Field-Sensitive Program Dependence Analysis

Shay Litvak
Tel Aviv University & Panaya Inc.
shay.litvak@cs.tau.ac.il

Nurit Dor
Panaya Inc.
nurit@panayainc.com

Rastislav Bodik
University of California, Berkeley
bodik@cs.berkeley.edu

Noam Rinetzky
Queen Mary University of London
maon@dcs.qmul.ac.uk

Mooly Sagiv
Tel Aviv University
msagiv@tau.ac.il

ABSTRACT

Statement st transitively depends on statement st_{seed} if the execution of st_{seed} may affect the execution of st. Computing transitive program dependences is a fundamental operation in many automatic software analysis tools. Existing tools find it challenging to compute transitive dependences for programs manipulating large aggregate structure variables, and their limitations adversely affect analysis of certain important classes of software systems, e.g., large-scale *enterprise resource planning* (*ERP*) systems.

This paper presents an efficient conservative interprocedural static analysis algorithm for computing field-sensitive transitive program dependences in the presence of large aggregate structure variables. Our key insight is that program dependences coming from operations on whole substructures can be precisely (i.e., field-sensitively) represented at the granularity of substructures instead of individual fields. Technically, we adapt the interval domain to concisely record dependences between *multiple* pairs of fields of aggregate structure variables by exploiting the fields' spatial arrangement.

We *prove* that our algorithm is as precise as any algorithm which works at the granularity of individual fields, the most-precise known approach for this problem. Our empirical study, in which we analyzed industrial *ERP* programs with over 100,000 lines of code in average, shows significant improvements in both the running times and memory consumption over existing approaches: The baseline is an efficient field-insensitive *whole-structure* that incurs a 62% false error rate. An *atomization*-based algorithm, which disassemble every aggregate structure variable into the collection of its individual fields, can remove all these false errors at the cost of doubling the average analysis time, from 30 to 60 minutes. In contrast, our new precise algorithm removes all false errors by increasing the time only to 35 minutes. In terms of memory consumption, our algorithm increases the footprint by less than 10%, compared to 50% overhead of the atomizing algorithm.

Categories and Subject Descriptors: D.2.4 [SOFTWARE ENGINEERING]: Software/Program Verification

General Terms: Verification

Keywords: ADAS, Aggregate structure variables, ERP, Field-sensitivity, Transitive program dependences

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

1. INTRODUCTION

A statement st has an *immediate program dependence* on a statement st_{seed} if the content of a memory location assigned by st_{seed} affects either the value computed by st or whether st is executed at all. A statement st has a *transitive program dependence* on a statement st_{seed} if st and st_{seed} are in the transitive closure of the immediate program dependence relation. Computing transitive program dependences is a fundamental operation in many software understanding and manipulation tools. These tools provide valuable mechanized support for the practicing software engineer. For example, transitive program dependences are used in tools for program maintenance [7, 8], debugging [18], testing [4, 3], semantic differencing [11], slicing [32, 14], reuse [22], and merging [13].

The problem of computing transitive program dependences has been extensively studied. Most of the advances in the efficiency and precision of the analysis come from improving the analysis' *context sensitivity*, i.e., its treatment of procedure calls (see, e.g., [27, 28]) and *object-field sensitivity*, i.e., its treatment of dynamically dispatched methods in object-oriented programs (see, e.g., [29]). In contrast, the problem of improving the analysis' *field-sensitivity*, i.e., its treatment of *large aggregate structure variables*[1] has not received much attention. This is rather unfortunate because, as we have found in our extensive empirical study, imprecise or costly handling of fields in aggregate structure variables may have adverse effect on the quality of the analysis' results and its applicability to certain classes of important software systems, e.g., large-scale enterprise resource planning (*ERP*) systems.

There are two known approaches for computing program dependences for programs manipulating large aggregate structure variables. The *whole structure* (*WS*) approach [23, 19, 20] treats the whole structure as a single variable of a primitive type. Intuitively, it handles a read statement from a field f of a structure variable v as if it may read from any of the fields of v, and a write statement to a field f of v as if it may write to any field of v. The *atomization* (*ATOM*) approach [26, 21] reduces the field-sensitive dependence problem into a dependence problem for programs without aggregate structures by disassembling aggregate structures to their primitive components. Unfortunately, in our experimental study, in which we analyzed industrial *ERP* programs with over 100,000 lines of code in average, we found the *WS* approach to be efficient but imprecise and the *ATOM* approach to be precise but inefficient. (See Sections 2 and 8.)

In this paper we present *ADAS*, an interprocedural static analysis algorithm for computing field-sensitive transitive program dependences in the presence of large aggregate structure variables. (*ADAS* stands for *Aggregate Dependence via Arrangement Seg-*

[1] By a *large aggregate structure variable*, we mean a record variable which contains dozens, or even hundreds, of fields.

ments.) In our experiments, we found that *ADAS* has comparable cost to the *WS* approach. In [17], we prove that *ADAS* provides the same precision as the *ATOM* approach.

Our key insight is that program dependences coming from operations on whole substructures can be precisely (i.e., field-sensitively) represented at the granularity of substructures instead of individual fields. The key reason for the efficiency of *ADAS* is the use of an interval-like abstract domain which allows to accurately record dependences between multiple pairs of (adjacent) fields in a concise manner.

ADAS works in two stages. In the first stage, it computes a *range-labeled program dependence graph*. A *program dependence graph (PDG)* is a graph whose nodes are the program statements and its edges represent immediate program dependences. In existing dependence analyses, a program dependence representing a *data flow* from st_1 to st_2, means that the value of the variable defined in statement st_1, denoted by $def(st_1)$, may be used in statement st_2. The labels used by *ADAS* allow to refine this information by recording on every edge the fields of variable $def(st_1)$ which *get defined in* st_1 *and may be used in* st_2. In the second stage, *ADAS* utilizes the range labels to perform an efficient and precise computation of transitive program dependences over the range-labeled *PDG*.

Example 1.1. *Fig. 1(a) shows a program which is comprised only of a sequence of assignments.*[2] *The analysis goal is to compute the transitive dependences on the variable defined in a specified* seed statement, *which in this example is variable* `seed` *defined in statement* l_1. *In this example, ADAS accurately infers that field* $b.g$ *may (only) depend on field* $seed.f$ *and that field* $b.f$ *may (only) depend on field* $seed.g$.

Fig. 1(b) shows the range-labeled PDG of the program. We denote the edge emanating from, e.g., node l_1 *(representing statement* l_1 : `seed:=exp`*) and entering node* l_2 *(representing statement* l_2: `a := seed`*), by* $l_1 \rightarrow l_2$. *The range-label of edge* $l_1 \rightarrow l_2$ *is* $\widehat{seed}$, *which indicates that the whole value of variable* `seed` *is defined in statement* l_1 and *gets used in statement* l_2. *We refer to assignment statements in which whole aggregate structures are assigned as* Big L-value *operations. For example, statements* l_1 *and* l_2 *are Big L-value operations.*

The range-label of the edge $l_2 \rightarrow l_3$ *is* $\widehat{a.f}$, *which indicates that only the* f*-field of* a *is used in statement* l_3. *Note that* $a.f$ *is a* part *of variable* a *whose entire value gets defined in statement* l_2.

The range-label of the edge $l_3 \rightarrow l_5$ *is* $\widehat{i}$, *which indicates that the entire value of variable* i *is defined in statement* l_3 and *gets used in statement* l_5. *(In this example* i *is an integer variable, in general,* i *may be an aggregate structure variable with dozens of fields.)*

ADAS uses the range-labels to accurately and efficiently compute transitive program dependences. For example, the dependence $\langle\widehat{seed}, \widehat{seed}\rangle$ *at* l_2 *is "generated" by the distinguished seed statement. Its propagation over the edge* $l_2 \rightarrow l_3$ *transforms it into* $\langle\widehat{a.f}, \widehat{seed.f}\rangle$, *indicating that the field* $a.f$, *which is used in* l_3, *may (only) depend on field* $seed.f$. *The latter dependence, when propagated over edge* $l_3 \rightarrow l_5$, *gets transformed into dependence* $\langle\widehat{i}, \widehat{seed.f}\rangle$, *indicating that the value of variable* i *at* l_5 *depends on* $seed.f$ *which allows ADAS to infer that* $b.g$ *may (only) depended on* $seed.f$.

We experimentally evaluated the *ADAS* algorithm by comparing its precision and cost with those of the known algorithms in the context of an important software engineering application: helping *ERP* engineers understand the impact of a change in the system's configuration database.[3] We analyzed a variety of industrial *ERP* programs with over a 100,000 line of code in average. Our experiments show that *in practice ADAS* provides the same precision as the most-precise known algorithm for this problem, with average improvement of the running time by 43% and of the memory consumption by 31%. These improvements enabled us to effectively analyze programs that could not have been analyze using the known approaches. (See Section 8.3).

Main Results. The main contributions of this paper can be summarized as follows:

- We present *ADAS*, a novel algorithm for computing program dependences in the presence of large structure variables.
- We prove that *ADAS* is as precise as any algorithm based on the atomization approach in [17].
- We implemented *ADAS* inside *PanayaIA*, an industrial impact analysis tool [6, 24], and successfully applied it to real-life *ERP* programs of over a million lines of code.
- Our extensive experimental evaluation, done in the context of an important engineering application, shows that *in practice ADAS* compares favorably with existing approaches.

We note that while *ADAS* is evaluated in the context of *ERP* systems, the algorithm itself is of generic nature and is of value for analyzing any program which uses aggregate structure variables. (See Section 9).

Outline. The rest of the paper is organized as follows. Section 2 describes related work. Section 3 describes standard notions pertaining to reaching definition analysis and program dependence graphs. Section 4 states the field-sensitive dependence analysis research problem. Section 5 describes our novel range-based reaching definition analysis. Sections 6 and 7 describe, respectively, the intraprocedural and interprocedural aspects of *ADAS*. Section 8 describe our experimental evaluation and Section 9 concludes.

We note that Sections 4-7 are written at a more formal level than the other sections, and can be skipped on first read. Also, due to space reasons, proofs and several technical details have been omitted from the paper and appear in [17].

2. RELATED WORK

Techniques for handling aggregate structures have been suggested in the context of program dependence analysis (see, e.g., [12, 14, 15]) as well as in the context of pointer analysis (see, e.g., [33, 10, 25]). Traditionally, these techniques have mainly focused on analyzing pointer-based manipulation of aggregate structures in recursive data structure and/or by recursive procedures. For example, an efficient analysis which computes data flow dependences in the presence of heap pointers and recursive data structure is presented

[2] We discuss the treatment of control-flow statements, multiple assignments to the same variable, destructive updates, and procedure invocations in the following sections.

[3] *ERP* systems are dominant in the world of business applications. They are typically comprised of hundreds or thousands of programs, where each program may contain over a million lines of code. One of the key reasons for the success of *ERP* systems is their modular design: Every business can add and customize pre-manufactured code modules to meet its particular needs by, what amounts to, loading configuration attributes into in-memory large aggregate structure variables. These variables have an immense effect on the way that the system operates. For additional discussion on *ERP* systems, see Section 8.1.

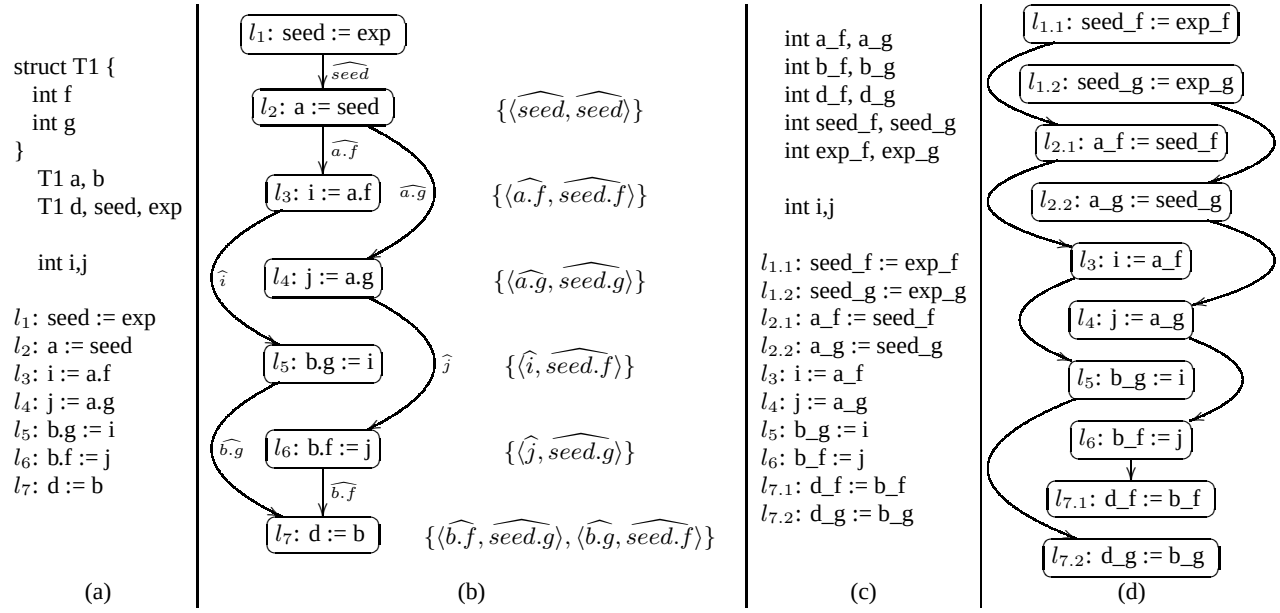

Figure 1: (a) a program manipulating fields, (b) its range-labeled PDG annotated with the dependences computed by the *ADAS* algorithm at every statement, (c) the atomized version of the program, and (d) the PDG of the atomized program

in [12]. Efficient analysis algorithms for handling procedures, especially recursive ones, are presented in works using the *System Dependence Graph* [14, 15].

The two known approaches for handling (large) aggregate structure variables in the area of program dependence analysis are the *whole structure* (*WS*) approach [23, 19, 20], and the *atomization* (*ATOM*) approach [21, 26], which have been described in Section 1. The two approaches represent different tradeoffs between efficiency and precision. The first one is efficient but not field sensitive. The second one has opposite properties.

Example 2.1. *Recall that ADAS successfully infers that at l_7 in Fig. 1 field $b.g$ may (only) depend on field $seed.f$ and that field $b.f$ may (only) depend on field $seed.g$.*

In contrast, any algorithm based on the whole structure approach would fail to infer such accurate information. Specifically, such an algorithm can manage to compute that at l_7 variable `b` *may depend on* `seed`*. However, the field dependence is not known. Thus, the results of a WS-based algorithm would indicate a* false *may dependence of, e.g.,* `b.g` *on* `seed.g`*.*

The atomization approach computes information at the same accuracy level as ADAS. It does so by transforming the original program into the one shown in Fig. 1(c) and then performing standard dependence analysis on the PDG of the transformed program, shown in Fig. 1(d). Note that, in particular, every big L-value operation (see Example 1.1) in which k primitive fields are assigned is transformed into k individual fields assignment operations.

As shown in Example 2.1, both approaches have drawbacks: the whole structure approach can yield superfluous dependences when manipulating structure fields, while atomization approach may be too expensive on programs with large structures (due to the large increase in the total number of variables and statements).

In our case study, different approaches for computing program dependence are evaluated by integrating different analyses in an industrial impact analysis tool. We applied the tool to large-scale *ERP* programs. These programs make extensive use of large aggregate structure variables. We found that, for the programs that we analyzed, using the known approaches is impractical.

We note that [26] suggests an interesting optimization for the atomization approach. The main idea is to use a flow-insensitive analysis that determines which fields need to be atomized. In our experimental evaluation, we have tested the applicability of this suggestion in the context of *ERP* systems. Our evaluation of this suggestion, described in Section 8.4, shows that this optimization is not effective for large-scale *ERP* programs.

3. PRELIMINARIES

In this section, we describe the standard notions of reaching definitions, program dependence graphs, and system dependence graphs.

Reaching definition is an iterative analysis over the control-flow graph (*CFG*) of the program that tracks the possible last definition point for each variable that may arise at execution of a program label. The analysis starts with an empty set of reaching definitions and tracks the effect of statements on the set of reaching definitions. Essentially, the effect of a statement st on a set of reaching definition is to remove some reaching definitions that are no longer true and to generate new ones (see, e.g., [2]).

A statement st *uses* resp. *defines* an *access path* if the latter denotes an l-value which is read resp. written by st. This information is extracted from the result of the reaching definition analysis.

A basic representation of data flow dependency is def-use chains [23]. A def-use edge between statement st_1 and st_2 represents that during execution of st_1 some access-path is assigned which may be (partially) used in the execution of st_2.

The *program dependence graph* (PDG) represents the data and control dependences of a (single) procedure [30]. The nodes of the graph are the statements. There are two types of edges: data and control. Data dependence edges are the def-use edges computed from a reaching definition analysis. Control dependence edges are computed by post-domination on the control flow graph. Fig. 2 contains an example of a PDG (ignore for now the labels on the edges) in which control edges are pictured as dotted edges. We

assume, without lose of generality that all control statements are represented as a guarded statements $p?st'$ where st' is executed only when p is evaluated to true. A control dependency between st_1 and (the necessarily guarded command) st_2 represents that st_1 is a condition over a variable c which is the guard of st_2. Consider the guarded command statement in l_{10} of the example. In the PDG this statement is split into two nodes: one for the predicate p (node $l_{10.1}$) and one for the guarded statement itself $i := q$ (node $l_{10.2}$). The outgoing edge of $l_{10.1}$ in the PDG is a control dependence edge.

A *system dependence graph* (*SDG*) represents the immediate program dependence of a program containing procedures. The *SDG* of a program is comprised the *PDG*s of its procedures. The *PDG*s are connected together by call edges (which represent procedure calls) and by parameter-in and parameter-out edges (which represent parameter passing and return values). The *SDG* representation is often used in inter-procedural dependence analysis [14, 15].

4. PROBLEM DEFINITION

The goal of this work is to find in a precise and efficient manner the *field sensitive transitive dependences* in the presence of aggregate structures with *Big L-value* operations. (Recall that Big L-value operations are assignments to whole (sub)structures.)

Traditionally, given a control flow graph of a program, a node n is *flow dependent* on a node m if there exists a path in the control flow in which the value assigned in m is directly used at n. Similarly, n is *transitively dependent* on m if there exists a path in the control flow graph in which the value assigned in m is indirectly used at n (of course this is an over approximation since not all CFG paths are feasible).

We extend the traditional definition to *field sensitive flow dependent* as follows: an access path, say $x.f$, used at node n is *field sensitive flow dependent* on an access path $s.g$ assigned at node m if there exists a path in the control flow in which the value assigned in m to $s.g$ is directly used as the value of $x.f$ at n. Thus, the traditional flow-dependent relation is between statements while the field sensitive flow dependent relation is between pairs of access-path and statement.

There are several factors that complicate this definition in real life programming languages: pointers and dynamic allocations (e.g., [12]) , Big L-Values, and procedures (e.g., [14]). We define *Field Sensitive Transitive Dependence* as a transitive dependence which handles precisely aggregate structures and Big L-values (without pointers). Notice that Big L-values make the problem more complex because flow dependences between statements are not transitive, i.e., it may be that m is directly flow dependent on n and n is directly flow dependent on p and yet m is not transitively flow dependent on p. For example, statement l_8 in Fig. 2 is flow dependent on l_7 (the use of $d.g1$ is flow dependent on the definition of d), and l_7 is flow dependent on l_6 (the use of c, in particular of $c.g2$, is flow dependent of the definition of $c.g2$), yet their transitive closure yields a spurious dependence of $d.g1$ in l_8 on the definition of t in l_5.

5. RANGE-BASED REPRESENTATIONS

In this section, we describe our range-based reaching definition analysis. We say that a *range* of a variable is a contiguous part of the memory allocated to the variable. A range denoted by $v[i, j]$ represents the memory area of v that starts with the i-th byte and ends with including the j-th byte. A single range of an allocated aggregate can be used to represent all fields that are allocated within this range. For ease of presentation, we use the notion $\widehat{v}$ to denote the whole range of a variable v and $\widehat{v.f}$ to denote the range aligned with the f field of v. A range needs not to be aligned with the fields of an aggregate structure. For example consider the variable sd of type T1 defined in Fig. 2 and assume each field is 4 bytes long. The range $sd[0, 3]$ represents the field $f1$ of sd. We assume the fact that an aggregate structure contains an ordered set of fields and every field has a start and end offset, thus the range $sd[0, 7]$ represents the fields $f1, f2$. We define a meet operation between ranges which provides the maximal range that is in both ranges as follows:

$$v[x,y] \cap v[x',y'] = \begin{cases} v[max(x,x'), min(y,y')] & x \le y' \wedge x' \le y \\ \emptyset & y < x' \vee y' < x \end{cases}$$

Where $\emptyset$ denotes that there is no overlapping between the two ranges. For example, $\widehat{sd.f1} \cap \widehat{sd.f2} = \emptyset$. As we shall see, this operation is used in filtering spurious dependence. Note that the meet operation is defined for ranges of the same variable.

Given the notion of ranges we further simplify the programs and assume that statements are over ranges:

- assignments are in the form $v[x, y] = u[x', y']$,
- expressions evaluation (computation assignments) are in the form $v[x, y] = exp(u_1[x_1, y_1], ..., u_n[x_n, y_n]$,
- guarded statements $p?st'$, and
- goto statements.

We denote by $\mathcal{V}$ the set of variables in the program and by $LABEL$ the set of statements.

5.1 Range-Based Dependency

A dependence $d = \langle v[x, y], seed[x', y'], l, vd\rangle$ implies that the (partial) value of v is transitively dependent on the (partial) value of *seed* which was assigned at line l. The dependence is a *value dependence* if vd is true and a *computational dependence* otherwise.

Value dependence means that the value assigned to the range $seed[x', y']$ is read from range $v[x, y]$. Note that if the variable v and *seed* are aggregate of the same type and the ranges are the same, i.e., $x' = x$ and $y' = y$, and the range spans more than one field, $f_1, ...f_n$ then it implies that $v.f_i$ is value dependent on $seed.f_i$. Note that if aggregate variables have hundreds of fields and if most dependencies are contiguous (as happens when big L-value operations are used) then this representation is much more efficient than a naive representation based on individual fields.

A computational dependence comes either from a *control dependence*, indicating that the value of *seed* was (transitively) used as a control predicate for the (transitively) use of the value of v, or from cases where the value of *seed* was (transitively) used at a computational expression which v is dependent on. In contrast to value dependences, computational dependences *do not* imply that if $seed[x', y']$ includes fields $f_1, ...f_n$ and $v[x, y]$ includes fields $g_1, ...g_m$ then every field $v.g_i$ is dependent on every field $v.f_j$ for $j = 1, ...n$.

5.2 Range Labeled PDG

We adjust the PDG to record fined-grained reaching definition. Using a range based reaching definition analysis, labels are added to def-use edges that represent the range that is actually in use. This enables handling of destructive updates (i.e., partial kill assignments) accurately. For example, in Fig. 2 the two edges from l_2 to l_4 are marked with the range of field $a.f1$ and $a.f3$ because of statement l_3 that kills $a.f2$.

Range based Reaching Definition. We enhance the standard reaching definitions to track reaching definitions of ranges. The domain for the analysis is $\Sigma = RANGE \rightarrow 2^{LABEL}$ where $RANGE = \mathcal{V} \times INT \times INT$. The join operator is a set union and the main transfer function is presented in Table 1 and explained below. The goal is to track aggregate variables as a whole structure and split into ranges only when needed. Handling assignments to

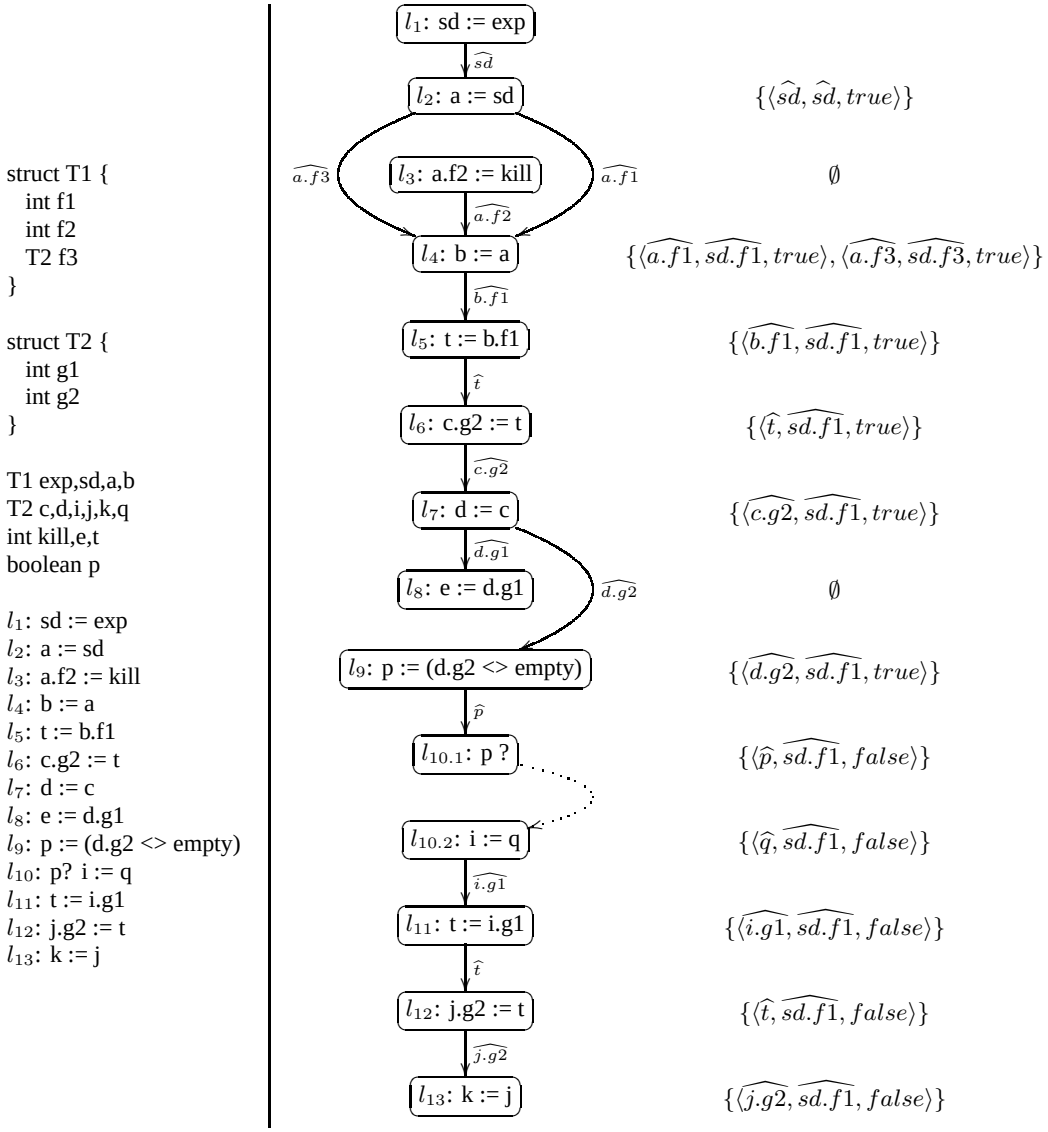

Figure 2: *ADAS* algorithm run example on PDG. The state next to each statement is a result of processing the incoming edges. Guarded commands were split in order to accommodate control dependence edges.

a whole variable v is rather straightforward. Reaching definitions on any range of v are "killed" and a new reaching definition for v is created. For example, the processing of statement l_2 creates the reaching definition $\{a \mapsto \{l_2\}\}$.

Handling assignments to fields requires some finesse as the definition of $v.f$ partially kills a prior definition to v. Assignments to a whole variable define the whole structure and thus do not create a "partial kill". In the example, statement l_2 defines the whole structure a and the statement in l_3 defines only $a.f2$, partially killing the previous definition to a. However, the previous reaching definition for ranges of the aggregate variable that were not killed are still valid and a new reaching definition only to the range that was reassigned is generated. To accommodate this case, previous reaching definition of ranges that overlaps with the assigned range are removed. Instead, up to three new ranges are generated, as defined in Table 1. In our example, the processing of statement l_3 yields the following reaching definition: $a[0, end(f1)] \mapsto \{l_2\}$, $a[start(f2), end(f2)] \mapsto \{l_3\}$, and $a[start(f3), end(a)] \mapsto \{l_2\}$.

Constructing the PDG. The reaching definition analysis computes a set of possible reaching definition points for ranges at each statement. Given this information, the range-labeled PDG is constructed. For each statement, l_2 and for each range in the reaching definition, $v[x, y] \mapsto \{l_1\}$, that is (partially) used at the statement, an edge from l_1 to l_2 is constructed. The label is the meet between the range in the reaching definition and the range used in the statement: Assume the used range in statement l_2 is $v[x', y']$ and the reaching definition is for range $v[x, y]$ then the label is $v[x, y] \cap v[x', y']$. If the meet operation is $\emptyset$ then no edge is added as statement l_2 does not use the reaching definition of $v[x, y]$. Control edges are added to the range-labeled PDG as in the traditional PDG, without labels.

	Statement	Transfer function
(a)	$l : v[x,y] := ...$	$\sigma[t[i,j] \mapsto \sigma(t[i,j])\|t \neq v \vee (t = v \wedge (v[i,j] \cap v[x,y] = \emptyset))] \cup$ $\{v[x,y] \mapsto \{l\}\} \cup$ $\{v[a,x-1] \mapsto l'\|\forall a,b : \sigma(v[a,b]) \mapsto l' \wedge (v[a,b] \cap v[x,y] \neq \emptyset) \wedge a < x\} \cup$ $\{v[y+1,b] \mapsto l'\|\forall a,b : \sigma(v[a,b]) \mapsto l' \wedge (v[a,b] \cap v[x,y] \neq \emptyset) \wedge y < b\}$
(b)	$l : v[0,end(v)] := ...$	$\sigma[t[i,j] \mapsto \sigma(t[i,j])\|t \neq v] \cup \{v[0,end(v)] \mapsto \{l\}\}$

Table 1: (a) The transfer function of the range reaching definition analysis where $\sigma \in \Sigma$ is the current state. (b) For clarity, a simplified version for a whole variable assignment is presented although it is handled as any range assignment.

6. INTRAPROCEDURAL ANALYSIS

The *PDG* contains immediate dependences and a transitive function is needed to compute transitive flow dependences. However, unlike the atomization approach or the whole structure approach where the transitive relation is simple, for the interval-based analysis we need to handle cases where the reaching definition is on a Big L-value. Consider the example of Fig. 2, when computing the transitive flow dependences from statement l_1 and from statement l_2 to l_4, the additional reaching definition of $a.f2$ is taken into account. Clearly, the transitivity of these two reaching definition is valid for $a.f1$ but invalid for $a.f2$. As we shall see, while tracking the transitive dependence, we keep track of the current range that is of interest. In addition, the transitive computation needs to avoid superfluous dependence. For example, consider the dependences between l_6 to l_7 and l_7 to l_8. The analysis needs to infer that there is no dependence between l_6 and l_8. This is further complicated when nested fields are taken into account.

Our Approach. The *ADAS* algorithm is a chaotic iteration algorithm over the range-labeled PDG. The domain is the set of all sets of dependences

$$2^{\langle RANGE \times RANGE \times LABLES \times BOOL \rangle}$$

where $RANGE = \mathcal{V} \times INT \times INT$ is the set of all ranges. A tuple $\langle v[x,y], s[x',y'], l, vd \rangle$ corresponds to a dependence as defined in Section 5.1: it implies that range $[x,y]$ of variable v is transitively dependent on the values assigned to range $[x',y']$ of the seed variable s at line l. Furthermore, the dependence is a value dependence if vd is true and a computational dependence otherwise. A dependence on $v[x,y]$ computed in a state for a statement l' implies that the statement l' uses a range of the variable v, say $v[x_u, y_u]$. However the dependent range can be a subrange of the used range, i.e., $x_u \leq x \leq y \leq y_u$. In the rest of the paper, we assume, without lose of generality, that the algorithm tracks a single seed variable assigned at a single seed statement and omit the label member of the tuple.

The join operator is the set union operator. The algorithm starts with a seed statement and a dependency on the seed range $\langle sd[x,y], sd[x,y], true \rangle$ and computes the transitive dependency of the seed. The analysis computes for each statement, st, a set of field sensitive dependencies on the used ranges of st. The propagation of dependencies takes into account the source and target statements and the type and label of the edge. Table 2 formalizes the transfer functions. A data flow edge between st_1 and st_2 is handled in two steps (Step 1 and 2 Table 2) as follows:

- Step 1 computes the dependence on the defined range (left-hand side) of statement st_1. At this step, assignments that are "copy" statements are treated differently than computational statements, as the first preserve the data flow and the second does not. Conditional statements do not change the dependence. At this step we take into account cases where the dependence is on a subpart of the used range and adjust the dependence to the subpart of the defined range.
- Step 2 filters and adjusts the computed dependence from the first step according to the edge by taking into account the label on the edge: A dependence $\langle v[x,y], \mathit{seed}[x',y'], vd \rangle$ is filtered if there is no overlap between the dependent range and the label on the edge. If it is not filtered, then the dependent range is refined according to the label. In addition, if the dependence processed is a data-flow one (vd is true), the dependence range of the seed is refined as well.

Control edges do not have a label and therefore never filter dependences. Because control edges always come from a condition expression, the analysis yields a computational dependences for each used range in the target statement to reflect that the dependency is not a data flow one.

The example in Fig. 2 shows the computed dependencies on the variable sd assigned at statement l_1. We now further explain the key issues in the analysis.

Handling Data Edges on Data Dependences. The first step of handling data edges is generating the dependence on the defined range of the source statement. For example, the first step in the processing of dependence $\langle \widehat{sd}, \widehat{sd}, true \rangle$ on the edge labeled $\widehat{a.f1}$ form l_2 to l_4 treats statement l_2 as an assignment $a[0, end(a)] := sd[0, end(sd)]$ and generates the same dependence on the variable a, $\langle \widehat{a}, \widehat{sd}, true \rangle$. The second step refines the dependence according to the label. In this case the label $v[x_l, y_l]$ is $a[0,3]$ and the dependent range $v[x,y]$ is $a[0, end(a)]$. Because the dependence is a data flow one (vd=true) and the meet between $a[0, end(a)]$ and $a[0,3]$ is not $\emptyset$ then the processing is according to the second case in Step 2 in Table 2. The computation of the dependent range is evaluated as $a[0,3] \cap a[0, end(a)] = a[0,3]$. The seed range is also refined according to $sd[x_s, y_s - (y - y_l)] = sd[0,3] = \widehat{sd.f1}$. Thus, the second step yields the dependence $\langle \widehat{a.f1}, \widehat{sd.f1}, true \rangle$.

In cases where the defined range is a subrange of a variable, the first step performs an adjustment of the dependent range. For example, consider the processing of dependence $\langle \widehat{t}, \widehat{sd.f1}, true \rangle$ on the edge from l_6 to l_7. The statement in l_6 is treated as an assignment $c[4,7] := t[0,3]$ and the processed dependence in explicit range format is $\langle t[0,3], sd[0,3], true \rangle$. According to the first case in Step 1 in Table 2, the used range $u[x_u, y_u]$ is $t[0,3]$, the defined range $v[x_d, y_d]$ is $c[4,7]$, and the dependent range $t[x,y]$ is $t[0,3]$. The generated dependence is $\langle c[4,7], sd[0,3], true \rangle$ which is represented as $\langle \widehat{c.g2}, \widehat{sd.f1}, true \rangle$.

Handling Partial Kill. Consider transition l_2 to l_4 which includes the processing of two edges due to the destructive update in l_3. For each outgoing edge from l_2 new dependences are computed for l_4 as described above in the processing of data edges. The result is

		Transfer Functions	
		Case	New Dependence
Data Edge	Step 1	st_1 in the form $v[x_d, y_d] = u[x_u, y_u]$	$\langle v[x_d + (x - x_u), y_d - (y_u - y)], \mathit{seed}[x_s, y_s], vd\rangle$
		st_1 in the form $v[x_d, y_d] = exp(u_1[x_1, y_1], ..., u_n[x_n, y_n])$	$\langle v[x_d, y_d], \mathit{seed}[x_s, y_s], false\rangle$
	Step 2	$v[x, y] \cap v[x_l, y_l] = \emptyset$	$\emptyset$
		$v[x, y] \cap v[x_l, y_l] \neq \emptyset \wedge vd = true$	$\langle v[x, y] \cap v[x_l, y_l], \mathit{seed}[x', y'], true\rangle$ where $x' = \begin{cases} x_s & x \geq x_l \\ x_s + (x_l - x) & x < x_l \end{cases}$ $y' = \begin{cases} y_s & y \leq y_l \\ y_s - (y - y_l) & y > y_l \end{cases}$
		$v[x, y] \cap v[x_l, y_l] \neq \emptyset \wedge vd = false$	$\langle v[x, y] \cap v[x_l, y_l], \mathit{seed}[x_s, y_s], vd\rangle$
Control Edge			$\{\langle u_i[x_i, y_i], \mathit{seed}[x_s, y_s], false\rangle \mid i = 1, .., n\}$

Table 2: Transfer functions of the *ADAS* algorithm for a dependence $d = \langle t[x, y], \mathit{seed}[x_s, y_s], vd\rangle$ for an edge between st_1 and st_2. For data flow edge, we assume that the edge is labeled $v[x_l, y_l]$, that $v[x_d, y_d]$ is the defined range of statement st_1, and that $v[x_u, y_u]$ is the used range in statement st_2. Note that step 2 processes the (intermediate) dependence computed by the previous step and specifically the handled dependency in on $v[x, y]$. Note that in the first case of step 1, $x_u \leq x \leq y \leq y_u$ always hold. Note that in this case it holds that $y_u - x_u = y_d - x_d$ and thus $y_d - (y_u - y) = x_d + (x - x_u) + (y - x)$. Also note that in the second case of step 2, if $v[x'', y''] = v[x, y] \cap v[x_l, y_l]$ then $x' = x_s + x'' - x$ and $y' = x_s + (y'' - x'')$. For control flow edges, we assume that a set of ranges $u_1[x_1, y_1], ..., u_n[x_n, y_n]$ are used in statement st_2.

joined to a single set of dependences. Due to the labels on the edges that represent the exact used range of a reaching definition, the fact that $a.f2$ is partial killed is accommodates into the results of the analysis without specific handling.

Filtering Unused Intervals. The processing of the edge from l_4 to l_5 on the dependence $\langle \widehat{a.f3}, \widehat{sd.f3}, true\rangle$ leads to a "dead end" as the dependent range after the first step is $\widehat{b.f3}$ and the use label is $\widehat{b.f1}$. Clearly, these two ranges are disjoint, $\widehat{b.f3} \cap \widehat{b.f1} = \emptyset$ and thus the second step filters out this dependence, according to the first case in Step 2 in Table 2.

Handling Big L-value Assignments. The first step of processing the edge from l_6 to l_7 yields a dependency on $\widehat{c.g2}$. This data enables us to filter out the spurious dependency when processing the edge from the Big L-value assignment at l_7 to l_8, despite the obvious immediate dependency between l_7 and l_8.

Handling Computational Assignments. The assignment to p in l_9 is of a computational form. In this case, according to the second case in Step 1, the whole defined range is computational dependent on the seed, i.e., the generated dependence is with $vd = false$. Additional propagation of this dependence are all computational ones.

Handling non Value Dependences. Processing of the edge labeled $\widehat{i.g1}$ between $l_{10.2}$ and l_{11} is on a computational dependence. Step 1 yields a dependence on the whole variable i. However, according to the third case in step 2, since the dependence is computational, only the dependent range is refined according to the label, yielding a dependency on $\widehat{i.g1}$.

7. INTERPROCEDURAL ANALYSIS

ADAS summarizes procedures as a dependence relation between the values of the formal arguments at the entry to the procedure (*formals in*) and the value of the return arguments at the procedure exit (*formals out*). Global variables are treated as additional formal arguments and return arguments. Thus the relation between formals in and formals out computed for every procedure also accounts for the procedure's effect on global variables.

Informally, the interprocedural dependence analysis can be described as a two phase process: The first phase summarizes every procedure by computing a transitive dependence relation between a procedure's formals-in and formals-out. The second phase computes the field sensitive dependences across procedures using procedure's summaries to account for the effect of procedure calls. Recursive procedures are handled by repeating the two steps until a fixed point is reached.

Technically, our implementation is based on existing summary algorithm [14]. The first step computes intra-procedural dependences between formal-in to formal-out in the form

$$d = \langle [f_{out}[x, y], f_{in}[x', y'], st_1, vd\rangle$$

where st_1 defines the variable f_{in}. Given a dependence at a return statement, a *dependence statement*, which are used to represent procedures' summaries, $ds(d)$ is constructed as follows:

$$ds(d) = \begin{cases} f_{out}[x, y] := f_{in}[x', y'] & \text{if } vd = true \\ f_{out}[x, y] := exp(f_{in}[x', y']) & \text{if } vd = false \end{cases}$$

The second step is an iterative process in which dependence statements of callees are used at call sites to compute the dependence statements of the caller. This is processed until no more dependence statements are computed.

We note that a similar modification can be applied on a more efficient summary algorithm [15], thus improving the running time of the inter-procedural *field sensitive* dependences analysis. We consider such a change as part of future work.

8. EXPERIMENTAL EVALUATION

This section presents a case study in which we evaluated *ADAS* in the context of an important software engineering application:

customization-change impact analysis for configuration management of *ERP* systems. Our empirical study was performed by integrating *ADAS* in *PanayaIA* [6], an industrial-strength analysis tool for computing the impact of customization changes for *ERP* systems made by *SAP* [1], a leading *ERP* software vendor and written in *ABAP*, a proprietary language developed by *SAP* since the 1970s [16]. We begin by providing a short background regarding analyzing *ERP* systems, and then we describe our evaluation methodology, and present our experimental evaluation of *ADAS*.

We note that while *ADAS* is evaluated in the context of *ERP* systems, the algorithm itself is of generic nature and is of value for analyzing any program which uses large aggregate structures. (See Section 9).

8.1 Analyzing ERP Systems

An *ERP* system integrates information technology with core business processes and is intended to manage all the information needs and business procedures of an (often geographically scattered) organization. (See, e.g., [31]). An *ERP* system is typically comprised of hundreds or thousands of programs. Each program represents a business process (e.g., human resources, warehouse management, project management, financials, client relation management, etc.), and may contain over a million lines of code.

ERP systems have become dominant in today's world of business applications. One of the key reasons for the popularity of ERP systems in the world of business is their modular design: Every business can add and customize modules to meet its particular needs. On the flip side, the flexibility of ERP systems makes their analysis a very challenging problem, as we discuss below.

Technically, the customization of an ERP system is done by storing *configuration* tables at the system's database and reading the configuration attributes at runtime into in-memory aggregates structures, which may be of significant size. (It is not unusual that a single structure contains hundreds of fields). The aggregate structures storing the configuration attributes have an immense effect on the way that the system operates. In practice, whenever a professional ERP engineer wants to modify the system behavior, she updates a proper configuration attribute in the system's database.

A common programming practice in *ERP* systems is to fetch attributes pertaining to a whole structure from the database. These attributes can affect the execution of multiple programs. It is the task of the analysis to determine which of the fields are needed for the currently analyzed program. Thus, an efficient and precise way for handling large aggregate structure variables is required to obtain a useful customization-changes impact analysis.

Due to the large code base of SAP and the sharing of libraries between programs, we follow a modular approach. First, each compilation-unit is analyzed separately without prior knowledge about different invocation of the compilation unit and without taking into account the code of external libraries invocations. Next, each program is analyzed by inter-procedural analyses of the compilation units that are part of this program. See [6] for more elaboration on the implementation of *PanayaIA*.

8.2 Evaluation Methodology

The experiments presented in this section are on a selected benchmark of 12 programs which vary in size, complexity, and associated *SAP* components. Time is measured in minutes. Experiments performed on a computer grid comprised of five Intel servers. Each server has two Dual Intel Xeon 5355 processors and 16GB memory running Windows XP operating system (64-bit) with Java 5.0. In our experiments we study the differences in the quality of the results and the cost of the analysis of three transitive dependences algorithms: 1. *ATOM*: An algorithm based on the atomization approach, 2. *WS*: an algorithm based on the whole structure approach, and 3. *ADAS*.

Note that all three algorithms compute the transitive inter-procedural dependences over the System Dependence Graph (*SDG*) using procedural summary dependences. We focus on two comparison criteria : the accuracy of the dependences in terms of number of false-positive (spurious dependences) and the scalability of the analysis approach in terms of time and memory. The results of the atomization approach (ATOM) served as the most precise results and we compared the other two algorithms' results to it.

Program	KLOC	ATOM	WS		*ADAS*	
		# Pairs	# Pairs	FP	# Pairs	FP
SAPMF02B	3	4	5	25%	4	0%
SAPF110V	8	3	18	500%	3	0%
SAPMV10A	23	5	12	140%	5	0%
SAPMA01B	42	40	120	200%	40	0%
SAPMM07R	65	36	72	100%	36	0%
SDBILLDL	115	23	46	100%	22	-4%
SAPLAMDP	178	168	308	83%	175	4%
SAPMV60A	333	275	492	79%	273	-1%
SAPMV50A	212	374	697	86%	391	4%
SAPLAIST	211	138	264	91%	144	4%
SAPMF02D	226	837	1054	26%	837	0%
SAPMM06B	419	-	560	-	456	-
Average	129	173	281	62%	175	1%

Table 3: The number of customization (table, column) pairs that impact the benchmark program and FP, the amount of false positives. The number of lines of code (LOCs) is in thousands. Negative percentages are due to implementation issues of the *PanayaIA* tool, which are unrelated to *ADAS*. (See Section 8.3).

8.3 Experimental Results

Table 3 compares the accuracy of the dependences for the three algorithms. For each program the table contains the number of customization (table, column) pairs that have an impact on the program according to each algorithm. Column 2 lists the size of the program, in thousands-of-lines of code. Column 3 contains the *ATOM* algorithm results which also serves as a reference for the other two algorithms. Columns 4-5 contains the WS algorithm results and the percentage of the false positives of the results compered to the base results. The results of the *ADAS* algorithm are specified in columns 6-7.

For the *WS* algorithm, the percentage of the false positives is relatively high—an average of 62%, and a peak of 500% for a specific program. For the *ADAS* algorithm, there is low false positive percentage of only 1% on average. Theoretically, *ADAS* has the same precision as the *WS* algorithm. Practically, there is a difference in the precision because the implementation of the *PanayaIA* tool uses some heuristics through the analysis that consider among other things the size of the program, which is quite different between the original program and the atomized program.[4]

Facing the high number of *ERP* programs that need to be analyzed and the significant size of the programs, the ability to scale the algorithms has been a key element in the success of the tool.

Table 4 lists a comparison of the three algorithms according to the execution time of the analysis for each program. As before, the

[4]In some cases the atomized program has 4 times more statements then the original program.

Program	Atom	WS		*ADAS*	
	Time	Time	Imp.	Time	Imp.
SAPMF02B	0.59	0.57	-3%	0.76	30%
SAPF110V	1.43	2.24	56%	1.26	-12%
SAPMV10A	3.99	4.63	16%	3.23	-19%
SAPMA01B	11.20	12.68	13%	10.65	-5%
SAPMM07R	10.11	9.27	-8%	8.24	-18%
SDBILLDL	24.97	20.88	-16%	21.41	-14%
SAPLAMDP	40.86	30.79	-25%	35.81	-12%
SAPMV60A	359.31	99.23	-72%	125.25	-65%
SAPMV50A	43.42	32.53	-25%	31.56	-27%
SAPLAIST	59.13	41.39	-30%	51.08	-14%
SAPMF02D	123.00	56.64	-54%	96.71	-21%
SAPMM06B	-	110.10	-	151.74	-
Average	61.64	28.26	-54%	35.09	-43%

Table 4: Elapsed time in minutes and the performance improvement (Imp.). Negative percentages are due to implementation issues of the *PanayaIA* tool, which are unrelated to *ADAS*. (See Section 8.3).

Prog	ATOM	WS		*ADAS*	
	Mem	Mem	Imp.	Mem	Imp.
SAPMF02B	405	397	-2%	401	-1%
SAPF110V	414	397	-4%	401	-3%
SAPMV10A	644	494	-23%	522	-19%
SAPMA01B	1,284	867	-32%	985	-23%
SAPMM07R	1,399	1,069	-24%	998	-29%
SDBILLDL	2,524	1,574	-38%	1,747	-31%
SAPLAMDP	2,480	1,770	-29%	1,884	-24%
SAPMV60A	8,023	5,044	-37%	5,058	-37%
SAPMV50A	5,538	3,305	-40%	3,556	-36%
SAPLAIST	3,051	1,993	-35%	2,313	-24%
SAPMF02D	5,393	3,345	-38%	3,532	-35%
SAPMM06B	-	4,831	-	5,755	-
Average	2,596	1,688	-35%	1,783	-31%

Table 5: The average memory usage (in MB) and the performance improvement. Negative percentages are due to implementation issues of the *PanayaIA* tool, which are unrelated to *ADAS*. (See Section 8.3).

ATOM algorithm results serve as the base for the WS and *ADAS* algorithms. We can see that the *WS* algorithm has a large decrease of the execution time of 54% on average with a peak of 72% on a specific program. Interestingly, *ADAS* is not far behind with a decrease of 43% on average with a peak of 65%. This is without taking into consideration the time saved for the analysis done prior to the actual computation of the transitive dependences (building the CFG, constant propagation, etc). Although the *WS* algorithm performs fewer actions on each step than the *ADAS* algorithm (as it does not need to calculate and compare field sensitive properties), due to its over approximated nature, it performs more steps than the *ADAS* algorithm. Overall, the improvements of both *ADAS* and *WS* algorithms in execution time is quite high, around the 50%. Whereas the *WS* algorithm includes a high portion of false-positives 62%, the *ADAS* algorithm achieves this performance boost without hurting the accuracy of the dependences computation.

Table 5 lists the memory usage of the three algorithms for each benchmark program. As expected the less precise WS algorithm (columns 3-4) has a 35% lower memory consumption than the base result of the *ATOM* algorithm (column 2). Columns 5-6 contains the results of the *ADAS* algorithm. We can see that the *ADAS* algorithm was able to decrease the memory consumption by 31% on average. *ADAS* successfully analyzed the largest program with a memory usage of 5.7GB while the *ATOM* algorithm has failed to analyze the program within the available heap size (15GB).

8.4 Evaluating a More Efficient Atomization

We also studied if a more efficient atomization algorithm in the style of [26], which have shown great improvement for Cobol programs, can reduce the performance penalty of the ATOM algorithm in our case of ERP programs. The main idea of the optimization suggested in [26] is to perform the atomization only for those structure fields that actually used/refered in the program explicitly and not according to the type information. Although, in worst case this approach may result in full atomization, in some cases the atomization can involve only a small portion of the structure fields. In order to do so, a flow-insensitive algorithm is used to group structure fields to equivalence classes. Two structure fields are in the same equivalence class if there is a direct or indirect assignment between them. These fields are considered from the same type (for type inference) and are disassembled (for atomization) if at least one of the fields is explicitly used.

For *ERP* programs, we determine that due to the algorithm's flow-insensitive nature and the vast use of large structure variables, the resulting equivalence classes are large and the number of unused structure fields is quite low, which induce atomization of large portion of the structure fields (i.e the optimization unable to improve performance): Table 6 contains the total number of structure fields (column 2) and the number of unused structure fields (column 3) and its ratio (column 4) for each program in our benchmark. We can see that on average approximately 20% of the fields are unused, i.e. the atomization needs to include most of the structure field (over 80%). Therefore, we conclude that (in the context of our case study) this approach does not provide a good optimization to the full atomization approach.

9. CONCLUSIONS AND FURTHER WORK

In this paper, we present *ADAS*, a static analysis algorithm which conservatively computes transitive field-sensitive program dependences. The algorithm provides a specialized efficient and precise handling of programs with large aggregate structure variables, e.g., large-scale *ERP* programs, by performing an interprocedural analysis over a novel form of representing reaching definitions using ranges.

We provide a detailed comparison between *ADAS* and the two known approaches for this problem: the *whole structure* (*WS*) approach and the *atomization* (*ATOM*) approach. We show that, theoretically, our algorithm is more precise than the *WS* approach and as precise as the *ATOM* algorithm, the most-precise known solution for this problem [17].

Practically, our empirical results show that *ADAS* has comparable performance in time and memory of the (less precise) *WS* algorithm while providing the same quality of results as the *ATOM* algorithm. Indeed, using *ADAS* we successfully analyzed large *ERP* programs that we could not analyze with the *ATOM* algorithm using the (rather vast) computational resources that we have. In a way, we can say that *ADAS* enjoys the benefits of both known approaches and does not suffers from their weaknesses.

Program	# Fields	# Unused Fields	% of unused
SAPMF02B	555	288	52%
SAPF110V	2,208	228	10%
SAPMV10A	9,029	1,024	11%
SAPMA01B	29,986	17,398	58%
SAPMM07R	22,600	4,491	20%
SDBILLDL	53,925	11,380	21%
SAPLAMDP	42,445	6,418	15%
SAPMV60A	69,399	16,711	24%
SAPMV50A	69,672	11,909	17%
SAPLAIST	59,936	21,151	35%
SAPMF02D	32,706	2,406	7%
SAPMM06B	82,490	11,355	14%
Average	39,579	8,730	22%

Table 6: Results of a study on the number and percentage of unused structure fields.

ADAS helps solving an important software engineering problem: automatically inferring the impact of customization-changes on *ERP* systems. It shows that it is possible to overcome a major challenge for program analysis tools: providing a scalable and precise enough analysis for industrial large-scale systems. We attribute the success of *ADAS* to the judicious choice of the abstract domain which it employs. A choice which was guided by the characteristic of the class of programs of interest and the opportunities for an efficient implementation.

In this work, we concentrate on analyzing programs that manipulate large aggregate structure variables, and provided an efficient and precise solution for the problem of computing field-sensitive program dependences for this class of programs. Further work can explore the possibilities that lie in combining our approach with dependence analyses that specialized in other programming features. Specifically, pointer manipulation of recursive data structure: A quite precise computation of immediate dependences for programs manipulating pointers and recursive data structure is presented in [12]. Unfortunately, their approach may compute false transitive dependences when facing Big L-value assignments (assignment operations that define the value of whole structures). We hope that combining our techniques with theirs will allow to accurately compute transitive program dependences for pointer programs which use Big L-value assignments.

Another area that can be addressed is the computation of dependences between array elements. We can think of an array as a large structure variables where indices are used to name subcomponents instead of fields. The challenging aspect of this extension is that array-copy operations are not as easily detectable from the syntax of the program: Instead of using Big L-values operations to perform a copy the contents aggregate structure locations, as commonly done in structure-manipulating programs, array-copy operations are usually implemented using a loop. Thus, the analysis would need to track the index, which we believe can be done by integrating techniques developed for array-manipulating programs [9, 5].

10. REFERENCES

[1] http://www.sap.com/.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, 2006.

[3] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Symp. on Principles of Prog. Lang.*, 1993.

[4] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Conf. on Soft. Maintenance*, 1992.

[5] I. Dilling, T. Dilling, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *European Symposium on Programming*, 2010.

[6] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, and D. Weiss. Customization change impact analysis for ERP professionals via program slicing. In *Int. Symp. on Soft. Testing and Analysis*, 2008.

[7] K. Gallagher. *Using program slicing in software maintenance*. PhD thesis, Comp. Sci. Dept., Univ. of Maryland, Baltimore Campus, 1990. Tech. Rep. CS-90-05.

[8] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *IEEE Trans. on Soft. Eng.*, 1991.

[9] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Symp. on Principles of Prog. Lang.* ACM, 2005.

[10] M. Hind. Pointer analysis: HavenÃt we solved this problem yet. In *Work. on Prog. Analysis for Soft. Tools and Eng.*, 2001.

[11] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Conf. on Prog. Lang. Design and Impl.*, 1990.

[12] S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence analysis for pointer variables. In *PLDI*, 1989.

[13] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *Trans. on Prog. Lang. and Syst.*, 1989.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 1990.

[15] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Symp. on the Foundations of Soft. Eng.*, 1994.

[16] H. Keller and S. Kruger. *ABAP Objects: Introduction to Programming SAP Applications*. Addison-Wesley, 2002.

[17] S. Litvak. Field sensitive program dependeces for large scale systems, 2009. http://www.cs.tau.ac.il/∼shaylitv/.

[18] J. Lyle and M. Weiser. Experiments on slicing-based debugging tools. In *Conf. on Empirical Studies of Programming*, June 1986.

[19] J. R. Lyle. *Evaluating variations on program slicing for debugging*. PhD thesis, University of Maryland, 1984.

[20] S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 8.12. Morgan Kaufmann, 1997.

[21] S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 12.2. Morgan Kaufmann, 1997.

[22] J. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Commun. ACM*, 1994.

[23] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Symp. on Practical Soft. Development Environments*, 1984.

[24] Panaya Inc. http://www.panayainc.com. 2009.

[25] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for c. In *Work. on Prog. Analysis for Soft. Tools and Eng.*, 2004.

[26] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Symp. on Principles of Prog. Lang.*, 1999.

[27] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Principles of Prog. Lang.*, pages 49–61. ACM, 1995.

[28] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167:131–170, 1996.

[29] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Conf. on Prog. Lang. Design and Impl.*, pages 112–122. ACM, 2007.

[30] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[31] B. Wagner and E. Monk. *Enterprise Resource Planning*. Course Technology Press, Boston, MA, United States, 2008.

[32] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, Ann Arbor, MI, USA, 1979.

[33] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Conf. on Prog. Lang. Design and Impl.*, 1999.