problems. However, it is useful to remember that in real-world deployment scenarios there is a single J2EE application deployed and running on the entire architecture. This deployment choice eradicates any potential conflicts related to configuring the architecture to run multiple applications.

In addition, the proposed object level lookup service might seem a potential bottleneck. There are well-known solutions to this problem, however, and one needs to look no further than current implementations of JNDI. The fundamental difference between JNDI and the proposed service is the contents of the respective dictionary: JNDI responds to lookups at the class level, while the proposed service does so at the object level. They are similar from a deployment point of view, however, and it is conceivable that scalable implementations of this service will be easy to create based on existing JNDI designs.

Finally, Figure 2.4 shows no connection whatsoever between physical machine 4 and physical machine 5. This choice was made for simplicity of presentation. In a real-world scenario it will still be necessary to replicate some objects within the EJB tier, namely stateful session beans and a small subset of the entity beans: those that must serve as bridges between partitions of the data space.

2.4 Motivation for Proposed Dynamic Analyses

As mentioned previously, there is a strict division of roles in the J2EE culture. The one unifying interface is the J2EE specification that all participants must adhere to. The roles are:

- Application server developer. Participants in this role must build a framework that conforms to the J2EE specification and provides the services described in it. They are not interested in the particular scenarios in which the framework is going to be used afterward.

- J2EE application developer. Participants in this role build applications within the framework provided by the application server, assuming that everything promised by the J2EE specification is delivered. They are not interested in *how* the application server delivers that functionality, and are allowed to focus on the specific needs of their application.

- J2EE administrator. Participants in this role are charged with configuring an application server, so that it is an efficient container for a particular J2EE application. To that end, they must be aware of various specifics of the respective J2EE application (and, commonly, of the specific application server used in their organization), without having participated in its coding or having had any input in it.

The lookup service described above is a potential service provided by an application server that does not exist in the J2EE specification, but alleviates some inherent issues in the implementation of such servers. A general version of that service should be provided by application server developers as part of the application server. It is the job of J2EE administrators to later configure that service with the specific information it needs from the J2EE application deployed on it. J2EE administrators cannot influence the way the J2EE application is built. Our work targets extracting the necessary information from an already existing J2EE application, built according

to the specification, so that it becomes possible to migrate or deploy that application on the object-level lookup service.

A lookup service needs data against which lookups are made. Assuming that the mapping between data partitions and EJB tier machines is available, any lookup request coming from a client needs to provide data that identifies the partition of that data. Consequently, an intelligent client proxy must be available that is aware of the data that will flow between the client and the remote EJB object. Designing and implementing such a proxy is a challenging task, especially when starting with an existing application that is not horizontally scalable. In order to migrate such an application, it is necessary to be aware of the data that flows between the client tier and the EJB tier and uniquely identifies the EJB tier components necessary to satisfy a particular client request.

As an example, consider migrating an existing J2EE application to the architecture presented in Figure 2.4. The application has been built according to the J2EE specification, and respectively the application developers focused on the business logic, without much thought about later deployment. The application server developers, on the other hand, provided the object-level lookup service, the intelligent proxy, and other necessary services. Those services must be aware of the specifics of the particular application in order to function correctly, and respectively the application server developers have left the configuration of these services to whoever uses them.

It is now the job of J2EE administrators to take the application, which they did not write, and deploy it on the general services that the application server provides by configuring them with the specifics of the application. To do that, they need to extract the data those services need from the application. In the next chapter we

propose a dynamic analysis that identifies a significant portion of that data. This analysis is executed once, at application configuration time (using some existing test cases) and is not being continuously run in the background for the lifetime of the application.

CHAPTER 3

IDENTIFICATION OF ENTITY BEAN IDS

The proposed new design for distributed J2EE applications requires additional services, as described in the previous chapter. One such service is an enhancement to the already existing web tier proxy service. That service will need specific information about the J2EE application that is deployed on it. The purpose of this chapter and the next one is to describe dynamic analyses that extract that information from the application.

3.1 Intelligent Proxy

As a reminder, the web tier proxy service, as it exists currently, has the following functionality:

- Whenever a lookup is made from the web tier to obtain the handle of a remote object, the lookup passes through the intelligent proxy at the web tier level.
- The proxy determines the physical machine in the EJB tier against which to execute the request and, correspondingly, obtain the handle of a remote object. The identity of that machine is decided according to a pre-existing policy.

```
import javax.naming.InitialContext;
public class Client {
 public static void main(String[] args) throws Exception {
  InitialContext ctx = new InitialContext();
  Calculator calculator =
      (Calculator)ctx.lookup("CalculatorBean/remote");
  System.out.println("1 + 1 = " + calculator.add(1, 1));
  System.out.println("1 - 1 = " + calculator.subtract(1, 1));
 }
}

import javax.ejb.Stateless;
@Stateless
public class CalculatorBean implements Calculator {
 public int add(int x, int y) {
  return x + y;
 }
 public int subtract(int x, int y) {
  return x - y;
 }
}
```

Figure 3.1: A remote client connecting to a session bean.

- Once the handle is obtained, the web tier can execute requests against it. The proxy is not invoked further, unless another lookup is made.

A very simple example of a client connecting to a remote session bean, and executing requests against it, is presented in Figure 3.1. Class Client and class CalculatorBean are deployed on different physical machines. CalculatorBean is a stateless session bean, which is bound to the JNDI tree of its application server under the name `CalculatorBean/remote`. The local configuration of Client includes the addresses of at least some of the application servers in the EJB cluster. Each application server has a running high-availability JNDI service, containing bindings for locally deployed classes, as well as availability and load information for the other members of the cluster. When the main method of Client is executed, first an InitialContext object is

created. Such objects are used as front-ends to the web tier proxy. Then a lookup request is made against that object. The proxy standing behind it chooses one of the remote EJB tier machines based on their availability and load information, and returns a local handle (an RMI stub object) to an object of the type the client needs.

Note that Figure 3.1 presents a console client as opposed to the typical case of a client belonging to the web tier of the application. We chose a console client to simplify the example, and our choice has no impact on the lookup code described in the previous paragraph.

According to the above description, the client proxy is agnostic of the exact data the client will later pass over the network. The enhancement to the web tier proxy, or the additional service that the proxy will provide, requires that the proxy is aware of the identity and significance of some of the parameters the client will later feed to the remote EJB tier object. Commonly, at least one of those parameters is a unique identifier that is passed to the remote EJB object, which usually is a session bean, so that a corresponding entity bean is invoked. The intelligent web tier proxy should be aware of at least one such parameter for every lookup it services. If a call uses an already existing RMI connection, it typically does not pass unique identifiers as parameters, because those have been passed previously at the time when the connection was established. For calls that do need to establish a connection, the proxy takes the value of the unique identifier, and executes a lookup against the object-level lookup service that was described previously. The result of this lookup is the address of a physical machine that contains the desired data. The proxy then executes a lookup against that physical machine, knowing that the state it is looking for resides there,

```
@Stateless
public class CustomerDAOBean implements CustomerDAO {
  @PersistenceContext
  private EntityManager manager;

  public Customer find(int id) {
    return (Customer)manager.find(Customer.class, id);
  }
}
```

Figure 3.2: A stateless session bean.

and establishes an RMI connection that can be used afterward to access the same remote session bean object.

3.2 Dynamic Analysis for the EJB Tier

The service described previously relies on an intelligent lookup on the client's part. To do this lookup, the client must be aware of relevant parameters that will be passed later to the EJB tier. Identifying this data is non-trivial, and is the goal of the described dynamic analysis.

The state of the EJB tier depends either directly or transitively on the particular entity bean objects that are located in its memory space. In J2EE applications entity beans are fetched from the database based on their ID or primary key by calling a special service object, called an Entity Manager.

Figure 3.2 represents a very simple stateless session bean. Its only purpose is to return Customer objects, which are Entity beans, based on their unique identifiers. When the session bean is instantiated, an EntityManager is automatically associated with it through dependency injection. Later this EntityManager is used to fetch entity beans from the database through its `<T> find(Class<T>, Object pk)` method.

```
@Embeddable
public class CustomerPK implements java.io.Serializable {
  private long id;
  private String name;
  public CustomerPK(long id, String name) {
    this.id = id;
    this.name = name;
  }
  ...
  public int hashCode() {return (int) id + name.hashCode();}
  public boolean equals(Object obj) {
    if (obj == this) return true;
    if (!(obj instanceof CustomerPK)) return false;
    if (obj == null) return false;
    CustomerPK pk = (CustomerPK) obj;
    return pk.id == id && pk.name.equals(name);
  }
}
```

Figure 3.3: A composite primary key.

The primary keys of entity beans are usually of primitive types, as in the above example. This does not always have to be the case. The EJB 3.0 specification defines the notion of a *composite primary key*, which is a Java bean object containing two or more primitive type fields. Such a composite primary key may be passed as the second parameter of the `EntityManager.find` method. It typically redefines its `hashCode` and `equals` methods so that any comparisons the EntityManager makes are consistent with the expected behavior. Figure 3.3 shows an example of a composite primary key object.

As discussed previously, a client obtains a remote reference to a session bean, which provides methods to handle the business logic the client needs. The session bean object works with entity bean objects, which contain the application data, by either obtaining their references from an EntityManager, or utilizing previously obtained such references. The client typically passes the identifiers of the entity bean

objects to the session bean, so that the session bean may fetch them as needed. The common use cases for such passing are:

- The client may directly pass a primitive-type identifier to the session bean as one of the parameters of a remote call.
- The client may directly pass a composite primary key object to the session bean as one of the parameters of a remote call. The composite object is serialized and passed over the wire.
- The client may pass the data that comprises a composite primary key object as two or more of the parameters of a remote call. The session bean instantiates the composite object.
- The client may pass the data that comprises a composite primary key object as parameters of two or more remote calls. The session bean holds the data until it is ready to instantiate the composite object.

Our dynamic analysis handles all of the use cases presented above. Its aim is to match input parameters for session bean methods with actual parameters passed to an `EntityManager.find` method. Thus, the dynamic analysis has well-defined entry points (a remote invocation of a session bean method), and a well-defined exit point (a call to `EntityManager.find`). The dynamic analysis covers typical cases, which have the following properties:

- The values that comprise an entity bean's primary key are never modified within the EJB tier.

- There are no exceptions at the EJB tier level. This assumption is made for the purpose of simplifying the algorithm. A generalization of the algorithm that handles EJB tier exceptions within the enterprise application code should be possible to design using the approach from [57].

In order to match input parameters to primary keys, the dynamic analysis tracks the flow of the values of the input parameters. Whenever a value that originated in an input parameter is certain to be an entity bean's primary key, or a part of one, the dynamic analysis outputs a match between the input parameter and the respective primary key.

3.2.1 Abstract Algorithm

The algorithm that comprises the dynamic analysis is based on code instrumentation. Whenever an assignment, i.e., a passing of a value without any modification, is about to take place within the EJB tier, the dynamic analysis intercepts the participants in that assignment. This information is used to update a data structure referred to as a *value flow graph* (VFG).

The abstract version of the algorithm, then, proceeds as follows:

- Whenever a remote call is made against a session bean object, intercept the formal parameters to the call, and remember information that uniquely identifies the parameters in their corresponding VFG nodes.

- Whenever an assignment takes place within the EJB tier, intercept the right-hand side of the assignment (both the memory location and the uniquely identifying information related to it), and look up that information in the VFG.

- If a location corresponding to the right-hand side of the assignment already exists in the VFG, then it contains a pointer to its origin, i.e., it knows the session bean method and parameter number that its value came from. Create an association between the location corresponding to the left-hand side of the assignment, and the origin of the value.

- Such an association is introduced even if the left-hand side does not yet correspond to a location within the VFG. In that case, the corresponding location is created.

- If a location corresponding to the left-hand side already exists in the VFG, it already has a pointer to an origin. In that case, the association modifies the pointer of that location, pointing it to the origin of the location corresponding to the right-hand side of the assignment.

- If an `EntityManager.find` method is reached, output the association between the second parameter to that method, and its origin. This association relates a remote value passed to a session bean method and an entity bean's primary key, which was the purpose of the algorithm.

This abstract algorithm presents a high-level picture of the necessary run-time processing; the following sections refine this picture and introduce additional details.

3.2.2 Code Instrumentation

The dynamic analysis relies on instrumented code to make callbacks into dynamic analysis functions. Code instrumentation is executed on the Jimple intermediate representation for analyzing and transforming Java bytecode provided by the Soot

framework [64]. Jimple is a typed 3-address intermediate representation, and is particularly useful for the purposes of the dynamic analysis, because it makes explicit all assignments encountered in the Java code. This includes, among other things, the use of reference `this` to the receiver object within a method, and the use of method parameters.

The following statements are instrumented:

- Direct assignment: $\texttt{v}_1$ `=` $\texttt{v}_2$
- Instance field write: $\texttt{v}_1$`.f =` $\texttt{v}_2$
- Instance field read: $\texttt{v}_1$ `=` $\texttt{v}_2$`.f`
- Static field write: `X.f = v`
- Static field read: `v = X.f`
- Static invocation: `w = X.m(`$\texttt{v}_1$`,...,`$\texttt{v}_k$`)` or `X.m(`$\texttt{v}_1$`,...,`$\texttt{v}_k$`)`
- Instance invocation: `w =` $\texttt{v}_0$`.m(`$\texttt{v}_1$`,...,`$\texttt{v}_k$`)` or $\texttt{v}_0$`.m(`$\texttt{v}_1$`,...,`$\texttt{v}_k$`)`
- Return: `return v` or `return`

In the above statements, $\texttt{v}_i$ denotes a local variable or a formal parameter (including `this`). Additional instrumentation is also necessary for the entry and exit points of the analysis: the exit points are invocations of `EntityManager.find` methods, and the entry points are at the beginning of the bodies of all remotely accessible methods of session beans.

Our experience shows that J2EE application programmers often wrap primitive-type entity bean IDs in their respective wrapper types — specifically, the common case

is when `int` parameters are passed to `Integer` constructors, and then the `Integer` object is passed to `EntityManager.find`. We treat boxings of `int` values as assignments, and instrument them accordingly. For example, for an assignment `x = new Integer(id)`, a node for `x` is created in the VFG and the origin for this node is set to the origin for `id`.

In summary, the following run-time events are instrumented:

- Assignment
- Method invocation
- Boxing of primitive types
- Method exit
- Source — that is, the entry point of a session bean method called remotely
- Sink — that is, a call to `EntityManager.find`

3.2.3 Data Structure

It is clear from the above discussion that the VFG is a set of nodes, with information associated with them. Those nodes represent either heap locations, corresponding to static and instance fields, or local variables within methods. The relationships (arcs) between the nodes represent the origins of the values in the heap locations or local variables represented by the nodes. Some nodes (root nodes) stand for an input parameter to a remote method invocation for a session bean. Those nodes have no origin relationship, i.e., their origin pointer is effectively `null`. Any other node contains an origin pointer, pointing to one of the root nodes.

Each node within the VFG must be uniquely identifiable, so that the analysis can relate left- and right-hand sides of assignments to already existing nodes. The unique identifier differs in structure for the two types of nodes. A node effectively contains a single reference to its origin, and a unique identifier, which may consist of multiple pieces of data.

The first node type, which corresponds to static and instance fields, has the following structure:

- Reference to an origin; `null` if the node is a root.
- The name of the corresponding field.
- The fully qualified name of the class that contains the static field, or the class whose instance contains the instance field.
- The identity hash code of the object that contains the instance field; `null` if the field is static.

The second node type, which corresponds to local variables (including formal parameters) within methods, has the following structure:

- Reference to an origin; `null` if the node is a root.
- The name of the corresponding local variable.
- The unique thread ID of the thread running the method containing the local variable.
- The stack frame depth of the method containing the local variable in the running thread. The role of this value is to distinguish between local variables in different stack frames.

Note that both node types ignore the actual value of the field or variable. The purpose of the analysis is to associate an entry point (parameter of a session bean remote method invocation) with an exit point (invocation of an `EntityManager.find` method). The value that actually flows through the assignments, be it a primitive type or a reference to a composite primary key, brings no additional useful information to the analysis.

The root nodes, which always correspond to formal parameters, additionally contain information identifying uniquely the session bean class, method, and formal parameter that served as an entry point.

3.2.4 Concrete Algorithm

This section presents a concrete version of the algorithm for the dynamic analysis, building on the details presented earlier. Whenever an assignment of the types discussed previously is executed, the analysis intercepts the assignment operation, as well as its left- and right-hand side. Whenever a method invocation is encountered, the analysis creates nodes of the local variable type. The number of those nodes is the same as the number of the formal parameters of the called method. (An exception is the `@this` formal parameter: no node is created for it, and assignments with its participation are not instrumented, because they cannot carry a value that is interesting to the analysis.) The thread ID of those nodes is the current thread ID, and the stack depth is the current stack depth plus one. The origins of those nodes are set to the origins of the nodes for the actual parameters at the call site. If the method invocation is also an assignment, i.e., the method has a return value, an additional local variable node is created for the left-hand side local at the call site (if a node for

this local does not already exist). That node has no origin (yet), its thread ID is the current one, and its stack depth is the current one.

Consider an assignment whose left-hand side is a local variable. The following cases exist for the right-hand side:

- It is a constant. This case is silently skipped.
- It is a local variable. In those cases, the analysis searches the VFG and finds the node corresponding to the right-hand local, based on the variable name, the current thread ID, and the current stack depth. The analysis then searches for a node corresponding to the left-hand side local, creating and populating one if it does not exist. Then its origin is set to point to the origin of the node corresponding to the right-hand local.
- It is a formal parameter of the current method. The analysis searches the VFG for the node corresponding to that formal, which has been created just before the method invocation. Then it proceeds with the left-hand side as discussed in the previous bullet.

For instance field writes, instance field reads, static field writes, and static field reads, the procedure is similar. The analysis finds the node corresponding to the right-hand side, no matter whether it is a local variable or a field. It then finds, or creates and populates, a node corresponding to the left-hand side, and sets its origin to the origin of the right-hand side node.

When a return statement is encountered, the following functionality takes place:

- If the value of the return statement is a variable, the analysis searches the VFG for a node corresponding to the left-hand side local variable at the caller. That

node contains the following information: it has no origin, its thread ID is the current one, and its stack depth is the current one minus one. When such a node is found, its origin is set to the origin of the local variable node corresponding to the value of the return statement.

- The analysis proceeds to remove from the VFG all local variable nodes that correspond to the method being left, i.e., all local variable nodes whose thread ID is the current one, and whose stack depth is the current one.

Whenever an entry point is encountered, i.e., the start to a remotely-invoked session bean method, the analysis creates root nodes corresponding to the formal parameters of the method. The root nodes have no origins. Whenever an exit point is reached, i.e., an invocation of an `EntityManager.find` method, the node corresponding to the primary key actual parameter of that method is captured. The analysis outputs the association between that node and its origin.

Clearly, the analysis simulates the run-time call stack of application code by increasing a counter for every method entry event, and decreasing the counter for every method exit event. Such a simulation is problematic in the presence of static initializers and finalizers, because they may be invoked by the JVM in a way that violates the proper ordering of entry/exit events. If such invocations happen during the algorithm, associations involving origins of primary keys may be incorrectly omitted or incorrectly introduced. While the algorithm could potentially be extended to identify such cases, this is beyond the scope of our current work. The experimental results presented later indicate that such cases did not influence the precision of the analysis implementation for our specific test cases. We manually traced all results and ensured that every output produced by our implementation was correct, and every

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
  @PersistenceContext
  private EntityManager manager;
  private int orderID;
  private Order order;

  public void setOrder(int id) { orderID = id;}
  public void buy(String product) {
    if (order == null)
      order = manager.find(order.class, id);
    order.addPurchase(product);
  }
  ...
}
```

Figure 3.4: A stateful session bean.

false negative was due to reasons other than out-of-order events such as invocations of finalizers or static initializers.

3.2.5 Example

Figure 3.4 presents an example of an integer primary key passed to a stateful session bean. The session bean does not immediately instantiate the entity bean related to it. Rather, it keeps the value in field `orderID` until it is forced to invoke method `addPurchase` in method `buy`. At that time, it asks the EntityManager for the entity bean it needs. The Jimple code of the two methods of the session bean is shown in Figure 3.5.

The instrumentation has placed a hook to a dynamic analysis method before each assignment in the Jimple code[1]. The algorithm proceeds as follows:

[1]Strictly speaking, the initial n assignments of the form `local := parameter`$_i$, where n is the number of formal parameters, represent parameter passing and instrumentation cannot be inserted before them. In these cases, we insert our instrumentation at the start of the actual method body, i.e., after those assignments, but before any other code is executed.

```
public void setOrder(int) {
  ShoppingCartBean r0;
  int i0;
  r0 := @this;
  i0 := @parameter0;
  r0.orderID = i0;
  return;
}
public void buy(java.lang.String) {
  java.lang.Class r3;
  Order r2;
  EntityManager r1;
  ShoppingCartBean r0;
  int i1;
  java.lang.String i0;
  r0 := @this;
  i0 := @parameter0;
  r2 = r0.order;
  if r2 != null goto label0;
  r1 = r0.manager;
  r3 = r2.class;
  i1 = r0.orderID;
  r2 = r1.find(r3, i1);
label0:
  r2.addPurchase(i0);
  return;
}
```

Figure 3.5: Jimple representation.

- A remote client calls method `setOrder` of the session bean with an integer parameter. The analysis catches the entry to the method and instantiates a root node corresponding to the first formal parameter of the method.
- The analysis does nothing for the first assignment (`r0 := @this`).
- For the second assignment (`i0 := @parameter0`), the analysis finds the node corresponding to `@parameter0` of the currently executing method, creates a new node corresponding to variable `i0` of the currently executing method, and sets the origin of that node to the node of `@parameter0`.

- For the third assignment (`r0.orderID = i0`), the analysis finds the node corresponding to `i0`, creates a new node corresponding to the field named `orderID` of the object with identity hash code `System.identityHashCode(r0)`, and sets its origin to the origin of `i0`, i.e., `@parameter0`.

- The analysis intercepts the empty return statement, and deletes the node corresponding to local variable `i0`.

- Some time later, a remote client calls method `buy(String)`. The analysis catches the entry to the method and instantiates a root node corresponding to the String formal parameter of the method.

- The analysis does nothing for the first assignment (`r0 := @this`).

- The analysis proceeds to create a node corresponding to the left-hand side of the second assignment (`i0 := @parameter0`), and points its origin reference to the node corresponding to `@parameter0` of `buy`.

- The next three assignments (`r2 = r0.order; r1 = r0.manager; r3 = r2.class;`) are skipped, as there are no nodes corresponding to their right-hand sides.

- For the next assignment (`i1 = r0.orderID`), the analysis creates a new node corresponding to `i1`, and points its origin reference to the origin of the already existing node corresponding to the `orderID` field of the current object. As a reminder, that node was created as a result of the previous call to `setOrder`. The result is a node corresponding to `i1` with an origin pointer to the `@parameter0` node for `setOrder`.

- The following method call is intercepted (`r2 = r1.find(r3, i1)`). This is a call to `EntityManager.find`. The analysis looks at the node corresponding to the second actual parameter of that call, and outputs a match between the first formal parameter of `setOrder` and a primary key of an entity bean.

The end result of the analysis in this example, assuming that there is no interesting functionality within `Order.addPurchase`, is a match between an entry parameter to the EJB tier and a primary key of an entity bean.

3.2.6 Handling of Composite Primary Keys

The above algorithm matches formal parameters if remotely-invoked session bean methods to entity bean primary keys, as long as the primary keys are of a primitive type, or are composites that are passed directly from the client tier as DTOs. With an alteration, the algorithm also associates the components of a composite primary key with their originators at the EJB tier level, no matter whether those components are passed together through an invocation of the same session bean method, or they originate from different session bean remote methods.

The necessary alteration is an additional exit case. The Jimple representation is instrumented at the instantiations of composite primary key objects. Those instantiations are trivial to find, because composite primary keys are annotated as such, as required by the J2EE component model. Whenever such an instantiation happens, the analysis captures the nodes corresponding to all actual parameters to the constructor of the composite primary key, and outputs the associations between those parameters and their origins.

3.2.7 Primary Keys in Arrays

Another enhancement to the algorithm allows tracking primary keys which were passed from the client tier as members of an array. Since array accesses do not differ significantly from other assignment statements, it was simple to instrument Jimple assignments featuring arrays, as well as augment the already existing VFG to support array-based memory locations. Since Java does not allow pointer arithmetic, the only way an array cell could possibly be modified is through an array reference. Respectively, yet another type of node is introduced in the VFG. Nodes of that type are very similar to nodes carrying information about object fields; the only difference is that, instead of the field name, such nodes carry an integer, which is the array index corresponding to that node. Thus an array is represented in the VFG as a set of nodes. The size of that set is the array length. Since none of the test applications utilized arrays containing primary keys, this functionality of the algorithm was not evaluated.

3.2.8 EJBCA Example

Figures 3.6 and 3.7 present a simplified example of an EJBCA use case, which the analysis encountered and successfully evaluated in our experimental evaluation. The two methods presented are located within a single session bean class, `LocalRaAdminSessionBean`. When a client invokes the first method in the figure, it passes several parameters to the EJB tier: an Admin object, an int value, a String value, and an EndEntityProfile object. The VFG is populated with four new nodes, holding data about the four parameters. The relevant state of the VFG is shown in Figure 3.8.

```
public void addEndEntityProfile(org.ejbca.core.model.log.Admin, int,
    java.lang.String, org.ejbca.core.model.ra.raadmin.EndEntityProfile)
    throws org.ejbca.core.model.ra.raadmin.EndEntityProfileExistsException {
        org.ejbca.core.ejb.ra.raadmin.LocalRaAdminSessionBean r0;
        ...
        int i0, $i1, $i2, $i3, $i4, $i5;
        ...
        boolean $z0, $z1;
        ...
        i0 := @parameter1: int;
        ...
        $z1 = specialinvoke r0.<org.ejbca.core.ejb.ra.raadmin.
          LocalRaAdminSessionBean: boolean isFreeEndEntityProfileId(int)>(i0);
        ...
}
```

Figure 3.6: EJBCA example, part 1.

The second parameter to the method, denoted by `@parameter0`, is assigned to local variable `i0`. Respectively, a new VFG node is created to hold that information.

After the execution of some code irrelevant to this particular use case, the execution trace reaches the method call noted on Figure 3.6. A new VFG node is created whose origin is the origin of the `i0` node, or the second parameter of the `addEndEntityProfile` method. This new node's name is `@parameter0`, with stack depth equal to the current one plus one.

When `isFreeEndEntityProfileId` is entered (Jimple code on Figure 3.7), the node just created is immediately used. Because the value of `@parameter0` is assigned to local variable `i0`, as per the Jimple code, yet another node is created in the VFG. Its name is, naturally, `i0`; its stack depth is the current one, which is one in this case; and its origin is the origin of the value assigned to it, which is root node `@parameter0` with stack depth 0.

```
private boolean isFreeEndEntityProfileId(int) {
        org.ejbca.core.ejb.ra.raadmin.LocalRaAdminSessionBean r0;
        int i0;
        boolean z0;
        org.ejbca.core.ejb.ra.raadmin.EndEntityProfileDataLocalHome $r2;
        java.lang.Integer $r3;
        r0 := @this: org.ejbca.core.ejb.ra.raadmin.LocalRaAdminSessionBean;
        i0 := @parameter0: int;
        z0 = 0;
     label0:
        if i0 <= 1 goto label1;
        $r2 = r0.<org.ejbca.core.ejb.ra.raadmin.LocalRaAdminSessionBean:
          org.ejbca.core.ejb.ra.raadmin.EndEntityProfileDataLocalHome
          profiledatahome>;
        $r3 = new java.lang.Integer;
        specialinvoke $r3.<java.lang.Integer: void <init>(int)>(i0);
        interfaceinvoke $r2.<org.ejbca.core.ejb.ra.raadmin.
          EndEntityProfileDataLocalHome: org.ejbca.core.ejb.ra.raadmin.
          EndEntityProfileDataLocal findByPrimaryKey(java.lang.Integer)>($r3);
     label1:
        ...
        return z0;
}
```

Figure 3.7: EJBCA example, part2.

The next relevant event is the creation of a new `Integer` object which wraps the value of local variable `i0`. Respectively, a new node is created in the VFG. The name of the node is `$r3`, which is the local variable name in Jimple for the reference to the `Integer` object. The stack depth is one, and the origin is root node `@parameter0`.

The execution then proceeds to make the call to `findByPrimaryKey` with local variable `$r3`. The analysis captures that information, finds the origin of that node, and outputs that the second formal parameter of `addEndEntityProfile` is an entity bean primary key. The relevant VFG state just before the call to `findByPrimaryKey` is depicted in Figure 3.9.

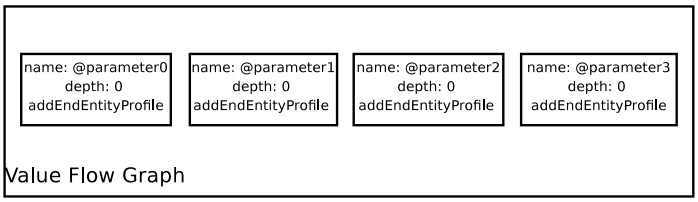

Figure 3.8: Initial VFG state

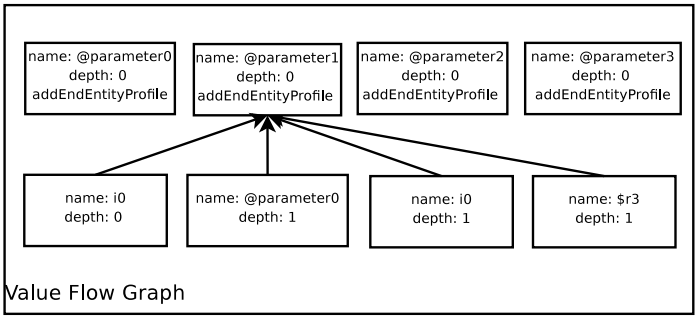

Figure 3.9: Final VFG state

3.3 Finer-Grain Tracking of Values

The algorithm as described previously is concerned with detecting the entry points of primary keys to the EJB tier. It tracks assignments within the EJB tier, setting origin pointers of VFG nodes to the respective entry nodes of the values. This results in losing the propagation chain of those values. The analysis keeps the distinction between local variable nodes in multiple calls to the same method by simulating a call stack, and assigning a stack depth value to each local variable node. When a method is exited, all local variable nodes for that method invocation are deleted, because at that moment they become irrelevant to the intended output of the analysis. That

```
public void addEndEntityProfile(org.ejbca.core.model.log.Admin, int, int,
    java.lang.String, org.ejbca.core.model.ra.raadmin.EndEntityProfile)
    throws org.ejbca.core.model.ra.raadmin.EndEntityProfileExistsException {
        org.ejbca.core.ejb.ra.raadmin.LocalRaAdminSessionBean r0;
        ...
        int i0, i1, $i2, $i3, $i4, $i5;
        ...
        boolean $z0, $z1;
        ...
        i0 := @parameter1: int;
        i1 := @parameter2: int;
        ...
        $z0 = specialinvoke r0.<org.ejbca.core.ejb.ra.raadmin.
          LocalRaAdminSessionBean: boolean isFreeEndEntityProfileId(int)>(i0);
        $z1 = specialinvoke r0.<org.ejbca.core.ejb.ra.raadmin.
          LocalRaAdminSessionBean: boolean isFreeEndEntityProfileId(int)>(i1);
        ...
}
```

Figure 3.10: Augmented EJBCA example

output is a list of session bean methods and their formal parameters that have been identified as primary keys.

A programmer may want to have more information regarding the values of primary keys, i.e., the chain of assignments and conversions that the values flowed through in order to reach the `findByPrimaryKey` call. Such information improves program comprehension and may be useful for testing, debugging, and maintenance tasks.

As an example, consider an extension of the code from Figure 3.6, presented in Figure 3.10. The addition is a new integer parameter to the method, which is used to call the method from Figure 3.7 for the second time. The programmer may be interested in the propagation of values, so a desirable piece of information for every assignment is the signature of the method whose local variable participates in the assignment. In addition, the information regarding value propagation should not be removed from the graph, meaning that local variable nodes should not be deleted at

method exit. This necessitates having three additional pieces of information for each variable node:

- A flag that says whether the node is "alive", signifying whether it may participate in future assignments, or just exists in the graph simply for the purpose of recording the propagation chains. There must be no more than one live node with the same uniquely identifying information.
- The full signature of the method containing the declaration of the variable (and, by extension, any assignments with its participation).
- A counter distinguishing between multiple calls of the same method. This counter must be different from the stack depth counter that already exists.

In addition, the origin pointer of a node is replaced with a parent pointer, which points to the node's immediate predecessor.

The method signature and the method call counter, which were not included in the previous version of the VFG, together may safely substitute the stack depth counter. As a reminder, the function of the stack depth counter is to distinguish between multiple local variables with the same name somewhere along a call chain. The above two additional pieces of information combine to form a method-specific invocation counter that serves the same purpose, and in addition distinguishes between local variables with the same name and the same stack depth, as Figure 3.10 shows. The latter is a complication introduced by the fact that local variable nodes are not removed from the VFG at method exit in this version of the algorithm.

Figure 3.11 represents the state of the VFG at the end of the second call to `isFreeEntityProfileId` if additional information about value propagation is desired.

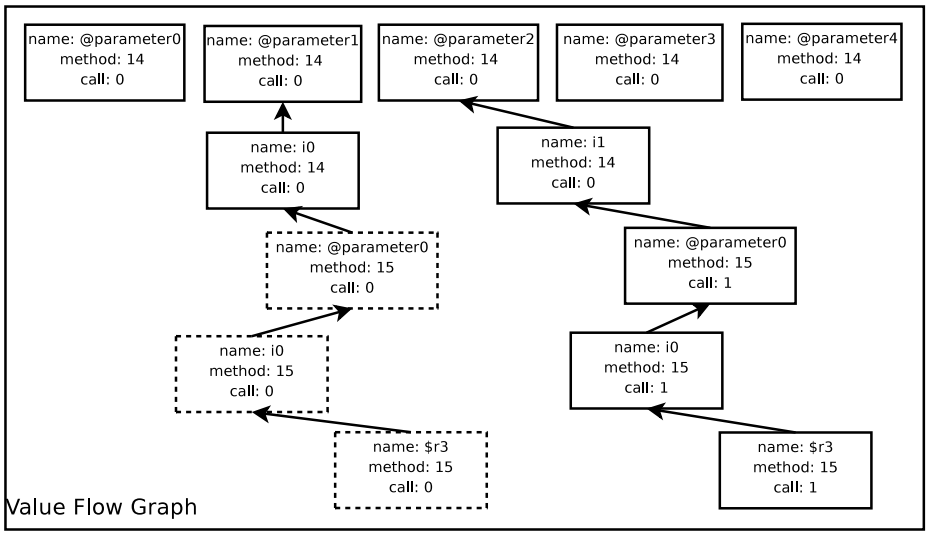

Figure 3.11: Final VFG state with additional information

Nodes marked with a solid line are tagged as live, while ones marked with a dotted line are tagged as dead and cannot participate in further assignments. For the sake of brevity, it has been assumed in the figure that the method represented in Figure 3.10 has a method signature internally identified by 14, while the corresponding number for the method in Figure 3.7 is 15. That final state can readily be output from the algorithm, resulting in a sequence of nodes for each primary key that uniquely identifies the propagation chain of its value.

To construct the VFG presented in Figure 3.11 the algorithm presented in the previous section must be changed as follows.

When the first method call in Figure 3.11 is reached, a new node is created in the VFG. The node corresponds to the first formal parameter of the called method. The method signature is indicated by 15, and the global execution counter for this method is 0, since this is its first execution. The parent pointer of that node is

the local node representing `i0` of the 0th execution of `addEndEntityProfile`. As a reminder, the method signature and the execution counter together form a method-specific call counter. The execution counter represents the successive instances of the respective method, e.g., nodes with `method:15,call:1` represent local variables in the 1st run-time instance of the method with signature 15. Instances are counted from zero.

Once that method is entered (Figure 3.7), local node `i0` is assigned the value of its first formal parameter. A lookup is made against the VFG, searching for a live local node variable with name `@parameter0` for the 0th execution of method 15. A new node is created, corresponding to local variable `i0`. If such a node already exists (meaning that this local variable has already been assigned a value), that node is tagged as dead. It remains in the VFG, however, for the sake of any chains in which it participates. The parent pointer of the new node is set to the `@parameter0` node. A similar functionality takes place when the value of `i0` is boxed in an `Integer` object, whose address is stored in local variable `$r3`. Then, when the `findByPrimaryKey` method is called, the entire chain of nodes related to `$r3` can be produced as output.

When the method exits, all its local variables are tagged as dead: they may not be returned from lookups any more. However, they remain in the VFG, because they may have taken part in chains with the participation of object fields, return statements, or in general chains that may later lead to a `findByPrimaryKey` method, even though this is not the case in this example.

Back in Figure 3.11 the next line of code is executed. It is another call to `isFreeEntityProfileId`, this time with a different parameter. The execution counter

for that method is increased to 1, and a new node is created in the VFG corresponding to `@parameter0` of that method. Afterward the algorithm proceeds as described above.

In general, every assignment creates a new VFG node for the left-hand side. Any old nodes having the same identifying information as the new one are marked as dead, but are not deleted from the VFG, so that any chains with their participation may be retrieved later. This happens without regard for the node type. This, of course, is necessary because a local variable or an object field may participate in the propagation of a primary key, but later may be reassigned with another value. It must still be possible to retrieve the original chain despite this reassignment.

3.4 Experimental Study

The dynamic analysis was evaluated on three separate Enterprise Java applications, running on the JBoss Application Server version 4.0.5 [31]. The largest and most complicated application used for the evaluation was EJBCA version 3.4.1 [22]. The reasons for our choice of this application were its realistic size (635 classes), robustness, source code availability under the Lesser General Public License, and the fact that it is an industrial-strength enterprise application that has been successfully deployed on a number of web sites. The two smaller applications were Duke's Bank [21] and Pet Store [30]. We used HSQLDB, the native Java database engine that is bundled with JBoss, for the database layer. The machine used for experimental evaluation had an AMD Athlon XP 2200+ CPU with 1 GB RAM. The operating system was Ubuntu 7.04 GNU/Linux. JBoss and HSQLDB, together with the Enterprise Java applications deployed on them, were run through Sun's Java 6 JVM.

For the evaluation we used applications built for J2EE up to version 1.4 (corresponding to EJB version up to 2.1). While there are significant differences between the latest Enterprise Java version (1.5, corresponding to EJB version 3.0) and J2EE 1.4, those differences are mostly irrelevant to the dynamic analyses described in this work. The relevant difference is the lack of an EntityManager object in J2EE 1.4. The method used to return primary keys of entity beans is a member method of the corresponding entity bean, with signature `Object findByPrimaryKey(Object id)`. This difference necessitated a trivial alteration of the algorithm.

We implemented two separate versions of the algorithm. The first version uses bytecode instrumentation to call functionality on the VFG from within the instrumented code. That is, whenever an interesting event (an entry point, an assignment of the kinds described previously, a method invocation, a value-carrying return statement, or an ending point) happens, there will also be a static invocation of a gateway to the VFG, passing all relevant data to it. This is the online version of the algorithm. We also implemented an offline version which uses bytecode instrumentation of all interesting events to simply dump all relevant data to a file. Of course, the bytecode instrumentation for the offline version is the same as the one for the online version. After the testing run ends, the trace is processed with a separate tool, which invokes the VFG functionality as necessary. A final implementational point of interest is that we did not track thread IDs. This simplification was made possible by the fact that all tests performed used single-threaded test suites.

3.4.1 Analysis Precision

For the experimental evaluation of EJBCA we utilized its own testing suite. It is run remotely (from a separate JVM), and connects to the running instance of the enterprise application directly through RMI, meaning that it bypasses the Web tier of the application. To evaluate the other two applications we manually tested their functionality through a browser, connected to their Web tiers.

Analysis precision is determined as follows: let n be the total number of calls to `findByPrimaryKey` methods, and let m be the total number of *matched* such calls, where a matched call is defined as a call whose parameter origin is known (i.e., the analysis has output a match between an input parameter to the EJB tier and this particular method invocation). Then the number of *unmatched* calls is, naturally, $n - m$. In the best possible case the parameter to every single call to `findByPrimaryKey` would originate at the EJB tier entrance. Unfortunately, in some tests there were calls to `findByPrimaryKey` whose parameters did not originate at that entrance, and we had to manually inspect the relevant source code in order to find the origin of the parameter.

Therefore, an unmatched call could be caused by one of the following:

- A deficiency in the algorithm, where the assignment chain between a value flowing in the EJB tier and the value leaving the EJB tier is not tracked precisely,
- Or a rare use case where the value leaving the EJB tier originates within the EJB tier.

There were a total of 152 invocations of `findByPrimaryKey` methods in EJBCA. Our analysis matched 141 of the invocations, meaning that there were 11 unmatched

ones. An inspection of the execution trace and EJBCA's source code showed that three of those unmatched invocations were instances of two separate method calls, which use a pre-configured constant value (which turned out to be `0` and `"0"` respectively) to fetch some kind of a "base" entity bean of a particular type. In short, this value was constant, and it originated within the EJB tier.

The other unmatched invocations were instances of method calls that take a String value. This String value is a manipulation of another String value, passed by a client tier to the EJB tier. The algorithm could not match the entry point to the exit point in this case because there was a manipulation of the value other than assignment.

On the whole, the tool achieved a precision of approximately 93% when evaluated on EJBCA. The other two Enterprise Java applications, being small in size and mostly trivial in complexity, provided a precision figure of 100%. Every single call to `findByPrimaryKey` was matched.

In addition, we evaluated the precision with respect to the call sites invoking `findByPrimaryKey` in the application code. The 152 invocations of `findByPrimaryKey` methods in EJBCA corresponded to 43 distinct call sites. Of those, a total of four call sites were unmatched (meaning that there was at least one invocation for the call site that was not matched by the tool), and 39 were fully matched. This corresponds to precision of approximately 91%.

3.4.2 Analysis Cost

Several overheads were measured:

- Startup overhead (time to start the JVM with everything loaded, before the execution of any tests)

- Run-time overhead during test execution
- Time to process the execution trace (offline version only)

The start-up time of JBoss and EJBCA deployed without instrumentation was 1 minute and 35 seconds. The start-up time with instrumentation according to the online version of the algorithm was 1 minute and 38 seconds. The time it took the framework to start with EJBCA instrumented with the offline version was 1 minute and 37 seconds. Those three numbers are very close, and fall within the possible error due to non-deterministic events such as hard drive swapping. Indeed, the start-up overhead should be either negligible or nonexistent, because none of EJBCA's code is executed. The only possible source of overhead is the slightly longer time to load EJBCA's classes as a result of the slightly increased size of those classes due to the instrumentation. The results for the other two applications were similar.

The run-time overhead was as follows. EJBCA completed its test suite without any instrumentation in 4 minutes and 48 seconds. The online version of the algorithm ran the test suite (and produced immediate matches) in 18 minutes and 12 seconds, while the offline version of the algorithm completed the test suite in exactly 12 minutes. This corresponds to an overhead of 279% for the online version, and 150% for the offline version. Again, those numbers match one's intuition: the online version of the algorithm processes the data on the fly. The gateway must create a new object wrapping the data passed to the VFG for every call to it, and then search for an already existing such object within the VFG (which is based on hash tables). Finally, if there is a match, the gateway must write the relevant information to a file. The offline version, on the other hand, only incurs overhead for invoking the gateway and pushing the data passed to it to a file, which is wrapped in a buffer to alleviate costs.

Note that for the offline version the above number measures only the time to generate the execution trace on disk. The size of the disk trace produced by the offline version was 183 MB.

For the two smaller applications, the overheads for the online version were slightly lower: approximately 183% for Duke's Bank, and approximately 195% for Pet Store. This could potentially be due to the smaller number of assignments in these applications, which corresponds to faster search in the VFG.

Finally, we measured the time to process the execution trace for the offline version of the algorithm. For EJBCA the time was 15 minutes and 33 seconds, and for the two smaller applications the time was approximately 1 minute per trace.

Note that even though the analysis is dynamic, it will typically have to be executed only once for a J2EE application because of the specifics of its intended use (see Section 2.4). Thus, the 2.8× slowdown for the online version is practical and quite reasonable.

3.5 Optimizations and Enhancements

Several optimizations could reduce the run-time overhead of the analysis. For example, when a reference type parameter flows in from the client, it is not necessary to track its value unless the type is interesting. A set of interesting types can be determined in advance. That set would include composite primary keys, and possibly other DTO objects. The determination of such a set should include scanning the source code of the enterprise application for certain annotations (e.g., annotations specifying composite primary keys), as well as DTO identification, as described in Chapter 5.

Enhancements to the algorithm are also possible. One such enhancement is the tracking of entity bean primary keys that are contained within other structures, namely primary keys passed from the client tier as parts of a DTO.

Supporting DTOs containing primary keys includes some non-trivial issues. First, DTO identification is a crucial pre-processing step in this case, which means a direct utilization of the algorithm presented in Chapter 5. Second, an additional step is necessary in the dynamic analysis, which will "unwrap" a DTO whenever one is passed from a client tier. The separate parts of the DTO can then be tracked according to the algorithm with no changes. Such parts may include primitive types, composite primary keys, and other DTOs. In the case of a DTO wrapped within a DTO, the inner DTO is again unwrapped, and its components are tracked separately.

Yet another enhancement includes tracking a composite key after its creation. It is possible, although extremely unlikely, that a composite key is created within the EJB tier, and is not used later to refer to an entity bean object. For example, a client may pass the components of a composite key to the EJB tier, which constructs the composite key object and returns it to the client without utilizing it further. In such an unlikely case, the dynamic analysis will incorrectly identify the remote call parameters as uniquely determining the partition of the session bean.

To alleviate that issue, the second version of the algorithm should track references to newly created composite keys until they flow into an `findByPrimaryKey` method. Such tracking will require alterations to the VFG, because a node corresponding to the composite key must have more than one origin in this case. The origins of such a node are nodes corresponding to the primitive-type values that comprised the composite key object at its construction.

Conclusions and Future Work. The dynamic analysis described in this chapter identifies the entry points of primary keys, or parts of primary keys, within the EJB tier of an enterprise application. Within the larger context of the dissertation this analysis contributes towards the object-level lookup service described previously by identifying the pieces of data that must be looked up, so that the client tier executes a remote call against the correct EJB tier machine. The analysis cost, while significant, is not prohibitive, and thus the analysis is practical. Moreover, analysis cost could potentially be reduced by the various optimizations described earlier. The analysis precision, while already good, could be further improved by utilizing additional algorithms (e.g., dynamic slicing), as described in Chapter 6.

CHAPTER 4

EJBQL RELATIONSHIP IDENTIFICATION

The previous chapter proposes an analysis that identifies a major portion of the data necessary to implement the object-level lookup service discussed in Chapter 2. There are, however, other pieces of information that flow between the client tier and the EJB tier, which uniquely identify the objects within the EJB tier that the application needs in order to service the corresponding request. That information comprises parameters used in queries that the EJB tier executes against the data tier to fetch corresponding objects.

4.1 EJBQL

This section provides a brief overview of the Enterprise JavaBeans Query Language (EJBQL), based in part on material from [24]. EJBQL defines the queries for the finder and select methods of an entity bean with container-managed persistence within a J2EE application. A subset of SQL92, EJBQL has extensions that allow navigation over the relationships defined in an entity bean's abstract schema. The scope of an EJBQL query spans the abstract schemas of related entity beans.

EJBQL queries are defined in the deployment descriptor of the entity bean. Typically, a tool will translate these queries into the target language of the underlying data

store. Because of this translation, entity beans with container-managed persistence are portable — their code is not tied to a specific type of data store.

The term container-managed persistence (CMP) means that the EJB container handles all database access required by the entity bean. The bean's code contains no database access (SQL) calls. As a result, the bean's code is not tied to a specific persistent storage mechanism (database). Because of this flexibility, even if an administrator deploys the same entity bean on different J2EE servers that use different databases, he will not need to modify or recompile the bean's code. In short, the entity beans are more portable. This is the main reason CMP is predominantly used in J2EE applications, as opposed to the other option, bean-managed persistence, which leaves all persistence support in the hands of the developer, and the resulting code is generally not portable (not to mention the additional effort and cost in implementing custom versions of services that the application server already provides). The applications we tested utilize CMP exclusively.

In order to generate the data access calls, the container needs information that application developers provide in the entity bean's abstract schema.

4.1.1 Abstract Schema

Part of an entity bean's deployment descriptor, the abstract schema defines the bean's persistent fields and relationships. The term abstract distinguishes this schema from the physical schema of the underlying data store. In a relational database, for example, the physical schema is made up of structures such as tables and columns.

The name of an abstract schema is specified in the deployment descriptor. This name is referenced by queries written in EJBQL. For an entity bean with container-managed persistence, the application developer must define an EJBQL query for every finder method (except `findByPrimaryKey`). The EJBQL query determines the query that is executed by the EJB container when the finder method is invoked.

4.1.2 Persistent Fields

The persistent fields of an entity bean are stored in the underlying data store. Collectively, these fields constitute the state of the bean. At run time, the EJB container automatically synchronizes this state with the database. During deployment, the container typically maps the entity bean to a database table and maps the persistent fields to the table's columns.

A `CustomerEJB` entity bean, for example, might have persistent fields such as `firstName`, `lastName`, `phone`, and `emailAddress`. In container-managed persistence, these fields are virtual. Developers declare them in the abstract schema, but they do not code them as instance fields in the entity bean class. Instead, the persistent fields are identified in the code by access methods (getters and setters).

4.1.3 Relationship Fields

A relationship field resembles a foreign key in a database table — it identifies a related bean. Like a persistent field, a relationship field is virtual and is defined in the enterprise bean class with access methods. But unlike a persistent field, a relationship field does not represent the bean's state.

4.1.4 Multiplicity

There are four types of multiplicities:

- One-to-one: Each entity bean instance is related to a single instance of another entity bean. For example, to model a physical warehouse in which each storage bin contains a single widget, `StorageBinEJB` and `WidgetEJB` would have a one-to-one relationship.

- One-to-many: An entity bean instance may be related to multiple instances of another entity bean. A sales order, for example, can have multiple line items. In the order application, `OrderEJB` would have a one-to-many relationship with `LineItemEJB`.

- Many-to-one: Multiple instances of an entity bean may be related to a single instance of another entity bean. This multiplicity is the opposite of a one-to-many relationship. In the example mentioned in the previous bullet, from the perspective of `LineItemEJB` the relationship to `OrderEJB` is many-to-one.

- Many-to-many: The entity bean instances may be related to multiple instances of each other. For example, in college each course has many students, and every student may take several courses. Therefore, in an enrollment application, `CourseEJB` and `StudentEJB` would have a many-to-many relationship.

4.1.5 Direction

The direction of a relationship may be either bidirectional or unidirectional. In a bidirectional relationship, each entity bean has a relationship field that refers to the other bean. Through the relationship field, an entity bean's code can access its

related object. If an entity bean has a relationship field, then it can be said that it "knows" about its related object. For example, if `OrderEJB` knows what `LineItemEJB` instances it has and if `LineItemEJB` knows what `OrderEJB` it belongs to, then they have a bidirectional relationship.

In a unidirectional relationship, only one entity bean has a relationship field that refers to the other. For example, `LineItemEJB` would have a relationship field that identifies `ProductEJB`, but `ProductEJB` would not have a relationship field for `LineItemEJB`. In other words, `LineItemEJB` knows about `ProductEJB`, but `ProductEJB` doesn't know which `LineItemEJB` instances refer to it.

EJBQL queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one bean to another. For example, a query can navigate from `LineItemEJB` to `ProductEJB`, but cannot navigate in the opposite direction. For `OrderEJB` and `LineItemEJB`, a query could navigate in both directions, since these two beans have a bidirectional relationship.

4.1.6 Finder Queries

As described above, EJBQL queries are included in an entity bean's configuration data, and are translated into Java code by tools provided by the application server. Each query maps to a method within its corresponding entity bean class. Finder queries correspond to methods that fetch from the data tier one or more objects that are instances of the entity bean class where the query was defined. Such queries often take parameters to filter a selection.

Figure 4.1 represents a simple finder query that retrieves from the data tier all players with the specified position and name (specifically, all objects of type `PlayerEJB`

```
SELECT DISTINCT OBJECT(p)
FROM Player p
WHERE p.name = ?1
```

Figure 4.1: Simple finder query example

```
SELECT DISTINCT OBJECT(p)
FROM Player p, IN (p.teams) AS t
WHERE t.league = ?1
```

Figure 4.2: Complex finder query example

whose `name` field equals the first and only parameter passed to the query). When translated to the code of the application, this query can be executed by calling a `findByName(String name)` method of that entity bean class.[2] The `name` element is a persistent field of the `PlayerEJB` entity bean. The `WHERE` clause compares the value of that field with the parameter of the `findByName` method. EJBQL denotes an input parameter with a question mark followed by an integer. The first input parameter is `?1`, the second is `?2`, etc.

The parameters to such finder queries typically originate in client requests. In the above example, a client will typically call remotely a session bean method with a `String` parameter. The session bean will pass that parameter to the entity bean method corresponding to the above query. After the entity bean method returns the collection of corresponding objects, the session bean might pass them back to the client, or it might process them additionally.

[2]The method name has been specified in the same configuration file that specifies the body of the query.

```
SELECT DISTINCT OBJECT(p)
FROM Player p
WHERE p.salary BETWEEN ?1 AND ?2
```

Figure 4.3: Relationship finder query example

Figure 4.2 depicts a slightly more complex example of a finder query. The additional complexity is introduced by the navigation of two relationships. The `p.teams` expression navigates the relationship between players and teams, and the `t.league` expression navigates the relationship between teams and leagues.

The query in that figure retrieves the players that belong to a specified league. The finder method is `findByLeague(LocalLeague league)`. Note that in this example the parameter is an object whose type is application-specific, and matches the `league` relationship field in the comparison expression of the `WHERE` clause.

Figure 4.3 represents a finder query that has a relationship between input parameters other than equality. Various operators may be present in finder queries that form relationships among the participating entities in the respective query, including the input parameters. In this particular case, the query returns all players whose salaries fall within the range between its two input parameters. The full grammar of EJBQL is specified in [23].

4.2 Dynamic Analysis

The intelligent proxy described in Chapter 3 relies on the knowledge of certain data that flows from the client tier to the EJB tier, and the fact that the data uniquely identifies the application state that the corresponding client request will access when serviced by the EJB tier. The proxy makes a lookup request against the object level

lookup service with that data, receives the address of the physical machine that holds the respective application state, and forwards the client request accordingly.

Chapter 3 proposed a dynamic analysis that focused on identifying the primary keys that flow from the client tier to the EJB tier. While primary keys constitute a significant portion of the data the intelligent proxy service needs, there exist client requests that do not pass such primary keys. However, the majority of those client requests feature at least one parameter that is passed to an EJBQL query somewhere within the EJB tier.

The intelligent proxy needs to be aware of such parameters in order to execute lookup requests against the object-level lookup service. That service in turn must be aware of both the values of those parameters and the EJBQL queries in which they participate, so that it can reliably identify the application state relevant to those parameters and return the (potentially multiple) addresses of physical machines holding that state to the proxy.

In the most general case, the object-level lookup service must execute the entire query that a value passed to it participates in. There often are cases, however, when the lookup service needs to execute only a certain part of the query. For example, if an EJBQL query requires joining three tables at the database level (which corresponds to following two entity bean relationship fields), and the selection of objects returned by the query is based on a parameter of the third entity bean, then the lookup service needs only that parameter, the entity bean it is a part of, and the relationship in which the parameter participates within the query. Such an example is depicted in Figure 4.2. The query in that figure joins two tables at the database level based on a relationship field (i.e., `Player.teams`), and returns objects of type `Player` based

on the value of a field in the other table (i.e., `Team.league`). The lookup service is not interested in identifying the location of every object in the memory space of the application, however. If the application state is partitioned optimally among physical machines, then relationships among objects such as the one in Figure 4.2 mean that all related objects are located within the same partition/physical machine. Therefore, the lookup service needs information identifying any of the related objects in order to return the address of the machine containing all of them. In Figure 4.2, the lookup service only needs the value of the `?1` parameter, as well as its name and relationship (i.e., "`Teams.league = ?1`"), in order to return the address of the physical machine that contains both the relevant `Team` and the related `Player` objects. In general, queries containing at least one table join at the database level can be optimized at the level of the object lookup service.

4.2.1 Abstract Algorithm

To achieve identification of the data described in the previous paragraphs, we propose a dynamic analysis that builds upon the ideas presented in the previous chapter. As a reminder, the dynamic analysis from Chapter 3 tracks assignments of values that originate in a client request to the EJB tier (from the point of view of the EJB tier). Whenever a call to `find` is encountered with the participation of such a value, the analysis outputs a match between the origin of the value and call to `find`. The previous chapter described a version of that analysis that identifies the entry points of primary keys. With some extensions, the analysis can be used to also identify EJBQL parameters and the query relationships in which they participate.

The analysis presented in this chapter is again based on code instrumentation. The information captured from instrumenting the application code is stored in a VFG. The abstract version of this analysis proceeds as follows:

- Whenever a remote call is made against a session bean object, intercept the formal parameters to that call. Filter those input parameters based on their type: if a value of such a type flows as input parameter to an EJBQL query somewhere within the EJB tier of the application, remember information that uniquely identifies this parameter in a VFG node.

- Whenever an assignment takes place within the EJB tier, proceed as in Chapter 3.

- If a method is called that executes an EJBQL query, intercept the parameters to that method. Output the association between each of those parameters and its origin at the entry to the EJB tier, if it exists. For each such parameter, report its complete type (primitive or entity bean), as well as the role it plays in the query.

- If the method called in the previous bullet returns a single object (as opposed to a Collection), create a new VFG node for the method's return value, and set its origin to the origin of the parameter input to the EJBQL query. If there are multiple such parameters, the new VFG node will have multiple origins.

The differences between this algorithm and the one presented in Chapter 3 are the additional information that the algorithm uses in the first bullet, and the additional information it outputs in the third bullet. There is also an additional step (represented

by the last bullet above) which takes care of a possible use case in J2EE applications: an entity bean is retrieved from the database based on one of its parameters, and is later used as a parameter in another query. The additional step ensures that the analysis outputs the true origin of the parameter used in the latter query. If that step does not exist, no origin would be reported for the second query.

It is clear from the above discussion that the analysis needs a preprocessing step as well as a modification of the VFG.

4.2.2 Preprocessing

There are two preprocessing steps that are necessary: determine the types of input parameters to EJBQL queries present in the application, and analyze the queries themselves to extract the relationships in which these parameters participate.

Input Parameter Types

The analysis, as presented in the bullet list above, needs to be aware of the types of all values passed to EJBQL queries. Note that those types have no constraints except those imposed by the application. It might well be that a value of type `double` is used in an EJBQL query, which would be impossible in the case of a primary key. It is clear that the J2EE application must be analyzed for the types of parameters passed to queries. We had access to the source code of our experimental applications, and we extracted the necessary information from the source code as a preprocessing step.

Figure 4.4 represents the declaration of an EJBQL finder method within EJBCA, one of our test applications. This declaration exists as a comment block within the source code of the `EndEntityProfileDataBean` entity bean, which implements

```
* @ejb.finder
*   description="findByProfileName"
*   signature="org.ejbca.core.ejb.ra.raadmin.EndEntityProfileDataLocal
      findByProfileName(java.lang.String name)"
*   query="SELECT OBJECT(a) from EndEntityProfileDataBean a WHERE a.profileName=?1"
```

Figure 4.4: EJBCA query example

the `EndEntityProfileDataLocal` interface. The details of this preprocessing are not interesting for the purpose of understanding the dynamic analysis, and are not included here.

Relationship Identification

An important piece of information the object-level lookup service needs, when presented with the values of parameters that will later participate in a query, is the relationships those parameters participate in. This information is not necessary for the correct operation of the service, because it is always possible to execute the entire query as it exists in the application. However, the availability of such information leads to potential optimizations of the object lookup service, as discussed previously. While it is possible to dynamically parse the EJBQL queries called during the analysis, that will introduce significant overhead at run-time. We chose to pre-parse all queries before the dynamic algorithm was run, and make the information collected during that parsing available to the analysis.

During this preprocessing step we extracted comparison relationships with the participation of query parameters. Specifically, we were interested in the six comparison operators (equals, greater than, less then, different from, greater or equal, less than or equal) and the `BETWEEN` keyword, and we extracted such relationships where at least one operand was an input query parameter. We also extracted logical

operations in cases where there are two or more comparisons, every one of which operates with at least one input parameter (e.g., "select all students with lastname ?1 and GPA greater than ?2" where ?1 and ?2 are input parameters). We considered only these kinds of relationships; if the query contains more complicated ones (e.g., functions such as `SUM` or queries whose `WHERE` clauses include relationships without the participation of input parameters such as a comparison between two fields of an entity bean) our current approach reports that it is not possible to express the effect of the query with the simple relationships we target. Our experimental results show that such simple relationships are enough to fully describe the effects of the large majority of queries in our test applications.

As a result of this preprocessing step our analysis could output information regarding the types of entity beans and the names of the fields participating in such relationships. As an example, parsing the query in Figure 4.4 produces the information that the first parameter of the `findByProfileName` query of `EndEntityProfileDataBean` participates in an equality check against a `profileName` member field of the same entity bean.

4.2.3 Value Flow Graph

We use a similar data structure to the one used for the first analysis. There are the same kinds of nodes as those described in the previous chapter. We made several additions, however, to account for the differences between the two algorithms, namely:

- Every value in the VFG may have multiple origins for the reasons explained previously. At the same time, the number of the origins of a VFG node typically

remains constant for the lifetime of that node. Therefore, we replaced the single origin reference we used in Chapter 3 with an array of such references.

- In addition, every node in the VFG except the root nodes may potentially represent an entity bean reference that was fetched as the result of a query. We set a flag for such nodes, and also keep the fully qualified name of the entity bean class, to allow the possibility for the potential enhancement described in more detail later in this chapter.

4.2.4 Code Instrumentation

The dynamic analysis relies on code instrumentation to intercept interesting events and information related to them from within the application code. Code instrumentation is again executed on the Jimple representation of application bytecode.

In addition to the instrumentation mentioned in the previous chapter, we instrument the entries of methods that execute EJBQL queries in a different way than other method entry events. Specifically, we do not create new VFG nodes for the formal parameters as we usually do for other methods, and we do not rely on the instrumentation of the return statements of those methods. Rather, we check whether the return type of the method is a `Collection`. If it is, we silently skip instrumenting the return value; otherwise, we insert a call to a hook that will associate the variable carrying the return value from the method with the actual parameters to that method. In effect, if the return value is not a `Collection`, we treat the method call as a direct assignment, whose left side is the variable carrying the return value, and whose right side are the actual parameters to the method.

Note that for every such method we also insert a call to the sink hook of the analysis. That hook will take the information regarding the actual parameters and output various information related to them, which is the main purpose of the analysis.

4.2.5 Concrete Algorithm

This section presents a detailed version of the concrete algorithm for the dynamic analysis. Note that portions of this description repeat that of the analysis proposed in Chapter 3. We present the full algorithm here for the sake of completeness.

Whenever a session bean method is called remotely, the analysis intercepts the call and its parameters. It proceeds to check the parameter types against the list of types participating in EJBQL queries somewhere within the EJB tier of the application. The analysis has obtained this list from one of the preprocessing steps described previously. If a parameter's type is present in the list, the analysis creates a root VFG node of the local variable type and populates it accordingly.

Whenever an assignment is encountered, the analysis intercepts information that fully identifies the right-hand side and the left-hand side of the assignment. If a VFG node that matches the right-hand of the assignment is present, there are two possible cases:

- A node that matches the left-hand side of the assignment is present in the VFG. In that case its origin is set to point to the origin of the VFG node representing the right-hand side of the assignment.

- Such a node does not exist. In that case it is created, and its origin is set accordingly.

Whenever a call to a method that is different from a sink (i.e., different from a query method) is encountered, the analysis creates a VFG node for each formal parameter of the method. It names those nodes accordingly and sets their stack depth to the current one plus one, so that they will be found when the method's formals participate in assignments within the method. It also sets the origins of those nodes to the origins of the actuals that carry the values at the method call. In addition, if the method call returns a value, the analysis creates an additional local variable node in the VFG that will represent the left-hand-size variables at the call site; this node has no origin yet.

Whenever a value-carrying return statement is encountered, the analysis finds the VFG node that corresponds to the left-hand size at the call site. This node has no origin, is not a root node, and its stack depth is the current one minus one. When it is found, the analysis sets its origin to the origin of the local variable carrying the value of the return statement. If one is not found, the case is silently skipped. For any return statement, including value-carrying ones, all local variable nodes whose stack depth is the current one are removed from the VFG.

Whenever a call to a query method is encountered, the analysis outputs a match between that query and the origins of its actual parameters. In addition, it looks up information about the relationships in which those parameters participate within the query, and outputs it as well. This information is available as a result of the second preprocessing step defined previously. Also, the analysis checks if the return value of the method is a `Collection`; if it is not, it creates an entity bean VFG node corresponding to the left-hand-side local variable at the call site, and sets its origin to the origins of the actual parameters of the call.

```
public void cloneEndEntityProfile(org.ejbca.core.model.log.Admin, java.lang.String,
    java.lang.String) throws org.ejbca.core.model.ra.raadmin.
    EndEntityProfileExistsException {
  ...
  java.lang.String r2, r3, r4, r6, r7, $r8, r34;
  org.ejbca.core.ejb.ra.raadmin.EndEntityProfileDataLocal r13;
  ...
  r2 := @parameter1: java.lang.String;
  r3 := @parameter2: java.lang.String;
  ...
  r13 = interfaceinvoke $r12.<org.ejbca.core.ejb.ra.raadmin.
      EndEntityProfileDataLocalHome: org.ejbca.core.ejb.ra.raadmin.
      EndEntityProfileDataLocal findByProfileName(java.lang.String)>(r2);
  ...
  interfaceinvoke $r17.<org.ejbca.core.ejb.ra.raadmin.
      EndEntityProfileDataLocalHome: org.ejbca.core.ejb.ra.raadmin.
      EndEntityProfileDataLocal findByProfileName(java.lang.String)>(r3);
  ...
}
```

Figure 4.5: EJBCA query invocation

4.2.6 Example

Figure 4.5 illustrates an EJBCA use case. The session bean method that is presented executes the same EJBQL query twice with different parameters. The entity bean method and the query it executes are depicted in Figure 4.4.

When a client invokes the session bean method, it passes an Admin object and two String values. The preprocessing step to the analysis has determined that Admin objects are not participants in queries within the application, so this formal parameter is ignored. The analysis does create two root VFG nodes corresponding to the two String formal parameters.

The next interesting event is the invocation of `findByProfileName`, which our preprocessing has determined to be an EJBQL query method. The analysis outputs a match between the first parameter of that query and the second parameter of the

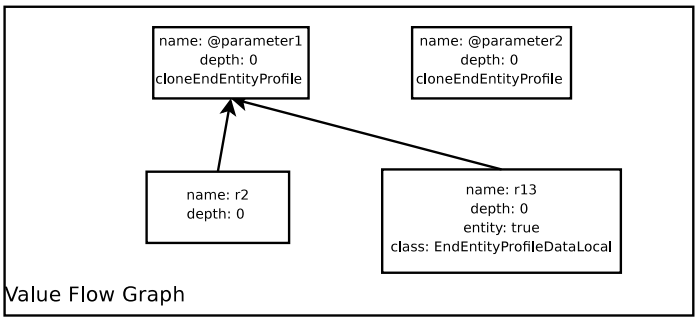

Figure 4.6: Final VFG state

`cloneEndEntityProfile` method of this session bean, as well as the additional relationship information regarding the first parameter of that query. This information is output in the form "`type.fieldname relationship queryParameter`" with potentially multiple such clauses, joined by logical operators. In this specific case, the following relationship is reported: `EndEntityProfileDataBean.profileName = ?1`. In addition, the analysis creates a new VFG node corresponding to the return value of the query method. That node represents local variable `r13`, which holds a reference to an entity bean of type `EndEntityProfileDataLocal`. It has a single origin in this case, because the query has a single parameter. The origin of that node is set to the origin of the query parameter, which is itself a root node.

The last event of interest in this example is another call to the same query method, but with a different value. In addition, there is no return value from that call. In this case the analysis outputs a match between the first parameter to the query and the third parameter of `cloneEndEntityProfile`. There are no new VFG nodes created as a result of the execution of this query. The relevant state of the VFG after the second execution of the query method is presented in Figure 4.6.

4.2.7 Finer-Grain Tracking of Values

It is possible to extend the algorithm to capture the complete flow of query parameters within the EJB tier of the application similarly to the extension described in Section 3.3. The changes are similar to those described in the previous chapter, namely: a tag that specifies whether a VFG node is "live" or "dead"; the full signature of the method where the variable is declared; and a counter distinguishing between multiple calls to the same method. In addition, the enhanced algorithm must allow for VFG nodes with multiple parents. Respectively, there may be more than one chain from a VFG node to EJB tier entry parameters. Such multiple chains must be reported appropriately.

4.3 Experimental Study

The experimental setup we used was the same one as that for the analysis proposed in Chapter 3. As before, the experimental subjects were Enterprise Java applications built on J2EE up to version 1.4, corresponding to EJB version up to 2.1. We next describe several changes and additions that may be made to the analysis so that it is compatible with the latest Enterprise Java version (1.5, corresponding to EJB 3.0).

One difference between EJB 2.1 and EJB 3.0 is the change in the persistence mechanism, which was already briefly mentioned in Chapter 3. While J2EE 1.4 applications rely on entity beans providing their own persistence (either container-managed or bean-managed), EJB 3.0 applications make use of an `EntityManager` object for their persistence needs, including executing queries against it. While this is a significant difference in terms of application design, it matters little for our analysis,

and we could trivially replace the exit points of our analysis with the respective `EntityManager` calls.

Another difference is the place of query specification. In EJB 2.1 applications that place is a configuration file or a comment block within the source code of the respective entity bean. In EJB 3.0 applications, queries are specified either in configuration files, in annotations within the source code, or simply as Strings passed to specific `EntityManager` methods. The last case necessitates a change in our dynamic analysis: it should investigate all such String values in order to become aware of the particular query to be executed and to report correct information regarding its parameters. In effect, the analysis must intercept the String values passed to certain `EntityManager` methods and parse those values at run time, thus incorporating the preprocessing steps described previously in the online algorithm. Finally, the query language of EJB 3.0 applications, which is called JPQL, is much richer than EJBQL, which will require changes to the preprocessing steps.

We only implemented an online version of the algorithm, which processes information as it is passed from the instrumented code. Also, we did not have to track thread IDs because of the use of single-threaded test suites.

4.3.1 Analysis Precision

The experimental setup was the same as in Chapter 3. We used EJBCA's test suite, and manually tested the functionality of the two smaller applications through a web browser. The precision was defined as the number of executed queries for which the analysis could find a matching entry point for their parameters, relative to the total number of executed queries that were instrumented.

Note that the preprocessing step identifies all EJBQL queries that take input parameters. There are queries that return *all* entity beans of a specific type, as well as queries that return purely statistical information on all entity beans, e.g., the average of all product prices. All such queries were ignored because there is no useful output the analysis could have produced for them.

The analysis matched 96% of all queries with EJBCA's own test suite (i.e., 43 out of 45), and 100% when tested on the two smaller applications (7 queries for Duke's Bank and 4 queries for Pet Store). A match in this context corresponds to a match between every parameter flowing into the query and an entry point to the EJB tier. After investigating the source code of EJBCA, we found the following causes for analysis imprecision:

- There was a case of an unmatched parameter whose value was manipulated within the EJB tier. Specifically, it is a case of calculating a one-way hash of a String password.

- A query took as its input parameter the value of a member field of an entity bean. We later discuss a potential enhancement of the analysis that will account for such cases.

As with the previous analysis, we also measured the number of matched query call sites in the code of EJBCA. The 45 query invocations corresponded to 13 query call sites in EJBCA's code. We consider a call site to be matched only when all runtime invocations of the call site were fully matched. Of the 13 call sites, 11 were matched, which corresponds to precision of approximately 85%.

We also tested the coverage of parameter relationships. The analysis was able to classify the relationships within 87% of the executed EJBQL queries in EJBCA, and 100% for the two smaller applications. From a call site point of view, 10 out of the 13 call sites in EJBCA had their relationships fully identified, which corresponds to precision of approximately 77%. This is fundamentally a measurement of how well the simple relationships described earlier capture the common cases occurring in real code. Later we discuss a potential approach for increasing this precision value. This number is significant because it indicates that the majority of executed queries are either simple enough to be executed without penalty by the object-level lookup service, or it is possible that a very simple part of them be extracted and used to satisfy the needs of that service.

4.3.2 Analysis Cost

The run-time overhead of the online analysis was 293% for EJBCA. The results for the two smaller applications were comparable.

4.4 Enhancements

There are several possible enhancements to the algorithm. One such enhancement is the tracking of query parameters that are contained within other structures, e.g., query parameters passed from the client tier as parts of a DTO. This enhancement is similar to the one described in Chapter 3.

Another enhancement consists of "unwrapping" an entity bean returned from a query and tracking its parts separately. An entity bean typically consists of a set of values corresponding to database columns, with their associated getters and setters. These values can be tracked by the algorithm by introducing new VFG nodes for

them, and setting the origins of those nodes to the origin of the node corresponding to the entity bean that contains the values. This enhancement will immediately improve the precision of the algorithm, e.g., for the EJBCA application.

A third possibility is taking care of Java Collections of entity beans returned from query methods. While our test applications did not have such use cases, it is conceivable that an entity bean in such a collection, or parts of it, may later be used in another query. The analysis may potentially be improved to track members of such collections, and even parts of such members.

Yet another enhancement includes capturing more complex relationships in EJBQL queries in the second preprocessing step to the dynamic analysis. As an example, one could extract the arithmetic operations on input parameters from queries such as "select all students whose GPA is between ?1 and ?1 + ?2" where ?1 and ?2 are the two input parameters to the query. The object-level lookup service could use such information to recreate only the relevant part of the respective arithmetic expression.

Conclusions. In summary, we have designed and implemented a dynamic analysis that identifies the entry points of query parameters to the middle tier of an EJB application, as well as certain commonly used relationships in which those parameters participate within the queries. The implementation has practical overhead and achieves excellent precision. These results add to those of the dynamic analysis of entity bean IDs, and contribute information necessary for the implementation of an object-level lookup service.

CHAPTER 5

DTO IDENTIFICATION

This chapter proposes a dynamic analysis that can be used to contribute information towards future enhancements of the analyses described in Chapter 3 and Chapter 4.

5.1 Introduction

Data Transfer Object (DTO) [16, 2, 62] is a design pattern that is commonly used in distributed systems in general and Enterprise Java applications in particular. Every method call made to a business object in an enterprise system is potentially remote. In EJB 2.1 applications such remote invocations use the network layer regardless of the proximity of the client to the server, creating network overhead. Such method calls may permeate the network layers of the system even if the client and the enterprise application layer are both running in the same Java Virtual Machine (JVM). When multiple attribute values need to be obtained, using multiple calls to `getX` methods (one per attribute) is highly inefficient.

A DTO, also called a transfer object or a value object,[3] encapsulates a set of values, allowing remote clients to request and receive the entire value set with a single remote call.

We define a dynamic program analysis that identifies classes which implement the DTO pattern. The identification of DTOs is useful in several contexts. First, it can assist *program comprehension* by pointing out instances of this Enterprise Java pattern; this can also be used to create additional documentation (e.g., JavaDoc comments). Second, DTOs may involve serialization, which can create performance bottlenecks [52, 39]. As a *performance optimization*, identified DTOs can be subjected to customized serialization mechanisms (i.e., type-aware serialization and acyclic-graph serialization [66, 56]). Third, identifying DTO instances is an important step towards *software evolution* for migrating Java Enterprise applications based on the older EJB 2 specifications to the new EJB 3 model. In EJB 3, entity beans can be detached from the persistence context related to a database, modified elsewhere in the application, and merged back to the respective persistence context. As DTOs in older EJB applications usually mirror the state of entity beans, it is highly desirable to simplify the design by using the same entity bean object to represent state throughout the application layer stack. Finally, *a DTO is an ideal candidate for a composite primary key* of an Entity Bean as defined in Chapter 3 that is passed directly from a client tier to the EJB tier. In addition, a primary key may be passed from the client tier as part of a DTO. Consequently, identifying DTO instances may serve as a preprocessing step for the dynamic analysis defined in Chapter 3.

[3]Note that a DTO is different from the GoF Value Object pattern [26].

We propose a dynamic analysis for identifying DTOs in Enterprise Java applications. The analysis tracks the reads and writes of object fields, and maintains information about the application tier that initiates the field access. The lifecycle of a DTO is represented by a finite state automaton (FSA) that captures the relevant run-time events and the location of the code that triggers these events. We implemented the proposed approach using the JVMTI infrastructure in Java 6, and performed a study on the same EJBCA application used in the previous two chapters. The experimental results indicate that the dynamic analysis achieves high precision and has acceptable overhead.

5.2 Background and Problem Statement

As discussed previously, an Enterprise Java application consists of layers, or tiers. The EJB tier usually provides various means for clients to access it. There are interfaces to both clients within the same JVM (i.e., the web tier) or remote clients. Thus it is possible for a user to access the functionality of a well-designed J2EE application by going to the web site of the application and browsing its web tier, or by running a desktop application designed specifically for that purpose, which connects directly to the EJB tier through Java RMI or a similar remoting mechanism.

5.2.1 Uses of Data Transfer Objects

The information that a client (be it the web tier or a remote application) receives from the EJB tier could be a primitive value, a Java Collection object, or a DTO that is specific to the particular enterprise application. The client may also create DTOs and pass them to the EJB tier in order to decrease the network overhead. A DTO may even be created in one tier (EJB, web, or remote client), carry its state

```
public class UserDataVO implements Serializable {
  private String username;
  private String subjectDN;
  private int caid;
  private String subjectAltName;
  private String subjectEmail;
  private String password;
  private int status;
  private int type;
  private int endentityprofileid;
  private int certificateprofileid;
  private Date timecreated;
  private Date timemodified;
  private int tokentype;
  private int hardtokenissuerid;
  private ExtendedInformation extendedinformation;
  public void setUsername(String user) { ... }
  public String getUsername() { ... }
  public void setDN(String dn) { ... }
  public String getDN() { ... }
  public int getCAId() { ... }
  public void setCAId(int caid) { ... }
  ...
}
```

Figure 5.1: A DTO example from EJBCA.

to another tier, and then the same object may be used to carry new state back to the tier where it was created. This more complex pattern is referred to as *Updatable DTO*.

The single most important part of the definition of a DTO is that it must be capable of being passed over the wire. Indeed, there are multiple explanations and/or definitions of the DTO design pattern [16, 2, 62], but the only property they all strictly require of a DTO class is that it must implement either interface `java.io.Serializable` or (in extremely rare cases) `java.io.Externalizable`. Figure 5.1 shows a part of a DTO class from the EJBCA application used in the experimental study.

5.2.2 DTO Lifecycle

A DTO passes through several states during its lifecycle. In State 1, the object has just been created, and it does not yet carry any application-specific state, meaning that its fields have not been initialized. Populating the just-created DTO with application state by initializing its fields leads to State 2. When in State 1 or State 2, the DTO exists within the same J2EE tier (we will refer to it as Tier 1); this is usually the EJB tier. The fields of the DTO may be accessed and/or modified in Tier 1 after initialization for various reasons, for example reading a field indicating the application-specific status of the DTO.

The DTO enters State 3 when it is passed to an object belonging to a different tier (Tier 2). The application state carried by the DTO is read and/or written in Tier 2. The DTO then might be passed back to State 4 in Tier 1, where its data is read and potentially modified again. In fact, the object can "oscillate" between State 3 and State 4. Finally, when the use case ends, the DTO is prepared for garbage collection (in either Tier 1 or Tier 2) and enters State 5. The FSA for this lifecycle is shown in Figure 5.2.

5.2.3 Problem Definition

The goal of this work is to perform dynamic analysis of the execution of a J2EE application in order to identify classes whose instances implement the DTO pattern. To achieve this goal, the analysis tracks the lifecycle events of potential DTO objects, and matches the observed sequence of events (on per-object basis) with the DTO FSA. The key research questions that need to be answered are as follows:

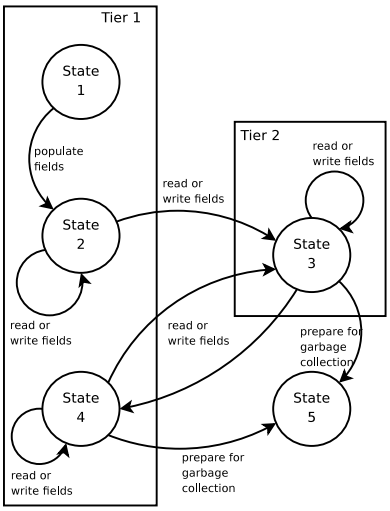

Figure 5.2: Lifecycle of a DTO.

- What specific kinds of run-time information should be collected during the dynamic analysis, and how should this information be used to identify DTO objects?

- What is the false positive rate? That is, how often does the analysis observe objects whose behavior matches the DTO state transition diagram, even though these objects do not actually implement the DTO pattern?

- What is the false negative rate? In other words, how many DTO objects violate the pattern described above, and therefore are not reported by the analysis?

- What is the run-time overhead of the analysis?

5.3 Dynamic Analysis for DTO Identification

DTOs always carry state, and sometimes also implement application logic. This approach is focused on the state, which makes it unnecessary to track call/return or

method entry/exit events. To identify DTOs, the analysis tracks field accesses (reads and writes) for objects of potential DTO classes, together with the location of the code that triggers the events. The relevant run-time events are observed with the help of the Java Tool Interface (JVMTI), which provides a portable and standardized infrastructure for implementing the dynamic analysis.

5.3.1 Processing at Class Loading

Classes that are DTO candidates are Java classes that (1) belong to the enterprise application, and (2) implement interface `java.io.Serializable`. The analysis intercepts all *class load* events within the application server's JVM, considers whether the loaded class is a DTO candidate, and if so, tags all its fields. Tagging enables run-time events related to tagged fields: whenever a field in such a class is accessed (read or written) anywhere within the same JVM, a JVMTI event is generated and processed by the analysis. Events are generated for a tagged field regardless of whether the field is static (i.e. the field directly pertains to a DTO candidate class) or instance (i.e. the field belongs to an object that is an instance of a DTO candidate class).

5.3.2 Processing of Field Reads and Writes

When a *field read* or a *field write* event is observed, the analysis identifies the object that caused that event and the location (i.e., application tier) of that object. To find the object that caused the event, the analysis obtains a handle to the call stack of the current thread. A traversal of the stack frames, starting from the most recent one, identifies the first method that belongs to an application class different from the class that the field belongs to. The receiver object of that method is the

one accessing the candidate DTO. The analysis has to consider the state of the run-time call stack because it is very common for DTOs to implement getter and setter methods. If a field is read through a getter, or written through a setter, the top call stack frame would be within the context of a method belonging to the same class as the field. However, the analysis is interested in the object that caused the field read or write to happen in the first place.

Note that the traversal of the call stack must identify a method that belongs to an application class. This is necessary due to the frequent use of reflection in J2EE applications and application servers. The analysis matches the current frame method's class against the packages of the J2EE application to ensure that it does not identify the cause of the field read/write event as a `java.lang.Class` object, `java.lang.reflect.Method` object, or some other irrelevant object which is only used as part of a mechanism internal to the application server.

Once the analysis pinpoints the method causing the field read/write event, it determines the tier that contains the method's declaring class, and thus the tier that accesses the DTO candidate. This determination is done using precomputed lists of class names for each application tier. A simple static analysis of the package hierarchy of the J2EE application can be used to create such lists. J2EE applications usually follow a strict package hierarchy, with web tier classes consolidated into a package (or a package hierarchy), EJB tier classes consolidated in another package hierarchy, etc. The application is even deployed as a combination of different modules — for example, a web module is deployed as a .war archive, and is strictly separated from other modules. The analysis uses that information to find the location of the object

(or class, if the method is static) that caused the field read/write event. That location (i.e., application tier) is also the current location of the DTO candidate.

The analysis maintains information about a DTO candidate object, including the fully-qualified class name, the location of its creation, as well as a flag indicating whether the object has moved to a different location during its lifetime. When a field read/write event is observed and the current location of the DTO candidate is identified, the analysis checks whether this is the first such event ever to happen for that object. If this is the case, the object has just been created, and is entering State 2 of its lifecycle. The location (i.e., tier) of its creation is recorded as the object's current location.

If this is not the first such event, the object has already been accessed. If this is not a DTO, its current location should always be the same as the location of its creation. If the object a DTO, however, it may change its location, which means it might be entering State 3 of the DTO lifecycle. If the location of the DTO candidate is different from the location of its creation, the "moved" flag is set to true, which essentially means that the object is marked as a DTO.

5.3.3 Processing at Garbage Collection

When a DTO candidate is being garbage collected, the analysis observes the corresponding event and processes it. At this time no information about the object is accessible, including its class name, so the analysis has to rely on its data structures for all information related to that object. At this stage the analysis simply checks whether the object that is being garbage collected has moved from the location of its creation, and if so, the object's class is reported as a DTO class. The implementation ensures

that a garbage collection event is observed for every DTO candidate by aggressively forcing garbage collection through JVM's internal mechanisms.

5.4 Implementation Details

The Java 6 version of the JVM Tool Interface (JVMTI) was used to implement the dynamic analysis outlined above. JVMTI provides the various capabilities that are necessary, as well as the required event hooks to the internal workings of a Java 6 JVM. The code is written completely in C, and interfaces with the JVM through the Java Native Interface mechanism, of which JVMTI is an extension. The JVMTI capabilities that the tool requires (and enables) are the ability of the JVM to generate field read events, field write events, object free events, and the capability for the agent to set and get object tags.

JVMTI provides the useful capability for agents to set and get object tags, which the analysis uses extensively. Unfortunately, there is a constraint on the use of JVMTI object tags: an object tag is a single integer, which is supposed to be used by agents as an internally-generated (possibly auto-incrementing) key. A single integer of 32 bits cannot fit all the information tracked about an object.

Since the agent is written in C, the analysis simply uses for the object tag (of type integer) the memory address of an instance of the book-keeping data structure. The analysis allocates memory for an instance of its data structure the first time an object is used, and populates the object's tag with the memory address of that instance. During the lifetime of the object, the agent casts the tag from an integer to pointer and vice versa as needed. Finally, when the object is being garbage collected, the analysis also deallocates the memory used by the associated data structure.

We did consider tracking method entry and exit events instead of field read and write events. Since the goal is DTO detection, which can be achieved through tracking state changes and their location, both sets of events would be able to produce results suitable for interpretation. However, there is an important difference in the implementation of these event hooks in JVMTI. Once the capability of sending method entry/exit events is enabled, every single such event (i.e., every call stack push or pop) is reported, which introduces very significant overhead. On the other hand, it is possible to flag only certain fields, and manipulating only those fields would trigger field read/write events. To evaluate run-time overhead, a simple test was performed with only method entry/exit events enabled. The EJB application server took over two hours to initialize and start, with no enterprise applications running or even deployed. As a result of this test, we decided to track state changes through the much more lightweight field read/write events.

A useful optimization that JVMTI provides automatically is its sending garbage collection events only for objects which have had their tag set. This functionality ensures that events are sent when objects tracked by the analysis are garbage collected, but no overhead is incurred for any other objects. We have not made any other efforts to optimize the implementation of the algorithm or the data structures. Clearly, there is ample room for improvement in terms of optimizations, which can be pursued in future work.

5.5 Experimental Study

The dynamic analysis was evaluated on the same physical machine and application server as the previous two analyses. The single experimental subject was EJBCA, which was also used in the experiments from the previous two chapters.

5.5.1 Analysis Precision

Various uses of EJBCA were tested by accessing the web tier from a web browser. The results can be summarized as follows. There were a total of 132 classes that implement `java.io.Serializable` and are part of EJBCA, and whose bytecode was loaded from disk by the JVM for potential execution. Since there was no other way of detecting the actual EJBCA DTOs, and because documentation is almost nonexistent, we had to manually inspect the source code of every potential DTO class (i.e., each of these 132 classes) and decide whether that class was a DTO or not based on our experience with J2EE applications. After this manual analysis, 13 out of the 132 classes were deemed to be DTO classes. Of those 13, 11 were actually used by EJBCA when the test cases were executed. (The other two remained loaded and prepared, but unused: no objects were instantiated from them at any point.) Ideally, the dynamic analysis would report these 11 classes.

The dynamic analysis reported 11 classes, of which 10 were DTO classes as determined by the manual examination of the code. These 10 classes demonstrated various levels of complexity of behavior. There were classes that were true DTOs and nothing more — they only had fields, constructors, getters, and setters. There were classes that implemented `java.lang.Comparable` in addition to being DTOs, which meant they had additional logic for comparison. There were classes that had many

fields, with the corresponding getters and setters, but also had methods capable of returning additional information based on the values of the fields and complex hashing algorithms. Finally, there was a class `org.ejbca.util.Query` that had four fields, two of which were Vector references, but it never returned them directly. The fields of that class were only used to answer certain boolean queries through corresponding methods.

The single false positive that the analysis reported was `org.ejbca.util.Query`. This class may actually be considered a DTO. It was reported by the analysis because its instances move between the J2EE tiers. The problem is that these `Query` instances never allow other objects to have direct access to their fields through setter and getter methods, and as such the class does not fit the "standard" definition of a DTO. Therefore, in the manual examination of the code it was classified as a non-DTO class. However, the purpose of this class is still to carry data between the tiers, and it allows some access (albeit very circumspect) to this data. Arguably, this class implements the DTO pattern "in spirit".

Of the 11 classes that were manually determined to be DTOs and were also used during the execution of the test cases, one class was not reported by the analysis — that is, this was a false negative. The reason the class was not reported is because it was wrapped by another DTO class. The agent detected the wrapper DTO, not the inner one, because the cause for all reads/writes for the inner DTO is actually the wrapper DTO, according to the analysis algorithm (the wrapper is different from the inner DTO, but it still is a part of EJBCA, and belongs to the same tier). Consequently, according to the dynamic analysis, the inner DTO has never moved because

all locations that cause field reads and writes (i.e., methods of the wrapper DTO) are part of the same tier as the inner DTO.

In summary, the analysis correctly identified 10 DTOs, and had one false positive and one false negative. These promising results indicate that DTO identification can be performed with high precision using the proposed run-time analysis techniques.

5.5.2 Analysis Cost

The start-up time of JBoss with EJBCA deployed with and without the agent running in the background was measured. The purpose of this test is to estimate the startup overhead due to the analysis. The application server started completely in 1 minute and 32 second without the agent, and in 2 minutes and 56 seconds with it. These results correspond to run-time overhead of about 91%.

We also ran a batch of test use cases against the RMI interface of EJBCA to track the execution time with and without the agent. We used this method to estimate the overhead, as opposed to tracking the times for the web use cases, because of the event-driven structure of the analysis. When testing web use cases, both the web tier and the EJB tier are located within the same JVM. Due to the HTTP session objects the web tier keeps for individual users, it would be nearly impossible to draw hard lines between the separate test use cases in terms of memory and processing time, unless we tracked the time it took every single method to execute and return. As discussed earlier, tracking method entry/exit events incurs impractical overhead. We chose instead to test the RMI interface of the EJB tier only, and to track the execution time on the client side. Because of the remote client, in this experiment the analysis did not report any DTOs — from its perspective, no objects were ever

moved to a client tier. However, the analysis still responded to all relevant events and performed all stages of the algorithm, resulting in a meaningful estimate of overhead.

The running time of the batch of remote tests without the agent was 4 minutes and 53 seconds. The corresponding time with the agent turned on was 17 minutes and 44 seconds. These results correspond to run-time overhead of approximately 263%. While significant, this overhead is not prohibitive, and the approach remains suitable for practical use. As mentioned earlier, no attempts were made to optimize the performance of the analysis implementation; future work may be able to improve this performance and to reduce significantly the run-time overhead.

Conclusions. The dynamic analysis presented in this chapter identifies instances of the DTO design pattern with high precision and acceptable run-time cost. A DTO is an ideal candidate for a composite primary key that is passed directly from a client tier to the EJB tier. What is more, primary keys (both composite and primitive) of Entity Beans may be wrapped inside a DTO and passed between tiers. Consequently, this analysis constitutes a potential preprocessing step for the dynamic analyses presented in Chapter 3 and Chapter 4 within the larger context of the dissertation. In addition, DTO identification is useful for a number of other tasks related to program comprehension, performance optimization, and software evolution.

CHAPTER 6

RELATED WORK

6.1 Identification of Entity Bean IDs and Query Parameters

The dynamic analyses described in Chapter 3 and Chapter 4 trace the flow of certain values through the EJB tier of a J2EE application. Thus, they resemble a dynamic slicing algorithm. The basic approach to dynamic slicing is to execute the program once and produce an execution trace which is processed to construct a dynamic data dependence graph that in turn is traversed to compute dynamic slices [34, 35]. A dynamic program slice is an executable subset of the original program that produces the same computations on a subset of selected variables and inputs. Informally, a dynamic slice consists of all statements that influence the value of a variable occurrence for specific program inputs, and a dynamic slice with respect to a set of variables may be obtained by taking the union of slices with respect to individual variables in the set. Slices in general, and dynamic slices in particular, are used for debugging purposes to decrease the number of program statements a human must manually inspect in order to find the cause for a program bug.

Research exists on imprecise dynamic program slicing. Reference [1] includes a total of four dynamic slicing algorithms, representing a range of solutions with

varying space-time-accuracy trade-offs. Unfortunately, the first two of those dynamic slicing algorithms, which are imprecise, have been proven to produce slices many times larger than the respective precise slices [69]. The work in [69, 71] defines three precise dynamic slicing algorithms with full preprocessing (FP), limited preprocessing (LP), and no preprocessing (NP), respectively. Preprocessing in this context means building a dependence graph by recovering dynamic dependencies from the program execution trace. Thus FP builds the entire dependency graph before slicing. NP does not perform any preprocessing; it uses demand-driven analysis for recovering dynamic dependencies, and caches the recovered dependencies for future reuse. LP first augments the execution trace with summary information, allowing for faster traversal of the trace later, and then uses the same demand-driven analysis as NP on the compacted execution trace. The experimental results presented in this work show that the FP algorithm is impractical for real programs because of the amount of memory it uses: it runs out of memory during the preprocessing phase, because the dynamic dependence graphs are extremely large. The NP algorithm has no such problems, but is slow. The LP algorithm never runs out of memory, while at the same time is relatively fast, which implies that a carefully designed precise dynamic slicing algorithm can be practical, and there is no need to trade precision for space or time.

The original work on dynamic slicing included only backward analysis, i.e., after the execution trace of the program is first recorded, the dynamic slicing algorithm traces backwards the execution trace to derive dynamic dependence relations that are then used to compute dynamic slices. Korel and Yamanchili proposed a *forward* approach to dynamic program slice computation [36]. Dynamic slices are computed

during program execution without major recording of the execution trace. The major advantage of the forward approach over the backward approach is that space complexity is bounded, unlike the case with the painstaking execution trace recording that backward program slice computation must perform. The main disadvantage of such algorithms is the space and time required to maintain a large set of dynamic slices. Reference [70] analyzes the characteristics of dynamic slices, identifying space efficient representation of a set of dynamic slices, and proposes a representation based on reduced ordered binary decision diagrams. This work then defines a space and time efficient forward computation algorithm, based on the analysis and the proposed representation. The experimental results are favorable in both time and space complexity.

Finally, there is the dynamic slicing algorithm described in [68], which comprises the state of the art in dynamic program slicing. This algorithm could have been used for the purposes of identifying Entity Bean IDs. The trouble with it is its generality, which results in significant overhead. While more optimal than all other dynamic slicing algorithms, this algorithm still compares unfavorably against our specialized algorithm in terms of processing time and run-time overhead. Having in mind the multitude of potential optimizations to our algorithm, the performance gap between the two algorithms is likely to increase. A possible future enhancement in this area is improving our algorithm's precision by incorporating parts of this optimal dynamic slicing algorithm, so that the causes of the few unmatched `findByPrimaryKey` calls can be reliably identified.

Another field that is very relevant to the algorithms presented in Chapters 3 and 4 is information flow analysis. Static information flow analysis is well researched [18, 19],

especially within the domain of language-based security mechanisms. For instance, [12, 45] introduce a typed assembly language that alleviates the difficulties inherent in information flow analysis for low-level languages. There is also considerable research in statically ensuring information flow control for high-level languages, e.g., Java and Java-like languages [46, 7, 4, 54, 5, 61].

Dynamic information flow analysis is also known as dynamic taint analysis. It consists, intuitively, of marking and tracking certain data in a program at run-time. Dynamic taint analysis is a relatively new field. Most research in the field of tainting is related to the context of information security. The most studied type of software exploit is overwrite attacks, a class of attacks where sensitive program data is overwritten by an attacker. The data overwritten typically consists of return addresses, function pointers, or format strings. By suitably overwriting this data, attackers are able to hijack a program and execute arbitrary code. The two most common types of overwrite attacks are buffer overflows and format string attacks.

Newsome and Song [47] present one of the first dynamic-taint-based approaches for preventing overwrite attacks. Their approach taints any data read from a network socket. The tainted data is then propagated as the program executes. Finally, the approach enforces the security of a program by checking that tainted data is not used as the target of a jump, a format string, or a system-call argument. Several other techniques for detecting overwrite attacks were developed at a similar or later time. In particular, [33] proposes a hardware-based approach.

Dynamic tainting has also been used to prevent SQL injection attacks, in which attackers submit maliciously-crafted strings to a web application to access its underlying database. Most dynamic taint based approaches against SQL injection operate

by tainting and tracking unsafe data, similar to the approaches to prevent overwrite attacks. Before a query string is sent to the database, it is checked to ensure that no tainted data was used to create the string or specific parts of it. The work in [48] proposes an instance of this approach for web applications written in PHP, whereas [53, 27] target Java-based applications.

Dynamic-taint-based approaches have been successfully used in the context of information flow security to enforce information flow policies. Such policies define limits on how information is used within a system. An example of an information flow security policy is a military system where classified information must not be transferred to individuals without the appropriate clearance level. Dynamic tainting is an ideal technique for that purpose. Different taint markings can be used to label sensitive information, and then the analysis can check whether marked data reaches parts of the system forbidden to it according to the policies in place. For example, Chow and colleagues [14] present TaintBochs, a simulator for tracking tainted data through an entire system, including applications, operating system, and hardware. They use the simulator to research the lifetime of sensitive information in several common applications. More recently, McCamant and Ernst [44, 43] presented a tainting technique for dynamically tracking information flow in C programs. Their technique quantitatively measures information flow at the instruction level using bit-tracking analysis. Finally, Masri and Podgurski have contributed an empirical study [41] and techniques [42, 40] for dynamic information flow analysis.

All of the above approaches are limited in that they are defined in an ad-hoc manner, and for the specific purposes of a single application area. These inherent limitations are addressed in [15] by Clause, Li, and Orso, who have presented a generic

dynamic taint analysis framework that is general and flexible, as well as DYTAN, an implementation of the framework that works on x86 binaries. This framework allows for performing both data-flow and control-flow tainting, and does not rely on a customized run-time system.

The dynamic analysis for Entity Bean ID identification can be thought of as a dynamic taint analysis targeting a specific problem. The algorithm it uses utilizes a single class of taint markings. It computes the subset of the data in the program that is affected by a given set of data. The sources of tainting are the parameters of remote calls to Session Bean methods, and the sinks of tainting are the second parameters of `EntityManager.find` calls. Unfortunately, none of the existing approaches target directly the case outlined in this work, because of the complexity and specifics of Enterprise Java applications. DYTAN is a general framework that could have been used with some additional overhead. Unfortunately, it only supports x86 binaries, as opposed to Java bytecode.

A field related to the algorithm presented in Chapter 4 is dynamic analysis of SQL injection attacks in Web applications. At the heart of the issue here is verifying that the user has not altered the syntax of the query. Respectively, much of the work focuses on analyzing string inputs. For example, Buehrer et al. enforce a policy that user input must be a single token in the query [13]. They bound user input, parse the query, and check whether the parse tree retains the same structure when the user input is replaced by a single dummy node. They have also provided an implementation of their technique for J2EE. WASP, by Halfond et al., uses positive tainting: it allows only trusted characters in keywords unless the programmer specifies with a regular expression that user input may include certain keywords [27, 28]. Su

and Wassermann use delimiters to track user input into generated queries, and parse the queries based on a modified grammar [60]. The latter work is very similar to [13].

6.2 DTO Identification

Researchers have proposed a number of techniques based on static analysis to recover design patterns from existing programs (e.g., [3, 37, 6, 55, 49, 51]). There is also a body of work related to formalizing design patterns (e.g., [58, 63, 8]). Such formalizations can later be used to match structural patterns in the source code of a program to the structure of a design pattern. However, many design patterns in general, and the DTO pattern in particular, have significant behavioral aspects that static analysis cannot capture precisely. For example, in order to model precisely the temporal sequence of events that constitutes a pattern instance, a static analysis may have to employ expensive algorithms with flow/context/path sensitivity, which creates significant scalability challenges for real-world Enterprise Java applications. As another example, dynamic features such as reflection and dynamic class loading (commonly used in enterprise Java applications) present a serious challenge for static analysis. Even though some existing work has addressed scalability problems related to analysis of FSA-based properties for large programs (e.g., [25, 20, 10, 11]) as well as handling of dynamic features (e.g., [59, 38]), the current state of the art does not provide enough evidence that static analysis of patterns in EJB application can achieve correctness and precision at a practical cost. Thus, we believe that for such applications the use of dynamic analysis is a more natural choice, at least until more advances are made in static analysis research.

Wendehals and Orso [67] propose an approach that combines static and dynamic analysis. A static analysis examines structural properties in order to identify pattern-instance candidates. A dynamic analysis considers a set of such candidates and checks whether the run-time interactions match the behavioral properties of the pattern. A FSA is used to match the observed method calls to the expected behavior. This approach is similar to our work in that it uses a predefined FSA-based abstract pattern specification to capture relevant program behavior and match it to design patterns. More generally, there is a body of work on performing dynamic analyses based on properties expressed by finite state automata (e.g., [9, 17, 29, 32]). Our work uses a similar technique, with the focus being on (1) reads and writes of fields, and (2) the application tier that initiates the read or write, as determined by examining the run-time call stack. Most existing approaches target properties related to method entry/exit events, with or without information about the identity of the receiver object. However, our experience with JVMTI indicates that such an approach may be impractical for J2EE applications due to the complexity of the underlying middleware (i.e., a JBoss application server), which leads to substantial run-time overhead.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This work proposed three dynamic analyses to extract information from J2EE applications that makes possible their deployment on an application server featuring a partitioned architecture and an object-level lookup service. Such an architecture and service will, in turn, ameliorate the significant problems of memory and network scalability in existing J2EE configurations.

The first dynamic analysis identifies the entry points of primary keys to the EJB tier of a J2EE application. This information can be used by the application server to intercept client calls to the EJB tier and, after consultation with the object-level lookup service, to route those calls to the appropriate EJB tier machine. Such information is crucial for the correct operation of the proposed architecture. The analysis achieves that identification by tracking the flow of values participating in assignments and originating at entry points to the EJB tier until a value flows into a specific method call, determined by the J2EE specification to pass a primary key to the data tier of the application. The analysis also handles certain special cases, namely the existence of composite primary keys, which are complete objects that are used as primary keys as per the J2EE specification, and the participation of primary keys in arrays. An additional enhancement of the algorithm was proposed that outputs

the complete chain of assignments that a primary key passes through within the EJB tier. Such information may be used for better program comprehension.

The analysis was implemented via code instrumentation of interesting events within the code of the EJB tier of a J2EE application. Such events include assignments, method calls and exits, EJB tier entries and exits, as well as events necessary for the more complex cases such as composite primary key creation. Two separate versions of the analysis were implemented, an online and an offline version. The implementations were tested on a major, widely used and commercially deployed open-source J2EE application (EJBCA), as well as two smaller J2EE applications. The experimental evaluation indicates that the cost of the analysis is practical, especially in light of its intended use, while the analysis results achieve excellent precision.

Future optimizations to the implementation could potentially reduce analysis cost. The algorithm may also be enhanced in several ways. One potential enhancement is tracking primary keys that enter the EJB tier via more complex structures, e.g., instances of the DTO pattern or instances of Java Collection classes. Another enhancement is tracking composite primary keys after their creation to ensure that they are indeed used by the application in the way the J2EE specification intends.

The second dynamic analysis identifies another significant portion of the data necessary to enable a partitioned architecture and an object-level lookup service. It extracts the entry points of EJBQL query parameters to the EJB tier. Within the context of J2EE applications, queries are necessary to fetch objects from the data tier to the EJB tier based on more complicated logic than identification by primary keys. Such logic may include comparisons between object fields and the values flowing into a query in the EJB tier, and may span multiple relationships among objects.

The analysis tracks the flow of query parameters from their entry to the EJB tier to their exit via a call to a query method within the EJB tier. It outputs a match between those two points for every such value, enabling the application server to intercept client calls carrying such parameters and route them appropriately. It also outputs information identifying the most common relationships in which such parameters participate within the queries, enabling potential future optimizations of the object-level lookup service. The analysis also takes care of the case when an object returned by a query may be later used as a parameter in another query within the EJB tier.

The analysis modifies and enhances the previous algorithm to achieve its purpose. It consists of two preprocessing steps, namely identifying the possible parameter types participating in queries, and identifying the most common relationships that query parameters participate in, and the main step, which is executed at run-time and tracks the flow of query parameters. It was implemented via code instrumentation, and was tested on the same J2EE applications as the previous one. The experimental evaluation of the analysis indicates that it achieves practical cost and excellent precision. In addition, it captured the overwhelming majority of relationships of query parameters in our test applications.

Potential enhancements to the algorithm include increasing the coverage of the relationships in which query parameters participate. Another enhancement is tracking query parameters entering the EJB tier as parts of a DTO or a Java Collection. A third potential enhancement is unwrapping an object returned from a query and tracking its parts separately in case they are later used as parameters in another

query. Yet another enhancement consists of analyzing Java Collections returned from queries, unwrapping them, and tracking the objects contained in them separately.

The third dynamic analysis presented in this work identifies instances of the Data Transfer Object design pattern. This pattern is commonly used in J2EE applications for alleviating network overhead. Instances of it are used to wrap multiple values (objects or primitive-typed) and carry them together over the network. The analysis outputs such instances entering the EJB tier of a J2EE application, thus contributing to potential enhancements to the previous two dynamic analyses, as well as to better program comprehension, performance optimization, and software evolution.

The analysis algorithm investigates the lifecycle of certain objects living at the boundary between the EJB tier and a client tier, and matches that lifecycle against a state transition diagram that describes the behavior of an instance of the DTO pattern. The implementation used the Java Virtual Machine Tool Interface, which provides the capability to insert agents written in C in the Java Virtual Machine itself. Run-time tracking is performed for interesting events happening within the JVM, such as loading of serializable application classes, reading from and writing to fields of instances of such classes, and garbage collecting such instances. The implementation builds a history of the use of the instances in question, and matches that history against the usual behavior of a DTO instance.

The implementation was tested on EJBCA, which is a sufficiently large application to necessitate the use of DTOs. The experimental evaluation indicates that the analysis achieves very high precision and practical run-time cost. Future work could consider the tracking of parts of a DTO instance separately, due to the possibility

that such parts themselves are DTO instances. This functionality will increase the precision of the analysis.

Long-term future work includes investigating the challenges for building the object-level lookup service itself. The contributions of this dissertation have laid the grounds for such a service and the partitioned J2EE architecture that employs it. We envision that once such a service is built, it will naturally become a part of the Enterprise Java ecosystem, and will seamlessly solve some of the major problems J2EE administrators and application server developers are trying to tackle, namely those of memory and network scalability.

BIBLIOGRAPHY

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[2] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns, Second Ed.* Prentice Hall PTR, 2003.

[3] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *International Workshop on Program Comprehension*, pages 153–160, 1998.

[4] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2):131–177, Mar. 2005.

[5] A. Banerjee, S. Rosenberg, and D. Naumann. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353, May 2008.

[6] E. Baniassad, G. Murphy, and C. Schwanninger. Design pattern rationale graphs: Linking design to source. In *International Conference on Software Engineering*, pages 352–362, 2003.

[7] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *IEEE Symposium on Security and Privacy*, pages 230–242, 2006.

[8] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of design patterns in Java. In *International Conference on Automated Software Engineering*, pages 224–232, 2005.

[9] E. Bodden. J-LO: A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, Nov. 2005.

[10] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming*, pages 525–549, 2007.

[11] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2008.

[12] E. Bonelli, A. Compagnoni, and R. Medel. Information-flow analysis for a typed assembly language with polymorphic stacks. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, pages 37–56, 2006.

[13] G. Buehrer, B. Weide, and P. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *International Workshop on Software Engineering and Middleware*, pages 106–113, 2005.

[14] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. *USENIX Security Symposium*, pages 321–336, 2004.

[15] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[16] W. Crawford and J. Kaplan. *J2EE Design Patterns*. O'Reilly and Associates, 2003.

[17] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *International Workshop on Dynamic Analysis*, pages 1–7, 2005.

[18] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[19] D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[20] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *International Symposium on Software Testing and Analysis*, pages 12–22, 2004.

[21] *Duke's Bank*. `java.sun.com/j2ee/1.4/download.htm`.

[22] *EJB Certificate Authority*. `www.ejbca.org`.

[23] *EJBQL Full Syntax*. `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL5.html`.

[24] *EJBQL Tutorial*. `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL.html`.

[25] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis*, pages 133–144, 2006.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[27] W. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–185, 2006.

[28] W. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, 2008.

[29] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[30] *Java Pet Store*. `java.sun.com/developer/releases/petstore/petstore1_3_1_02.html`.

[31] *JBoss Application Server*. `jboss.org`.

[32] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. V. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[33] J. Kong, C. Zou, and H. Zhou. Improving software security via runtime instruction-level taint checking. *Workshop on Architectural and System Support for Improving Software Dependability*, pages 18–24, 2006.

[34] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[35] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[36] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *International Symposium on Software Testing and Analysis*, pages 66–79, 1994.

[37] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering*, pages 208–215, 1996.

[38] B. Livshits, J. Whaley, and M. Lam. Reflection analysis for Java. In *Asian Symposium on Programming Languages and Systems*, pages 139–160, 2005.

[39] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, Nov. 2001.

[40] W. Masri and A. Podgurski. Using dynamic information flow analysis to detect attacks against applications. In *Software Engineering for Secure Systems*, 2005.

[41] W. Masri and A. Podgurski. An empirical study of the relationship between information flow and program dependence. In *International Workshop on Dynamic Analysis*, 2006.

[42] W. Masri and A. Podgurski. Measuring the strength of information flows in programs. *ACM Transactions on Software Engineering and Methodology*, 2008.

[43] S. McCamant and M. Ernst. A simulation-based proof technique for dynamic information flow. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 41–46, June 2007.

[44] S. McCamant and M. Ernst. Quantitative information flow as network flow capacity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–205, June 2008.

[45] R. Medel, A. Compagnoni, and E. Bonelli. A typed assembly language for noninterference. In *Italian Conference on Theoretical Computer Science*, pages 360–374, 2005.

[46] A. Myers. JFlow: Practical mostly-static information flow control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[47] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium*, 2005.

[48] A. Nguyen-Tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*, 2004.

[49] J. Niere, W. Schaefer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *International Conference on Software Engineering*, pages 338–348, 2002.

[50] A. Pantaleev and A. Rountev. Identifying data transfer objects in EJB applications. In *International Workshop on Dynamic Analysis*, 2007.

[51] G. Pappalardo and E. Tramontana. Automatically discovering design patterns and assessing concern separations for applications. In *ACM Symposium on Applied Computing*, pages 1591–1596, 2006.

[52] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[53] T. Pietraszek and C. Berghe. Defending against injection attacks through context-sensitive string evaluation. *Recent Advances in Intrusion Detection*, 2005.

[54] M. Pistoia, A. Banerjee, and D. Naumann. Beyond stack inspection: a unified access-control and information-flow security model. In *IEEE Symposium on Security and Privacy*, pages 149–163, May 2007.

[55] J. Seemann and J. W. von Gudenberg. Pattern-based design recovery of Java software. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 10–16, 1998.

[56] M. Sharp and A. Rountev. Static analysis of object references in RMI-based Java software. *IEEE Transactions on Software Engineering*, 32(9):664–681, Sept. 2006.

[57] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, Sept. 2000.

[58] N. Soundarajan and J. O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *International Conference on Software Engineering*, pages 666–675, 2004.

[59] V. Sreedhar, M. Burke, and J. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, 2000.

[60] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[61] Q. Sun, A. Banerjee, and D. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Static Analysis Symposium*, pages 84–99, 2004.

[62] *Transfer Object Pattern*. `java.sun.com/blueprints/corej2eepatterns/Patterns`.

[63] B. Tyler, J. O. Hallstrom, and N. Soundarajan. Automated generation of monitors for pattern contracts. In *ACM Symposium on Applied Computing*, pages 1779–1784, 2006.

[64] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, pages 18–34, 2000.

[65] R. Vallée-Rai and L. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, McGill University, July 1998.

[66] R. Veldema and M. Philippsen. Compiler optimized remote method invocation. In *IEEE International Conference on Cluster Computing*, pages 127–137, 2003.

[67] L. Wendehals and A. Orso. Recongizing behavioral patterns at runtime using finite automata. In *International Workshop on Dynamic Analysis*, pages 33–40, 2006.

[68] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.

[69] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *International Conference on Software Engineering*, pages 319–329, 2003.

[70] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *International Conference on Software Engineering*, pages 502–511, 2004.

[71] X. Zhang, R. Gupta, and Y. Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Languages and Systems*, 27(4):631–661, July 2005.