The Journal of Systems and Software 122 (2016) 311–326

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Method-level program dependence abstraction and its application to impact analysis

Haipeng Cai[a,*], Raul Santelices[b]

[a] School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, 99163
[b] Delphix, Atlanta, GA, 30022

ARTICLE INFO

Article history:
Received 14 September 2015
Revised 22 June 2016
Accepted 27 September 2016
Available online 28 September 2016

Keywords:
Dependence analysis
Dependence abstraction
Method dependence graph (MDG)
Impact analysis
Accuracy
Cost-effectiveness

ABSTRACT

The traditional software dependence (TSD) model based on the system dependence graph enables precise fine-grained program dependence analysis that supports a range of software analysis and testing tasks. However, this model often faces scalability challenges that hinder its applications as it can be unnecessarily expensive, especially for client analyses where coarser results suffice.

This paper revisits the static-execute-after (SEA), the most recent TSD abstraction approach, for its accuracy in approximating method-level forward dependencies relative to the TSD model. It also presents an alternative approach called the method dependence graph (MDG), compares its accuracy against the SEA, and explores applications of the dependence abstraction in the context of dependence-based impact analysis.

Unlike the SEA approach which roughly approximates dependencies via method-level control flows only, the MDG incorporates more fine-grained analyses of control and data dependencies to avoid being overly conservative. Meanwhile, the MDG avoids being overly expensive by ignoring context sensitivity in transitive interprocedural dependence computation and flow sensitivity in computing data dependencies induced by heap objects.

Our empirical studies revealed that (1) the MDG can approximate the TSD model safely, for method-level forward dependence at least, at much lower cost yet with low loss of precision, (2) for the same purpose, while both are safe and more efficient than the TSD model, the MDG can achieve higher precision than the SEA with better efficiency, both significantly, and (3) as example applications, the MDG can greatly enhance the cost-effectiveness of both static and dynamic impact analysis techniques that are based on program dependence analysis.

More generally, as a program dependence representation, the MDG provides a viable solution to many challenges that can be reduced to balancing cost and effectiveness faced by dependence-based tasks other than impact analysis.

1. Introduction

Program dependence analysis has long been underlying a wide range of software analysis and testing techniques (e.g., Podgurski and Clarke, 1990; Bates and Horwitz, 1993; Santelices and Harrold, 2010; Baah et al., 2010). While traditional approaches to dependence analysis offer fine-grained results (at statement or even instruction level) (Ferrante et al., 1987; Horwitz et al., 1990), they can face severe scalability and/or usability challenges, especially with modern software of growing sizes and/or increasing complexity (Jász et al., 2008; Acharya and Robinson, 2011a), even more so when high precision is demanded with safety guarantee (Jackson and Rinard, 2000; Binkley, 2007).

On the other hand, for many software-engineering tasks where results of coarser granularity suffice, computing the finest-grained dependencies tends to be superfluous and ends up with low cost-effectiveness in particular application contexts—in this work, a (dependence) analysis is considered cost-effective (measured by the ratio of effectiveness to cost) if it produces effective (measured by accuracy, or precision alone if with constantly perfect recall) results relative to the total overhead it incurs (including analysis cost and human cost inspecting the analysis results) (Cai et al., 2016). One example is impact analysis (Bohner and Arnold, 1996), which analyzes the effects of specific program components, or changes to them, on the rest of the program to support software evolution and

* Corresponding author.
E-mail addresses: hcai@eecs.wsu.edu (H. Cai), rasantel@gmail.com (R. Santelices).

http://dx.doi.org/10.1016/j.jss.2016.09.048
0164-1212/

many other client analyses, including regression testing (Jász et al., 2012; Schrettner et al., 2014) and fault localization (Ren et al., 2006). For such tasks as impact analysis, results are commonly given at method level (Law and Rothermel, 2003; Apiwattanapong et al., 2005; Jász, 2010), where fine (e.g., statement-level) results can be too large to fully utilize (Acharya and Robinson, 2011a). In other contexts such as program understanding, method-level results are also more practical to explore than those of the finest granularity.

Driven by varying needs, different approaches have been explored to abstract program dependencies to coarser levels, including the *program summary graph* (Callahan, 1988) used to speed up interprocedural data-flow analysis, the *object-oriented class-member dependence graph* (Sun et al., 2010), the *lattice of class and method dependence* (Sun et al., 2011), the *influence graph* (Breech et al., 2006), that are all used for impact analysis, and the *module dependence graph* (Mancoridis et al., 1999) used for understanding and improving software structure. While these abstractions have been shown useful for their particular client analyses, they either capture only partial dependencies among methods (Breech et al., 2006; Sun et al., 2010) or dependencies at levels of classes (Sun et al., 2011) even files (Mancoridis et al., 1999), which can be overly coarse for many dependence-based tasks. More critically, most such approaches were not designed or fully evaluated as a general program dependence abstraction with respect to their accuracy (both precision and recall) against that of the original full model they approximate as ground truth.

Initially intended to replace traditional software dependencies (TSD) that are based on the system dependence graph (SDG) (Horwitz et al., 1990; Jász et al., 2008), a method-level dependence abstraction, called the *static-execute-after/before* (SEA/SEB) (Jász et al., 2008), has been proposed recently. It abstracts dependencies among methods based on the interprocedural control flow graph (ICFG) and was reported to have little loss of precision with no loss of (100%) recall relative to static slicing based on the TSD model (i.e., the SDG). Later, the SEA was applied to static impact analysis shown more accurate than peer techniques (Tóth et al., 2010) and capable of improving regression test selection and prioritization (Schrettner et al., 2014).

However, previous studies on the accuracy of SEA/SEB either exclusively targeted procedural programs (Jász et al., 2008), or focused on backward dependencies based on the SEB (against backward slicing on top of the SDG) only (Jász, 2010). The remaining relevant studies addressed the accuracy of SEA-based forward dependencies, with some indeed using object-oriented programs and compared to forward slicing on the TSD model, yet the accuracy of such dependencies was assessed either not at the method level, but at class level only (Beszédes et al., 2007), or not relative to ground truth based on the TSD model, but those based on repository changes (Jász et al., 2012; Schrettner et al., 2014) or programmer opinions (Tóth et al., 2010), and only in the specific application context of impact analysis.

While forward dependence analysis is required by many dependence-based applications, including *static* impact analysis that the SEA/SEB has been mainly applied to, the accuracy of this abstraction with respect to the TSD model, for forward dependencies and object-oriented programs in particular, remains unknown. In addition, it has not yet been explored whether and, if possible, how such program dependence abstractions would improve *dynamic* analysis, especially hybrid ones that utilize both static dependence and execution data of programs, such as hybrid dynamic impact analysis (Maia et al., 2010; Cai and Santelices, 2014, 2015b).

In this paper, we present and study an alternative method-level dependence abstraction using a program representation called the method dependence graph (MDG). In comparison to the SDG-based TSD models which represent a program in terms of the data and control dependencies among all of its statements, an MDG serves also as a general graphical program representation, but models those dependencies at method level instead. The method-level dependencies could be simply obtained from a TSD model by lifting statements in the SDG up to corresponding (enclosing) methods. Yet, our MDG model represents these dependencies directly with statement-level details within methods (i.e. intraprocedural dependencies) abstracted away and, more importantly, does so with much less computation than constructing the SDG would require. The MDG computes transitive interprocedural dependencies in a context-insensitive manner with flow sensitivity dismissed for heap-object-induced data dependencies too. Thus, it is more efficient than TSD models (Horwitz et al., 1990; Yu and Rajlich, 2001). On the other hand, this abstraction captures whole-program control and data dependencies, including those due to exception-driven control flows (Sinha and Harrold, 2000), thus it is more informative than coarser models like call graphs or ICFG. With the MDG, we attempt to not only address the above questions concerning the latest peer approach SEA/SEB, but also to attain a more cost-effective dependence abstraction over existing alternative options in general.

We implemented the MDG and applied it to both static and dynamic impact analysis for Java,[1] which are all evaluated on seven non-trivial Java subject programs. We computed the accuracy of the MDG for approximating forward dependencies in general and the cost-effectiveness of its specific application in static impact analysis; we also compared the accuracy and efficiency of the MDG with respect to the TSD as ground truth against the SEA approach. To explore how the MDG abstraction can be applied to and benefit dynamic analysis, we developed on top of the MDG a variant of DIVER, the most cost-effective hybrid dynamic impact analysis in the literature (Cai and Santelices, 2014), and compared its cost and effectiveness against the original DIVER.

Our results show that the MDG can approximate the TSD model with perfect recall (100%) and generally high precision (85–90% mostly) with great efficiency, at least for forward dependencies at the method level. We also found that the MDG appears to be a more cost-effective option than the SEA for the same purpose, according to its significantly higher precision with better overall efficiency. The study also reveals that, for the object-oriented programs we used at least, SEA can be much less precise for approximating forward dependencies at method level than previously reported at class level for object-oriented programs (Beszédes et al., 2007) and at method-level for procedural programs (Jász et al., 2008; Jász, 2010). The study also demonstrated that the MDG as a dependence abstraction model can significantly enhance the cost-effectiveness of both the dependence-based static and dynamic impact analysis techniques over the respective existing best alternatives. More broadly, the MDG as a general program abstraction approach could benefit any applications that are based on program dependencies at method level (e.g., testing and debugging) and that utilize the dependencies at this or even higher levels (e.g., refactoring and performance optimizations).

In summary, the contributions of this paper are as follows:

- An approach to abstracting program dependencies to method level, called the MDG, that can approximate traditional software dependencies more accurately than existing options, including dependencies due to exception-driven control flows (Section 3).
- An implementation of the MDG and two application analyses based on it, a static impact analysis and a hybrid dynamic impact analysis (Section 4.1).

[1] Source code, documentation, and study results are available at https://chapring.github.io/mdg.