Parametric Process Model Inference

Saurabh Sinha
IBM India Research Lab
saurabhsinha@in.ibm.com

G. Ramalingam
Microsoft Research India*
grama@microsoft.com

Raghavan Komondoor
IBM India Research Lab
rkomondo@in.ibm.com

Abstract

Legacy applications can be difficult and time-consuming to understand and update due to the lack of modern abstraction mechanisms in legacy languages, as well as the gradual deterioration of code due to repeated maintenance activities. We present an approach for reverse engineering process model abstractions from legacy code. Such a process model can provide a quick initial understanding of an application, and can be a useful starting point for further program exploration. Our approach takes as input a user specification of interesting events, and creates a representation (i.e., a process model) that concisely depicts the occurrences of the events and the possible control-flow among them. The key features of our approach are the use of a logical data model of the program for specifying the events, and graph-projection techniques for creating the process model.

1 Introduction

Legacy applications constitute a large proportion of the application portfolio of many organizations. These applications can be difficult to maintain for several reasons: they are typically large and complex; their logical structure may have deteriorated due to evolution over decades; they are written in legacy languages that lack modern abstraction mechanisms; they suffer from code tangling (a single piece of code that implements several unrelated functions) and code scattering (code that performs one logical function may be scattered across multiple programs). Tools that reverse engineer higher level, logical, abstractions from legacy code can be of significant value in maintaining and transforming such legacy systems. In this paper, we address the problem of recovering *process model abstractions* from code to help developers in understanding the functionality implemented in legacy applications.

Fig. 1 is an example of an abstract process model recovered from the Cobol program shown in Fig. 2. It is evident

*Work done while the author was with IBM Research

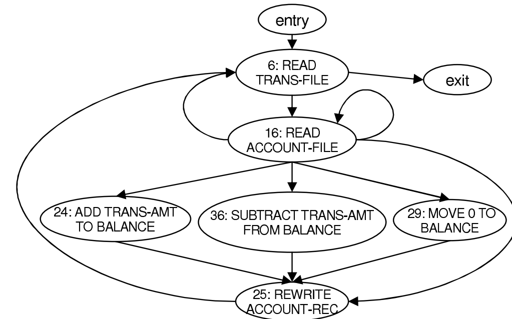

Figure 1. An example process model abstraction of a program.

that such a model can be of value in understanding an unfamiliar legacy application. In fact, we have learned from practitioners involved in maintaining Cobol-based legacy applications that an understanding of the occurrences of certain "events" in the application, and the order in which they occur, is often central to the program-maintenance tasks that they perform. Such events could be reads from (or writes to) persistent data stores, or could involve certain types of entities; *e.g.*, reading an Account object from persistent store, updating it, etc.

Although the value of process models is widely acknowledged, how to extract such models is not well understood, partly because the problem is not technically well defined. In this paper, we address this deficiency. Our hypothesis is that an application ought to have not a *unique* process model that distills its functionality, but that the user be able to specify the kinds of events they care about. Therefore, we present a two-step *parametric process model inference* approach that, in the first step, accepts a user specification of interesting events, and in the second step, produces (automatically) a process model abstraction that captures the order in which the interesting events occur in the application. The process model shown in Fig. 1 was generated by a specification that the events of interest were updates to "`Account Balance`".

A key novelty of our approach is to specify interesting events using a *linked logical data model* of the legacy

system. A logical data model for an application describes the logical structure of the data manipulated by the application at a high (abstract) level that is easy to understand, and can be expressed as an E-R model or a UML-style object-oriented data model. For example, the logical data model may contain an entity-type or class Account with an attribute Balance. Users can specify that the events of interest are updates to Account.Balance. This adds significant value to our approach because it simplifies the user's task of specifying events; the user need not know the actual implementation details to specify the events. *Links* connect the logical data model to the program and show how the program realizes the logical data model. Thus, links can be used to map the specified events to program elements.[1]

Once occurrences of interesting events in a program have been identified, it is possible to project the control-flow relations among the event occurrences. However, doing this in a straight-forward fashion can produce models that are less than ideal. A second key element of our approach is the use of techniques that have been used to create sparse evaluation representations (*e.g.*, [2, 5]) to produce the process model from the set of event occurrences.

The process model that is produced in this fashion can be used as a starting point for the user to understand the given system. Users can drill-down into the system as necessary to get a deeper understanding of the system. Specifically, the model can be extended interactively by users by adding other statements (*e.g.*, data dependence or control dependence predecessors of certain event occurrences) that they consider to be relevant and interesting.

In this paper, we illustrate our ideas using Cobol programs as examples, and assume that events map to statements in the program. However, the ideas presented in this paper can be applied to programs written in any language. Further, events can be mapped to computational steps at various levels of granularity: *e.g.*, at one extreme, events can be mapped to statements in a program, while at the other extreme, events can correspond to a JCL step in a JCL job. (JCL is a mainframe scripting language used widely to create batch jobs, where each job step corresponds to the invocation of a program.)

The contributions of the paper are as follows.

- We define process model extraction as consisting of an event-specification step and a model-inference step.
- We introduce the idea of specifying interesting events in a program using a logical data model of the program.
- We present an algorithm for projecting a program with respect to interesting events, to produce a process model that captures control flow among the interesting events and abstracts out uninteresting events.
- We illustrate, using an example program and a real Cobol application, how such process models can help practitioners with some recurring program understanding questions they face during maintenance.

[1] Section 2.3 discusses logical data models in greater detail.

The rest of the paper is organized as follows. In the next section, using an illustrative example, we discuss how process model abstractions can help in program understanding. In Section 3, we present our approach for inferring process models, and discuss a case study performed using a real Cobol application. Sections 4 and 5 discuss related work and directions for future work, respectively.

2 Process models for program understanding

In this section, we illustrate how developers can use our approach to recover process model abstractions to understand different functional aspects within an application. For this, we use the example Cobol program `ACCTRAN` shown in Fig. 2. This is an illustrative small program, potentially understandable as-is by a developer. However, depending on the functional aspect that needs to be understood, in a real application, the number of uninteresting events can vastly outnumber interesting events; the abstracted process model that is restricted to interesting events adds value.

Our approach (as detailed in Section 3) addresses interprocedural projections. However, to simplify the discussion in this section, we assume that all `PERFORM` statements (which essentially "call" paragraphs of code) in the example program have been inlined. We will use the term "projection" to refer to the process model produced by our approach.

2.1 Events related to I/O behavior

Consider a scenario where the user wants to understand the I/O behavior of a program; *e.g.*, the order in which files are read or written. Fig. 3 shows the projection of `ACCTRAN` with respect to read and write statements. (In all the process models we show in this paper, the numbers inside the nodes correspond to statement numbers in the underlying application program.)

The graph illustrates several aspects of file I/O in `ACCTRAN`. The transaction file is read in a loop (statement 6), and for each transaction, the account file in read (also in a loop) to locate the related account (statement 16). Following the read of the account file, the following events may occur: (1) the customer account may be read (statement 40) and updated (statement 39), or (2) the locked-account file may be updated (statement 47). Finally, before the next transaction is read, the account file is updated (statement 25). Thus, the projection not only provides a summary of how file I/O occurs in `ACCTRAN`, but also reveals information about the functionality of the program.

```
FD CUSTOMER-FILE.
01 CUSTOMER-REC.
   05 CUSTOMER-ID PIC 9(10).
   05 CUSTOMER-INFO.
      10 CUSTOMER-NAME PIC X(30).
      10 CUSTOMER-ADDRESS PIC X(100).
      10 CUSTOMER-TELEPHONE PIC 9(10).
   05 CUSTOMER-STATUS PIC XX.
      88 IS-PREFERRED-CUST    VALUE "PC".
      88 IS-NORMAL-CUST       VALUE "NC".
   05 NUM-OVERDRAFT-TRANS PIC 9(3).
FD TRANS-FILE.
01 TRANS-REC.
   05 TRANS-ID.
      10 TRANS-DATE PIC 9(8).
      10 TRANS-ACC-ID PIC 9(10).
   05 TRANS-CODE PIC X(2).
   05 CUST-ID PIC 9(10).
   05 TRANS-INFO.
      10 TRANS-TYPE PIC X.
         88 IS-WITHDRAW-TRANS    VALUE "W".
         88 IS-DEPOSIT-TRANS     VALUE "D".
      10 TRANS-AMOUNT PIC 9(10).
FD ACCOUNT-FILE.
01 ACCOUNT-REC.
   05 ACCOUNT-ID PIC 9(10).
   05 ACCOUNT-OWNER-ID PIC 9(10).
   05 ACCOUNT-STATUS PIC X.
      88 IS-CURRENT-ACC    VALUE "C".
      88 IS-LOCKED-ACC     VALUE "L".
   05 BALANCE PIC 9(10).
   05 OVERDRAFT-LIMIT PIC 9(5).
   05 NUM-CREDIT-OVERRUNS PIC 99.
FD LOCKED-ACC-FILE.
01 LOCKED-ACC-REC.
   05 LOCKED-ACC-ID.
      10 L-ACC-CUST-ID PIC 9(10).
      10 L-ACC-ID PIC 9(10).
   05 LOCKED-ACC-CUST-INFO PIC X(140).
WORKING-STORAGE SECTION.
01 ACC-ID PIC 9(10).
01 TRANS-AMT PIC 9(10).
01 MAX-CREDIT-OVERRUN-TRANS PIC 99 VALUE 11.
   PROCEDURE DIVISION.
 1.   OPEN INPUT TRANS-FILE.
 2.   OPEN I-O ACCOUNT-FILE.
 3.   OPEN I-O CUSTOMER-FILE.
 4.   OPEN OUTPUT LOCKED-ACC-FILE.
 5.   PERFORM PROCESS-TRANSACTIONS.
   PROCESS-TRANSACTIONS.
 6.   READ TRANS-FILE AT END
 7.      CLOSE TRANS-FILE
 8.      CLOSE ACCOUNT-FILE
 9.      CLOSE CUSTOMER-FILE
10.      CLOSE LOCKED-ACC-FILE
11.      STOP RUN.
12.   MOVE TRANS-ACC-ID TO ACC-ID.
13.   PERFORM GET-ACCOUNT-RECORD.
14.   PERFORM PROCESS-TRAN.
15.   GO TO PROCESS-TRANSACTIONS.
   GET-ACCOUNT-RECORD.
16.   READ ACCOUNT-FILE.
17.   IF ACCOUNT-ID NOT EQUAL ACC-ID THEN
18.      GO TO GET-ACCOUNT-RECORD END-IF.
19.   IF IS-LOCKED-ACC THEN
20.     GO TO PROCESS-TRANSACTIONS END-IF.
   PROCESS-TRAN.
21.   MOVE TRANS-AMOUNT TO TRANS-AMT.
22.   IF IS-WITHDRAW-TRANS THEN
23.      PERFORM PROCESS-WITHDRAW-TRAN
24.   ELSE ADD TRANS-AMT TO BALANCE END-IF.
25.   REWRITE ACCOUNT-REC.
   PROCESS-WITHDRAW-TRAN.
26.   IF TRANS-AMT IS GREATER THAN BALANCE THEN
27.      SUBTRACT BALANCE FROM TRANS-AMT
28.      IF TRANS-AMT IS LESS THAN OR EQUAL TO
         OVERDRAFT-LIMIT THEN
29.         MOVE 0000000000 TO BALANCE
30.         SUBTRACT TRANS-AMT FROM OVERDRAFT-LIMIT
31.         PERFORM UPDATE-CUST-FILE
         ELSE
32.         ADD 1 TO NUM-CREDIT-OVERRUNS
33.         IF NUM-CREDIT-OVERRUNS IS GREATER THAN
            MAX-CREDIT-OVERRUN-TRANS THEN
34.            MOVE 'L' TO ACCOUNT-STATUS
35.            PERFORM UPDATE-LOCKED-ACC-FILE END-IF END-IF
36.   ELSE SUBTRACT TRANS-AMT FROM BALANCE END-IF.
   UPDATE-CUST-FILE.
37.   PERFORM GET-CUSTOMER-RECORD.
38.   ADD 1 TO NUM-OVERDRAFT-TRANS
39.   REWRITE CUSTOMER-REC.
   GET-CUSTOMER-RECORD.
40.   READ CUSTOMER-FILE.
41.   IF CUSTOMER-ID NOT EQUAL ACCOUNT-OWNER-ID THEN
42.      GO TO GET-CUSTOMER-RECORD END-IF.
   UPDATE-LOCKED-ACC-FILE.
43.   MOVE CUSTOMER-ID TO L-ACC-CUST-ID.
44.   MOVE ACCOUNT-ID TO L-ACC-ID.
45.   MOVE TRANS-DATE TO L-TRANS-DATE.
46.   MOVE CUSTOMER-INFO TO LOCKED-ACC-CUST-INFO.
47.   WRITE LOCKED-ACC-REC.
```

Figure 2. Program `ACCTRAN`.

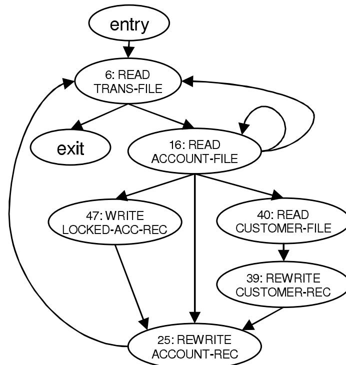

Figure 3. The projection of `ACCTRAN` with respect to read and write statements.

2.2 Events related to variables

Suppose that a developer wants to understand how the balance of an account is updated. The developer identifies field `BALANCE` of record `ACCOUNT-REC` as the relevant field, and indicates that statements that update `BALANCE` or perform an I/O operation with `ACCOUNT-REC` as the argument are the interesting events. Fig. 1 shows the projection of `ACCTRAN` (created by our approach) with respect to these events. The projection shows that updates to `BALANCE` in statements 24, 29, and 36, are followed by a write of `ACCOUNT-REC` to the account file in statement 25. Statement 6, which reads the transaction file, was not specified as interesting by the developer, but is nevertheless treated as an interesting event and included in the projection by the algorithm, to avoid creating a representation that could have a quadratic increase in the number of edges (as discussed in Section 3.2). Note that the 47-line program has been abstracted to a graph with only 8 nodes.

2.3 Events related to logical types

In many situations, events related to classes of variables may be of interest; *e.g.*, all variables that store account information or all variables that store customer IDs. However, because Cobol has no notion of user-defined types, information about such classes is not evident from examining the data declarations in the program. In previous work [7, 12], we have presented algorithms for inferring a logical data model from a Cobol application. A *logical data model* links program variables to classes (logical types), thus identifying variables of the same class; it also contains inheritance and containment relationships between the classes. Thus, our approach is to infer a logical data model of an application, and to let user specify interesting events in terms of all variables linked to a selected class in the data model.

For instance, say the user wanted to determine how `ACCTRAN` processes a transaction amount. The user may not know the program variable(s) that store transaction amounts, to specify in the query. By browsing through the variables, the user could select a variable, such as `TRANS-AMOUNT`, based on its name, and ask for a projection with respect to statements that refer to this variable. However, because `TRANS-AMOUNT` is assigned to `TRANS-AMT` (statement 21) and `TRANS-AMT` is used in all subsequent processing (`TRANS-AMOUNT` is not referenced again in the program), the projection with respect to `TRANS-AMOUNT` would contain only two nodes—corresponding to statements 6 and 21. Clearly, this provides an incomplete picture of how a transaction amount is processed.

To get a complete picture, the user should compute the projection with respect to the variables that are of the same type (i.e., are linked to the same class) as `TRANS-AMOUNT`.

Table 1. Some of the classes inferred for ACCTRAN in its logical data model, and the variables linked to these classes.

Class	Variables in class
Id	CUSTOMER-ID, ACCOUNT-OWNER-ID, L-ACC-CUST-ID
AccountRec	ACCOUNT-REC
AccId	ACCOUNT-ID, TRANS-ACC-ID, ACC-ID, L-ACC-ID
Credit	NUM-CREDIT-OVERRUNS, MAX-CREDIT-OVERRUN-TRANS
Trans	BALANCE, TRANS-AMOUNT, TRANS-AMT
TransDate	TRANS-DATE, L-TRANS-DATE
CustomerRec	CUSTOMER-REC
CustId	CUST-ID
CustomerStatus	CUSTOMER-STATUS
Info	CUSTOMER-INFO, LOCKED-ACC-CUST-INFO

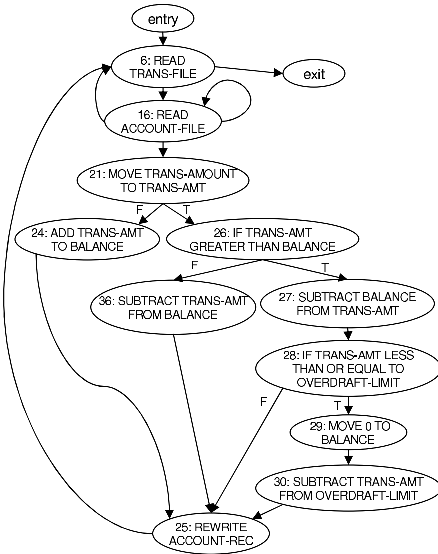

Figure 4. The projection of ACCTRAN with respect to class Trans.

Table 1 lists (a portion of) a logical data model for the program ACCTRAN. Note that variable TRANS-AMOUNT is linked to class Trans; variable TRANS-AMT is also linked to this class (because TRANS-AMOUNT is copied to TRANS-AMT), as is variable BALANCE (see statement 24). Fig. 4 presents the projection of ACCTRAN with respect to class Trans. This projection, by including statements that define or use TRANS-AMT and BALANCE, provides a better view of how ACCTRAN processes a transaction amount.

2.4 Extending a projection iteratively

A process model is a starting point for further program exploration, *e.g.*, for program browsing, or for applying other techniques such as program slicing and partial evaluation. After visualizing a projection, a developer may extend it by (a) expanding the set of interesting events, or (b) selecting one or more nodes in the current projection (interactively), and asking for control- and/or data-dependence predecessors of these nodes (either direct or transitive predecessors) to be added to the projection. In this way, the developer can start with a basic projection that provides very concise information, and then extend it iteratively, until they are satisfied with the level of detail. Such iterative extension would, in practice, be more useful than having to specify the events for, and having to view, the complete projection all upfront.

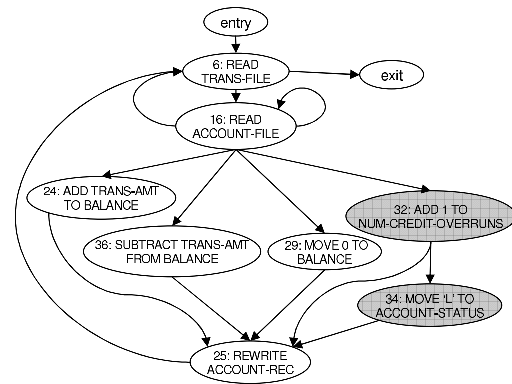

Figure 5. The projection for ACCTRAN with respect to updates to ACCOUNT-REC and I/O operations on ACCOUNT-FILE.

Consider the projection in Fig. 1; this projection illustrates that there are one or more paths, represented by edge (16, 25)[2] along which BALANCE is *not* updated, but the account file is written nonetheless. The developer would want to check why the balance is not updated—and whether some other field of ACCOUNT-REC is updated—along those paths. To do this, the developer can *extend* the set of specified events, by asking for the projection with respect to updates to all other fields of ACCOUNT-REC, in addition to BALANCE; this (now extended) projection is shown in Fig. 5. (The shaded nodes in Fig. 5 are additional nodes over the projection in Fig. 1.) This projection shows that along the two paths in which BALANCE is not updated, NUM-CREDIT-OVERRUNS is updated (statement 32); additionally, it shows that along one of these two paths, ACCOUNT-STATUS gets updated to 'locked' (statement 34). This helps the developer understand that the balance is not updated because a credit overrun occurs.

While here, the developer might also want to know the condition under which statement 34 (the update to ACCOUNT-STATUS) executes. To do this, the developer can interactively, using the approach, find the control-dependence parent of statement 34 (i.e., statement 33), and add it to the projection. For the sake of brevity, we omit the resulting projection.

[2] An edge in a projection can represent one or more paths in the program; edge (16, 25) in the example represents two paths to node 25: one with final edge (33, 25), and the other with final edge (47, 25).

2.5 Process models vs. program slices

Program slicing [16] is a related projection technique used for program understanding. A slice includes the transitive closure of data and control dependences, starting at a given program point and set of variables. The original definition of slicing [16] required a slice to be an executable program; thus, ordering among the statements in a slice can be computed by building the CFG of the sliced program.

However, to understand the ordering among an *arbitrary* set of statements (i.e., statements that match interesting events), slicing may not be the appropriate technique. To use slicing for computing such an ordering, the slice must include the transitive closure of control dependences of each statement. (Without this constraint, the CFG of the sliced program cannot be constructed.) This would cause unnecessary predicate nodes to appear in the projection; *e.g.*, if slicing were used to compute the ordering among the statements in ACCTRAN that update account balance (shown in Fig. 1), the projection would include the predicates in statements 17, 19, 22, 26, and 28.

Unlike a slice, a process model is intended to neither be semantically complete nor produce an executable program. The goal of creating a process model is to show the ordering among an arbitrary set of statements. Creating a process model—without the constraint of including all transitive control dependences, and that has no more edges than the CFG (as discussed in Section 3.2)—is a non-trivial problem, which has not been addressed in existing research.

3 Parametric process model inference

We propose process model extraction as a *two-step problem*: (1) specification of a set of interesting events, and (2) inference of the order in which the specified events can occur. The notion of an "event" is central to our approach. Informally, an *event* is any computational step or operation that occurs in an application. The first step requires interesting events to be specified, and matches of these events in the application code to be found; this is the topic of Section 3.1. The second step, which computes the potential control flow among these event matches, is the topic of Section 3.2.

3.1 Event specification and matching

An event specification is a set of events.

$$\mathcal{E} ::= \{e_1, e_2, \ldots, e_n\}, n \geq 1$$

An event e_i is a pair consisting of an entity and a set of operations on the entity.

$$e_i ::= \langle \mathit{Entity}, \{\mathit{Operations}\}\rangle$$

An entity can be any element of an application (*e.g.*, a variable), or an abstract element (*e.g.*, a class in the logical data model). Each such entity has different operations that can be performed on the entity; *e.g.*, for a variable or a class of variables, an operation can be a definition, a use, a read from a data store, or a write to a data store.

Events can be generalized to incorporate a wide variety of entities (and their related operations) in an application. We consider certain simple kinds of events, for which matches are individual statements and finding the matching statements is straightforward (extending the power of events-specification is left to future work). Examples of such events and their matching statements are:

- $\langle \texttt{BALANCE}, \{\mathit{def}, \mathit{use}, \mathit{read}, \mathit{write}\}\rangle$: each statement that defines, uses, reads from a file, or writes to a file, the variable BALANCE.
- $\langle \textsf{Trans}, \{\mathit{def}, \mathit{read}\}\rangle$: each statement that defines or reads from a file a variable that is linked to class Trans in the logical data model.
- $\langle \texttt{ACCOUNT-FILE}, \{\mathit{read}, \mathit{write}, \mathit{update}\}\rangle$: each statement that reads, writes, or updates a record in the ACCOUNT-FILE.

The main idea behind our two-step approach, of which the event-specification step is manual, is to let developers specify and understand the ordering among events they care about. Thus, complete automation of the event specification step is not a goal. Nonetheless, automatic recovery of abstractions, other than logical data models, can be used to make it easier to specify events.

Note that, formally, an event is a specification or a pattern for a statement, whereas event matches are the actual statements with respect to which the projection is done. (A statement can match more than one event.) However, where there's no confusion, we have used (and will continue to use) "event" to mean "event match." An event specification can also give a logical name to an event, which can be used in the generated process model to make it more readable.

3.2 Graph projection

In this section, we present a graph-projection algorithm that, given a program and a set of event matches, first constructs the control flow graph (CFG) for the program, and then creates a projection of the CFG (i.e., the process model abstraction) in which most non-matching nodes are omitted. We call a node that matches any event an *interesting* node. In Section 3.4, we describe a set of rules that can be used in a postpass to reduce the size of the projection, if desired.

Characterization of the projection produced

Given a CFG G and a set X of nodes in G, we define the projection of G with respect to X as follows.

Definition 1 *Let $G = (N, E)$ be a CFG and $X \subset N$ be a set of nodes. The* projection *of G with respect to X is a graph $G_X = (N', E')$: $N' = X$; E' contains an edge (u, v) if and only if there exists a path*

$(u, n_1, n_2, \ldots, n_k, v)$, $k \geq 0$, in G such that for $k \geq 1$ and $1 \leq i \leq k$, $n_i \notin X$.

The projection G_X can be constructed very simply by deleting (in any order) nodes not in X, and for each deleted node n, replacing each pair of edges (n_1, n) and (n, n_2) with a single edge (n_1, n_2). Therefore, a naive approach for creating a process model abstraction is to simply project the CFG with respect to the interesting nodes. However, a process model created in this manner may not be a desirable representation. Although the model would be minimal in terms of the number of nodes (i.e., it would contain no uninteresting node), it may contain a quadratic blowup in the number of edges. If an uninteresting node x has n interesting predecessors and m interesting successors, deletion of x would lead to the introduction of mn edges, one from each predecessor of x to each successor of x.

Therefore, our approach is to create a projection of the CFG with respect to the set of interesting nodes *plus* certain other (uninteresting) nodes, which we call *join nodes*. The join nodes, although uninteresting, are retained in the process model to avoid a quadratic increase in the number of edges. From an understanding perspective, the join nodes preserve some topology of the CFG with respect to branches and loops.

For example, consider the illustration in Fig. 6. The shaded nodes in the CFG (part (a) of the figure) are the interesting nodes. Fig. 6(d) shows the projection created by our algorithm, which includes join nodes 2 and 10, while Fig. 6(e) shows the projection with respect to just the interesting nodes. It is evident that the presence of several additional edges in part (e) makes the naive projection cluttered and difficult to understand. In fact, in this example, the projection has more edges than the original CFG; in the worst case, a projection with respect to an arbitrary set of nodes can have number of edges quadratic in the original number of edges. This blowup can cause the topology (i.e., the branching and looping structure) of the CFG to be completely lost. For instance, the outer loop in the CFG (Fig. 6(a)) is difficult to identify in the naive projection (Fig. 6(e)), whereas it is easily discernible in the projection created by our algorithm (Fig. 6(d)).

In general, the projection created by our algorithm will have no more edges than the original CFG.

Intraprocedural graph projection

To construct the projection, our approach uses elementary graph transformations that have traditionally been used for generating sparse evaluation representations to improve the efficiency of data-flow analysis [2, 5, 11].

A sparse evaluation representation can be generated by applying graph transformations that remove unnecessary nodes from a flowgraph [11]. Given a data-flow analysis problem, the transformations potentially remove nodes that preserve the solution (called *p-nodes*) or nodes at which the solution is not required (called *u-nodes*); the remaining nodes (called *m-nodes*), which may affect the solution, are retained in the final representation.

In our approach, a node is classified as interesting or uninteresting: an interesting node corresponds to an m-node; an uninteresting nodes corresponds to a p-node or a u-node. Thus, the transformations are directly applicable once nodes have been classified. To construct the projection, we use the following elementary graph transformations [11].

T2 Transformation. The T2 transformation, applicable to an uninteresting node that has only one predecessor, merges the node with its unique predecessor. Given an uninteresting node u and its unique predecessor v, the T2 transformation removes the edge (v, u) from the graph and replaces each edge (u, w) with an edge (v, w).

T4 Transformation. The T4 transformation, applicable to a strongly-connected set of uninteresting nodes, reduces such a set of nodes to a single uninteresting node. Given a set N of uninteresting nodes that is strongly connected, the T4 transformation (1) replaces the nodes in N with a single node w, (2) replaces each edge (u, v), where $u \notin N$ and $v \in N$, with an edge (u, w), and (3) replaces each edge (u, v), where $u \in N$ and $v \notin N$, with an edge (w, v), and (4) deletes edges (u, v), where $u \in N$ and $v \in N$.

T5 Transformation. The T5 transformation, applicable to an uninteresting node that has only one successor, merges the node with its unique successor. Given an uninteresting node u and its unique successor v, the T5 transformation removes the edge (u, v) from the graph and replaces each edge (w, u) with an edge (w, v).

The algorithm for computing the projection has the following steps.[3]

1. Given original CFG G, compute a subgraph G_u of G that contains only the uninteresting nodes. G_u is constructed by removing all interesting nodes (and their incident edges) from G. Identify the strongly-connected components in G_u. Let $N_1, N_2, \ldots, N_k$ denote the set of strongly-connected components in a topological sort order.
2. Apply the T4 transformation to each N_i in G, reducing each N_i to node n_i, resulting in CFG G_1. Mark n_i as "scc-node".
3. Visit nodes n_1 to n_k of G_1 in that order (a topological order). For each node n_i, apply the T2 transformation, if it is applicable. We denote the resulting CFG as G_2.
4. Let $n_1, n_2, \ldots, n_k$ be the set of uninteresting nodes in G_2 in a reverse topological sort order. Visit nodes n_1 to n_k in that order. For each node n_i, apply the T5 transformation, if it is applicable. Let the resulting graph be G_3.
5. For each scc-node n in G_3, add edge (n, n).

Fig. 6 illustrates the construction of the projection (inter-

[3]The algorithm is a simplified version of the algorithm for constructing a partially equivalent flowgraph presented in the approach of Ramalingam [11].

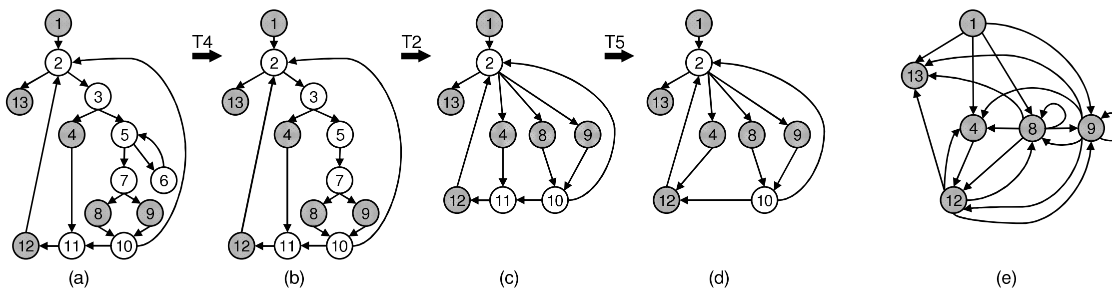

Figure 6. Construction of the projection. (a) Original CFG. (b)–(d) Graphs resulting from T4, T2, and T5 transformations. (e) Projection created by the naive approach.

esting nodes are shaded). Part (a) of the figure shows the original CFG; parts (b), (c), and (d) show the graphs that result after T4, T2, and T5 transformations, respectively. The T4 transformation replaces the strongly-connected component consisting of nodes 5 and 6 with a single node (node 5). Next, the T2 transformation merges each uninteresting node that has a single predecessor (i.e., nodes 3, 5, and 7) with its predecessor. Finally, the T5 transformation merges node 11 with its unique successor.

Interprocedural graph projection

In the interprocedural context, first call sites have to be marked as interesting or uninteresting. A *procedure* is interesting if it contains an interesting statement or a procedure called from it directly or transitively contains an interesting statement. A *call site* is interesting if it calls an interesting procedure. After call sites have been marked, the CFG of each procedure can be projected independently [11].

The algorithm for classifying procedures first marks each procedure that contains an interesting statement as interesting. Next, it traverses the call graph in a reverse topological order, and marks a procedure as interesting if that procedure calls an interesting procedure. After the procedures have been marked, the algorithm marks each call site based on whether it calls an interesting procedure.

Complexity analysis

Step 1 of the intraprocedural algorithm identifies strongly-connected components; the complexity of this step is $O(N + E)$. Steps 2, 3, and 5 require time $O(N)$. Step 4 processes CFG nodes in a reverse topological order, which requires $O(N + E)$. Therefore, the intraprocedural algorithm is linear in the number of edges in the CFG. The interprocedural algorithm incurs the costs of computing a topological order of the call graph (linear in the number of edges in the call graph), and processing each procedure.

3.3 Projection of a real application

We implemented a prototype tool that, given a Cobol application and a specification of events, creates the interprocedural projection of the application with respect to the events. The user can specify I/O events, events related to variables, or a set of event matches (i.e., statement numbers) as inputs to the tool.

Using the tool, we computed a projection for a real Cobol application (around 32000 lines) from the financial domain. The goal was to understand how the application updates the status of an Order; therefore, the projection was done with respect to statements that refer to variable `ORDER-STATUS`. Fig. 7(a) shows the interprocedural projection of one of the programs (2300 lines) in the application that processes order registrations.[4] The four shaded nodes represent assignments to `ORDER-STATUS`. (The projection includes entry/exit/call nodes and call/return edges to show control flow across paragraphs.) The projection shows that paragraph `P1` conditionally calls `P2`, in which `ORDER-STATUS` is first set to `'R'` ("received" status), in node 2115, and is then potentially updated to `'C'` ("accepted" status), in node 2129. `P2` calls `P3`, which can either set `ORDER-STATUS` to `'R'` (node 2223) or `'C'` (node 2232), or not update it at all.

The developer can interactively expand the initial projection to get a deeper understanding of the program. Because all the four shaded nodes update `ORDER-STATUS`, the question arises as to what conditions these nodes execute under. Therefore, the developer asks for all control-dependence ancestors of these nodes to be added to the projection, the result of which is shown in Fig. 7(c). This projection shows that fairly complex logic controls the updates to `ORDER-STATUS`. Under several conditions, `P2` is not called, and if `P2` is called, under further conditions, `ORDER-STATUS` is not updated. These conditions, which we have not shown for brevity, check, *e.g.*, whether a transaction is of a certain type or is rejected, whether a registration is premature, etc. After being initially set to `'R'` in `P2`, additional properties of an Order determine whether its status is further updated in either `P2` or `P3`. `P3` checks error conditions and time constraints before updating `ORDER-STATUS`.

In this way, by computing an initial projection that provides very concise information (12 nodes), and interactively expanding it to include conditionals, the developer can un-

[4] The node contents have been abbreviated, and variable names have been changed for confidentiality.

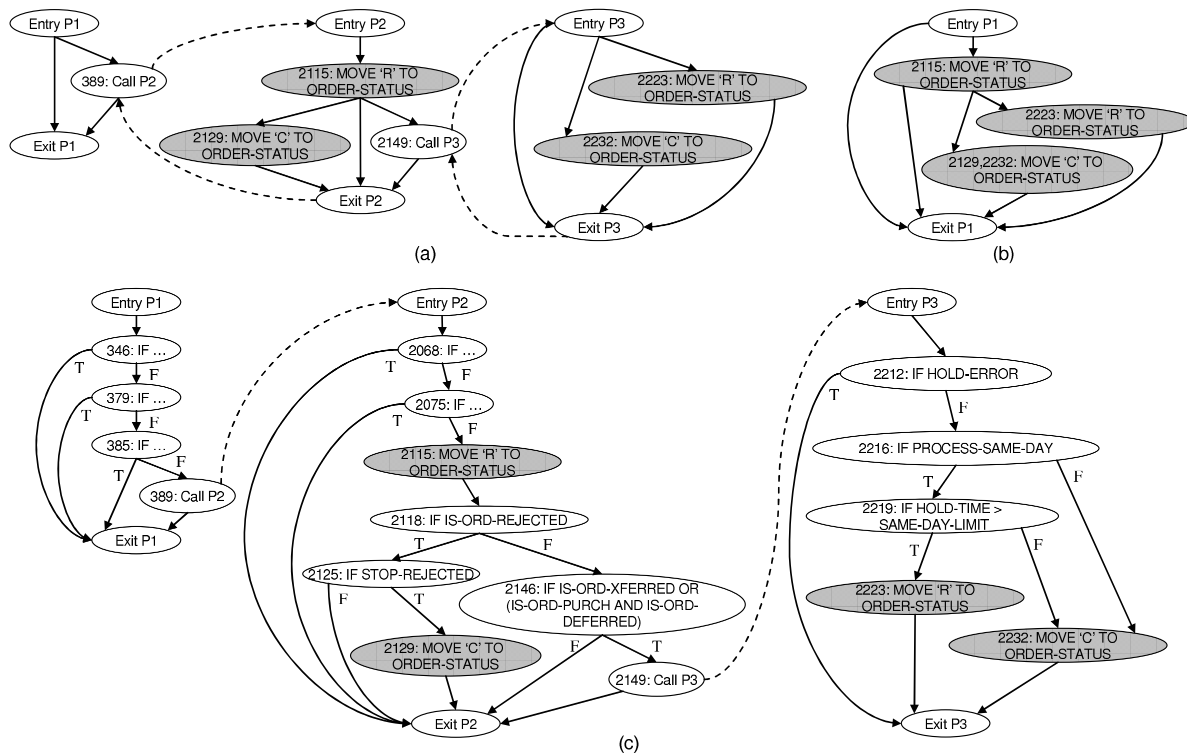

Figure 7. (a) The projection of a real Cobol program. (b) The projection after the application of postpass rules. (c) The projection including conditions that control the event matches.

Table 2. Data about the `ORDER-STATUS` **projection of the financial application.**

	Procs	Nodes	Total cyclomatic complexity	Num procs with cyclomatic complexity > 20
Original	728	28269	3720	42
Projection	73	365	170	1

derstand an important aspect of the processing done by a fairly large program that is also doing many other things not pertinent to the update of an order's status.

The application contains 84 statements that update `ORDER-STATUS`. There are so many updates because the application processes many types of orders, and performs multiple updates for each type of order. The interprocedural projection for the entire application contained 365 nodes, which included 250 entry/exit/call nodes and 31 join nodes. Although the projection is somewhat large, using the projection as a starting point instead of examining the entire application still simplifies the task of understanding this aspect of the application's functionality. Table 2 presents some quantitative data to illustrate the reduction in complexity.

3.4 Postpass to reduce projection size

The initial projection of the financial application is somewhat large because of the large number of event matches (84 updates to `ORDER-STATUS`) in the application, and the inclusion of entry/exit/call nodes in the projection. A postpass phase can be used to prune out some of the nodes, and make a projection more compact. First, some of the entry/exit/call nodes can be eliminated by inlining projections for called procedures at call sites. This should be done selectively, based on whether inlining would reduce the size of the projection. Second, some of the event matches—those that have either the same successors or the same predecessors—can be merged. We state these rules more formally as follows.

1. *Inline*. For each procedure P (in a reverse topological order of the call graph) with CFG G such that G has N nodes and there are C call sites that call P: if $C(N-2) < C+N$, inline G at each call site.
2. *Same-successors*. Given nodes e_1 and e_2 that match event set e and that have the same successors $s_1, s_2, \ldots, s_k$: (1) replace e_1 and e_2 with node e_{12}, (2) replace each edge (u, e_1) or (u, e_2) with edge (u, e_{12}), and (3) replace each pair of edges (e_1, s_i) and (e_2, s_i) with edge (e_{12}, s_i), $1 \leq i \leq k$.
3. *Same-predecessors*. Given nodes e_1 and e_2 that match event set e and that have the same predecessors $p_1, p_2, \ldots, p_k$: (1) replace e_1 and e_2 with node e_{12}, (2) replace each edge (e_1, u) or (e_2, u) with edge (e_{12}, u), and (3) replace each pair of edges (p_i, e_1) and (p_i, e_2) with edge (p_i, e_{12}), $1 \leq i \leq k$.

After applying the first rule, Rules 2 and 3 can be applied repeatedly, until neither of them is applicable. Using these rules, the size of the projection for `ORDER-STATUS` decreased from 365 to 173. The number of procedures in the projection decreased from 73 to 19. Rule 1 reduced the number of entry/exit/call nodes from 250 to 86; Rules 2 and 3 reduced the number of event matches from 84 to 56.[5] Fig. 7(b) shows the graph produced after application of the three rules to the projection shown in part (a) of the figure.

The graph that is produced after the postpass phase is no longer a projection of the CFG with respect to interesting nodes because the nodes in the graph do not have a one-to-one correspondence with nodes in the CFG—some nodes in the graph represent more than one CFG node. However, this is not a problem because the intention of a process model is not necessarily to show control flow among individual event matches (i.e., by distinguishing distinct matches of the same event), but to show the order among the events (i.e., treating any match of an event as the same). Because such a process model is not a projection of the CFG with respect to interesting nodes, we restate the property that it satisfies. In a postpass process model, a node n can represent a set of CFG nodes such that each node in the set matches the same event set e; in this case, we say that n matches e.

Definition 2 *Let $G = (N, E)$ be a graph and $\mathcal{E}$ be an event specification. Let $\mu : N \rightarrow \mathscr{P}(\mathcal{E})$ be an event function that maps node n to the event set to which n matches. If s is a sequence of event sets, let non-empty(s) be the subsequence of s obtained by omitting all empty sets from s. Let $\psi = (n_1, n_2, \ldots, n_k)$ be a valid interprocedural path in G.*[6] *We define the* event-label *for the path wrt event function μ, $\mathcal{L}_\mu(\psi)$, to be non-empty$(\mu(n_1)\mu(n_2)\ldots\mu(n_k))$.*

Theorem 1 *Let $G = (N, E)$ be a CFG, $\mathcal{E}$ be an event specification, and $G_p = (N_p, E_p)$ be the postpass process model created from G with respect to $\mathcal{E}$. Let $\mu : N \rightarrow \mathscr{P}(\mathcal{E})$ and $\mu_p : N_p \rightarrow \mathscr{P}(\mathcal{E})$ be event functions over the nodes in G and G_p, respectively. G_p contains a valid interprocedural path ψ_p with event-label $\mathcal{L}_{\mu_p}(\psi_p) = l$ if and only if there exists a valid interprocedural path ψ in G such that $\mathcal{L}_\mu(\psi) = l$.*

A simple algorithm for the postpass phase is to process procedures in a reverse topological algorithm. For each procedure, first apply Rule 1; then, iteratively examine pairs of nodes in the procedure's (possibly, inlined) projection, and determine whether either Rule 2 or Rule 3 can be used to merge the nodes. The algorithm can be expensive: $O(N^3 * A^2)$, where N is the number of nodes, and A is the maximum number of predecessors or successors that a node can have. In practice, the algorithm may be reasonable because it is applied to a *selectively inlined intraprocedural projection*, which typically should be small.

The postpass phase is an optional step in our approach, which can be skipped if the size of a projection is too large. Developing a more efficient algorithm for this step is left to future work.

4 Related work

Techniques such as program slicing [4, 16] and partial evaluation (or program specialization) [6] are well-known techniques for extracting a view or part of a program that can aid program understanding. The approach outlined in this paper does incorporate some elements of both slicing and partial evaluation, albeit indirectly. Specifically, our algorithm for logical data model inference incorporates a generalized form of data-dependence analysis that combines elements of slicing and partial evaluation. The results of this model inference are utilized in identifying events of interest.

Slices and process models provide different views of a program, and can be useful for program understanding in different ways. A slice is intended to be semantically complete, whereas a process model is intended to show only the ordering among an arbitrary set of statements. Understanding the order among a set of interesting events can be an end goal in itself, for which slicing is not an appropriate technique. Alternatively, a process model, because of its compactness, can be a useful starting point for program exploration. It can provide a quick initial understanding of a system; then, slicing can be used to drill down, and examine dependences selectively, as illustrated in Section 2.4.

Murphy et al. [10] exploit a human-specified mapping between source files and logical modules (each module may be linked to a set of source files). Given the mapping, they produce a *source model* that describes the calling relationships between the logical modules. Our use of a linked logical data model and the process model we produce are analogous to these concepts, but our use of *data* models and *event specifications* leads to models different from the kind of source models discussed in [10].

Several others (*e.g.*, [9]) have proposed languages for describing certain code patterns, and have described techniques to find matches (either statically or dynamically) to the specified patterns in code. These have typically been used to find bugs in program (*e.g.*, code fragments that violate certain design rules). Such techniques can be used to enrich our event specification language. The distinguishing aspects of our work, in relation to the above work, is that we exploit a logical data model for event specifications (which is particularly important for systems written in weakly typed languages such as Cobol), and we produce

[5] The postpass rules are currently not implemented in our prototype; we applied the rules manually to determine the reduction.

[6] A *valid* interprocedural path is one that respects calling contexts—i.e., it contains properly matched calls and returns.

a process model abstraction from the program rather than just identify event occurrences.

Discovery of API usage rules is another area that has attracted interest recently (e.g., [1, 17]). The goal here is to identify constraints (especially temporal constraints) on correctly using an API/library. Wagner and Dean [15] use static analysis to capture the order in which system calls may occur. Their goal is to infer a model that can be used at runtime to detect divergence from expected behavior.

Others (e.g., [3, 8]) have discussed, in various contexts, projecting the control flow of a program with respect to a subset of statements. However, they do not address the problem of quadratic increase in the number of edges in the projection. To extract output file formats, Lim et al. [8] construct a hierarchical finite state machine, which is a projection of the ICFG with respect to entry/exit/call nodes and output operations. In the context of intrusion detection systems, Giffin et al. [3] create a projection of a program with respect to system calls. They label each edge of a CFG with the name of a system call or with ϵ. Next, they use a T4-like transformation to reduce strongly-connected components of ϵ-edges. Following that, they use the naive approach to remove the remaining ϵ-edges.

Rountev et al. [13] discuss reverse engineering of UML sequence diagrams from source code. However, they focus more on the translation of source language constructs into UML constructs, and do not discuss producing abstractions of the program, as we do.

Another related area is reverse engineering of state diagrams. For example, the Shimba tool [14] uses dynamic analysis to collect event traces that show object interactions, and synthesizes state diagrams from the traces.

5 Future work

This paper attempts to characterize process model extraction as a technically well-defined problem, by decomposing it into two steps: definition of interesting events, and inference of the order in which the events can occur. Such a characterization is useful because it clearly identifies the two sub-problems that should be addressed in developing an approach for process model inference, and can, therefore, guide future research in this space in developing solutions that are designed to address the sub-problems.

Future research could extend our approach in several ways. The notion of events could be generalized to cover a wide variety of computational steps that may be of interest in the context of program understanding. This may require more sophisticated analysis to identify occurrences of events in programs. The projections could be presented at coarser levels of granularity, such as a program level. Future work could also define a formal language in which (generalized) events could be specified.

The following is an example of a scenario that could drive such generalization. Many entities have a lifecycle, which can be captured concisely using a notation such as UML state-transitions diagrams. Transitions between the different states of such an entity is an example of an interesting event. One could potentially extract the lifecycle of specified entities in the form of UML state diagrams.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Symp. on Principles of Prog. Lang.*, pages 4–16, Jan. 2002.

[2] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proc. of the 18th ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, pages 55–66, Jan. 1991.

[3] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proc. of the 11th USENIX Security Symp.*, pages 61–79, Aug. 2002.

[4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[5] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proc. of the ACM SIGPLAN 1994 Conf. on Prog. Lang. Design and Impl.*, pages 171–185, June 1994.

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

[7] R. Komondoor, G. Ramalingam, S. Chandra, and J. Field. Dependent types for program understanding. In *Proc. of the 11th International Conf. on Tools and Algorithms for the Construction and Analysis of Syst.*, pages 157–173, Apr. 2005.

[8] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *Proc. of the 13th Working Conf. on Reverse Eng.*, pages 167–178, Oct. 2006.

[9] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. of the 20th annual ACM SIGPLAN Conf. on Object Oriented Prog. Syst. Lang. and Applications*, pages 365–383, Oct. 2005.

[10] G. Murphy and D. Notkin. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. of the 3rd Symp. on the Foundations of Softw. Eng.*, Oct. 1995.

[11] G. Ramalingam. On sparse evaluation representations. *Theoretical Comput. Sci.*, 277(1–2):119–147, Apr. 2002.

[12] G. Ramalingam, R. Komondoor, J. Field, and S. Sinha. Semantics-based reverse engineering of object-oriented data models. In *Proc. of the 28th Intl. Conf. on Softw. Eng.*, pages 192–201, May 2006.

[13] A. Rountev, O. Volgin, and M. Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. In *ACM SIGPLAN-SIGSOFT Workshop on Prog. Analysis for Softw. Tools and Eng.*, pages 96–102, Sept. 2005.

[14] T. Systä. Understanding the behavior of Java programs. In *Proc. of the 7th Working Conf. on Reverse Eng.*, pages 214–223, Nov. 2000.

[15] D. Wagner and D. Dean. Intrusion detection vis static analysis. In *Proc. of the 2001 IEEE Symp. on Security and Privacy*, pages 156–169, May 2001.

[16] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.

[17] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. of the International Symp. on Softw. Testing and Analysis*, pages 218–228, July 2002.