PAPER 6

Abstract—**Software developers often need to port applications written for a source platform to a target platform. In doing so, a key task is to replace an application's use of methods from the source platform API with corresponding methods from the target platform API. However, this task is challenging because developers must manually identify mappings between methods in the source and target APIs, *e.g.,* using API documentation.**

We develop a novel approach to the problem of inferring mappings between the APIs of a source and target platform. Our approach is tailored to the case where the source and target platform each have independently-developed applications that implement similar functionality. We observe that in building these applications, developers exercised knowledge of the corresponding APIs. We develop a technique to systematically harvest this knowledge and infer likely mappings between the APIs of the source and target platform. The output of our approach is a ranked list of target API methods or method sequences that likely map to each source API method or method sequence. We have implemented this approach in a prototype tool called Rosetta, and have applied it to infer likely mappings between the Java2 Platform Mobile Edition and Android graphics APIs.

I. Introduction

Software developers often wish to make their applications available on a wide variety of computing platforms. The developer of a gaming app, for instance, may wish to make his app available on smart phones and tablets manufactured by various vendors, on desktops, and via the cloud. The key hurdle that he faces in doing so is to *port* his app to these software and hardware platforms.

Why is porting software a difficult problem? Consider an example: suppose that we wish to port a Java2 Platform Mobile Edition (JavaME)-based game to an Android-powered device. Among other tasks, we must modify the game to use Android's API [10] (the *target* platform) instead of JavaME's API [21] (the *source* platform). Unfortunately, the process of identifying the API methods in the target platform that implement the same functionality corresponding to that of a source platform API method is cumbersome. We must manually examine the SDKs of the source and target APIs to determine the right method (or sequence of methods) to use. To add complexity, there could be multiple ways in which a source API method can be implemented using the target's API methods. For example, the `fillRect()` method in JavaME's graphics API, which fills a specified rectangle with color, can be implemented using either one of these two sequences of methods in Android's graphics API: `setStyle();drawRect()` or as `moveTo();lineTo();lineTo();lineTo();lineTo();drawPath()` (we have omitted class names and the parameters to these method calls).

One way to address this problem is to populate a database of *mappings* between the APIs of the source and target platforms. In this database, each source API method (or method sequence) is mapped to a target API method (or method sequence) that implements its functionality. The database could contain multiple mappings (possibly ranked) for each source API method in cases where its functionality can be implemented in different ways by the target API. The mapping database significantly eases our task. We need only consider the mappings in this database to find suitable target API methods to replace a source API method, instead of painstakingly poring over the SDKs and their documentation. Such mapping databases do exist, but only for a few source/target API pairs (*e.g.,* Android, iOS and Symbian Qt to Windows 7 [30] and iOS to Qt [25]), and they are populated by domain experts well-versed in the source and target APIs.

Our contribution. We present an approach to automate the creation of mapping databases for any given source/target API pair. To bootstrap our approach, we rely on the availability of a few *similar application pairs* on the source and target platform. A source platform application $\mathcal{S}$ and a target platform application $\mathcal{T}$, possibly developed independently by different sets of programmers, constitute a similar application pair if they implement similar high-level functionality. For example, both $\mathcal{S}$ and $\mathcal{T}$ could implement the TicTacToe game on JavaME and Android, respectively. This situation is not uncommon in modern app markets, where independently-developed versions of popular apps are available in markets hosted by different vendors. Our approach builds upon the observation that *in implementing $\mathcal{S}$ and $\mathcal{T}$, their developers exercised knowledge about the APIs of the corresponding platforms*. We provide a systematic way to harvest this knowledge into a mapping database, which can then benefit other developers porting applications from the source to the target platform. Our approach works by recording traces of $\mathcal{S}$ and $\mathcal{T}$ executing similar tasks, structurally analyzing these traces, and extracting likely mappings using probabilistic inference. Each mapping output by our approach is associated with a probability, which indicates the likelihood of the mapping being true. The intuition is that the more evidence we see of a mapping, *e.g.,* the same pair of API methods being used across many traces to implement similar functionality, the higher the likelihood of the mapping. The output of our approach is a ranked list of mappings inferred for each source API method.

We demonstrate our approach by building a prototype tool called ***Rosetta*** (in reference to the legendary Rosetta Stone) to infer likely mappings between the JavaME and Android graphics APIs. We chose JavaME and Android because both platforms use the same language for application development. However, this requirement is not germane to our approach,

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

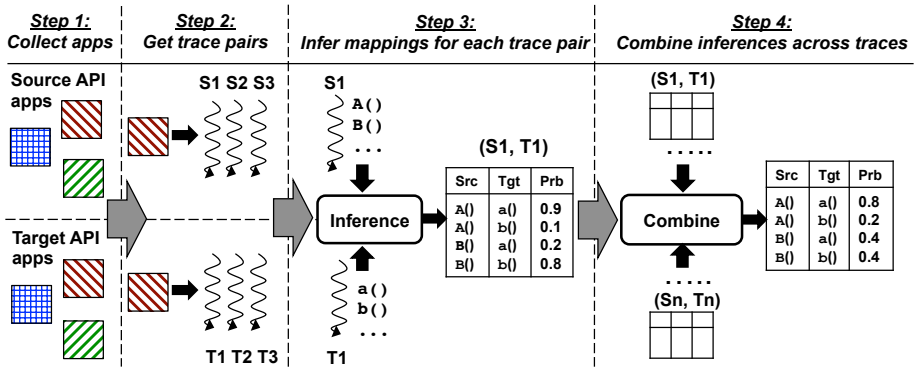

Fig. 1. Workflow of our approach to inferring likely API mappings.

JavaME Trace	Android Trace
Graphics.setColor()	Paint.setStyle()
Graphics.fillRect()	Color.parseColor()
Graphics.setColor()	Paint.setStyle()
Graphics.fillRect()	Color.parseColor()
Graphics.fillRect()	Canvas.drawLine()
Graphics.fillRect()	Canvas.drawLine()
	Canvas.drawLine()
	Canvas.drawLine()
	Canvas.drawLine()

Note: Each trace snippet shows the method invoked, and the class in which the method is implemented. JavaME and Android classes are prefixed with `javax/microedition/lcdui` and `android/graphics`, respectively. For brevity, we only refer to method names and not their classes.

Fig. 2. Snippets from traces of similar executions of JavaME and Android-based TicTacToe games.

and it may be possible to adapt Rosetta to work with source and target APIs that use different languages for application development. We evaluated Rosetta with a set of twenty-one independently-developed JavaME/Android application pairs. Rosetta was able to find at least one valid mapping within the top ten ranked results for 70% of the JavaME API methods observed in our traces. Further, for 40% of JavaME API methods, the top-ranked result was a valid mapping.

II. Approach Overview

We present the workflow of our approach (Figure 1), tailored to JavaME and Android as the source and target platforms, respectively. We only provide informal intuitions here, and defer the details to §IV. The workflow has four steps.

STEP 1: Collection of application pairs. The first step is to gather a database of applications in both the source and target platform. For each source application in the database, we require a target application that implements the same high-level functionality. For example, if we have a TicTacToe game for JavaME, we should locate a TicTacToe game for Android that is as functionally and visually (GUI-wise) similar to the JavaME game as possible.

Given the popularity of modern mobile platforms, and the desire of end-users to use similar applications across platforms, such application pairs are relatively easy to come by. Of course, given that the games were independently developed for these two platforms, there may be minor differences in functionality. For example, the Android game may offer menu options that are different from those of the JavaME game. However, as we discuss in Step 2, we can restrict ourselves to inducing functionally similar execution paths in these applications. Any functional differences that still make their way into these execution paths will manifest as inaccuracies during probabilistic inference in Step 3. However, the effect of these inaccuracies can be mitigated by combining inferences across multiple applications pairs and their executions in Step 4.

STEP 2: Execution and collection of trace pairs. In this step, we take each application pair, and execute them in similar ways, *i.e.,* we provide inputs to exercise similar functionality in these applications. As we do so, we also log a trace of API calls invoked by the applications. This gives a *trace pair*, consisting of one trace each for the source and target applications. Figure 2 presents a snippet from a trace pair that we gathered for TicTacToe games on the JavaME and Android platforms. They were collected by starting the game, waiting for the screen to display a grid of squares, and exiting.

Since the traces in each pair were obtained by exercising similar functionality in the source and target applications, these traces must contain some API calls that can be mapped to each other. This is the key intuition underlying our approach. Of course, an application can be invoked in many ways, and in this step, we collect several trace pairs for each application pair. The output of this step is a database of functionally equivalent trace pairs (Trace_S, $\mathsf{Trace}_\mathcal{T}$) across all of the application pairs collected in Step 1. Steps 1 and 2 involve manual effort, while steps 3 and 4 are automated.

STEP 3: Trace analysis and inference. In this step we analyze each trace pair to infer likely API mappings implied by the pair. Our inference algorithm relies on various attributes that are determined by the structure of the traces. We cast the attributes as inputs to a probabilistic inference algorithm (§IV). The output of this step is a probability for each pair of source and target API calls `A()`/`a()` that indicates the likelihood of `A()` mapping to `a()` (denoted here as `A()`→`a()`). Our algorithm can also infer mappings between method sequences (*e.g.,* `A()`→`a();b()` or `A();B()`→`a()`).

We now discuss the attributes used by our approach using our running example. Our aim is to find likely mappings for the JavaME calls `setColor()` and `fillRect()`. For simplicity, we restrict our discussion to the snippets of the traces shown in Figure 2. In reality, our analysis considers the entire trace.

(**1**) *Call frequency.* If `A()` in the source API maps to the `a()` in the target API, then the frequency with which these method calls occur in functionally-similar trace pairs must also match. The trace pairs may differ in the absolute number of method calls that they contain, so we focus on the *relative count* of each method call, which is the raw count of the number of times that a method is called, normalized by trace length. The table below shows the raw and relative counts of various method calls based upon the snippets in Figure 2. Using this attribute, the following API mappings appear likely, `setColor()`→`parseColor()`, `setColor()`→`setStyle()`, `fillRect()`→`drawLine()`, while the others appear unlikely. In fact, our approach works on method sequences as

well using the same reasoning as above, and infers that `setColor()→setStyle();parseColor()` is a mapping.

API Call	Raw Count	Relative Count
`setColor()`	2	0.33
`fillRect()`	4	0.67
`setStyle()`	2	0.22
`parseColor()`	2	0.22
`drawLine()`	5	0.56

(**2**) *Call position.* The location of method calls in a trace pair provides further information to determine likely mappings. Since the traces exercise the same application functionality, method calls that map to each other should appear at "similar" positions in the trace pair. The table below shows the position of each of the method calls in our running example, roughly categorized as belonging to the first half or the second half of the corresponding traces. We can therefore reinforce the beliefs about API mappings inferred using call frequencies.

API Call	First Half	Second Half
`setColor()`	✓	×
`fillRect()`	×	✓
`setStyle()`	✓	×
`parseColor()`	✓	×
`drawLine()`	×	✓

(**3**) *Call context.* The context in which a method call `A()` appears is defined to be the set of API methods that appears in its vicinity in the execution trace (e.g., within a pre-set threshold distance, preceding or following `A()` in the trace). For example, both `setColor()` calls appear in the preceding context of `fillRect()` calls in the JavaME trace (using a threshold distance of 2 calls). Likewise, `parseColor()` and `setStyle()` appear in the preceding context of `drawLine()` in the Android trace. In fact, the sequence `setStyle(); parseColor()` appears in the preceding context of the first `drawLine()`. From this, we can infer that *if* `fillRect()→drawLine()` holds, *then* the mapping `setColor()→setStyle();parseColor()` is likely to hold.

(**4**) *Method names.* While trace structure, as captured by the attributes above, contributes to inference, method (and class) names also contain useful information about likely mappings, and we leverage this attribute as well. We use Levenshtein edit distance to compute the similarity of method names. Using this attribute, for instance, we can lend credence to the belief that `setColor()` maps to `parseColor()`.

Our approach combines these attributes to output likely mappings, each associated with a probability. We can rank the results using the corresponding probabilities, and use thresholds to limit the number of results that are output.

STEP 4: Combining inferences across traces. The inference algorithm of Step 3 works by analyzing a single trace pair. In the final step, we combine inferences across multiple traces. During combination, we weight inferences obtained from the analysis of individual trace pairs using the confidence of each inference. Intuitively, more data we have about an inferred mapping from a trace pair, the stronger our confidence in that inference. Thus, our confidence in mapping `A()→a()` obtained from a trace pair where these calls occur several times is stronger than the same mapping obtained from a trace pair where these calls occur infrequently. We use this heuristic to combine inferences across trace pairs.

III. Framework to Represent and Infer Mappings

We now describe the framework used to represent and infer likely mappings between APIs. We restrict ourselves to identifying likely mappings between methods of the source and target APIs. We do not currently consider the problem of determining mappings between arguments to these methods; that is a related problem that can be addressed using parameter recommendation systems (*e.g.,* [33]). Let $\mathcal{S}$ = {`A`, `B`, `C`, ...} and $\mathcal{T}$ = {`a`, `b`, `c`, ...} denote the sets of methods in the source and target APIs, respectively. Our goal is to determine which methods in $\mathcal{T}$ implement the same functionality on the target platform as each method in $\mathcal{S}$ on the source platform.

To denote mappings, our framework considers a set of Boolean variables X^{s}_{t}, where $s \in \mathcal{S}$ and $t \in \mathcal{T}$. The Boolean variable X^{A}_{a} is set to TRUE if the method call `A()` maps to the method call `a()`, and FALSE otherwise.

This framework extends naturally to the case where a method (or sequence of methods) in $\mathcal{S}$ can be implemented with a method sequence in $\mathcal{T}$. For example, suppose that the method `A()` is implemented using the sequence `a();b()` in the target. We can denote this by assigning TRUE to the Boolean variable X^{A}_{ab}. Likewise, we can also denote cases where the functionality of a sequence of methods `A();B()` in the source platform is implemented using a method `a()` on the target platform by setting the Boolean variable X^{AB}_{a} to TRUE. Although our framework theoretically supports inference of mappings between arbitrary length method sequences (*e.g.,* $X^{ABC...}_{abc...}$ = TRUE), for performance reasons we configured Rosetta to only infer mappings between method sequences of length two.

We approach the problem of inferring API mappings by studying the structure of execution traces of similar applications on the source and target platforms. We use the trace attributes informally discussed in §II to deduce API mappings. Our approach is inherently probabilistic. It cannot conclude whether a call `A()` definitively maps to a call `a()`; rather it determines the likelihood of the mapping. Thus, it infers $Pr[X^{A}_{a} = \text{TRUE}]$ for each Boolean variable X^{A}_{a}. As it observes more evidence of the mapping X^{A}_{a} being true, it assigns a higher value to this probability. Therefore, our framework treats each Boolean variable X^{A}_{a} as a random variable, and our probabilistic inference algorithm determines the probability distributions of these random variables.

Each application trace has several method calls, and the inference algorithm must leverage the structure of these traces to determine likely mappings. The inference algorithm draws conclusions not just about *individual* random variables, but also about how they are *related*. For example, consider a trace snippet $\text{Trace}_{\mathcal{S}}$ = (...; `A();A();B();B();` ...) of an application from the source API, and a snippet $\text{Trace}_{\mathcal{T}}$ = (...; `a();a();b();b();` ...) from the corresponding execution of a similar application on the target. Suppose also that these are the only occurrences of `A()`, `B()`, `a()` and `b()` in $\text{Trace}_{\mathcal{S}}$ and $\text{Trace}_{\mathcal{T}}$, respectively, and that these execution traces have approximately the same number of method calls in total. By just observing these snippets, and relying on the frequency of

method calls, each of the following cases is a possibility: X_a^A = TRUE, X_b^A = TRUE, X_a^B = TRUE, X_b^B = TRUE. However, if X_b^A = TRUE, then because of the relative placement of method calls in these traces (*i.e.,* call context), it is unlikely that X_a^B = TRUE. Now suppose that by observing more execution traces, we are able to obtain more evidence that X_b^A = TRUE, then we can leverage the structure of this pair of traces to deduce that X_a^B is unlikely to be a mapping. Intuitively, the structure of the trace allows us to propagate the belief that if X_b^A is TRUE, then X_a^B is FALSE. Our probabilistic inference algorithm uses factor graphs [13], [32] for belief propagation.

Factor Graphs. Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean random variables and $\mathcal{J}(x_1, x_2, \ldots, x_n)$ be a joint probability distribution over these random variables. $\mathcal{J}$ assigns a probability to each assignment of truth values to the random variables in $\mathcal{X}$. Given such a joint probability distribution, it is natural to ask what the values of the *marginal probabilities* $\mathcal{M}_k(x_k)$ of each random variable x_k in $\mathcal{X}$ are. Marginal probabilities are defined as $\mathcal{M}_k(x_k) = \sum_{i \neq k, i \in [1..n], x_i \in \{\text{TRUE,FALSE}\}} \mathcal{J}(x_1, x_2, \ldots, x_n)$. That is, they calculate the probability distribution of x_k alone by summing up $\mathcal{J}(\ldots)$ over all possible assignments to the other random variables. $\mathcal{M}_k(x_k)$ allows us to compute $\Pr[x_k =$ TRUE$]$. Factor graphs allow efficient computation of marginal probabilities from joint probability distributions when the joint distribution can be expressed as a product of *factors*, *i.e.,* $\mathcal{J}(x_1, x_2, \ldots, x_n) = \prod_i f_i(X_i)$. Each f_i is a factor, and is a function of some subset of variables $X_i \subseteq \mathcal{X}$.

For example, consider a probability distribution $\mathcal{J}(x, y, z)$. Let this distribution depend on three factors $\mathsf{f}(x, y)$, $\mathsf{g}(y, z)$ and $\mathsf{h}(z)$, *i.e.,* $\mathcal{J}(x, y, z) = \mathsf{f}(x, y).\mathsf{g}(y, z).\mathsf{h}(z)$, defined as follows:[1]

$$\begin{aligned} \mathsf{f}(x, y) &= \quad 0.9 \text{ if } x \vee y = \text{FALSE}, 0.1 \text{ otherwise.} \\ \mathsf{g}(y, z) &= \quad 0.9 \text{ if } y \vee z = \text{TRUE}, 0.1 \text{ otherwise.} \\ \mathsf{h}(z) &= \quad 0.9 \text{ if } z = \text{TRUE}, 0.1 \text{ otherwise.} \end{aligned}$$

A factor graph denotes such joint probabilities pictorially as a bipartite graph with two kinds of nodes, *function* nodes and *variable* nodes. Each function node corresponds to a factor, while each variable node corresponds to a random variable. A function node has outgoing edges to each of the variable nodes over which it operates. Figure 3 depicts the factor graph of $\mathcal{J}$. The AI community has devised efficient solvers [13], [32] that operate over such graphical models to determine marginal probabilities of individual random variables. We do not discuss the details of these solvers, since we just use them in a black box fashion.

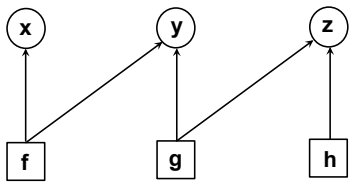

Fig. 3. Factor graph of $\mathcal{J}(x, y, z)$.

Factor graphs cleanly represent how the random variables are related to each other, and can influence the overall probability distribution. One of the key characteristics of factor graphs, which led us to use them in our work, is that they were designed for belief propagation, *i.e.,* in transmitting beliefs about the probability distribution of one random variable to determine the distribution of another.

To illustrate this, consider the probability distribution $\mathcal{J}(x, y, z)$, discussed above. $\mathcal{J}$ can be interpreted as denoting the probabilities of the outcomes of an underlying Boolean formula for various assignments to x, y, and z. Under this interpretation, we could say that the Boolean formula evaluates to TRUE for those assignments to x, y and z for which the value of $\mathcal{J}(x, y, z)$ is above a certain threshold, *e.g.,* if the threshold is 0.6, and $\mathcal{J}$(TRUE, TRUE, FALSE) is 0.7, we say that the formula is TRUE under this assignment. Now suppose that y is likely to be FALSE, *i.e.,* $\Pr[y =$ FALSE$]$ is above a threshold. We are asked to find under what conditions on x and z the Boolean formula still evaluates to TRUE, *i.e.,* $\mathcal{J}(x, y, z)$ is above the threshold. From the definitions of the factors f, g, and h, we know that $\mathcal{J}$ obtains values that are likely to exceed the threshold if $x \vee y$ = FALSE, $y \vee z$ = TRUE and z is TRUE. Given that y is likely to be false FALSE, these factors lead us to deduce that x is also likely to be FALSE (giving $\mathsf{f}(x, y)$ a high value) and z is likely to be TRUE (giving $\mathsf{g}(y, z)$ and $h(z)$ high values), thereby pushing the value of $\mathcal{J}$ above the threshold. Intuitively, the factor graph allows us to propagate the belief about the value of y into beliefs about the value of x and z. §IV shows how we cast the problem of inferring likely API mappings using factor graphs, thereby allowing us to transmit beliefs about one mapping into beliefs about others.

[1]Because $\mathcal{J}$ is a *probability* distribution, the product $\mathcal{J}(x, y, z)$ is in fact $Z.f(x, y).g(y, z).h(z)$, where Z is a normalization constant introduced to ensure that the probabilities sum up to 1. Henceforth, Z is implied, and will not be shown explicitly.

IV. Design and Implementation of Rosetta

We now describe in detail the Rosetta prototype, which currently infers mappings between the JavaME and Android graphics APIs. Specifically, we focus on the machinery that enables Steps 2-4 discussed in §II.

A. Infrastructure for Trace Collection

To record execution traces, we instrumented JavaME programs via bytecode rewriting. We used the ASM toolkit [4] to insert logging functionality that records the name of each method call (and class name), prior to invocation. During runtime, this results in a trace of all methods invoked. We then filter out just those methods that derive from the class `javax/microedition/lcdui`—the JavaME graphics API. In all, this API has 281 distinct methods. Rosetta infers mappings for those methods that appear in application traces.

We did not employ bytecode rewriting for Android applications because of the lack of publicly-available tools to rewrite Android's `dex` bytecode. Instead, we leveraged the Dalvik virtual machine (v2.1r1) to record the names of all methods invoked by an application. We record all method and class names, and then filter methods in the following classes (prefix: `android/`) `graphics`, `text`, `view`, `widget`, `content/DialogInterface`, `app/Dialog`, `app/AlertDialog`, `app/ActionBar`. This API has 3,837 distinct methods. The difference in the sizes of these APIs illustrates in part the difficulties that a programmer manually porting an application would face.

With this infrastructure, a Rosetta user can collect traces for a pair of similar applications on both platforms. As discussed in §II, the user must exercise similar functionality in both applications, thereby collecting a pair of traces that record this functionality. This process can be repeated multiple times for the same application, exercising different functionalities, thereby resulting in a database of trace pairs. Although it is hard to provide concrete guidelines to "exercise similar functionality," we found that it was relatively easy to do so for gaming applications. Given similar games, they will likely have the same logic and similar GUIs on both applications. We simply performed the same moves on games in both platforms, avoiding situations that involve randomness where possible (*e.g.,* choosing the two-user mode to avoid the computer picking moves at random). However, randomness is not always avoidable, *e.g.,* some games only support user versus computer modes, and this randomness may manifest in the corresponding portions of the traces as well. Despite this, Rosetta infers high-quality mappings because its inference algorithm prioritizes method mappings that persist both across the entirety of each trace pair, and across multiple trace pairs.

B. Trace Analysis and Inference

In this step, Rosetta analyzes each trace pair collected in the previous step and draws inferences about likely API mappings implied by that trace pair. Recall from §III that we use a Boolean random variable X^{A}_{a} to denote a mapping between `A()` and `a()` (likewise X^{A}_{ab} *etc.*, for method sequences). In this step, Rosetta uses factor graphs to compute $\Pr[X^{M}_{m} = \text{TRUE}]$, for each such random variable, where M and m denote individual methods or method sequences from the source and target APIs, respectively. The value of this probability determines the likelihood that the corresponding mapping holds.

The intuition behind Rosetta's use of factor graphs is as follows. The set of random variables X^{M}_{m} implicitly defines a joint probability distribution $\mathcal{J}$ over these random variables: $\mathcal{J}(X^{A}_{a}, X^{A}_{b}, \ldots, X^{B}_{a}, X^{B}_{b}, \ldots X^{A}_{ab}, X^{B}_{ab}, \ldots)$. As in §III, we can assign $\mathcal{J}$ a Boolean interpretation. That is, we treat $\mathcal{J}$ as a probability distribution that estimates the likelihood of an underlying Boolean formula being TRUE, under various truth assignments to the random variables X^{A}_{a}, X^{A}_{b}, *etc*. From this joint distribution, our goal is to find the probability distributions of the individual random variables. Under this Boolean interpretation, if $\Pr[X^{A}_{a} = \text{TRUE}]$ acquires a high value, it means that the Boolean formula underlying $\mathcal{J}$ is likely to be TRUE if X^{A}_{a} is TRUE, thereby leading us to conclude that `A()` is likely to map to `a()`. Likewise, if $\Pr[X^{A}_{a} = \text{TRUE}]$ acquires a low value, `A()` is unlikely to map to `a()`.

The main challenge in directly realizing this intuition within an inference tool is that the Boolean formula underlying $\mathcal{J}$ is unknown (for if it *were* known, then any satisfying assignment to it would directly yield an API mapping!). As a result, the joint probability distribution $\mathcal{J}$ cannot be explicitly computed. However, the attributes described in §II determine the conditions that are likely to influence $\mathcal{J}$. Thus, we formalize each of these attributes as factors, and estimate the joint probability distribution as the product of these factors. Of course, these factors are not comprehensive, *i.e.,* there may be other factors that influence the value of $\mathcal{J}$. Rosetta can naturally accommodate any new factors; they are simply treated as additional factors in the product.

Rosetta's trace analysis computes four families of factors, one each for the four attributes. It combines them and uses them for probabilistic inference of likely mappings as shown in Algorithm 1.

(1) Call frequency (f_{freq}). The intuition underlying this factor is that if M maps to m (where M and m are individual methods or method sequences), then the frequency with which they appear in functionally similar traces must match. Thus, we

Algorithm 1: Inferring likely mappings.

Input : A trace pair ($\text{Trace}_{\mathcal{S}}$, $\text{Trace}_{\mathcal{T}}$).
Output : $\Pr[X^{M}_{m} = \text{TRUE}]$ for each X^{M}_{m}, where M and m are methods and sequences in $\text{Trace}_{\mathcal{S}}$, $\text{Trace}_{\mathcal{T}}$, respectively.

$\text{MethSeq}_{\mathcal{S}}$ = set of methods and method sequences that appear in $\text{Trace}_{\mathcal{S}}$ (sequences up to length 2 in our prototype)
$\text{MethSeq}_{\mathcal{T}}$ = set of methods and method sequences that appear in $\text{Trace}_{\mathcal{T}}$
foreach ($M \in \text{MethSeq}_{\mathcal{S}}$ and m in $\text{MethSeq}_{\mathcal{T}}$) **do**
  $f_{\text{freq}}(X^{M}_{m}) = \text{simCount}(M, m, \text{Trace}_{\mathcal{S}}, \text{Trace}_{\mathcal{T}})$
  $f_{\text{pos}}(X^{M}_{m}) = \text{simPos}(M, m, \text{Trace}_{\mathcal{S}}, \text{Trace}_{\mathcal{T}})$
  $f_{\text{name}}(X^{M}_{m}) = \text{simName}(M, m)$
foreach ($M, N \in \text{MethSeq}_{\mathcal{S}}$ and m, n in $\text{MethSeq}_{\mathcal{T}}$) **do**
  $f_{\text{ctxt}}(X^{M}_{m}, X^{N}_{n}) = \text{simCtxt}(M, N, m, n, \text{Trace}_{\mathcal{S}}, \text{Trace}_{\mathcal{T}})$
  $f_{\text{ctxt}}(X^{M}_{n}, X^{N}_{m}) = \text{simCtxt}(M, N, n, m, \text{Trace}_{\mathcal{S}}, \text{Trace}_{\mathcal{T}})$
Let the set $\mathcal{F}$ denote the factors gathered above.
Let $\mathcal{J}(X^{A}_{a}, X^{A}_{b}, \ldots, X^{B}_{a}, X^{B}_{b}, \ldots X^{A}_{ab}, X^{B}_{ab}, \ldots) = \prod_{f \in \mathcal{F}} f$.
Use factor graphs to obtain marginal probabilities for each X^{M}_{m} from $\mathcal{J}$.

Algorithm 2: Subroutines invoked by Algorithm 1.

$\text{simCount}(M, m, \text{Trace}_{\mathcal{S}}, \text{Trace}_{\mathcal{T}})$
begin
  $\text{relCount}(M) = \frac{\#\text{Occurences of } M \text{ in } \text{Trace}_{\mathcal{S}}}{\text{Length}(\text{Trace}_{\mathcal{S}})}$
  $\text{relCount}(m) = \frac{\#\text{Occurences of } m \text{ in } \text{Trace}_{\mathcal{T}}}{\text{Length}(\text{Trace}_{\mathcal{T}})}$
  Return $\min[\frac{\text{relCount}(M)}{\text{relCount}(m)}, \frac{\text{relCount}(m)}{\text{relCount}(M)}]$
end
$\text{simPos}(M, m, \text{Trace}_{\mathcal{S}}, \text{Trace}_{\mathcal{T}})$
begin
  $\text{relPos}(M)[.]$ = relative positions of M in $\text{Trace}_{\mathcal{S}}$
  $\text{relPos}(m)[.]$ = relative positions of m in $\text{Trace}_{\mathcal{T}}$
  $\text{avgRP}(M)$ = average of values in $\text{relPos}(M)[.]$
  $\text{avgRP}(m)$ = average of values in $\text{relPos}(m)[.]$
  if (the values in the array $\text{relPos}(M)[.]$ are within a threshold (1% of the trace length) of $\text{avgRP}(M)$ and likewise for $\text{relPos}(m)[.]$ and $\text{avgRP}(m)$) **then** Return $\min[\frac{\text{avgRP}(M)}{\text{avgRP}(m)}, \frac{\text{avgRP}(m)}{\text{avgRP}(M)}]$
  else Return UNDECIDED (the value of UNDECIDED is 0.5)
end
$\text{simName}(M, m)$
begin
  if (M and m are individual methods) **then** Return LEVENSHTEIN.RATIO(M, m) **else** Return UNDECIDED
end
$\text{simCtxt}(M, N, m, n, \text{Trace}_{\mathcal{S}}, \text{Trace}_{\mathcal{T}})$
begin
  $\text{relCount}(M, N) = \frac{\#\text{Occurences of } M \text{ in preceding context of } N \text{ in } \text{Trace}_{\mathcal{S}}}{\text{Length}(\text{Trace}_{\mathcal{S}})}$
  $\text{relCount}(m, n) = \frac{\#\text{Occurences of } m \text{ in preceding context of } n \text{ in } \text{Trace}_{\mathcal{T}}}{\text{Length}(\text{Trace}_{\mathcal{T}})}$
  $\text{relCount}(N, M) = \frac{\#\text{Occurences of } N \text{ in preceding context of } M \text{ in } \text{Trace}_{\mathcal{S}}}{\text{Length}(\text{Trace}_{\mathcal{S}})}$
  $\text{relCount}(n, m) = \frac{\#\text{Occurences of } n \text{ in preceding context of } m \text{ in } \text{Trace}_{\mathcal{T}}}{\text{Length}(\text{Trace}_{\mathcal{T}})}$
  if $\text{ApproxMatch}(\text{relCount}(M, N), \text{relCount}(m, n))$ and $\text{ApproxMatch}(\text{relCount}(N, M), \text{relCount}(n, m))$ **then** Return HIGH (we configued HIGH to be 0.7), **else** Return (1 - HIGH).
end

compute the relative count of M and m as the number of times that they appear, normalized by the corresponding trace length. We then use the ratio of relative counts of M and m to compute $\mathsf{f}_{\mathsf{freq}}$. This is described in the subroutine simCount, shown in Algorithm 2.

(2) Call position ($\mathsf{f}_{\mathsf{pos}}$). We observed in our experiments that certain API methods and sequences appear only at specific positions in the trace. For example, API methods that initialize the screen or game state appear only at the beginning of the trace. To identify such methods and sequences, we use a similarity metric that determines the *relative position* of the appearance of the method call or call sequence in the trace, *i.e.,* its offset from the beginning of the trace, normalized by the trace length. Of course, there may be multiple appearances of the method call or sequence in the trace, so we average their relative positions.

In this factor, we restrict ourselves only to calls and sequences that are localized in a certain portion of the trace, *i.e.,* if the relative positions are not within a threshold (1% of the trace length) of the average, this factor does not contribute positively or negatively to the likelihood of the mapping (UNDECIDED is a probability of 0.5). This is described in the subroutine simPos in Algorithm 2.

(3) Method names ($\mathsf{f}_{\mathsf{name}}$). We use the names of methods in the source and target APIs to determine likely mappings. Unlike the other factors, which are determined by trace structure (*i.e.,* program behavior), this is factor relies on a syntactic feature. The simName subroutine in Algorithm 2 uses a ratio based upon the Levenshtein edit distance, computed using a standard Python library [24]. This ratio ranges from 1 for identical strings, to 0 for strings that do not have a common substring. simName only returns a valid ratio for individual methods; for sequences, it returns UNDECIDED.

(4) Call context ($\mathsf{f}_{\mathsf{ctxt}}$). We define the context of a method call `A()` in a trace as the set of method calls that appear in the vicinity of `A()`. Likewise, the context of a sequence `A();B()` is the set of method calls that appear in the vicinity of this sequence (if it exists in the trace). Considering context allows us to propagate beliefs about likely mappings. Recall the example presented in §III, where considering the frequency of the method calls `A()`, `B()`, `a()`, and `b()` alone does not allow precise inference of mappings. In that example, the context of the calls allows us to infer that if $\mathsf{X}_{\mathtt{b}}^{\mathtt{A}}$ is TRUE, then $\mathsf{X}_{\mathtt{a}}^{\mathtt{B}}$ is unlikely to be TRUE. Of the four factors that we consider, $\mathsf{f}_{\mathsf{ctxt}}$ is the only one that relates pairs of random variables; the others assign probabilities to individual random variables.

We define the context of a method call or sequence M in the trace as the set of method calls and sequences that appear within a fixed distance k of M in the trace; in our prototype, k=4. When computing the context of M, we also consider whether the entities its context precede M or follow M. To compute context as a factor, we use the function simCtxt, which considers all pairs of methods and method sequences (M, N) that appear in the source trace, and all pairs (m, n) that appear in the target trace. We then count the number of times M appears in the preceding context of N in the source trace (*i.e.,* within k=4 calls preceding each occurence of N) and normalize this using the trace length (relCount(M,N)); likewise we compute relCount(N,M), and the corresponding metrics for the target trace. We then check whether relCount(M,N) "matches" relCount(m,n), and relCount(N,M) "matches" relCount(n,m) We do not require the relative counts to match exactly; rather their difference should be below a certain threshold (10% in our prototype); ApproxMatch encodes this matching function.

If both the counts match, then the factor $\mathsf{f}_{\mathsf{ctxt}}(\mathsf{X}_m^M, \mathsf{X}_n^N)$ positively influences the inference that if X_m^M is TRUE, then X_n^N is also TRUE, and vice-versa. The simCtxt function ensures this by returning a HIGH probability value. Likewise, if the counts do not match, $\mathsf{f}_{\mathsf{ctxt}}(\mathsf{X}_m^M, \mathsf{X}_n^N)$ would indicate that X_m^M and X_n^N are unlikely to be TRUE simultaneously. Note that this does *not* preclude X_m^M or X_n^N from being TRUE individually. Intuitively, the Boolean interpretation of $\mathsf{f}_{\mathsf{ctxt}}(\mathsf{X}_m^M, \mathsf{X}_n^N)$ is $(\mathsf{X}_m^M \wedge \mathsf{X}_n^N)$. In our prototype, we set the value of HIGH as 0.7. We conducted a sensitivity study by varying the value of HIGH between 0.6 and 0.8, and observed that it did not significantly change the set of likely mappings output by Rosetta.

To illustrate the context factor, consider again the example from §III. There, simCtxt(`A`, `B`, `a`, `b`) would be HIGH, while simCtxt(`A`, `B`, `b`, `a`) would be 1-HIGH (using exact matches for relCount instead of ApproxMatch, to ease illustration). Therefore, we can infer that $\mathsf{X}_{\mathtt{a}}^{\mathtt{A}}$ and $\mathsf{X}_{\mathtt{b}}^{\mathtt{B}}$ could both be TRUE, but that $\mathsf{X}_{\mathtt{b}}^{\mathtt{A}}$ and $\mathsf{X}_{\mathtt{a}}^{\mathtt{B}}$ are unlikely to be TRUE simultaneously.

We implemented Rosetta's trace analysis and factor generation algorithms in about 1300 lines of Python code. We used the implementation of factor graphs in the Bayes Net Toolbox (BNT) [19] for probabilistic inference. Rosetta generates one factor for each Boolean X_m^M for each of the three factor families $\mathsf{f}_{\mathsf{freq}}$, $\mathsf{f}_{\mathsf{pos}}$, $\mathsf{f}_{\mathsf{name}}$. Letting S and T denote the number of unique source and target API calls observed in the trace, there are $O(S^2T^2)$ such Boolean variables (because M and m include individual methods and method sequences of length two). Likewise, Rosetta generates two $\mathsf{f}_{\mathsf{ctxt}}$ factors for each pair (M, N) and (m, n), resulting in a total of $O(S^4T^4)$ factors. We restricted Rosetta to work with method sequences of lengths one and two because of the rapid growth in the number of factors. Future work could consider optimizations to prune the number of factors, thereby allowing inference of mappings for longer length method sequences.

C. Combining Inferences Across Traces

As discussed so far, we apply probabilistic inference to each trace pair, which results in different values of $\Pr[\mathsf{X}_m^M = \text{TRUE}]$ for each Boolean variable X_m^M. In this step, we combine these inferences across the entire database of trace pairs. One way to combine these probabilities is to simply average them. However, if we do so, we ignore the *confidence* that we have in our inferences from each trace pair. The more occurrences we see of a method call or sequence M in a source trace, the more confidence we have in the values of $\Pr[\mathsf{X}_*^M = \text{TRUE}]$. Therefore, we compute a weighted average of these probabilities, with the relative count of each source call as the weight.

Game (#Traces)	JavaME Traces		Android Traces		Factor graphs	
	AvgLen	MaxLen	AvgLen	MaxLen	MaxNodes	MaxEdges
Backgammon (1)	33,733	33,733	197,073	197,073	15,230	13,435
Blackjack (3)	690	731	272	369	11,408	10,256
Bubblebreaker (4)	858	2,033	2,366	7,377	2,844	2,358
Checkers (1)	111,790	111,790	1,014	1,014	4,923	4,191
Chess (4)	80,483	175,018	1,377	1,594	7,562	9,664
Four in a Row (3)	4,828	8,751	12,757	23,450	19,868	16,021
FreeCell (4)	9,159	15,271	330	746	128,128	96,096
Hangman (3)	3,715	4,282	3,477	3,491	10,319	11,263
Mahjongg V1 (5)	8,035	18,321	20,198	49,887	8,891	7,062
Mahjongg V2 (4)	93,739	150,206	17,241	22,025	4,703	4,806
Memory (3)	153,282	190,199	26,796	35,369	15,387	16,399
Minesweeper (4)	2,091	5,396	1,161	1,939	14,105	12,025
Roulette (5)	5,003	6,232	185	227	26,580	19,935
Rubics Cube (3)	16,521	19,343	159	255	30,660	22,995
Scrabble (4)	7,834	14,358	207	497	105,300	78,975
SimpleDice (5)	654	965	581	593	37,152	27,864
Snake (2)	33,399	63,104	11,681	20,372	1,528	1,356
Soltaire (3)	8,016	12,471	10,860	20,803	21,714	20,228
Sudoku (5)	3,897	9,968	17,347	42,567	16,306	24,317
Tetris (2)	486	916	6,116	11,991	13,105	14,520
TicTacToe (4)	154	475	183	418	3,840	4,690

Fig. 4. Statistics of traces and factor graphs for various JavaME and Android games. The actual runtime traces of the games are filtered to leave only JavaME and Android graphics API calls. We report the average and maximum lengths of these filtered traces. For each game, we also show the size of the largest factor graph across all its traces.

JavaME class	#Methods	#Top-10 (#TotValid)		#Top-1
`Alert`	4	3	(11)	3
`Canvas`	8	5	(18)	4
`Command`	2	2	(3)	0
`Display`	7	3	(5)	1
`Displayable`	6	4	(12)	3
`Font`	5	3	(9)	1
`Form`	4	2	(6)	1
`game.GameCanvas`	5	5	(13)	2
`game.Layer`	3	3	(7)	3
`game.Sprite`	4	4	(8)	1
`Graphics`	21	18	(61)	11
`Image`	4	4	(8)	2
`List`	1	0	(0)	0
`TextField`	6	0	(0)	0
Total	**80**	**56**	**(161)**	**32**

Fig. 5. Results of applying Rosetta to the traces obtained from the games shown in Figure 4. We have shown the number of unique JavaME methods (categorized by class) for which Rosetta inferred at least one valid mapping in the top ten (#Top-10), and the total number of valid mappings found in the top ten (#TotValid). Also shown is the number of JavaME calls for which Rosetta's top-ranked inference was a valid mapping (#Top-1). See also Figure 6 and Figure 7 for a more detailed rank distribution of mappings.

$$\Pr[X_m^M = \text{TRUE}]\Big|_{combined} = \frac{\sum_{Traces} \text{relCount}(M) \times \Pr[X_m^M = \text{TRUE}]}{\sum_{Traces} \text{relCount}(M)}$$

We chose this approach because of its modularity. As we collect more trace pairs and inferences from them, we can combine them with mappings inferred from other traces in a straightforward way using the weighted average approach. Alternatively, we could have chosen to concatenate individual traces (in the same order for both components of each trace pair) to produce a "super-trace," and perform probabilistic inference over this super-trace. However, if we do so, then we would have to reproduce the super-trace after each new trace pair that is collected, and execute the inference algorithm over the ever-growing super-trace. This approach is neither memory-efficient nor time-efficient. In contrast, our weighted average approach provides more modular support to add inferences from new trace pairs as they become available.

This weighted average is presented to the user as the output from Rosetta. We present the output of Rosetta to the user in terms of the inferred mappings for each source API call. For each source API method or method sequence, we present a list of mappings inferred for it, ranked in decreasing order of the likelihood of the mapping.

V. Evaluation

A. Methodology

To evaluate Rosetta, we collected a set of 21 JavaME applications for which we could find functionally-equivalent counterparts in the Android market. In particular, we chose board games, for two reasons. First, many popular board games are available for both the JavaME and Android platforms. Checking the functional equivalence of two games is as simple as playing the games and ensuring that the moves of the game are implemented in the same way on both versions. Second, the use of board games also eases the task of collecting trace pairs. Moves in board games are easy to remember and can be repeated on both game versions to produce functionally-equivalent traces on both platforms. Note that a pair of "functionally-equivalent" games on JavaME and Android could differ in the features that they implement. However, when tracing these games, we restrict ourselves to the subsets of features that are common to both games.

To collect traces, we ran JavaME games using the emulator distributed with the Sun Java Wireless Toolkit, version 2.5.1 [22]. For Android games, we used the emulator that is distributed with the Android 2.1 SDK. Figure 4 shows the games that we used, the number of trace pairs that we collected for each game, and the sizes of the traces for the JavaME and Android versions.

We ran Rosetta on these traces to obtain a set of likely mappings. This set is presented to the user as a ranked list of mappings inferred for each JavaME method (or method sequence). To evaluate the list associated with each JavaME method, we consulted the documentation of the JavaME and Android graphics APIs to determine which members of the list are valid mappings.

B. Quality of Inferred Mappings

Figure 5 presents the results of running Rosetta on the traces that we collected. This figure shows the number of distinct JavaME methods that we observed in the trace, grouped by the parent JavaME class to which they belong. Thus, for example, there were four unique JavaME methods belonging to the `Alert` class in our traces, namely, `Alert.setCommandListener`, `Alert.setString`, `Alert.setType`, and `Alert.setTimeout`. In all, there were 80 unique JavaME methods in our traces.

For each of these JavaME methods, we determined whether the top ten ranked mappings reported for that method contained a method (or method sequence) from the Android API that would implement its functionality. When interpreting Rosetta's output for a JavaME method, we mark a method sequence from the Android API as a valid mapping even if it is only a subsequence of a longer method sequence that implements the JavaME method's functionality. We did so because Rosetta currently only supports inference of mappings with method sequences up to length two. As reported in Figure 5, we found such valid mappings for 56 of the observed

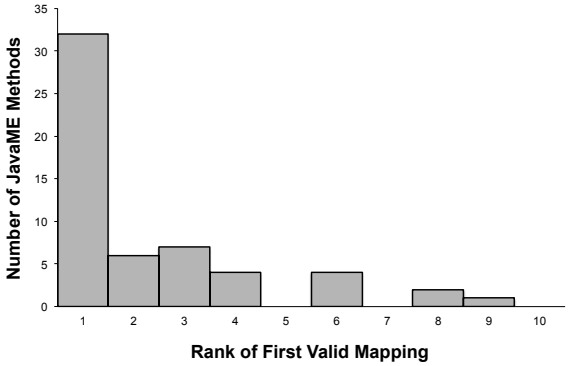

Fig. 6. This figure shows the rank distribution of the first valid mapping found for each JavaME method. In all, for each of 56 JavaME methods, a valid Android mapping appeared within the top ten results reported for that method. For 32 JavaME methods, the top-ranked mapping was a valid one.

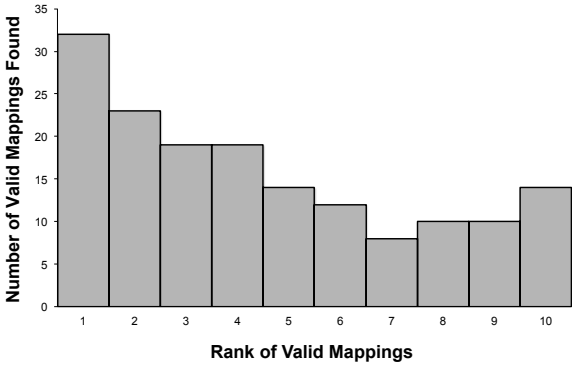

Fig. 7. This figure shows the rank distribution of all valid mappings found by Rosetta. For each rank, it shows the number of valid mappings that appear at that rank across all JavaME API methods observed in our traces. In all, we found 161 valid mappings ranking in the top ten.

JavaME methods (70%). Figure 6 depicts in more detail the rank distribution of the first valid mapping found for each of these 56 JavaME methods. The top-ranked mapping for 32 of these methods (40%) was a valid one.

Recall that a JavaME method can possibly be implemented in multiple ways using the Android API. Thus, the ranked list associated with that method could possibly contain multiple valid mappings. Rosetta's output contained a total of 161 valid mappings within the top ten results of the 56 JavaME methods. Figure 7 depicts the rank distribution of all the valid mappings found by Rosetta. Below, we illustrate a few examples of mappings inferred by Rosetta:

(**1**) The `Graphics.clipRect()` method in JavaME intersects the current clip with a specified rectangle. Rosetta correctly inferred the Android method `Canvas.clipRect()` as its top-ranked mapping.

(**2**) In JavaME `Graphics.drawChar()` draws a specified character using the current font and color. In Rosetta's output, the sequence `Paint.setColor();Canvas.drawText()`, which first sets the color and then draws text, was the second-ranked mapping for `Graphics.drawChar()`.

(**3**) The `Graphics.drawRect()` JavaME method was mapped to the Android methods `Canvas.drawRect()` (rank 1) and `Canvas.drawLines()` (rank 7), all of which can draw rectangles.

C. Impact of Individual Factors

Rosetta's output is the result of combining all the four factors discussed in §IV-B. We also evaluated the extent to which each of these factors contributes to the output. To do so, we constructed factor graphs (using the traces in Figure 4) individually with f_{freq}, f_{pos}, and f_{name}. Because f_{ctxt} correlates two mappings, we did not consider it in isolation. We also considered a few combinations of factors, to study how the mappings change as factors are added.

Variant	% Valid
f_{freq}	62.2%
f_{name}	44.0%
f_{pos}	44.0%
$f_{freq} \times f_{ctxt}$	95.6%
$f_{freq} \times f_{name}$	69.8%
$f_{freq} \times f_{pos}$	77.0%

Fig. 8. Studying the impact of various combinations of factors.

We inferred mappings with these factor graphs, and compared the resulting mappings with those obtained by Rosetta. For each factor graph variant in Figure 8, we report the fraction of Rosetta's 161 valid mappings that correspond to a valid mapping inferred by the variant. As we did with Rosetta, we only report valid mappings that rank in the top ten in the output of these factor graph variants. As Figure 8 shows, the addition of more factors generally improves the accuracy of the reported mappings. This is because the addition of more factors incorporates more trace features into the probabilistic inference algorithm, thereby removing spurious mappings.

D. Runtime Performance

We ran Rosetta on a PC with an Intel Core2 Duo CPU, running at 2.80Ghz and 4GB memory, running Ubuntu 10.04. For each of the traces reported in Figure 4, Rosetta took between 2–55 minutes to analyze traces and output mappings. On average, approximately 80% of this time was consumed by the algorithms in §IV-B which produce factor graphs, and 20% was consumed by the factor graph solver. The solver consumed 2GB memory for the largest of our factor graphs.

As discussed, Rosetta currently supports inference on method sequences of length up to two. We also configured Rosetta to work with longer method sequences, which would produce larger factor graphs. However, the factor graph solver ran out of memory in these cases.

E. Experiments with MicroEmulator

There have been some recent efforts [1], [5], [18] to allow the execution of legacy JavaME applications on the Android platform. MicroEmulator [18] is one such open-source tool that works on JavaME applications. It rewrites JavaME API calls in the input application with the corresponding Android API calls. The implementation of MicroEmulator therefore implicitly defines a mapping between the JavaME and Android graphics APIs. However, MicroEmulator does not translate the entire JavaME graphics API, and only allows JavaME applications with rudimentary GUIs to execute on Android devices. Nevertheless, the mappings that it does implement provide a basis to evaluate Rosetta.

We considered a subset of five JavaME games from Figure 4 that MicroEmulator could successfully translate (Chess, Minesweeper, Snake, Sudoku, TicTacToe), executed

them, and collected the corresponding traces of JavaME API calls. We then repeated the same set of experiments on the MicroEmulator-converted versions of these applications and collected the corresponding traces of Android API calls, and fed these trace pairs to Rosetta. In our evaluation, we determined whether the API mappings inferred by Rosetta contained the mappings implemented in MicroEmulator.

Across the JavaME traces for these five games, we observed 18 distinct JavaME graphics API methods translated by MicroEmulator. In Rosetta's output, we found at least one valid mapping for 17 of these JavaME methods within the top ten ranked results for the corresponding JavaME method. Rosetta only failed to discover an API mapping for the JavaME method `Alert.setString()`. Out of 17 JavaME methods with valid mappings, 8 methods had valid top-ranked mappings to their Android counterparts.

F. Threats to Validity

There are a number of threats to the validity of our results, which we discuss now. First, although we attempted to find JavaME and Android games that are functionally equivalent, differences do exist in their implementations. This is because JavaME is an older mobile platform that does not support as rich an API as Android; many Android calls do not even have equivalents in JavaME. Together with randomness that is inherent in certain board games (§IV-A), this could result in traces which contain Android calls implementing functionality unseen in the source application. They may mislead the attributes used by our inference algorithm (*e.g.,* frequency of calls), leading to both invalid mappings as well as valid mappings being suppressed in the output.

Second, there is no "ground truth" of JavaME to Android mappings available to evaluate Rosetta's output (the mappings in MicroEmulator aside), as a result of which we are unable to report standard metrics, such as precision and recall. We interpreted Rosetta's results by consulting API documentation. Such natural language documentation is inherently ambigious and prone to misinterpretation. We mitigated this threat by having two authors independently cross-validate the results.

Finally, a threat to external validity, *i.e.,* the extent to which our results can be generalized, comes from the fact that we only inferred mappings for a pair of graphics APIs using board games. It is unclear how many mappings we would have inferred using other graphical applications (*e.g.,* office applications) or for other families of APIs.

VI. Related Work

An upcoming survey article by Robillard *et al.* [26, §6] provides a good overview of prior work on mining API mappings. Among these, the work most directly related to ours is the MAM project [34]. MAM's goal is the same as Rosetta's, *i.e.,* to mine software repositories to infer how a source API maps to a target API. The MAM prototype was targeted towards Java as the source API and C# as the target API. Despite sharing the same goal, MAM and Rosetta differ significantly in the approaches that they use, each with its advantages and drawbacks. To mine API mappings between a source and a target API, MAM relies on the existence of software packages that have been ported manually from the source to the target platform. For each such software package, MAM then uses static analysis and name similarity to "align" methods and classes in the source platform implementation with those of the target platform implementation. Aligned methods are assumed to implement the same functionality.

MAM's use of static analysis allows it to infer a large number of API mappings (about 25,800 mappings between Java and C#). It also allows MAM to infer likely mappings between arguments to methods in the source and target APIs, which Rosetta does not currently do. However, unlike Rosetta, MAM requires that the same source application be available on the source and target platforms. MAM's approach of aligning classes and methods across two implementations of a software package does not allow the inference of likely API mappings if there are similar, but independently-developed applications for the source and target platforms. MAM's approach is also limited in that it uses name similarity as the only heuristic to bootstrap its API mapping algorithm. In contrast, Rosetta uses a number of attributes combined together as factors, and can easily be extended to accommodate new similarity attributes as they are designed. Most importantly, while MAM uses a purely *syntactic* approach to discover likely API mappings, Rosetta's approach uses similarities in application *behavior*.

Nita and Notkin [20] develop techniques to allow developers to adapt their applications to alternative APIs. They provide a way for developers to specify a mapping between a source and a target API, following which a source to source transformation automatically completes the transformations necessary to adapt the application to the target API. Rosetta can potentially complement this work by inferring likely mappings.

Androider [28] is a tool to reverse-engineer application GUIs. Androider uses aspect-oriented programming techniques to extract a platform-independent GUI model, which can then be used to port GUIs across different platforms. While Androider provides a GUI for a target platform by analyzing GUIs of a source platform at runtime, Rosetta instead infers API mappings, and is not restricted to GUI-related APIs. Bartolomei *et al.* [2] analyzed wrappers between two Java GUI APIs and extracted common design patterns used by wrapper developers. They focused on mapping object types and identified the challenges faced by wrapper developers. Method mappings given by Rosetta can possibly be used along with their design patterns to ease the job of writing wrappers.

A number of prior projects provide tool support to assist programmers working with large, evolving APIs (*e.g.,* [6], [8], [17], [27], [31], [33], [35]). The programmer's time is often spent in determining which API method to use to accomplish a particular task. These projects use myriad techniques to develop programming assistants that ease the task of working with complex APIs. Rosetta is complementary to these efforts, in that it works with *cross-platform APIs*.

A related line of research is on resources for API learning (*e.g.,* [3], [7], [11], [23], [29]). These projects attempt to ease

the task of a programmer by synthesizing API usage examples, evolution of API usage, and extracting knowledge from API documentation. Again, most of these techniques work on APIs on a single platform. Rosetta's *cross-platform* approach can possibly be used in conjunction with these techniques to facilitate cross-platform API learning.

Finally, Rosetta's probabilistic inference approach was inspired by other uses of factor graphs in the software engineering literature. Merlin [15] uses factor graphs to classify methods in Web applications as sources, sinks and sanitizers. Such specifications are useful for verification tools, which attempt to determine whether there is a path from a source to a sink that does not traverse through a sanitizer. To infer such specifications, Merlin casts a number of heuristics as factors, and uses belief propagation. Kremenek *et al.* [12] also made similar use of factor graphs to infer specifications of allocator and deallocator methods in systems code. Factor graphs are just one approach to mining specifications; a number of prior software engineering projects have considered other mining techniques *e.g.,* [9], [14], [16]. Future work could consider the use of such techniques in Rosetta as well.

VII. Summary and Future Work

We presented a generic approach to infer mappings between the APIs of a source and target platform. Our Rosetta prototype used this approach to infer likely mappings between the JavaME and Android graphics APIs. While we are encouraged thus far by the success of Rosetta, future work could extend it in a number of ways to overcome its current shortcomings.

(**1**) *Mapping method arguments.* Rosetta could be extended to also discover mappings between method call arguments. To do so, the tracing infrastructure must be extended to log method arguments and their types, and new factors and inference techniques must be developed to map arguments.

(**2**) *Mapping longer method sequences.* Although our framework supports the inference of mappings between method sequences of arbitrary length, we restricted the Rosetta prototype to sequences of length two to limit the number of factors produced by our algorithms in §IV-B. Future work could investigate optimizations to prune the number of factors, so that Rosetta can scale to longer method sequences.

(**3**) *Hybrid approaches.* Rosetta's use of runtime techniques fundamentally limits the mappings inferred to only those source and target API methods that appear in the trace. It may be possible to increase coverage using a hybrid technique that combines static analysis (*e.g.,* MAM [34]) with Rosetta's trace-based approach.

(**4**) *More platforms.* Finally, the Rosetta prototype can be extended to work on larger subsets of the JavaME and Android APIs, and also to infer mappings between other API pairs (*e.g.,* iOS and Android).

Acknowledgments

We thank Shakeel Butt, Liviu Iftode, Pratyusa Manadhata, and Tina Eliassi-Rad for their contributions to the project. This work was funded by NSF grants 0931992 and 1117711.

References

[1] Assembla. J2ME Android bridge. http://www.assembla.com/spaces/j2ab/wiki.
[2] T.T. Bartolomei, K. Czarnecki, and R. Laandmmel. Swing to SWT and back: Patterns for API migration by wrapping. In *ICSM*, 2010.
[3] R. Buse and W. Weimer. Synthesizing API usage examples. In *ICSE*, 2012.
[4] OW2 Consortium. ASM toolkit. http://asm.ow2.org.
[5] Netmite Corporation. App runner. http://www.netmite.com/android/.
[6] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. *ACM TOSEM*, 20(4), 2011.
[7] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *ICSE*, 2012.
[8] D. Dig, C. Comertoglu, D. Marinov, R. Johnson, and D. Thomas. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.
[9] D. Engler *et al*. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP*, 2001.
[10] Google. Android API reference. http://developer.android.com/reference/packages.html.
[11] S. Hens, M. Monperrus, and M. Mezini. Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In *ICSE*, 2012.
[12] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, 2006.
[13] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theory*, 47(2), 2001.
[14] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, 2005.
[15] B. Livshits, A. Nori, S. Rajamani, and A. Banerjee. Merlin: Inferring specifications for explicit information flow problems. In *PLDI*, 2009.
[16] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *FSE*, 2005.
[17] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *PLDI*, 2005.
[18] Microemulator. http://www.microemu.org/.
[19] K. Murphy. Bayes Net toolbox for Matlab, October 2007. http://code.google.com/p/bnt/.
[20] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *ICSE*, May 2010.
[21] Oracle. Java ME API reference. http://docs.oracle.com/javame/config/cldc/ref-impl/midp2.0/jsr118/index.html.
[22] Oracle. Sun Java wireless toolkit for CLDC, 2.5.1. http://java.sun.com/products/sjwtoolkit/download-2_5_1.html.
[23] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. M. Paradkar. Inferring method specifications from natural language API descriptions. In *ICSE*, 2012.
[24] python-levenshtein-0.10.2. pypi.python.org/pypi/python-Levenshtein.
[25] Qt API mapping for iOS developers. http://www.developer.nokia.com/Develop/Porting/API_Mapping.
[26] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE TSE*. To appear.
[27] T. Schafer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *ICSE*, 2008.
[28] E. Shah and E. Tilevich. Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms. In *Workshop on Next-generation Applications of Smartphones*, 2011.
[29] G. Uddin, B. Dagenais, and M. P. Robillard. Temporal analysis of API usage concepts. In *ICSE*, 2012.
[30] Windows phone interoperability: Windows phone API mapping. http://windowsphone.interoperabilitybridges.com/porting.
[31] Z. Xing and E. Stroulia. API-evolution support with DiffCatchUp. *IEEE TSE*, 33(12), 2007.
[32] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 2003.
[33] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, and J. Zhao. Automatic parameter recommendation for practical API usage. In *ICSE*, 2012.
[34] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *ICSE*, 2010.
[35] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending API usage patterns. In *ECOOP*, 2009.