Static Discovery and Remediation of Code-Embedded Resource Dependencies

Nikolai Joukov*, Vasily Tarasov†, Joel Ossher‡, Birgit Pfitzmann*, Sergej Chicherin§, Marco Pistoia* and Takaaki Tateishi¶

*IBM T.J. Watson Research Center, Hawthorne, NY USA
†Computer Science Department, Stony Brook University, NY USA
‡Informatics Department, UC Irvine, Irvine, CA USA
§IBM Russian Systems and Technology Laboratory, Moscow, Russia
¶IBM Tokyo Research Laboratory, Tokyo, Japan

***Abstract*—Many enterprises perform data-center transformation, consolidation, and migration in order to improve the efficiency of their IT infrastructures. These transformation projects begin with the discovery of the existing infrastructure, in particular the dependencies between applications. These dependencies are needed in planning, in order to determine how components influence one another, and in relinking, so that the component names and addresses can be updated. Typically, dependency discovery is done by network monitoring and middleware configuration analysis. These existing approaches will often fail to detect dependencies expressed in the application code.**

In this paper, we present the first method and tool for automatically identifying code-embedded external dependencies in Java Enterprise Edition applications. In addition, our tool can automatically alter the application code to update the dependencies, or externalize them to configuration files. We analyzed over 1000 Java EE applications from three enterprise environments. The results demonstrate the prevalence of code-embedded dependencies that would otherwise have to be identified manually, often causing failures during user-acceptance testing.

I. INTRODUCTION

Major enterprises are increasingly focused on improving the efficiency of their IT infrastructures. Data center transformation, IT optimization, consolidation, green projects, virtualization, migration to cloud; these are just some of the buzzwords associated with this trend. The recent economic downturn has further accelerated investment in infrastructure tranformation projects as a cost-cutting measure. In service management terms, the ITIL "service transition" area represents such projects.

Large-scale IT transformation projects generally follow the same sequence. They begin with a discovery phase, in which the configuration of the originating system is explored and documented. This is followed by a planning phase, where the target configuration is specified, and a plan is developed for performing the migration. Next is the actual migration, where server images, applications, databases, etc. are moved. Finally, the resulting system is tested.

Dependencies between applications are a major source of complexity in the transformation process. They must be discovered for planning, as one must know how components influence one another and how they can be grouped together for migration (which typically occurs over the course of several months). Dependencies are again needed during the migration itself, as the component names and addresses must be updated, in a process we call *relinking*.

In some cases, address changes can be hidden from applications through network-level measures, in particular DNS updates. This is not always effective. For example, IP addresses often change during a migration, yet many applications use them directly instead of relatively constant DNS names. DNS names can also change in mergers, acquisitions, and major enterprise restructuring. In other cases, applications that were co-located are split apart onto several virtual images, thereby requiring different DNS names. In yet other cases, applications, databases, or shared file systems are reorganized and change their names. Typically, when names or addresses change, enterprises prefer to update them directly (for the sake of cleanliness and maintainability), even when it is possible to handle both old and new DNS names, or put proxies in place to translate IP addresses. This has been the case in all of the large-scale migration projects that we have participated in. Even in cases where address updates are not planned, one still must discover all the dependencies during the planning phase in order, among other things, to know what applications will be down while certain components are being moved, and whether there may be application performance problems if some components are moved and others not.

Figure 1 shows a typical dependency structure for a candidate system for IT transformation. An arrow from Component A to Component B means that A depends on B. One might expect that dependencies as in Figure 1 would already be known, especially for production servers running business-critical applications. Yet in reality this is almost never the case. There are numerous reason for this, ranging from the inevitable accumulation of older-generation technologies to the haphazard integration of infrastructures after a merger or acquisition. Our team has been involved in many real-life transformation projects, and in every instance dependency discovery was required.

There are many commercial offerings for IT discovery. They

generally work by either monitoring the running system or statically exploring configuration files. When used in conjunction, these tools give a good picture of a deployed system. However, configuration-based approaches depend on the configuration files accurately capturing the dependencies, while monitoring-based approaches are ill-suited for discovering rare behavior, and provide little information to aid with relinking, i.e., little information about where an address that needs to be changed can be found.

In theory, configuration files should accurately represent the system configuration; in some cases they should even capture it completely. For example, according to the Java Enterprise Edition (Java EE) specification, all references to other components, servers, and network or storage resources should be placed in standardized configuration files. As Java EE progams are important to our migration scenarios, we decided to verify that the applications follow the specification. Initially, we semi-manually analyzed a number of deployed systems. Unfortunately, we discovered many instances where dependencies were either specified in non-standard resource files or wholly or partially specified in the code itself.

These dependencies pose a major problem to existing discovery systems. They cannot be captured by configuration-based approaches, and monitoring-based approaches face the issues described above. This can cause issues to turn up during the end-user acceptance test or even after cut-over, where they lead to time-consuming root-cause analysis and possible delays in the project schedule. Testing and remediation currently account for about $1/3$ of typical transformation costs, and a lot of that is spent on identifying dependencies that have not been fixed. Thus, any method to identify and relink these dependencies in the earlier transformation phases offers huge potential cost savings.

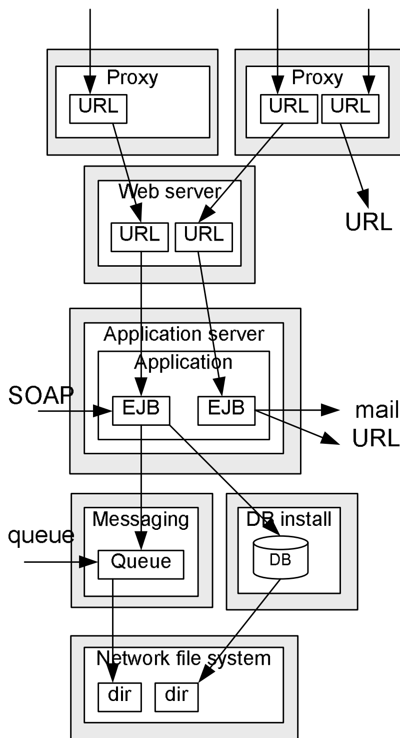

Fig. 1. Dependencies between software components. Grey boxes represent servers.

In this paper, we present the first tool for automatically detecting and relinking dependencies in standard Java and Java EE applications. Our tool detects dependencies that are entirely or partially specified in the code itself; we call this *code-embedded*. Hence it looks for instances where the actual code accesses the file system or network resources. For each access, our tool attempts to determine the target, which is specified directly through expressions based on string constants, indirectly through external resource files, or through a combination of both.

Our tool works on Java bytecode or source code, and understands standard packaging formats, such as jar, web archive (WAR) and enterprise archive (EAR). Given an application, it first identifies library calls that reference external resources. This includes accessing the file system, creating network connections, and invoking remote procedure calls. The parameters to these calls are then traced back to determine (a) what resources are being accessed and (b) where the references to the resources are stored. If a reference is stored in an external resource file, its location and the property accessed are reported. Our tool can also modify the code-embedded references to new addresses of the resources, and/or externalize them to a resource file. Our analysis is static, and is built by extending the open source T.J. Watson Libraries for Analysis (WALA) [22] and WALA-SA, a research system for string analysis [11]. We also enhanced Galapagos [18], a discovery tool we developed, to fetch application code and related configuration files and program environment data.

Our tool aids the discovery phase of transformation projects, augmenting static configuration discovery with information previously unavailable. It also simplifies the migration phase, as the information it provides allows migration engineers to more easily locate and change the non-standard configuration information. Finally, it shortens the testing phase because fewer errors will be left after the prior phases.

The remainder of this paper is organized as follows: We introduce the problem setting and provide a motivating example in Section II, followed by a review of the related work in Section III. In Section IV we present the details of our algorithms, and in Section V we discuss the capabilities and limitations of our approach. Section VI contains statistics about the occurance of different types of dependencies in over 1000 applications from real enterprise environments and the performance of our tool. We conclude in Section VII.

II. Motivating Examples

In this section, we show how our tool can be used, present details of the enterprise production code setting, and give examples of increasingly complex code dependencies on external resources.

A. Migration Scenario

Imagine you are a migration engineer in an IT services company. You are asked to migrate a retailers's HR software from their old data center to a new virtualized environment. You already use discovery tools including network observation and configuration analysis to discover significant information about the existing systems. Additionally, you now start our tool on the Java applications you discovered, to find code-embedded dependencies. Tyipcally, some of these you had not found at all so far. Others you may have known from network observation without knowing where they were defined. Using the overall information, you make a plan what needs to move where.

Now it is time to move ahead with the migration. Your other tools have already identified various configuration files you need to update, so you make those changes. You then load our tool's discovery results and switch it to migration mode. From the migration plan, you input a mapping of old to new addresses. Our tool makes the updates that it can do unambigously, and shows you any accesses where it was unable to determine exactly what to do. You try to solve those manually, possibly by discussions with application owners or interactive debugging. You are glad that at least you have far less of such work than without such a tool, and a lot less failed dependencies later.

B. Enterprise Production Environment

Figure 2 shows a program in a standard enterprise production environment. Solid arrows show environment elements that are always present, dashed arrows show elements that are typically but not always present. Some relevant external resources are shown in bold on the right. From the point of view of our analysis, *external* means everything outside of the runtime instance of the program.

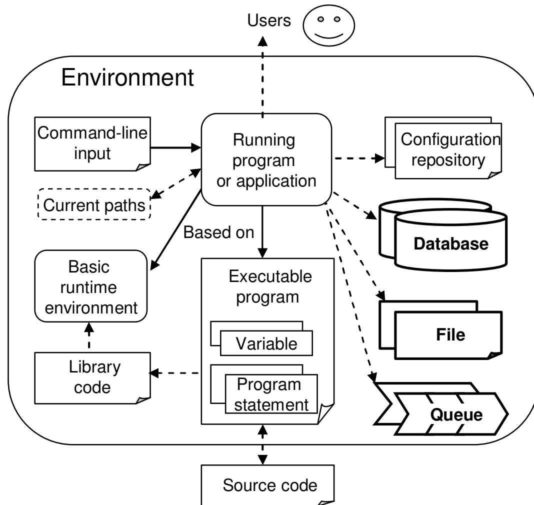

Fig. 2. An application with its environment

The program runs in a basic runtime environment, such as a Java VM or in enterprises typically a Java EE application server. It may depend on other external libraries. A *configuration repository* means any configuration files or configuration databases that the program may have. While command-line inputs and configurations can be changed, for typical long-running enterprise applications they are effectively static and thus we can use them in our analysis. Even if an enterprise program does not run continuously, it is typically restarted every time by a fixed script with the same configuration.

C. Examples of Code-Embedded Dependencies

External dependencies can be embedded in code as constants or loaded from external resource files. Figure 3 shows two examples of the first case. In both examples, the program attempts to access a file by opening a `FileInputStream` with a path that is directly specified in the code. For `f0`, the path to the file is given as a single constant string. This type of code-embedded dependency is easy to detect and fix, but, as we will see in Section VI, it is not very common. More commonly the path is constructed by multiple statements. For `f1`, the path is obtained by concatenating a constant string and the value of variable `name`. This value depends on another variable, `version`. Without string analysis, we would overapproximate the path as `/data/*`. With string analysis, we can properly compute the two possible paths as `/data/matrix.old` and `/data/matrix`.

```
FileInputStream f0, f1;
f0 = new FileInputStream("/data/matrix");
if (version == "old")
   name = "matrix.old";
else
   name = "matrix";
f1 = new FileInputStream("/data/" + name);
```

Fig. 3. Two examples of code-embedded file dependencies

```
Properties props = new Properties();
props.load(getClass().getResourceAsStream(
    "settings.properties"));
String db = props.getProperty("db.dbname");
Connection cn = DriverManager.getConnection(
    "jdbc:db2://" + db, "admin", "pwd");
```

Fig. 4. Dependency on a database via properties

Figure 4 shows an example of an indirectly specified dependency, where the resource path is itself stored in an external properties file. In a real application, all four statements would not necessarily be contiguous. In this example, `DriverManager.getConnection` attempts to open a database connection. Its first parameter is the database URL, which is the concatenation of two strings. As the value of `db` is not specified in the code, existing string analysis would overapproximate the URL as `jdbc:db2://*`, which is not very helpful.

Our extension of the string analysis goes further. It detects that the variable `db` is assigned a value from a properties file, because of the expression `props.getProperty("db.dbname")`. Now our analysis performs two additional steps. First, it determines the property name, here `"db.dbname"`, using string analysis on

the parameter to `getProperty`. Second, it resolves from which file `props` was loaded. We then load the properties file ourselves, e.g., the example file in Figure 5, and determine that the value of `db` is `sales`.

In this example, the properties file is loaded using `getResourceAsStream("settings.properties")` rather than from a specific path. This uses Java's class loader to locate the file. Similar approaches are common in portable programs. In order to locate `"settings.properties"`, our algorithm mimics Java's class loader behavior based on the run-time environment for the application.

```
# DATABASE configuration
db.dbname=sales
db.maxcon=5
db.mincon=2
```

Fig. 5. Java property file `settings.properties`

III. Related Work

Our work is related to two main areas: IT asset and dependency discovery and string analysis.

Today IT asset and dependency discovery tools are available from many vendors. They probe network nodes with requests [2, 9], monitor network traffic [4, 10, 15, 16, 23], or analyze software configurations [1, 3, 12, 13, 18, 21]. The configuration analysis is done for packaged middleware and applications such as databases, Java EE servers, and ERM systems. For Java EE servers, only the objects and relations explicitly configured at the server level are analyzed, such as which EJBs are deployed and what resources are declared. We are not aware of any tools like ours that perform discovery in actual code.

Automated modification of dependency configurations has been considered in SOA redeployment [17, 20]. Yet these approaches focus on configuration files, ignoring the code. For example, Sethi et al. [20] explicitly state that for Java EE applications, data source descriptions are changed. These are the Java EE-compliant configuration files that should be used, but that we have found often are not.

An early introduction to relinking can be found in [8]. However, their discovery is limited to activity time series inference, which does not enable automated relinking. In [7], again only externalized configurations are changed, and the changes are based on models from product designers rather than discovered source systems. In our use cases, no such models are available.

String analysis is a form of static program analysis used for inferring the string values that can arise at runtime, and is an instantiation of abstract interpretation [6]. Java-based string analysis was introduced by Christensen et al. [5, 14] in their Java String Analyzer (JSA). JSA approximates the value of a string expression with a regular language. Minamide [19] increased the precision of the analysis by using context-free languages. Geay et al. developed a string analysis for Java and Common Language Runtime (CLR) applications with a novel labeling feature for tracking the origin of each component in the resulting string [11]. Their prototype implementation was built on IBM's Watson Libraries for Analysis (WALA) [22]. It is this implementation that we extended for our work.

We are not aware of prior work that uses information about the runtime environment when performing static code analysis, nor of prior work on altering or externalizing existing code-embedded constants, as our relinking does.

IV. Static Dependency Discovery and Relinking

In this section, we describe our method for discovering and relinking code-embedded dependencies. It generalizes from the examples in Section II-C. Our tool proceeds in seven steps, which we describe in the following subsections.

A. Gather Code and Configuration Information

We mainly consider running production environments where little is known in advance. Hence, before analyzing an application, we must detect it and fetch its code and related state and configuration information. Techniques to detect applications include examining the currently running processes, registered packages, and standard installation paths. Disk scanning is an option if audit and performance constraints permit it. One can also look for processes that start others, e.g., `inetd` can start a program upon receiving a network request. None of these techniques alone is perfect, hence a combined approach is warrented. We have augmented Galapagos with several such features since our earlier publications.

Java EE applications usually store configuration files inside their Enterprise Archive (EAR) files, or somewhere on the `classpath`. By fetching the EAR files and files from the `classpath`, our Galapagos extensions obtain most of the related configuration files. In addition to code and configuration files, we also record some system parameters defining the application's environment: its root directory, environment variables, and command-line arguments. We also fetch the configuration of the underlying application servers where our Java code runs.

B. Construct Call Graph

Next we start static analysis by pointing our tool, i.e., extended WALA, to the gathered application archive files. One of these extensions was made to support the multi-level nested archive files common in Java EE applications.

We then generate call graphs for each module using WALA's standard propagation call graph builder. These call graphs are rooted at specific entrypoints. Proper entrypoint determination is important, as missed entrypoints result in sections of the program being considered dead and being ignored for the remainder of the analysis. For standard Java applications we include all *main* methods and static initializers. For Java EE, we also include Java Servlet methods entrypoints and entrypoints derived from the deployment descriptor.

C. Identify Stop Methods

We define a *stop method* as any method that provides access to an external resource. We distinguish direct stop methods,

which do this directly through native code, e.g., many methods in `FileInputStream`, from indirect stop methods, which do this through other Java methods, e.g., `FileReader`. We do not want to identify all indirect stop methods, as there are simply too many such methods. One approach would be to limit the identification to direct stop methods. This would be effective, as in theory every indirect stop method must somewhere invoke a direct stop method. However, this is rather confusing for users, as people are not generally familiar with the private native methods that actually implement the access functionality.

We therefore decided to identify any direct or indirect stop method in the public API for accessing external resources. As a direct native method can only be invoked via the public API, every access is captured. For example, `FileInputStream`'s private native method `open` is only invoked in the constructors of the class. So by identifying the constructors as stop methods, we catch every access to `open`.

An additional constraint is that we only consider methods that name the resource being accessed. For example, the constructors of `FileInputStream` take the path to the file as input, and so are useful stop methods, while the `read` method is not useful, as it does not provide any information about the resource. If the resource is not named in any method, then its location is likely specified in the configuration files that are analyzed by other approaches.

We annotate each stop method with the parameters that describe the location of the resource it accesses. In most cases, this location is expressed as a primitive type or a `String`. When this is not the case, such as the `FileInputStream` constructor that has a `File` as the input parameter, we do not include that method as a stop method. Instead, we use the method that creates the parameter type (in this case, the constructors to `File`).

So far, we identified stop methods in the Java Standard Library and some Java EE APIs. Our list of stop methods was compiled by hand by someone familiar with these libraries. The list includes many methods in the `java.io` and `java.net` packages, as well as methods relating to loading resources and creating database connections. Our tool can easily be adapted to treat any method as a stop method, and so can be extended to cover custom libraries or other frameworks.

Actually identifying the invocations of stop methods is a straightforward search through the call graph. As with most static analysis approaches, we cannot identify non-trivial invocations that use reflection. However, we can detect uses of reflection and alert the migration engineer of their locations.

D. Trace back invocation parameters

In this step, we analyze each stop method invocation to determine the provenance of the location-describing parameters. A parameter value can be derived from code-embedded constants (as in Figure 3), external resources (as in Figure 4), or a combination of both. We extended a WALA-based string analysis package [11] to perform this analysis.

Given a WALA call graph and a single parameter, the string analysis computes a context-free grammar (CFG) that overapproximates the parameter's value; it is a superset of all values that the parameter might take during execution. The CFG is annotated with labels describing the origin of each terminal. In many cases, the resulting CFG resolves to a single string or a set of strings, e.g., as we computed the two possible paths for `f1` from Figure 3.

As described earlier, in many situations the location of the resource is specified in an external configuration file. In the standard string analysis, such cases result in a wildcard, as the analysis does not know how to handle the methods accessing these configuration files. We extended the string analysis by adding transducers for those methods.

Transducers are used when the string analysis attempts to solve the grammar (initially derived from the single static assignment form of the bytecode) for a certain parameter. If the parameter value depends on a method invocation, the string analysis looks for a transducer for that method. The transducer rewrites the CFG to remove the method invocation while mimicking its behavior. For example, the `append` method for `StringBuffer` has a transducer that adds a production rule to the grammar concatenating the current value with the parameter's CFG value. If no transducer is found, a default transducer replaces the invocation with a wildcard.

We added transducers for methods like `getProperty` in `java.util.Properties`. These transducers record that an access has occured, and replace each invocation with a unique key string. In the example from Figure 4, the original string analysis would result in `jdbc:db2://*`. With the custom transducer added, the result is instead `jdbc:db2://$1`, where `$1` is the unique key string inserted by the transducer. It will be resolved in the next step.

E. Resolve configuration access

The final step in the discovery phase is to resolve the parameters whose values depend on external configuration files, such as `props.getProperty("db.dbname")` from Figure 4. For each access, we attempt to determine the location of the configuration file (`"settings.properties"`) and the part of it that was accessed (`"db.dbname"`).

A variant of the string analysis is used to accomplish this. To determine the portion of the configuration file being accessed, we trace back the parameter value just as in the previous section. For the example, this means identifying the code-embedded string `"db.dbname"`. To determine the location of the configuration file, we apply the analysis to the access' receiver object, here `props`. For example, if the access was an invocation of `getProperty` on a `Properties` object, we trace back the value of the `Properties` object. In this case, it means determining that `props` was loaded from `"settings.properties"`. While the object is not itself a string, we consider it to have the string value of the location it was loaded from.

Once the configuration file location and property name are known, we load the file to extract the possible values for the

property. In situations where the file location is fully specified, this step is straightforward. If only a relative path is given, we must use the information collected by Galapagos about the system's runtime configuration to reconstruct the full path. For cases where Java's Class Loader is used to load the resource, we mimic its behavior to find the resource on the classpath.

We built transducers for the methods of `Properties` and `ResourceBundles`, as those are the two main ways the Java Standard Library provides for handling configuration files. If a program uses another approach, our identification of stop methods in Step IV-C will still detect the loading of the configuration file, but we cannot automatically extract the correct parameter value from it yet.

F. Explore results

Typically at this point, the user will want to see the results, even if automatic relinking is also planned. The user can browse through the stop method invocations, sorted by invocation location or target. For each invocation, the user can see the possible values of its parameters, and where those values come from. If some or all of a value is externalized, the relevant configuration file and location is reported.

G. Correct code-embedded dependencies

Given the analysis results, the user may choose to automatically modify code-embedded dependencies. This may mean altering the addresses, externalizing the constants directly specified in the code, or both. The user chooses the type of change to be made, and provides a mapping from old to new addresses for resources that are being migrated. The tool compares this mapping with the discovered resources to generate an automated relinking plan. For the example in Figure 3, if the user wanted to map `"/data/matrix.old"` to `"/share/data/matrix.jan2010"` then our tool generates a suggested plan to change the constant `"/data/"` to `"/share/data/"` and `"matrix.old"` to `"matrix.jan2010"`. It also warns the user that this change results in `"/data/matrix"` now mapping to `"/share/data/matrix"`.

To generate these plans, we first identify the addresses computed by our analysis that have to change. For each of these, we compare the target address with the original address using a variant of the standard dynamic programming technique for computing edit distance. We break the original address into substrings based on how the address was constructed. In our example, the address gets broken into `"/data/"` and `"matrix.old"`. We then iterate through the substrings in reverse order, and use the edit distance table to pick the closest match for each substring. This results in the suggested mapping described above.

If the user chose to only update the addresses, we modify the class files and properties files to replace one constant value with another. Otherwise, we automatically externalize the constants to a resource file for easier future changes. We synthesize a class containing a static final field for each constant to be externalized. We then create a static initializer for this class that loads a properties file and initializes each field to the value specified in the properties file. This properties file contains the constant values either from the original program or the mapping plan. In the initial class files, we replace every relevant constant load instruction with a field load instruction, pointed at the corresponding field. This modification imposes a slight performance penalty on each access, and a large penalty when the constants are first loaded. In our testing, we have found that even for string-intensive long-running applications this impact is negligible.

In cases where the source code is available, we can automatically generate patches for the source code that implement the same behavior as our bytecode modifications.

V. Completeness and Correctness

Discovering precisely which resources a program accesses is undecidable. As a result, our tool is not complete; it is unable to discover every dependency. Fortunately, this does not negatively impact its usefulness. Our tool's main objective is to aid the discovery and relinking process in server migrations. No current tool in this domain is complete (whether they use static code analysis or not), and each contributes something unique. Every dependency we can automatically handle is one less that has to be found manually or that will turn up as a hard-to-trace fault during user-acceptance testing.

One reason for the lack of completeness is the use of reflection and native methods, which cause problems for static analysis techniques. Our tool flags all instances of reflection and native method invocation, so the user is aware of possible missed accesses.

Another issue related to completeness, as mentioned in Section IV-B, is that improper entrypoint determination will result in portions of the code being ignored. While in standard Java only `main` methods are entrypoints, Java EE is significantly more complicated. To partially mitigate the effect of missing entrypoints, our tool reports all stop method invocations that occur in dead code. To eliminate this issue entirely, we are exploring a form of entrypoint-less call graph construction using the type hierarchy. However, this will also lead to a loss in precision because all parameters of entrypoint methods must be assumed to have arbitrary values.

Another potential source of error is the classification of stop methods. We assume that the human classifying the APIs did it correctly. However, we do not claim that we classified all APIs a program might use. Our tool has the option to output all unknown external method calls, giving the user the opportunity to identify important methods that had been missed.

Our tool will correctly identify any possible resource paths for stop methods that it detects. For example, if it discovers an instantiation of `FileInputStream`, the list of potential paths will be complete, albeit sometimes an overapproximation using wildcards. This property is guaranteed by the underlying string analysis and by our new transducers, which also overapproximate.

When automatically remediating code-embedded dependencies, our tool would ideally do this without otherwise

impacting the semantics of the program. This is not possible in general, as one could always come up with a pathological case where the behavior of the program depended on the value of a string constant in bizzare ways. However, the effect of our approach is identical to someone changing those string constants manually. Additionally, our tool detects the effect of any changes on other external resource accesses, something a user would not be able to do manually.

We cannot automatically generate a relinking plan in every circumstance, but our tool fails gracefully and will detect any conflicts. It does guarantee the proper implementation of any specified plan.

VI. Evaluation

We examined three real-world enterprise environments, which we call `A`, `B`, and `C`. Environment `A` is the oldest; discovery was performed as part of its sunsetting. Discovery in Environments `B` and `C` was performed as part of optimization projects. Servers in environments `A` and `C` mostly run AIX, servers in `B` mostly Solaris OS, and a significant number of servers in all environments run Linux. During the discovery, we fetched all applications deployed on WebSphere application servers (WAS) and related system configuration information. Table I shows the statistics of the WAS installations we found in these environments, the application instances in them, and how many of these applications were unique. We ran our tool on each unique application.

	A	B	C	Total
WAS installations	45	17	204	266
Application instances	111	104	5040	5255
Unique applications	56	49	1034	1097

TABLE I
Statistics of the analyzed applications

A. Prevalence of Code-embedded Dependencies

Our first goal was to determine the prevalence of code-embedded dependencies in the three environments. Table II shows the percentage of applications in each environment that connect to messaging queues and databases using addresses not specified in standard configuration files. We were conservative in our detection, and ignored all dependencies not related to messaging queues or databases. In particular, we excluded dependencies on files inside the application archives, as addresses to such files do not need to be updated during migration. If we were to include additional dependency types, these percentages would only increase. Even using this conservative approach, we found code-embedded dependencies in between 30% and 90% of the applications we analyzed, depending on environment. This indicates that static discovery with prior tools would miss many dependencies.

There are interesting differences between the three environments. In environments `A` and `C`, most databases are referenced according to the Java EE specification, while most messaging queues are not. In environment `B`, it was evenly distributed.

	A	B	C
Messaging queues (%)	94	31	92
Databases (%)	7	25	5

TABLE II
Non-standard external dependencies in our environments

B. String Analysis

Our second goal was to determine the necessity of the string analysis. In particular, if stop method parameters were mostly simple string constants, as in `File("/data/data.xml")`, then the string analysis would not be needed. Table III shows the percentages of stop methods called using directly loaded string constants. The low percentages indicate that the string analysis is indeed needed to reconstruct the majority of parameter values.

	A	B	C
Directly loaded string constants (%)	8	10	7

TABLE III
Calls to stop functions with constant parameters

C. External Configuration Files

In order to determine the importance of analyzing configuration file accesses, we computed the number of external configuration files present in each environment. Table IV shows these results. While these numbers do not directly prove that addresses are being loaded from these configuration files, the sheer number of such files suggests that ignoring them would be a problem.

	A	B	C
Properties	439	99	26,732
XML	0	13	29

TABLE IV
Numbers of property and XML files used by the applications

D. Relinking

When testing the relinking aspect of our tool, we did not use applications from the three environments described above. As we cannot personally run the applications from those three environments, we instead focused our relinking evaluation on a collection of test programs, such as a running web server. Through our testing, we were able to successfully relink addresses and did not discover any situations where our bytecode modifications unexpectedly altered the semantics of the program.

E. Performance

Table V shows the performance of our tool on several applications of different sizes. We ran our tool on a single stop method invoked only once in the code, using a 2 GHz Linux machine with 4 GB of memory. For each application, we list the number of classes, the number of callgraph *nodes*, the number of production *rules* instantiated by the string analysis, and the time required to perform the analysis. About two thirds of the execution time is spent on building the callgraph

and doing other once-per-application operations. Automated relinking adds a negligable amount of time to the overall total.

In cases of a large enterprise applications, the string analysis becomes more complex, which leads to relatively long execution times (typically the total per-application processing takes 10–100 times longer than the one stop method times listed in Table V). Fortunately, we perform this analysis off-line after we have fetched the application code and related configuration files. Therefore, the time required to perform the analysis is tolerable. In addition, the off-line analysis allows us to use powerful servers. For our large-scale experiments with all the applications, we used a set of IBM System p5 575 16-core systems that are a lot more powerful than the machine used for Table V.

Application	Classes	Nodes	Rules	Time (sec)
App1	71	822	11130	33
App2	835	7828	545792	150
App3	1585	11648	406645	173
App4	3141	20510	674450	319

TABLE V
PERFORMANCE ON APPLICATIONS OF VARIOUS SIZES

VII. CONCLUSIONS

In this paper, we have presented the first method and tool for statically discovering and remediating code-embedded dependencies. Our tool identifies references to external resources such as databases, messaging queues, and files. It can also alter or externalize the detected addresses in order to enable migration.

We implemented our tool for Java and Java EE applications and analyzed three enterprise environments comprising 1097 unique applications. The resulting statistics show that our analysis is indeed needed to capture all the dependencies, despite Java EE standards that state dependencies should be defined in special resource files and not in the code. The percentage of applications with code-embedded dependencies, even if one only counts databases and messaging queues, ranges from 31 to 94% depending on the environment. This indicates that for a large percentage of applications, existing static discovery tools, including our own Galapagos tool, will miss dependencies. We also discovered that less than 10% of the dependencies are specified as simple string constants, making string analysis aspect of our tool necessary. Furthermore, we saw that components of these strings often come from other resources in the environment of the program.

Acknowledgments: We would like to thank Norbert G. Vogl, Daniel A. Prener, Murthy V. Devarakonda, HariGovind V. Ramasamy, Igor Peshansky, and Julian Dolby for their invaluable contributions to this paper.

REFERENCES

[1] HP discovery and dependency mapping (DDM) inventory software. www.hp.com/hpinfo/newsroom/press_kits/2007/softwareuniversebarcelona/ds_inventory.pdf.

[2] ISI-snapshot... agent-less accurate and rapid IT infrastructure inventory, configuration and utilization collection using a single tool. www.isiisi.com.

[3] M. Bowker, B. Garrett, and B. Laliberte. EMC smarts application discovery manager. Technical report, ESG Lab Validation Report, July 2007.

[4] A. Caracas, D. Dechouniotis, S. Fussenegger, D. Gantenbein, and A. Kind. Mining semantic relations using NetFlow. In *3rd IEEE/IFIP Int. Workshop on Business-Driven IT Management (BDIM)*, pages 110–111, 2008.

[5] A. S. Christensen, A. Moller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of 10th International Symposium on Static Analysis (SAS)*, 2003.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.

[7] T. Eilam, M. H. Kalantar, A. V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. Managing the configuration complexity of distributed applications in internet data centers. *IEEE Communications Magazine*, 44(3):166–177, 2006.

[8] C. Ensel. New Approach for Automated Generation of Service Dependency Models. In *2nd Latin American Network Operation and Management Symposium (LANOMS'01)*, 2001.

[9] Fyodor. *NMAP(1)*, 2003. www.insecure.org/nmap/data/nmap_manpage.html.

[10] D. Gantenbein and L. Deri. Categorizing computing assets according to communication patterns. In *Tutorial on Asset Inventory and Monitoring in a Networked World, Conf. on Networking*, pages 83–100, 2002.

[11] E. Geay, M. Pistoia, T. Tateishi, B. G. Ryder, and J. Dolby. Modular string-sensitive permission analysis with demand-driven precision. In *Proceeeding of 31st International Conference on Software Engineering (ICSE)*, 2009.

[12] HP discovery and dependency mapping software. https://h10078.www1.hp.com/cda/hpdc/navigation.do?action=downloadPDF&caid=9607&cp=54_4000_100&zn=bto&filename=4AA1-4991ENW.pdf.

[13] N. Joukov, B. Pfitzmann, H. V. Ramasamy, and M. V. Devarakonda. Application-storage discovery. In *3rd Annual Haifa Experimental Systems Conference (SYSTOR'10)*, Haifa, Israel, May 2010. ACM.

[14] Java string analyzer (JSA). www.brics.dk/JSA.

[15] A. Kind, D. Gantenbein, and H. Etoh. Relationship discovery with NetFlow to enable business-driven IT management. In *1st IEEE/IFIP Int. Workshop on Business-Driven IT Management (BDIM)*, pages 63–70, 2006.

[16] A. Kind, P. Hurley, and J. Massar. A light-weight and scalable network profiling system. *ERCIM News*, January 2005.

[17] Q. Ma, Y. Li, K. Sun, and L. Liu. Model-based dependency management for migrating service hosting environment. In *Proc. IEEE Intern. Conf. on Services Computing (SCC 2007)*, pages 356–363, 2007.

[18] K. Magoutis, M. Devarakonda, N. Joukov, and N. Vogl. Galapagos: Model-driven Discovery of End-to-End Application-Storage Relationships in Distributed Systems. *IBM J. Research and Development*, 52:367–378, 2008.

[19] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of 14th International World Wide Web Conference (WWW)*, 2005.

[20] M. Sethi, K. Kannan, N. Sachindran, and M. Gupta. Rapid deployment of SOA solutions via automated image replication and reconfiguration. In *Proc. IEEE Intern. Conf. on Services Computing (SCC 2008)*, volume 1, pages 155–162, 2008.

[21] Tivoli Application Dependency Discovery Manager. www.ibm.com/software/tivoli/products/taddm.

[22] Watson libraries for analysis (WALA). wala.sourceforge.net.

[23] X. Zheng, M. Zhan, Z. M. Mao, and P. Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proc. 8th Symp. on Operating Systems Design and Implementation (OSDI 2008)*, pages 117–130, San Diego, CA, December 2008.