Semantics-Based Reverse Engineering of Object-Oriented Data Models

G. Ramalingam
grama@us.ibm.com

Raghavan Komondoor
rkomondo@in.ibm.com

John Field
jfield@us.ibm.com

Saurabh Sinha
saurabhsinha@in.ibm.com

IBM Research

ABSTRACT

We present an algorithm for reverse engineering object-oriented (OO) data models from programs written in weakly-typed languages like Cobol. These models, similar to UML class diagrams, can facilitate a variety of program maintenance and migration activities. Our algorithm is based on a semantic analysis of the program's code, and we provide a bisimulation-based formalization of what it means for an OO data model to be correct for a program.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, maintenance, and enhancement—*restructuring, reverse engineering, and reengineering*; F.3.2 [**Logics and Meanings of programs**]: Semantics of programming languages—*program analysis*

General Terms: Algorithms, Languages

Keywords: type inference, program understanding

1. INTRODUCTION

Despite myriad advances in programming languages since business computing became widespread in the 1950s, legacy applications written in weakly-typed languages like Cobol still constitute the computing backbone of many businesses. Such applications are notoriously difficult and time-consuming to update in response to changing business requirements. This difficulty very often stems from the fact that the logical structure of these applications and the data they manipulate is not apparent from the program text. Two sources for this phenomenon are the lack of modern abstraction mechanisms in legacy languages, and the gradual deterioration of the structure of code and data due to repeated ad-hoc maintenance activities.

In this paper, we focus on the problem of recovering object-oriented data models from legacy programs, which can facilitate a variety of program maintenance activities by providing a better understanding of logical data relationships. Our recovered models, similar to UML class diagrams, incorporate classes, which contain data fields, and inheritance relationships between classes. A key aspect of our approach to constructing a data model is that it is based on an analysis of the *code that manipulates the data*, rather than an analysis of the *declaration of the variables* that store the data.

A second contribution of this paper is a (bisimulation-based) formal characterization of what it means for an object-oriented model to be a correct data model for a program. Our inference algorithm either produces correct models according to our characterization, or fails to produce any model (this happens in certain unusual situations, as described in Section 2).

We illustrate our techniques using Cobol, but we believe our approach is applicable to other weakly-typed languages (e.g., PL/I, 4GLs, and assembly languages) also.

1.1 A motivating example

Consider the example program in Figure 1. We will use this as a running example to illustrate the key deficiencies of Cobol[1] that hinder program understanding, as well as the working of our inference algorithm.

What do the declarations say? The initial part of the program contains variable declarations. Variables are prefixed by *level numbers*, e.g., `01` or `05`, which serve to indicate nesting, akin to record-field relationships, among variables. Thus, `account-rec` is a structured variable (record) consisting of "fields" `ar-acc-num`, `ar-user-name`, and `ar-data`. Other variables in the example prefixed by level `01` are similarly structured variables. Clauses of the form `PIC X(`n`)` declare the corresponding variable to be of size n, meaning that it stores byte sequences of length n. The `redefines` clause used in the declaration of variable `ir-acc-num` indicates that it is an overlay of variable `ir-user-name`, i.e. that the two variables occupy the same memory locations.

What does the program do? The executable statements follow the data declarations. The program first reads a transaction record into `input-record` (in statement /1/). Next, the

[1]We actually use a variant of Cobol that incorporates a few deviations from the standard syntax for the purpose of clarity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'06, May 20–28, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.

```
01 input-record.
  05 ir-trans-code pic x(1).
  05 ir-user-name pic x(8).
  05 ir-acc-num redefines ir-user-name.
  05 ir-data pic x(12).
01 account-rec.
  05 ar-acc-num pic x(8).
  05 ar-user-name pic x(8).
  05 ar-data pic x(5).
01 withdrawal-info.
  05 wi-amount pic x(6).
  05 wi-date pic x(6).
01 date-range.
  05 dr-from pic x(6).
  05 dr-to pic x(6).
01 log-record.
  05 lr-header pic x(9).
  05 lr-filler pic x(12).

/1/  READ input-record FROM transact-file.
/2/  READ account-rec FROM account-file
       WHERE ar-user-name = ir-user-name.
/3/  MOVE ar-acc-num TO ir-acc-num.
/4/  IF ir-trans-code = 'w' THEN
/5/    MOVE ir-data TO withdrawal-info
/6/    WRITE ir-acc-num, wi-amount, wi-date TO withdr-file
     ELSE
/7/    MOVE ir-data TO date-range
/8/    WRITE ir-acc-num, dr-from, dr-to TO inquiry-file
     ENDIF
/9/ MOVE input-record TO log-record.
/10/ WRITE lr-header TO log-file.
```

Figure 1: Running example

program uses the user name in `input-record.ir-user-name` to look up the corresponding account number (in statement /2/). (The `READ..WHERE` statement retrieves a record `account-rec` whose `ar-user-name` field equals `ir-user-name` from an indexed file.) Then, this account number is copied to `ir-acc-num` (we use field names without qualification when there is no ambiguity); note that `ir-acc-num` and `ir-user-name` are overlays, so the user name gets overwritten. Next the transaction record's `ir-trans-code` field is checked; depending on whether the code indicates a "withdraw" or an "inquiry" transaction, the transaction data in `ir-data` is copied to the appropriate top-level variable (`withdrawal-info` or `date-range`), and then appended to a file (`withdr-file` or `inquiry-file`) for further processing. Finally, in statements /9/ and /10/, the first two fields in the transaction record are extracted (by copying the record to the top-level variable `log-record`, then appended to `log-file`.

What's missing? An examination of the program logic reveals the following facts:

- the variable `ir-data` is not a *scalar* variable, but a *structured* variable.
- `ir-data` is in fact a *polymorphic* variable – it stores values of different types.
- `ir-data` stores values of the same type as `date-range` or values of the same type as `withdrawal-info`.
- `ir-user-name` and `ir-acc-num` constitute a *logically disjoint union* – i.e., they are not used to refer to the same data even though they occupy the same memory locations; we will later present an example with a contrasting use of redefined variables.
- variables `ir-acc-num` and `ar-acc-num` have the same type – i.e., they are used to store values from the same logical domain.
- variables `wi-amount` and `wi-date` do *not* have the same type.

However, there is *nothing* in the variable declarations (except the variable names themselves, which can be an unreliable source of information) to give the user any hint about these facts.

The key deficiency in Cobol that leads to these problems is that it has no type declaration mechanism. Naturally, there is no means to declare subtyping either.

We will now show that an object-oriented data model can be used to compactly convey all of the abovementioned facts, as well as other useful information. Furthermore, the inference algorithm we present in this paper can automatically create this model by analyzing the program's logic.

1.2 Linked object-oriented models

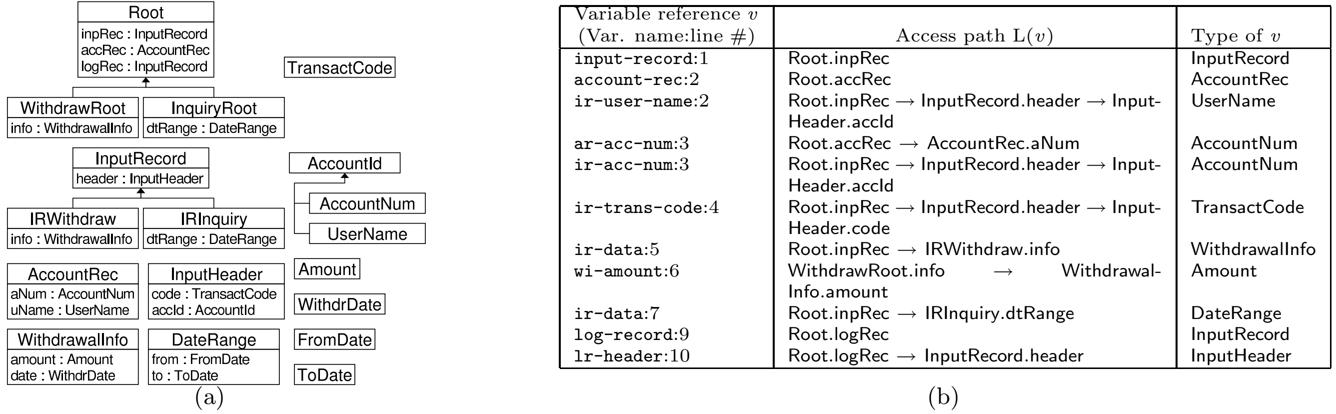
Figure 2 contains the output of our inference algorithm for the example in Figure 1. Figure 2(a) contains the object-oriented model (OOM), drawn as a UML class diagram. An OOM consists of a set of class definitions as usual: each class inherits from zero or more classes (its base classes), and has zero or more fields, while each field has a type which is a class. In Figure 2(a) each box is a class, with its name at the top, and list of fields below; inheritance relationships are shown as arrows from the subclass to the base class. Classes such as Amount, WithdrDate, which have no explicit fields, are called atomic classes; they represent scalar values, and actually have one "implicit" field of type *String* not shown here. Note that our inference algorithm does not automatically generate meaningful names for classes and fields (the names in Figure 2 were supplied manually for expository purposes); however, heuristics can be used to suggest names automatically based on the variable names in the program.

The object-oriented model is only one component of our inference algorithm's output. The second component is what we call a *link component*. The link component is intended to connect the declared variables in the program to elements of the inferred model, to illustrate, among other things, the type of a declared variable. However, in general, a variable may be used with different types in different parts of the program, and our inference algorithm is capable of capturing such information. So, the link component actually connects *variable occurrences* in the program with elements of the inferred model. The object-oriented model together with the links constitute a *linked object-oriented model* (LOOM).

We will now explain what information the link component captures for every variable occurrence. We first note that our use of the term "variable" is somewhat misleading. A symbol such as `ir-trans-code` in our running example plays a role somewhat different from conventional program variables. It identifies a *part of a structured datum.* One could say that it plays the role of a *field* (in a class/record definition) as well.

Hence, the links capture, in addition to the type of a variable occurrence, a *qualified access path* (which we define below) that identifies the part of a structured datum that the variable occurrence denotes.

We now formally define the link component. A qualified field name is an ordered pair (`C`,`f`), which we will also denote `C.f`, consisting of a class `C`, and a field `f` in class `C`. If no confusion is likely, we will omit the class name `C` when referring to a qualified field. A qualified access path *ap* is a sequence of one or more qualified field names



Variable reference v (Var. name:line #)	Access path L(v)	Type of v
input-record:1	Root.inpRec	InputRecord
account-rec:2	Root.accRec	AccountRec
ir-user-name:2	Root.inpRec → InputRecord.header → InputHeader.accId	UserName
ar-acc-num:3	Root.accRec → AccountRec.aNum	AccountNum
ir-acc-num:3	Root.inpRec → InputRecord.header → InputHeader.accId	AccountNum
ir-trans-code:4	Root.inpRec → InputRecord.header → InputHeader.code	TransactCode
ir-data:5	Root.inpRec → IRWithdraw.info	WithdrawalInfo
wi-amount:6	WithdrawRoot.info → WithdrawalInfo.amount	Amount
ir-data:7	Root.inpRec → IRInquiry.dtRange	DateRange
log-record:9	Root.logRec	InputRecord
lr-header:10	Root.logRec → InputRecord.header	InputHeader

(b)

Figure 2: (a) Object-oriented model, and (b) links from source code (not all links shown), produced by inference algorithm for the example in Figure 1

$(\mathtt{C_1.f_1}) \rightarrow (\mathtt{C_2.f_2}) \cdots \rightarrow (\mathtt{C_k.f_k})$ such that for each $1 \leq i < k$: $\mathtt{C_{i+1}}$ is equal to or is a derived class of the type of $\mathtt{C_i.f_i}$.

The links in a LOOM take the form of a function L that maps each variable occurrence `v` in the program to an ordered pair $((\mathtt{C_1.f_1}) \rightarrow (\mathtt{C_2.f_2}) \cdots \rightarrow (\mathtt{C_k.f_k}), \mathtt{C_{k+1}})$ consisting of a qualified access path and a type $\mathtt{C_{k+1}}$, where $\mathtt{C_{k+1}}$ is equal to or is a derived class of the type of $\mathtt{C_k.f_k}$. Such a link may be seen as making the following assertions about the program state when the statement containing the variable occurrence `v` executes, expressed, however, using the vocabulary of the object-oriented model:

- the program's full memory is of type $\mathtt{C_1}$; let us refer to this as object O_1
- for each $1 \leq i < k$ the value stored in the $\mathtt{C_i.f_i}$ field of object O_i (referred to as object O_{i+1}) is of type $\mathtt{C_{i+1}}$
- variable occurrence `v` refers to object O_{k+1}, which is of type $\mathtt{C_{k+1}}$

(Our formalization of the LOOM semantics, in Section 3, will clarify how to interpret the above assertions about the program's state expressed in terms of the model's vocabulary.)

Figure 2(b) illustrates the links inferred by our algorithm for the running example. Each row in the table contains a variable reference v (the left column), v's access path (the middle column), and the type of v (right column).

1.3 Usefulness of LOOMs

LOOMs recovered by our algorithm make explicit the data abstractions that programmers use implicitly when writing programs in weakly-typed languages. As a result LOOMs enhance program understanding, facilitate certain program transformations, and can serve as a basis for porting such programs into newer object-oriented languages that allow the abstractions to be made explicit. We now illustrate these advantages using our running example.

OOM by itself is valuable. We first note that just the object-oriented model (e.g., in Figure 2(a)) gives a valuable overall summary of the logical data domains manipulated by the program, and the relationships (nesting as well as inheritance) between them. This summary enhances program understanding.

Subtyping. The occurrence of `input-record` in statement /1/ has type InputRecord. InputRecord has two subtypes, IRWithdraw and IRInquiry. This means that values belonging to two logical domains – withdraw transactions and inquiry transactions – reside in `input-record` at statement /1/.

Base class factoring. Though the data stored in `input-record` belongs to one of two logical domains, some of this data is common to both logical domains. This common data has been lifted to the base class InputRecord (as field header), while the data that are unique to the two logical domains are modeled as fields of the corresponding derived classes.

Record structure of a declared scalar. `ir-data` is declared as if it were a scalar variable 12 bytes long. However, the LOOM shows that its type in statement /5/ is WithdrawalInfo, which is a class with fields. This means `ir-data` actually stores a value that is logically record-structured, in spite of the declaration to the contrary.

Impact analysis. Consider the following two toy programs that use the same variables `r` and `s`:

```
       Variables          Program 1      Program 2
01 r.          01 s.          READ r.        READ r.
  05 r1 pic x.   05 s1 pic x. MOVE r TO s.   MOVE r1 TO s1.
  05 r2 pic x.   05 s2 pic x. WRITE s1.      MOVE r2 TO s2.
                                             WRITE s1.
```

Observe that it is possible to reorder the fields of record `s` or add fields to `s` in program 2, without affecting the program's behavior. The same is, however, not true for program 1. This is clearly very useful information from a program maintenance perspective. The models we infer for these programs capture this information. For Program 1 our algorithm gives the same type (a class C) to all occurrences of `r` and `s`. On the other hand, for Program 2, our algorithm gives the occurrence of `r` a type C1 and the occurrence of `s` a different type C2; C1 and C2 both have two fields, and the corresponding fields in the two classes have the same type. The fact that `r` and `s` are given the same type in Program 1 means that they are tightly coupled with respect to their internal representations. Thus, the inferred model can assist in *impact analysis*: i.e., understanding the impact of a proposed change in the program.

Redefinitions. `ir-user-name` and `ir-acc-num` are overlays. They are disjointly used, in the sense that both variables are never used to access the same runtime value. The LOOM

makes this explicit by giving the occurrences of these two variables (in statements /2/ and /3/) different types (UserName and AccountNum, respectively). Had they been used non-disjointly (e.g., by writing a value into `ir-user-name` and then reading the *same* value via `ir-acc-num`) they would have been assigned the same type.

Improved Program Analysis Apart from its use for program understanding, a LOOM can also be used as the basis for more precise static program analysis. E.g., many analyses tend to lose precision in the presence of redefinitions since they do not distinguish between the different variables occupying the same memory location (for the sake of conservativeness). The LOOM can indicate when it is safe to treat such variables separately.

1.4 Correctness of LOOMs

How can we formalize the notion of a LOOM, which captures certain semantic aspects of a program, as being correct for that program? Consider the following example:

```
Variables                          Program 1   Program 2
01 r1 pic x(10).                   READ r1.    READ r1.
01 r2 redefines r1 pic x(10).      WRITE r1.   WRITE r2.
                                   READ r2.
                                   WRITE r2.
```

In the above example `r1` and `r2` occupy the same memory locations, due to the redefinition clause in the declaration of `r2`. Note that the redefinition is not essential to program 1: if we changed the declaration so that the two variables occupy disjoint memory locations, program 1's execution behavior will not be affected. In contrast, if we omit the redefinition, program 2's behavior will be affected. Specifically, the `WRITE` statement will now write out the initial value of `r2` as opposed to the value read in the first statement.

This idea serves as the basis for our approach to defining a notion of correctness of LOOMs. A LOOM for a program may be seen as describing an alternative way to represent data during the program's execution. Hence, a LOOM can be defined to be correct for a program if the program's "observed execution behavior" does not change if the alternative data representation determined by the LOOM is used during program execution.

The rest of the paper is structured as follows: We describe our algorithm in Section 2. Section 3 specifies the alternate execution semantics based on the LOOM, as well as the correctness characterization for LOOMs. Finally, we discuss related work in Section 4.

2. LOGICAL MODEL INFERENCE ALGORITHM

We present an outline of our algorithm and use our running example to informally illustrate the main aspects of our algorithm in Section 2.1. We then present a complete and formal description of the algorithm in Sections 2.2 through 2.4.

2.1 Overview and illustration of algorithm

Here is an outline of the steps in the algorithm. In Step 1 we compute, using a bidirectional dataflow analysis, a set of *cuts*; each cut identifies a certain range (interval) of memory locations at a certain program point that, at certain times during execution, stores values all of which must be represented by a single class in the data model. The class corresponding to any cut c will contain fields that in turn correspond to the smaller cuts nested immediately inside

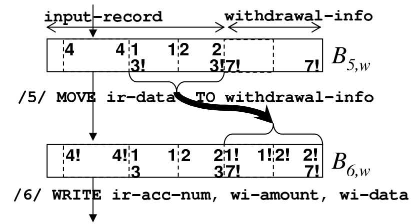


Figure 3: Illustration of cut inference. Cuts marked with "!" are seed cuts.

the cut c. Thus, the cuts are the bits and pieces from which we will construct an OOM. Next, in Step 2, we identify *field equivalence* and *class equivalence* relationships that must exist between the fields and between the classes, respectively, that we created in Step 1. Specifically, a field equivalence between a field f_1 of class C_1 and a field f_2 of a class C_2 indicates that the two fields must be lifted into a common base class of C_1 and C_2, while a class equivalence between two classes indicates that they must be unified in to a single class (thus, class equivalence is a full equivalence while field equivalence is a partial equivalence). In Step 3, we convert the set of classes and fields determined above (which we call "candidate" classes and fields) into a class hierarchy (the OOM) by factoring equivalent fields into appropriate base classes (after creating the necessary base classes). In Step 4 we create the links, by identifying for every variable occurrence its type as well as its access path in the OOM. Finally, in Step 5, we apply a set of (optional) rules to simplify the OOM.

In the rest of Section 2.1 we will use our running example to informally illustrate Steps 1 through 4 of the algorithm (Step 5, which is simple, is described in Section 2.4).

2.1.1 Step 1: Inferring cuts

Note that every variable corresponds to a *range* of memory locations: e.g., in our running example, `ir-user-name` corresponds to the range [2,9], while `input-record` itself corresponds to the range [1,21].

Introduction of graphical notation to illustrate cut inference. Consider Figure 3, which focuses on statements /5/ and /6/. The "boxes" labeled $B_{5,w}$ and $B_{6,w}$ are associated with the program points before statement /5/ and statement /6/, respectively. (A program point is the point between two statements.) Each box at a program point represents the contents of the *program's entire range of memory locations* at that point.

For purposes of illustration in the figure we show only a portion of the two boxes, the portion corresponding to top-level variables `input-record` and `withdrawal-info`. Cuts are shown in Figure 3 using pairs of dashed vertical lines, with both lines having the same label. We will soon show how these cuts are inferred. The thin arrow on the left, with the `MOVE` statement as its label, between the boxes is what we call a *transition* edge. It indicates that the program state represented by box $B_{5,w}$ transitions to a program state represented by the box $B_{6,w}$ by executing the `MOVE` statement. The bold arrow is a *value-flow* edge. Value-flow edges exist between boxes that precede and succeed a `MOVE` statement; a value-flow edge has a source interval in the preceding box (corresponding to the source variable of the `MOVE`), and a

target interval in the succeeding box (corresponding to the target variable of the `MOVE`).

First in cut inference – inferring "seed" cuts. For each variable *used* in a statement a seed cut is created for the range corresponding to this variable at the program point preceding the statement; for each variable *defined* in a statement, a seed cut is similarly created at the program points before and after the statement. This ensures, as we will see later, that the inferred model contains a field and class to which these variable occurrences can be linked. Consider statement /5/: We infer a cut corresponding to the range of `ir-data` at the program point before statement /5/ and a cut corresponding to the range of `withdrawal-info` at the program points before and after statement /5/. These are shown as cuts labeled 3! and 7! in Figure 3 (we use "!"s to denote seed cuts).

Second step in cut inference – propagating cuts. Next, we infer more cuts by "propagating" already inferred cuts. The propagation rules are based on the intuition that each cut represents an object at run time.

Some of these rules are based on *value flow*; the intuition here is that the "structure" of a structured value (i.e., the set of classes and fields used to model the value) is not changed by a statement that preserves this value, and we use cuts to represent the structure of values. Thus, in the example in Figure 3, cut 4 is propagated from $B_{6,w}$ to $B_{5,w}$ and cut 3 from $B_{5,w}$ to $B_{6,w}$, because the memory area containing these cuts is not written to by the `MOVE` statement that intervenes between these two program points; additionally, cuts 1 and 2 in $B_{6,w}$ are propagated (backward, via the value-flow edge) to box $B_{5,w}$, because they are inside the value that is copied by the `MOVE` statement.

Other propagation rules are based on the notion of *supercut flow*. If a statement assigns a new value to a range then any cut in the box that precedes (succeeds) this statement that completely contains the overwritten range is propagated to the box that succeeds (precedes) this statement. This rule can be understood by viewing the assignment statement as updating a (possibly transitive) *field* of the object represented by the containing "super cut", while preserving the object itself. In contrast, any cut that is completely contained within overwritten range before the statement will *not* be propagated. (As we will explain later, if a cut *partially overlaps* the overwritten range, our algorithm will halt.)

Besides boxes for program points, our approach also introduces a box for each data-source statement; these are statements that create new values in a program (e.g., `READ` statements, assignments of constants or arithmetic expressions to variables), as opposed to `MOVE` statements that simply copy existing values. The data-source box is an interval whose size is the same as that of the variable being defined, and has a value-flow edge flowing out of it to the interval corresponding to the variable being defined in the program-point box that follows the data-source statement. Cut propagation along these value-flow edges happens just as described above.

2.1.2 Value partitions and exploded CFGs

We have so far seen simple cuts: those that correspond to a range at a program point. In general, however, we will get an unsatisfactory model if we treat all data that resides in a certain range (of memory locations) at a certain program point uniformly (i.e., if we use a *single* class to describe all this data). Consider our running example. It follows from our description of the program's logic that the variables `input-record` and `ir-data` actually store different "types" of data for a "withdraw transaction" and an "inquiry transaction". A better model is obtained by creating separate classes to describe the data corresponding to these two cases.

We achieve this by generalizing the concept of a cut so that it can describe the data stored in a certain range of memory locations at a certain program point *under certain conditions*, as follows.

A *value partition* of a program is a mapping of each program point u and each data-source statement u to a finite set of predicates $\mu(u)$ (known as the value partition at u); for a program point its predicates refer to variables in the program, while for a data-source statement its predicates refer to the variable defined at that statement; further, for any program state that can arise at a program point u, $\mu(u)$ must contain at least one predicate that the program state satisfies; similarly, for any value generated by a data-source u, $\mu(u)$ must contain at least one predicate that the value satisfies. (Strictly speaking, we require the set of predicates in $\mu(u)$ to only *cover* the state spaces arising at u, not partition them; still, partitions would often make sense in practice, and hence we continue to call $\mu(u)$ a "value partition".)

Consider the running example in Figure 1. Here is a candidate value partition for this example, using the shorthand notation w for the predicate `ir-trans-code = 'w'`, and the notation i for the negation of this predicate:

$\{w, i\}$	for all program points that are after statement /1/ and outside the "if" statement, as well as for the data-source statement /1/ .
$\{true\}$	for the program point before statement /1/ as well as for the data-source statement /2/.
$\{w\}$	for the program points inside the "then" branch of the "if" statement.
$\{i\}$	for the program points inside the "else" branch.

At a high-level, our approach is to (1) compute a suitable value partition for the given program, (2) construct an *exploded* graph using the value partition (as described below), wherein each program point u and each data-source u is represented by several boxes, one for each predicate in $\mu(u)$, and (3) apply all five steps of the inference algorithm (as outlined in the beginning of Section 2.1) to this exploded graph. By having multiple boxes at a single program point or single data-source for inferring cuts pertaining to distinct logical domains, we produce better models. We will later describe how a suitable value partition can be computed for a program. The primary focus of this paper is, however, item (3) mentioned above.

Figure 4 shows the exploded graph for the running example derived from the value partition given above. Boxes B_w and B_i (at the top of the figure) are for the data-source statement /1/, and correspond to predicates w and i, respectively, while box B_a is for the data-source statement /2/. Each program-point box is labeled $B_{n,x}$, where n in the number of the statement that follows the program point to which the box pertains, and $x \in \{w, i, \text{true}\}$ is the predicate to which the box corresponds in the value partition at that program point. As in Figure 3, we only show certain interesting portions of the boxes, not the entire boxes.

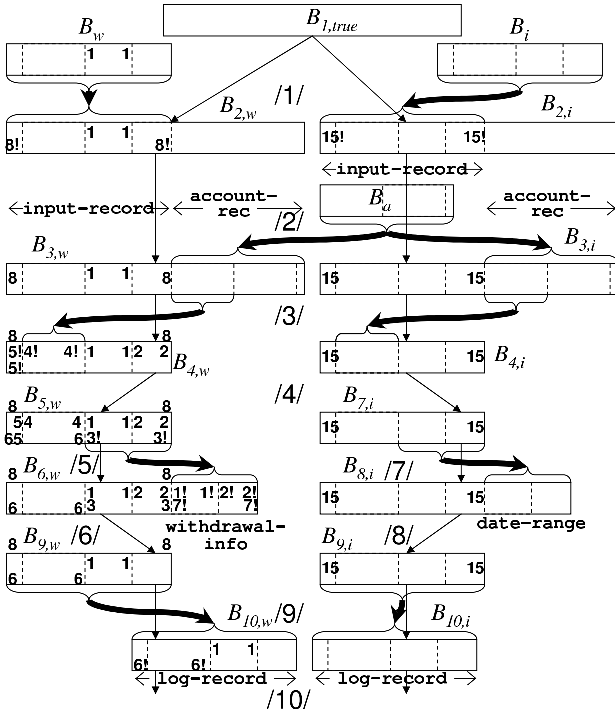


Figure 4: Exploded graph of example in Figure 1, with cuts produced by inference algorithm

The two types of edges in the exploded graph are added as explained in Section 2.1.1, but under additional constraints: (a) an edge (transition or value-flow) is created from a program-point box B_j to a program-point box B_k only if there exists a program state that satisfies B_j's predicate that is transformed by the statement intervening between these two boxes into a state that satisfies B_k's predicate, (b) a value-flow edge is created from a data-source box B_j to a program-point box B_k only if the conjunction of the predicates of the two boxes is not *false*.

As a consequence of rule (a) above, edges between program-point boxes in Figure 4 flow from "w" boxes to "w" boxes, and from "i" boxes to "i" boxes, but not across these categories. As a consequence of rule (b) the value-flow edge out of data-source box B_w goes to $B_{2,w}$, while the value-flow edge out of B_i goes to $B_{2,i}$.

The initial cut creation works on the exploded graph as described in Section 2.1.1, with the extension that a variable reference causes cuts to be created in *all* boxes in the program point preceding/succeeding (as appropriate) the statement that contains the reference. Cut propagation works as described in Section 2.1.1; in particular, we propagate cuts from one box to another only if there is an edge between them. This means, e.g., that no cuts are propagated from "w" boxes to "i" boxes, or vice versa, in the example in Figure 4, resulting in a better model.

Figure 4 contains all the cuts for the running example after cut propagation is over (some of those cuts have been labeled for illustrative purposes).

2.1.3 Step 2: inferring class and field equivalences

As mentioned earlier, each cut c in each box is a candidate class C for the OO model. Each smaller cut d nested immediately inside c (i.e., d is a "child" of c) corresponds to a field of class C, and the type of that field is the candidate class corresponding to d. However, we cannot simply create these classes and fields, e.g. for cuts in two different boxes, independently of each other. In this step we identify the constraints between these classes and fields that a correct model must satisfy.

Class equivalences. Here, we identify certain *corresponding* cuts in adjacent boxes connected by edges in the exploded graph, and add a class equivalence constraint between the corresponding cuts, which indicates that the corresponding cuts should be modeled by the same class in the model. Consider the two program points and boxes in Figure 3. Firstly, all cuts that are obtained by propagation from the same cut correspond. That is, the three cuts (in the two boxes) labeled 1 correspond, the two cuts labeled 3 correspond, etc. The intuition is the same as for the cut propagation. Similarly, a class equivalence constraint is added between entire boxes related by a transition edge (a program-point box can be thought of as an outermost-level cut), unless the intervening statement overwrites the entire memory. The reasoning for this is similar to that for the "supercut flow" rule explained in Section 2.1.1.

Field equivalences. Consider the reference to `ir-trans-code` in the conditional test labeled /4/ in our running example. The value partition at the program point before this predicate consists of two elements, as represented by the two boxes $B_{4,w}$ and $B_{4,i}$. We explained earlier that the basic idea is to create distinct classes, say C_1 and C_2, to describe the data represented by these boxes. Note, however, that the program makes a reference to `ir-trans-code`, after this program point, regardless of which box the program state corresponds to. We treat this as an indication that `ir-trans-code` is *common* to both classes C_1 and C_2 – i.e., that it really is part of a common base class of these two classes. We generate a *field equivalence* constraint between the cut corresponding to `ir-trans-code` in $B_{4,w}$ and $B_{4,i}$ to capture this requirement.

It turns out that class-equivalence between two candidate classes means the same as field-equivalences between all corresponding pairs of fields in the two classes; therefore, in our algorithm we employ only field equivalences.

2.1.4 Step 3: generating the object-oriented model

As mentioned in the beginning of Section 2.1, this step takes the candidate classes, the candidate fields, and the equivalence relation on the candidate fields; it unifies each equivalence class of candidate fields, and pulls up this unified field to an appropriate (perhaps newly created) base class of all the leaf classes from which the fields were pulled. We defer the details of how this is done (using *concept analysis*) to Section 2.4. For an illustration, consider the cuts labeled 3 in boxes $B_{5,w}$ and $B_{6,w}$. The candidate classes corresponding to each of these two cuts has two fields, corresponding to cuts 1 and 2. However, as explained in Section 2.1.3, the two "1" fields in these two classes are field equivalent, as are the two "2" fields. Therefore, both fields are pulled up to a base class, WithdrawalInfo (see Figure 2(a))), which means the two (leaf) candidate classes disappear entirely.

2.1.5 Step 4: Link generation

We discuss link generation in detail in Section 2.4, but

provide an example here. Consider the reference to variable `ir-data` in statement /5/ in Figure 1. Let *ap* be the qualified access path of this reference (we wish to generate *ap*). This reference corresponds to the cut labeled 3 in box $B_{5,w}$ in Figure 4. (If there had been multiple boxes at the point preceding statement /5/ we could have used any one of them, and due to the field-equivalence constraints generated in Step 2, we would have generated the same access path.) We now visit the cuts it is nested in, from outside to inside, and concatenate their representative fields to create *ap*: the outermost cut that contains cut 3 is numbered 8, and corresponds to a field (the first field) of the candidate class that corresponds to box $B_{5,w}$. The representative of this field in the model, namely Root.inpRec in Figure 2(a), becomes the first field in *ap*. Next, cut 3 corresponds to a field of the candidate class corresponding to cut 8; the representative of this field in the model is IRWithdraw.info. Therefore, *ap* = Root.inpRec → IRWithdraw.info (this is exactly what is shown in the row beginning with `ir-data`:5 in Figure 2(b).

2.2 Terminology and Notation

We introduce here the formal notation and terminology that we use in the rest of Section 2. We assume that the program is represented by a control-flow graph whose vertices denote *program points* and edges are labeled with statements. Any conditional test P is represented by a statement "`Assume` P" labeling the *true* branch and a statement "`Assume` $!P$" labeling the *false* branch. We will use the notation $u \xrightarrow{S}_c v$ to denote an edge from u to v labeled with the statement S. We address a subset of Cobol, which we call MiniCobol. MiniCobol incorporates the abovementioned `Assume` statement, `READ` statements, `MOVE` statements, and `WRITE` statements (as these statements suffice to illustrate all aspects of our algorithm). We use the term *variable occurrence* to denote an occurrence of a variable in the program.

We will refer to `READ` statements as well as assignment statements that assign a constant value to a variable as a *data-source* statement. We refer to any *use* of a variable in a statement other than a `MOVE` statement as a *data-sink*. Note that during program execution, values are generated by data-sources, and then copied around by `MOVE` statements, and eventually used at data-sinks. (Thus, our algorithm can be easily extended to handle Cobol's computational statements such as `COMPUTE X = Y+Z` by treating the `Y` and `Z` as data-sinks, and the statement itself as a data-source.)

Every variable in our language occupies a consecutive set of byte-sized memory locations $[i, j]$, which we refer to as the *range* corresponding to the variable. Given a statement S, let $\mathit{refs}(S)$ denote the set of ranges corresponding to variables referred to in that statement, and let $\mathit{defs}(S)$ denote the set of ranges corresponding to variables that are assigned a value in statement S. For MiniCobol, $\mathit{defs}(S)$ will contain at most one element. We also define $\mathit{lval}(S)$ to be unique element of $\mathit{defs}(S)$ if $\mathit{defs}(S)$ is non-empty, and the empty range ϕ otherwise. We say that $[i_1, j_1] \subset [i_2, j_2]$ if range $[i_1, j_1]$ is properly contained within range $[i_2, j_2]$: i.e., if $i_2 \leq i_1$ and $j_1 \leq j_2$ and $[i_1, j_1] \neq [i_2, j_2]$. Similarly, we use $r_1 \cap r_2 = \phi$ to indicate that the ranges r_1 and r_2 are disjoint.

The size $|\mathtt{R}|$ of a variable `R` is the sum of the size of all of its fields (excluding the fields that have `redefines` clauses). Let M denote the size of the total memory used by the given program, which is the sum of the sizes of its `01`-level variables. The range $[1, M]$ represents the total memory used by the program. Let μ denote a value partition for the program (see Section 2.1.2). We use the term *box* to denote an ordered pair (u, P) where u is a program point or data-source statement and $P \in \mu(u)$, and use the symbols B_i to refer to boxes. The length of a box $B = (u, P)$, denoted $|\mathrm{B}|$, is defined to be M if u is a program point and $|\mathtt{R}|$ if u is a data-source statement that assigns to variable `R`. We define $\mathit{range}[B]$ to be $[1, |\mathrm{B}|]$.

We define a relation $\xrightarrow{S}$, representing the transition edges between boxes described in Sections 2.1.1 and 2.1.2, as follows: let $B_1 = (u, P_1)$ and $B_2 = (v, P_2)$; we say $B_1 \xrightarrow{S} B_2$ iff $u \xrightarrow{S}_c v$ and there exists a program state satisfying P_1 that the execution of S transforms into a state satisfying P_2.

We will use the notation $\langle B, r \rangle$ to identify an ordered pair consisting of a box B and a range r. We define a relation $\Rightarrow$ on such pairs, a formal representation of the value-flow edges between boxes (see Sections 2.1.1 and 2.1.2), as follows. Let $B_1 = (x, P_1)$ and $B_2 = (v, P_2)$. We say $\langle B_1, r_1 \rangle \Rightarrow \langle B_2, r_2 \rangle$ iff: *either* x is a data-source statement, $u \xrightarrow{x}_c v$, $P_1 \wedge P_2 \neq$ false, $r_2 \in \mathit{defs}(x)$ and $r_1 = [1, |r_2|]$, *or* x is a program point, $x \xrightarrow{S} v$, S is a `MOVE` statement, $r_1 \in \mathit{refs}(S)$, and $r_2 \in \mathit{defs}(S)$.

2.3 Computing a Value Partition

The constant-valued function μ defined by $\mu(u) = \{\text{true}\}$ for all program points and data-sources u is a trivial value partition. This leads to an exploded graph with a single box to be used at all program points and data sources, which means the model will not use distinct classes (subtypes) to describe values corresponding to distinct logical domains.

The type inference algorithm described in [3] can be used to produce a better value partition as follows: This algorithm produces for every program-point u a set of union-free types $\Gamma(u)$ that describe the set of all program-states at that program-point. It also produces, for every data-source statement S, a set of union-free types $\Gamma(S)$ that describe the set of all values produced by that data-source statement. Every union-free type f has an associated predicate $\mathit{pred}(f)$. The function μ defined by $\mu(x) = \{\mathit{pred}(f) \mid f \in \Gamma(x)\}$ is a suitable value partition.

In the remaining part of this section we will assume that we are given the set of boxes, as well as the relations $\xrightarrow{S}$ and $\Rightarrow$ on the boxes. We note that our inference algorithm is correct as long as we use any conservative over-approximations of these relations.

2.4 The Model Inference Algorithm

This section contains a formal presentation of the model-inference algorithm, an overview of which was provided in Section 2.1.

Steps 1 & 2: Inferring Cuts and Equivalences. In this step we infer a set $\mathit{cuts}(B)$ of ranges, for every box B, as well as field-equivalences between cuts. For every r in $\mathit{cuts}(B)$, we define $\mathit{parent}\langle B, r \rangle$ to be the smallest range r' in $\mathit{cuts}(B) \cup \{\mathit{range}[B]\}$ such that $r' \supset r$. (For now, let us assume that the parent of a range r in $\mathit{cuts}(B)$ is well-defined. We will later discuss the case when the set $\{r' \in \mathit{cuts}(B) \cup \{\mathit{range}[B]\} \mid r' \supset r\}$ does not have a smallest range, which we expect to happen only rarely in practice.)

An inferred range r in $\mathit{cuts}(B)$ denotes several things. First, it identifies that the inferred model should include a class to represent $\langle B, r \rangle$, which we will denote by $\mathcal{C}\langle B, r \rangle$.

$$\frac{B_1 \xrightarrow{S} B_2, r \in \mathit{refs}(S)}{r \in \mathit{cuts}(B_1)} \quad \text{[REF]}$$

$$\frac{B_1 \xrightarrow{S} B_2, r \in \mathit{defs}(S)}{r \in \mathit{cuts}(B_1), r \in \mathit{cuts}(B_2), \; \langle B_1, r\rangle \sim_f \langle B_2, r\rangle} \quad \text{[DEF]}$$

$$\frac{B_1 \xrightarrow{S} B_2, r \in \mathit{cuts}(B_1), r \cap \mathit{lval}(S) = \phi}{r \in \mathit{cuts}(B_2), \langle B_1, r\rangle \approx \langle B_2, r\rangle} \quad \text{[VALUE-FLOW-1]}$$

$$\frac{B_1 \xrightarrow{S} B_2, r \in \mathit{cuts}(B_2), r \cap \mathit{lval}(S) = \phi}{r \in \mathit{cuts}(B_1), \langle B_1, r\rangle \approx \langle B_2, r\rangle} \quad \text{[VALUE-FLOW-2]}$$

$$\frac{\langle B_1, r_1\rangle \Rightarrow \langle B_2, r_2\rangle, r_3 \in \mathit{cuts}(B_1), \; r_3 \subset r_1, r_4 = r_3 + (r_2.\text{left} - r_1.\text{left})}{r_4 \in \mathit{cuts}(B_2), \langle B_1, r_3\rangle \approx \langle B_2, r_4\rangle} \quad \text{[VALUE-FLOW-3]}$$

$$\frac{\langle B_1, r_1\rangle \Rightarrow \langle B_2, r_2\rangle, r_4 \in \mathit{cuts}(B_2), \; r_4 \subset r_2, r_3 = r_4 + (r_1.\text{left} - r_2.\text{left})}{r_3 \in \mathit{cuts}(B_1)\langle B_1, r_3\rangle \approx \langle B_2, r_4\rangle} \quad \text{[VALUE-FLOW-4]}$$

$$\frac{\langle B_1, r_1\rangle \Rightarrow \langle B_2, r_2\rangle,}{\langle B_1, r_1\rangle \sim_i \langle B_2, r_2\rangle} \quad \text{[VALUE-FLOW-5]}$$

$$\frac{B_1 \xrightarrow{S} B_2, r \in \mathit{cuts}(B_1), r \supset \mathit{lval}(S)}{r \in \mathit{cuts}(B_2), \langle B_1, r\rangle \approx \langle B_2, r\rangle,} \quad \text{[SUPERCUT-FLOW-1]}$$

$$\frac{B_1 \xrightarrow{S} B_2, r \in \mathit{cuts}(B_2), r \supset \mathit{lval}(S)}{r \in \mathit{cuts}(B_1), \langle B_1, r\rangle \approx \langle B_2, r\rangle,} \quad \text{[SUPERCUT-FLOW-2]}$$

$$\frac{B_1 \xrightarrow{S} B_2, [1, M] \supset \mathit{lval}(S)}{\langle B_1, [1, M]\rangle \sim_i \langle B_2, [1, M]\rangle} \quad \text{[SUPERCUT-FLOW-3]}$$

$$\frac{B_1 \xrightarrow{S} B_2, B_3 \xrightarrow{S} B_4, r_2 \supseteq r_1, \; r_1 \in \mathit{refs}(S) \cup \mathit{defs}(S), r_2 \in \mathit{cuts}(B_1) \; B_1 = (u, P_1), B_3 = (u, P_2)}{r_2 \in \mathit{cuts}(B_3), \langle B_1, r_2\rangle \sim_f \langle B_3, r_2\rangle} \quad \text{[ACCESS-PATH]}$$

Figure 5: Inference rules for computing $\mathit{cuts}(B)$ as well as the field equivalence relations.

Second, it also has to be modeled as an *explicit* field of $\mathcal{C}\langle B, \mathit{parent}\langle B, r\rangle\rangle$, which we denote by $\mathcal{F}\langle B, r\rangle$. Finally, such a cut also identifies an *implicit* field of class $\mathcal{C}\langle B, r\rangle$, which we denote by $\mathcal{I}\langle B, r\rangle$, which represents the data in the range r not accounted for by the explicit fields of $\mathcal{C}\langle B, r\rangle$.

The inference rules in Fig. 5 show how we infer the cuts (ranges in $\mathit{cuts}(B)$), as well as two binary relations $\sim_f$ and $\sim_i$ on the cuts that represent *field equivalence*. The relation $\langle B_1, r_1\rangle \sim_f \langle B_2, r_2\rangle$ represents field equivalence between the fields $\mathcal{F}\langle B_1, r_1\rangle$ and $\mathcal{F}\langle B_2, r_2\rangle$, while the relation $\langle B_1, r_1\rangle \sim_i \langle B_2, r_2\rangle$ represents field equivalence between the fields $\mathcal{I}\langle B_1, r_1\rangle$ and $\mathcal{I}\langle B_2, r_2\rangle$. We use the shorthand notation $c_1 \approx c_2$ to indicate that $c_1 \sim_f c_2$ and $c_1 \sim_i c_2$.

At the end of this step, we check to see that for each box B, and each pair of cuts r_1 and r_2 in $\mathit{cuts}(B)$, r_1 and r_2 are either disjoint or one is contained completely within the other. If this condition does not hold, our model inference algorithm halts with failure. Informally, this failure situation indicates that the program contains references to two overlapping subranges of the same data; we expect this to happen rarely in practice, and note that modeling it would require a more complex and less intuitive version of a link that can associate a variable occurrence to a *sequence* of access paths (as opposed to a single access path).

Step 3: Generating the Class Hierarchy. For each box B and for each range $r \in \mathit{cuts}(B)$ the cut $\langle B, r\rangle$ defines a *candidate class*. We use CCS to denote the set of all candidate classes. The cuts also help define the set of fields $\mathit{CF}(\mathcal{C}\langle B, r\rangle)$ in candidate class $\mathcal{C}\langle B, r\rangle$, as shown below.

$$\begin{aligned}
\mathit{CCS} &= \{\mathcal{C}\langle B, r\rangle \mid B \text{ is a box}, r \in \mathit{cuts}(B) \vee \\
&\quad\; r = \mathit{range}[B]\} \\
\mathit{parent}\langle B, r\rangle &= \text{smallest range } r' \in (\mathit{cuts}(B) \cup \\
&\quad\; \{\mathit{range}[B]\}) \text{ such that } r' \supset r \\
\mathit{childcuts}(B, r) &= \{r_1 \in \mathit{cuts}(B, r) \mid r = \mathit{parent}\langle B, r_1\rangle\} \\
\mathit{CF}(\mathcal{C}\langle B, r\rangle) &= \{\mathcal{F}\langle B, r_1\rangle \mid r_1 \in \mathit{childcuts}(B, r)\} \cup \\
&\quad\; \{\mathcal{I}\langle B, r\rangle \mid (\exists r_1 \text{ s.t. } r_1 \neq \phi \wedge r_1 \subset r) \wedge \\
&\quad\quad (\forall r_2 \in \mathit{childcuts}(C, r) : r_2 \cap r_1 = \phi)\}
\end{aligned}$$

We utilize the inferred cut equivalence relations $\sim_f$ and $\sim_i$ to define an equivalence relation $\sim$ on fields: we say that $\mathcal{F}\langle B_1, r_1\rangle \sim \mathcal{F}\langle B_2, r_2\rangle$ if $\langle B_1, r_1\rangle \sim_f \langle B_2, r_2\rangle$, and we say that $\mathcal{I}\langle B_1, r_1\rangle \sim \mathcal{I}\langle B_2, r_2\rangle$ if $\langle B_1, r_1\rangle \sim_i \langle B_2, r_2\rangle$. As explained in Section 2.1.3, if we have two candidate classes C_1 and C_2, and fields $f_1 \in \mathit{CF}(C_1)$ and $f_1 \in \mathit{CF}(C_2)$, such that $f_1 \sim f_2$, then we need to create a common base class B for C_1 and C_2, and create a single field f in B that represents both f_1 and f_2.

We use *concept analysis* [9] to create a class hierarchy that respects the above field equivalences. Concept analysis is a general technique for hierarchically clustering entities that have shared features. The input to concept analysis is a triple (O, A, R), where O and A are finite sets of *objects* and *attributes*, respectively, and R is a binary relation between O and A. We say that object $o \in O$ *features* attribute $a \in A$ if $(o, a) \in R$. A triple (O, A, R) uniquely identifies a set of *concepts*, which can be automatically generated using concept analysis. A concept is a pair (X, Y) such that X is a set of objects (a subset of O), Y is a set of attributes, X is exactly the set of all objects that feature all attributes in Y, and Y is exactly the set of all attributes featured in all objects in X; X is called the *extent* of the concept and Y is called the *intent* of the concept. Concepts are partially ordered under an ordering $\leq_R$, defined as follows: $(X_0, Y_0) \leq_R (X_1, Y_1)$ iff $X_0 \subseteq X_1$. In fact, this partial order induces a complete lattice on the concepts, known as the *concept lattice*.

Before proceeding, we introduce some terminology. Let $\mathit{CFS} = \{f \mid f \in \mathit{CF}(C) \wedge C \in \mathit{CCS}\}$ be the set of all fields in all candidate classes. The equivalence relation $\sim$ on the candidate-class fields partitions CFS into a set of equivalence classes. For each equivalence class ec we define $\mathit{candTypesOf}(ec) = \{\mathcal{C}\langle B, r\rangle \mid \mathcal{F}\langle B, r\rangle \in ec\}$. We create input for concept analysis as follows: Each candidate class $C \in \mathit{CCS}$ becomes an *object* for the concept analysis. Each equivalence class ec of CFS defines two *attributes* $\mathit{repOf}(ec)$ and $\mathit{typeOf}(ec)$. Intuitively, $\mathit{repOf}(ec)$ is the field in the final OO model that represents all candidate-class fields in ec, and $\mathit{typeOf}(ec)$ is its type. Therefore, we define the candidate classes (i.e., concept-analysis objects) in the set $\{C \mid C \in \mathit{CCS} \wedge \exists f \in \mathit{CF}(C) \text{ s.t. } f \in ec\}$ as featuring $\mathit{repOf}(ec)$, and define the candidate classes in $\mathit{candTypesOf}(ec)$ as featuring $\mathit{typeOf}(ec)$. At this point we apply concept analysis. Each resulting concept con_1 becomes a class $\mathit{classOf}(\mathit{con}_1)$ in the model; for each concept con_2 such that $\mathit{con}_1 \leq_R \mathit{con}_2$ and there exists no concept con_3 satisfying $\mathit{con}_1 \leq_R \mathit{con}_3 \leq_R \mathit{con}_2$, $\mathit{classOf}(\mathit{con}_1)$ is made a direct subclass of $\mathit{classOf}(\mathit{con}_2)$. For each candidate class $C \in \mathit{CCS}$ its representative $[C]$ in the model is defined as $\mathit{classOf}(\mathit{con}_4)$, where con_4 is the concept whose *intent* is equal to $\{\mathit{repOf}(f) \mid f \in \mathit{CF}(C)\}$. For each equivalence class ec we place the field $f_{ec} = \mathit{repOf}(ec)$ in the class $\mathit{classOf}(\mathit{con}_5)$, where con_5 is the concept whose *extent* is the set $\{C \mid C \in \mathit{CCS} \wedge f \in \mathit{CF}(C) \wedge f \in ec\}$;

if $candTypesOf(ec)$ is non-empty then we set the type of f_{ec} to be the "lowest common" base class of the classes $\{[C] \mid C \in candTypesOf(ec)\}$, else we set its type to be a primitive string (f is an "implicit" field which is not directly referred to in the program). For all $f \in ec$ we let $[f]$ denote f_{ec}.

Step 4: Generating Links. As explained in Section 1.2, the link component of the LOOM consists of a map from variable occurrences in the program to qualified access-paths. Consider any variable occurrence v in S, and r be the range in memory corresponding to v. Let $B_1 \xrightarrow{S} B_2$ be some transition. If v is the target of a data-source or a `MOVE` then it is linked to the qualified access-path corresponding to $\mathcal{F}\langle B_1, r\rangle$, else it is linked to the qualified access-path corresponding to $\mathcal{F}\langle B_2, r\rangle$. Our class hierarchy construction guarantees that this access path is independent of transition $B_1 \xrightarrow{S} B_2$ that is chosen.

We now explain how to generate the access path of any field $\mathcal{F}\langle B, r\rangle$ that corresponds to a variable occurrence v. Clearly $r \in cuts(B)$. Let $r' = parent\langle B, r\rangle$. Let B be the class in the OOM that contains the field $[\mathcal{F}\langle B, r\rangle]$. We define the *qualified field* corresponding to $\mathcal{F}\langle B, r\rangle$ to be $B.[\mathcal{F}\langle B, r\rangle]$. The *qualified access-path* to $\mathcal{F}\langle B, r\rangle$ is obtained (recursively) as follows: if $parent\langle B, r\rangle = range[B]$, then the qualified access-path to $\mathcal{F}\langle B, r\rangle$ consists of just the qualified field corresponding to $\mathcal{F}\langle B, r\rangle$; otherwise, the qualified access-path is obtained by concatenating the access-path to $parent\langle B, r\rangle$ with the qualified field corresponding to $\mathcal{F}\langle B, r\rangle$.

Step 5: Model Simplification. Finally, we eliminate certain irrelevant parts of the model by applying the following three rules repeatedly until no changes occur: (a) Remove a field from the model if it does not occur in any access path (in the link component) and does not correspond to any interval in a box that contains *live* data (we omit the definition of liveness for conciseness) (b) Remove a class if it does not occur in any access path and has no derived classes (c) If a class C has no fields and one derived class D: eliminate C, replace all occurrences of C in fields and access paths with D, and make D a subclass of the base classes of C.

This simplification produces models that are more suitable for program understanding and maintenance tasks; it is optional, and is not required for correctness.

3. CORRECTNESS CHARACTERIZATION FOR LOOMS

MiniCobol is a weakly typed language that uses an untyped data representation. All runtime values (the values of variables as well as the value of the whole program state) are simply strings. An OOM defines a universe $\mathcal{O}$ of strongly typed values. As we show later, the link component of a LOOM can be used to execute MiniCobol programs using this universe of strongly typed values. This execution halts if the value that arises at any context (during execution) is not of the type expected in that context. Thus, a LOOM determines an *alternate semantics* for a given program.

We say that a LOOM is correct for a program if the program's execution, under the alternate semantics determined by the LOOM, is "equivalent" to the program's execution under the standard semantics. What does it mean for these two executions to be "equivalent"? First, the program execution must follow the same path through the program in both cases. Second, the value of each data-sink in the *corresponding* execution of a statement in both cases must be the same.

Given a program P and input I, let $trace(P, I)$ denote the sequence $(S_1, m_1) \cdots (S_k, m_k)$, where S_i denotes the i-th statement executed by P on input I, m_i denotes a map from the data-sinks in statement S_i to their values during the execution of S_i, and S_k is the last statement executed, all under the standard semantics. (Note that the "input" to a MiniCobol program is the contents of the set of files that are read by the program.)

Next, we present a similar definition for the alternate semantics by a LOOM. However, the claim we made above that a LOOM determines an alternate semantics is not completely accurate. A LOOM does not have all the information necessary for defining the alternate semantics. The missing piece is something we call a *serialization model*, and it tells us how to convert strings into typed values (at a data-source statement) and vice versa (at a data sink). We note that it is straightforward to extend our algorithm to generate a serialization model as well. Section 3.1 presents a formal definition of a serialization model (α, γ).

Given a LOOM L for P and a serialization model (α, γ), we define $trace_{L,(\alpha,\gamma)}(P, I)$ just as $trace(P, I)$ was defined, except using the alternate semantics determined by L and (α, γ).

DEFINITION 1. *A LOOM L is said to be correct for a program P if there exists a serialization model* (α, γ) *such that, for any input I,* $trace_{L,(\alpha,\gamma)}(P, I) = trace(P, I)$.

THEOREM 1. *For any program P, if our inference algorithm produces a LOOM L, then L is correct for P.*

PROOF. Omitted due to space constraints. □

3.1 Details of alternate execution semantics

An OOM identifies a universe of typed values as follows. Let *String* denote the set of all strings, which constitute the primitive values in the system. For a class `C`, let *fields*(`C`) denote the set of fields of class `C` (including its inherited fields). An *object* of type `C` is an ordered pair $(\mathtt{C}, m)$, where m is a map (function) from *fields*(`C`) to other objects of the appropriate type or strings (as per the type of the fields) or a special value *null*. Let $\mathcal{O}$ denote the set of all typed objects, including the special value *null*, and let $\mathcal{U}$ denote the set $\mathcal{O} \cup String$. An object $o \in \mathcal{O}$ is said to be an *instance* of class B iff $o = (\mathtt{C}, m)$ where `C` is a derived class of B.

Fig. 6 defines various auxiliary functions used to define the alternate semantics based on a LOOM. The program state in the alternate semantics is represented by a single object $\sigma \in \mathcal{O}$. An access path ap serves to identify a field of a subobject (of the program state object σ); the function $lookup_P(\sigma, ap)$ defined in Fig. 6 retrieves the value of this field. As observed earlier, an access path implicitly incorporates downcasts. Hence, the lookup may fail, and, in this case, the lookup function returns *null*. The function $update_P(\sigma, ap, v)$ updates the value of the field identified by ap with its new value v. Note that this is a functional update and returns an object σ' representing the updated state (object).

We now present the alternate semantics for MiniCobol statements. Consider a `MOVE` statement S of the form `MOVE X TO Y`. Let $\mathtt{X}_S$ denote the occurrence of `X` in statement

/* Field lookup function: $lookup_F : \mathcal{O} \times Field \rightarrow \mathcal{U}$ */
$lookup_F(null, \mathtt{f}) = null$
$lookup_F((\mathtt{D}, m), \mathtt{f}) =$ if $(f \in fields(\mathtt{D}))$ then $m(\mathtt{f})$ else $null$
/* Path lookup function: $lookup_P : \mathcal{O} \times AccessPath \rightarrow \mathcal{U}$ */
$lookup_P(o, \epsilon) = o$
$lookup_P(o, \mathtt{qf_1}...\mathtt{qf_k}) = lookup_F(lookup_P(o, \mathtt{qf_1}...\mathtt{qf_{k-1}}), \mathtt{qf_k})$
/* Var occur. lookup: $lookup_V : \mathcal{O} \times VarOccurrence \rightarrow \mathcal{U}$ */
$lookup_V(o, \mathtt{X}_u) =$ let $(\alpha, \mathtt{C}) = link(\mathtt{X}_u)$ in
let v $= lookup_P(o, \alpha)$ in
if (v instanceof $\mathtt{C}$) then v else $null$
/* Class extension function: $extend_C : \mathcal{O} \times Class \rightarrow \mathcal{O}$ */
$extend_C(null, \mathtt{C}) = (\mathtt{C}, \lambda f.null)$
$extend_C((\mathtt{B}, m), \mathtt{C}) =$ if ($\mathtt{B}$ derived class of $\mathtt{C}$) then $(\mathtt{B}, m)$
elsif ($\mathtt{C}$ derived class of $\mathtt{B}$) then
$(\mathtt{C}, \lambda f.$ if $f \in fields(\mathtt{B})$ then $m(f)$ else $null)$
else $null$
/* Path extension fn: $extend_P : \mathcal{O} \times AccessPath \rightarrow \mathcal{O}$ */
$extend_P(o, \epsilon) = o$
$extend_P(o, \mathtt{qf_1}...\mathtt{qf_k}) =$ let $(\mathtt{C_1}, \mathtt{f_1}) = \mathtt{qf_1}$ in
let $in_1 = lookup_F(o, \mathtt{qf_1})$ in
let $out_1 = extend_P(in_1, \mathtt{qf_2}...\mathtt{qf_k})$ in
let $(\mathtt{D}, m) = extend_C(o, \mathtt{C_1})$ in
$(\mathtt{D}, \lambda g.$ if $g == \mathtt{f_1}$ then out_1 else $m(g))$
/* Field update function: $update_F : \mathcal{O} \times Field \times \mathcal{U} \rightarrow \mathcal{O}$ */
$update_F(null, \mathtt{f}, v) = null$
$update_F((\mathtt{D}, m), \mathtt{f}, v) =$ if $(f \notin fields(\mathtt{D}))$ then $null$
else $(\mathtt{D}, \lambda g.$ if g==f then v else $m(\mathtt{f}))$
/* Path update function: $update_P : \mathcal{O} \times AccessPath \rightarrow \mathcal{U}$ */
$update_P(o, \epsilon, v) = v$
$update_P(o, \mathtt{qf_1}...\mathtt{qf_k}, v) =$ let $in_1 = lookup_F(o, \mathtt{qf_1})$ in
$update_F(o, \mathtt{qf_1}, update_P(in_1, \mathtt{qf_2}...\mathtt{qf_k}, v))$
/* Var occur. update: $update_V : \mathcal{O} \times VarOccurrence \times \mathcal{U} \rightarrow \mathcal{O}$ */
$update_V(o, \mathtt{X}_u, v) =$ let $(\alpha, \mathtt{C}) = link(\mathtt{X}_u)$ in
let $o' = extend_P(o, \alpha)$ in
if (v instanceof $\mathtt{C}$) then $update_P(o', \alpha, v)$ else $null$

Figure 6: Definition of semantic functions

S. Executing statement S in a state σ produces the state $update_V(\sigma, \mathtt{Y}_S, lookup_V(\sigma, \mathtt{X}_S))$. The execution of the program halts if any top-level call to lookup or update returns *null*. We now consider `READ` and `WRITE` statements. A `READ` statement reads a *string* from the input file, while a `WRITE` statement must write out a *string* to the output file, even in the alternate semantics. This motivates the following definition. Let $String_k$ denote the set of all strings of length k. For any data-source (data-sink) x, let $|x|$ denote the length of the variable defined (used) in x. A *serialization model* (α, γ) consists of a pair of functions: a deserialization function α that associates every data-source x with a function $\alpha(x) : String_{|x|} \rightarrow \mathcal{O}$ and a serialization function γ that maps every data-sink r with a function $\gamma(x) : \mathcal{O} \rightarrow String_{|r|}$.

Given a serialization model, in addition to the LOOM, it is straightforward to define the alternate semantics for `READ` and `WRITE` statements. The execution of a `READ X` statement S reads a string of the appropriate length from the input file, deserializes it into an object v, and then produces the state $update_V(\sigma, \mathtt{X}_S, v)$. The execution of a `WRITE X` statement S first retrieves the value $lookup_V(\sigma, \mathtt{X}_S)$, serializes it into a string, and writes it out.

4. RELATED WORK

While there has been previous work in recovering object-oriented models [1, 8] and other kinds of type abstractions [4, 2, 5, 3] from weakly-typed programs, these approaches, except that of Komondoor *et al* [3], are not *path sensitive*. That is, our approach distinguishes program states satisfying different predicates at the same program point, and uses this mechanism both for more accurate analysis (less pollution), and for inferring subtyping in a general manner (previous approaches for inferring OO models [1, 8] are flow- and path-insensitive, and infer subtyping either heuristically or in limited situations). Furthermore, while our approach infers the nesting structure of classes by analyzing the *actual usage* of variables in the code, previous approaches either do not infer nesting structure at all or use the existing *declarative* structure for doing so. A third contribution of this paper is a semantic characterization of correct LOOMs and an accompanying alternate execution semantics for LOOMs.

Our current work is a follow on to that of Komondoor *et al* [3], which infers path-sensitive types, but not an OOM. Our new contributions include the following. Our approach infers nesting, which is an integral feature of OO models, while their proposal is a "flat" type system with no nesting. Both approaches make distinctions based on value partitions (i.e., can infer multiple types at a single program point), but ours introduces *factoring* in the model by bringing in the notion of *equivalent* fields (fields that are referred to by a common variable occurrence), and by unifying and pulling up such fields to common base classes. Their approach uses a hard-coded value-partition computation for path sensitivity, while ours can use any value partition given as a parameter (including the one computed by their approach).

Our approach to data-model reverse engineering is similar in certain respects to previous work on algorithms for analyzing and specializing *existing* class hierarchies in programs in OO languages, most notably the work of Tip *et al* [7] and Snelting *et al* [6]. These approaches use type inference to analyze how the existing classes are actually used in the original program (like we do). Their approach is flow- and path-insensitive, but this still yields satisfactory results for them because they are not generating an OOM from scratch.

5. REFERENCES

[1] G. Canfora, A. Cimitile, and G. A. D. Lucca. Recovering a conceptual data model from cobol code. In *Proc. 8th Intl. Conf. on Softw. Engg. and Knowledge Engg. (SEKE '96)*, pages 277–284. Knowledge Systems Institute, 1996.

[2] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sorensen, and M. Tofte. Annodomini: from type theory to year 2000 conversion tool. In *Proc. Symp. on Principles of Prog. Langs.*, pages 1–14. ACM Press, 1999.

[3] R. Komondoor, G. Ramalingam, S. Chandra, and J. Field. Dependent types for program understanding. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 157–173, 2005.

[4] R. O'Callahan and D. Jackson. Lackwit: a program understanding tool based on type inference. In *Proc. 19th intl. conf. on Softw. Engg.*, pages 338–348. ACM Press, 1997.

[5] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proc. Symp. on Principles of Prog. Langs.*, pages 119–132, 1999.

[6] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Trans. Prog. Lang. Sys.*, 22(3):540–582, May 2000.

[7] F. Tip and P. F. Sweeney. Class hierarchy specialization. *Acta Inf.*, 36(12):927–982, 2000.

[8] A. van Deursen and L. Moonen. Understanding COBOL systems using inferred types. In *Proc. 7th Intl. Workshop on Program Comprehension*, pages 74–81, 1999.

[9] R. Wille. Restructuring lattice theory: an approach based on hierarchies of concept. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, Dordecht/Boston, 1982.