# Report on Dataflow Dependencies in Billing Processing Systems

Parosh Abdulla* Noomene Ben Henda*
Johann Deneux* Bengt Jonsson* Tomas Reidmar†

**Abstract**

Data processing systems in business applications typically consist of a number of database modules and a number of data processing modules. Upgrading such systems requires changing the functionalities of individual modules. It is therefore necessary to investigate the impact of these changes. In particular, one must know the dataflow dependencies between input and output parameters. This is hard in practice since designers usually do not provide accurate specifications for their systems. In this paper, we present a tool that learns dataflow dependencies of a processing module by using blackbox techniques on supplied test data. We have applied the tool on a billing system provided by our industrial partner.

## 1 Introduction

Data processing systems in business and administrative applications are often large and complex, They typically consist of several database modules and a number of data processing modules, interconnected to process information about transactions etc., to generate updates to the system together with customer bills etc. A particular application in the telecommunications area is billing processing systems of telecommunication operators, which generate of customer bills from information about performed telephone calls and customer pricing profiles. For each telephone call, a network switch generates a so-called toll ticket containing relevant information about the call. The

*Department of Computer Systems, Uppsala University
†WM Data AB

billing processing system processes toll-tickets, combines their information with customer information stored in databases, and through a sequence of processing steps ultimately generates customer bills.

The company WM Data AB, formerly Validation AB, has been concerned with testing of billing processing systems, including to test upgrades. Often an upgrade concerns only one module, but it must anyway be verified that the entire system functions appropriately. This testing effort becomes substantially easier if one understands the impact of the upgrade on other modules. For instance, if an upgrade changes the meaning of a particular field in an output record of a module is changed, then the effect of that change on other modules must be investigated. Possibly the other modules must be change to accommodate the change of semantics of one field in their input. For instance, an upgrade may change the way that a particular field in an output record is calculated. other modules that (directly or indirectly) use this particular field must be upgraded and/or retested. The testing effort becomes manageable if one knows how inputs of other modules depend on output fields that have been affected by the upgrade.

To determine exactly how different data field depend to each other is, of course, infeasible in general, but a useful approximation can be obtained in the form of dataflow dependencies between inputs and outputs of each particular module. In principle, this information can be obtained by inspecting specifications of modules and using implicit knowledge possessed by their developers and maintainers. This is a costly manual process, and it would be desirable to obtain dataflow dependencies by automated techniques. There is a large literature on program analysis techniques for calculating dataflow, or information flow, properties from program text, e.g., (* references *). For our application, this solution is less attractive for the reason that different modules are written in different source languages, causing a high retargeting cost. We have therefore turned our attention to black-box techniques, aiming to extract dataflow dependencies by observing input-output behavior of modules.

Let us describe the problem in a little more detail. Each telephone call generates a *toll ticket* from one of the network switches. A toll ticket is a large record (typically $50 - 100$ fields), containing information about the performed call. A data processing module processes one or several such records together with database records containing information about pricing policies for subscribers to generate another similar record where some information concerning the price of the telephone call has been synthesized. By a *dataflow dependency*, we mean that the value of a particular output field may depend on the value of a particular input field. In some cases, a dataflow dependency may be present only when some particular condition on the other input fields is satisfied. Our primary goal is to extract the dataflow dependences for a module by inspecting a large sample of its input-output behavior. It is clear that black-box techniques can not guarantee to find all dataflow dependencies, in contrast to program analysis (white-box) techniques. However, black-box techniques have lower retargeting costs, and it is reasonable to assume that the information obtained will save significant costs when testing module upgrades.

In our work, we have developed algorithms to detect dataflow dependencies by analyzing samples of input-output data. One of these is based on the simple idea that a dataflow dependency is "witnessed" by two different input records that differ only in the value of a particular input field; the fields that differ in the corresponding output records must then depend on this input field. This idea gives rise to a simple procedure that compares pairs of input and corresponding output records. The problem is that in a given sample of reasonable size (on the order of $10^5$ - $10^7$ input/output samples), there do not in general exist pairs of input records that differ only in the value of one particular input field. Our solution to this has been to synthesize new input data with this property, guided by an analysis of a given sample.

We have implemented our algorithms for detecting dataflow dependencies in large samples of input/output behavior, and applied it to a module in an actual billing processing system, under test by WM Data AB. The tool was able to detect "all" (in a sense which we defined precisely in the paper) dependencies that were visible in a sample of $300,000$ input/output pairs. For practical reasons, we were not able to use samples that revealed some specific customer profiles, meaning that our analysis of the module itself was not complete.

The rest of the paper is structured as follows. In the next section, we formalize the problem of detecting dataflow dependencies together with other concepts. The algorithm for comparing pairs of input/output is presented in Section **??**. The application to a module in a billing processing system is presented in Section **??**. Section **??** contains conclusions and directions for future work.

## 2 Problem Statement

In this section, we give precise formulations as a basis for the further description in Sections ??. In general, we consider data processing modules the take one or several input records and produce one or several output records, where the data in each field of the output records is computed from the data of some fields in the input records. Let us abstract from the fact that there may be different input and output records, and just regard a data processing module as a function $f$ from a fixed number of input parameters $x_1, x_2, \ldots, x_m$ to a fixed number of output parameters $y_1, y_2, \ldots, y_o$. An *i/o-pair* (of $f$) is a pair $\langle(\mathtt{x}_1, \ldots, \mathtt{x}_m), f(\mathtt{x}_1, \ldots, \mathtt{x}_m)\rangle$ consisting of a record of input values and a record with corresponding output values. An *i/o-sample* is a set of i/o-pairs. Throughout, we will use italic font ($x_i$, $y_j$, etc.) to denote the formal parameters of $f$, and typewriter font ($\mathtt{x}_i$, $\mathtt{y}_j$, etc.) to denote actual values of these parameters.

We are interested in finding out whether a particular output parameter $y_j$ may depend on a particular input parameter $x_i$. For instance, if the module calculates $y_7$ as $y_7 = x_3 + x_8$, then we say that $y_7$ may depend on $x_3$ and $x_8$. However, the dependency may be visible only under certain conditions. For instance, $y_7$ is calculated as $y_7 = \textbf{if } x_4 = 7 \textbf{ then } x_3 + x_8 \textbf{ else } 5$, then we still say that $y_7$ may depend on $x_3$ and $x_8$ although this can be observed only when $x_4 = 7$ (hence $y_7$ also depends on $x_4$). We formalize the relation "may depend on" as a binary relation $\mathcal{D}$ which contains the pair $(i, j)$ if there is a particular input record with the property that when the value of the $i$th input parameter changes, then the value of the $j$th output parameter changes. In the above example, the relation $\mathcal{D}$ contains at least the pairs. $(3, 7)$ and $(8, 7)$.

Define the *distance* between two input records $(\mathtt{x}_1, \ldots, \mathtt{x}_m)$ and $(\mathtt{x}'_1, \ldots, \mathtt{x}'_m)$ as the number of indices $i$ such that $\mathtt{x}_i \neq \mathtt{x}'_i$.

**Definition 2.1** Given a function $f$ from $m$ input parameters to $o$ output parameters, the relation $\mathcal{D} \subseteq \{1, \ldots, m\} \times \{1, \ldots, o\}$ (pronounced "may depend on") contains all pairs $(i, j)$ for which there are values $\mathtt{x}_1, \ldots, \mathtt{x}_{i-1}, \mathtt{x}_{i+1}, \ldots, \mathtt{x}_m$ of the other input parameters, and two values $\mathtt{x}_i$ and $\mathtt{x}'_i$ of $x_i$ such that

$$f_j(\mathtt{x}_1, \ldots, \mathtt{x}_i, \ldots, \mathtt{x}_m) \neq f_j(\mathtt{x}_1, \ldots, \mathtt{x}'_i, \ldots, \mathtt{x}_m) \quad .$$

Letting $\overline{\mathtt{x}} = (\mathtt{x}_1, \ldots, \mathtt{x}_i, \ldots, \mathtt{x}_m)$ and $\overline{\mathtt{x}}' = (\mathtt{x}_1, \ldots, \mathtt{x}'_i, \ldots, \mathtt{x}_m)$, we say that the two i/o-pairs

$$\langle\overline{\mathtt{x}}, f(\overline{\mathtt{x}})\rangle \qquad \langle\overline{\mathtt{x}}', f(\overline{\mathtt{x}}')\rangle$$

is a *witness* of the dependency $(i, j)$ $\square$

We will also define what we mean by finding "all" dependencies in a given sample. Let us use $f_j$ to denote the function which computes the $j$th output parameter, i.e., $f_j(x_1, \ldots, x_m) = y_j$ where $y_1, \ldots, y_o = f(x_1, \ldots, x_m)$.

**Definition 2.2** A dependence relation $\mathcal{D}$ *covers* a sample if for any two i/o pairs $\langle \overline{\mathtt{x}}, f(\overline{\mathtt{x}}) \rangle$ and $\langle \overline{\mathtt{x}}', f(\overline{\mathtt{x}}') \rangle$ in the sample, and all $j$ such that $f_j(\overline{\mathtt{x}}) \neq f_j(\overline{\mathtt{x}}')$ there is an $i$ with $\mathtt{x}_i \neq \mathtt{x}'_i$ and $(i, j) \in \mathcal{D}$.

In other words, $\mathcal{D}$ covers a sample if any change in some output field $j$ can be attributed to a change in some input field on which $j$ may depend. Note that in general, there can be several dependency relations that cover a given sample (e.g., think of a sample containing only one i/o-pair).

**Problems to be Investigated** Given a sample, we will concretely investigate the following problems.

**Dependencies:** Find a relation $\mathcal{D}$ which covers a given sample.

**Copied Fields:** Find all pairs of indices $(i, j)$ such that output $y_j$ is directly copied from input $x_i$ in the sample.

**Constant Fields:** Find all indices $j$ such that output $z_j$ is constant (always has the same value, independently of input) in the sample.

Not considered here, but a goal for future work, is to be able to detect also some properties of the calculated function $f$. An example could be to find out that if the A-number has a certain prefix (e.g., 020), then the price is always 0 or the price depends only on whether the day is a Sunday.

# 3 Algorithms to Detect Dataflow Dependencies

In this section, we present a simple algorithm to detecting dataflow dependencies in a given sample.

Definition **??** suggests a simple algorithm for detecting dependencies from a sample, by comparing pairs of i/o-pairs with distance 1. However, typical samples contain very few such pairs, and we must therefore use pairs of i/o-pairs that differ in only "a few" fields. Assume that the distance between two inputs $\overline{\mathtt{x}} = \mathtt{x}_1, \ldots, \mathtt{x}_m$ and $\overline{\mathtt{x}}' = xval'_1, \ldots, \mathtt{x}'_m$ is $k$, and that $\overline{\mathtt{x}}$ and $\overline{\mathtt{x}}'$ differ in the values of $x_{i_1}, \ldots, x_{i_k}$. If the corresponding outputs $\overline{\mathtt{y}} = f(\overline{\mathtt{x}})$ and $\overline{\mathtt{y}}' = f(\overline{\mathtt{x}}')$ differ in (at least) their value of some parameter $y_j$, then we can infer that $(i, j) \in \mathcal{D}$ for at least one index $i$ in $i_1, \ldots, i_k$. We document this by a tuple of form $\langle (\overline{\mathtt{x}}, \overline{\mathtt{y}}), (\overline{\mathtt{x}}', \overline{\mathtt{y}}'), \{i_1, \ldots, i_k\}, j \rangle$, called a *witness constraint*, which contains the two i/o-samples, the indices where their input records differ, and an output index where the output records differ.

We can continue to inspect many i/o-pairs in a samples and generate many constraints of this form, but are not guaranteed to gain any precision. For this, we need a sample with many pairs of i/o-pairs of distance 1. Our approach will be to synthesize such pairs based on given i/o-pairs that are a "small" distance apart.

Assume, as above, that two "interesting" inputs $\overline{\mathtt{x}}$ and $\overline{\mathtt{x}}'$ differ in the values of $x_{i_1}, \ldots, x_{i_k}$, and that the corresponding outputs $\overline{\mathtt{y}} = f(\overline{\mathtt{x}})$ and $\overline{\mathtt{y}}' = f(\overline{\mathtt{x}}')$ differ in their value of some parameter $y_j$. We would like to know more precisely what caused the output to change from $f_j(\overline{\mathtt{x}})$ to $f_j(\overline{\mathtt{x}}')$. To this end, we generate a new sample which contains input records of distance one, which are certain to generate a witness for the change in the parameter $y_j$. We can do this in several ways.

- A very thorough strategy is to generate all $2^k$ different input vectors, whose value for input parameter $x_{i_l}$ have either $\mathtt{x}_{i_l}$ or $\mathtt{x}'_{i_l}$.
- A less thorough strategy is to generate $k + 1$ input records, each of which have the values $\mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_l}$ for the parameters $x_{i_1}, \ldots, x_{i_l}$ and the values $\mathtt{x}'_{i_{l+1}}, \ldots, \mathtt{x}'_{i_k}$ for the parameters $x_{i_{l+1}}, \ldots, x_{i_k}$.

By observing the output of the module in response to the synthesized input vectors, we are certain to find at least one witness that explains the change to output parameter $y_j$.

When testing synthesized input, a potential problem is that it can become invalid. We assume that system tests can tell us when an input vector is not "valid input". In this case, we do not know exactly how to proceed. What is clear is that the reasons for "invalid input" must be a combination of values of several fields (since we are combining fields of valid input vectors). We could try to record how many fields must be changed simultaneously in order to produce "valid input".

By synthesizing and executing new input vectors, we should be able to find a good approximation to the dependency relation $\mathcal{D}$ of $f$. We should document detected pairs in $\mathcal{D}$ by corresponding witnesses. In addition, it should be checked whether the detected dependency relation $\mathcal{D}$ indeed covers the sample consisting of all i/o-pairs that have been observed for the module. Here, the naive algorithm suggested by Definition **??** is often too costly, since it requires a quadratic (in the size of the sample) number of comparisons. A more efficient algorithm of (almost) linear complexity is to perform the following for each output parameter $y_j$.

1. Partition all i/o-pairs into blocks, where each block is characterized by its values of those input parameters $x_i$ such that $(i, j) \in \mathcal{D}$.

2. If the value of the output parameter $y_j$ in each block is constant, then $\mathcal{D}$ covers the sample. Otherwise a block with several different values of $y_j$ reveals that $\mathcal{D}$ must be extended. Missing pairs can be found by examining two i/o-pairs with different values of $y_j$ in some block, and use these to generate new input records that are executed and added to the sample.

# 4 Algorithms

In this section, we present more detailed descriptions of the algorithms that are used in the preceding section. We will use the following notation. We use $p, p'$ to denote i/o-pairs. For an i/o-pair $p = (\overline{\mathtt{x}}, \overline{\mathtt{y}})$, we use $p_I$ to denote the input record $\overline{\mathtt{x}}$ and $p_O$ to denote the output record $\overline{\mathtt{y}}$. The distance between two input records $\overline{\mathtt{x}}$ and $\overline{\mathtt{x}}'$ is denoted $d(\overline{\mathtt{x}}, \overline{\mathtt{x}}')$. A witness constraint is a four-tuple of form $\langle p, p', W, j \rangle$ where $W$ is the set of indices at which $p_I$ and $p'_I$ differ, and $j$ is an index at which $p_O$ and $p'_O$ differ.

**Pairwise comparison** The procedure GENERATE_WITNESS_CONSTRAINTS generates a set of witness constraints, given a sample $P$ and a maximal distance parameter maxdist The basic idea is to compare pairs of i/o-pairs which are at most maxdist apart, and collect the witness constraints that they generate in a set $T$ of witness constraints. Before storing a witness constraint in $T$, it is check whether it is already subsumed by an existing witness constraint, in the sense that the existing witness constraint gives more precise explanation of why the output index changes.

The procedure uses the function *Witnesses* which for two i/o-pairs $p$ and $p'$ generate all witness constraints of form $\langle p, p', W, j\rangle$, such that $W$ is the set of indices at which $p_I$ and $p'_I$ differ, and $j$ is an index at which $p_O$ and $p'_O$ differ.

**Algorithm 1** Computing Witness Constraints

```
procedure GENERATE_WITNESS_CONSTRAINTS(P : sample, maxdist : 𝒩)
  T ← ∅
  for all p, p' in P do
    if d(p_I, p'_I) ≤ maxdist then
      for all ⟨p, p', W, j⟩ in Witnesses(p, p') do
        if ∀⟨q, q', W', j'⟩ ∈ T . ¬(W' ⊈ W ∧ j = j') then
          T ← T ∪ ⟨p, p', W, j⟩
  return T
```

**Data-Generation process** The procedure SYNTHESIZE_DATA generates a new set of input records, given a set of witness constraints, which contains enough records which are distance 1 apart to guarantee that a witness will be found when feeding the input records to the module to get the corresponding output records.

**Algorithm 2** Synthesizing New Input Records

```
procedure SYNTHESIZE_INPUT(T : set of witness constraints)
  U ← ∅
  for all ⟨p, p', W, j⟩ in P do
    for l from 1 to size(W) − 1 do
      new x : array[1..m] of data
      for i from 1 to W[l] do x[i] ← p_I[i]
      for i from W[l] + 1 to m do x[i] ← p'_I[i]
      U ← U ∪ x
  return U
```

Given a set $T$ of witness constraints and a sample $P$, from each witness constraint $\langle p, p', W, j\rangle$ in $T$, where $W = \{i_1, \ldots, i_k\}$, the procedure generates $k-1$ input records, where the values for $x_{i_1}, \ldots, x_{i_l}$ are taken from $p_I$, and the values for $x_{i_{l+1}}, \ldots, x_{i_k}$ are taken from $p'_I$.

The dependencies results will be based on running the first algorithm with a maximal distance maxdist $= 1$. In that case, we may capture exactly the dependencies. Afterward, we may run a completeness test to check whether our results cover the changes in the initial data pair set or not.

**Completeness-Check** We present two algorithms for checking whether a certain dependency relation $D$ covers a given sample $P$.

---

**Algorithm 3** Completeness Check 1

---

**procedure** COMPLETENESS_CHECK_1($P$ : sample, $D$ : dependency relation)
  $\mathit{Not\_Covered} \leftarrow \emptyset$
  **for all** $p, p'$ **in** $P$ **do**
    **for all** $\langle p, p', W, j\rangle$ **in** $\mathit{Witnesses}(p, p')$ **do**
      **if** $\forall i \in W\ .\ (i, j) \notin D$ **then**
        $\mathit{Not\_Covered} \leftarrow \mathit{Not\_Covered} \cup \langle p, p', W, j\rangle$
  **return** $\mathit{Not\_Covered}$

---

The first closely follows Definition ??. It compares i/o-pairs in $P$ pair-wise and checks if their differences are explained by $D$.

---

**Algorithm 4** Completeness Check 2

---

**procedure** COMPLETENESS_CHECK_2($P$ : sample, $D$ : dependency relation)
  $\mathit{Not\_Covered} \leftarrow \emptyset$
  **for all** $j$ **in** $\{1, ..., o\}$ **do**
    **new** $H$ : hashtable
    $W \leftarrow \{i\ :\ (i, j) \in D\}$
    **for all** $p$ **in** $P$ **do**
      $H[\mathrm{Hash}(p_I, W)] \leftarrow H[\mathrm{Hash}(p_I, W)] \cup p_O[j]$
    **for all** indices $n$ of $H$ **do**
      **for all** $\mathrm{y}_j, \mathrm{y}'_j$ **in** $H[n]$ **do**
        **if** $\mathrm{y}_j \neq \mathrm{y}'_j$ **then return** *"not covered"*
  **return** *"covered"*

---

The second algorithm organizes all i/o-pairs into a hash table indexes by values of those input parameters $x_i$ such that $(i, j) \in \mathcal{D}$. If the value of the output parameter $y_j$ in each block is constant, then $\mathcal{D}$ covers the sample. Otherwise a block with several different values of $y_j$ reveals that $\mathcal{D}$ must be extended. Missing pairs can be found by examining two i/o-pairs with different values of $y_j$ in some block, and use these to generate new input records that are executed and added to the sample.

# 5 Experiments

We have implemented the techniques described in Sections **??** - **??**, and applied them to detect dependencies in a module of a billing system, which has been under test at Validation AB. The initial sample contained $10^5 - 10^6$ i/o pairs. By synthesizing new input records and executing them, we could extract a dependency relation which covers the sample. The results have been validated with system experts at Validation AB.

**Acknowledgments** The work has been supported by Validation AB and WM Data AB in the context of the ASTEC Competence center. Thanks to Kristin Andersson, Mats Ekman, Martin Eriksson, and Lena Nyberg for support and fruitful discussions during the project.