Scalable and Approximate Program Dependence Analysis

Seongmin Lee
bohrok@kaist.ac.kr
Korea Advanced Institute of Science and Technology
Yuseong-gu, Daejeon, Republic of Korea

ABSTRACT

Program dependence is a fundamental concept to many software engineering tasks, yet the traditional dependence analysis struggles to cope with common modern development practices such as multilingual implementations and use of third-party libraries. While Observation-based Slicing (ORBS) solves these issues and produces an accurate slice, it has a scalability problem due to the need to build and execute the target program multiple times. We would like to propose a radical change of perspective: a useful dependence analysis needs to be scalable even if it approximates the dependency. Our goal is a scalable approximate program dependence analysis via estimating the likelihood of dependence. We claim that 1) using external information such as lexical analysis or a development history, 2) learning dependence model from partial observations, and 3) merging static, and observation-based approach would assist the proposition. We expect that our technique would introduce a new perspective of program dependence analysis into the likelihood of dependence. It would also broaden the capability of the dependence analysis towards large and complex software.

KEYWORDS

Program Analysis, ORBS, Program Slicing

ACM Reference Format:

1 PROBLEM AND RESEARCH STATEMENT

Understanding dependencies between program elements is a fundamental task in software engineering [18]. Program dependence analysis provides a basis for many software engineering tasks, including program comprehension [23], software testing [4], maintenance [10], refactoring [9], security [14], and debugging [13]; hence, it often acts as a prelude for other tasks. While, traditionally, program dependency analysis is done in a static approach based on the dependence graph [11], it suffers from various issues. Static dependency analysis is not capable of handling a multi-lingual system, a system tightly coupled with the external libraries or other systems such as databases and web services. In those cases, one gets the result containing lots of false positives or no result at all.

Observation-based slicing (ORBS) [5] considers the dependency relation from a different perspective. If a program element, $e_1$, behaves the same when another program element, $e_2$, is deleted, ORBS determines that $e_1$ is independent of $e_2$. ORBS generates a program slice for a given program element by iteratively deleting other program elements while observing the behavior of the target program element. If the behavior changes, ORBS rolls back the deletion. ORBS moves on to other program elements and repeats, until there is no deletable program element. ORBS’s purely dynamic approach solves the issue of the static dependence analysis and generates the slice without false positive. Yet, it requires lots of compilations and executions, making it not scalable.

While it is promising that ORBS provides accurate dependence information, the lack of scalability makes ORBS not suitable for the role of an informant for other tasks that applied to a large system. Therefore, program dependence analysis for a large and complex systems is still an open issue. Here, we argue that it would be much applicable if the analysis scales, even if the result approximates the program dependence.

2 GOAL AND RESEARCH HYPOTHESIS

The goal we propose is a scalable approximate program dependence analysis. Instead of scrutinizing every dependence through observations with 'true' or 'false', we aim to estimate the likelihood of the existence of a dependence relationship using various external information and applying statistical methods. Compared to the classical Boolean representation, representing the dependence as a likelihood has the advantage of being able to sort the program element for the postprocessing performed in tasks such as debugging or automated program repair. Our main hypothesis is that we can approximate the program dependence without using traditional static analysis.

We first consider whether the lexical similarity (e.g., two lines that both include the string `tax_rate`) can provide a useful approximation to the dependence information [17]. For example, a source line, $l_1$, might assign the variable `tax_rate`, which is subsequently used in $l_2$: `tax = tax_rate * sale`. The dependence between $l_1$ and $l_2$ might be estimated with the lexical connection between $l_1$ and $l_2$ (i.e., the common occurrence of the words ‘tax’ and ‘rate’).

We evaluate the performance of the lexical analysis on dependence approximation with its application of program slicing.

While a complete set of observations can produce a dependence model without any false positive, we investigate whether we can learn a dependence model from the partial observations. By setting the location of modification of the program as an independent variable and the observation as a dependent variable, we can model the dependency relation with the various statistical models.
While ORBS showed that observing the runtime behavior can be a good indicator of the dependence, the cost of compilations and executions remains a heavy burden. We suggest that, without actually running the test for artificially changed programs, we can extract the observations from the program’s development history with the result of the regression test. Mining the observations from history, rather than generating them, would significantly reduce the cost of compilations and executions.

Finally, we will investigate whether merging static and observation-based approach can increase the scalability of the program dependence analysis. When static dependence analysis builds a dependency graph, it may not know the existence of some of the edges due to the limitation of the static analysis. Instead of drawing the edges for safe analysis, we will discover their existence by modeling the dependencies with the observation-based approach. Focusing on what each approach performs well, the whole program dependence analysis may become scalable.

3 PROPOSED RESEARCH AND PRELIMINARY RESULTS

The planned thesis will consist of four parts. The first [17] and the second [16] part have already been published; there are ongoing collaborations for the third and the fourth part.

3.1 Evaluating lexical approximation of program dependence

We first try the lexical analysis as a way to approximate the program dependence. The main claim here is that the lexical similarity between the source code lines may be a good proxy for the functional similarity [20]. Developers often name an identifier or a method using words that are associated with the functionality it implements. In addition, a lexical analysis preserves the language-agnostic nature, as the observation-based approach does.

We use two methods to analyze the source code lexically. The first method is the Vector Space Model (VSM), which has been used in Information Retrieval (IR) to calculate the distances between a collection of text documents and a query [21]. The second method is Latent Dirichlet Allocation (LDA), which models a collection of documents using two probability distributions: each document is represented as a probability distribution of topics where each topic is a probability distribution over the words in the vocabulary [6]. These two lexical models represent each code line as a numerical vector. The similarity between the code lines is computed by the cosine similarity between the corresponding numerical vectors.

We evaluate the performance of whether a lexical analysis can approximate the program dependence with its application of program slicing. The original ORBS uses a bounded deletion window operator that iteratively attempts to delete a few consecutive lines together (we call the operator “window-deletion” and the original ORBS “W-ORBS”). Assuming that we know several code lines functionally related to each other, if one of the lines can be deleted from the slice, we may safely delete other lines. Hence, we develop two lexical deletion operators that try to delete a set of lines, whose similarities between them are above a certain threshold, together. We name each of the lexical deletion operator as vsm-deletion and lda-deletion according to which lexical model it uses. If the lexical model could approximate the functionality well, the lexical deletion operator would delete a large number of lines together, which will increase the efficiency of the slicing.

We compare VSM- and LDA- to W-ORBS on 24 slicing criteria from six C programs in Siemens suite [12] and three real-world open source Java project: commons-cli, commons-csv from Apache Commons Project, and guava. The result shows that, while the lexical deletion operator deletes fewer lines, it uses significantly fewer compilations and executions, reducing wall clock time. The lexical deletion operators using either the lexical model are highly attractive in terms of their per-deleted-line efficiency.

A small and efficient deletion by the lexical deletion operators shows that the lexical analysis could approximate the partial dependence of the program. The result motivates us to apply lexical deletion operators, which efficiently perform in certain parts of the program, together with the window-deletion, which generally performs in detail, to achieve both efficiency and effectiveness. To study the range of possibilities, we introduce MOBS: Multi-operator Observational Slicing, which selectively applies multiple deletion operators while slicing concerning the probability distribution over the deletion operators [17]. Based on the experiment result performed on the same slicing criteria with above, MOBS could be both effective and efficient, being capable of deleting an average of 69% of the number of lines deleted by W-ORBS, while requiring only 36% of wall clock execution time required by W-ORBS.

Through the research, we show that lexical analysis could approximate the partial dependence of the program. The result of VSM-, LDA-ORBS, and MOAD demonstrates that we could increase the scalability of the program dependence analysis via approximate dependence information.

3.2 Learning approximate dependence from partial observations

While ORBS generates the 1-minimal slice [5] with no false positive, all the compilations and executions spent are only used to figure out the dependency relation for a single slicing criterion, and cannot be reused. Thus, it takes a huge amount of effort to analyze the dependency relation on multiple program elements. Hence, rather than focusing on generating an exact slice for a single program element, we investigate approximate dependence analysis on the entire program by learning from partial observations.

Our technique, called MOAD (Modeling Observation-based Approximate Dependence) [16], reformulates program dependence as the likelihood that one program element is dependent on another, instead of the classical Boolean relationship. MOAD generates various partial programs by deleting program elements from the original program. Running each of the partial programs with its test cases, MOAD observes the impact of the deletions on various program points. From the observations, MOAD infers a program dependence model that explains the dependency relation between the modification point and the observation point.

In detail, for an input program, MOAD indexes the program elements (e.g., statements) as candidates for the deletion. Partially deleted programs are constructed by deleting one or more program elements. We made two deletion schema, 1-hot and 2-hot, that generates a set of modified programs. 1-hot deletes every program
element one at a time. In addition to what 1-hot deleted, 2-hot also
deltes every pair of the program elements. All variables in every
position in the program become the observation points of MOAD.
While running the modified programs, MOAD records whether
each of the variable shows the same behavior with the original
program or not, and saves the data for training the dependency
model.

We introduce three different inference models for program de-
pendence. Once Success model determines that the variable is in-
dependent of the deleted program element if the behavior of the
variable preserved at least once in the training data. The second
inference model, Logistic, trains a logistic regression model using
the deletion of each program element as the dependent variable and
the behavior of the variable as the independent variable. Then, each
learned coefficient represents the relative impact of the program
element on the variable. The model infers the dependence with
the sign of the coefficient. The third inference model is Bayesian
model. The probability of a variable behaves the same under the
condition of specific program elements that have been deleted in-
dicates how the variable is independent of the program element.
The model estimates the probability with the frequency of such
behavior preservation, and infers the dependence by whether the
probability is less than the average over all program elements.

We evaluate MOAD by its application to program slicing. We
investigate the number of observations needed, the size of the slices,
and the difference of the slices compared to ORBS slice over ten C
programs, which are widely used on program slicing study.

In terms of efficiency, the result shows that the 1-hot scheme
requires multiple orders of magnitude fewer observations compared
to what ORBS needed. Also, the 2-hot scheme uses less than 20%
of observations compared to ORBS. While the cost significantly
reduced by MOAD, the size of the slices generated by MOAD is 45%
larger than ORBS’s slice, on average. Among three inference models,
the best result is shown by the Once Success model; on average,
the size of the slice using 2-hot scheme data is only 16% larger than
the ORBS’s slice. Further analysis has shown that the Once Success
model generates not only the smallest slice but also the most similar
slice considering the number of missed or excessively deleted lines
compared to ORBS’s slice.

Through the research, we show that, the dependency relation in
the program could be approximated through learning the statistical
model using the subset of the observations.

3.3 Observing information from the history

The fundamental principle of the observation-based approach is
that we can model the program dependence using the information
of the change of the program and the corresponding behavioral
change. Our previous two research focus on the dependence anal-
ysis on the program of the present. To do so, we need to make
various artificial changes to the program and observe the effects.
Every observation of artificial changes needs a compilation and an
execution; hence, it burdens the observation-based approach.

We propose that, instead of compiling and executing the changed
program of the present, we can observe the behavioral difference
from the change of the result of the regression testing performed
during the development process. Here, every intermediate version
of the program in the development history becomes an observation
candidate, and every change between two consecutive program
versions becomes a cause of the behavioral difference. If, after a
change, there is a test whose result is different from what it was
before, it indicates that the code under the test depends on what just
have changed. Gathered observations from the regression testing
can be used to approximate the program dependence.

Using the intermediate version of the program as an observation
candidate gives an advantage in terms of efficiency. The artificial
changes, such as deleting a random code line or statement made
by ORBS, ignore the syntax of the program. Thus, most of them
produce a compilation error or a runtime error, which leads to a
waste of effort. In contrast, the intermediate version of the program
in the development history is usually required to be compilable
and executable before it pushed to the repository.

There are some issues to be considered when using the infor-
mation of the development history for dependence analysis. First,
we need a test case that covers the change of the source code to
observe its effect. In case of a bug fix, the test case that reveals
the bug covers the fix. Existing test cases may not cover the change
that adds a new feature to the program immediately. If the test case
that covers the new feature is introduced later in the development
history, we may use it. If there is no such test case, we may use an
automated test case generation technique to generate the test case
that covers the new feature.

It is a well-known problem that the huge or tangled code change
can debase the code review system. Similarly, a huge and tangled
code change creates another issue that makes it difficult to figure
out which part of the change causes the observed difference of the
program behavior. We may use existing untangling code change
techniques [8] to split the code change by the purpose. the effect of
each of the split changes can be observed through running the test
regarding them as consecutive individual changes.

While using the result of the regression testing reduces the cost
of compilation and execution, it also restricts the observation to
the pass and failure of the regression test. If there are not enough
observations due to the restriction, we may use an automated test
case generation technique to create and execute the necessary test
case, and fill in the insufficient observation.

3.4 Merging static and observation-based
approach

Although static analysis can analyze the large-scale source code
without the program compilation and execution, it cannot be ap-
plied (or have large false positives) if the source code is inaccessible
or the semantic is unknown, in case of another programming lan-
guage. The observation-based approach resolves this by verifying
dependencies only through the observation of run-time informa-
tion without static modeling. We propose to combine these two
complementary approaches to create a dependence model.

As a related work, ‘dynamic’ slicing is a concept introduced by
Korel and Laski [15]. It slices the program while preserving the
behavior on a subset of inputs. While the original work suggests
several algorithms to compute dynamic slices, the majority of the
subsequent work on dynamic slicing ‘defines’ dynamic slicing based
on the algorithms [1, 7]. To the author’s knowledge, although many
We propose a scalable approximate program dependence analysis. We expect to complete the proposed work within two years. During the first year, we will focus on searching the software development data used for observation. We will target an open-source repository that 1) has been developed in the long-term, and also 2) has a regression test. Once we find several candidate repositories, we will extract the observation, and do the initial program dependence analysis. If it is required, we will perform an untangling code change technique on tangled code changes. Based on the initial result, we will further discuss whether any additional observation is needed. The second year will be spent merging static and observation-based approach. We will first concentrate on selecting the benchmark program and the static analysis tool to cooperate with. Then, the main issue of the development is to figure out which form of dependency relation could two analysis compatible with. We will apply the merged analysis on the benchmark, and discuss the result.

5 ANTICIPATED CONTRIBUTIONS

We propose a scalable approximate program dependence analysis. We investigate various approaches that can significantly increase the scalability, while they may approximate the dependence. Our research will enables dependence analysis on large and complex software, which existing program dependence analysis is incapable of. Since the dependence analysis serves as the cornerstone of other software engineering tasks, our research may broaden the applicability of program comprehension, software testing, maintenance, refactoring, security, and debugging.

Our research also aims to achieve a breakthrough in the program dependence analysis itself. Representing the program dependence with the likelihood makes it possible to order the program elements considering the relative dependence on another program element. This will allow prioritizing the program elements for other software techniques that need to select program elements considering the dependencies, such as software debugging or automated program repair; hence, it will enhance the efficiency of the techniques.

REFERENCES