Customization Change Impact Analysis for ERP Professionals via Program Slicing

Nurit Dor
Panaya Inc.
nurit@panayainc.com

Tal Lev-Ami*
Tel-Aviv University
tla@post.tau.ac.il

Shay Litvak
Panaya Inc.
shay@panayainc.com

Mooly Sagiv*
Tel-Aviv University
msagiv@post.tau.ac.il

Dror Weiss
Panaya Inc.
dror@panayainc.com

ABSTRACT

We describe a new tool that automatically identifies impact of customization changes, i.e., how changes affect software behavior. As opposed to existing static analysis tools that aim at aiding programmers or improve performance, our tool is designed for end-users without prior knowledge in programming. We utilize state-of-the-art static analysis algorithms for the programs within an Enterprise Resource Planning system (ERP). Key challenges in analyzing real world ERP programs are their significant size and the interdependency between programs. In particular, we describe and compare three customization change impact analyses for real-world programs, and a balancing algorithm built upon the three independent analyses. This paper presents PanayaImpactAnalysis (PanayaIA), a web on-demand tool, providing ERP professionals a clear view of the impact of a customization change on the system. In addition we report empirical results of PanayaIA when used by end-users on an ERP system of tens of millions LOCs.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; D.3.1 [**Formal Definitions and Theory**]: Semantics

General Terms

Algorithms, Experimentation, Management

Keywords

Customization Change Impact Analysis

1. INTRODUCTION

Enterprise Resource Planning systems (ERPs) provide comprehensive sets of software tools for all the business processes of an organization, regardless of the organization's business or character. Businesses, non-profit organizations, non-governmental organizations, governments, and other large entities utilize ERP systems. Today, only few organizations choose to implement management software in-house. Instead, ERP vendor packages are integrated into the organization. The ERP system is cross-functional and enterprise wide. All functional departments involved in operations or production are integrated into one system. In addition to manufacturing, warehousing, logistics, and information technology, also accounting, human resources, marketing, and strategic management may be included. ERP systems are composed of a large code base shared between all customers of the system together with a rich set of customization options. There is clear separation between the developers of the ERP system and the ERP professionals responsible for the customization of the system for a specific organization.

Clearly, integrated systems impose challenges: changes made in one business area can affect several business processes in different business practices. Indeed, ERP systems have thousands of customization properties. In SAP R/3 4.6C, a leading ERP vendor, has approximately 30,000 dialogs for customization of the system, modifying approximately 9,000 customization tables. The customizations are configured in order to control the behavior of the system. For example, a specific SAP installation can customize not only the list of warehouses but also the types of transfers into and out of each warehouse. ERP professionals have great power to customize the application for different environments. With that power comes considerable complexity.

ERP professionals, who usually master a limited number of applications, are responsible for applying customization changes to the system. However, in many cases it is difficult to predict the effect of a change on the whole system. Current techniques for aiding ERP professionals in understanding the potential impact of a change are mostly organizational, such as expert consulting and change steering committees. ERP professionals making a change need to work together with their peers in order to ensure that the proper effect is correctly achieved. Sometimes, it is not clear which modules are affected, leading to complications and problems in production. In a survey held among ERP professionals, 72% answered that customization problems are a serious technical problem, which ranked as second highest technical problem, after integration with existing systems [25]. Therefore, it is not surprising that the capital spent on maintenance tasks is very high.

Code change impact analysis [1, 22, 19] determines the effect of a source code modification and aims at supporting programmers in focused testing and debugging. This paper describes new techniques for *customization change impact analysis* that aim at aiding end-users (in our case, ERP professionals) in understanding and testing the behavior of the software after a customization change.

*This work was conducted while visiting Panaya Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-59593-904-3/08/07 ...$5.00.

Contrary to code change impact analysis, in a customization change impact analysis the source code has not been modified. We present PanayaIA, a static analysis tool that provides customization change impact analysis for ERP professionals. For a given customization change, PanayaIA lists the possible affected parts of the ERP system with a detailed description of the effect, in the ERP professional's terminology. Although PanayaIA applies state-of-the-art program analysis techniques, the input and output of the tool does not involve the analyzed code. The techniques presented here are applicable to many ERP systems and other customization-based software. We present our tool for SAP R/3 4.6C.

The contributions of this paper can be summarized as follows:

- A definition of the customization change impact problem, a real world problem that can lead to failures and unpredictable behavior on mission critical systems. Section 2 formalizes the change impact analysis problem.

- A description of the implementation of PanayaIA, a tool that provides customization change impact information. The tool is a web-based on-demand service, a nonstandard architecture in the domain of program analysis, which is feasible due to the ERP nature in which the code is shared among all installations. The user interface of the tool, implemented for non programmers, requires attention in order to be clear, coherent and self explanatory. Section 3 describes the tool with an elaboration on the architecture and user interface of the tool.

- Utilization of static analysis techniques, which have been greatly improved over the last decades, for solving this problem and aiding ERP professionals in one of their most critical and risky tasks. We presents three *basic* algorithms for solving the customization change impact problem providing approximation to the customization change impact. The designed algorithms tackle specific challenges emerging from analyzing real ERP code and vary in their precision and cost. In addition, we present a *balancing algorithm*, which aims at gathering the "best of all worlds" by combining the three basic algorithms. Section 4 details the algorithms that provide an abstraction to the change impact.

- A comparison of the different algorithms in terms of results and cost on a real, large code base and a measurement of the precision by studying end-users' experience with the results. The implementation of the tool is presented in Section 5. The study of the effectiveness of the algorithms and the tool is presented in Section 6.

2. CUSTOMIZATION CHANGE IMPACT

One of the key advantages of an ERP system and in particular SAP is its flexibility, which is achieved by customization tables that drive the program functionality. Customization tables can control the code functionality, for example, whether a report can be saved or the conditions taken into account in calculation of a price. Therefore, new and changed business requirements can be quickly implemented and tested in the system. Within SAP, there are thousands of database tables that may be used to control the behavior of the application. A customization can dramatically affect the output of the program in terms of the user dialog invoked, the database manipulation performed, the output to the screen (e.g., error messages or reports), the invocation of external programs or interfaces, or any other output to any device, denoted as output identifers of the program.

2.1 Defining Customization Change Impact

In this section we define the notion of customization change impact and consider abstractions that allow static analysis tools to approximate this notion in realistic settings. We define two levels of customization change impact results: *program-level impact*, which defines which programs are affected from a change, and *detailed impact*, which provides information regarding which parts of a program is affected by the change. The program-level impact is, as we shall see, easier to compute than the detailed impact. The detailed impact aims at providing details that are understandable to the ERP professional. Customization in ERP systems is performed by modifying designated database tables. Formally, a *configuration specification* is a set of relation symbols (designating the customization tables) and a *configuration* is the corresponding set of relations. An *attribute* is a position within a tuple in a relation (corresponding to a column in a customization table). Thus, customization amounts to changes of attributes in the configuration (the current setting of the customization options).

EXAMPLE 1. *Figure 1 contains an example of a customization based program, referred to as* `Report#3`. *It is written in C-like langauge with SQL statements. This program represents a customizable report. One of the customizable behaviors of the report is whether it can be printed. If printing is enabled and the user chooses to print, then the document is printed to the user's default printer (if customized). If no default printer is set, a dialog in which the user needs to choose a printer is displayed. For this program, there are two customization tables:* `ReportConfig` *and* `UserConfig`. *The first indicates for each report whether printing is allowed and the second table indicates for each user her default printer (among other things). A configuration to this program includes the set of entries for the customization tables, as displayed in the first row, second column in Table 1. The* `Printable` *attribute of the first tuple in the* `ReportConfig` *relation is false.*

An ERP program's input can be seen as a pair $\langle C, I\rangle$ where C is the given configuration, known before execution starts and I is the user specified input. Formally, a program $P : \Omega \times \Gamma \rightarrow O$ where: Ω is the set of all configuration; Γ is the set of possible user inputs to P, O is the set of all outputs. To be able to discuss specific outputs of the program, we consider the output of the program as a map of a set OID of output identifiers to their values. Therefore O is a set of all maps from OID the set of possible output identifiers and their corresponding values. We define a partial evaluation, $[\![P]\!]$ of a program P as a function from configurations to specialized programs [9].[1] The specialized program takes as input only the user specific input and not the configuration. The partial evaluator guarantees that the generated program is equivalent to the original program with the same configuration. Thus, for each program P and for each configuration $C \in \Omega$, $P(C, I) = ([\![P]\!](C))(I)$ for all input $I \in \Gamma$.

EXAMPLE 2. *For the* `Report#3` *program the OID set includes the following: (i)* o_{error} *of Boolean type which holds when the error message is reported, (ii)* o_{select} *of Boolean type which holds when the printer select dialog is shown, and (iii)* $o_{printed}$ *of String type which holds the name of the printer the report was printed to. Table 1 shows the partial evaluation of the program* `Report#3`. *The first row represents the program* $[\![\texttt{Report\#3}]\!](C_{old})$ *when printing is disabled. The program is specialized such that on any request to print an error message is displayed. In this case, the output is* $o_{error} = true$ *when the user chooses to print. The second*

[1]This definition is done for explanation purposes only. We do not construct or use partial evaluation in the current implementation.

```
/* DBtables.h */
typedef struct {
    Char user[10];
    Char printer[10];
    Char phoneNumber[10];
    Char departmentCode[10];
} UserConfig
typedef struct {
    Int reportNum;
    Boolean printable;
} PrintReportConfig
/* main.c */
#include DBtables.h;
UserConfig userConfig;
PrintReportConfig printReportConfig;
Report3() {
  computeAndViewReport();
  if (getUserCommand()=="Print") {
    SQL(select * from ReportConfig into printReportConfig
              where ReportNum==3);
    if (printReportConfig.printable) {
      SQL(select * from UsersConfig into userConfig
                where User==getCurrentUser());
      if (isEmpty(userConfig.printer))
        userConfig.printer = selectPrinterDialog();
      printReport(userConfig.printer);
    } else {
      reportErrorMessage("Report #3 is not printable"); }
  }
}
```

Figure 1: An example of a toy customization based program.

row in Table 1, which corresponds to configuration C_{new}, represents the program when printing is allowed and there are two users u_1 and u_2 and two printers. In this case the output depends on the input. For example, for the user input

$$[print(user = u_2), select(printer = ptr2)]$$

the output is [$o_{select} = true, o_{printed} = prt2$].

Programs $[\![P]\!](C_1)$ and $[\![P]\!](C_2)$ are *observationally equivalent* if for every input $I \in \Gamma$ they yield the same output:

$$([\![P]\!](C_1))(I) = ([\![P]\!](C_2))(I), \forall I \in \Gamma$$

We say that a customization change from C_{old} to C_{new} *program-level impacts* P if $[\![P]\!](C_{old})$ and $[\![P]\!](C_{new})$ are not observationally equivalent. We define $impactProg(C_{old}, C_{new})$ to be the set of programs affected by the change from C_{old} to C_{new}.

EXAMPLE 3. *The two specialized programs $[\![$`Report#3`$]\!](C_{old})$ and $[\![$`Report#3`$]\!](C_{new})$ shown in Table 1 are not observationally equivalent. Clearly, the behave differently on user input request for printing. Therefore, $impactProg(C_{old}, C_{new})$ contains `Report#3`. An example in which a customization change has no impact on `Report#3`, is a modification of `ReportConfig` customization table for a different report, say `Report#1`. In this case, program `Report#3` is not affected, as the resulting programs are observationally equivalent.*

Although program level customization change impact information can aid in test scoping and understanding the change, users need more elaboration as to which parts are affected and where to focus testing effort within a program. As we shall see in Section 6, the number of output identifiers can be thousands even for a medium sized program, indicating that a complete test of a program is tedious and time consuming. For this, we define a observationally equivalent with respect to specific output identifier, and the detailed impact of a customization change as follows. Outputs O_1 and O_2 are *equivalent with respect to set $\Phi \subseteq$ OID* of output identifiers if for every $id \in \Phi$ we have $O_1(id) = O_2(id)$. Programs $[\![P]\!](C_1)$ and $[\![P]\!](C_2)$ are observationally equivalent with respect to Φ if for every input $I \in \Gamma$ the outputs of $[\![P]\!](C_1)$ and $[\![P]\!](C_2)$ are equivalent with respect to Φ. Notice that for $\Phi =$ OID this coincides with the notion of observational equivalence defined above. The *detailed impact of a customization change* from C_{old} to C_{new} on a program P is the minimal set $\Phi \subseteq$ OID s.t., $[\![P]\!](C_{old})$ and $[\![P]\!](C_{new})$ are observationally equivalent with respect to OID $\setminus \Phi$. Formally, $impactDetail(C_{old}, C_{new})$ is a set of pairs, {(P , Φ_p)}, of affected programs and output identifiers.

EXAMPLE 4. *The detailed impact of the customization change from C_{old} to C_{new} presented in Table 1, contains all the output identifiers, since there is at least one input for which the output is changed for each one of the output identifiers. For input [$print(user = u2), select(printer = ptr2)$] the output map changes from [$o_{error} = true$] to [$o_{select} = true, o_{printed} = ptr2$], and therefore all three output identifiers are affected. Notice that the detailed impact depends on the set of configurations, specifically on the `UserConfig` relation. In a case where all the users have a default printer set, the detailed impact would not include o_{select} as no input sequence would cause the printer select dialog to be shown.*

2.2 Abstracting Customization Change Impact

Customization tables are customer-specific and proprietary. They are regarded as a competence factor since they usually hold critical missions' specific behavior. Because PanayaIA is service based and is not deployed on the customers' machines, it has limited access to the specific configuration. Instead, the tool abstracts the configuration and estimates impacts as formulated below.

To abstract the concrete customization changes we consider a set of customization changes represented as a relation Δ, i.e., set of pairs $\langle C_{old}, C_{new}\rangle$. The impact of such a change is defined by a pointwise extension, i.e.,

$$impactProg(\Delta) = \bigcup_{\langle C_{old}, C_{new}\rangle \in \Delta} impactProg(C_{old}, C_{new})$$

Similarly, $impactDetail$ is extended to a delta relation. As the set of configuration pairs is potentially unbounded, the representation of an *abstract customization change*, Δ, is the set of database columns for which there was an attribute change in one of the pairs $\langle C_{old}, C_{new}\rangle \in \Delta$. An abstract customization change is said to be *atomic* if it contains a single database column.

PanayaIA supports three types of concrete customization changes: Adding a tuple to a given relation, removing a tuple from a relation and changing an attribute. The abstraction provides a mapping from concrete customization change $\langle C_{old}, C_{new}\rangle$ to an abstract customization change Δ. For example, the Δ for adding a tuple to a relation R is the set of all pairs of configurations C_{old}, C_{new} s.t., the only change between C_{old} and C_{new} is the addition of new tuples to R. We abstract Δ and represent it as the set of all columns of R.

EXAMPLE 5. *The abstract customization change of modification to the `Printable` column of `ReportConfig` customiza-*

<table>
<tr><th rowspan="2"></th><th colspan="2">Configuration</th><th rowspan="2">Specialized program after partial evaluation</th></tr>
<tr><th>ReportConfig</th><th>UserConfig</th></tr>
<tr><td>C_{old}</td><td><table><tr><th>ReportNum</th><th>Printable</th></tr><tr><td>1</td><td>false</td></tr><tr><td>3</td><td>false</td></tr></table></td><td><table><tr><th>User</th><th>Printer</th><th>...</th></tr><tr><td>u_1</td><td>prt1</td><td>...</td></tr><tr><td>u_2</td><td>null</td><td>...</td></tr></table></td><td>Report3() {
  computeAndViewReport();
  if (getUserCommand()=="Print")
    reportErrorMessage("Report #3 is not printable");
}</td></tr>
<tr><td>C_{new}</td><td><table><tr><th>ReportNum</th><th>Printable</th></tr><tr><td>1</td><td>false</td></tr><tr><td>3</td><td>true</td></tr></table></td><td><table><tr><th>User</th><th>Printer</th><th>...</th></tr><tr><td>u_1</td><td>prt1</td><td>...</td></tr><tr><td>u_2</td><td>null</td><td>...</td></tr></table></td><td>Report3() {
  computeAndViewReport();
  if (getUserCommand()=="Print") {
    SQL(select * from UsersConfig into userConfig
        where User==getCurrentUser());
    if (isEmpty(userConfig.printer))
      userConfig.printer = selectPrinterDialog();
    printReport(userConfig.printer);
  }
}</td></tr>
</table>

Table 1: Partial evaluation of `Report#3` for two configurations which differ in the attribute `Printable` in relation `ReportConfig` for `Report#3`. The declaration part is the same as in Figure 1.

tion table abstractly represents all possible customization changes in which the tuple of one report was changed from allow to disallow and vice-versa. The abstract detailed impact of this change on `Report#3` are all OID of the program, since it also abstracts all possible configurations in which some users have default printer and some do not. In the example of adding a tuple to the `UserConfig` customization table, the abstract impact includes object identifiers o_{select} and $o_{printed}$. The object identifier o_{error} is not affected since there is no configuration set in which this customization change impacts o_{error}.

Given an algorithm al that abstractly computes $impactProg_{al}$, the impact of an abstract customization change, we say that al is sound if for every abstract customization change Δ:

$$impactProg_{al}(\Delta) \supseteq impactProg(\Delta)$$

We define soundness of $impactDetail_{al}$ in a similar manner.

3. CHANGE IMPACT TOOL FOR SAP

This section presents characteristics of the SAP system that are relevant for the tool and for the analyses presented in Section 4, as well as a high level overview of the PanayaIA tool for ERP professionals. The PanayaIA tool is built for ERP professionals to interactively utilize during their task of performing customization changes for maintenance and enhancement of their SAP system. During this task, ERP professionals verify the changes and perform testing to the system. In order to be incorporated into this process, the tool needs to provide an online and immediate response to queries regarding impact of changes. For clear understanding of the impact results, the tool needs to report the impact in the ERP professionals terminology. Due to the large system, the impact of a change can be very large either due to a change that indeed impacts a significant part of the system or due to false-positives of the analysis. PanayaIA utilizes a few techniques to aid the user with managing the impact result.

3.1 Characteristics of the SAP System

The SAP system contains a proprietary language, a development environment, and an application server, all used to develop and activate the different application modules. SAP functionality is programmed in its own proprietary language called ABAP, first developed in the 1970s and has evolved greatly over the years [16], starting from a macro-assembler used exclusively for reports, through an interpreter language aiming at creating dialog programs in the 1980s. In the 1990s, ABAP has continued to evolve as a forth-generation language and became the base language for all SAP application modules. Toward the end of the 1990s, ABAP was enhanced with object oriented paradigm. Today's ABAP is a powerful language with thousands of constructs including database processing constructs and object oriented paradigm.

The SAP programs are interactive (dialog-based) and database intensive as most of the data and configuration are stored in the database. Some database tables have a large number of columns (up to hundreds of columns), which are mirrored in the code as large structures. The columns have diverse purposes and different parts of the code manipulate small portions of the columns according to their functionality. To support the customizable business processes, large decision trees are a common implementation paradigm, in addition to dynamic dispatchers that are used in dynamic computation of the program flow. The memory allocation is typically on the stack and mostly as global variables, which are commonly used for storing and passing information between procedures. Dynamic allocation, though possible is rarely used. Some ABAP constructs, such as pointer manipulation, object-oriented code, and concurrency, are not commonly used and therefore, for ease of implementation, are not supported by the current version of PanayaIA.

In addition, some of the SAP programs are maintained for more than a decade during which more and more functionality has been added. The code itself is tightly coupled and probably contains a rather large portion of unreachable code.

SAP code base is comprised of about 60,000 compilation units which are linked together into approximately 9,000 programs. Many compilation units are shared between many programs. In section Section 4 we present our customization change impact analysis algorithms that analyze the SAP code in a general manner, In Section 6 we elaborate how the SAP specific complications are handled in our implementation.

3.2 PanayaIA Tool

PanayaIA is an on-demand web tool designed for ERP professionals. It provides a clear view of the impact of a customization change on the entire ERP system. PanayaIA consists of two main processes: an online process that is responsible to communicate with the end-user and an offline process that orchestrates the different analysis algorithms and constructs the Customization Change Impact Repository (CCI Repository).

PanayaIA's offline process uses a computer grid (cluster) to perform the analysis of all the ERP programs. The duration of the analysis of approximately 4,000 programs, on a grid of several dozen of processors, running Linux RedHat 4[2], can be up to a few hours and depends on the exact analysis chosen. The deviation of analysis time is considerable. The average is a few minutes per program, while the analysis of the largest programs requires a few hours. Memory consumption, which is less of a concern than running time, is on average less than 1GB (running on a 64-bit system) and ascends to 20GB for the largest programs.

Each program is analyzed in a batch mode and the customization change impact results are stored in the CCI repository. The repository contains a mapping from each customization table and column (an atomic abstract customization change) to the affected program and its affected output identifiers (dialog fields, error messages, etc.).

PanayaIA's online process contains two major components: a front-end and a back-end. The front-end module handles the communication with the ERP professional. The users report the change as they are about to transfer to the production system and after executing the query the user receives the impact it might have on the whole system. The impact is displayed in the terminology of the ERP professional: not procedures and statements but dialog fields, error messages, and other business-related output identifiers. The back-end module uses a computer cluster to answer efficiently on the users' requests online over the web (as an on-demand service). Each impact analysis request is transformed into queries against the CCI repository. The user's customization change is disassembled into atomic abstract changes, and the back-end module retrieves the corresponding data from the CCI repository. The final result is the combination (superposition) of all the affected elements that satisfies the customization change criteria.

One of the big challenges in the user interface and user experience is the ability to manage a large impact result set. PanayaIA uses several techniques in order to give the user a clear image of the impact on the system as a result of a customization change: (i) Grouping — the affected programs are grouped into application components each one represents a different business module (e.g., accounting, sales, human resources) and the user can drill in the results relevant to her; (ii) Similarity — programs that have a similar impact (usually, in the library code) are grouped and only one representative is displayed to the user; (iii) Filtering — some of the users may supply usage information (i.e. a list of the programs that are used in their organization, usually gathered by a build-in audit mechanism) thus enabling PanayaIA to focus and display only the impact on this sub group of programs.

4. CUSTOMIZATION CHANGE IMPACT ANALYSES

We present three basic static analysis algorithms for customization change impact analysis. Each one analyzes the program's source code for a given atomic abstract customization change Δ and reports an approximation of the impact either $impactProg_{al}(\Delta)$, or $impactDetail_{al}(\Delta)$. This approximation can include false-negatives and false-positives. False-negatives may result due to unsound treatment of program constructs, in particular dynamic constructs, such as dynamic procedure calls (a call with a computed target) and dynamic SQL statements. Additional unsoundness comes from unconservative abstraction of customization change impact as explained in this section. In many cases, conservative methods that do not produce false-negatives, report a significant amount of false-positives. In practice, losing soundness in a controlled manner may yield better value to the end-user than a large number of false positives. The set of analyses enables study of the differences, precision gain and loss, and the cost (memory and time) of each analysis. The balancing algorithm, presented in this section, aims at gathering the best results with fewer false positives and false negatives by a very specific combination of the results of the three basic algorithms.

[2]Our experimental study shows that running time of the analysis on Linux machines is about 70% of the running time on Windows XP x64.

4.1 Naive Syntactic Algorithm

The naive syntactic algorithm defines impact as the syntactic existence of a SQL select statement from the customized table. The entire abstract syntax tree (AST) of the program is scanned to find select statements from each customization table. The analysis is flow- and context-insensitive. This analysis reports $impactProg_{naive}$ impact on a program level, and does not detail which output identifiers are affected. Except for cases of undetermined dynamic constructs, which may cause analyzing incomplete code of a program, this analysis is sound and does not have false negatives. Clearly, this analysis has a rather high false positives rate, especially for update customization in which one attribute is modified.

EXAMPLE 6. *Since there is an SQL select statement from the* `UserConfig` *table in program* `Report#3`*, the naive syntactic algorithm yields that* `Report#3` *is affected by an update customization change to the* `PhoneNum` *attribute of the* `UserConfig` *configuration of an existing user. Clearly, since there is no concrete impact on this program, this is an over approximation.*

4.2 Column Usage Analysis

In many cases, only a subset of the attributes of a customization relation are used in a program. From the syntactic point of view there is an SQL select statement that retrieves an entire relation, however, the semantics of the program use the values of only a subset of the retrieved attributes. The goal of the column usage analysis is to improve precision over the naive syntactic algorithm for customization changes that only modify a subset of the columns.

Inferring which columns are used in a program is a non trivial problem since the number of columns is huge and since the programming language provides low level operations which do not necessarily respect the field boundary (e.g., casts between different structure types, copying a structure into a scalar and back).

The column usage analysis is a flow-insensitive analysis that first computes *equivalent access-paths* in the program. Intuitively, two access paths, e.g., a reference to variable or to a field of a structure variable, are equivalent if there may be a transitive memory copy between them. Thus, if two access paths belong to different equivalent classes there cannot be any data dependency between them. We claim that a program is not affected by a customization change if there is no *direct* use of any of the equivalent access paths of the changed attribute.

In [21] a computation of low level atoms, a sub-part of a field, is presented. We implemented a variant of [21] which is a unifi-

Statement	Pseudo code
x = y	unify(group-id(x),group-id(y))
structure copy S1 = S2 index based	for each field f1 of S1 for each field f2 of S2 s.t f1 overlaps f2 unify(group-id(S1.f1),group-id(S2.f2)) process assignment S1.f1 = S2.f2
x = (cast)S1	unify(group-id(x),group-id(e)) where e is an access-path with prefix S1

Table 2: Handling of assignments in the column usage analysis. Symbols x and y denote primitive type access paths, S1 and S2 denote structure type access-paths.

cation based algorithm, similar in nature to [24], that can handle type casting between different structure types, arrays or SQL statements. The basic access paths are variables, field-paths (up to some k limit), a cell of an array, or a column of a database table. Table 2 displays handling of some interesting basic assignments in building equivalence classes. Each access-path is associated with a group-id. The `unify` operation, unifies two equivalence classes into one equivalence class. Assignment of structures of different types is processed as a memory buffer copy, similar to memcpy() in C. In this case, we unify fields that their indexes are overlapping, as follows: field f, which has starting and ending indexes f_s and f_e, overlaps f' if $[f_s, f_e] \cap [f'_s, f'_e] \neq \emptyset$.

EXAMPLE 7. *In program* `OrderHandling()` *shown in Figure 2 the following three equivalent access-paths are found: (i) The column* `DepartmentCode` *in table* `UserConfig`*; (ii) The access path* `userConfig.departmentCode` *due to the SQL select statement that assigns to this access path the column value; (iii) The access paths* `userDetails.departmentHead` *and* `userDetails.subDepartment` *over the formal parameter of* `checkAuthorization()` *due to the type casting in the parameter passing.*

The impact of an atomic abstract customization change, Δ, can be addressed as follows: the column c of table t, which is contained in Δ, is computed to be a member of equivalence set S; If any access path $p \in S$ is used in the program's control or output statements then we say that the customization change impacts the program. Similar to the naive algorithm, there is no inference of the detailed impact.

EXAMPLE 8. *An abstract customization change to column* `DepartmentCode` *in table* `UserConfig` *impacts the program* `OrderHandling()` *according to the column usage analysis since access path* `userDetails.departmentHead` *is used in the program and is in the same equivalence access-path as the changed column.*

In the program `Report#3` *the equivalence access-path set of* `DepartmentCode` *column of* `UserConfig` *which includes also* `userConfig.departmentCode`*, is not used in the program. Therefore, the analysis is able to infer that a customization change to the* `DepartmentCode` *attribute has no effect on* `Report#3`*.*

4.3 Slicing Algorithm

Program slicing algorithms [12, 13, 27, 5] are widely used for code change impact analysis, program refactoring and other program understating and maintenance applications. This technique, which computes the part of the program that are potentially affected by some point of interest, is also applicable for the customization

```
#include DBtables.h;
UserConfig userConfig;
PrintReportConfig printReportConfig;
typedef struct {
    Char user[10];
    Char printer[10];
    Char phoneNumber[10];
    Char departmentHead[1];
    Char subDepartment[9];
} DepUserConfig;
OrderHandling() {
    if (getUserCommand()=="View") {
      ... }
    else if (getUserCommand()=="New") {
      SQL(select * from UsersConfiguration into userConfig);
      checkAuthorization(cast<DepUserConfig>(userConfig));
      .. }
}
Boolean checkAuthorization(DepUserConfig userDetails) {
    if (userDetails.departmentHead == "H")
      return true;
    else {
      reportErrorMessage("No authorization to create a new order");
      return false;
    }
}
```

Figure 2: An example of a program with type casting. File DBtables.h is shown in Figure 1.

change impact and can provide a detailed report of the output identifiers affected.

The core data structure used for slicing is a Program Dependence Graph (PDG) whose nodes are the statements of the program, and includes two types of dependency edges: control dependency and data dependency. Control dependency occurs when a statement affect the execution of another statement. Data dependency occurs when a value written in one statement is read in the other statement. Slicing, from a very high level view, amounts to computing transitive closure of the PDG.

In the context of a customization change, we apply a forward slice with the SQL select statements from the customized table as the seed. Only output identifiers used or modified at statements that are contained in the slice are reported as affected outputs. A slice is a safe approximation to the customization change impact problem. We claim that if an output identifier is in the concrete detailed impact then there is a statement that uses this output identifier in the slice.

Conversely, we claim that if an output identifier, id is not used or modified in any statement in the slice for column c then id is not affected by a customization change to c. Intuitively, if the partial evaluated programs $[\![P]\!](C_{old})$ and $[\![P]\!](C_{new})$ are not observationally equivalent with respect to output identifier id, then there exists an input $I \in \Gamma$ for which the output maps: O_1 form $([\![P]\!](C_{old}))(I)$ and O_2 from $([\![P]\!](C_{new})(I)$ are not equivalent with respect to id. Thus, $O_1(id) \neq O_2(id)$. This implies that id must be dependent on the change, through control and/or data flow. The source code corresponding to this dependency must be dependent on the SQL select that fetched the column.

One of the main challenges in slicing is the accuracy. Slicing is an over approximation to the abstract change impact problem since

it can not conclude whether or not a particular output identifier may change. Assume for example, that a conditional statement is contained in slice, however there does not exists a concrete customization change (C_{old}, C_{new}) and an input in which the evaluation of the condition differs at the two configuration. This implies that output identifiers that are only dependent on the condition are not affected. However according to the static slice, any statement that is control dependent on the condition is included in the slice, and output identifers used or modified at those statements are reported as affected.

4.3.1 *Limited Alternation Slicing Algorithm*

Applying slicing algorithms to huge and complicated programs such as the SAP code base, is challenging because of the complexity of the slicing algorithm and because the resulting slices can be too big to manage. PanayaIA offers an option to handle these two problems by using the limited alternation heuristic described below. When studying cases of large slices we concluded that control dependencies account for a large portions of those slices. The following are common reasons in which the handling of condition had caused an over approximated slice:

- Unreachable Code. The condition is always evaluated to either false or true. This is very common especially in library code, in which some portions of the code are unreachable in a particular program. Our unreachable code analysis (based on constant propagation [28]) fails to infer some of the unreachable code and therefore a slice may contain unreachable code.

- Equivalent code. In some cases, equivalent code sections appear in both the true and the false part of a condition. The slice contains this duplicated code sections as part of the slice, however, the side effect of this code is independent to the evaluation of the condition.

The *limited alternation* heuristic limits the number of times a path in the PDG can alternate traversing control dependency and data dependency edges. Note that these heuristic are unsound as there may be valid impacts which involve more than k alternations. Thus we remove some false positives but may introduce false negatives.

4.4 Balancing Algorithm

It is well known that slicing can yield large impacts due to imprecisions in the static analysis and the dynamic nature of software. The PanayaIA tool computes slices by combining several heuristics. The main idea is to combine the results of different potentially unsound and imprecise approaches aiming towards smaller sets of impact details while not missing obvious impacts.

The choice between the naive syntactic approach and column usage approach is clear, if the change considered is an update to a column, the column usage approach is superior, otherwise, the naive syntactic approach has the same precision with better performance. The slicing based approaches can give more detailed information, i.e., the affected outputs. Furthermore, since the algorithm is flow-sensitive it can yield more precise information. However, due to the size and complexity of the code, unsound heuristics are used to handle issues such as dynamic call resolving. This may cause the slicing based algorithms to miss important impacts.

We implemented an algorithm, named *balancing algorithm*, that provides $impact_{balance}(\Delta)$ by combining the results from the limited alternation slicing, full slicing and flow insensitive algorithms in the following way, and as defined in Figure 3. For every change

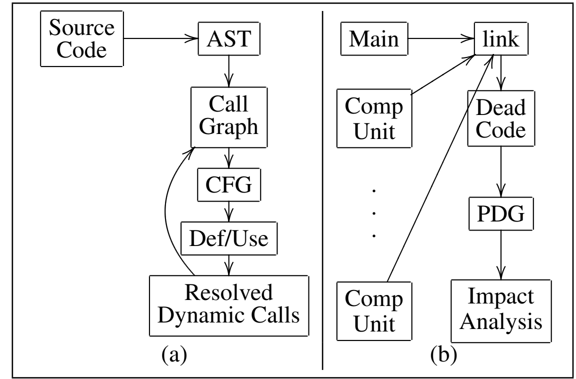

Figure 4: High level overview of the modular analysis of a compilation unit (a) and of the analysis of a whole program (b).

the detailed impact is computed using the limited alternation slicing, $impactDetail_{2Alt}$. If a program is not affected, the full slicing algorithm is consulted ($impactDetail_{full}$)and if it discovers affected output identifiers, they are presented instead. If for a specific customization change both slicing algorithms return that there is no impact in any of the programs, the flow insensitive results are used instead. Note that in this case, only the affected programs are returned, and not the affected output identifiers. This is useful in case the unsoundness of the heuristics used in the slicing fully prevents the system from discovering any impact of a change.

5. IMPLEMENTATION

This section presents PanayaIA's analysis for SAP code version 4.6C and lessons learned while building and studying the results of the tool.

5.1 High Level Overview

Due to the large code base of SAP and the sharing of libraries between programs, we incorporate 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. During this phase the impact analysis is performed in a batch mode as described in Section 3.

5.2 Compilation Unit Level Analysis

During the modular analysis of a compilation unit, the order of invocation of procedures that are external entries is undetermined. Figure 4 (a) sketches the analysis of a compilation unit. First an abstract syntax tree is generated by parsing the source code. Next, a call graph is constructed containing only static calls that are within the compilation unit. A control flow graph, with basic blocks as nodes, is constructed and handles arbitrary control flow statements, such as break and continue, in a standard way. Def/use relation within the compilation units is computed. An analysis that computes possible targets of dynamic calls is performed, which has a side effect of updating the analyses performed.

5.2.1 *Modular Def/Use Chains*

The def/use analysis computes for each definition point p (a modification of an access-path at a program point) which program points may use the value set at p. This is a rather standard analysis

$$impact_{balance}(\Delta) \supseteq \begin{cases} (p, \Phi_p) & (p, \Phi_p) \in impactDetails_{alt2}(\Delta) \\ (p, \Phi_p) & (p, \Phi_p) \in impactDetails_{full}(\Delta) \wedge \nexists \Phi'_p(p, \Phi'_p) \in impactDetails_{alt2}(\Delta) \\ impactProg_{col}(\Delta) & \text{if } \Delta \text{ is attribute change and } impactDetails_{alt2}(\Delta), impactDetails_{full}(\Delta) = \emptyset \\ impactProg_{naive}(\Delta) & \text{if } \Delta \text{ is tuple change and } impactDetails_{alt2}(\Delta), impactDetails_{full}(\Delta) = \emptyset \end{cases}$$

Figure 3: Definition of the balancing algorithm.

for the purpose of slicing. One complication for the def/use analysis for SAP programs is the very large number of global variables. Each compilation unit may declare its own global variables. The scope of the variable is static within the compilation unit. Thus, a variable may be live between two invocations of procedures in the same compilation unit. It is rather common programming practice to declare variables as global, and in a typical compilation unit this can amount to hundreds of global variables. In addition there are program global variables which are in the scope of the program (all compilation units). Another complication is that each compilation units contains procedures which are external entries to this compilation unit (i.e. procedure that may be invoked from a different compilation unit).

A common solution is to add all global (compilation unit and program level) variables to the signature of each procedure. This requires adding all global variables (from all compilation units) in order to pass to callee, which are unknown during the modular analysis of a single compilation unit. Our solution follows this approach and simulates parameter passing by adding a structure program global variable to contain all global variables as fields. Procedures entries and return from procedure calls, are regarded as def points for all parameter passing. The def/use chains are computed intra-procedural. The inter-procedural phase adds def/use relations for procedure call and return statements.

To improve scalability and efficiency, we implemented two optimizations to reduce the amount of parameter passing. First, a flow-insensitive context-sensitive analysis computes which variables are transitively used or modified at each procedure. Variables that are not used need not be passed as parameters, and variables not modified need not to be returned. Another optimization is *localization*, which computes program points where global variables can be safely replaced with local ones. Section 6 shows the improvement which reduced the number of global variables drastically.

5.2.2 Resolving Dynamic Calls

Dynamic procedure calls are rather common and in some cases can be determined at the compilation unit level. We compute possible targets and update the call graph, CFG and use/def with respect to the new targets. With the updated model of the compilation unit, additional targets for dynamic calls can be resolved. This process can be repeated until no new targets are resolved. We implemented a constant propagation algorithm that computes for each dynamic call the possible targets. In some cases the static analysis cannot infer all possible targets, leaving this call as partially resolved and may cause some false negatives.

5.3 Program Level Analysis

Each program in SAP has a main compilation unit which is the entry point for the program. The main compilation unit transitively calls other compilation units. Figure 4 (b) illustration the program level analysis phase. The first step in analyzing a whole program is to *link* information from all compilation units' analyses together. This includes building a call graph that represents all inner and external calls, by composing all call graphs of the compilation units and resolving external calls' targets. At this phase, procedures of a compilation unit may become dead as a program may use only parts of a library's public entry points. During the link phase, the control-flow graph and def/use chains are updated to represent the data passed at procedure calls between compilation units.

5.3.1 Call Graph

Since SAP is a highly integrated system, the boundary of a single program is not always clear. The semantics of programs overlap , as through one program the user can perform actions that are carried out by another program. In addition, there is a very high percentage of sharing of library code. This is mainly due to services that are dispatchers. A particular activation of a service may perform a small portion of the functionality, including a subset of the possible procedure calls. When syntactically analyzing the program, the number of potential reachable procedures is very high. We performed a mini study on 160 programs consisting of 94,000 procedures. On average each procedure is potentially called by 30 programs, where approximately half the procedures are potentially called by a single program and half are potentially called by more than 1/3 of the programs. Building a complete program with all the reachable procedures caused huge overhead both in scalability and in false positives.

To overcome this problem, we have built a set of SAP knowledge based heuristics that define the boundaries of a program. First, we define a set of compilation units that are regarded as SAP basis and are not included in any program as impact on them are of no interest. Next, each compilation unit is associated with a component. We have built a mechanism that, given the main compilation unit, defines which components are related and should be included as part of the program. This reduced the size of programs drastically, and our study showed that the false negatives caused by this heuristic are minor compared to the false positives removed.

5.3.2 Unreachable Code Analysis

Some SAP programs have been maintained for a few decades, thus, include a high portion of unreachable code. In addition, some procedures are developed for multiple purposes but each program utilizes them for specific uses. Inferring unreachable code is important in order to avoid false positives. Dead statements are marked as infeasible and are ignored by the impact analysis algorithms either as seed statements or as data or control dependent. We implemented an unreachable code analysis based on constant propagation in the style of [28].

5.3.3 Program Dependency Graph

We implemented a program dependency graph with two layers: a data flow layer and a control flow layer. The data flow layer consists of the def/use chains computed intra-procedurally at the compilation unit layer and the inter-procedural def/use chains that are computed at the linking phase of a program. The ABAP programming language contains constructs that have arbitrary control flow such as conditional exit from a loop or procedure. Control dependency is computed via post-domination relationship [20].

Program	Size				Optimization Algorithms			Call Graph Complexity		
	Stmts	Comp	Procs	BB	Dead	Globs	Locals	#Nodes	#SCCs	Max
Prog1	2,053	4	109	2,513	15%	174	51%	366	330	28
Prog2	11,582	41	455	12,096	19%	2,513	86%	1,721	1,344	142
Prog3	28,646	39	843	23,965	31%	4,856	85%	1,721	1,344	9
Prog4	61,394	107	1,437	48,347	25%	14,156	88%	5,697	5,582	29
Prog5	83,954	74	1,972	67,286	28%	9,946	78%	6,904	6,702	138
Prog6	141,109	221	3,579	119,513	32%	25,246	83%	12,862	1,2534	223
Prog7	194,436	171	4,557	171,078	20%	16,988	76%	20,184	19,142	358
Prog8	195,113	258	5,368	160,769	46%	30,918	83%	17,879	17,430	223
Prog9	211,423	227	5,021	175,222	16%	19,602	79%	22,031	21,144	470
Prog10	244,090	264	7,212	216,822	32%	41,564	86%	25,966	25,366	138
Prog11	264,054	194	5,517	226,579	17%	21,999	80%	26,072	24,296	358
Prog12	273,362	122	7,847	306,095	16%	21,175	82%	42,251	39,993	851
Prog13	480,248	311	11,998	457,908	14%	38,094	77%	53,937	48,839	1,445
Prog14	890,159	512	19,686	808,820	16%	63,111	78%	94,983	88,661	1,445

Table 3: Size (counted as statements, compilation units, procedures, and CFG basic blocks), optimization algorithms (unreachable code and localization) and call graph complexity measurements of the benchmark programs.

5.3.4 Slicing

The impact analysis algorithms are all implemented via a parametric slicing framework. The framework is parametric as to which dependencies to use during the slicing and when to stop. Zero alternations implies the naive algorithm. To achieve scalability and precise inter-procedural slicing, procedure summary information is computed [13]. The summary phase computes for each procedure the control and data dependency from each formal-in of the procedure to the formal-outs. The next phase is a slicing algorithm that starts from the SQL select statements in the programs. It utilizes the summary information when the slicing reaches an actual-in argument and continues with the dependent actual-out arguments. The result of the slicing are written into a repository. In the case of update change customization, where one column is modified, it is essential to report the slice that is relevant for the specific column and not the slice for the complete table. There are various alternatives in order to achieve this precision. In PanayaIA, a field sensitive analysis is implemented by propagating relevant field names along flow and control dependencies.

6. EMPIRICAL RESULTS

This section presents our empirical study of algorithms. First, we study and compare the three basic algorithms: naive, column-usage and slicing. The aim of this study is to understand the differences in terms of time and in terms of impact results. In addition we present our study of precision for end-users of the application, which uses the balancing algorithm.

6.1 Comparing the Basic Algorithms

The experiments presented in this section are on a selected benchmark of 14 programs which vary in size, complexity, and associated SAP component. Table 3 lists the selected programs. Columns 2–5 provide information regarding the size of the programs: (i) number of lines of code, (ii) number of compilation units, (iii) number of procedures, and (iv) number of basic blocks in the control flow graph. Columns 6–8 indicate the percentages of dead basic blocks, the number of declared global variables, and the percentage of variables that are localized by the localization algorithm. The localization savings are high and have a positive effect on the scalability of the impact analyses in terms of time and space.

Since we learned that program size is not the only factor for the impact analyses run time, we study the complexity of programs. We measure complexity by building the strongly connected components (SCC) [4] of the call graph. The call graph comprises of procedures and procedure call statements. Columns 9–11 in Table 3 report the number of nodes, the number of SCCs, and the number of nodes in the largest SCC. The reason for the rather large SCCs is the dialog invocations which are sometimes recursively called.

Table 4 lists a comparison of the naive, column-usage, and slicing algorithms by counting the number and size of slices obtained for each program. The balancing algorithm is not compared using these metrics since it is on a system level and not per program. The second column lists the number of database tables that have at least one database column which affects the program. This result is the same in all algorithms. For each analysis, the table lists: (i) The total number of database columns (from all customization tables) that impact the program; (ii) The average number of affected output identifiers per impacting column. These vary between the algorithms. For the naive algorithm, the number of database columns is the greatest (on average 9.3 columns per table) as all syntactically retrieved columns are regarded as impacting the program. All output identifiers of the program are regarded as affected since this analysis can not distinguish between affected and non-affected output identifiers. The number of output identifiers increases with the size of the program, and on average is more than 5,000 elements. The column usage analysis reduces the number of columns impacting a program to an average of 5.2 per table. This is due to the fact that for a large portion of SQL select statements a whole record is retrieved (via the select * command) but not all columns are used.

The results of two slicing algorithms are presented, the first is slicing with 2-alternation limitation. The 2-alternation analysis reports the fewest affected output identifiers. However, the number of columns impacting a program is rather low (on average 3.7), causing false negatives. The last pair of results are for the full slicing algorithm, which reports that more columns are impacting a program (on average 4.4). The number of average affected output identifiers is higher by about 35% ,from 17 to 23 in the 2-alternation algorithm vs. the full slice, respectively.

Elapsed time comparison of the algorithms is displayed in the last part of Table 4. Clearly, the naive and column usage analy-

sis are efficient especially when taking into account that a non redundant portion of the elapsed time is on writing the results to the repository. The full slicing algorithm's elapsed time is about 25% longer as compared to the 2-alternation algorithm. Although the number of statements in a program is highly influential on the running time, there are additional factors. The number of customization tables retrieved and the total number of SQL select statements from customization tables (it is rather common to have more than one statement as the seed statement) affects the running time. For example, the analysis time of Prog12 is comparably lower due to the lesser number of tables and impacting columns. The complexity of the program is another factor; Large SCCs may cause the analysis to perform additional iterations until a fixed point is reached. For example, Prog7 has rather large SCCs, requiring a three-fold analysis time as compared to Prog8 being approximately the same size. Memory consumption, which is less of a concern, is usually around the 1GB [3]. The memory peak for the analysis of the largest program in our benchmark is 12GB.

6.2 Understating Precision

In general it is not trivial to measure precision of static analysis algorithms. Moreover when the algorithm may contain both false-positives and false-negatives. We presents studies conducted and our ongoing effort at obtaining an accurate measurement for precision as well as increasing precision. These studies have been performed on analysis results of the production run as describes in Section 3.

In order to measure the precision of the different algorithms, we performed a human experience test to the results. Five internal expert SAP professionals have worked with the tool and provided feedback concerning the precision. The experts applied a few customization change queries upon repositories from different analyses. The common feedback from the 2-alternation algorithm is that the reported affected output identifiers are correct but there is an under-approximation as some affected programs are missing. For the full slicing analysis, the general impression was of excessive noise in the results. The results from the balancing algorithm yielded the best feedback.

In order to assess the precision of the production results, obtained via the balancing algorithm, we performed two independent studies. The first consist of external expert SAP professionals building a set of expectations without prior viewing the results of the PanayaIA tool and the second obtained by end-users of PanayaIA. We have asked external expert SAP professionals to create customization changes and provide a set of expected results for each change. Overall, we obtained 291 customization changes with a total 2657 expected results. The expected results were only at program level, as providing expectations regarding output identifiers is much more complicated. When utilizing this information for determining false-positives and false-negatives, the following complications evoked: (i) Some results reported by PanayaIA that were not in the expectations were counted as false positives but turned out to be correct and overlooked by the SAP professionals; (ii) Some false-negatives on output identifiers were not counted as such, due to other expectations on the same program. In order to increase precision, we have an ongoing effort in which we are iteratively fixing issues resulting from studying false-negatives and false-positives and have the experts update the expectations. Currently, we have reach a ratio of 113% false-positives and 11% false-negatives.

Our second study is obtained by allowing PanayaIA users to provide feedback regarding the results. A user may indicate for an impacted program whether it is a correct impact or a wrong impact (implying a false-positive). In addition, users may add the missing impacted programs (implying false-negatives). On collected feedback from 10 customers regarding a total of 61 customization changes that yields a total of 559 impacted programs, we measure false-negative and false-positive ratios. The false-negative ratio, measured as the amount of missing programs with respect to the expectation (missing + correct) is 15%. This implies that six out of seven expected programs are reported by the tool. The false-positives, measured as the amount of wrong information with respect to the expectation, is 71%, implying that on every three correct programs there are two additional false-positive programs.

[3]We run all analyses on a 64-bit system to overcome limitations to the Java heap size.

7. RELATED WORK

The contribution of this paper is in two main dimensions and has related work in the area of customization change impact analysis and in the field of application and utilization of program slicing.

Customization Change Impact Analysis.

The process of customization change is risky and costly. Organizations need to put a significant amount of human resources on change management [26]. There are limited tools that aid in understanding and verifying a customization change. Intellicorp support package [14] provides impact analysis for support packages (upgrade of code changes) to aid in determining which parts of the system need regression testing. The underlying technique used by the tool is unclear but believed to be rather syntactic. The tool reports impacts only on a program level.

The SAP development environment provides a set of tools for aiding in understanding programs [15]. These tools, which include dynamic traces of SQL statements and syntactic references lookup, can be used by ABAP programmers to understand the impact of a source or customization change. Our solution differs drastically from the above works, as we aim at aiding SAP professionals and not programmers in customization change without observing programs' source code or traces.

Utilizing Slicing Algorithms.

Slicing and subsequent manipulation of slices shows great promise for supporting many software-engineering tasks: It has applications in program understanding, maintenance [7, 8], debugging [17], testing [3, 2], semantic differencing [10], specialization, reuse [18], and merging [11]. More applications of slicing are described in [27]. PanayaIA uses slicing to identify potential impacts of customization changes, which is a new application for slicing.

One of the major limitations of employing slicing for performing software engineering tasks in general and identifying impacts in particular is the size of the slices that may be very large. Determining which statements are most relevant to the user is non-trivial. In [23] a solution for reducing the slice to relevancy data computation statements is presented. Another possibility is to limit the slice traversal. For example, in [5] a depth-limited slice over a value dependency graph is presented. Another possibility is to specialize the slice to a given set of inputs [6].

8. CONCLUSION

This paper presents a new tool that enables ERP professionals to easily perform customization changes. This is a novel approach for aiding ERP professionals in one of their most risky and costly tasks. The PanayaIA tool consists of a collection of adaptations of well-known static analysis techniques in order to achieve scalability on large code base and balance between false-positives and

Program	Tables	Naive		Usage		2-Alter		Full		Elapsed time (min)			
		Cols	IDs	Cols	IDs	Cols	IDs	Cols	IDs	Naive	Usage	2-Alter	Full
Prog1	5	69	139	22	139	23	7	24	11	0.1	0.1	0.3	0.3
Prog2	37	548	454	211	454	69	6	123	9	0.4	0.7	0.6	0.8
Prog3	28	412	454	212	454	125	18	129	18	0.3	0.4	1.1	1.5
Prog4	122	1,223	1,597	831	1,597	442	38	599	59	0.8	1.4	3.7	4.0
Prog5	55	852	1,339	328	1,339	134	8	205	15	0.6	0.6	2.9	3.4
Prog6	183	1,908	3,009	1,115	3,009	627	7	664	15	1.5	1.9	7.9	9.6
Prog7	198	2,032	4,759	1,300	4,759	956	39	1,032	33	1.4	2.3	23.3	26.8
Prog8	293	2,029	4,401	1,265	4,401	628	8	759	17	1.4	2.1	7.4	8.5
Prog9	175	1,691	4,948	1,138	4,948	596	9	784	14	1.3	1.9	8.6	11.0
Prog10	274	2,492	6,603	1,808	6,603	800	9	1,135	14	1.8	2.8	9.7	14.1
Prog11	228	2,084	6,237	1,342	6,237	999	28	1,164	36	1.5	2.5	39.4	42.4
Prog12	205	1,432	10,230	735	10,230	554	6	606	9	1.4	2.1	20.6	37.3
Prog13	284	2,888	10,346	1,828	10,346	1,462	27	1,621	38	2.3	3.9	44.5	50.6
Prog14	442	3,962	21,363	2,818	21,363	2,031	26	2,230	32	4.4	5.9	87.8	118.5

Table 4: Elapsed time and result comparison: The number of customization tables which are impacting a specific program; The number of columns impacting a program and the average number of affected output identifiers per column, for each analysis.

false-negatives. The paper also provides evidence of the applicability of the method and tool for end-users on real customization change cases.

9. REFERENCES

[1] R. Arnold and S. Bohner. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Pr, 1996.

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

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

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.

[5] M. D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, 1994.

[6] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Symp. on Principles of Prog. Lang.*, 1995.

[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.g*, 1991.

[9] J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors. *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lecture Notes in Computer Science*. Springer, 1999.

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

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

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

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

[14] Intellicorp Inc. http://www.intellicorp.com. 2007.

[15] H. Keller and S. Krüger. *"ABAP Objects: ABAP Programming in SAP NetWeaver*. Addison-Wesley, 2002.

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

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

[18] J. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Commun. ACM*, 94.

[19] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Int. Conf. on Soft. Eng.*, 2004.

[20] K. Pingali and G. Bilardi. Optimal control dependence computation and the Roman chariots problem. *Trans. on Prog. Lang. and Syst.*, 1997.

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

[22] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *Conf. on Object-oriented Prog., Syst., Lang., and App.*, 2004.

[23] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Conf. on Prog. Lang. Design and Impl.*, 2007.

[24] B. Steensgaard. Points-to analysis in almost-linear time. In *Symp. on Principles of Prog. Lang.*, 1996.

[25] M. Themistocleous, Z. Irani, R. O'Keefe, and R. Paul. ERP problems and application integration issues: An empirical survey. In *Hawaii Inter. Conf. on System Sciences*, 2001.

[26] M. Themistocleous, Z. Irani, R. O'Keefe, and R. Paul. Change management underpins a successful ERP implementation at Marathon Oil. In *Journal of Organizational Excellence*, 2004.

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

[28] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *Trans. on Prog. Lang. and Syst.*, 1991.