Codifying Hidden Dependencies in Legacy J2EE Applications

Geoffrey Hecht*¶, Hafedh Mili*, Ghizlane El-Boussaidi*†, Anis Boubaker*†, Manel Abdellatif*‡, Yann-Gaël Guéhéneuc*§, Anas Shatnawi*, Jean Privat*, Naouel Moha*
*LATECE Laboratory, Université du Québec à Montréal, Canada
† Ecole de Technologie Superieure, in Montréal, Canada
‡ Ecole Polytechnique de Montréal, Canada
§ Computer Science and Software Engineering, Concordia University, Canada
¶ DCC, University of Chile

***Abstract*—J2EE applications tend to be multi-tier and multi-language applications. They rely on the J2EE platform and containers that offer infrastructure and architectural services to ensure distributed, secure, safe, and scalable executions. These mechanisms hide many program dependencies, which helps development but hinders maintenance, evolution, and re-engineering of J2EE applications. In this paper, we study (i) the J2EE specifications to extract a *declarative* specification of the dependencies that are *inherent* in the services offered and that are not visible in the user code that uses them. Then, we introduce (ii) a codification of the dependencies into rules, and (iii) a tool that supports the specification of those dependencies and their detection in J2EE applications. We validate our approach and tool on a sample of 10 J2EE applications. We also compare our tool against JRipples, a state-of-the-art tool for change-impact analysis tasks. Results show that our tool adds, on average, 15% more call dependencies, which would have been missed otherwise. On change impact analysis tasks, our tool outperforms JRipples in all 10 applications, especially for the early iterations of change propagation exploration.**

I. Introduction

The first generation of Java Enterprise applications (J2EE) are showing signs of age, including difficulties in their maintenance, evolution, and re-engineering [1]. Hence, many organizations are modernizing first-generation J2EE applications to modern service-oriented architectures, such as REST and micro-services [2], [3], [4].

J2EE applications are multi-tier, multi-language applications that rely on the J2EE platform to benefit from infrastructural and architectural services, such as distribution, persistence, scalability, and security. The platform relieves developers from the need to design, implement, test, and maintain such services. They also hide many of the *control dependencies* between the components of the platform and applications by using inversion of control and various late-binding techniques, which do not explicitly appear in the application source code. They also often use techniques like reflection or code generation that prevent static code-analysis tools and developers from identifying and understanding these hidden dependencies. For example, in Enterprise JavaBeans 2 (EJB2), there is an implicit call between the create(...) method on the home interface of an EJB, client-side, and the corresponding ejbCreate(...) method of the bean class, server-side. This call introduces hidden dependencies that are not visible for developers and tools which solely rely on the source code of the application.

A number of software development and maintenance tasks such as program comprehension, change impact analysis, refactoring, and debugging, rely on the dependencies between program elements (variables, functions, classes, methods, etc.) [5], [6], [7], [8]. Therefore, the already complex maintenance and evolution tasks of legacy applications may be hindered when hidden dependencies are not discovered. To continue with our example, a developer willing to modify the implementation of the ejbCreate(...) server-side method of an EJB, needs to analyze the impacts of this modification in all create(...) client-side methods. Moreover, previous approaches exist (e.g., [9], [10]) and many organizations have developed technology-specific tools to support migration process of legacy application. Even if this remain an essentially knowledge-intensive manual migration, these approaches also rely on similar dependencies. However, to the best of our knowledge, these approaches pertain to COBOL or technologies that do not involve the layers of abstractions found in J2EE. Hence, they offer processes applicable to the migration of legacy applications but they do not help dealing with hidden dependencies such as the one found in J2EE applications.

Hence, we propose a new approach to static code-analysis in which we can specify hidden dependencies and with which we can build model of legacy J2EE applications, including both explicit and hidden dependencies. Such dependencies connect the call graph obtained by statically analyzing the Web tier to that obtained for the EJB tier, otherwise unconnected.

In this paper, we describe our approach to specify hidden dependencies and apply it on EJB. We study the specification of EJB2 to characterize its hidden dependencies and encode them in rules. Then our approach uses call graphs, produced by a static-code analysis, which we complement by adding, programmatically, the control dependencies inherent to EJB2 to create augmented call graphs. We rely on the OMG Knowledge Discovery Metamodel (KDM) to represent the program elements and dependencies of these call graphs [11]. KDM allows the representation of different artifacts of applications, including JSP pages, Java files, configuration files independently of their languages. We use MoDISCO [12], an Eclipse-based implementation of KDM.

We evaluate our approach by applying our tool on 10 open-source J2EE applications. We also compare the performance of our augmented call graph on two maintenance tasks: (1) change impact analysis, using JRipples [7] and (2) call hierarchy analysis, using Eclipse's native functionality. Both tasks perform better with our augmented call graphs.

In the following, Section II discusses the analysis of legacy J2EE applications and presents our approach. Section III describes specifically container service contracts. Section V introduces our tool implementation. Section VI describes our evaluation. Section VIII summarizes related work. Section IX concludes with future work.

II. Issues in Analyzing J2EE applications

Call graphs are built using two techniques or combinations thereof: (1) *static code-analysis*, by parsing the source code of a program and–or (2) *dynamic tracing*, by executing a program and recording call sequences. The choice of the technique depends on many factors including (1) the availability of the *source code*, (2) the ease with which the program can be executed, (3) whether the programming language(s) is statically typed, and (4) the required level of accuracy (precision and recall of the identified dependencies) of the graph [13].

While dynamic techniques have to deal with complex setup, multiple execution scenario and difficulty to cover all the dependencies on both client and server side. The J2EE platform also make static techniques difficult to apply on application due to three problems. *First*, they are multi-language. A typical J2EE application combines several languages: (1) Java, for both server and client code (embedded in HTML, JSP, or JSF tags), (2) JavaScript, on the client side, embedded in HTML, JSP, or JSF tags, (3) various property files (key/value pairs), and (4) various XML configuration files (web.xml, ra.xml, ejb-jar.xml), all of which may introduce new dependencies.

Second, J2EE relies on dynamic binding-techniques, using a combination of:

1) *reflexion* or *introspection*: clients can invoke methods of classes *intensionally* as opposed to *nominatively*;
2) *data-driven control*: data-values control calls between clients and servers, as with a message-driven style
3) *run-time input*: control-data values control calls between clients and servers; data supplied at runtime, which hinders *data-flow analyses*.
4) *runtime code-generation*: executable code is generated at *load-time* or *runtime*, from configuration files.

Third, J2EE provides *services* that are typically offered using *inversion of control*: to take advantage of the services, client code must implement specific interfaces or *callback methods*, which embody the *service contract* between the clients and the containers. Obviously, calls to these callback methods are *not* explicit in the client code.

These three problems hinder current static program-analysis techniques that are limited to user supplied/developed code, yielding incomplete program call graphs. We want to augment the traditional *unilingual* static program-analysis techniques with other kinds of analyses, involving other kinds of artifacts, but also by codifying the services offered by containers, as illustrated in Fig. 1.

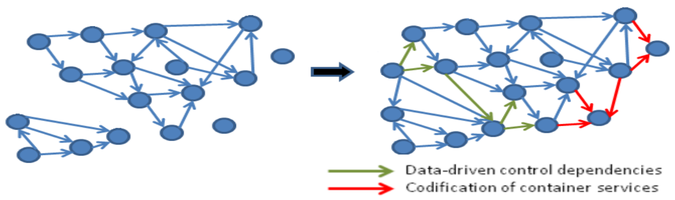

Data-driven control dependencies
Codification of container services

Fig. 1: Java code analyses, e.g., must be complemented with the analysis of (1) other program artifacts and (2) services offered by the platform.

Previous work proposed partial solutions to the first and second problems (Section VIII). To the best of our knowledge, no previous work addressed the third problem.

The J2EE platform and containers simplify user code development by one of two mechanisms: (1) by relieving developers from the need to *explicitly* invoke services—using inversion of control—or (2) by offering abstract interfaces that simply the invocation of those services, and hide many of their internal complexities. Either mechanism hides call relationships to, from, or between parts of the developer code.

If the source code of the platform at hands is available, one may be tempted to include that code in the analysis to uncover the missing call relationships. However, current static program-analysis techniques would still not work for two reasons. First, it is *not* practical to parse the code for a J2EE application server (millions of LOCs) to identify dependencies. Second, they would miss dependencies embodied in services that rely on *inversion of control*. To capture such dependencies in J2EE containers, we follow three steps :

Step 1: We study the specifications/documentation of EJB 2 to identify the dependencies inherent in the services offered.

Step 2: We encode the dependencies using *rules* whose condition parts characterize patterns of user code where the dependencies apply and action parts add those dependencies.

Step 3: We apply these rules on models of J2EE applications to add the missing dependencies to classic models already enclosing the explicit dependencies of the source code.

III. Step 1: Studying The EJB 2 Specifications

J2EE applications are built atop application-servers that offer services to their hosted applications. Developers of the applications can focus on their business logics, while application-servers provide the services supporting the applications. Without loss of generality, we focus in our study on EJB 2 to illustrate how these services work and the range of dependencies that they introduce.

We distinguish between two categories of services because they are specified and used differently in various software artifacts, thus they require different treatments :

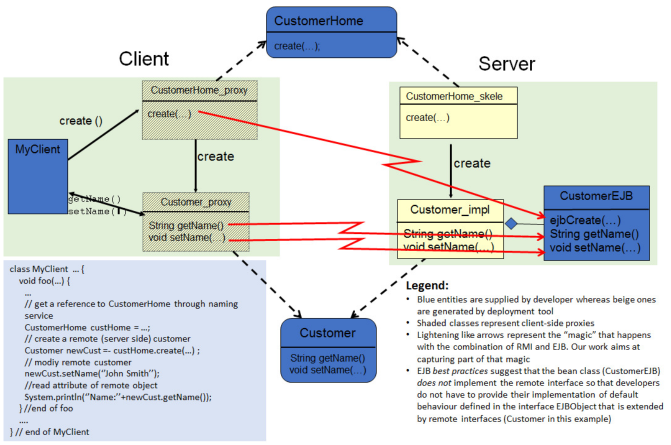
Configurable services, which are not offered by default. These services may be specified at the *class/bean level*, either in deployment descriptors or in code annotations (with EJB 3), or be invoked *explicitly* by user code at the instance level. This is the case of *security* and *transaction* services. For example, a developer can specify the *roles* that can invoke a bean method in the deployment descriptor, or explicitly call the security API method isCallerInRole(...) to check that the caller is authorized to call the method.

General/pervasive services such as remote method-invocation (RMI) and life-cycle management, that apply to *all* hosted applications that confirm specific 'obligations' stipulated in the 'service contract'. Those obligations take the form of an *interface* that a class needs to implement or extend (e.g. java.rmi.Remote, javax.ejb.SessionBean).

Both kinds of services embody hidden call dependencies within developer code (e.g., from client-side to server-side) and between developer code and server code. For the sake of space, we focus on the second kind because it offers more challenges to encode and recover dependencies between developer code and J2EE application-servers.

A. RMI Services

Fig. 2 shows an example two-tier client–server J2EE application that uses EJBs (an *entity bean*). The client application (class MyClient) manipulates customer objects managed by the remote server. The blue boxes represent developer-supplied code, which includes (1) the client-side application (MyClient class) and (2) the customer EJB. The entity bean representing *shared, remote instances* of customer objects requires (1) a Java interface for the remotely-available public methods of customer objects (Customer), (2) a Java interface, called home interface, for the remote factory of customer objects (CustomerHome), and (3) the server-side class that implements customer objects (CustomerEJB). By studying the J2EE specification, we learn that a client-side invocation of the method getName(...) on the variable newCust declared with the client application method foo(...) calls the method getName(...) on a service-side instance of class CustomerEJB (one of the red broken arrows in Fig. 2), which no static code analysis/parsing could have inferred.



```
class MyClient  ... {
   void foo(...) {
     ...
     // get a reference to CustomerHome through naming
     service
     CustomerHome custHome = ...;
     // create a remote (server side) customer
     Customer newCust =- custHome.create(...) ;
     // modiy remote customer
     newCust.setName("John Smith");
     //read attribute of remote object
     System.println("Name:"+newCust.getName());
   } //end of foo
   ....
} // end of MyClient
```

Legend:
- Blue entities are supplied by developer whereas beige ones are generated by deployment tool
- Shaded classes represent client-side proxies
- Lightening like arrows represent the "magic" that happens with the combination of RMI and EJB. Our work aims at capturing part of that magic
- EJB *best practices* suggest that the bean class (CustomerEJB) *does not* implement the remote interface so that developers do not have to provide their implementation of default behaviour defined in the interface EJBObject that is extended by remote interfaces (Customer in this example)

Fig. 2: An example J2EE application with RMI inherent dependencies

When the entity bean is deployed on the server, the deployment tool generates a JAR file to be included in client-side applications that contains the Java interfaces *and the client-side proxies*. In our example, the JAR would contain the interfaces Customer and CustomerHome (shown in blue in Fig. 2) and the classes Customer_Proxy and CustomerHome_Proxy (shaded beige classes in Fig. 2). A minimally 'intelligent' type analysis of the client code may infer that the run-time type of the variable *newCust*, declared of type *Customer* is *necessarily* Customer_Proxy, because it is the *only client-side class that implements the interface Customer*. However, it is not possible to infer that a call to Customer_Proxy.getName(...) ultimately results into calling CustomerEJB.getName(...) on the server side because: (1) *per the recommended EJB practice, the EJB class should not implement the remote interface* (If we did, we would have to provide default implementations for infrastructure methods that developers rarely need to worry about) and (2) an analysis of the code of the client-side Customer_Proxy.getName(...) *will not show* an invocation of the server side Customer_Impl.getName(...), as outgoing calls are translated into generic remote-invocation objects by the RMI infrastructure.

In summary, doing RMI in EJB makes it impossible for basic static analyses to link client-side invocations of remote methods to their sever-side implementations.

B. Life-cycle Management Services

Life-cycle management services manage EJBs in the application server to speed up object creation, often time-consuming, and to optimize resource usage. J2EE servers maintain *pools of EJB objects* (class *CustomerEJB* in our example) at runtime, which they assign to client applications to reduce object-instantiation time, and return to the pool when they are no longer used, so that can serve other client applications. Depending on its type (entity or session, and then stateless versus stateful session), an EJB goes through different stages during its lifetime. Fig. 3 shows the life-cycle of *entity* and *stateful session* beans. Space limitations do not allow us to explain the full life-cycle for all the kinds of beans; we will explain just enough to illustrate two additional, more complex dependencies than the ones for RMI: 1) establishing call relationships across *different signatures*, 2) the call to a single method, client side, results into a call sequence, server side, and 3) the distinction between *guarded* versus *unguarded* method invocation, to be explained below.

As shown in Fig. 3, entity beans can be in one of three states: (1) *Does-not-exist state*, (2) the *pooled state* in which an instance is available to client applications, and (3) the *ready state* in which the instance is assigned to a specific business object and waits for calls from a client application. In the following, we will illustrate the 'calls across different signatures' for the create(...)/ejbCreate(...) method; a similar issue arises with *remove(...)*.

With EJBs, the *create(...)* method is specified in the *home interface* which represents the *factory* object for the EJBs with a searchable JNDI name; this corresponds to the interface

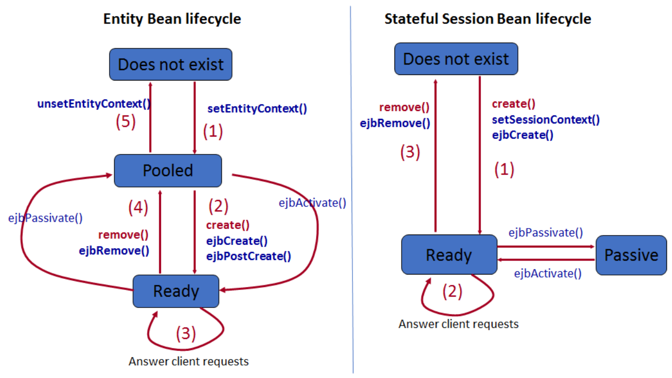


Fig. 3: The life-cycle of EJBs. Red methods belong to the *home* interface. They invoke the blue methods on the EJB class. Bold methods are invoked through explicit declarations in the client applications. Non-bold methods are invoked by the server to manage its resources.

$\mathrm{CustomerHome}$ in our example (Fig. 2 in many cases). When a client application calls a *create(...)* method on the client side home proxy, an $\mathrm{ejbCreate}(\ldots)$ method is called on the server-side EJB class, $\mathrm{CustomerEJB}$ in our example. The return types are not identical. Whereas the client-side/home $\mathrm{create}(\ldots)$ returns an instance of the remote object (interface $\mathrm{Customer}$ in our example), the server-side EJB class version (method $\mathrm{ejbCreate}(\ldots)$ in $\mathrm{CustomerEJB}$ in our example) returns an instance of *a primary key class*.

The second difference with RMI is the situation with $\mathrm{ejbPostCreate}()$: if the client code invokes *create(...)* on the client side home proxy, the server invokes *ejbCreate(...) followed by ejbPostCreate(...)*, which has a similar parameter list, and a void return type. Whereas $\mathrm{ejbCreate}(\ldots)$ is meant to initialize persistent attributes, $\mathrm{ejbPostCreate}(\ldots)$ is used to acquire/initialize additional, transient resources (e.g. connections to external resources).

The third difference is illustrated by the behavior of the $\mathrm{ejbActivate}(\ldots)$ and $\mathrm{ejbPassivate}(\ldots)$, which are best explained for the case of *stateful session bean*. A stateful session bean is a session bean (and thus is not meant to be persistent) *that needs to maintain conversational state between two method invocations* whereas a *stateless* session bean can be switched between client applications at will. If there are many 'assigned' client stateful sessions, even idle ones, the server eventually exhausts the pool. If that happens (lest we oversimplify), it swaps out some stateful session beans to secondary memory, according to some swap policy (e.g. least recently used). To this end, it calls *ejbPassivate(...)* which should serialize the parts of the session object's state that the developer cares to save, so that it can be recovered, using *ejbActivate(...)*, when the session is needed again. What makes this dependency complicated is that: (1) the method is *not* called/callable directly by client code, but may be called as a side-effect of other seemingly unrelated methods (e.g. many *create(...)* invocations), (2) it may involve some *fairly complex business logic* (A developer must choose which fields to make transient, which may have to be recomputed at deserialization, and which to make persistent, which will be serialized. A tradeoff between memory consumption and performance but also between passivation and activation performances.) , and (3) it is *the server that makes the decision, based on its internal state*.

The three cases will be illustrated in the rules shown in the next section.

IV. Step 2: Specifying Dependencies in J2EE

A. Overview

Table I shows the action rules that encode all of the hidden dependencies for the RMI and lifecycle management services for EJB 2, which relies on the explicit definition of interfaces. While similar underlying method calls exist in other versions of EJB, focusing on EJB 2 illustrates the range of dependencies that must be specified explicitly without loss of generality. The next two subsections discuss the rules induced by RMI and lifecycle management.

The rules use functions and predicates to identify and relate the different components of an EJB. We will define the main ones here. An EJB (E) is a aggregate consisting of three components: 1) a remote interface (I), 2) a home interface (H), and an EJB Class (C). This aggregate is defined in the *EJB deployment descriptor*, which is an XML file that we parse, along with the other project artifacts.

- $isHomeInterface(T)$ is a predicate that indicates that type T is a home interface
- $EJB(T)$ is a function that returns the EJB of which type T is a component, be it a home interface, a remote interface or an EJB class. In the example of Fig. 2, $EJB(\mathrm{CustomerHome})$ is the customer EJB, called $\mathrm{custEjb}$.
- $EjbClass(x)$ takes an EJB x as parameter and returns the corresponding EJB class. In our example, $EjbClass(\mathrm{custEjb})$ returns the class $\mathrm{CustomerEJB}$.
- $EjbPrefix(m)$ returns a method whose signature is the same as that of method m, with its name prefixed by ejb.
- $isRemoteInterface(T)$ is a predicate that returns true if T is a remote interface, i.e., a subclass of $\mathrm{EJBObject}$.

B. Codifying RMI Services

The rules for RMI are R1, R2 and R3. Intuitively, R1 says "given a home interface T (*CustomerHome* in our example) and a method $m(\ldots)$ (*create(...)* in our example) belonging to T, add a call dependency from $T.m(\ldots)$ (*Customer-Home.create(...)*) to $C.ejbM(\ldots)$, where C is the EJB class for the EJB whose home interface is T (*CustomerEJB* in our case), and $ejbM(\ldots)$ is a method of C (*ejbCreate(...)*) with the same signature as m but for the 'ejb' prefix.

To understand how this works, we should mention that static code analysis on the client code will identify a call from the client method ($\mathrm{MyClient.foo}(\ldots)$ in our example) to the home interface method $\mathrm{CustomerHome.create}(\ldots)$. Rule R1 adds a call dependency from $\mathrm{CustomerHome.create}(\ldots)$ to $\mathrm{CustomerEJB.postCreate}(\ldots)$. On the surface of it, this

TABLE I: Rules adding dependencies introduced by RMI and lifecycle management

Rule	**Condition:** $\forall$ *type T and method m of T such that*	**Action:** Add call dependency from $T.m$ to
R1	$isHomeInterface(T), \quad m = create(paramlist)$	$EjbClass(EJB(T)).ejbCreate(paramlist)$
R2	$isRemoteInterface(T), \quad m = f(param_1, ..., param_i)$	$EjbClass(EJB(T)).m$
R3	$isHomeInterface(T), \quad m = remove(paramlist)$	$EjbClass(EJB(T)).ejbRemove(paramlist)$
R4	$isHomeInterface(T), \quad m = create(paramlist),$ $\exists\ method\ mpost = ejbPostCreate(paramlist) \ \epsilon\ EjbClass(EJB(T))$	$EjbClass(EJB(T)).ejbPostCreate(paramlist)$
R5	$isHomeInterface(T), \quad isEntity(EjbClass(EJB(T)),$ $m = create(paramlist)$	$EjbClass(EJB(T)).setEntityContext(ctx : EntityContext)$
R6	$isHomeInterface(T), \quad isSession(EjbClass(EJB(T)),$ $m = create(paramlist)$	$EjbClass(EJB(T)).setSessionContext(ctx : EntityContext)$
R7	$isHomeInterface(T), \quad m = remove(paramlist)$	$EjbClass(EJB(T)).unsetEntityContext()$
R8	$isRemoteInterface(T), \quad m = f(param_1, ..., param_i),$ $\forall\ obj\ instance\ of\ T\ s.t.isPassive(EJB(T))$	$EjbClass(EJB(T)).ejbActivate()$
R9	$isHomeInterface(T), \quad m = create(paramlist),$ $\forall\ obj\ instance\ of\ T\ s.t.\ isActive(EJB(T)), isPoolSizeLow(SERVER)$	$EjbClass(EJB(T)).ejbPassivate()$
R10	$isRemoteInterface(T), \quad isEntity(EjbClass(EJB(T)),$ $\forall\ obj\ instance\ of\ T\ s.t.\ isPassive(EJB(T))$	$T.ejbLoad()$
R11	$isRemoteInterface(T), \quad isEntity(EjbClass(EJB(T)),$ $m = create(paramlist), \quad \forall\ obj\ instance\ of\ T\ s.t.\ isActive(EJB(T))$	$T.ejbStore()$

does not make sense because CustomerHome is an *interface*, and the actual call, as indirect as it may be (through the RMI infrastructure) is actually between CustomerHome_proxy.create(...) and CustomerEJB.postCreate(...). But CustomerHome_Proxy is only generated at EJB deployment time, and the real dependency, signature wise, is at the interface level. This call dependency enables us to link MyClient.foo(...) to CustomerEJB.postCreate(...) in two hops, which does link the two methods, and keeps client code immediately dependent on the *interfaces*, as it should.

Using similar principles, Rule R2, by linking methods of the *remote interface* to methods of the EJB class, in effect links client-side invocations of methods of the remote interface, to their service-side implementations–going through the remote interface as an intermediary step.

C. Life-cycle Management Services

Section III-B explained that calls to create(...) on the home interface result in calls to ejbCreate(...) on the bean class. Consider the lifecycle of an *entity bean* as described in Fig. 3. In fact, the calls to create(...) on the home interface also results in the call to the method setEntityContext(ctx:EntityContext) which sets the entity context. In the other hand, the ejbPostCreate(args) method, if present, is called *after* ejbCreate(...). Likewise, calls to remove(...) on the home interface result in invocations of ejbRemove(...) and unsetEntityContext methods. This is captured by rules R3 to R7 of Table I. Rules R3, R4 and R6 are specialized versions of rule R1 (see Section III-A). Whereas R1 deals with all the methods of the home interface (including all the *finders*), rules R3, R4 and R6 deal specifically with create and remove methods of the home interface.

We now look at the behavior of ejbActivate and ejbPassivate. These methods behave differently, depending on whether the EJB is an *entity bean* or a *stateful session bean*. We explained the behavior of these methods for stateful session beans in section III-B, which is the more complex of the two. For *entity beans*, the ejbActivate method is called when a bean instance is pulled from the pool and assigned to an EJB object whereas ejbPassivate is called when the bean instance is returned to the pool. Developers can use the ejbActivate method to initialize *non-persistent fields* and acquire whichever resources the bean needs, and the ejbPassivate method to release those resources.

We mentioned in III-B that the call dependencies to ejbActivate and ejbPassivate for stateful session bean are examples of *guarded dependencies* in the sense that the call relationship between a method $T.m(...)$ and $C.ejbPassivate(...)$ depends on the *server alone, based on its internal state*, whereas a call to a create(...) method on the home interface will *always* result into the invocation of the corresponding ejbCreate(...) on the bean class. This is illustrated in rules R8 (activate) and R9 (passivate). These rules use a number of predicates that can only be evaluated at run-time, and hence the name *guarded*:

- $isPassive(obj)$ is a predicate that returns true if object *obj* has been passivated. For an entity bean, it means 'pooled' state, and for a stateful session bean, it means 'passive' or swapped out (see Fig. 3).
- $isActive(obj)$ is a predicate returns true if object *obj* is in a ready state (see Fig. 3).
- $isPoolSizeLow(SERVER)$ is a predicate that indicates that the server pool size is below a given threshold.

Because our rules are evaluated against the structural representation of the code derived from its *static analysis*, these predicates, which refer to individual objects and run-time state *cannot be evaluated*. Thus, in the actual implementation (Section V), they are not used as conditions parts: they are used as *annotations* or *decorations* on the dependency links that are added to the KDM model so that we can distinguish them from other call dependencies (We envision run-time scenarios where they could be evaluated as a debugging aid to compare expected/theoretical behavior, to actual one. It is beyond the scope of this paper).

Another important aspect when dealing with an *entity bean* is the synchronization of its persistent fields with the values stored in the database. The EJB container synchronizes these fields by invoking the ejbLoad and the ejbStore methods of the EntityBean interface. The ejbLoad method refreshes the fields from the database while the ejbStore method writes the fields to the database: In container managed persistence, the ejbLoad notifies the bean instance that it has been synchronized with the contents of the database. With bean managed persistence (BMP), ejbLoad synchronizes the fields of the bean instance from the database. The ejbStore method is called *after* the bean instance has been saved to notify the bean, whereas with BMP, it is this method that does the saving.

There are several "moments" for these methods. For ejbLoad: (1) Following a bean instance activation, when a bean instance moves from pooled state to a ready state (see Fig. 3); (2) At the beginning of a transaction. If the EJB object is included within a transaction scope, the ejbLoad method is called at the beginning of the transaction; and, (3) At the beginning of business methods. If the business method is transaction enabled, this reduces to the previous case. However, transaction or not, we would always want the business methods to execute on *current/up-to-date* object state. It turns out that the J2EE standard does *not* dictate *how often* ejbLoad is called–i.e. at every business method call, or more parsimoniously–as long as objects stay current.

For ejbStore: (1) When the bean is passivated, the container calls ejbStore *first*, and *then* ejbPassivate (With bean managed persistence, ejbStore does the database synchronization/saving itself; with container managed persistence, the synchronization is done some other way, and ejbStore is only used to *notify* the bean that it has been saved); (2) At the end of a transaction; and, (3) As often as it needs to be called to make sure that the object stays up-to-date. The J2EE specification leaves it at the discretion of the vendors to decide how *often*.

Our goal is to characterize situations under which entity beans are activated, passivated or synchronized (i.e. when the EJB container calls ejbActivate, ejbPassivate, ejbLoad or ejbStore), and to try to link them to user actions. In general, activation happens when an EJB in the passive state receives a call to one of its methods, and passivation happens if (1) an EJB is idle and (2) the pool size falls below a given threshold. Rules R8 to R11 of Table I formalize these dependencies.

V. Tool Implementation

Our implementation relies on the interoperable, language-independent and extensible Knowledge Discovery Metamodel (KDM) to represent the various artifacts that compose J2EE applications. We use MoDISCO, a KDM-compliant Eclipse-based re-engineering toolkit to build a KDM representation of a J2EE application, and RedHat's DROOLS flexible and reusable rule language to codify the J2EE applications-specific dependency detection rules using a language to represent application elements, and to execute those rules on the KDM representation of the applications. Fig. 4 shows the architecture of our tool: dark gray boxes represent preexisting components that we (re)use.

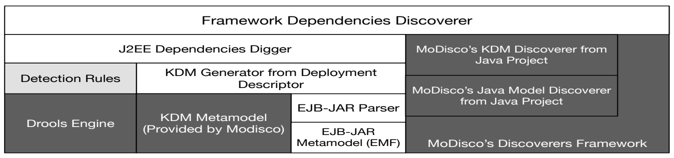

Framework Dependencies Discoverer
J2EE Dependencies Digger
Detection Rules
KDM Generator from Deployment Descriptor
Drools Engine
KDM Metamodel (Provided by Modisco)
EJB-JAR Parser
EJB-JAR Metamodel (EMF)
MoDisco's KDM Discoverer from Java Project
MoDisco's Java Model Discoverer from Java Project
MoDisco's Discoverers Framework

Fig. 4: Our tool architecture

a) Building a KDM Representation of J2EE Applications: We extend MoDISCO, that implements the KDM metamodel using the Eclipse Modeling Framework (EMF) to handle any kind of software artifact by (1) implementing a KDM-compliant representation of the artifact and (2) developing a tool to build KDM-compliant representations of applications, called a *discoverer*. MoDISCO comes with metamodels and discoverers for Java and other common programming languages. We develop metamodels and discoverers for other, non-Java artifacts that compose J2EE applications, like EJB deployment descriptors. The KDM metamodel is generic, language-independent, and granular, so we must implement a J2EE-specific *domain specific language* (DSL) on top of the KDM metamodel to ease analyzing J2EE application models. In particular, this DSL allows the representation of relationships between beans classes and remote or local interfaces and homes.

b) Codifying the Dependency Detection Rules: We encode the dependency detection rules using typical **when** <condition> **then** <action> rules, where the condition represents a KDM model patterns and the action adds a dependency to the KDM model. Fig. 4 shows the set of rules as the light-gray component. We provide the complete set of DROOLS rules in our companion Web site (https://github.com/CodifyingHiddenDependenciesJ2EE18/Replication).

c) Executing the Dependency Detection Rules: We use the DROOLS engine to execute the rules. The engine takes as input: (1) a set of rules and (2) a set of data objects, which come from KDM model of the application. It then matches the condition parts of the rules against the KDM model of the application. When a match occurs, i.e., a hidden dependency exists, it executes the action part of the rule to add that dependency to the KDM model. To add the dependency we complete the call graph in the KDM model by adding hidden dependencies as calls between methods. Listing 1 shows a sample rule to detect invocations of create(...) methods on the home interface of an EJB from within client application to create a dependency between the invoking EJB client application to the corresponding ejbCreate(...) method of the bean class. The lines after the *when* clause (lines 3-5) represent the condition of the rule (i.e., the KDM model pattern to match), while lines following the *then* clause show the actions to add the dependency (lines 7-11).

VI. Evaluation

In this section, we present the evaluation of our approach. First, we present the applications we selected for our validation. Then, we demonstrate the correctness and precision of our implementation in a technical validation. Afterwards, we perform an empirical study to show that hidden dependencies can be present in significant proportion in J2EE applications. Finally, we illustrate the usefulness of codifying hidden dependencies for maintenance tasks such as change impact analysis.

A. Data Selection

We selected 10 applications for our evaluation, presented in Table II, links to download them are available in our

Listing 1: Rule to detect EJB invocation and linking it to the corresponding bean ejbCreate(...) method.

```
rule "Bean Create"
 when
  $ejbHomeInterface : InterfaceUnitAdapter( isExtends("javax.ejb.EJBLocalHome") ||
          isExtends("javax.ejb.EJBHome"))
  $toMethod: MethodUnitAdapter ( getEnclosingInterface() == $ejbHomeInterface,
         isNamedLike("(create)(.*)") )
  $call : CallsAdapter( toMethod == $toMethod )
 then
  ClassUnitAdapter beanClass = $ejbHomeInterface.getRlatedCodeUnitByStereotype("
         BeanClass");
  if(beanClass!=null){
   MethodUnitAdapter method4NewCall = beanClass.getMethodByNameAndParameters("
          ejbCreate", $toMethod.getSignatureParameters());
    $call.createSiblingCallTo(method4NewCall);
  }
end
```

replication package. Interesting open-source applications using EJB2 are hard to find because the usage of J2EE is mostly relevant for companies [14], which rarely publish the proprietary source code of their legacy applications. Also, online repositories, like GitHub, were created after the release of EJB v3 and, thus, contain few instances of EJB v2 applications. Therefore, to find these 10 applications, we queried *SearchCode*, *SourceForge*, *Google*, and *GitHub* with queries for the presence of beans or deployment descriptors, which led us to collect more than 100 applications. Then, we rejected applications with no deployment descriptors or with missing libraries. We also rejected simple tutorial applications with only one bean and few classes. Consequently, we retained 10 applications. Our goal was to cover as many beans usage scenario as possible, so our set contains applications of different size, from various domains and developers, and with different levels of dependency on beans during execution. In these 10 applications, we retained EJBPool, which we used to test the implementation of our approach and which contains one of each type of hidden dependencies.

B. Technical Validation

We assessed our approach with a manual validation of the results provided by our implementation on all the applications presented in Table II. We want to assess whether our implementation is correct and that it identifies all and only the existing hidden dependencies. Our implementation adds each dependency as a call instance in the KDM model of the analyzed application. Each rules is executed once. We manually analyzed the Java source-code and the deployment descriptors of each application to identify hidden dependencies and compared the results of our analysis with the obtained KDM model. Such a manual analysis is time and effort consuming so we only evaluated the expected missing hidden dependencies for applications with small numbers of deployed beans and calls leading to their home and remote interfaces. We explored carefully all these calls for the applications task6, EJBPool, vaza, and doalist. We manually found a total of 352 hidden call dependencies. Our tool found all and only all the expected calls. Following, we computed the precision of our implementation and, hence, indirectly, of our approach, by manually inspecting 50 random call dependencies found by our tool applied on Java Petstore, Changeset, osv45, Springstore, Creezo and all of the 39 call dependencies found in mqbuffer. We obtained a precision of 1 because all of the 289 call dependencies added by our tool were legitimate.

Thus, we confirmed that our implementation and, indirectly, our approach is correct and precise.

C. Empirical Validation

We performed an empirical validation to evaluate to what extent the studied hidden dependencies can be present in applications relying on EJB2. We applied our tool on the applications in Table II, whose results are presented in Table III. The first part of the table shows *unguarded dependencies*, i.e., hidden call dependencies independent of runtime conditions. The remaining results are *guarded dependencies*, dependent on runtime conditions, such as container behaviors, bean pool sizes, numbers of clients, etc. Some of these call dependencies may not be observed during all executions but would help developers in debugging and maintenance, see Section I. Here, the given percentage are computed with the number of non-hidden dependencies as reference.

On average, we add 8% of *unguarded dependencies*. The results vary between applications: 0.3% for mqbuffer but 19% for doalist and 16% for osv45 (adding from 3,872 calls to the 24,321 found by MoDISCO). These results depend on the use of the beans in the logic of the applications. They are high when beans expose many methods via their remote interfaces, for example in crezoo and osv45, and low if an application use few beans.

While analyzing the identified hidden dependencies, we observed many interesting facts. First, the method ejbPostCreate() is only useful for Entity beans and not mandatory to be implemented and, as expected, we observed that it is not always implemented by bean classes. In all the applications, client code rarely invokes the remove() method of the home interface. In 6 of the 10 applications, there are no call to this method. Consequently, unsetEntityContext() is also uncommon because it is only invoked before remove() on Entity beans. Application tend to use mostly Entity beans or Session beans, but rarely both in similar quantity, except PetStore, which showcases J2EE technology.

Concerning *guarded dependencies*, we observe that they also depend on the types of beans used but on average we add 6.7% of such dependencies. Activation and passivation-related methods can only be invoked on entity or stateful session beans. Activation may happen every time there is a call to a remote method of these types of beans whereas passivation may happen when there is a creation of this kind of entities (for lack of space left in the pool). Loading and storing-related methods are specific to entity beans. Loading may happen every time there is a call to a remove method on an entity bean, hence high numbers for creezoo and osv45. If an application has only entity and stateless session bean (e.g., changeset, crezoo, doalist, osv45, and vaza), then the number of calls to activate() is equal to that of load() and that of passivate() is equal to that of store(). We observe that stateful session bean are quite rare. Some applications (mqbuffer and task6) have only stateless session beans and hence no *guarded dependencies*.

For all dependencies combined, we add around 15% of extra hidden dependencies which would have not been considered by classic static analysis, and thus could not be exploited by tools and developers during maintenance or evolution tasks.

TABLE II: Characteristics of the Analyzed Applications

Applications	**Classes/Interfaces**	**Methods**	**Calls**	**Beans**
Changeset	385	2,625	9,889	13
Crezoo	574	4,198	14,263	36
Doalist	21	186	246	4
EJBPool	21	83	75	4
Mqbuffer	371	3,121	13,576	16
Osv45	694	7,669	24,321	24
Petstore	346	1,964	5,664	35
Springstore	317	1,748	3,104	15
Task6	45	161	238	5
Vaza	239	1,874	6,242	3

D. Evaluation on Change Impact Analysis

To demonstrate the usefulness of codifying hidden dependencies for maintenance tasks, we compare the information provided by a model with no hidden dependencies to our model with hidden dependencies for change impact analysis. We compare the numbers of classes reachable through change propagation with JRipples with both models.

JRipples is an Eclipse plugin supporting developers during change impact analysis. It is semi-automated because it relies on the developers' feedback for its different phases. Each phase is divided into steps in which developers evaluate if a class is impacted and if a change should be propagated or not. If developers mark a class as impacted or the change to be propagated, then JRipples proposes to evaluate neighboring classes iteratively. JRipples consider two classes neighbor if there is a call dependency between these two classes, independent of the call direction. Prior to this phase, developers can filter and group classes into concepts for finer analysis. In the following, we do not filter/group classes for a fair comparison.

When hidden dependencies are not codified in the model, some classes may be missed during the change impact analysis, as illustrated in Fig. 5. We observe that, starting from $\mathrm{CustomerEJB}$, the dependencies to $\mathrm{Account}$ and $\mathrm{AccountHome}$ are detected by JRipples but not the entity Bean $\mathrm{AccountEjb}$ although it exists: for example, when the method $\mathrm{Account.getDetails()}$ is called, then $\mathrm{AccountEJB.getDetails()}$ is effectively executed. Consequently, in a next step of the analysis, all classes with dependencies (hidden or not) to $\mathrm{AccountEJB}$, like $\mathrm{ContactInformation}$, would be missed (and iteratively their own dependencies in the following steps). Missing these dependencies may hinder the developers' maintenance task by requiring *them* to remember the dependency between classes and propagate changes to $\mathrm{CustomerEJB}$ and $\mathrm{ContactInformation}$.

To evaluate how hidden dependencies may improve the results of change impact analysis, we perform such an analysis starting from each classes of the applications presented in Table II. From each starting classes, we construct a graph of classes explored with a condition on the depth n of the

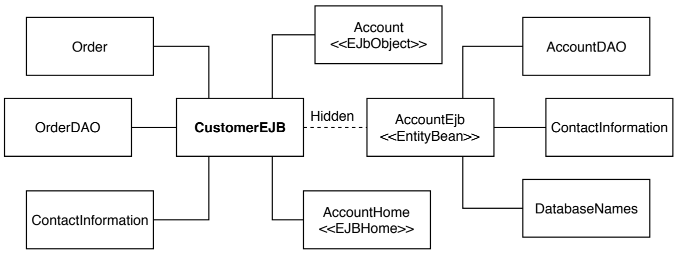


Fig. 5: Example of classes possibly explored during change propagation from the class $\mathrm{customerEJB}$ in the Java Petstore application

exploration: with $n = 2$, we explore the direct neighbors of the starting class, with $n = 3$, we also explore the neighbors of the neighbors...

Some of the hidden dependencies are redundant because we consider only classes, not particular methods. Also, we do not have to distinguish between *guarded* and *unguarded dependencies* because there is at least one *unguarded dependency* for each *guarded dependency* at the class level. Hence, they are redundant but it does not affect the results.

Table IV shows the results with and without hidden dependencies with n the maximum depth, i.e., the maximum numbers of classes reachable from any starting class. Following, # explo. is the number of explorations performed for each application and Improv. the percentage of these explorations with hidden dependencies. Overall, hidden dependencies improve by 42.7% the explorations.

Globally, the more an application relies on beans for its functionalities, the more the results of change propagations improve as shown in the columns showing the average numbers of classes in Table IV. For all applications, on average, 5.7% more classes are explored with hidden dependencies from 0.3% for mqbuffer up to 127.9% for doalist.

When looking at specific application and specific depth, we observe that the results are improved with the hidden dependencies, for example for PetStore, as presented in the top of Fig. 6. As the depth increases, the differences of numbers of classes found through hidden dependencies or not decreases. However, these differences may never become null, confirming that some classes can only be reached through hidden dependencies implied by the platform. For the three applications, changeset, crezoo, and mqbuffer, we also observe a convergence of the numbers after a certain depth as illustrated for Changeset, which may happen when some utility classes, or POJO data classes, are used both by the client and the server: at some depth, the beans are reached indirectly even without the hidden dependencies. The results with hidden dependencies would be better with unidirectional exploration of the graph, for example the call hierarchy provided by Eclipse, because there would be no indirect way to reach the bean from the client (and vice versa). Fig. 7 provides such an example for the method $\mathrm{AccountEJB.getDetails()}$ of Petstore.

TABLE III: Number of hidden dependencies found in the dataset

	Unguarded dependencies								Guarded dependencies				
	R1 Create	**R2 RMI**	**R3 Remove**	**R4 PostCreate**	**R5 Session**	**R6 Entity**	**R7 Unset**	**Total**	**R8 Activate**	**R9 Passivate**	**R10 Load**	**R11 Store**	**Total**
Petstore	43	162	0	28	18	25	0	276	141	29	115	25	310
Changeset	13	381	22	11	2	11	22	462	247	11	247	11	516
Osv45	24	3740	30	24	0	24	30	3872	3740	24	3740	24	7528
Task6	5	7	0	0	5	0	0	17	0	0	0	0	0
Mqbuffer	16	7	0	0	16	0	0	39	0	0	0	0	0
Springstore	15	39	0	14	5	14	0	87	22	15	21	14	72
EJBPool	7	6	3	3	2	1	1	23	3	2	2	1	8
Vaza	61	66	15	5	56	5	5	213	5	5	5	5	20
Doalist	4	40	0	2	1	2	0	49	33	2	33	2	70
Crezoo	45	1368	0	43	2	43	0	1501	982	43	982	43	2050

TABLE IV: Summary of the Results for Change Impact

		Graph		Average # of classes	
	n	**# explo.**	**Improv.**	**No hidden**	**With hidden**
Changeset	10	3465	20.5%	123.8	125.0 **(1%)**
Crezoo	9	4,592	35.7%	262.3	272.2 **(3.8%)**
Doalist	4	63	77.8%	5.6	12.8 **(127.9%)**
EJBPool	5	84	79.8%	2.1	3.8 **(82.8 %)**
mqbuffer	11	3,710	11.8%	119.7	120.1 **(0.3%)**
osv45	9	4,088	70.5%	338.8	353.9 **(4.5%)**
Petstore	14	4459	61.7%	101.7	119.1 **(17.1%)**
Springstore	14	4,121	40.6%	49.0	54.8 **(11.9%)**
Task6	6	225	24.9%	4.3	4.7 **(7.4%)**
Vaza	10	2,133	58.7%	92.8	95.9 **(3.4%)**

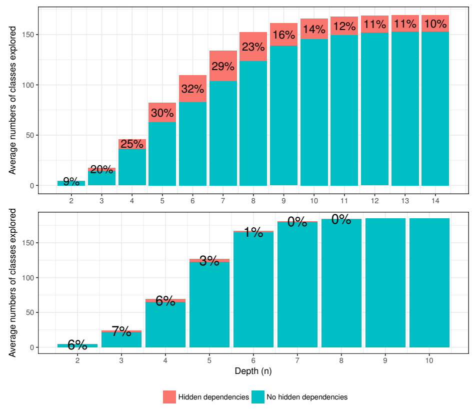
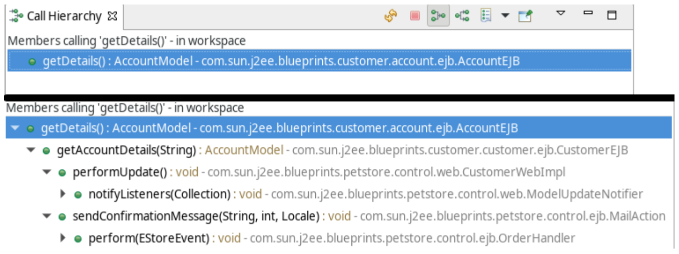


Fig. 6: Change propagation on Petstore and Changeset with rounded percentage of improvements



```
Members calling 'getDetails()' - in workspace
  getDetails() : AccountModel - com.sun.j2ee.blueprints.customer.account.ejb.AccountEJB
```

```
Members calling 'getDetails()' - in workspace
getDetails() : AccountModel - com.sun.j2ee.blueprints.customer.account.ejb.AccountEJB
  getAccountDetails(String) : AccountModel - com.sun.j2ee.blueprints.customer.customer.ejb.CustomerEJB
    performUpdate() : void - com.sun.j2ee.blueprints.petstore.control.web.CustomerWebImpl
      notifyListeners(Collection) : void - com.sun.j2ee.blueprints.petstore.control.web.ModelUpdateNotifier
    sendConfirmationMessage(String, int, Locale) : void - com.sun.j2ee.blueprints.petstore.control.ejb.MailAction
      perform(EStoreEvent) : void - com.sun.j2ee.blueprints.petstore.control.ejb.OrderHandler
```

Fig. 7: Call hierarchy from a bean method in Eclipse without (top) and with (bottom) hidden dependencies

VII. Threats to Validity

Internal Validity: We cannot generalize the recall or precision values to any J2EE applications. However, during our manual validation, we found all the expected hidden dependencies and no more. It is possible that both our tool and our manual analysis missed some dependencies but we accept this threat as unlikely. Our approach does not currently apply to third-party J2EE frameworks, like Spring.

Construct and Reliability Validity: Our tools relies on MoDISCO and is dependent of its models and of its precision and recall. We accept this threat because MoDISCO has been used (and presumably debugged) extensively. We provide all the necessary details to replicate our validation on-line (https://github.com/CodifyingHiddenDependenciesJ2EE18/Replication), including the rules and open-source J2EE applications.

External Validity: We only considered 10 pre-EJB 3, J2EE applications. We do not and cannot cover all existing hidden dependencies that may exist in J2EE applications if they use third-party frameworks. However, our approach is extensible (1) to new rules and (2) to new technologies thanks to its use of the KDM metamodel. Concerning our comparison with JRipples, we are not aware of the existence of change-impact analyses that consider hidden dependencies.

VIII. Related Work

Several research works exist in the literature on the extraction of program dependencies of software systems. Some of these works relied on KDM to model the reverse-engineered systems and represent their dependencies. For example, Yazdanshenas and Moonen [15] built a homogeneous KDM model from heterogeneous component-based systems developed in C, C++, Java and–or with configuration files in XML. They used these models to represent intra- and inter-component dependency graphs. However, they did not consider hidden dependencies among components, assuming that all components are given and that dependencies are explicit in their source code. Following the work by Yazdanshenas and Moonen [15], Shatnawi et al. [16] relied on KDM model of J2EE applications to analyze Servlets and JSPs. They described a set of difficulties in analyzing such applications and proposed DeJEE, an approach and a tool to build dependency graphs for J2EE applications. They focused only on JSP and servlets and did not consider bean classes and their hidden dependencies with the platform. Ricca et al. [17] relied on dynamic analyses to extract dependencies between PHP-based servers and their JavaScript clients to extract the program elements relevant to a specified computation. Also, Lucca et al. [18] proposed a reverse-engineering tool to recover UML class diagrams, sequence diagrams, and use-case diagrams from Web applications. However, these two works did not provide support for explicit and hidden dependencies between distributed components. Naumovich et al. [19] proposed an approach to analyze statically J2EE access-control policies of security-sensitive fields of server-side objects, such EJBs. They only considered EJB fields that are accessed or modified (directly or indirectly) by a EJB method to control for access. They did not generalize their approach to all hidden dependencies. Perin [20] introduced a meta-model and reverse-engineering techniques to model the components of multilanguage systems, focusing on J2EE applications. They used the proposed techniques to map databases and transactions flows and to identify hidden dependencies. However, they did not cover all types of J2EE dependencies and only rely on inferring dependencies starting from the direct connections defined by the programmer. Kirkegaard et al. [21] proposed a context-free grammar to analyze output streams of Servlets and JSP pages and verify their conformance to specifications. They showed that their proposed grammar is extensible to other programming languages. However, they did not validate it on other systems and did not consider hidden dependencies. Pinzger et al. [22] proposed a tool to represent static dependencies in source code of Java systems. They proposed visualization and navigation techniques but did not analyze hidden dependencies. Some other research works studied dependencies between specific combinations of programming language, for example through the Java Native Interface, which bridges Java and C/C++ code [23], [24], [25], [26]. They studied the explicit dependencies that arise from the use of these specific combinations and the, typically, native interfaces between them.

IX. Conclusion

Legacy J2EE applications are multi-tier, multi-language applications that rely on the J2EE platform to benefit from infrastructural and architectural services. The platform relieves developers from the need to design, implement, test, and maintain these services but also hide many of the *control dependencies* between the components of the platform and the applications. Indeed, they use inversion of control and various late-binding techniques that prevent developers and current static code-analysis tools to identify and understand these hidden dependencies.

We proposed a new approach to static code-analysis applied to J2EE containers in which we specify hidden dependencies and build models of J2EE applications, including both explicit and hidden dependencies. Our approach uses call graphs, produced by a static-code analysis, which we do complement by adding, programmatically, the control dependencies inherent in the platform.

We described our approach and its implementation, which uses MoDISCO [12], an Eclipse-based open-source implementation of the KDM. We evaluated our approach by applying it to J2EE applications by applying our tools on 10 open-source applications and showed that hidden dependencies are substantially present in the analyzed applications with an average of 15% new dependencies added compared to traditional static analysis tool. We also showed that codifying these dependencies may help developers during maintenance tasks such as change impact analysis (around 6% more classes can be explored) or tool-based refactoring.

In future work, we want to exploit the extensibility of our approach to codify the hidden dependencies of frameworks like Spring and possibly other languages. Moreover, we want to exploit our model to develop tools to help practitioners to modernize and debug their applications.

Acknowledgements : This work is supported by the Fonds Québécois de Recherches - Nature et Technologies. The first author is also supported by the Proyecto FONDECYT de Postdoctorado N3180561.

References

[1] F. Perin, T. Grba, and O. Nierstrasz, "Recovery and analysis of transaction scope from scattered information in java enterprise applications," in *IEEE ICSM*, 2010, pp. 1–10.

[2] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big"' web services: Making the right architectural decision," ser. WWW 2008. ACM, pp. 805–814.

[3] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas, "Towards the understanding and evolution of monolithic applications as microservices," in *CLEI 2016*. IEEE, pp. 1–11.

[4] R. Rodríguez-Echeverría, F. Maclas, V. M. Pavón, J. M. Conejero, and F. Sánchez-Figueroa, "Generating a rest service layer from a legacy system," in *Information System Development*, 2014, pp. 433–444.

[5] B. Li, Q. Zhang, X. Sun, and H. Leung, "Wave-cia: A novel cia approach based on call graph mining," 2013, pp. 1000 – 1005.

[6] V. Musco, M. Monperrus, and P. Preux, "A large-scale study of call graph-based impact prediction using mutation testing," *Software Quality Journal*, vol. 25, no. 3, pp. 921–950, 2017.

[7] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "Jripples: a tool for program comprehension during incremental change," in *IWPC 2005*.

[8] D. H. Vo, "Jcia: A tool for change impact analysis of java ee applications," in *Information Systems Design and Intelligent Applications: INDIA 2017*, vol. 672. Springer, 2018, p. 105.

[9] A. D. Ionita, A. D. Ionita, M. Litoiu, and G. Lewis, *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, 2012.

[10] R. Khadka, A. Saeidi, S. Jansen, J. Hage, and G. P. Haas, "Migrating a large scale legacy application to soa: Challenges and lessons learned," in *WCRE 2013*, Oct 2013, pp. 425–432.

[11] G. E. Boussaidi, A. B. Belle, S. Vaucher, and H. Mili, "Reconstructing architectural views from legacy systems," in *WCRE 2012*.

[12] H. Brunelire, J. Cabot, G. Dup, and F. Madiot, "Modisco: A model driven reverse engineering framework," *IST 2014*, pp. 1012 – 1032.

[13] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *SIGSOFT*, 1984.

[14] A. F. Riquelme, "Are Java and Java Enterprise Edition Still Relevant Technologies?" https://bit.ly/2KXZ3bO, 2018, [accessed April-2018].

[15] A. Yazdanshenas *et al.*, "Crossing the boundaries while analyzing heterogeneous component-based software systems," in *ICSM*, 2011.

[16] A. Shatnawi, H. Mili, G. E. Boussaidi, A. Boubaker, Y. G. Guhneuc, N. Moha, J. Privat, and M. Abdellatif, "Analyzing program dependencies in java ee applications," in *MSR 2017*, 2017.

[17] F. Ricca and P. Tonella, "Construction of the system dependence graph for web application slicing," in *SCAM 2002*.

[18] G. A. D. Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. D. Carlini, "Ware: a tool for the reverse engineering of web applications," in *CSMR*, 2002, pp. 241–250.

[19] G. Naumovich *et al.*, "Static analysis of role-based access control in j2ee applications," *ACM Software Engineering Notes*, 2004.

[20] F. Perin, "Reverse engineering heterogeneous applications," Ph.D. dissertation, Universitt Bern, 2012.

[21] C. Kirkegaard and A. Møller, "Static analysis for java servlets and jsp," in *SAS 2006*.

[22] M. Pinzger, K. Graefenhain, P. Knab, and H. C. Gall, "A tool for visual understanding of source code dependencies," in *ICPC 2008*.

[23] D. L. Moise and K. Wong, "Extracting and representing cross-language dependencies in diverse software systems," in *WCRE*, 2005.

[24] M. Furr and J. S. Foster, "Polymorphic type inference for the jni," in *European Symposium on Programming*. Springer, 2006, pp. 309–324.

[25] P. K. Linos, Z. hong Chen, S. Berrier, and B. O'Rourke, "A tool for understanding multi-language program dependencies," in *IWPC 2003*.

[26] L. Deruelle, N. Melab, M. Bouneffa, and H. Basson, "Analysis and manipulation of distributed multi-language software code," in *SCAM*, 2001.