Recovering Data Models Via Guarded Dependences

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

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

Abstract

This paper presents an algorithm for reverse engineering semantically sound object-oriented data models from programs written in weakly-typed languages like Cobol. Our inference is based on a novel form of guarded transitive data dependence, and improves upon prior semantics-based model inference algorithms by producing simpler, easier to understand, models, and by inferring them more efficiently.

1 Introduction

Legacy applications written in weakly-typed languages like Cobol can be difficult and time-consuming to update in response to changing business requirements. Reasons for this difficulty include a lack of modern abstraction mechanisms in legacy languages and legacy data-stores (such as files), and the deterioration of the structure of code and data due to repeated ad-hoc maintenance activities. As a result, the logical structure of legacy applications and the data they manipulate is often not apparent from the program text.

In this paper, we focus on the problem of *recovering object-oriented logical data models* from weakly-typed programs via static analysis. These models provide a better understanding of the logical relationships among data entities in data-stores as well as in programs, and can facilitate a variety of program understanding and maintenance activities, such as migration of legacy data (e.g., stored in files) to relational databases, generation of wrappers around legacy files to give a modern (e.g., relational) interface to them [23], field expansion, migration of applications to modern object-oriented languages, identifying and extracting web services from monolithic applications (e.g., see [6]), etc. See [27, 4] for more about applications of such logical data models.

Our key contributions are (a) a novel form of guarded (i.e., conditional) data dependence, which provides information about conditional value flow in a program, including transitive value flow through one or more copy statements, (b) a polynomial-time path-sensitive dataflow analysis to compute the above data dependences, (c) an efficient algorithm to infer an object-oriented logical data model from a program using the above data dependences, and (d) an implementation of our algorithm and its preliminary evaluation. One key aspect of our inference algorithm is that it is semantics-based: the inferred model may be interpreted as certain assertions about the execution behavior of the program (particularly with respect to dataflow).

While we illustrate our ideas using Cobol, they are applicable to other weakly-typed languages (e.g., C, PL/I, and 4GLs) also. Several aspects of the problem we study show up in the context of analysis of binary code as well (see [1]).

1.1 A motivating example

We will use the example program shown in Fig. 1 to illustrate key deficiencies of Cobol[1] that hinder program understanding, as well as our inference algorithm. The program reads an input value into variable `CARD-TRANSACTION-REC` in statement /1/. The declaration of `CARD-TRANSACTION-REC` earlier indicates that it is a record with four fields. Each field is declared to be a string of certain length: e.g., the third field `CARD-INFO` is declared to be a string 23 bytes long. However, there is more structure to the data stored in this field, which is revealed by the program logic in statements /5/ through /7/.

If the first byte of `CARD-INFO`, referred to in statement /5/ using Cobol's subrange notation[2] as `CARD-INFO[1:1]`, contains a `'C'`, the rest of `CARD-INFO` contains credit card details, consisting of a credit card number and expiry date, which are extracted using the subrange notation in statement /6/. Otherwise, the rest of `CARD-INFO` contains debit card details, and the debit card number is extracted and written out in statement /7/.

Similarly, the data stored in `LOCATION-DETAILS` is also structured, as revealed by statements /2/-/4/ and /8/-/10/. (Here, auxiliary variables with named fields are used, in-

*Part of this work was done while the author was with IBM Research

[1] We use Cobol with slightly modified syntax for purposes of clarity.

[2] Though Cobol represents ranges as [*starting-offset,length*], we will use the notation [*starting-offset,ending-offset*] in this paper.

```
01. CARD-TRANSACTION-REC.
  05. LOCATION-TYPE PIC X.
  05. LOCATION-DETAILS PIC X(20).
  05. CARD-INFO PIC X(23).
  05. AMOUNT PIC X(4).
01. ATM-DETAILS.
  05. ATM-ID PIC X(5).
  05. ATM-ADDRESS X(12).
  05. ATM-OWNER-ID PIC X(3).
01. MERCHANT-DETAILS.
  05. MERCHANT-ID PIC X(8).
  05. MERCHANT-ADDRESS PIC X(12).

/1/ READ CARD-TRANSACTION-REC.
/2/ IF LOCATION-TYPE = 'M'
/3/  MOVE LOCATION-DETAILS TO MERCHANT-DETAILS
    ELSE
/4/  MOVE LOCATION-DETAILS TO ATM-DETAILS
    ENDIF
/5/ IF CARD-INFO[1:1] = 'C'
/6/   WRITE CARD-INFO[2:17], CARD-INFO[18:23],
        AMOUNT TO CREDIT-CARD-CHARGES-FILE
    ELSE
/7/   WRITE CARD-INFO[2:17], AMOUNT TO DEBIT-CARD-CHRGS-FILE
    ENDIF
/8/ IF LOCATION-TYPE = 'M'
/9/   WRITE MERCHANT-ID, AMOUNT TO MERCHANT-PAYMENTS-FILE
    ELSE
/10/  WRITE ATM-ID, ATM-OWNER-ID TO ATM-STATS-F.
    ENDIF
```

Figure 1. Running example

stead of subranges, to extract data.) Thus, CARD-INFO and LOCATION-DETAILS are both *polymorphic*: their types can be one several subtypes, each with its own structure.

The key point to note here is that the program logic reveals quite valuable information about the type of the data that is read, which is missing or is not explicit in the variable declaration. The algorithm presented in this paper exploits the program logic to recover a logical data model for the program that makes explicit the data abstractions that are implicit in weakly-typed programs.

Our algorithm recovers an OO data model for each "input" datum coming into a program (e.g., via a parameter, or via a READ statement). Fig. 2 shows the inferred model for the data read into CARD-TRANSACTION-REC in statement /1/. The model is drawn as a UML class diagram: each box is a class, with its name at the top, and the list of typed fields below. Each class inherits from zero or more classes (its base classes), with inheritance relationships shown as arrows from the subclass to the base class. Classes such as LocType, AtmID[3], which have no explicit fields, are called atomic classes. They represent scalar values (i.e., primitive types). The following aspects of the inferred model illustrate its value.

Record structure of a declared scalar. In our example, CARD-INFO was declared to be a scalar variable. This variable, however, is represented in our inferred model by the two fields cardType and cardDtls of CardTran, reflecting the way CARD-INFO is used in the program.

[3]Our algorithm does not automatically generate meaningful names for classes and fields. The names in Fig. 2 were supplied manually for expository purposes; however, heuristics can be used to generate such names automatically from variable names, e.g., see [27].

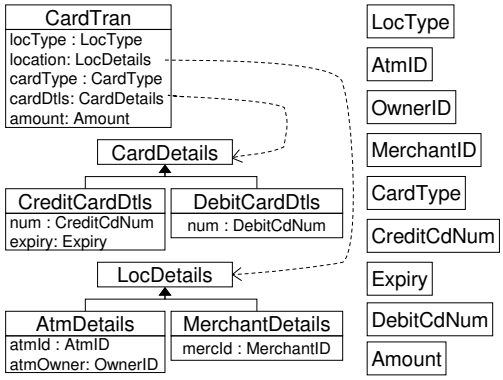

Figure 2. Object-oriented model inferred by our algorithm for the example in Fig. 1

Implicit subtyping. Furthermore, the model indicates that the type of cardDtls is the class CardDetails, which has two subtypes (derived classes) CreditCardDtls and DebitCardDtls. This naturally captures the implicitly polymorphic use of the variable CARD-INFO in the program.

Independently polymorphic variables. Note that the LOCATION-DETAILS and CARD-INFO are *independently* polymorphic: the (runtime) type of (or logical structure of the data stored in) one has no correlation to the type of the other. Multiple independent polymorphic variables are common in real programs, and a key strength of our approach is the ability to model this idiom compactly and naturally.

1.2 Guarded Dependences

One of the central aspects of our work is the use of *guarded transitive data-dependences* to perform model inference. Informally, these dependences capture information about value flow in the program, including the conditions under which such flow exists. E.g., a single dependence can capture the fact that bytes 2 through 9 of the value read in statement /1/ (i.e., the first 8 bytes of field LOCATION-DETAILS) reach the occurrence of MERCHANT-ID in statement /9/, under the condition that byte 1 (i.e., LOCATION-TYPE) contains the character 'M'. (Note that this value flow occurs via the copy statement /3/.)

In this paper, we present an efficient (polynomial time) algorithm for computing guarded transitive data-dependences, and show how the set of dependences can be distilled into an OO data model. Thus, the model inferred by our algorithm may also be viewed as a compact representation of data-dependences in the program.

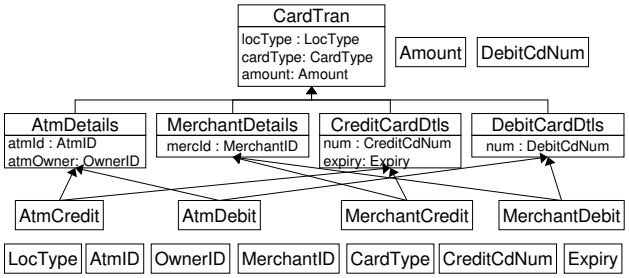

Figure 3. Model inferred by the algorithm of [18] for the example in Fig. 1

1.3 Our contributions and related work

Our first contribution is the notion of guarded transitive dependence (as well as efficient algorithms for computing these dependences). These dependences are similar to the notion of "value-point" equivalence [15], in that both track value flows through copy statements. The key difference is that value-point equivalence does not employ guards (predicates), and hence captures less information than guarded dependences. Our notion of guards is similar to the concept of path-conditions discussed by Snelting et al. [21], but differs in incorporating transitivity and addressing aspects relevant to languages such as Cobol. Our algorithms for computing these dependences are very different from the Snelting et al. approach as well.

Our next contribution is an algorithm for reverse-engineering an OO data model from programs, using these dependences. This algorithm differs from and improves upon our previous data model inference algorithm [18], referred to as the RKFS algorithm below, as follows.

Delegation vs. inheritance. Our new approach infers models that differ from the models inferred by the RKFS algorithm. The model inferred by the RKFS algorithm for our running example is shown in Fig. 3. Notice that in this multiple-inheritance based model, four derived classes, AtmCredit, AtmDebit, MerchantCredit, and MerchantDebit describe all the combinations of data that can be stored in a CardTran. In contrast, our new algorithm captures the same information using a class with two *polymorphic* fields, where each field can independently take on one of two types. Thus, the new inference algorithm produces *delegation*-based models, while the RKFS algorithm produces *inheritance*-based models.

Delegation has been suggested as the better approach for most object-oriented design settings (e.g., by the "Gang of Four" [10]). In particular, when there are multiple independent polymorphic variables, delegation can result in drastically smaller models, with no loss of precision. In fact, this is also the intuition behind why our approach is scalable.

Efficiency. Our new algorithm is polynomial time, while the RKFS algorithm always takes time exponential in the number of independent polymorphic variables. Note that a large number of independent polymorphic variables is common in Cobol programs.

Komondoor et al [12] describe an approach to compute guarded dependent types, which serves as the basis for the RKFS algorithm. This analysis takes time exponential in the number of independent polymorphic variables.

Other related work. There has been previous work by others in recovering OO data models [2, 25] and other kinds of data abstractions [7, 16, 8, 17, 3, 1] from weakly-typed programs. Some of these approaches [7, 2, 3] are based mainly on heuristics and the declared structure of data (as opposed to code analysis), Such approaches do not offer any correctness characterization or semantic properties. Several other approaches are based on code analysis. Most of these [16, 8, 17, 1] handle the idiom of *polymorphic* variables (variables that store data of different types) imprecisely. This imprecision stems from the underlying analyses, which do not exploit the order in which statements execute or the conditions under which statements execute. van Deursen and Moonen's algorithm [25] infers a subtype relation between the source and target of an assignment statement and a union type for Cobol *redefinitions*. However, their algorithm does not exploit the order in which statements execute or the conditions under which statements execute either. Other publications describe applications of such inferred models [13, 27] and describe empirical evaluation of the techniques [26].

Tip *et al* [24] and Snelting *et al* [22] present algorithms for analyzing and specializing *existing* class hierarchies in programs in object-oriented languages, but do not address the issues in inferring models from weakly-typed languages. Jhala et al [11] consider a given *union* type in a C program and *check* whether it is used safely; i.e., whether for each variant of the union there is a specific condition under which this variant is always referred to, such that the conditions governing the different variants are disjoint.

Fisher et al. [9] describe the wide prevalence of *ad hoc data formats*, and the advantages of formal descriptions of such data, including the ability to generate applications that can analyze data in the specified format. Our techniques can be used to infer such data formats automatically, from programs that consume the data. Lim et al [14] describe an approach to recover file formats from object code; their focus is on recovering the order in which entities occur in a file, whereas our focus is on recovering the record structure within and subtyping relationship between the entities.

The rest of this paper is organized as follows. Section 2 introduces terminology. Section 3 provides an informal outline of the algorithm, while Sections 4 through 6 provide the details. Section 7 describes a prototype implementation; finally, Section 8 mentions future work opportunities.

2 Terminology and Notation

A *data-source* is a statement that creates a new value in an executing program, e.g., a `READ` statement. Note that during program execution, values are generated by data-sources, and then copied around by `MOVE` statements. A program's variables occupy a statically fixed number of locations (bytes). A range is a closed interval $[i : j]$, and represents bytes i through j (inclusive). (On the other hand, the notation `VAR[I:J]` inside a Cobol program refers to bytes `I` through `J` *within* variable `VAR`.) Each variable occupies a certain range of memory locations. We will often use a program variable in places where a range is required. A *data-reference* is a specific occurrence of a program variable, or a reference to a range of memory locations, in a program statement. We will use the notation $[i : j]@S$ to indicate a data reference to locations $[i : j]$ in a statement S. Two predicates are said to be *disjoint* iff their conjunction equals false; otherwise, they are said to *overlap*.

3 Algorithm Outline

In this section, we present an informal outline of our algorithm. Details appear in subsequent sections.

3.1 Guarded Dependence Analysis

The first step in our algorithm is the computation of guarded transitive dependences. A guarded dependence of the form $g \rhd r_1@p_1 \rightsquigarrow r_2@p_2$ indicates that the value stored in a range of locations r_1 at program point p_1 may eventually (after being copied zero or more times) reside in a range of locations r_2 at program point p_2 if the condition g is true about the program state at p_1. We present a formal definition of this dependence relation in Section 4. We utilize a polynomial-time path-sensitive backward dataflow analysis to compute all guarded transitive data dependences that originate at a data-source statement and terminate at some data-reference. Section 4 has the details of this analysis.

3.2 From Guarded Dependences to Cuts

The next step towards inferring an object-oriented model is to identify what we call *cuts* at data-source statements. A cut is an abstraction of a guarded data dependence that retains information only about the source of the dependence. Specifically, for each data-source statement *st*, and for each guarded dependence $g \rhd r_1@p_1 \rightsquigarrow r_2@p_2$ originating at *st* (i.e., p_1 is the program point following *st*) and terminating at some data reference in the program (i.e., the statement that immediately follows program point p_2 refers to range r_2), we infer a cut $g \rhd r_1$ at *st*. Informally, this cut signifies that the data stored in r_1 by *st* is used

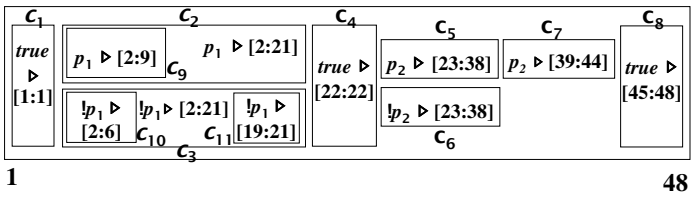

Figure 4. Cuts inferred for running example. p_1 is the predicate "`LOCATION-TYPE = 'M'`", and p_2 is the predicate "`CARD-INFO[1:1] = 'C'`".

in a particular way by the program under a certain condition (as described by predicate g on the program-state at p_1). As an example, the guarded dependence mentioned at the beginning of Section 1.2 identifies the following cut at the data-source statement /1/: `(LOCATION-TYPE = 'M')` ▷ `CARD-TRANSACTION-REC`[2:9]

3.3 The Cut-Structure Tree

Fig. 4 shows the set of all cuts inferred at statement /1/ of our running example. (As we will explain soon, the cut c_8 actually represents multiple inferred cuts.) The outermost rectangle represents the entire data read at statement /1/, while each inner rectangle represents a cut; the predicate of each cut, and its range (relative to the outermost rectangle), are written inside the cut's representing rectangle.

The horizontal and vertical axis of each rectangle represents the range and predicate of the corresponding cut, respectively. Cuts that occupy overlapping ranges, e.g., c_2 and c_3, overlap horizontally; cuts that occupy non-overlapping ranges, e.g., c_1 and c_2, do not overlap horizontally; and cuts that have "disjoint" predicates, e.g., c_2 and c_3, do not overlap vertically. (However, ensuring the converse will complicate the figure and so we do not attempt this; so cuts that do not overlap in the vertical dimension may have overlapping predicates; e.g., cuts c_3 and c_5.)

After inferring cuts at a data-source, our algorithm organizes them into a tree structure, which we call the cut-structure tree. Fig. 4 shows this tree structure, using nesting among the rectangles to represent parent-child relationships. The tree satisfies the following properties:

- A cut $p_1 \rhd r_1$ is an ancestor of a cut $p_2 \rhd r_2$ iff p_2 implies p_1 and r_2 is equal to r_1 or a subrange of r_1.
- If a cut c_i is not an ancestor or descendant of a cut c_j, then either their predicates are disjoint or their ranges are disjoint.

We say such a cut-structure tree is *well-structured*. In general, the set of cuts initially generated for a data-source may not form a well-structured tree of cuts. In such cases, we merge certain cuts and widen the predicates of certain cuts, as described in Section 5, and try to produce a well-structured tree. This step will succeed unless there exist

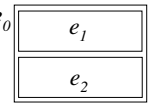

Figure 5. The disjoint-union model strategy

cuts with overlapping predicates and improperly overlapping ranges. Failure typically indicates that the predicates of the cuts are not precise enough; this happens if the abstraction used for path-sensitivity (which is a parameter to our analysis – see Section 4.2) is not precise enough.

In our running example, the three different references to `AMOUNT` produce three different cuts at statement /1/, all with the range $[45 : 48]$ but with different predicates, namely `CARD-INFO[1:1] = 'C'`, `CARD-INFO[1:1] <> 'C'`, and `LOCATION-TYPE = 'M'`. Because the predicate for the third cut overlaps the predicates for the other two cuts, the three cuts are merged into the single cut $c_8 = \mathit{true} \triangleright [45 : 48]$, producing a well-structured tree.

3.4 From Cuts to a Class Hierarchy

Cuts identify logically cohesive units of data. We create a data model by creating classes to represent cuts in the cut-structure tree with appropriate relations between these classes, as shown below.

Atomic Data. An innermost cut (i.e., a leaf of the cut structure tree), such as cut c_{10}, represents *atomic data*: that is, the program does not ever refer to any proper subsequence of the data represented by the cut. We represent such cuts using atomic classes. These represent the primitive logical types in the model. In our example, cut c_{10} corresponds to the class AtmID (see Fig. 2).

Structured Data. The cut c_3 in Fig. 4 has two child cuts c_{10} and c_{11}, represented, respectively, by atomic classes AtmID and OwnerID in Fig. 2. c_3 represents structured data consisting of the two parts c_{10} and c_{11} and is modeled naturally as a class AtmDetails with two fields of types AtmID and OwnerID. As this example illustrates, a cut, such as c_{10}, may generate, in general, both a *class* (representing its type) as well as a *field* (in its parent cut's class). We refer to this strategy (of creating a class consisting of one field for each child cut) as the *concatenation model strategy*.

Subtyping. Consider the example shown in Fig. 5. The two leaf cuts e_1 and e_2, which have *disjoint* predicates, can be modeled by two classes E_1 and E_2 respectively. However, because the two cuts are never alive *simultaneously*, the parent cut e_0 may contain either e_1 or e_2 at any time, but never both. This example is best modeled by making E_1 and E_2 *derived* classes of the class E_0 used to model e_0. We refer to this strategy, which is applicable only on cuts whose children all have disjoint predicates, as the *disjoint-union model strategy*.

Factoring. Consider the cut c_0 (the outermost box), and its children $c_1, c_2, \ldots, c_8$, in Fig. 4. The child cuts don't all have the same predicate, nor are all their predicates disjoint. Informally, we could think of this tree as being *incomplete*: it does not reveal the complete structure/grouping of the children of c_0. In such situations our approach tries to insert new cuts into the cut structure tree, such that, to the extent possible, the children of each cut either all have the same predicate as the parent (i.e., the parent can be modeled using the concatenation strategy with no loss of precision), or all have disjoint predicates (the parent can be modeled using the disjoint-union strategy).

We call this step *factoring*; we present the details of this in Section 6, but illustrate the results of factoring cut c_0 in Fig. 7. The top part of the figure shows c_0 and its children as they are originally, while the bottom part shows the factored tree (ignore the middle part, for now). Note the newly introduced nodes labeled "U", "U", and N_3. The nodes labeled "U" have children with disjoint predicates and are modeled using the disjoint-union strategy; all remaining nodes are modeled using the concatenation strategy. The resultant model for our running example is shown in Fig. 2.

4 Guarded Dependence Analysis

In this section we introduce the concept of a *guarded demonic dependence* and present an analysis to compute these dependences.

4.1 Guarded Demonic Dependences

Let s_{in} and s_{out} denote the program points just before and after a statement s respectively. Let p_1, p_2 denote program points, and let r_1, r_2 denote ranges, and let g denote a predicate (over the program state). A guarded demonic dependence is of the form $g \triangleright r_1@p_1 \rightsquigarrow r_2@p_2$, and indicates that the data residing in the range of locations r_1 at p_1 may eventually (after being copied zero or more times) reside in the range of locations r_2 at p_2 if g is true at p_1.

The distinguishing characteristics of these dependences are (a) they capture data dependences that are transitive over `MOVE` statements (similar to the notion of value-point equivalence [15]), (b) they capture only those transitive flows where the byte sequence is copied in its entirety (not in portions) by the intervening copy assignments, (c) they may involve arbitrary (memory) ranges, not just variables, (d) they have guards, indicating the conditions under which the dependence may be manifested, and (e) the dependences are "demonic", as illustrated by the example below.

```
s: READ [1:5].
MOVE "AB" TO [1:2].
MOVE "CDE" TO [3:5].
t: WRITE [1:5].
```

The dependence $true \rhd [1:5]@s_{out} \rightsquigarrow [1:5]@t_{in}$ holds for the above program. Specifically, in this dependence computation, we treat an assignment statement as "killing" the data under consideration only if it completely overwrites the data. But a sequence of statements that each partially overwrites the data is not considered to "kill" the data under consideration even if they collectively overwrite the complete data. Informally, the above dependence indicates that in a non-standard semantics of the program that exploits a strongly-typed object-oriented representation of data, the object created at s may reach the use at t. The intervening assignments merely update the values of fields of the object.

4.2 Backward Dependence Computation

In this section, we show how we can identify (a conservative over-approximation of the) guarded demonic dependences $g \rhd r_1@p_1 \rightsquigarrow d$ that terminate at data-references d occurring in the program, using a backward dataflow analysis. We refer to the computed set of dependences as *Deps*. We present a sequence of analyses, of increasing sophistication, to compute these dependences. Each analysis is described by a semi-lattice $(\mathcal{L}, \sqcup)$ and a function $\alpha[S] : \mathcal{L} \rightarrow \mathcal{L}$ that maps every statement S to a function from $\mathcal{L}$ to $\mathcal{L}$. This determines the following collection of equations over all statements S in the program (where $succs(S)$ denotes the set of successors of S), whose least fixed point is computed using standard techniques:

$$\begin{aligned} S_{in} &= \alpha[S](S_{out}) \\ S_{out} &= \bigsqcup_{T \in succs(S)} T_{in} \end{aligned}$$

A Path-Insensitive Dependence Computation We first present a path-insensitive analysis for computing dependences without guards (i.e., whose guards are *true*). We abbreviate the dependence notation by dropping the guard g in this setting. Let *DataRef* denote the set of all data-references in the given program. Let M denote the number of locations used by the program. Let *Ranges* denote the set $\{\ [i:j] \mid 1 \le i \le j \le M\ \}$. Let $\mathcal{D}_{pi}$ denote the set $Ranges \times DataRef$, which is the domain of demonic dependences without guards. We will overload our notation and depict an element (r, d) of $\mathcal{D}_{pi}$ as $r \rightsquigarrow d$.

The analysis computes for every statement s an over-approximation of the set $\{\ r \rightsquigarrow d \in \mathcal{D}_{pi} \mid r@s_{in} \rightsquigarrow d\ \}$. We utilize a backward analysis with the powerset lattice $2^{\mathcal{D}_{pi}}$, with set-union as the join operation, for this purpose. Table 1 shows the abstract state transformer semantics used for the backward analysis. (Conditional branches are encoded using `ASSUME pred` statements attached to branches, where `pred` describes the condition under which the branch is executed. Other forms of Cobol statements can be reduced to primitives of the form shown here.)

Statement S	$\alpha_{pi}[S] : 2^{\mathcal{D}_{pi}} \rightarrow 2^{\mathcal{D}_{pi}}$
`WRITE` $\mathtt{Y}^d$	λOut. Out $\cup \{\ Y \rightsquigarrow d\ \}$
`READ` $\mathtt{Y}^d$	λOut. $\{\ Y \rightsquigarrow d\ \} \cup$ $\{\ r \rightsquigarrow t \mid r \rightsquigarrow t \in$ Out and $r \not\subseteq Y\}$
`MOVE` $\mathtt{X}^{dX}$ `TO` $\mathtt{Y}^{dY}$ where Y = $[y_1 : y_2]$ and X = $[x_1 : x_2]$	λOut. $\{\ X \rightsquigarrow dX, Y \rightsquigarrow dY\ \} \cup$ $\{\ r \rightsquigarrow t \mid r \rightsquigarrow t \in \text{Out} \wedge r \not\subseteq Y\} \cup$ $\{\ r' \rightsquigarrow t \mid r \rightsquigarrow t \in \text{Out}, r \subset Y,$ and $r' = r - y_1 + x_1\ \}$
`ASSUME pred`	λOut. Out $\cup \{r \rightsquigarrow d \mid$ pred contains a data-reference d to a range $r\ \}$

Table 1. Path-insensitive statement abstraction. Superscripts on data-references are used to identify them in the second column.

This analysis can be implemented efficiently, in polynomial time. Note that for a given target data-reference, at most $O(M)$ dependences can exist at a program point (though, in practice, one would expect the number of such dependences to be far smaller).

Making the Analysis Path-Sensitive We now extend the above analysis to compute dependences *with guards*. Our approach to path-sensitivity is similar to the one presented in [5] and is parametric over the abstraction used for the guards. Assume that we are given a lattice $\mathcal{G}$ of guards, and a corresponding abstraction function $\alpha_g[t] : \mathcal{G} \rightarrow \mathcal{G}$ for every program statement t that conservatively approximates the weakest-precondition semantics of t: i.e., the state before execution of t must satisfy $\alpha_g[t](g)$ if the state after execution of t is to satisfy g. Let $\sqcup_{\mathcal{G}}$ and $\sqcap_{\mathcal{G}}$ be the join and meet operations of the lattice. (Since elements of $\mathcal{G}$ represent predicates, these operations are conservative approximations of logical disjunction and logical conjunction.)

As an example, a lattice that is useful in practice allows conjunctions of elementary guards of the form $v = c_1$ and $v \notin \{\ c_1, \cdots, c_k\ \}$ where v is a variable and each c_i is a constant. It is sufficient, in practice, if we consider only constants c_i explicitly mentioned in the program. This yields a useful guard lattice of polynomial height.

Exponential Path-Sensitivity We first sketch a straightforward way to augment the dependences propagated by the path-insensitive algorithm with a guard. Let $\mathcal{D}_{eps}$ denote the set $\mathcal{G} \times Ranges \times DataRef$. The goal of the analysis is to compute for every statement s an over-approximation of the set $\{\ g \rhd r \rightsquigarrow d \in \mathcal{D}_{eps} \mid g \rhd r@s_{in} \rightsquigarrow d\ \}$. We will utilize the powerset lattice $2^{\mathcal{D}_{eps}}$, with set-union as the join operation, as the abstraction lattice. The abstract transformer $\alpha_{eps}[t] : 2^{\mathcal{D}_{eps}} \rightarrow 2^{\mathcal{D}_{eps}}$ for a statement t can be

defined in terms of the path-insensitive abstraction and the guard abstraction as follows:

$$\alpha_{eps}[t](\text{Out}) = \{true \rhd r' \rightsquigarrow d' \mid r' \rightsquigarrow d' \in \alpha_{pi}[t]\{\}\} \bigcup \{\alpha_g[t](g) \rhd r' \rightsquigarrow d' \mid g \rhd r \rightsquigarrow d \in \text{Out}, r' \rightsquigarrow d' \in \alpha_{pi}[t]\{r \rightsquigarrow d\}\}$$

The first term says that any dependence "generated" by the statement has an associated guard "true". The second term considers any dependence $g \rhd r \rightsquigarrow d$ known to exist after the statement. If the backward propagation of the dependence $r \rightsquigarrow d$ by the path-insensitive algorithm produces the dependence $r' \rightsquigarrow d'$ before the statement, then we will generate the guarded dependence $\alpha_g[t](g) \rhd r' \rightsquigarrow d'$ (where the guard $\alpha_g[t](g)$ is obtained by transforming the guard g using the guard abstraction).

Note that the guard lattice described earlier has a polynomial height but an exponential number of elements. As a result, the powerset lattice $2^{\mathcal{D}_{eps}}$ has an exponential height, and the above analysis can take exponential time.

Polynomial Path-Sensitivity Note that for a given range r and data-reference d, the above analysis may generate (possibly exponentially) many dependences $g \rhd r \rightsquigarrow d$ at a given program point. These different dependences capture different paths along which the dependence $r \rightsquigarrow d$ may be manifested. We avoid this exponential blow-up by allowing only one guard per dependence at a program point: i.e., if two guarded dependences $g_1 \rhd r \rightsquigarrow d$ and $g_2 \rhd r \rightsquigarrow d$ arise at a program point, the two are approximated by the single guarded dependence $(g_1 \sqcup_{\mathcal{G}} g_2) \rhd r \rightsquigarrow d$.

Formally, we use the lattice $\mathcal{D}_{pps} = \mathcal{D}_{pi} \rightarrow \mathcal{G}$: the lattice elements are maps from $\mathcal{D}_{pi}$ to $\mathcal{G}$, and the join $m_1 \sqcup m_2$ is defined to be $\lambda x.m_1(x) \sqcup_{\mathcal{G}} m_2(x)$. A map m represents the fact that the dependence $r \rightsquigarrow d$ holds at a point only when $m(r \rightsquigarrow d)$ holds true at that point. The abstract state transformer used for statements is automatically induced by this abstraction, as outlined below.

We define the function *set-to-map* : $2^{\mathcal{D}_{eps}} \rightarrow \mathcal{D}_{pps}$ that merges a set of guarded dependences into a map as follows: *set-to-map*$(S) = \lambda(r \rightsquigarrow d).\sqcup_{\mathcal{G}} \{g \mid g \rhd r \rightsquigarrow d \in S\}$. We define the function *map-to-set* : $\mathcal{D}_{pps} \rightarrow 2^{\mathcal{D}_{eps}}$ as follows: *map-to-set*$(m) = \{m(r \rightsquigarrow d) \rhd r \rightsquigarrow d \mid r \in$ *Ranges*$, d \in DataRef\}$. We now define the abstract state transformer $\alpha_{pps}[t] : \mathcal{D}_{pps} \rightarrow \mathcal{D}_{pps}$ as follows:

$$\alpha_{pps}[t](\text{Out}) = \textit{set-to-map}(\alpha_{eps}[t](\textit{map-to-set}(\text{Out})))$$

5 Generating cuts and the cut-structure tree

As mentioned in Section 3.2, guarded dependences originating at the program point following a data-source and terminating at a data reference are used to identify cuts at that data source. Let s be a data-source and r the range that is assigned a value at s. The set of cuts *Cuts*(s) is defined to be $\{true \rhd r\} \cup \{g \rhd r' \mid r' \subset r, g \rhd r' @ s_{out} \rightsquigarrow d \in$ *Deps*$\}$, where *Deps* is the set of guarded dependences computed by the algorithm presented in Section 4.

We then attempt to organize the cuts in *Cuts*(s) as a tree. A cut $p_1 \rhd r_1$ is defined to be an ancestor of a cut $p_2 \rhd r_2$ iff p_2 implies p_1 and r_2 is equal to r_1 or a subrange of r_1. Our algorithm requires this relation to induce a tree (rather than DAG) structure on the cuts *and* requires the tree to be *well-structured* (see the definition of this in Section 3.3). If it does not, we repeatedly apply the following two transformations until either the cuts become a well-structured tree or no more transformations apply:

Cut Merging: Replace any two cuts $p_1 \rhd r$ and $p_2 \rhd r$ in *Cuts*(s) such that p_1 and p_2 overlap but neither one implies the other by the single cut $p_1 \sqcup_{\mathcal{G}} p_2 \rhd r$.

Cut Widening: Consider any pair of cuts $c_a = p_a \rhd r_a$ and $c_d = p_d \rhd r_d$ in *Cuts*(s) such that r_a is a super-range of r_b and p_d overlaps but *does not imply* p_a. We then replace c_a by $p_a \sqcup_{\mathcal{G}} p_d \rhd r_a$.

We consider all cuts in increasing order of their range size and apply the above transformations if applicable. If the resulting cuts do not form a well-structured tree our algorithm halts with failure (does not produce a model).

6 Factoring the cut-structure tree and producing the class hierarchy

We now present an algorithm for *factoring* a given cut-structure tree, which may add new nodes to the tree while preserving the ancestor-descendant relationship on the original nodes. In addition, it labels all non-leaf nodes in the resulting tree as *disjoint-union* nodes or *concatenation* nodes, where distinct children of a disjoint-union node are guaranteed to have disjoint predicates. The goal of the transformation is to capture as much disjointness information as possible (in the form of disjoint-union nodes) to enable the generation of a better data model, as explained in Section 3.4. Note that the newly inserted nodes will have no predicates (though it is possible to compute predicates for them).

The algorithm is a top-down traversal of the tree. (See Fig. 6 for a pseudo-code description of the algorithm.) When we visit a node p, we attempt two different transformations of p and it's children as follows.

Disjoint Union Identification: In this transformation, we create a graph consisting of the children of p as vertices. We add edges between any two children that have an overlapping predicate. (We will refer to this graph as the *overlap-edge graph*.) If the resulting graph has two or more connected components, then p is transformed into a disjoint-union node by replacing all its children by a set of new

```
process (p : node) {
    if (! disjointUnion(p)) then horizontalPartition(p) endif
    for every child c of p do process (c) endfor
}
boolean disjointUnion(p : node) {
    C = children(p)
    E = { (u, v) ∈ C × C | pred(u) and pred(v) overlap }
    CC = connected components of graph <C, E>
    if |CC| > 1 then
      for every cc in CC do makeSubGroup(p, cc) endfor
      mark p a disjoint union node; return true
    else
      mark p a concatenation node; return false
    endif
}
makeSubGroup (p: node; group: subset of p's children) {
    if (|group| > 1) and (group ≠ children(p)) then
      create a new node g; add g to children(p)
      for each c in group do
        remove c from children(p); add c to children(g)
      endfor
    endif
}
horizontalPartition (p: node) {
    C = children(p)
    E = { (u, v) ∈ C × C | pred(u) and pred(v) are disjoint }
    CC = connected components of graph <C, E>
    for each cc in CC do makeSubGroup (p, cc) endfor
}
```

Figure 6. The factoring algorithm.

nodes, one for each connected component. All original children of p are made children of the corresponding connected component node. (Note that this creation of extra nodes is avoided for connected components with a single node, as detailed in Fig. 6.) If the graph has only one connected component, then the transformation fails, and we attempt the next transformation on p.

This transformation captures the disjoint-union model strategy presented in Section 3.4: distinct connected components of the overlap-edge graph represent cuts (or data) that do not exist simultaneously, which are naturally modeled as distinct derived classes of a common base class.

Horizontal Partitioning: This is almost the dual of the previous step. We construct a graph consisting of the children of p as the vertices, with an edge between any two nodes that have disjoint predicates. (We will refer to this graph as the *disjoint-edge graph*.) We identify the connected components of this graph. We replace the children of p with new nodes, one for each connected component. As before, we avoid the creation of unnecessary nodes (either if there is only one connected component, or the connected component contains only one vertex).

The horizontal partitioning essentially implements the concatenation model strategy described in Section 3.4.

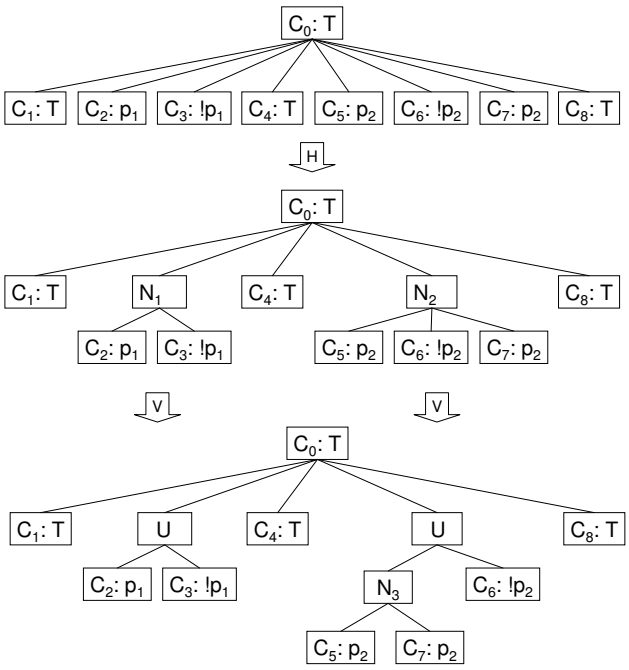

Figure 7. Illustration of the factoring algorithm. *true* **has been abbreviated to T.**

However, it first identifies sub-groups of children to which the disjoint-union model strategy may (subsequently) apply. These are essentially the connected components of the disjoint-edge graph. (The iterative traversal of the tree will later process these sub-groups appropriately and produce disjoint-union nodes if possible.)

Algorithm Illustration. Fig. 7 illustrates how factoring works for part of our example. The figure shows the root node and its children of the cut-structure tree (see Fig.4). The disjoint-union transformation is not applicable at the root node C_0 since its children form a single connected component in the overlap-edge graph. We apply horizontal partitioning to C_0: The disjoint-edge graph produces two non-trivial connected components, $\{ C_2, C_3 \}$ and $\{ C_5, C_6, C_7 \}$. Creating new nodes N_1 and N_2 for these components produces the tree shown next. (Each of the remaining children C_1, C_4, and C_8 forms its own connected component and is not affected by the transformation.)

We recursively visit the children of C_0 (in the transformed tree). Consider how N_2 is processed. The disjoint-union transformation identifies two connected components $\{ C_5, C_7 \}$ and $\{ C_6 \}$ in the corresponding overlap-edge graph. Creating a new node for the component $\{ C_5, C_7 \}$ produces the tree shown next. Node N_2 is also marked as a disjoint-union node (shown in the figure by simply labeling it "U"). Node N_1 is also similarly marked as a disjoint-union node. All other remaining nodes are marked as concatenation nodes (which is not shown in the figure).

Generating the class hierarchy. Generating a class hi-

Prog.	LOC	# stmts	# vars	# data refs	data size	time (sec)
A	968	210	391	225	8728	20
B	2448	211	882	213	9846	29
C	2780	555	1275	746	27136	190

Figure 8. Summary of real programs

erarchy from a factored cut-structure tree is straightforward. Create a class C_n for each node n in the tree. Let $c_1, \cdots, c_k$ be the children of n. If n is labeled as a disjoint-union node, then make each C_{c_i} a derived class of C_n, otherwise, create a field f_{c_i} of type C_{c_i} in C_n for $1 \leq i \leq k$.

Finally, we repeatedly apply a few rules that simplify the class diagram: (a) if all subtypes of a base type have leaf fields that all pertain to the same memory range, move up these fields to a unified field in the base type, (b) eliminate subtypes all of whose fields get moved up, and (c) merge a base type that has only one subtype with its sub type.

7 Implementation

We have implemented a prototype of our algorithm. The output of the prototype tool is a class hierarchy (model) of the data read at each data-source in the given program. The implementation addresses the (full) Cobol language. The initial step (guarded dependence analysis) is implemented in Java. We used the constant propagation lattice (including predicates of the form $v \notin \{ c_1, \cdots, c_k \}$) as the guard lattice, as described in Section 4.2; we address procedure calls context sensitively in this analysis by annotating dataflow facts with *call strings* [20]. This is feasible because Cobol does not allow recursion. (We believe that the IFDS approach of [19] can be used to handle procedures more efficiently, but this is future work.) The subsequent steps of the algorithm, namely generating the cut-structure tree, and cut-structure tree factoring, are implemented in OCaml.

We ran the tool on several test-cases we wrote as well on a few real programs (from two separate applications from the financial industry). We verified the algorithm output on our test-cases; e.g., for the program in Fig. 1, the tool produced the class diagram shown in Fig. 2. Fig. 8 provides statistics relating to input size and analysis time for the three real programs that we analyzed. The columns in this figure, from left to right, are the number of lines of code (LOC), number of Cobol statements, number of declared variables, number of data references in statements, and total data size of the program (sum of sizes of all declared variables). The last column gives the analysis time on a laptop with Intel Centrino 1.8 GHz processor and 1.5 Gig of RAM.

Programs A, B, and C had three, four, and four data sources, respectively. The implementation successfully produced a class diagram for 10 of the 11 data sources. Fig. 9 presents statistics for 5 of the 10 class diagrams produced,

	Class diagrams				
	1	2	3	4	5
#bytes in data source	1500	1075	200	4000	80
# classes	72	27	12	9	5
# leaf classes	71	18	7	5	3
# classes w/ fields	1	5	4	3	2
# base classes	0	4	1	1	0
# derived classes	0	9	3	2	0
# fields	71	17	8	6	4

Figure 9. Summary of five class diagrams

with one class diagram per column. (We omit the 5 smallest class diagrams due to space limitations – in total these 5 diagrams contained only 11 classes). The first row shows the size of the variable initialized at the data source. The second row is the total number of classes in the class diagram; every class is either a leaf class, or a class with fields, or a base class. (Our algorithm produces single-inheritance models where base classes do not have fields and derived classes are themselves not base classes.) The last row shows the total number of fields in the structured classes.

We manually inspected the class diagram summarized in Column 2 of Fig. 9 (from Program B), to validate it. The inferred model consisted of a "root" class. The type of one field of the root class was a base class with three derived classes; the types of all other fields in the entire class diagram were leaf classes. We then inspected program B and observed that the inferred class diagram modeled an input parameter v of the program, and that it was a good model of v. The leaf fields in the "root" class corresponded to fields of v that the program unconditionally defined (v was a reference parameter). The three subtypes corresponded to three distinct groups of fields within v. Depending on the value of another "flag" parameter, the program assigned values to fields of exactly one of these three groups.

The model concisely shows what fields in v are referred to in the code, and which ones are accessed under the same condition. This information is not obvious from the code because (a) only a small percentage of v's fields are referred to in the program, and these are not declared contiguously, and (b) only a small percentage of the program's statements refer to v, and these statements are not contiguous.

We also inspected Program C, which had the (sole) data source for which the algorithm failed. The failure was because of two cuts at this data source with overlapping predicates and improperly overlapping ranges (see the discussion in Section 3.3). A powerful guard lattice (e.g., one that can represent disjunctions such as $v = c \ \vee \ w = d$) would avoid this problem. Such a powerful guard lattice may be required only in a small number of cases (as evidenced by our study). Thus, it may be useful in practice to try model inference with a simpler guard lattice first, and resort to a more sophisticated lattice if the simpler lattice fails.

8 Semantics and Extensions

We have presented an algorithm for inferring an OO data model for the input data of programs. There is a semantic basis for our approach, and a corresponding notion of correctness of the model inferred. Specifically, every part (subsequence) of the input data at a source statement that is referred to in the program (potentially after copying and moving the data around) is guaranteed to yield a field (potentially nested) in the data model inferred for that source. (As a special case, this implies that any part of the data that is described by a leaf class in the model is guaranteed to be treated *atomically* by the program: i.e., the program always treats such datum as a unit, and never references a *part* of the datum.) Furthermore, every derived class in the model is associated with a predicate; these predicates can be conjoined (in the presence of nested fields) to describe the *condition* under which (the data corresponding to) any field in the model may be referenced in the program. We omit a more detailed discussion due to space limitations.

A natural extension of this algorithm would be to integrate the data models inferred for different data-sources, as well as to infer types of variables at every point in the program. We believe this can be done, e.g., using some of the ideas that appear in [18].

Acknowledgments

We thank John Field and Saurabh Sinha of IBM Research for their contributions to previous joint work, upon which the current work builds. We also thank Saurabh for help with the implementation.

References

[1] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Proc. Intl. Conf. Compiler Constr.*, pages 5–23, 2004.

[2] 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.

[3] G. Canfora, A. Cimitile, A. D. Lucia, and G. A. D. Lucca. Decomposing legacy systems into objects: an eclectic approach. *Information & Software Technology*, 43(6):401–412, 2001.

[4] S. Chandra, J. de Vries, J. Field, H. Hess, M. Kalidasan, K. V. Raghavan, F. Nieuwerth, G. Ramalingam, and J. Xue. Using logical data models for understanding and transforming legacy business applications. *IBM Syst. J.*, 45(3):647–655, 2006.

[5] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 57–68, June 2002.

[6] H. M. Edwards and M. Munro. Recast: Reverse engineering from cobol to ssadm specification. In *Proc.* 1^{st} *Working Conf. on Reverse Engg.*, pages 44–53, 1993.

[7] H. M. Edwards and M. Munro. Deriving a logical data model for a system using the recast method. In *Proc.* 2^{nd} *Working Conf. on Reverse Engg.*, pages 126–135, 1995.

[8] 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.

[9] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *Proc. Symp. on Principles of Prog. Langs.*, pages 2–15, 2006.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[11] R. Jhala, R. Majumdar, and R.-G. Xu. State of the union: Type inference via craig interpolation. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[12] 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.

[13] T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Proc. 8th Intl. Workshop on Program Comprehension*, page 221, 2000.

[14] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *Proc.* 13^{th} *Working Conf. on Reverse Engg.*, pages 167–178, 2006.

[15] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 2001.

[16] 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.

[17] 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.

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

[19] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. Symp. on Principles of Prog. Langs.*, pages 49–61, San Francisco, California, 1995.

[20] M. Sharir and A. Pneuli. Two approaches to inter procedural dataflow analysis. In S. S. Muchnik and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice Hall, Englewood Cliffs, NJ, 1981.

[21] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

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

[23] P. Thiran and J.-L. Hainaut. Wrapper development for legacy data reuse. In *Proc.* 8^{th} *Working Conf. on Reverse Engg.*, pages 198–207, 2001.

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

[25] A. van Deursen and L. Moonen. Type inference for cobol systems. In *Proc. Working Conf. on Reverse Engg. (WCRE'98)*, pages 220–230, 1998.

[26] A. van Deursen and L. Moonen. An empirical study into cobol type inferencing. *Sci. Comput. Program.*, 40(2–3):189–211, 2001.

[27] A. van Deursen and L. Moonen. Documenting software systems using types. *Sci. Comput. Program.*, 60(2):205–220, 2006.