Program Specialization and Verification using File Format Specifications

Raveendra Kumar Medicherla
TCS Limited, Bangalore, India
Indian Institute of Science, Bangalore, India
raveendra.kumar@tcs.com

Raghavan Komondoor S. Narendran
Indian Institute of Science, Bangalore, India
{raghavan,narendran}@csa.iisc.ernet.in

Abstract—**Programs that process data that reside in files are widely used in varied domains, such as banking, healthcare, and web-traffic analysis. Precise static analysis of these programs in the context of software transformation and verification tasks is a challenging problem. Our key insight is that static analysis of file-processing programs can be made more useful if knowledge of the input file formats of these programs is made available to the analysis. We instantiate this idea to solve two practical problems – specializing the code of a program to a given "restricted" input file format, and verifying if a program "conforms" to a given input file format. We then discuss an implementation of our approach, and also empirical results on a set of real and realistic programs. The results are very encouraging in the terms of both scalability as well as precision of the approach.**

I. INTRODUCTION

Processing data that resides in files or documents is a central aspect of computing in many organizations and enterprises. Standard *file formats* or *document formats* have been developed or evolved in various domains to facilitate storage and interchange of data, e.g., in banking [13], [16], health-care [23], enterprise-resource planning (ERP) [40], billing [19], and web-traffic analysis [5]. The wide adoption of such standard formats has led to extensive development of software that reads, processes, and writes data in these formats. However, there is a lack of tool support for developers working in these domains that specifically targets the idioms commonly present in file-processing programs. The primary contribution of this paper is a family of dataflow analysis approaches, all of which leverage a given input file format specification. These approaches target the problems of program specialization, file-format conformance verification, and finally, of improving the usefulness of any existing static analysis approach.

A. Motivating Example

Our work has been motivated in particular by *batch programs* in the context of enterprise legacy systems. Such programs are typically executed periodically, and in each run process an input file that contains "transaction" records that have accumulated since the last run. In order to motivate the challenges in analyzing file-processing programs, we introduce as a running example a small batch program, as well as a sample file that it is meant to process, in Figure 1.

1) Input file format: Although our example is a toy example, the sample file shown in Figure 1(b) adheres to a simplified version of a real banking format [16]. Each record is shown as a row, with fields being demarcated by vertical lines. In this file format, the records are grouped logically into "batches", with each batch representing a group of "payments" from one customer to other customers. Each batch consists of a "header" record (value 'HDR' in the first field), which contains information about the paying customer, followed by one or more "item" or "payment" records ('ITM' in the first field), which identify the recipients, followed by a "trailer" record ('TRL'). There are two such batches in the input file in Figure 1(b). Figure 1(c) gives the names of the fields of header as well as item records. Other than the first field `typ`, which we have discussed above, another field of particular relevance to our discussions is the `src` field in header records, which identifies whether the paying customer is a customer of the bank that is running the program ('SAME'), or of a different bank ('DIFF'). The meanings of the other fields are explained below part (b) and (c) of the figure.

2) The code: Figure 1(a) shows our example program, which is in a Cobol-like syntax. The "`DATA DIVISION`" contains the declarations of the variables used in the program, including the input file buffer `in-rec` and output file buffer `out-rec`. `in-rec` is basically an overlay (or *union*, following the terminology of the C language), of the two record layouts shown in Figure 1(c). After any record is read into this buffer the program interprets its contents using the appropriate layout based on the value of the `typ` field. The output buffer `out-rec` is assumed to have fields `pyr`, `rcv`, `amt`, as well other fields that are not relevant to our discussions. These field declarations have been elided in the figure for brevity.

The statements of the program appear within the "`PROCEDURE DIVISION`". The program has a *main loop*, in lines 3-26. A record is read from the input file first outside the main loop (line 2), and then once at the end of each iteration of the loop (line 25). In each iteration the most recent record that was read is processed according to whether it is a header record (lines 12-19), item record (lines 4-11), or trailer record (lines 20-21). The sole `WRITE` statement in the program is in the item-record processing block (line 11), and writes out a "processed" item record using information in the current item record as well as in the previously seen header record. Lines 7

```
DATA DIVISION.
INPUT FILE in-file    BUFFER in-rec.
OUTPUT FILE out-file  BUFFER out-rec.
/a/ char  same-flag = ‘ ’.
/b/ digit eof-flag = 0.
PROCEDURE DIVISION.
/1/  OPEN in-file, out-file
/2/  READ in-file INTO in-rec, AT END MOVE 1  TO eof-flag
/3/  WHILE eof-flag = 0
/4/   IF in-rec.typ = ‘ITM’ //Item record processing
/5/      MOVE in-rec.rcv, in-rec.amt TO out-rec.rcv, out-rec.amt
/6/      IF same-flag = ‘S’
/7/         Itm record processing for SAME batch header
/8/      ELSE
/9/         Itm record processing for DIFF batch header
/10/     END-IF
/11/     WRITE out-file FROM out-rec
/12/  ELSE IF in-rec.typ ='HDR' //Header Record Processing
/13/     MOVE in-rec.pyr TO out-rec.pyr
/14/     IF in-rec.src = ‘SAME'
/15/          MOVE ‘S’          TO same-flag
/16/     ELSE
/17/          MOVE ‘D’          TO same-flag
/18/     END-IF
/19/     Rest of header record processing
/20/  ELSE IF in-rec.typ =‘TRL' //Trailer Record Processing
/21/        TrL record processing
/22/  ELSE
/23/        Terminate program with error
/24/  END-IF
/25/  READ in-file INTO pmt-record, AT END MOVE 1  TO eof-flag
/26/ END-WHILE.
/27/ CLOSE in-file,out-file.
/28/ GOBACK.
```

(a)

HDR	10205	9000	SAME
ITM	10201	3000	
ITM	10103	4000	
ITM	18888	2000	
TRL			
HDR	20221	6000	DIFF
ITM	19999	2000	
ITM	10234	4000	
TRL			

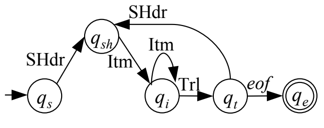
(b)

typ	pyr	tot	src
HDR	10205	9000	SAME

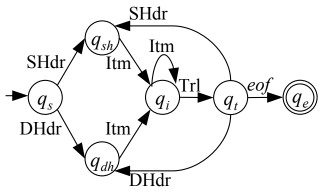
typ	rcv	amt
ITM	10201	3000

(c)

typ: *main type.* pyr: *payer account number.* tot: *total batch amount.* src: *source bank.* rcv: *receiver account number.* amt: *item amount.*



(d)



(e)

Record Type	Constraint
SHdr	typ = ‘HDR’$\wedge$ src = ‘SAME’
DHdr	typ = ‘HDR’$\wedge$ src = ‘DIFF’
Itm	typ = ‘ITM’
Trl	typ = ‘TRL’

(f)

Fig. 1. (a) Example program (b) Sample input file (c) Input file record layouts (d) Specialization automaton (e) Well-formed automaton (f) Input record types

and 9 represent code (details elided) that populates certain fields of out-rec in distinct ways depending on whether the paying customer is from the same bank or from a different bank.

B. Program Specialization

Program specialization is the activity of modifying a program to restrict its functionality to a subset of all possible inputs, as specified via a *condition* or *constraint* on the inputs to the program. In this paper, our notion of specialization of a program is to *delete* program statements that are irrelevant to the given input constraint. Numerous applications of program specialization have been investigated in the software engineering literature over a long period of time. For instance, it can be used to untangle conceptually distinct but interleaved functionalities from monolithic programs, and to support programmer comprehension of these functionalities [2], [35].

It can be used to support (forward) conditioned program slicing [17], [4], [11], which involves obtaining a slice of a program from a set of selected output variables under a given constraint on the program's input, which has been shown to have many applications in software maintenance and evolution tasks.

The novel aspect of our problem is that we address specialization of file-processing programs, wherein the input constraint is a restriction on the sequences of records that may appear in the input file to a program. For instance, in our running example, we might be interested only in the parts of the program that are reached when input files contain batches whose header records always have ‘SAME’ in the src field; these parts constitute all the lines in the program except lines 9, 17, and 23. Line 17 cannot be reached because in-rec.src is constrained to only have the value ‘SAME’. Line 9 cannot be reached, because (a) line 17, which assigns ‘D’ to same-flag is unreachable, and (b) the initialization of same-flag to ' ' (before the procedure division) cannot reach line 6, because an item record cannot be seen in the input file before seeing a header record first.

File-format specifications, which are usually readily available because they are organization-wide or even industry-wide standards, have been used by previous programming languages researchers in the context of tasks such as parser and validator generation [19], and white-box testing [21]. Our proposal is to use a file-format specification, represented as a finite-state *input automaton*, whose transitions are labeled with record types, as a specialization constraint. In this setting, we call an input automaton a *specialization automaton*. For instance, Figure 1(d) depicts a specialization automaton corresponding to our running example. This specialization automaton describes the (restricted) file-format wherein input files contain batches whose header records always have ‘SAME’ in the src field. Figure 1(f) shows the associated input record-type definitions as *dependent* types [41] (for time being ignore the record type DHdr in table). Intuitively, a record is of a certain type if the record's fields satisfy the constraint that constitutes the type's definition. Thus, the first record in the file in Figure 1(b) is of type SHdr, the second record in this file is of type Itm, etc.

A key observation we make is the need for an automaton-like specification of the constraint, which distinguishes our work from previous program specialization approaches, which primarily target programs that accept a fixed-size input at the beginning of the program. For instance, in the specialization scenario mentioned above, it does not suffice to say that the input file contains only Itm, Trl, and SHdr records (in any order). The constraint that an SHdr record precedes any Itm record also needs to be captured, else the initial value of `same-flag` (i.e., `' '`) will appear to reach line 6, which leads to an imprecise specialization (namely, that line 9 gets included in the specialized program). In general, capturing the ordering among the record types is a natural pre-requisite to precise program specialization.

It is noteworthy that program specialization using our approach can be followed by slicing, which results in a form of (forward) conditioned slicing (which we call *specialized* slicing). We explore this application further in Sections V and VI.

C. File Format Conformance Checking

A "well-formed" input file for a program is an input file that users of a program expect it to process. An input automaton can be used also to specify the format of all well-formed input files to a program, in which case we call it a *well-formed automaton*. For instance, the input automaton depicted in Figure 1(e) is a well-formed automaton in the context of our running example.

The second novel problem that we address in this paper is of verifying, using static analysis, whether a file-processing program may (a) "reject" a well-formed input file with a warning that it is ill-formed, or (b) silently "accept" an ill-formed input file without a warning, which may potentially lead to incorrect outputs (or corrupted persistent tables). For instance, in our running example program, control can never reach the warning in line 23 during executions well-formed input file. However, this program actually accepts ill-formed files that contain an Itm record as the first record; in this scenario, the program writes out an output record in line 11 whose `pyr` field is not yet set, and hence contains garbage (this field is set in line 13, which is executed only when a header record is seen). In this example, the programmer ought to have included error-handling code that checked for the well-formedness of the given input file.

D. Our Contributions

The primary contribution of this paper is a family of sound static analysis approaches to solve transformation and verification problems in the domain of file-processing programs. In Section III of this paper we present the first approach in this family, which is for specializing a program based on a constraint that is expressed as a specialization automaton. In Section IV we present the second approach, which is to verify conservatively whether a program rejects well-formed files or accepts ill-formed files, based on a input automaton that captures the format of all well-formed files. Then, in Section V we sketch a generalization of the two approaches mentioned above that enables *any* existing static analysis approach to leverage a given input automaton during analysis of a file-processing program to compute more useful results.

We have implemented our approach, and applied it on several real as well as realistic Cobol batch programs. We found that our specialization approach was surprisingly precise. We also found that specialized slices produced using our approach were significantly smaller than corresponding unconditioned slices. Our file-format conformance checking approach found genuine conformance issues in several programs, was also able to verify the absence of such errors in other programs. We describe our implementation as well as experimental results in Section VI of this paper.

Finally, Section VII discusses related work, while Section VIII concludes the paper.

II. Assumptions and Definitions

Definitions (Records and record types): A *record* is a contiguous sequence of bytes in a file. A *field* is a labeled non-empty sub-string of a record. A *record type* R_i is a specification of the length of a record, the names of its fields and their lengths, and a *constraint* on the contents of the record. We say that a record r *is of type* R_i iff r satisfies the length as well as value constraints of type R_i.

Definitions (Files and "read" operations): A file is a sequence of records, of possibly different lengths. Successive records in a file are assumed to be demarcated explicitly. A `READ` statement in a program, upon execution, retrieves the next so-far unread record from the input file, and copies it into the file buffer associated with this file in the program.

Definition (Input automaton): An *input automaton* S is a tuple $(Q, \Sigma, \Delta, q_s, Q_e)$, where Q is a finite set of states, which we refer to as file states, $\Sigma = \mathcal{T} \cup \{eof\}$, where $\mathcal{T}$ is a set of record types, Δ is a set of transitions between the file states, with each transition labeled with an element of Σ, q_s is the designated *start* state of S, and Q_e is the (non-empty) set of designated *final* states of S. A transition is labeled with *eof* iff the transition is to a final state. There are no outgoing transitions from final states.

A file f consisting of a sequence of records R *adheres* to an input automaton S, or S *accepts* f, iff the types of the records in R, concatenated with *eof*, take S from its start state to some final state.

III. Our Program Specialization Approach

In this section we describe our approach for specialization a given program P using a given specialization automaton S as the constraint. The output from our approach is a set of nodes in the control-flow graph (CFG) of P that are unreachable during executions on inputs that satisfy the given constraint, and that can hence be removed from the program.

A. Overview Of Our Approach

Existing approaches for specialization or conditioned slicing of (non file-processing) programs use varying underlying techniques to perform this transformation; e.g., several approaches [6], [4], [20] use symbolic execution [27], others use partial evaluation [25], while at least one approach [2] uses constant propagation (CP) [33]. Symbolic execution and partial evaluation are powerful, precise approaches, that could potentially be adopted to the domain of file-processing programs. However, they are relatively expensive, and need to solve challenging sub-problems such as loop invariants generation or binding-time analysis to ensure termination. Constant propagation, on the other hand, is a simpler, fully automated technique, that identifies conditionals in the program that are guaranteed to always evaluate in a fixed direction under the given specialization constraint. However, it lacks sufficient precision in the context of analysis of file-processing programs. For instance, at the point after line 25 in our running example, CP would not be able to identify any field of the input record as having a constant value; this is because SHDr, Itm, and Trl records have different values in their `typ` field. Now, since nearly every other point in the program is reachable from this point, no part of the program will be found to be unreachable under the specialization constraint.

The approach we propose is based on CP, but identifies *multiple* CP facts per program point, one for each file state of the specialization automaton S. A CP fact at a program point p associated with a file state q basically describes the constant values of variables upon executions that reach p while being "in" file state q. Intuitively, an execution is "in" a file-state q at a program point if the types of the records read from the input file so far take the automaton S from its start state to the state q. The main intuition behind our proposal is that values of important file-format related flags in the program (e.g., the flags `same-flag` and `eof-flag` in the running example) correlate well during execution with the file-state that an execution "is in" at any point of time; hence, allowing a CP fact per file state enables precise value-analysis of these flags, which in turn enables precise identification of the portions of code that are unreachable under a specialization constraint.

We now present our approach in detail. It consists of three steps: 1) perform a dataflow analysis, 2) identify unreachable CFG nodes and edges, and 3) perform post-processing to identify other statements that are not relevant to the specialization constraint.

B. Step 1: Dataflow Analysis

We assume a given sound [9] "underlying" abstract analysis domain $U \equiv ((L, \sqsubseteq_L), F_L)$, where L is a *join* semi-lattice (of finite height), and F_L is the set of transfer functions of the statements in the language. Each transfer function has the signature $L \to L$. Intuitively, U can be any analysis domain that allows the detection of unreachable program points; e.g., it could be CP (which is what we use in our implementation), or the *interval* domain [9], or a more precise relational domain such as the *octagon* domain [32]. Intuitively, the more precise the underlying analysis is, the bigger the fraction of the genuinely unreachable nodes that will actually be identified as unreachable.

The abstract lattice that we actually use for our dataflow analysis is $D \equiv Q \to L$, where Q is the set of file states in the given specialization automaton S. The partial ordering for this lattice is a "point wise" ordering based on the underlying lattice L:

$$d_1 \sqsubseteq_D d_2 \ =_{def} \ \forall q \in Q.\, d_1(q) \sqsubseteq_L d_2(q)$$

The initial abstract value that we supply at the entry of the program is (q_s, i_L), where $i_L \in L$ is a user-given initial value, and q_s is the start state of S. For instance, if none of the variables have a known value at the entry of a program, then i_L could be the "top" element $\top$ of L.

We now present our transfer functions for the lattice D. We consider the following three categories of CFG nodes: Statements *other* than `READ` statements, conditionals, and `READ` statements. The key to understanding the transfer functions is that statements other than `READ` do not change the file state that an execution is in.

Let n be any node that is neither a `READ` statement nor a conditional. Let $f_L^n : L \to L \ \in \ F_L$ be the "underlying" transfer function for node n. Since n cannot change the file state, our transfer function for node n is simply:

$$f_D^n(d \in D) \ = \ \lambda q \in Q.\, f_L^n(d(q))$$

Let c be a conditional node, with a *true* successor and a *false* successor. Let $f_{t,L}^c \in F_L$ and $f_{f,L}^c \in F_L$ be the underlying *true*-branch and *false*-branch transfer functions of c. Since a conditional node cannot modify the file-state either, our transfer function for conditionals is:

$$f_{b,D}^c(d \in D) = \lambda q \in Q.\, f_{b,L}^c(d(q))$$

where 'b' stands for t or f.

Finally, consider a `READ` node r, which is the most interesting case. Since a `READ` statement obtains a record from the input file and places it in the input file buffer in the program, the dataflow fact returned by the underlying transfer function $f_L^r \in F_L$ ought to depend not only on the incoming dataflow fact, but also on the type of the record that was read. In other words, f_L^r ought to have the signature $(L \times \Sigma) \to L$, where Σ is the set of record types plus *eof*. Intuitively, this transfer function ought to (i) remove all value-constraints associated with the input file buffer from the incoming dataflow fact, and then, (ii) if the second argument is a record type $t \in \mathcal{T}$ (as opposed to *eof*), add new constraints for the input file buffer based on the given constraints associated with t's type definition.

We are now ready to present our transfer f_D^r for a `READ` node r:

$$f_D^r(d) \ = \ \lambda q_j \in Q.\ \bigsqcup_{(q_i \to q_j) \in \Delta} \{f_L^r(d(q_i), \mathit{label}(S, q_i, q_j))\}$$

where $\mathit{label}(S, q_i, q_j)$ returns the label (which is a type, or *eof*) of the transition $q_i \to q_j$ in S. The intuition behind this transfer function is as follows. For any file state q_j, after executing the `READ` the fact from the lattice L that is to be associated with q_j at the point after the `READ` statement can be obtained as follows: (1) For each file state q_i such that there is a transition $q_i \to q_j$ labeled t in the automaton, transmit the fact $f_L^r(l_1, t)$, where $l_1 \in L$ is the fact that q_i is mapped

q_{sh}	q_i	q_t	q_e
`in-rec.typ='HDR'` `in-rec.src='SAME'` `eof-flag=0`	`in-rec.typ='ITM'` `same-flag='S'` `eof-flag=0`	`in-rec.typ='TRL'` `same-flag='S'` `eof-flag=0`	`same-flag='S'` `eof-flag=1`

Fig. 2. Fix point solution for program in Figure 1(a), using automaton in Figure 1(d), at the entry to the main loop

to at the point before the `READ` statement. (2) Take a join of all these transmitted facts.

Our presentation above was limited to the intra-procedural setting. However, our analysis can be extended to the inter-procedural setting using standard techniques, some details of which we discuss in Section VI.

C. Steps 2 and 3: Removing Unreachable Nodes, and Post-processing

Step 2 is straightforward. The program points at which every file state is mapped to the "bottom" element $\perp$ of L are marked as unreachable. Then, nodes that immediately follow these program points are "projected out"[1] of the program.

The post-processing of Step 3 is an optimization, in which the following two are applied repeatedly as long as applicable: (a) Any conditional node such that all nodes that are control-dependent on it on one side (true/false) have already been projected out can itself be projected out. (b) Any assignment statement whose definition reaches uses all of whom have already been projected away can itself be projected away.

D. Illustration

For our running example, using the input automaton that was depicted in Figure 1(d), Figure 2 depicts the fix-point solution produced by Step 1 (dataflow analysis), at the program point just before the conditional of the main loop in line 3 of the program. The solution here is a *join* of the solution just after line 2 and the solution just after line 25. Due to space limitations we are not able to depict the full fix-point solution at all points in the program.

A notable feature of the fix-point solution that is obtained is that for all three lines of code that ought to have been found unreachable for the given specialization criterion, namely, lines 9, 17, and 23 (see the discussion in Section I-B), the fix-point solution at these lines indeed maps all the file states to $\perp$. To see why, let us consider, for instance, line 9. Of the four "columns" in the solution in Figure 2 (each column represents a file-state and its associated CP fact), only one of them, namely, q_i, reaches the conditional in line 6. The other columns end up with a '$\perp$' CP fact at line 6, because the CP facts of those columns contradict the conditional in line 4. Now, in the q_i column, `same-flag` has the constant value `'S'` (this happens, in turn, because line 17 is not reached when all header records are of type SHdr). Therefore, the fix-point solution at line 9 has $\perp$'s associated with all file states.

Therefore, Step 2 of the approach projects out lines 9, 17, and 23 from the program. Step 3 first projects out the conditionals in lines 6, 14, and 20 (projecting out line 20 amounts to converting the "`ELSE IF`" in that line into an "`ELSE`"). Then, since the only use of variable `same-flag` (in line 6) has been projected away, Step 3 is also able to project away all assignments to this variable, namely in lines /a/ and 15 (thus effectively eliminating this variable from the program).

E. Correctness Of Our Approach

Our correctness claim (which is not difficult to prove) basically is that our dataflow analysis will infer a $\perp$ fact for *all* file-states at a program point p only if the underlying analysis U will identify a $\perp$ fact for this program point, when considering only executions on input files that adhere to the given input automaton. The claim above, combined with an assumption that the underlying analysis U is sound (i.e., the outgoing dataflow fact from any underlying transfer function is $\perp$ only if the corresponding edge cannot be executed under the incoming dataflow fact), implies the result that Step 2 will only project out nodes that are unreachable under executions on input files that adhere to the given input automaton.

IV. File Format Conformance Checking

In this section we describe how our program specialization approach of Section III can be adapted to solve a different problem that we had mentioned in the introduction, namely file format conformance checking. As mentioned in the introduction, a verification question that would be useful from the developer perspective is whether a program can potentially silently "accept" an ill-formed input file and possibly write out a corrupted output file ("over acceptance"). Or, conversely, could the program "reject" a well-formed file via an abort or a warning message ("under acceptance")? Note that whether an input file is well-formed or ill-formed is essentially a part of the requirements specification of the program.

Different programs use different kinds of idioms to "reject" an input file; e.g., generating a warning message (and then continuing processing as usual), ignoring an erroneous part of the input file and processing the remaining records, and aborting the program via an exception. In order to target all these modes in a generic manner, our approach relies on the developer to identify file-format related *rejection points* in a program. These are the statements in a program where format violations are flagged, using warnings, aborts, etc.

A. Detecting Under-Acceptance

Under-acceptance warnings can be detected simply by specializing a program with a well-formed input automaton. If any rejection point remains that has not been projected out, it signifies a potential under-acceptance problem. This approach is sound, in that it will not miss any under-acceptance scenarios. This follows from our correctness claim in Section III-E.

[1]The details of this operation are omitted, since they are available in the literature, for instance, in [38].

In our example program there is only one rejection point, which is line 23. This point gets projected out using the well-formed automaton that was depicted in Figure 1(e). Therefore, there are no under-acceptance warnings for this program.

B. Detecting Over-Acceptance

Intuitively, a program has *over acceptance* errors with respect to a given well-formed automaton if the program can reach the end of its "main" procedure *without* going through any rejection point when run on an input file that *does not* conform to the well-formed automaton. We check this property as follows:

1) We first *extend* the given well-formed automaton S to a *full* automaton, which accepts *all* input files (including ill-formed ones), by adding extra states and transitions to it systematically. We describe this procedure formally in an associated technical report [31].
2) We modify the transfer functions of our dataflow analysis (of Section III-B) at rejection points such that their outgoing dataflow fact maps *all* file-states to $\perp$, irrespective of the incoming dataflow fact. Intuitively, the idea behind this is to "block" paths that go through rejection points.
3) We then apply the modified dataflow analysis mentioned above, using the full automaton. Once a fix-point solution is obtained, we flag an over-acceptance warning if the dataflow value associated with any file state that is not a final state in the original well-formed automaton is not $\perp$ at the final point of the "main" procedure.

Since our dataflow analysis (of Section III-B) identifies a fact at a program point as $\perp$ only if it definitely is $\perp$ (see the claim in Section III-E), it follows that the approach mentioned above will not miss any over-acceptance scenarios as long as the developer does not wrongly mark a non-rejection point as a rejection point.

In our running example, the full automaton obtained by extending the input automaton in Figure 1(e) will have a file-state, say q_{er}, to accept ill-formed files. In this automaton, there will be transitions from all other states other than q_e to q_{er}. In the fix-point solution, at line 28 the data flow value associated with q_{er} is non-$\perp$, flagging an over-acceptance error. This is because the full automaton accepts a file containing a header record followed by *eof*. The program execution with such an input file does not reach the sole rejection point at line 23.

V. Leveraging Input Automatons within Existing Static Analysis Approaches

In this section we will discuss an adaptation of our basic specialization approach of Section III-B to solve a more general problem, which is to leverage an input file format to improve the precision or usefulness of *arbitrary* static analysis approaches.

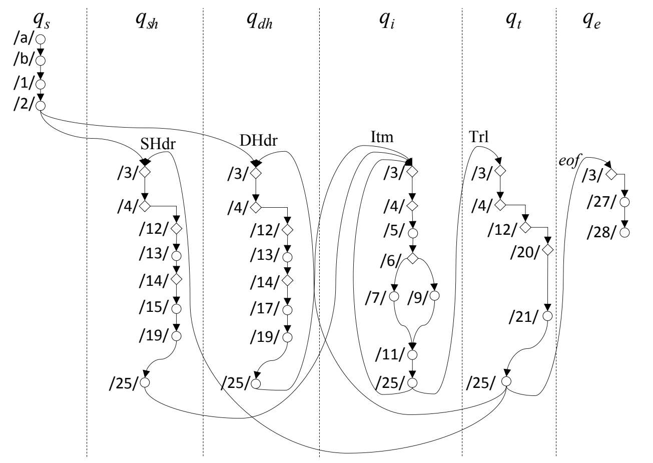


Fig. 3. Program File State Graph (PFSG) for the running example

A. Motivation

To motivate this adaptation, we first consider a couple of example static analyses that could benefit from input file formats.

1) Conditioned slicing: Recall that specialized slicing (see Section I-B) is performed by first specializing the program wrt a specialization automaton using our approach of Section III, and then slicing the specialized program. However, it is known in the literature [4] that such an approach is less precise than *conditioned slicing*. In our running example, if we use the well-formed automaton that was depicted in Figure 1(e), the "specialized" program would miss only line 23 compared to the full program. A slice of this program with the occurrence of `same-flag` in line 6 as the slicing criterion ends up including the definition of this variable in line /a/ into the slice, even though no execution on a well-formed input file flows directly from line /a/ to line 6 without going via lines 15 or 17 (which also assign to `same-flag`). Conditioned slicing would avoid this imprecision, but can only be performed if the format of well-formed files can be explicitly taken into account during slicing.

2) Uninitialized variables analysis: The field `out-rec.pyr` is definitely initialized at the `WRITE` statement in line 11 during executions on well-formed input files. However, a standard uninitialized variables analysis would not be able to determine this, unless information can somehow be given to the analysis that a header record will always be read and line 13 will always execute before line 11 is reached during any execution on a well-formed input file.

B. The Program File State Graph (PFSG)

In order to address issues such as the above, we propose a new program representation for file-processing programs called a Program File State Graph (PFSG). Figure 3 depicts a PFSG for our running example, wrt the well-formed input automaton in Figure 1(e). The PFSG is basically an "exploded" CFG (control-flow graph), with multiple copies of each CFG node, one per file state in a given input automaton S. That is, for each node n of the original CFG G and for each file state q in the input automaton, there is a node (n, q) in the PFSG, having the same internal content (i.e., statement or conditional as node n). In the PFSG in Figure 3, the nodes corresponding to different file states are depicted for visual clarity under different columns. For instance, the first column contains nodes (/a/,q_s), (/b/,q_s), (/1/,q_s), and (/2/,q_s); we omit the second components of these pairs from the figure because these are implied by the columns in which the nodes are present.

The PFSG is intuitively a bounded partial evaluation [25] of the original program wrt the given input automaton S. That is, a column corresponding to a file state q contains a specialized version of the program, containing only paths that an execution can take while remaining in file state q. Notice, in Figure 3, that that the q_{sh}, q_{dh}, q_i and q_t columns each contain a differently specialized version of the main loop, depending on the paths within the main loop that an execution can take while being in the corresponding file state. The transitions between the columns happen at line 25 (the `READ` statement), and happen as per the transitions in the input automaton S.

The PFSG as a program representation enjoys the following properties.

1. The PFSG is a control-flow graph itself. Therefore, *any* existing static analysis that is designed to run on a CFG, including, for example, forward or backward slicing, forward or backward dataflow analysis, model checking, and Hoare-logic based assertion checking, can be applied *without any modifications* on a PFSG.

2. Every path in a PFSG corresponds to some path in the CFG from which it is obtained. However, certain paths in the CFG that are infeasible under executions on input files that adhere to the given input automaton S could be elided from the PFSG. It is this feature that enables, for instance, the PFSG in Figure 3 be successfully applicable on all the analysis tasks that were mentioned above in Section V-A.

C. Producing a PFSG

Our basic analysis technique of Section III-B can be adapted to generate a PFSG from a CFG G and an input automaton S, as follows:

1) For each CFG edge $m \to n$ such that m is not a `READ` node, and for each file state q such that q is mapped to a non-$\perp$ fact in the fix-point solutions at m and at n, add a PFSG edge $(m, q) \to (n, q)$.
2) For each CFG edge $m \to n$ such that m is a `READ` node, and for each transition $q_i \to q_j$ in the input automaton S such that q_i is mapped to a non-$\perp$ fact in the fix-point solution at m and q_j is mapped to a non-$\perp$ fact in the fix-point solution at n, add a PFSG edge $(m, q_i) \to (n, q_j)$.

We provide more details about the PFSG and its properties in the associated technical report [31].

VI. Implementation and Evaluation

We have implemented our dataflow analysis of Section III-B. Our implementation is targeted at Cobol batch programs. These are very prevalent in large enterprises [7], and are based on a variety of standard as well as proprietary file formats. Another motivating factor for this choice is that one of the authors of this paper has extensive professional experience with developing and maintaining Cobol batch applications. We have implemented our analysis using a proprietary program analysis framework *Prism* [26]. Our implementation is in Java. We use the *call strings* approach [37] for precise context-sensitive inter-procedural analysis. In order to ensure scalability we have used an approximated (but conservative) variant of the call-strings approach [30]. We ran our tool on a laptop with an Intel i7 2.8 GHz CPU with 4 GB RAM.

S.No name	Prog.	No. of LoC	CFG Nodes
1	ACCTRAN	155	73
2	SEQ2000	219	115
3	DTAP	632	275
4	CLIEOPP	1421	900
5	PROG1	1177	762
6	PROG2	1052	724
7	PROG3	2780	1178
8	PROG4	49846	32258

Fig. 4. Benchmark program details

A. Benchmark Programs

We have used a set of eight programs as benchmarks for evaluation. Figure 4 lists key statistics about these programs. The programs ACCTRAN and SEQ2000 are example programs taken from a previous paper [38] and a textbook [34], respectively. The program DTAP was developed by the authors of this paper, while CLIEOPP was developed by a professional developer for training purposes; these programs are based on standard file formats [16] and [13], respectively. Programs PROG1-PROG4 are proprietary real-world programs. All of the programs, except SEQ2000, are from the banking domain. Program SEQ2000 is an inventory management program.

In the case of programs that accessed multiple input files, we chose one of the input files as the "primary" input file, and modeled `READ`s from the other files as always returning non-deterministic values.

We evaluate our specialization approach of Section III, as well as the file-conformance verification approach of Section IV.

B. Program Specialization and Slicing

The objective of this experiment was to evaluate the effectiveness of our program specialization approach in the context of *specialized slicing*; i.e., we first specialize a program wrt to a specialization constraint, and then slice the specialized program wrt a slicing criterion. We have currently not implemented the more precise *conditioned slicing* approach that was discussed in Section V-A. While Step 1 of our specialization approach (dataflow analysis) is carried out via our implementation, we have carried out Step 2 (projecting out irrelevant statements) manually as of now. Step 3 (optimizations) was not performed. We have performed slicing on the specialized

S. No (1)	Program name (2)	Num. lines in main loop (3)	Num. write stmts (4)	Num. lines in full slice (5)	Specialization constraint name (6)	Num. lines in specialized slice (7)
1	ACCTRAN	43	3	37	Deposit	13
					Withdraw	35
2	SEQ2000	66	1	61	Add	45
					Change	45
					Delete	38
3	DTAP	166	2	113	DDBank	85
					DDCust	85
					CTBank	85
					CTCust	85
4	CLIEOPP	482	6	189	Payments	146
					DirectDebit	169
5	PROG1	410	1	162	Edit	96
					Update	138
6	PROG2	236	4	112	Form	81
					Telex	80
					Modified	81
7	PROG3	454	20	407	TranCopy-1	73
					TranCopy-7	21
8	PROG4	4435	72	-	DAccts	-
					MAccts	-

Fig. 5. Results from specialized slicing

programs using the existing prototype slicing capability in the Prism framework.

Methodology of the experiment. As a first step, we identified one or more natural specialization constraints for each program, based on our understanding of the functionalities offered by the programs. We have given meaningful names to these constraints, as depicted in Column 6 of Figure 5. Due to space constraints, we are not able to describe these constraints in detail.

Regarding the slicing step, for comparative purposes, we performed slicing both on "full" (i.e., unspecialized) programs, as well as on the specialized versions of each program. In each case (i.e., whether it was a full program or a specialized program), we first identified the main processing loop in the program, and then selected the `WRITE` statements within this main loop that write to primary output files, and then used all of these `WRITE` statements together as the slicing criterion. (We ignored `WRITE` statements that write to error files, log files, etc.). We restricted the slice also to be within the main loop; this is because in batch programs, typically, the code within the main loop contains important business logic, while the code outside the main loop is primarily concerned about other aspects such as resource acquisition, recovery, and logging.

Results. Figure 5 also summarizes the results from this experiment. Column 3 indicates the number of lines of code in the main loop of the program, while Column 4 indicates the number of write statements in the slicing criterion. Both these columns pertain to the "full" (i.e., unspecialized) programs. Column 5 indicates the number of statements in the slice obtained on the unspecialized program (restricted to the main loop). Finally, Column 7 indicates the number of lines in the slices obtained on the specialized versions of the programs (again, restricted to the main loop).

The slicing infrastructure we used was not able to scale to the very large program, namely, PROG4. Therefore, we have omitted the slicing results for this program from Figure 5. Our specialization approach did scale to this program; it was able to identify 2022 lines of code in the main loop as unreachable under the DAccts criterion, and 1424 lines of code in the main loop as unreachable under the MAccts criterion.

The running time of our analysis (excluding the slicing step) was a few seconds or less on all programs except PROG4, and was about 1 hour on PROG4.

The two main takeaways from these results are:

- Slicing is useful in narrowing down the size of code to be inspected by a developer to identify key business logic. In most programs the number in Column 5 is much smaller than the number in Column 3 (up to 60% smaller in programs like CLIEOPP and PROG1).
- Slicing on specialized programs gives significant additional benefit in scenarios where developers would like to identify business logic pertaining to an input file constraint of interest. In most programs the individual numbers in Column 7 are significantly smaller than the numbers in Column 5 (e.g., 15% smaller and 40% smaller, respectively, for the two specialized variants of PROG1).

Precision of specialization. We also manually examined the specialized programs (before slicing) to evaluate the precision of our specialization approach. We did this for all programs except PROG4, which is very large. To our surprise, our approach was 100% precise on all programs, for all specialization constraints, except on the program SEQ2000. That is, it did not fail to mark as unreachable any CFG node that was actually unreachable (as per our human judgment) during executions on input files that conformed to the given specialization automaton. This is basically evidence that our CP-based approach in conjunction with specialization automatons is sufficiently precise to specialize file-processing programs.

C. File Format Conformance Checking

The objective of this experiment was to evaluate the usefulness of our file-conformance verification approach that was discussed in Section IV. The first step in this experiment was to manually create well-formed input automatons as well as "full" input automatons for each program. We did this using our understanding of the programs in the case of proprietary programs, and using the published standards where available. Note that even for proprietary programs organizations are typically conversant with the formats of the input files used by these programs. The biggest well-formed automaton was for CLIEOPP, which had 21 states and 48 transitions. The smallest well-formed automaton was for ACCTRAN, which had 4 states and 8 transitions.

The next step of the experiment was to manually identify *rejection points* in each program. This is a reasonably involved effort, because different programs use different idioms for dealing with file-format violations.

Figure 6 summarizes the results of this experiment. Column 2 in this figure indicates the number of under-acceptance

Prog. Name	File format conformance warnings	
	Under acceptance	Over acceptance
ACCTRAN	0	1
SEQ2000	3	1
DTAP	0	1
CLIEOPP	13	* 0
PROG1	5	9
PROG2	6	10
PROG3	0	1
PROG4	0	* 10

Fig. 6. Conformance checking results

warnings, which is the number of instances of a file state of the well-formed automaton being associated with a non-$\perp$ value at a rejection point. Column 3 indicates the number of over-acceptance warnings, which is the number of file states of the full automaton (excluding final states) that reach the final point of "main" with a non-$\perp$ value.

Precision of results. A noteworthy aspect of the under-acceptance results is that four of the eight programs, namely, ACCTRAN, DTAP, PROG3, and PROG4 have been *verified* as having no under-acceptance errors.

We manually inspected many of the under-acceptance warnings on the other programs. Many of the inspected warnings happened to be false positives. However, some of the warnings on CLIEOPP were genuine (i.e., true positives), and indicated programming errors that cause rejection of well-formed files.

Our implementation reports over-acceptance warnings on all the programs. (The numbers marked with a "*" are potentially lower than they should really be; this is because the full automatons had to be pruned in these cases, as otherwise they would have become very large.)

Our manual examination revealed that some of warnings reported for two of the programs – DTAP and PROG4 – turned out to be genuine. In the case of the program PROG4, the maintainers of this program were able to confirm this genuineness. They also added that at present there is another program that runs before PROG4 in their standard workflow that ensures that ill-formed files are not supplied to PROG4.

D. Discussion

In summary, our specialized slicing allows developers to narrow the amount of code to be inspected to identify business logic under specialized usage scenarios more than with traditional slicing. Furthermore, our specialization approach was 100% precise on six out of the eight programs.

On the novel problem of file-conformance verification, our implementation was able to verify four of the eight programs as not rejecting any well-formed files, and was able to find genuine under-acceptance as well as over-acceptance violations in several programs.

VII. Related Work

We discuss related work broadly in several categories.

Analysis of record- and file-processing programs. There exists a body of literature, of which the work of Godefroid et al. [21] and Saxena et al. [36] are representatives, on testing of programs whose inputs are described by grammars or regular expressions, via *concolic execution*. Their approaches are more suited for bug detection (with high precision), while our approach is aimed at conservative verification, as well as program understanding and transformation tasks.

Various approaches have been proposed in the literature to recover record types and file types from programs by program analysis [28], [3], [10], [15], [14]. These approaches complement ours, by being potentially able to infer input automatons from programs in situations where pre-specified file formats are not available.

A report by Auguston [1] shows the decidability of verifying certain kinds of assertions in file-processing programs.

Program specialization and conditioned slices. There is a significant body of work in using the technique of symbolic execution for program specialization or conditioned slicing, a sampling of which we cite [6], [4], [20]. Partial evaluation and amorphous slicing are related techniques, which are sophisticated forms of program specialization, involving loop unrolling to arbitrary depths, simplification of expressions, etc. [25], [29], [8], [22]. Hong et al. [24] propose an approach for conditioned slicing on an "exploded" CFG (similar to our PFSG), but where the nodes are duplicated based on a user-provided set of predicates (as opposed to file states). The primary novelty of our work in this line of research is the ability to specifically target file-processing programs, which are prevalent in many domains, using file-format specifications. Another difference is that we use a simpler (and potentially more practical) dataflow approach, based on a novel lattice whose elements map file-states to underlying facts. In our experiments we found that our approach is sufficiently precise on typical file-processing programs.

"Lifted" dataflow analyses. Our dataflow analysis uses a "lifted" lattice, namely, $Q \to U$, wherein a given underlying analysis domain U is "lifted" by the set of file states Q. Lifted analyses have been employed in the literature for different purposes. For instance, *type states* [39] have been used to lift the CP domain [12], while sets of predicates have been used to lift arbitrary underlying domains [18]. Also, the type-state approach mentioned above [12] was applied on file-processing programs, but to analyze the possible states that a file could be in at different points in the program (e.g., "open", "closed", "error"). To our knowledge ours is the first work to use file *content* automatons to lift an analysis.

VIII. Conclusion and Future Work

In this paper we presented a novel approach for specialization and verification of file-processing programs using file-format specifications.

A key item of future work is to allow richer (logical) constraints on input file contents. We would also like to explore in-depth the PFSG, and its usefulness in the context of various applications. Finally, we would like to investigate our techniques on domains other than batch programs, such as image-processing programs and XML-processing programs.

References

[1] M. Auguston. Decidability of program verification can be achieved by replacing the equality predicate by the "constructive" one. Technical Report NMSU-CSTR-9907, New Mexico State University, 1999.

[2] S. Blazy and P. Facon. SFAC, a tool for program comprehension by specialization. In *Proc. IEEE Workshop on Program Compr.*, pages 162–167, Nov 1994.

[3] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. ACM Conf. on Computer and Comm. Security (CCS)*, pages 317–329, 2007.

[4] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11):595 – 607, 1998.

[5] Apache Common Log Format. http://httpd.apache.org/docs/current/logs.html, 2013.

[6] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli. Software specialization via symbolic execution. *Software Engineering, IEEE Transactions on*, 17(9):884–899, 1991.

[7] Brain drain: Where Cobol systems go from here. *ComputerWorld*, May 21 2012. http://www.computerworld.com/s/article/9227263.

[8] C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(3):463–493, 1993.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[10] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proc. ACM Conf. on Comp. and Comm. Security (CCS)*, pages 391–402, 2008.

[11] S. Danicic, C. Fox, M. Harman, and R. Hierons. Consit: a conditioned program slicer. In *Proc. Int. Conf. on Software Maintenance (ICSM)*, pages 216–226, 2000.

[12] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. Conf. Prog. Langs. Design and Impl. (PLDI)*, pages 57–68, 2002.

[13] David Krop. CLIEOP Client Orders - File description. http://www.equens.com/Images/CLIEOP%20EN.pdf.

[14] P. Devaki and A. Kanade. Static analysis for checking data format compatibility of programs. In *Proc. Foundations of Softw. Tech. and Theor. Comput. Science (FSTTCS)*, pages 522–533, 2012.

[15] E. Driscoll, A. Burton, and T. Reps. Checking conformance of a producer and a consumer. In *Proc. Foundations of Softw. Engg. (FSE)*, pages 113–123, 2011.

[16] Retail Payment System(RPS),Deutsche Bundesbank. http://www.bundesbank.de/Redaktion/EN/Downloads/Core_business_areas/Payment_systems/procedural_rules_rps.pdf, 2013.

[17] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Proc. Sym. on Principles of Prog. Langs. (POPL)*, pages 379–392, 1995.

[18] J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *ACM SIGSOFT Int. Symp. on Foundations of Softw. Engg. (FSE)*, pages 227–236, 2005.

[19] K. Fisher and D. Walker. The PADS project: An overview. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 11–17, 2011.

[20] C. Fox, M. Harman, R. Hierons, and S. Danicic. Backward conditioning: a new program specialisation technique and its application to program comprehension. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 89 –97, 2001.

[21] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proc. ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 206–215, 2008.

[22] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.

[23] Introduction to HL7 Standards. http://www.hl7.org/implement/standards/index.cfm?ref=nav, 2013.

[24] H. Hong, I. Lee, and O. Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 25–34, 2005.

[25] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[26] S. Khare, S. Saraswat, and S. Kumar. Static program analysis of large embedded code base: an experience. In *Proc. India Software Engg. Conf. (ISEC)*, 2011.

[27] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[28] R. Komondoor and G. Ramalingam. Recovering data models via guarded dependences. In *Working Conf. on Reverse Engg. (WCRE)*, pages 110–119, 2007.

[29] J. Launchbury. *Project Factorisations in Partial Evaluation*, volume 1. Cambridge University Press, 1991.

[30] R. K. Medicherla and R. Komondoor. Precision vs. scalability: Context sensitive analysis with prefix approximation. In *Proc. IEEE Int. Conf. on Softw. Analysis, Evolution, and Reengineering (SANER)*, 2015.

[31] R. K. Medicherla, R. Komondoor, and S. Narendran. Static analysis of file-processing programs using file format specifications. *CoRR*, abs/1501.04730, 2015.

[32] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[33] S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[34] M. Murach, A. Prince, and R. Menendez. How to work with Sequential files. In *Murach's Structured COBOL*, chapter 13. Murach, 2000.

[35] S. Rugaber, K. Stirewalt, and L. Wills. Understanding interleaved code. *Automated Softw. Engg.*, 3(1-2):47–76, June 1996.

[36] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proc. Int. Symposium on Softw. Testing and Analysis (ISSTA)*, pages 225–236, 2009.

[37] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Application*. Prentice Hall Professional Technical Reference, 1981.

[38] S. Sinha, G. Ramalingam, and R. Komondoor. Parametric process model inference. In *Working Conf. on Reverse Engg. (WCRE)*, pages 21–30, 2007.

[39] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *Software Engineering, IEEE Transactions on*, SE 12(1):157–171, 1986.

[40] The United Nations rules for Electronic Data Interchange for Administration, Commerce and Transport. UN/EDIFACT DRAFT DIRECTORY. http://www.unece.org/trade/untdid/texts/d100_d.htm.

[41] H. Xi. *Dependent types in practical programming*. PhD thesis, Carnegie-Mellon University, 1998.