A Framework for Analyzing Programs Written in Proprietary Languages

V. Krishna Nandivada[1] Mangala Gowri Nanda[1] Pankaj Dhoolia[1] Diptikalyan Saha[1]
Anjan Nandy[2] Anup K Ghosh[2]

IBM Research - India[1], IBM GBS - India[2]

{nvkrishna, mgowri, pdhoolia, diptsaha, anjan.nandy, anup.ghosh}@in.ibm.com

Abstract

There are several commercial products that use proprietary languages, which typically look like a wrapper around (some proprietary extension of) the standard SQL language. Examples of these languages include ABAP, Informix, XBase++, SQR and so on. These application are difficult to analyze not only because it is hard to model the semantics of the underlying database systems but also because of the lack of standard tools for analysis. One naive way to analyse such programs is to collect dynamic trace using proprietary debuggers and run the analyses on the trace. However, this form of dynamic trace collection can be a severe performance bottleneck. In this paper, we present our experience with building a framework to help in efficient program analysis in the context of ticket resolution for ABAP programs.

In our framework, we first translate the given ABAP programs to semantically equivalent annotated Java programs. These Java programs are then executed to generate the required dynamic trace. Our framework allows the plugging of off-the-shelf static analysis tools (applied on the Java programs) and dynamic trace analysis tools (on the generated trace) and maps the results from these analysis tools back to the original ABAP programs. One novel aspect of our framework is that it admits incomplete ABAP grammar, which is an important aspect when dealing with proprietary languages where the grammar may not be publicly available. We have used our framework on several benchmarks to validate the translation, and establish the efficiency and the utility of our instrumented Java code along with the collected trace.

Categories and Subject Descriptors: D.3.4 [**Processors]** Debuggers, Code generation
General Terms: Verification, Experimentation, Languages
Keywords: error recovery, source to source translation

1. Introduction

Ticket resolution is an important part of a service organization. In a typical context, a client discovers a bug in the field, which needs to be resolved as fast as possible. For the scope of this paper, we assume the code and not the database to be source of the bug; this is reasonable, because the same database typically feeds into several other applications that do work properly.

Working with proprietary code in an proprietary environment, in an industrial settings brings in interesting set of challenges: a) hard to reproduce the bug outside the client execution environment, b) the original code writer may not be the debugger of the code. This problem gets compounded when the number of available program analysis tools (proprietary or otherwise) for that language are limited. In this paper, we discuss our experience with bug resolution in the context of a proprietary language ABAP. However, the techniques and methodologies developed here can be applied to other languages with a similar purpose (database access and report generation).

A naive way of analyzing these programs is to analyze the trace of the faulty program, obtained via running it through a proprietary debugger inside an automated script that collect a trace of the values of all variables at all program points. Unfortunately, this can be a major performance hurdle; for instance, such a trace collection for a small ABAP program, that otherwise runs well under 60 seconds, takes more 20-30 minutes. It may also be noted that, in general the approach of (automated) instrumentation of the ABAP source programs to collect the trace is not a feasible option, as executing a modified version of the source program is typically not permitted by the clients.

In this paper, we present a scheme to overcome these hurdles. Our solution works in three steps

1. We first do a semantics preserving translation of the input ABAP program to Java. We generate the Java code with built-in instrumentation, so that it outputs a trace when it is executed; this helps in overcoming the performance issues discussed above.
2. In the second step we execute the Java program with the

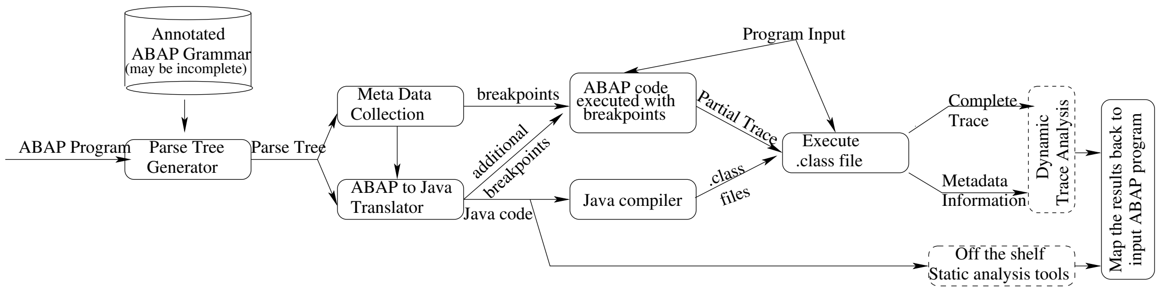


Figure 1. The block diagram for our ABAP program analysis framework

given input and collect a trace dump of the execution.
3. Our proposed framework allows plug-and-play of standard off-the-shelf static analysis techniques on the Java code and fault localization techniques on the generated trace, whose results are mapped back to the input ABAP program.

One main feature of our framework is that it decouples the source language of the input program from the actual analysis, thereby opening the doors for using arbitrary analysis tools on programs written in proprietary languages.

Translating programs written in languages like ABAP to a general purpose language like Java comes with its own set of challenges, such as development of the grammar from the scratch from language manuals, and implementation of complex database operations in the target language. To overcome these hurdles we present a scheme that a) results in the generation of correct trace even in the absence of complete grammar, and b) emulates physical database operations by gathering the results of the database operations in the proprietary debugger and plugging them back in the Java program[1]. Since we are using the debugger to extract information only about very specific commands, the process continues to be comparatively efficient. We present a specialized ABAP parser that helps identifying the statements and the corresponding defined variables requiring debugger-based trace collection. Note that, unlike the code in three address code form [2], identifying the *uses* and *defs* is non-trivial in the context of programs with database statements.

Figure 1 shows the overall block diagram of our analysis framework. Given an ABAP program, and possibly incomplete ABAP grammar, we first generate the parse tree. The parse tree generator can handle cases where valid ABAP statements are not parsed because of the incomplete nature of the available ABAP grammar. The parse tree is used to generate metadata such as data structures, type information of the required underlying system libraries, line number information for the different database related operations (to be used for setting breakpoints later), and use-def annotations on the variables used in the statements, by using our annotated ABAP grammar. Our ABAP to Java translator uses the parse tree and the data structure information from the metadata to generate equivalent Java code, and some additional breakpoints for collecting use-def information for statements that could not be translated during the translation phase. We execute the ABAP code in the debugger and collect partial trace for the breakpoints set in the previous steps. We use the generated partial trace and class files from the Java files to execute the program and generate the complete trace, which is then fed to the plugged in dynamic trace analysis tool (shown in a dashed box) such as the one by Saha et al [14]. Similarly, the geneated Java code can be fed to a plugged-in static analysis tool (also shown in a dashed box) such as Findbugs [8]. Finally, the inferences derived from the analyses are mapped back to the input ABAP programs.

The contributions of this paper are given below.
• We present a framework for program translation in the presence of incomplete grammar rules.
• We present a grammar annotation based scheme to obtain the used and defined variables for each statement. Compared to the standard techniques where the task of generating use-def information is relegated to the semantic translation phase [2], our scheme considerably reduces the development time and bugs in the process.
• A fail proof translation of ABAP code to annotated Java: We present a scheme for ABAP to Java translation which ensures that the translation rules are written in an incremental fashion for different grammar production rules, such that, unhandled grammar productions do not lead to incorrect translation.
• We let plug-and-play of off-the-shelf static and dynamic analysis tools (for Java programs) to analyze ABAP code.
• To argue about the applicability of our framework, besides plugging-in different existing tools, we have developed a new pattern-check based analysis to reason about both "good" and "bad" patterns in ABAP programs and have successfully applied it on many existing real world programs.

Organization: We present an overview of ABAP commands that are relevant to this paper in Section 2. We discuss our grammar annotation and exception handling schemes in Section 3. Challenges and details about our ABAP to Java translation are presented in Section 4. Our pattern-check based analysis is discussed in Section 5. Details of the overall framework is presented in Section 6, and that of the implementation in Section 7. We present an evaluation of our framework in Section 8 and conclude in Section 9.

[1] ABAP allows the programs to read and update the physical `screen`, which can be seen as a database table with some additional attributes.

Related Work

We are not aware of any past works, that create a translation and analysis bridge between different programming languages. We instead present related work aligned to different components of our framework.

Program Translation for reverse engineering

Source-to-source program translation approaches in support of reverse engineering and migration, may largely be classified into -

• *Translation via Transliteration and Refinement* [6, 20]: Translators in this class - first transliterate a source program into a target language on a line by line basis, by translating each line in isolation; and then apply various refinements to improve the target program produced.

• *Translation via abstraction and re-implementation* [7, 18]: In this approach the source program is first analyzed to obtain an abstract description of the computation being performed. The program is then re-implemented in the target language based on the abstract description.

These techniques do not support incremental evolution of the translator itself.In the context of evolving tools, where their coverage of the language increases in an incremental fashion, unexpected errors may be thrown when they encounter un-handled instructions. Our generic translation error handling strategy is able to meaningfully continue in the presence of translation exceptions, while ensuring trace semantics preserving translation.

Parser Error Recovery

A large body of work [1, 4, 9, 13, 17] has looked at the area of handling syntax errors and recovery in language translation systems by automatic correction of missing, or erroneous tokens. These techniques focus on continuation of the parsing process in the presence of errors to list all the possible errors. Our framework addresses the challenge of producing a trace semantic preserving translation in the presence of incomplete grammar. This, goes beyond the artificial addition of missing, or deletion of not comprehended tokens, or just continuing after the first error to locate other errors.

Pattern Analysis

There are a number of pattern detection tools, such as FindBugs [8], SPLINT [5], Flawfinder [19], MOPS [3] and so on. The analyses performed by these tools are static in nature, and suffer from the common drawback of generating false positives. In our framework we use pattern analysis tool on dynamic traces, and as a result false positives due to infeasibility of path is removed. Further, unlike the above tools that only encode patterns for 'bad' behavior, we allow encoding of both 'good' and 'bad' behavior. Further, we admit complex patterns (going beyond simple regular expressions) involving data flow relations between variables. This leads to a significant reduction in false positives. We use an Extended Finite State Automata (EFSA) to specify the desired properties (by extending the specification language of Sekar et al [16]) to describe and check good and bad behavior.

2. Background

In this section, we will be presenting a subset of ABAP language constructs relevant to this paper. Details can be found in the ABAP language reference manual [10].

`SELECT <fields> FROM tab INTO itab WHERE <cond>.` projects selected columns from a physical or internal table to an internal (in-memory) table in the program
`SORT itab BY <keys> [ASCENDING|DESCENDING].` sorts the internal table on the keys
`DELETE FROM tab WHERE <cond>.` deletes rows that satisfy the condition
`LOOP AT itab INTO rec WHERE <cond>. <loop-body> ENDLOOP.` executes the instructions in loop body for each record in the table.
`AT NEW fld. <at-body> ENDAT` / (`AT END fld. <at-body> ENDAT`) Occurs inside a loop over the records of a table. Equivalent to an `if`-statement, whose predicate evaluates to `true`, if the current record is (not) *fresh* with respect to the *criterion* fields therein; starting from the first field of the record to till `fld`.
`READ itab INTO rec WHERE <cond>.` selects a row from table based on the `WHERE` clause. If more than one row matches, the last row is returned
`WRITE <vars>.` prints the specified variables

Figure 2. Basic ABAP syntax

Figure 2 presents a subset of ABAP statements for the database related operations. While the actual ABAP language commands are more involved and have many variations, for the sake of presentation in this paper, we restrict ourselves to this subset. Each command in ABAP terminates with '.' (dot). Fields of a record are dereferenced by using the operator '−'.

```
SELECT uid name price FROM dbtab INTO itab.
SELECT uid discount FROM dbftab INTO ftab.
SORT itab BY uid DESCENDING.
LOOP AT itab INTO wa WHERE price > 0.
  AT NEW uid.
    sum = 0.0;
  ENDAT.
  sum = sum + wa-price;
  READ ftab INTO fa WHERE uid = wa-uid.
   IF NOT IS INITIAL fa.
     sum = sum - fa-discount.
  ENDIF.
  AT END uid.
    WRITE fa-name, sum.
  ENDAT.
ENDLOOP.
```

Figure 3. Sample ABAP program

To help understand the ABAP syntax, in Figure 3, we show an example ABAP program processing item discounts. It first reads `uid`, `name`, and `price` from a physical table into an internal table `itab`. It then reads the discount information from another physical database and stores relevant information into another internal table `ftab`. It then sorts the item table (`itab`) and iterates over it. For each *fresh* record `wa`, it initializes the variable `sum` to 0. The report then adds the price of that item to `sum`, reads the discount information and subtracts the discount from the `sum`. Finally, the report prints to the screen the name of the item and the total cost (after discount), once for each unique item.

3. ABAP Grammar for Java translation

Program analysis tools for domain specific complex programming languages face two main challenges. First one is that of parsing and translating the complete program (irrespective of availability of the complete grammar). The second one (important for dynamic trace analysis) is that for each statement the trace must include the variables used and defined in that statement. In the context of traditional compilers that deal with some form of three address codes, identifying the variables used and the variables defined in a statement is trivial. However, in a language like ABAP which includes many macro-level statements (that update multiple variables present at different positions in the statement), identifying the use-def variables is a challenge. In this section, we present our approach to solve these problems.

3.1 Exception Handling in Grammar

Unavailability of complete grammar forms a major hurdle in building program analysis tools. Most of the language documentation is available in the reference manual. However, this tends to become incomplete [11] as the language evolves in various versions of the software; our experience was similar. While grammar inference techniques such as [15] can be used to fix the grammar automatically to some extent, the completeness of the grammar obtained is guaranteed to be limited by the sample set used to learn the grammar. Thus for an unknown program which has no syntactic errors, the existing grammar may be incomplete to parse the program.

As shown in Figure 1, parsing a given program and generating the parse tree is the first step of our framework, which makes the absence of complete grammar a severe impediment. In this paper, we present an exception handling strategy to overcome the above problems.

Given a program, the parse-tree-generator has a preprocessing stage to collect all the included files, and parse each file. If the file is not parsed, then an exception handling strategy is invoked. The main goal of the exception handling strategy is to determine the statements (ending with '.' (dot)) in the file that are responsible for the unsuccessful parsing, and subsequently replace each such statement with a new type of statement, called `parser-error` statement that has the variables of the original statement. We explain our exception handling strategy by the help of an example.

Consider the example program given in Figure 3. Say two of the syntactic constructs, `DESCENDING` at Line 3 and `WHERE clause` at Line 4 are not handled by the grammar. Here `SORT` is a *simple* statement and `LOOP` is a *compound* statement. This program will fail to parse. We first extract each simple statement (Lines 1, 2, 3, 6, 8, 9, 11, 14), and try to parse them. Only `SORT` statement will fail to parse. We replace the sort statement by the statement '`parser_error itab, f1.`', by collecting all the terminals and removing the keywords `SORT`, `BY` and `DESCENDING`. The grammar is augmented to parse the `parse_error` statement:

```
parse_error_stmt: parse_error pe_clause? DOT ;
pe_clause: id (COMMA! id)* ;
```

The terminal `id` represents an identifier and ';' (semicolon) is used to terminate a rule.

After this change, the file is parsed again, and we still encounter a parser failure which is due to the compound statement `LOOP`. Since, all the simple statements of the loop body are parsing successfully, the loop statement is determined as the cause of the parser failure. Subsequently, its header at Line 4 is replaced by a statement '`loop_parser_error itab, wa, f2.`'. by collecting all potential variables. To be able to parse such a statement, the grammar for the particular compound statement is augmented. For instance, the modified rule for parsing the loop_statement is given below, which leads to successful parsing:

```
loop_stmt: loop_header statement+ endloop
         | loop_error statement+ endloop ;
endloop: ENDLOOP DOT ;
loop_header: LOOP AT id INTO id WHERE expr DOT ;
loop_error: LOOP_PARSER_ERROR pe_clause DOT ;
```

While the modification of the program allows it to be parsed, it does not assist the Java code generation, as the semantics of the underlying statement is still unknown. We use a process of *state synchronization* during the translation to help generate correct trace (c.f. Section 4.3).

Limitations

If a statement is not parsed by our grammar, and it is not a recognized compound statement by the grammar, then we assume it to be a simple statement, and accordingly generates the `parse_error` statement. While in general this assumption can lead to incorrect code generation, in practise we have found our strategy to be quite sound because our initial grammar did take into account all the compound statements and all the observed parse errors were coming from either missing variations for a known statement or an unknown simple statement.

3.2 Use-Def Generation

Unlike programs in imperative languages, it is not always straightforward to infer the set of defined and used variables in 4GL languages like ABAP. Thus to generate a trace in these declarative programs, trace generation algorithm needs to know the exact set of variables defined and used in each statement. In this section we present a simple yet effective way to infer the used and defined variables in each statement.

There are essentially two tasks in finding use/def information for each statement, finding all variables in the statement, and determine whether they are used or defined. Below we describe an easy yet effective process to obtain such information specifically in the context of large grammars. Our methodology is dependent on the rewrite rules which are used to modify the AST in ANTLR ([13]) grammars.

We represent the ABAP grammar in ANTLR grammar format, in Extended Backus Naur Form (EBNF) form.

ANTLR provides rewrite rules to construct Abstract Syntax Tree (AST) generated by parsing. Typical rewriting syn-

```
sort_stmt: SORT itab  sort_by_cl? sort_option? DOT
       -> ^(SORT itab sort_by_cl? sort_option? DOT)
itab: id -> ^(USEDEF id);
sort_by_cl: BY sort_by_item -> ^(BY sort_by_item)
sort_by_item: id -> ^(USE id);
sort_option:   DESCENDING |   ASCENDING;
```

Figure 4. Grammar for the SORT statement.

tax includes ^ to specify a node as parent node. For example, for this rewrite rule: '$x : yz \rightarrow \hat{}(yz)$', rewriting makes the token accepted by y as the root node, and the token accepted by z as its child node in the AST. The main usefulness of rewrite rules is the ease of traversal of the AST.

The use-def information of all statements are maintained by annotating the grammar. The annotations are determined by manually identifying the variable part of each statement, and then annotating *each occurrence* of a variable with its *ud-type*. The ud-type for a variable can either be USE, DEF, or USEDEF, to denote used, defined or used+defined nature of the variable, respectively. The annotations are made using rewrite rules in such a way that the generated AST satisfies the following invariant: *Every node corresponding to the variable of a statement appears as a leaf node of the AST, and its immediate parent node is one of the ud-type nodes (USE/DEF/USEDEF).* An example rewrite grammar for the SORT statement (Figure 2), is presented in Figure 4.

The main advantage of this technique is that the grammar rewriting followed by a general traversal is considerably easier than developing AST traversal for each possible variation in the AST generated for each statement. Further, this technique is arguably easier to maintain as the new variations in the language constructs are simple to understand.

These use-def annotations can be used by the Java translator to emit code to output the defined and used variables during the final trace collection. Another use of these annotations is found in generating code in the presence of different incompletenesses in the grammar and the translator (discussed in Section 4.3).

4. ABAP to Java translation

Translating ABAP programs to Java programs can pose interesting challenges because of the loose nature of the ABAP language semantics; for instance, it is legal to assign a string to an integer variable (provided the string contains an integer) and vice versa, so a straightforward translation would lead to uncompilable Java code. In this section, we discuss some of the challenges and our proposed solutions.

We first note that a full fledged translation would be quite challenging for the following reasons: (a) It would require implementation of a complex runtime and library implementing all the abstractions of database related activities. (b) It can be challenging to execute all the database related commands and queries more efficiently than a commercial package like SAP. (c) the issues relating to incomplete grammar and incremental development of the translator make the

```
 class BaseStruct {
   public Vector<String> names;
   ... }
class record1 extends BaseStruct{
 String fld1; String fld2;
 public record1(){
   names.add("fld1"); names.add("fld2");...}
 public String getValue(int index){
   if (index < 0) return null;
   return getValue(names.elementAt(index)); }
 public String getValue(String fldName){
   if ("fld1".equals(fldName)) return fld1;
   if ("fld2".equals(fldName)) return fld2; }...}
```

Figure 5. Code generated for a structure `record1` with two String fields `fld1` and `fld2`.

problem much harder. We now present the details of our ABAP to Java translation scheme that tries alleviate some of these problems.

4.1 Data Structures Design

The ABAP to Java translation has two aspects to it: generated Java code and the underlying libraries. We first present the architecture of our generated Java programs, and discuss some interesting aspects of the underlying libraries in the later part of the section. For each ABAP program, we generate a corresponding Java class. As discussed in in Section 2, in ABAP programs many commands update global variables (such as sy_subrc). We first create a base class BaseABAP which acts as the parent class for each of the generated Java class; BaseABAP contains declarations for the global variables, and wrappers for the different scalar types required by the ABAP programs.

In this section, we discuss the internal representation of the variables and data structures used in our generated Java programs. We organize this discussion by separating the discussion on scalars and non scalar variables.

Scalar variables In ABAP, every variable (scalar or otherwise) has five attributes: *name*, *type*, *length*, *padding-character*, and optionally the *decimal precision* for floating point numbers. These details are used in the Java program to a) serialize the contained value of a scalar, and b) compare with other scalar variables by comparing the serialized strings. The serialization of a value of a variable outputs a string of size *length*. If the actual number of characters required to represent the value are more than *length* then value is truncated, else *padding-character* is used to pad for the remaining characters. The number of decimal places in the serialized representation depend on the desired precision.

Non scalar variables Our runtime libraries are organized around the definition of BaseStruct, the base class of all the *records* and *rows* of the tables in the program. In a language like ABAP, untyped records can be assigned to each other, inserted to a table, as well as dereferenced via explicit field names or indices. Such a requirement enforces a unified structure for all the records and table rows.

To enable reflection, BaseStruct stores all the field names and all the attributes of the fields (such as length, type, padding character and decimal precision). Each structure used in the program is represented using a class extending the BaseStruct, and adds new fields in the derived class, corresponding to the fields of the structure. The BaseStruct contains two setValue (and getValue) methods that are extended by the derived classes; to help update (and read) fields of the structure using the name of the field and the index of the field. The compare method does a field wise comparison and returns *true*, if all the fields of the current object match the fields of the argument. The BaseStruct class has a method to copy the fields by value, that is useful in the definition of the Table data structure. Figure 5 shows the (partial) code generated for a structure containing two fields.

To facilitate the creation of tables of scalar types (such as Table of integers), we create a wrapper class for each of the scalar types (such as WrapperInt for the scalar type int); these wrapper classes extend the parent class BaseStruct.

For efficient translation of ABAP code to Java, we have also implemented much of the underlying ABAP library functionality in Java. These include all the string processing, data processing, numerical process operations. These translations are mostly standard and are not discussed here.

We now discuss some features of our generic Table class that is used to represent internal tables. The ABAP language allows the programmer to access the work-area of a table, that contains the current record under consideration. Our Table class contains a field WA to represent the work area. We allow three different types of tables: sorted, indexed and standard (to represent tables other than sorted and indexed). Most of the implementation details of the Table class is standard. Some specific facets are discussed now. Each element of a table is a class that extends BaseStruct and thus we have a unified mechanism to read/update rows of tables, as well as records. The Table class has a method add to insert a new row/record; the elements of the record are copied by value (unlike the usual Java Collections where the references are directly stored); this ensures that changes made to the object (after the insertion) are not reflected in the rows of the table.

Generated program structure All variables declared in the ABAP report are declared as static variables in the generated Java class. All the struct type declarations are translated to Java classes. The entry point for the generated Java code is derived from translating the code present in different events such as start-of-selection (See [10]).

4.2 Translation of simple ABAP commands

We now present the translation rules for some of the ABAP commands and overview of the overall translation scheme. Figure 6 presents some rules to translate a few of the ABAP commands. We show a typical translation rule for a database statement (SELECT) over an internal table as a library call (accessing of physical tables is discussed in Section 4.3). The DELETE, INSERT, and SORT commands are translated alike - we insert code to make appropriate transformation on the internal table (implemented in our library code for the class Table). Each ABAP statement results in one or more Java statement; each of these Java statements is annotated with the line number of the ABAP statement. Note: in our implementation, the annotations also include the file name, to take into account the multi-file ABAP program scenario.

An interesting aspect of ABAP language is that it allows numeric (integers, floating point) values to be stored in a string and accessed as desired. For instance, we can assign an integer to a string. A naive translation would lead to a type mismatch error in the generated Java code. Our ABAP to Java translator maintains and uses the type information to do a correct translation; for instance, see rules 2(a), 2(b), and 2(c). We omit the translation rules for AT NEW and AT END (handled as nested if statements), and LOOP (translated into Java loops).

Trace generation via annotated Java code Besides generating a semantically equivalent Java code, we also generate an instrumented translation, which on execution dumps the complete trace for the original ABAP program; this trace is used by the plugged-in dynamic trace analysis tool.

For each of the generate Java statement s, we emit code (after s) to output the values of all the variables that are used and defined in that statement. Such a translation helps output a forward path trace, by executing the generated Java program. The set of used and defined variables is obtained by analyzing the annotations of the AST (USE, DEF or USEDEF) for the input program.

4.3 Translation via Exception Handling

In this section, we discuss our efforts at handling incompleteness in the grammar and incompleteness in the translation, while ensuring that the resulting Java code generates a trace that matches the hypothetical trace generated by the input ABAP program. Our translator throws an exception when it encounters any of these incompletenesses. Further, in some rare situations our translator throws an exception (such as NullPointerException) because of some unhandled corner cases. There are also practical issues in our translator that leads to some bugs in the generated Java code that lead to a) compilation error or b) incorrect traces. Our translation tries to handle each of these cases. Such a scheme is also useful for developing the ABAP to Java translator in an incremental fashion wherein even the incomplete translator can be used in the framework in an effective way. We now discuss the details of these techniques.

Syntax not handled: These are the statements that are not yet handled by the translator. Instances of such commands include, physical database accesses, calls to unknown library functions (with no side effects) etc. The translator throws an exception for the code-repairer which performs *state-synchronization* to help ensure a correct trace generation.

1. Select :	`L1:SELECT <flds> FROM tab WHERE <conds> INTO itab` }	$\Longrightarrow$	`/*L1*/ itab.select(tab, <flds>, <conds>);`
2(a). Assign :	`L1 : a = e //typeOf(a) = typeOf(e)`}	$\Longrightarrow$	`/*L1*/ a = e;`
2(b). Assign :	`L1 : a = e//typeOf(a) = String, typeOf(e) = int`}	$\Longrightarrow$	`/*L1*/ a = ""+e`
2(c). Assign :	`L1 : a = e//typeOf(a) = int, typeOf(e) = String`}	$\Longrightarrow$	`/*L1*/ a = Integer.toInteger(e)`

Figure 6. Rules to translate ABAP programs to Java. `tab` is an internal table in Rule 1.

The state-synchronization happens at two levels: 1) at the ABAP to Java translator side and 2) at the debugger side which creates a *partial trace*. The translator side steps are as follows: a) Roll back any code that might have been generated by the translator for the statement under consideration. b) Extract the *DEF* variables from the parse tree of the statement. c) Create a breakpoint in the source program, after the statement under consideration. d) In case the statement is a conditional/compound statement create a *predicate-marker* for the statement and a *target-marker* for each of the target statements. e) Generate code to read the values of the *DEF* variables from the partial trace. f) If the statement is a conditional/compound statement then generate code to read the target-marker from the partial trace and transfer the control to the statement corresponding to the target-marker.

At the debugger side, we create a partial trace file: For each breakpoint created by the translator the debugger stops to collect the values of the marked variables. Further, for conditional/compound statements it uses the introduced *target-marker* to note the statement that was executed after the statement marked with the *predicate-marker*.

We present a minor optimization for unhandled loop statements, whereby the number of lookups at the time of the execution of the Java code to the partial-trace are co-located. We create breakpoints for the `LOOP` and the `ENDLOOP` statements (which are executed for each iteration of the loop) to collect the partial trace in the debugger. The looping condition in the generate Java code is decided based on the total iteration count, which in turn is computed from number of contiguous patterns of the form `LOOP` <loop-body> `ENDLOOP` in the partial trace, for the particular instance of invocation. A limitation of state-synchronization approach is that it cannot handle statements with implicit side effects.

Parser Flagged Exception: These are handled exactly similar to the previous case of *syntax not handled*. Note that the parser exception handling also specifies the type of marker statement as - normal, conditional, or loop.

Exception while translating a statement: We encountered these when because of our limited knowledge about the language, certain variation of an ABAP statement were not handled in the translation. These exceptions are also handled similar to the case of *syntax not handled*.

Generated Java code does not compile: We maintain a mapping between the line numbers of the statements of the ABAP code and the generated Java code. On a compilation error, we identify the source statement in ABAP and invoke the state-synchronization method on that statement and repeat the process.

Incorrect Execution of the Translated Code: Considering the complexity of the source ABAP language and large coding effort involved in writing the complete framework, we do not give any formal guarantees about the correctness of translation. In lieu of the formal guarantee, our technique samples the generated trace statements against the actual execution. The sample size is determined as a threshold percentage over the whole code and the statements there are marked with a *verification marker*. For a statement carrying a verification marker, apart from the normal translation the translator does the following:

• flags this statement to be included in partial trace collection for both its use and def variables

• inserts an assert statement after the translated statement which compares the actual values of the def variables of the statement with the ones collected in the partial trace.

5. Pattern Checking

Identifying patterns in the code can be considered as the most effective and scalable technique for finding common bugs [8]. In this context we have built an automatic pattern detection engine for ABAP language. The important feature of our pattern detector is that it works on a given execution trace, instead of static code. This helps in a) allowing runtime patterns with actual values, and b) finding out the exact problem causing the buggy behavior, which is important from the ticket resolution perspective. However, it reduces the scope of pattern checking to only parts of the code that have been exercised by the trace. This pattern checker can be used as a plugin in our proposed framework.

Pattern Specification We use Extended Finite State Automata (EFSA) to describe patterns. Informally, EFSA extends finite state automata with state variables in states, and constraints and assignments in state transitions. The transition is only enabled if the constraints are satisfied, and then assignments are used to assign values to the state variables.

Formally, a Pattern-EFSA is an 9-tuple $\mathcal{E} = (Q, Q_0, F, X, E, G, A, T, D)$, where

• Q is a finite set of *states* ∪ the set of *variables* X,

• $Q_0 \subseteq Q$ is the set of *start states*

• $F = \{\texttt{Good}, \texttt{Bad}\} \subseteq Q$ is the set of *final states*.

• E is the set of *events*, each event is associated with a set of attributes. *Attr* = set of all attributes over all the events.

• G is a set of *boolean conditions/guards* over the elements in X and all possible attributes of an event; the guards can use constant value, user inputs, and user-defined functions.

• A is a set of *assignment* statements which are of the form

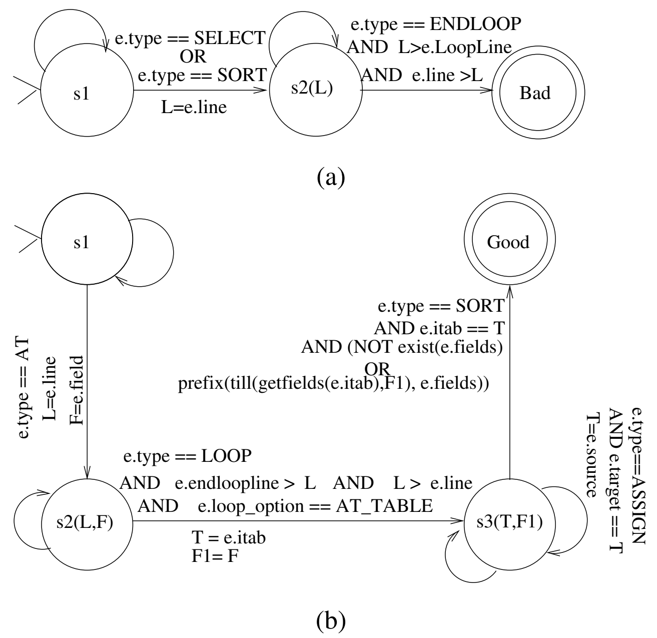


Figure 7. Sample EFSA: (a) InsideLOOP (b) SORT-AT

$x := y$, where $x \in X \cup Attr$, and $y \in X$.

• $T \subseteq Q \times (E \times G \times A) \times Q$ is a set of *transitions*. A transition $(q, (e, g, a), q')$ is *enabled* if in state q the event $e \in E$ is available and the boolean condition $g \in G$ is satisfied. When the transition is taken the assignment statements in a are executed. Additionally, a transition may be associated with a tag *non-vacuous* (discussed later).

• D is a direction property of the pattern which signifies whether the sequence of events should represent a trace from start to end (D=`forward`) or end to start (D=`backward`).

An execution trace is represented as a sequence of events (an event for each statement), each having a set of attribute-value pairs (for instance, `type` attribute of the sort statement is SORT) . Each attribute represents a static/dynamic property of a statement executed by the trace. We term a sequence of events as *good-accepted* (or *bad-accepted*) by an EFSA if it corresponds to a sequence of enabled transitions from the start state to the final `Good` (or `Bad`) state, leading to a a possible good (erroneous) execution.

We present two examples of pattern-EFSAs in Figure 7. The InsideLOOP Pattern-EFSA is an example of a forward pattern that represents a bad behavior of the trace. The InsideLOOP pattern represents a policy which states that having a *select and sort statement inside a loop is a bad behavior*, as it may lead to performance issues. The transition from state `s1` to state `s2`, first checks whether the input event has a type attribute having value SELECT or SORT. The transition then records the line number (given by the attribute `e.line`) of the candidate statement in a state variable `L` in state `s2`. The presence of a loop statement around the candidate statement can be identified by comparing the line numbers of the ENDLOOP and it's corresponding LOOP header. (the check for same filename is omitted for brevity). The self-loop transitions do not alter the values of the state variables and are used to consume the events unrelated to the current pattern. Note that, the `type`, `line`, `LoopLine` attributes are all static attributes of the statements that are collected by traversal of the parse tree in *meta-data collection* phase (See Figure 1). Another pattern that accounts of erroneous behaviour is that of deleting rows from a table in a loop that iterates over the rows of that table.

If a trace is *bad-accepted* by a pattern, then the trace contains possible bad behaviors, and all the events in the input sequence which are enabled by the transitions are highlighted to the user as a possible fault. For instance, if a trace is *bad-accepted* by the InsideLOOP Pattern-EFSA, then the sort statement and loop statement are highlighted.

Consider the SORT-AT pattern-EFSA presented in Figure 7(b). The pattern represents a policy which states that, AT statement on a table should be preceded by a sort statement on that table, and the table should be sorted on all the (implicit and explicit) fields on which the AT check is done. As discussed in Section 2, the semantics of the `AT` statement states that along with the field `f` mentioned in `AT`, any change in values of the fields that occur before `f` in the declaration of the table also makes the condition to be true.

As the pattern check to be done from existence of AT statement first, and then existence of SORT statement, the D property of the pattern has value `backward`. We now describe some of the transitions.

• $s2 \rightarrow s3$: Finds an enclosing `LOOP` statement that loops over table rows (checked by loop_option = AT_TABLE). The name of the table and the fieldname are passed on to $s3$.

• $s3 \rightarrow$ `Good`: We find a sort statement on the same table on which `AT` check was done. The predicate ensures that the table is sorted on the fields on which `AT` check was done in the same order: if the sort statement mentions no field names, then the sort is done on all the fields in the order that exist in table definition and thus needs no further checks. In case the field names are mentioned in the sort statement, we use a helper function `getfields` to obtain the fields from the table declaration and compare with the implicit fields on which `AT` check was done (using function `till`).

• $s3 \rightarrow s3$: This transition enables us to capture the propagation of table names through assignments.

If a pattern is expressed with good final states, then the non-acceptance of a sequence of events may denote a possible bad behavior. However, it is possible that a non-acceptance of a sequence happened without enabling any "relevant" transitions (such as $s1 \rightarrow s2$ or other words without finding any `AT` statement). And that sequence should not be described as a bad behavior based on this pattern. To handle such cases, our pattern language allows certain transitions to be tagged non-vacuous in pattern-EFSAs that have a `Good` final state. This tagging is done by the EFSA writer. Non-acceptance of a trace along with enabling of non-vacuous transitions represents bad behavior.

The examples illustrated so far present cases where checks are performed on the static properties of statement. We now present some possible patterns on the dynamic properties (all with direction property D is `forward`).

• Unsuccessful db commands: In ABAP, execution of a database commands sets a status variable called sy-subrc. A non-zero value of sy-subrc denotes an unsuccessful database commands. The policy checks for non-zero value of sy-subrc after execution of a database command, before the updated records/tables are used.

• Large Internal Tables: One of the common problem noticed in ABAP is use of insufficient keys in select statement which returns more than expected number of rows. The policy checks for the length of the internal table exceeding an user input value.

• Overflow: After an assignment statement like `a = b`, whether the value of a is equal to b. This is not the case if length of b is greater than that of a and truncation of value occurs. This is important if the sizes of `a` and `b` are different.

• Missing check: Given a predefined set of key-value pairs whether any deletion was performed on a table which deleted the rows associated with the given set of key-value pairs; useful in finding missing entries in report output.

6. Rubber meets the Road

6.1 Uses of the trace

Trace for debugging input program Our framework, allows that arbitrary trace analysis tools can be plugged in. For instance, we have plugged-in the fault-detection tool of Saha et al [14] in our framework to derive expected results. The main requirement for plugin to work with our framework is that the plugin must conform to the interface requirements of our framework. We now discuss the details of the interface.

Our framework presents the details of the trace in two different files. The first file contains the variable and type information. For each variable used in the program, we store the name of the variable, scope (function name, class name or global), and the type details: an unique type id, structure of the type (scalar or *struct* or *table*), the declared type and the Java type. A struct type has additional attributes for the fields of the structure. A table type has additional attributes for the type of the table (standard, sorted, indexed) and the type of the rows of the table.

The second file is the complete trace file and is used to interface with the trace analysis tools. which contains the use-def data for each program statement. Each line in the trace has two identifiers: a) static line number, and b) the dynamic sequence number of this statement during the execution. Each trace line gives the details of all the variables and literals referred (used and/or defined) in the statement.

Trace for debugging the translation As discussed in Section 1, a naive way to generate the dynamic trace is to debug a program by stepping-into each statement, and use a screen-scrapping methodology to collect the use-def information

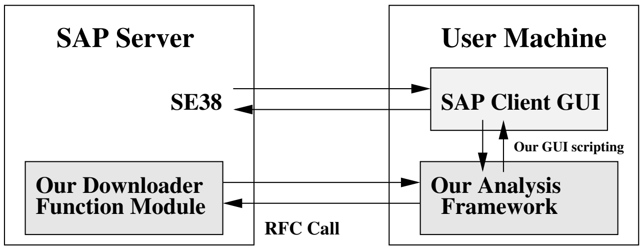


Figure 9. Interaction with SAP components

at each program point. We use this scheme (time consuming nevertheless) to establish the correctness of our translation. We generated the complete trace once using the screen-scrapping method, and then use it to compare the trace from our generated Java code. The translation is considered correct if both the traces match (both in terms of the order of the executed statements and values of the DEF variables at each program point). During the course of our initial development, this approach helped us to identify some corner cases in our implementation.

6.2 Application of standard tools

For a given ABAP program and its input, our framework generates its equivalent Java code, annotated with line numbers of the original ABAP code. One main advantage of such a translation is that since the generated code is written in Java, it allows a host of program analysis techniques written for Java; these can be plugged-in to our framework to derive desired results. Instances of the program analysis tools that can be applied are FindBugs [8], Khasiana [12] and so on.

6.3 Mapping the results

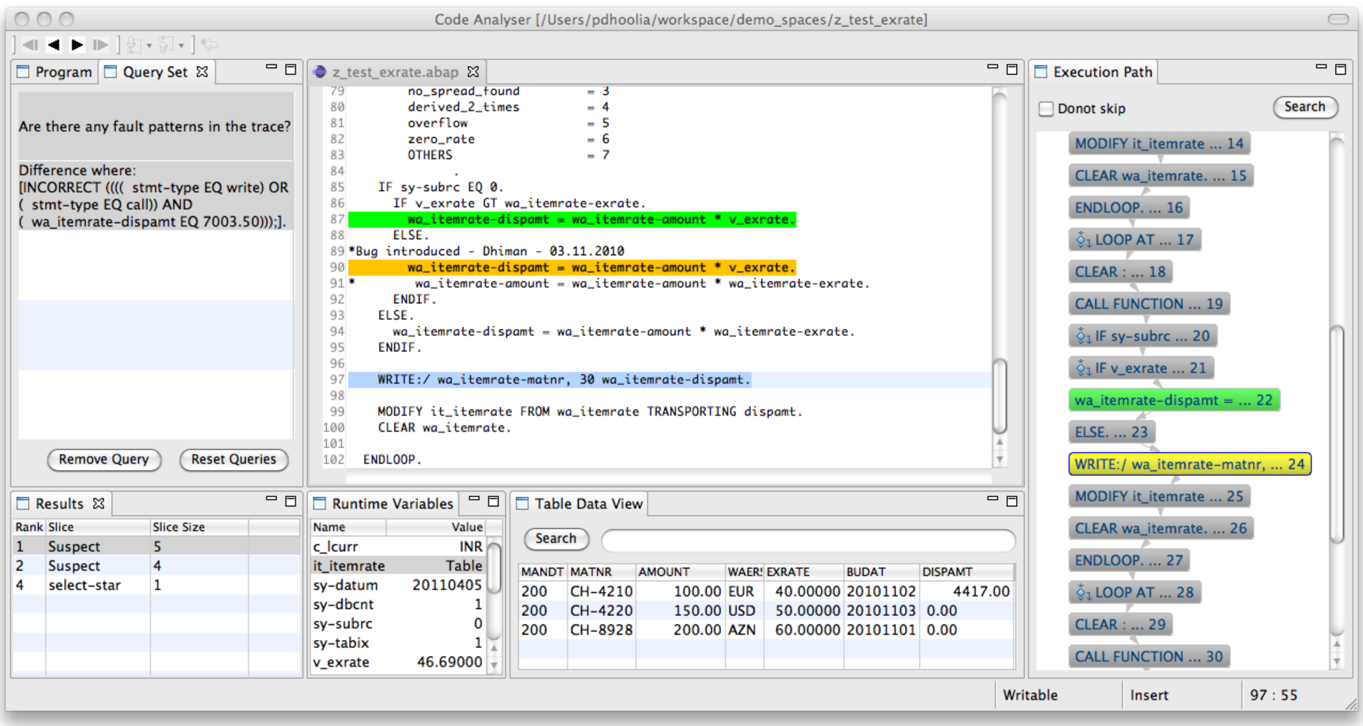
The results of all the plugged-in analyses is given in terms of the line numbers of the Java program or the line number of the trace. Since both the generated Java program as well as the dynamic trace are annotated with the line numbers of the original ABAP program, it is trivial to map the results back. We use this mapping technique to design a GUI based client (briefly discussed in the following section).

7. Implementation details

We have implemented the system illustrated in Figure 1. We now briefly discuss the implementation details.

• *Parser* - was generated for our ANTLR 3.2 based grammar for ABAP, with Java as the target. The complexity and the size of the ABAP grammar posed a unique challenge here. The compilation of the generated Java Parser ran into the ”Code too large” problem. To deal with this we devised an automatic technique to partition the grammar into separate logical chunks. We have observed that even a naive partitioning of the grammar based on a threshold on the production-rule-count in each chunk has been found quite effective.

• *Fault Patterns Analyzer* It took pattern descriptions declared using XML as input and matched them on the trace.



Are there any fault patterns in the trace?

Difference where:
[INCORRECT ((((stmt-type EQ write) OR
(stmt-type EQ call)) AND
(wa_itemrate-dispamt EQ 7003.50)));].

```
         no_spread_found          = 3
         derived_2_times          = 4
         overflow                 = 5
         zero_rate                = 6
         OTHERS                   = 7
                 .
     IF sy-subrc EQ 0.
       IF v_exrate GT wa_itemrate-exrate.
         wa_itemrate-dispamt = wa_itemrate-amount * v_exrate.
       ELSE.
*Bug introduced - Dhiman - 03.11.2010
         wa_itemrate-dispamt = wa_itemrate-amount * v_exrate.
*          wa_itemrate-amount = wa_itemrate-amount * wa_itemrate-exrate.
       ENDIF.
     ELSE.
       wa_itemrate-dispamt = wa_itemrate-amount * wa_itemrate-exrate.
     ENDIF.

     WRITE:/ wa_itemrate-matnr, 30 wa_itemrate-dispamt.

     MODIFY it_itemrate FROM wa_itemrate TRANSPORTING dispamt.
     CLEAR wa_itemrate.

   ENDLOOP.
```

Rank	Slice	Slice Size
1	Suspect	5
2	Suspect	4
4	select-star	1

Name	Value
c_lcurr	INR
it_itemrate	Table
sy-datum	20110405
sy-dbcnt	1
sy-subrc	0
sy-tabix	1
v_exrate	46.69000

MANDT	MATNR	AMOUNT	WAER!	EXRATE	BUDAT	DISPAMT
200	CH-4210	100.00	EUR	40.00000	20101102	4417.00
200	CH-4220	150.00	USD	50.00000	20101103	0.00
200	CH-8928	200.00	AZN	60.00000	20101101	0.00

```
MODIFY it_itemrate ... 14
CLEAR wa_itemrate. ... 15
ENDLOOP. ... 16
LOOP AT ... 17
CLEAR : ... 18
CALL FUNCTION ... 19
IF sy-subrc ... 20
IF v_exrate ... 21
wa_itemrate-dispamt = ... 22
ELSE. ... 23
WRITE:/ wa_itemrate-matnr, ... 24
MODIFY it_itemrate ... 25
CLEAR wa_itemrate. ... 26
ENDLOOP. ... 27
LOOP AT ... 28
CLEAR : ... 29
CALL FUNCTION ... 30
```

Figure 8. Snapshot of our ABAP Code Analysis Framework

• *GUI* - an Eclipse Rich Client Platform based GUI was implemented as the user-interface for tool. A snapshot of the tool in action is illustrated in Figure 8. This tool works as an interface to load the analysis, and displaying the results. The details of the GUI client (usability, speed, interactions etc) are beyond the scope of this submission.

• *Integration with SAP system* - Interfacing with the SAP system was done as illustrated in Figure 9. We implemented a function module in ABAP, which we upload to the SAP system, and then use via an SAP RFC call for the purpose of extracting the ABAP source, dependencies, and data-dictionary of built-in types. We use SAP Client GUI 7.10 installed on the user's machine for the purpose of collecting selective traces. Our system generates a VB script based on SAP GUI scripting language. This script automatically - (a) opens the ABAP program in SE38 (debugger) transaction; (b) puts required breakpoints on the lines indicated by the translator to be collected; (c) collects the partial trace at those breakpoints and dumps it to an XML file. An interesting issue we encountered here was that the SAP debugger transaction (SE38) puts a limitation of maximum 30 breakpoints; which was not sufficient for some of the large programs. We generated the partial trace collection for 30 breakpoints at a time and merged the final trace. In our experience this resulted only in minor additional cost.

8. Evaluation

In this section, we present our experience in using our ABAP Code Analysis Framework. We divide the presentation into three sections: experience with parser exception strategy, experience with ABAP to Java translation, and the utility of our new pattern analysis technique.

Prog	LOC	# Err	Execution Time (in seconds)		
			Parsing	LineParsing	Writing
N1	198	1	1.8	0.3	0.01
N2	207	3	2.0	0.5	0.01
N3	630	5	1.9	0.5	0.07
N4	737	41	2.1	0.3	0.1
N5	925	2	2.1	0.6	0.1
N6	932	10	2.1	0.7	0.2
N7	966	2	2.1	0.4	0.2
N8	1078	3	2.0	0.6	0.2
N9	1401	2	2.0	1.1	0.3
N10	3281	1	2.7	1.8	1.0
N11	3330	3	2.7	1.7	1.0
N12	3838	6	2.2	3.1	1.2
N13	25766	238	4.1	46.7	23.9

Figure 10. Result: Parser Exception Strategy Handling

Parser Exception Handling In our ongoing project, we started with an initial version of the ABAP grammar derived from the online language manual (`http://help.sap.com`). For reasons discussed before, that grammar was not able to parse the complete set of benchmarks we had at hand. But our analysis tool development and testing continued even in the presence of those unparsed statements. Eventually the grammar evolved to parse all our benchmarks. To test the effectiveness of the exception strategy handling we therefore chose a new set of ABAP programs that we had got recently. In Figure 10, we present the evaluation of our current grammar with respect to a subset of these programs that could not be parsed with our existing grammar and the grammar exception handling strategy was used used derive analyzable parse trees. For each benchmark we show, the number of lines of code (LOC), number of parse errors (# Err), and the breakup of the time taken by the our grammar exception handling strategy. The figure shows the time it took the parse the whole program ("Parsing"), time it took for parsing individual lines to identify and extract exact details for parsing failures ("LineParsing"), and the time taken to rewrite the updated file with relevant parse error statements. The total time taken went upto 75 seconds (for a large program with 25K lines of code), and rest all programs took just a few seconds. Overall, we conclude that a) there is a need for grammar exception handling strategy, and b) the overhead incurred by our proposed methodology is reasonable.

ABAP to Java translation To evaluate the coverage and correctness of our ABAP to Java translation scheme, we designed 165 unit testcases to cover all our known variations of ABAP. We compared the generated trace with the ideal trace to gain confidence on our translation.

For evaluating the efficacy of our ABAP to Java translation, we use a set of heavily used proprietary ABAP programs provided by our development teams; these are different from the newly obtained program benchmarks used earlier in this section. Some characteristics of these benchmarks can be found in the first two columns of Figure 11. We have used two sets of benchmarks - the first set is one from real applications, and the second set consists of all synthetic benchmarks. The third column lists the number of breakpoints that are used by the debugger mode partial trace collection. The performance benefits of using the our scheme compared to the complete trace obtained via the debugger are shown in the columns 4-6. It shows that our approach may result up to `11x` speed ups.

Two observations: a) our approach may not yield benefit when the programs are trivially small which do not offset the overhead of our approach. b) For larger programs, the gains from our approach offsets the overheads we incur.

We have also used the FindBugs [8] tool on our generated Java code. It did correctly identify some simple warnings (such as unused variables, unused computation etc), but it did not find any programming bugs like null dereferences etc. This is quite expected as these programs are well tested and used in the industry for a while.

Pattern Analysis We encoded several static/dynamic patterns and run against traces of several programs. The results are shown in Figure 12. We give a short account of all the patterns in Figure 13. These patterns are suggested to us by ABAP practitioners. Only the pattern A8 has direction value D to be `forward`. To evaluate these patterns, the bugs were manually seeded in the benchmarks.

9. Conclusion

In this paper we shared our experience of building a framework for finding faults in ABAP programs in the context of ticket resolution. Due to the limited availability of debugging tools for localizing faults in such programs, we have taken the approach of generating equivalent translation to annotated Java programs, and executing it to generate the equivalent trace, such that we can use static and dynamic analysis tools for fault localization.

We present a new grammar annotation mechanism, along with different exception handling schemes to handle incompleteness in the grammar, and the translation. We observed that our proposed framework was able to cover all the benchmarks provided to us and the incurred performance overhead was kept to an acceptable limit. We have further developed and used a pattern matching tool which finds matching patterns in the generated traces. The use of both static and dynamic properties makes it particularly useful for fault localization in ABAP language.

Prog	dbg mode time (s)	num lines	num breakpts	a2j mode time (s)	impr
B1	4854	393	2517	446	5.64 x
B2	860	235	355	183	1.94 x
B3	15746	900	1817	633	2.87 x
B4	1484	426	369	548	2.70 x
B5	4189	421	3298	576	5.73 x
B6	2542	530	587	194	3.03 x
B7	1694	427	5115	486	11.98 x
B8	1066	186	1424	136	10.41 x
S1	48	13	84	21	4.00 x
S2	48	13	26	19	1.37 x
S3	43	18	35	22	1.59 x
S4	49	24	43	22	1.95 x
S5	67	15	26	21	1.24 x
S6	86	16	32	60	0.53 x
S7	85	26	66	32	2.06 x
S8	35	11	14	67	0.21 x
S9	89	22	29	36	0.81 x
S10	133	36	470	97	4.85 x

Figure 11. ABAP to Java translation: evaluation

Pattern	Description
A1	SORT-AT pattern described in Section 5
A2	InsideLOOP pattern described in Section 5
A3	Similar to SORT-AT; field names existed in On-Change are compared with the fields in SORT
A4	Delete adjacent should be performed on same field names as sort for the same table
A5	Does not use select * or provide * syntax
A6	Is initial check needs to be done on a variable before it is used in forall clause in select statement
A7	read with binary search option should use the same keys on which the table is sorted
A8	database commands be followed by sy-subrc check

Figure 13. Sample Dynamic Patterns

Acknowledgements: The authors would like to thank Vibha Sinha, Satish Chandra, Sugata Ghosal, Sourin Ghosh, Dhiman Ghosh, Asidhara Lahiri, and Amitava Parui for many fruitful discussions and help at different stages of the project.

Prog	Static Patterns								Time	Dynamic Patterns				Time
	A1	A2	A3	A4	A5	A6	A7	A8	(sec)	unsucc-db	overflow	bigtbl	missing	(Sec.)
B1	58/1	0	3/2	0	4/3	1/1	0	30/2	5.7	1/1	0	0	1/1	3.0
B2	0	0	0	0	0	3/3	1/1	0	0.1	0	0	0	-	1.5
B3	0	0	0	0	0	2/2	0	0	0.7	22/4	0	0	-	3.0
B4	0	0	0	0	3/2	0	0	3/2	0.1	0	0	0	-	0.1
B5	0	62/2	0	0	1/1	0	0	5/5	3.5	0	0	0	1/1	9.0
B6	0	0	0	0	0	0	0	0	1.2	0	0	0	-	0.1
B7	166/2	0	0	0	1/1	0	0	86/4	60.0	0	0	0	-	6.0
B8	0	0	0	0	0	0	0	1/1	1.3	84/1	84/1	0	-	0.1

Figure 12. Static and Dynamic Pattern Analysis: statement instances/statements

References

[1] Java compiler compiler (JavaCC): The Java parser generator. http://javacc.java.net/.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *CCS*, pages 235–244, New York, NY, USA, 2002. ACM.

[4] Julia Dain. Bibliography on syntax error handling in language translation systems.

[5] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19:42–51, Jan 2002.

[6] S.I. Feldman. A Fortran to C converter. In *ACM SIGPLAN Fortran Forum*, volume 9, pages 21–22. ACM, 1990.

[7] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.M. Jézéquel. Model-driven engineering for software migration in a large industrial context. *MDELS*, pages 482–497, 2007.

[8] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, pages 132–136. ACM, 2004.

[9] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, 1975.

[10] Horst Keller. *The official ABAP reference*. SAP Press, 2004.

[11] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Softw. Pract. Exper.*, 31(15):1395–1448, 2001.

[12] M G Nanda and S Sinha. Accurate interprocedural null-dereference analysis for java. In *ICSE*, pages 133–143, Washington, DC, USA, 2009. IEEE Computer Society.

[13] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[14] D. Saha, M.G. Nanda, P. Dhoolia, V.K. Nandivada, V. Sinha, and S. Chandra. Fault localization in ABAP Programs. In *FSE*, 2011.

[15] D. Saha and V. Narula. Gramin: a system for incremental learning of programming language grammars. In *ISEC*, pages 185–194. ACM, 2011.

[16] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *SOSP*, pages 15–28, 2003.

[17] P.N. van den Bosch. A bibliography on syntax error handling in context free languages. *ACM SIGPLAN Notices*, 27(4):77–86, 1992.

[18] R.C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, pages 1207–1228, 1988.

[19] David A Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder/`.

[20] K. Yasumatsu. SPiCE: a system for translating Smalltalk programs into a C environment. *Software Engineering, IEEE Transactions on*, 21(11):902–912, 1995.