Relative Equivalence in the Presence of Ambiguity

Oshri Adler, Cindy Eisner, and Tatyana Veksler

IBM Research - Haifa, Israel
{oshria,eisner,tatyana}@il.ibm.com

Abstract. We examine the problem of defining equivalence between two functions (pieces of code) that are intended to perform analogous tasks, but whose interfaces do not correspond in a straightforward way, even to the point of ambiguity. We formalize the notion of what equivalence means in such a case and show how to check it using constraints on a model checking problem. We show that the presence of constraints complicates the issue of predicate abstraction, and show that nevertheless we can use predicates no finer than those needed in the absence of constraints. Our solution is being used to verify the migration of tens of millions of lines of health insurance claims processing code from ICD-9 to ICD-10, two versions of the International Statistical Classification of Diseases and Related Health Problems (ICD), whose correspondence is complex and ambiguous in both directions. We present experimental results on 90,000 real life functions.

1 Introduction

Given two functions f and f' that are over the same domain, or in the terminology of hardware and software, have the same interface, it is easy to define equivalence between them, and to see what it means to check that equivalence. Also, if there is a one-to-one mapping between the domains of f and f', then the definition of equivalence and what it means to check it is trivial. In the real world, however, things are often not so neat and simple. Sometimes we have two functions that are intended to perform analogous tasks, but whose interfaces do not correspond in a straightforward way, even to the point of ambiguity. Given such a setting, what does it mean for f and f' to be equivalent and how can it be checked?

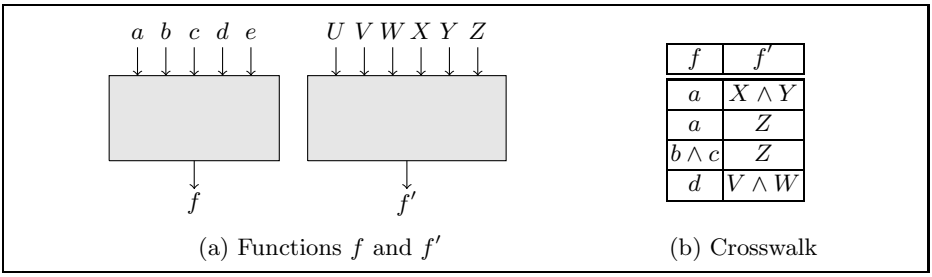
We encountered this interesting problem as part of an IBM engagement with NASCO®, an Atlanta, Georgia based company providing healthcare IT solutions to Blue Cross® and Blue Shield® (BCBS) Plans across the United States. The goal was to verify migration of insurance claims processing software, henceforth *benefit code*, from World Health Organization standard ICD-9 to ICD-10, two versions of the International Statistical Classification of Diseases and Related Health Problems (ICD) [1]. Correct migration of benefit code is of paramount importance to insurers, as benefit code directly affects the outflow of money.

Correspondence between ICD-9 and ICD-10 is given by a *schema crosswalk*, a table showing analogous elements of the interfaces. The correspondence is

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 430–446, 2013.

Table 1. Excerpt from an ICD-9/ICD-10 crosswalk

ICD-9		ICD-10	
041.81	Mycoplasma	$A49.3$	Mycoplasma infection, unspecified site
041.81	Mycoplasma	$B96.0$	Mycoplasma pneumoniae as the cause of diseases classified elsewhere
041.81 ∧ 466.0	Mycoplasma plus acute bronchitis	$J20.0$	Acute bronchitis due to mycoplasma pneumoniae
466.0	Acute bronchitis	$J20.9$	Acute bronchitis, unspecified



(a) Functions f and f'

f	f'
a	$X \wedge Y$
a	Z
$b \wedge c$	Z
d	$V \wedge W$

(b) Crosswalk

Fig. 1. A simple example

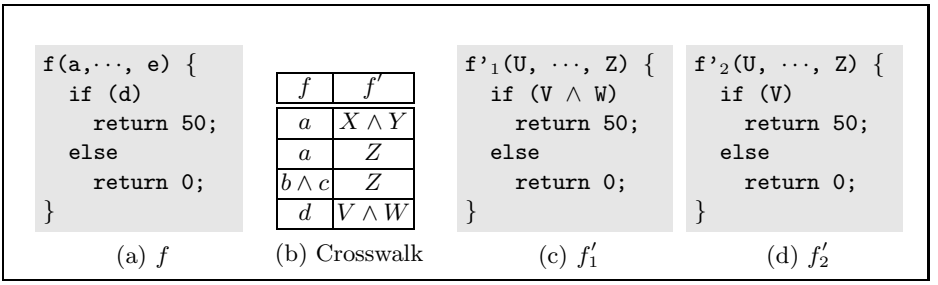
complex and ambiguous in both directions. For example, consider the excerpt shown in Table 1. ICD-9 diagnosis 041.81 corresponds to ICD-10 diagnosis $A49.3$ but also to $B96.0$, and if 041.81 appears in conjunction with 466.0, then together they correspond to diagnosis $J20.0$. But 466.0 by itself corresponds to $J20.9$.

Furthermore, not every ICD-9 code is expressible in ICD-10 and vice versa. For example, ICD-9 code $E927.0$ (Overexertion from sudden strenuous movement) has no comparable code in ICD-10, while ICD-10 code $T36.0X6$, (Underdosing of penicillins, initial encounter) has no comparable code in ICD-9.

The general problem is illustrated in Fig. 1: We are given two functions f and f' with completely different interfaces and a crosswalk describing how a single (Boolean) input of f is expressed in the interface of f' as the conjunction of one or more (Boolean) inputs of f', or vice versa. In this and all examples, we use lower case letters for inputs of f and upper case letters for inputs of f'.

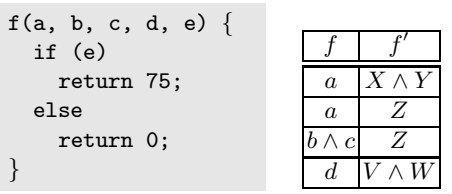
If an input is not expressible in the other interface, it is not mapped by the crosswalk. Input e of f and input U of f' are such inputs. Also, some inputs are expressible only in conjunction with others. For instance, $b \wedge c$ on the interface of f is expressible as Z in the interface of f', but there is no way to express b without c in the interface of f'. Finally, some inputs are expressible in more than one way – for instance, a on the interface of f and Z on the interface of f'.

Given such a crosswalk, our mission is to decide whether f and f' are equivalent relative to it. Intuitively, this means that they return the same value for analogous inputs. For example, for the f and f' shown in Fig. 1, we expect that f returns the same value on input d as an equivalent f' returns on input $V \wedge W$. But what about inputs not expressible in the other interface? Do we care what they return? We do. Such cases represent an exposure for the insurance company.



(a) f (b) Crosswalk (c) f'_1 (d) f'_2

Fig. 2. We want f to be equivalent to f'_1, but not to f'_2



(a) f (b) Crosswalk

Fig. 3. A fundamental mismatch

If a doctor diagnoses $V = 1$ and $W = 0$ in ICD-10, we can't know what she would have diagnosed in ICD-9. Explicitly paying a non-default amount for a claim not explicitly payable in the other interface introduces an element of uncertainty into the financial forecast. Thus we will require that equivalent functions treat such inputs in a default way – e.g., by falling into the "else" case.

Related Work. The term schema crosswalk is a term from database theory, and the question of relative equivalence can be asked about any pair of databases related by a crosswalk. However, we are unaware of related work; to the best of our knowledge the issues we explore here have not been explored in the context of databases, where the emphasis is usually on merging the contents of two databases rather than comparing code that interfaces with the databases as is.

2 Relative Equivalence

We now explore the issue of relative equivalence in more detail and define it precisely. We want to define that f and f' are equivalent if they each treat explicitly only inputs expressible in the other interface, and return the same value for analogous inputs. For example, we want to define that f and f'_1 of Fig. 2 are equivalent, and that f and f'_2 are not.

Consider now Fig. 3: f treats input e, not expressible in the other interface, in a non-default manner, thus we do not want it to be equivalent to any f'. We call such cases a *fundamental mismatch* between f and the crosswalk, because the inequivalence of f to f' is due to the crosswalk and not to the behavior of f'.

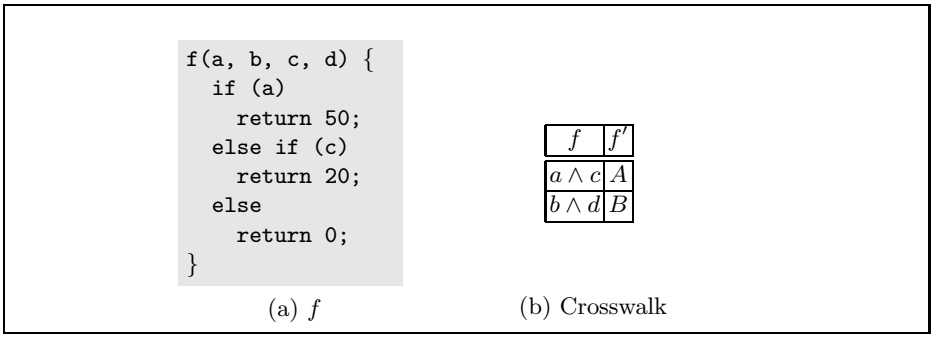


Fig. 4. Another fundamental mismatch

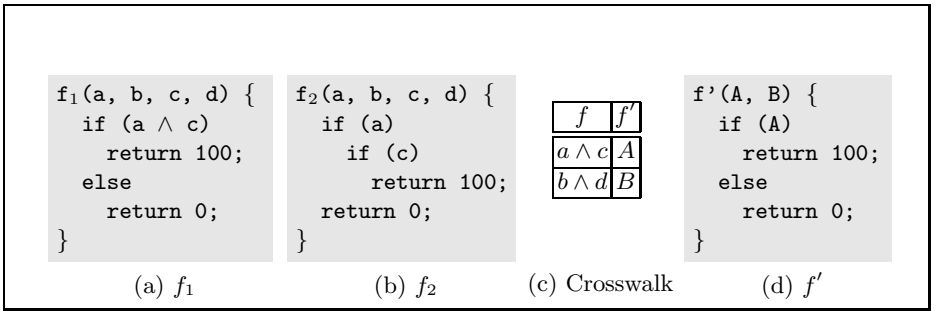


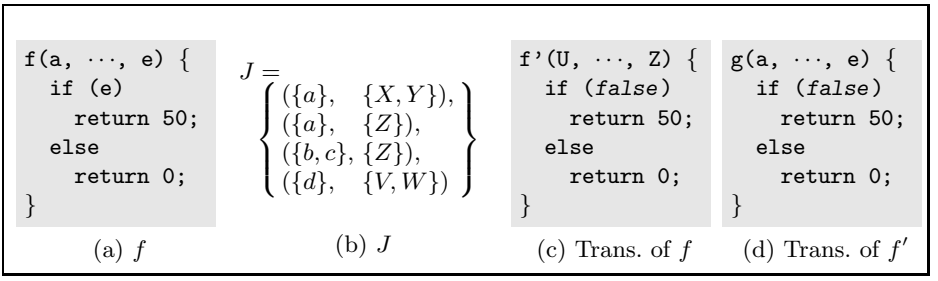
Fig. 5. f_1 and f_2 are equivalent, and we want both to be equivalent to f'

Figure 4 also shows a fundamental mismatch: f distinguishes between $a \wedge \neg c$, returning 50, and $\neg a \wedge c$, returning 20, but the interface of f' defined by the crosswalk cannot distinguish between them. It seems we should not allow the code to mention a without c or c without a, but this is going too far. The f_1 and f_2 of Fig. 5 are equivalent, so if f_1 is equivalent to f' (and intuitively, it is), then we want also that f_2 is equivalent to f'. Thus we must allow inputs not expressible in the other interface to stand alone, depending on the context.

That is, we want a semantic, not a syntactic, definition of equivalence, so we can identify equivalence between functions with differing control flows. Thus we will work over equivalence classes of input vectors to f and f', where an equivalence class is a set of input vectors for which the function returns the same value, and an input vector is a valuation of the individual inputs. For example, f_1 of Fig. 5 has two equivalence classes. One returns 100 and consists of vectors in which $a = c = 1$; the other returns 0, and consists of all other input vectors.

Definition 1 (Equivalence class). *An equivalence class of a function f is a maximal set of input vectors V such that for all $v_1, v_2 \in V$: $f(v_1) = f(v_2)$.*

We are now ready to define relative equivalence. Let P and P' be disjoint sets of atomic propositions. P and P' represent the inputs of f and f', respectively. The crosswalk is represented by the relation J, and we use $\varphi(v)$ and $\varphi(v')$ to move from elements of J back to conjunctions as used in the crosswalk.



(a) f (b) J (c) Trans. of f (d) Trans. of f'

Fig. 6. Translation of a fundamental mismatch

Definition 2 (Justified by the crosswalk (J)). *$J \subseteq 2^P \times 2^{P'}$ is a relation such that for every $(v, v') \in J$, either v or v' is a singleton.*

Definition 3 (Conjunctive formula ($\varphi(v), \varphi(v')$)). *Let $v \subseteq P$ and $v' \subseteq P'$. Then $\varphi(v)$ denotes the formula $\bigwedge_{p \in v} p$ and $\varphi(v')$ denotes the formula $\bigwedge_{p' \in v'} p'$.*

For example, the crosswalk of Fig. 5 is formalized as $J = \{(\{a, c\}, \{A\}), (\{b, d\}, \{B\})\}$ and we have that $\varphi(\{a, c\}) = a \wedge c$.

We now define translations between propositional formulas over P and P'.

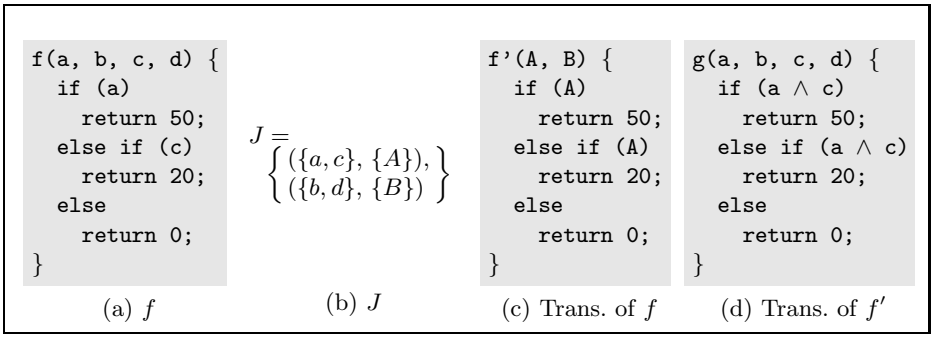
Definition 4 (Translation ($T(\psi), T'(\psi')$)). *Let Φ and Φ' be the set of proposition formulas over P and P', respectively. Let $p \in P$, $p' \in P'$ and $\psi, \psi_1, \psi_2 \in \Phi$, $\psi', \psi'_1, \psi'_2 \in \Phi'$. The functions $T : \Phi \mapsto \Phi'$ and $T' : \Phi' \mapsto \Phi$ are defined as follows:*

- $T(p) = \bigvee_{\{v' \mid (v,v') \in J \text{ and } p \in v\}} \varphi(v')$
- $T(\psi_1 \wedge \psi_2) = T(\psi_1) \wedge T(\psi_2)$
- $T(\psi_1 \vee \psi_2) = T(\psi_1) \vee T(\psi_2)$
- $T(\neg\psi) = \neg T(\psi)$

- $T'(p') = \bigvee_{\{v \mid (v,v') \in J \text{ and } p' \in v'\}} \varphi(v)$
- $T'(\psi'_1 \wedge \psi'_2) = T'(\psi'_1) \wedge T'(\psi'_2)$
- $T'(\psi'_1 \vee \psi'_2) = T'(\psi'_1) \vee T'(\psi'_2)$
- $T'(\neg\psi') = \neg T'(\psi')$

In the sequel, f and f' will be defined as *relatively equivalent* if the formulas characterizing their equivalence classes translate into each other. Before proceeding to the formal definition, let's take a look at how we can translate functions by translating the conditional expressions that form the basis of the equivalence classes, and what happens when we try to translate propositions not expressible in the other interface. Consider Fig. 5: $T(a) = T(c) = A$, and $T'(A) = a \wedge c$. Therefore the "if" statement of f_1 and both "if" statements of f_2 translate into "`if (A)`". Also, the "if" statement of f' translates into "`if(a ∧ c)`". Thus we can use T and T' to show that f_1 and f_2 are both equivalent to f'.

When we attempt to do the same with f's that are fundamental mismatches, we will get that translating forwards and back will not get us to where we started. Consider for example Fig. 6. We have that f translates into f', because there are no lines of the crosswalk containing e, so $T(e)$ is the empty disjunction *false*. Then, $\mathit{false} = p \wedge \neg p$ for some p, so we get that $T'(\mathit{false}) = \mathit{false}$ and translating f' back into the other interface gives us the g of Fig. 6d. In particular, f' does not



(a) f (b) J (c) Trans. of f (d) Trans. of f'

Fig. 7. Translation of another fundamental mismatch

translate back to f. The same thing will happen with the fundamental mismatch we saw in Fig. 4 – its translation forward and back is shown in Fig. 7.

Formally, an input vector of a function is a *valuation* v of a set Q (e.g., P or P'), associating each proposition with a value **T** or **F**. Note that equivalently, a valuation v can be associated with the subset $\{q \in Q \mid q \text{ has the value } \mathbf{T} \text{ in } v\}$. We abuse notation and use valuations of Q and subsets of Q (elements of 2^Q) interchangeably. For a valuation v and a propositional formula ψ, we use $v \models \psi$ to denote that ψ holds on valuation v and we use $[\psi]$ to denote the set of valuations on which ψ holds. We use $\psi_1 \equiv \psi_2$ to denote that $[\psi_1] = [\psi_2]$.

Recall that an equivalence class is just a set of valuations and thus can be denoted by $[\psi]$ for some formula ψ. For example, the equivalence classes of f of Fig. 7a are $[a]$, $[\neg a \wedge c]$ and $[\neg a \wedge \neg c]$. For an equivalence class $[\psi_i]$ of a function f, let $f([\psi_i])$ denote the value returned by f for every element of $[\psi_i]$. For simplicity we assume that there is a total order on the values returned by f and f'; if there is not, choose an order arbitrarily. Thus we can assume wlog that equivalence classes are ordered such that $f([\psi_i]) < f([\psi_{i+1}])$. We now define relative equivalence based on T and T' of Definition 4 as follows:

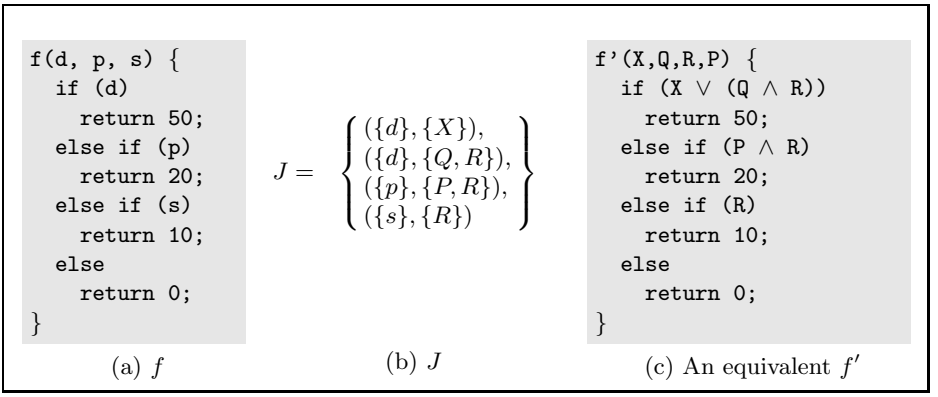
Definition 5 (Relative equivalence). *Let $f : 2^P \mapsto \mathcal{R}$ and $f' : 2^{P'} \mapsto \mathcal{R}$ for some range $\mathcal{R}$, and let $[\psi_i]$ for $i \leq n$ be the equivalence classes of f and $[\psi'_j]$ for $j \leq n'$ be the equivalence classes of f'. Then f is equivalent to f' relative to J, denoted $f \sim f'$, if:*

$$n = n' \text{ and for every } i \leq n : f([\psi_i]) = f'([\psi'_i]) \text{ and } \psi_i \equiv T'(\psi'_i) \text{ and } \psi'_i \equiv T(\psi_i)$$

The definition of T and T', and thus of relative equivalence, is based on syntax, but these are semantic notions, as stated by the following proposition.

Proposition 1. *Let $\psi_1, \psi_2 \in \Phi$ such that $\psi_1 \equiv \psi_2$ and $\psi'_1, \psi'_2 \in \Phi'$ such that $\psi'_1 \equiv \psi'_2$. Then $T(\psi_1) \equiv T(\psi_2)$ and $T'(\psi'_1) \equiv T'(\psi'_2)$.*

Consider now Fig. 8. The equivalence classes of f are given by $\psi_4 = d$, $\psi_3 = \neg d \wedge p$, $\psi_2 = \neg d \wedge \neg p \wedge s$ and $\psi_1 = \neg d \wedge \neg p \wedge \neg s$ (recall that we order equivalence classes according to their return value) and those of f' are given by $\psi'_4 = X \vee (Q \wedge R)$,



(a) f (b) J (c) An equivalent f'

Fig. 8. An equivalent f and f', this time formally

$\psi'_3 = \neg(X \vee (Q \wedge R)) \wedge (P \wedge R) \equiv \neg X \wedge \neg Q \wedge P \wedge R$, $\psi'_2 = \neg(X \vee (Q \wedge R)) \wedge \neg(P \wedge R) \wedge R \equiv \neg X \wedge \neg Q \wedge \neg P \wedge R$ and $\psi'_1 = \neg(X \vee (Q \wedge R)) \wedge \neg(P \wedge R) \wedge \neg R \equiv \neg X \wedge \neg R$. Also, from J we have that $T(d) = X \vee (Q \wedge R)$, $T(p) = P \wedge R$, $T(s) = R$, $T'(X) = d$, $T'(Q) = d$, $T'(P) = p$ and $T'(R) = d \vee p \vee s$. Substituting gives $T(\psi_i) = \psi'_i$ and $T'(\psi'_i) = \psi_i$ for $1 \leq i \leq 4$, thus $f \sim f'$.

Now consider functions f and f' that switch the first and third conditions in Figs. 8a and 8c. Doing so would change the equivalence classes. In particular we would have that $\psi_4 = s$ and $\psi'_4 = R$, but $T'(R) = d \vee p \vee s \not\equiv s$, thus such an f and f' would not be relatively equivalent.

3 Checking Relative Equivalence

Having defined relative equivalence, how can it be checked? One way would be to calculate equivalence classes and check whether they are in the correct relation. However, real life benefit code functions calculate tens of outputs, so doing so would require multiple semantic analyses. Thus we prefer a way that avoids calculating equivalence classes and instead uses a single run of a model checker. We are looking for a relation $R \subseteq 2^P \times 2^{P'}$, such that $f \sim f'$ can be checked by checking whether $f(v) = f'(v')$ for every $(v, v') \in R$.

In the sequel, we represent an input vector by a set, where the elements of the set are the inputs that have the value 1. For example, for the f of Fig. 8a, $\{d, s\}$ represents the input vector in which $d = s = 1$ and $p = 0$. Now, it seems that checking equivalence of f and f' relative to a crosswalk should entail checking that f and f' return the same value for corresponding input vectors, where by correspond we mean that they are analogous according to the crosswalk. For example, consider the J of Fig. 9. Input vector $\{a\}$ of f corresponds to input vector $\{X, Y\}$ of f', input vector $\{b\}$ of f corresponds to input vector $\{Z\}$ of f', and input vector $\{a, b\}$ of f corresponds to input vector $\{X, Y, Z\}$ of f'.

Is it sufficient to check all pairs of corresponding input vectors? No. Doing so will result in a verdict of "equivalent" for the f and f'_1 shown in Fig. 9, even

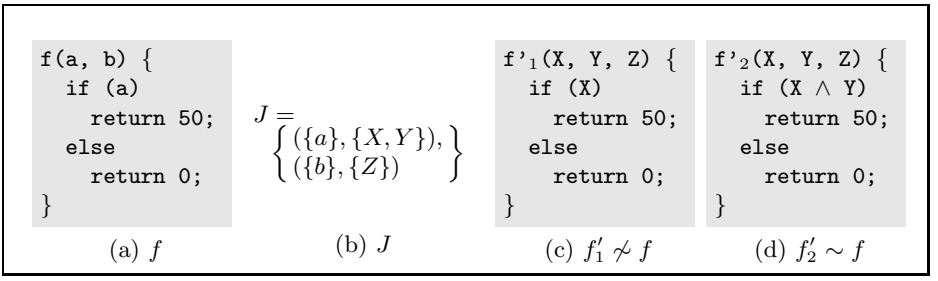


Fig. 9. Checking all pairs of corresponding input vectors is insufficient

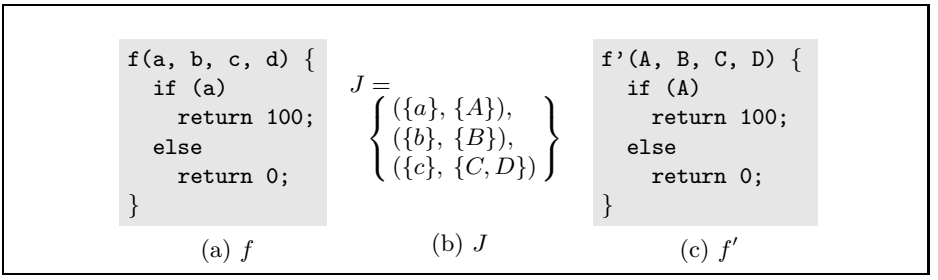


Fig. 10. Input vector $\{A, B, C\}$ of f behaves in a non-default way

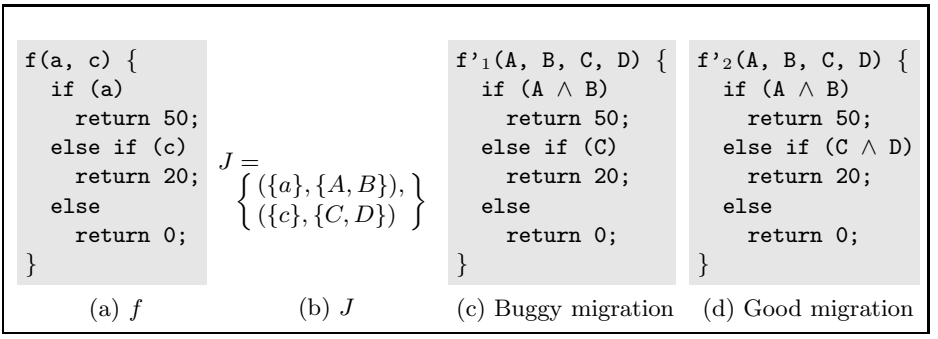


Fig. 11. A good and a buggy migration

though f is equivalent to f'_2 but not to f'_1. The only input vectors that can distinguish between f'_1 and f'_2 are ones in which X holds but Y does not, but such vectors have no corresponding vector in the other interface. It seems we can deal with this by requiring a "default" behavior from such input vectors, where by default we mean that the calculation falls into some "else" case. However, consider the example of Fig. 10. Input vector $\{A, B, C\}$ of f' has no equivalent in f, yet behaves in a non-default way. It makes no sense to require that $\{A, B, C\}$ falls into the "else" case, since f and f' are clearly equivalent as written.

So how should we pair a vector with no corresponding vector in the other interface? Consider Fig. 11, in which f has been migrated twice to a new interface.

Input vector $\{C\}$ has no corresponding input vector according to J, because C does not "stand alone" in any element of J. Thus we expect that in a correct migration, C cannot influence the returned value without D, so we expect that a correct migration f' returns the same value for $\{C\}$ that it returns on $\emptyset$, and in general that for any input vector v' of f' such that $C \in v'$ but $D \notin v'$, we have that $f'(v') = f'(v' \setminus \{C\})$. We call C in vectors $\{C\}$ and $\{A, B, C\}$ an *orphan* input, because the vector does not contain enough other inputs to complete a line in the crosswalk. The *remainder* of an input vector consists of all its orphans:

Definition 6 (Remainder ($r(v)$, $r'(v')$)). *$r : 2^P \mapsto 2^P$ and $r' : 2^{P'} \mapsto 2^{P'}$ are defined as follows:*

$$\begin{aligned} r(v) &= \{p \in v \mid \nexists w \subseteq v,\ w' \subseteq P' \ \ s.t.\ (\{p\} \cup w,\ w') \in J\} \\ r'(v') &= \{p' \in v' \mid \nexists w \subseteq P,\ w' \subseteq v' \ \ s.t.\ (w, \{p'\} \cup w') \in J\} \end{aligned}$$

For example, using the J of Fig. 11, we have that $r'(\{A, B, C\}) = \{C\}$.

Using the notion of remainder, we define the relation R as follows:

Definition 7 (Relevant input pairs (R)). *The set of relevant input pairs is given by $R \subseteq 2^P \times 2^{P'}$ defined as follows:*

$$\begin{aligned} R = \{(v, v') \mid\ & v \in 2^P \ \ \textit{and}\ v' \in 2^{P'} \ \ \textit{and}\ \exists v_1, v_2, \cdots v_k, v'_1, v'_2, \cdots v'_k \ \ s.t. \\ & v = v_1 \cup v_2 \cup \cdots \cup v_k \ \ \textit{and}\ v' = v'_1 \cup v'_2 \cup \cdots \cup v'_k \ \ \textit{and} \\ & v_k = r(v) \ \ \textit{and}\ v'_k = r'(v') \ \ \textit{and}\ \forall i < k : (v_i, v'_i) \in J\} \end{aligned}$$

For example, using the J of Fig. 11, we have that $(\{a, c\}, \{A, B, C, D\}) \in R$ because $\{a, c\} = v_1 \cup v_2 \cup v_3$ and $\{A, B, C, D\} = v'_1 \cup v'_2 \cup v'_3$ for $v_1 = \{a\}$, $v_2 = \{c\}$, $v_3 = \emptyset$, $v'_1 = \{A, B\}$, $v'_2 = \{C, D\}$ and $v'_3 = \emptyset$, and we have that $(v_1, v'_1), (v_2, v'_2) \in J$, $v_3 = r(\{a, c\})$, and $v'_3 = r'(\{A, B, C, D\})$. Also, $(\emptyset, \{C\})$, a vector pair that finds the migration bug, is in R, because $\emptyset = v_1$ and $\{C\} = v'_1$ for $v_1 = \emptyset$ and $v'_1 = \{C\}$ and $v_1 = r(\emptyset)$ and $v'_1 = r'(\{C\})$.

R includes every pair of corresponding vectors (and then the remainders are empty) and also pairs vectors without a corresponding vector in a way that expects that the remainders don't influence the function. Intuitively, checking that f and f' return the same value for every pair in R should be a way to show that f and f' are relatively equivalent. The following theorem confirms this.

Theorem 1 (Checking relative equivalence)

$$f \sim f' \ \ \textit{iff}\ \forall (v, v') \in R : f(v) = f'(v')$$

The proof of the $\Longrightarrow$ direction is based on the following lemma.

Lemma 1 (Relating R and T, T'). *Let $(v, v') \in R$ and let $\pi \in \Phi$ and $\pi' \in \Phi'$ such that $\pi \equiv T'(\pi')$ and $\pi' \equiv T(\pi)$. Then $v \models \pi \iff v' \models \pi'$.*

To prove the $\Longleftarrow$ direction, we observe that every equivalence class containing $v \in 2^P$ must contain as well all valuations in 2^P that are transitively related to v by R, and similarly for $v' \in 2^{P'}$. We define:

Definition 8 (Expected same-behavior valuations of (v, v') ($E(v, v')$). *Let $(v, v') \in R$. Then $E(v, v')$ is the subset of $2^P \cup 2^{P'}$ defined inductively as follows:*

- $v, v' \in E(v, v')$.
- *If $w \in E(v, v')$ and $(w, w') \in R$, then $w' \in E(v, v')$*
- *If $w' \in E(v, v')$ and $(w, w') \in R$, then $w \in E(v, v')$*

The challenge is to show that each $E(v, v')$ can be represented as $[\pi] \cup [\pi']$ such that $\pi \equiv T'(\pi')$ and $\pi' \equiv T(\pi)$, for some π, π', as stated by the following lemma:

Lemma 2 (Expressing $E(v, v')$). *Let $(v, v') \in R$. Then $\exists \pi, \pi'$ such that $\pi \equiv T'(\pi')$ and $\pi' \equiv T(\pi)$ and $E(v, v') = [\pi] \cup [\pi']$.*

The proof of Lemma 2 is based on showing that $E(v, v')$ "has no holes", that is, that if u and z are in $E(v, v')$, then every w such that $u \subseteq w \subseteq z$ is also in $E(v, v')$. This allows us to define a partial order on the various $E(v, v')$, and we form the necessary formulas starting from the single maximal element in the partial order, $E(P, P')$. Taking the disjunction of $\varphi(u)$ for all minimal elements $u \in 2^P$ in $E(v, v')$ gives us a formula characterizing every valuation $x \in 2^P$ that is in $E(v, v')$ or in some $E(w, w') \succ E(v, v')$. From there it is a simple matter to remove the valuations that are too big, by conjuncting with the negation of the formulas formed previously for larger $E(w, w')$'s.

4 Model Checking Setup

We have a theory of relative equivalence and a set of input pairs sufficient to check it. Using these, we set up our model checking problem as follows. We compile a pair of benefit code functions into an SMV [7] model constructed in a straightforward manner, similar to the method described in [5,6]. A dedicated state variable keeps track of the control flow, whose behavior may depend on the value of other state variables. Each input is allocated a state variable, which wakes up in a non-deterministic state and keeps its value throughout the run. Each variable of the original code is also allocated a state variable, and is assigned a value when the control flow reaches a relevant line.

The actual benefit code language is proprietary. While it does not contain loops, it does contain some constructs that are quite tricky to model. Due to space constraints, the details are beyond the scope of this paper. In Fig. 12a we show an example that uses pseudo-code based on the syntax of C, similarly to previous examples. The f and f' shown in Fig. 12a might compile to the model M_c shown in Fig. 12b. The behaviors of variables line1 and line2 represent the control flow of f and f', respectively, and the behaviors of variables pay1 and pay2 represent the behavior of variables pay in f and f', respectively.

It remains to constrain the inputs to pairs in R and to check that `pay1 = pay2` whenever both computations have ended (`line1 = 4` and `line2 = 4`).

For brevity, declarations and initializations are not shown in the SMV *model.*

```
  f(a, b) {
1   if (a ∧ b)
2     pay = 50;
    else
3     pay = 0;
4 }
```

```
  f'(V, W) {
1   if (V ∨ W)
2     pay = 50;
    else
3     pay = 0;
4 }
```

(a) f and f'

```
assign next(line1) := case
   line1=1:  if (a ∧ b) then 2 else 3 endif;
   else   :  4;
esac;

assign next(line2) := case
   line2=1:  if (V ∨ W) then 2 else 3 endif;
   else   :  4;
esac;

assign next(pay1) := case
   line1 = 2:  50;
   line1 = 3:  0;
   else:       pay1;
esac;

assign next(pay2) := case
   line2 = 2:  50;
   line2 = 3:  0;
   else:       pay2;
esac;
```

(b) The (concrete) SMV model M_c

Fig. 12. Compiling f and f' to an SMV model

For $p \in P$, let $J^p = \{(u, u') \in J \mid p \in u\}$ and similarly for $p' \in P'$ let $J^{p'} = \{(u, u') \in J \mid p' \in u'\}$. We constrain each element $p \in P$ and $p' \in P'$ as follows:

$$p \leftrightarrow \Big(\Big(\bigvee_{(u,u')\in J^p} (\varphi(u) \wedge \varphi(u'))\Big) \bigvee \Big(p \wedge \bigwedge_{(u,u')\in J^p} \neg\varphi(u)\Big)\Big) \tag{1}$$

$$p' \leftrightarrow \Big(\Big(\bigvee_{(u,u')\in J^{p'}} (\varphi(u) \wedge \varphi(u'))\Big) \bigvee \Big(p' \wedge \bigwedge_{(u,u')\in J^{p'}} \neg\varphi(u')\Big)\Big) \tag{2}$$

Definition 9 (Concrete constraint (C_c)). *Let C_c be the conjunction of Equations (1) and (2) for each $p \in P$ and $p' \in P'$.*

Constraining our model checking problem using C_c allows us to check relative equivalence, as stated by the following theorem.

Theorem 2 (Using the concrete constraint)

$$f \sim f' \text{ iff } \forall (v, v') \text{ s.t. } (v \cup v') \models C_c : f(v) = f'(v')$$

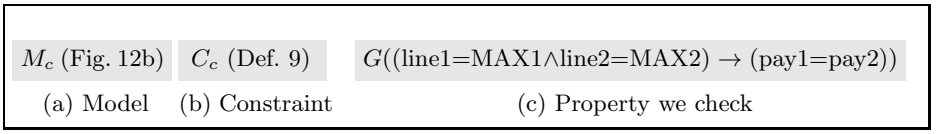


Fig. 13. The complete (concrete) model checking problem

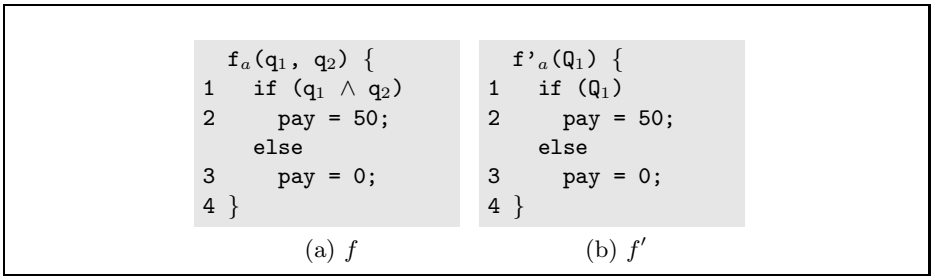


Fig. 14. Abstract versions of the f and f' shown in Fig. 12

For example, let $J = \{(\{a\}, \{A\}), (\{b\}, \{B, C\})\}$, then our constraint would be:

$$\begin{array}{llll} (a & \leftrightarrow ((a \wedge A) & \vee (a \wedge \neg a))) & \wedge \\ (b & \leftrightarrow ((b \wedge B \wedge C) & \vee (b \wedge \neg b))) & \wedge \\ (A & \leftrightarrow ((a \wedge A) & \vee (A \wedge \neg A))) & \wedge \\ (B & \leftrightarrow ((b \wedge B \wedge C) & \vee (B \wedge \neg(B \wedge C)))) & \wedge \\ (C & \leftrightarrow ((b \wedge B \wedge C) & \vee (C \wedge \neg(B \wedge C)))) & \end{array} \tag{3}$$

giving

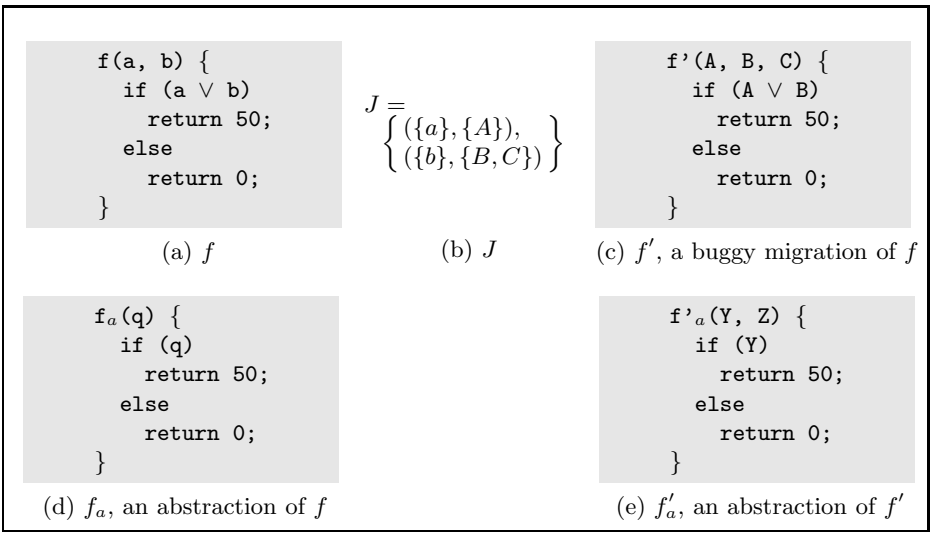
$$R = \left\{ \begin{array}{l} (\emptyset, \emptyset), (\emptyset, \{B\}), (\emptyset, \{C\}), (\{a\}, \{A\}), (\{a\}, \{A, B\}), \\ (\{a\}, \{A, C\}), (\{b\}, \{B, C\}), (\{a, b\}, \{A, B, C\}) \end{array} \right\} \tag{4}$$

Thus our complete (concrete) model checking problem, shown in Fig. 13, consists of the model M_c, the constraint C_c, and the property shown in Fig. 13c.

4.1 Complications Arising from Predicate Abstraction

A single line of benefit code can access files called *tables* representing large disjunctions of inputs, often consisting of hundreds of disjuncts each, so checking even a small function might involve thousands of state variables. Thus to avoid the size problem we use abstract versions of f and f', built by using predicates to represent disjunctions. We use the coarsest such abstraction that does not lose precision, thus our abstractions are *exact* in the sense of [3,4]. For example, the abstractions of the f and f' of Fig. 12 are shown in Fig. 14. Recall that we allocate predicates to disjunctions but not to conjunctions, thus f_a uses two predicates while f'_a uses only one. We define:

Definition 10 (Represents). *Let S be a set of atomic propositions, let $\{S_1, S_2, \cdots, S_\ell\}$ be a partition of S and for every S_i, let q_i abstract $\bigvee_{p \in S_i} p$. Then q_i represents s if $s \in S_i$.*



(a) f (b) J (c) f', a buggy migration of f

(d) f_a, an abstraction of f (e) f'_a, an abstraction of f'

Fig. 15. A buggy migration and its predicate abstraction

It is easy to see that if we partition P and P' carefully, we can get that the model M_a built from f_a and f'_a is bisimulation equivalent to the model M_c built from f and f'. However, taking our constraints into consideration, the abstraction seems to break down. For example, let J_a be obtained from J by replacing every element $p \in P \cup P'$ with the predicate q that represents it, and then constrain every $q \in A$ using a version of Equation (1) that uses J_a instead of J, and similarly for every $q' \in A'$. The constraints may make finer distinctions than those made by our predicates, thus using this abstraction we miss bugs.

For example, consider the buggy migration and its predicate abstraction shown in Fig. 15. Using predicates q abstracting $a \vee b$, Y abstracting $A \vee B$ and Z abstracting C, we get $J_a = \{(\{q\}, \{Y\}), (\{q\}, \{Y, Z\})\}$, which gives:

$$\begin{array}{l} (q \leftrightarrow ((q \wedge Y) \vee (q \wedge Y \wedge Z) \vee (q \wedge \neg q))) \qquad\qquad\qquad\quad \wedge \\ (Y \leftrightarrow ((q \wedge Y) \vee (q \wedge Y \wedge Z) \vee (Y \wedge \neg Y \wedge \neg(Y \wedge Z)))) \wedge \\ (Z \leftrightarrow ((q \wedge Y \wedge Z) \qquad\qquad\quad \vee (Z \wedge \neg(Y \wedge Z)))) \end{array} \tag{5}$$

representing the following (bad) abstract R, call it R_b:

$$R_b = \{(\emptyset, \emptyset), (\emptyset, \{Z\}), (\{q\}, \{Y\}), (\{q\}, \{Y, Z\})\} \tag{6}$$

R_b is bad because f_a and f'_a return the same value for every input pair in R_b, thus we have missed the migration bug.

One solution would be to build finer predicates that take the concrete constraints into consideration, but given how our constraints are built, with every atomic proposition on one side of an $\leftrightarrow$, that would seem to leave us with no abstraction at all. Happily, we can avoid building finer predicates than the coarsest

required for bisimulation – this makes intuitive sense, since if f cannot distinguish between input vector v_1 and v_2, there cannot be any reason to check both. What we want is to check every pair in R_a, obtained from R by replacing every $p \in P$ and $p' \in P'$ with the predicate that represents it. All that remains is to constrain the abstract inputs to pairs in R_a.

We define our abstract constraint C_a as follows:

Definition 11 (Abstract constraint (C_a)). *Let $\{P_1, P_2, \cdots, P_\ell\}$ be a partition of P represented by predicates $\{q_1, q_2, \cdots q_\ell\}$ and let $\{P'_1, P'_2, \cdots P'_{\ell'}\}$ be a partition of P' represented by predicates $\{q'_1, q'_2, \cdots q'_{\ell'}\}$.*

$$C_a = \exists_{p \in P} \exists_{p' \in P'} \left(\bigwedge_{i=1}^{\ell} \Big(q_i \leftrightarrow \bigvee_{p \in P_i} p \Big) \ \wedge \ \bigwedge_{j=1}^{\ell'} \Big(q'_j \leftrightarrow \bigvee_{p' \in P'_j} p' \Big) \ \wedge \ C_c \right)$$

Using C_a, built at compile time, allows us to check that f and f' are relatively equivalent by comparing f_a and f'_a, as stated by the following theorem.

Theorem 3 (Using the abstract constraint). *Let f_a be an abstraction of f and let f'_a be an abstraction of f'. Then*

$$f \sim f' \text{ iff } \forall (x, x') \text{ s.t. } (x \cup x') \models C_a : f_a(x) = f'_a(x')$$

For example, using our predicates q abstracting $a \vee b$, Y abstracting $A \vee B$ and Z abstracting C, we get the following abstraction of the R from Equation (4):

$$R_a = \{(\emptyset, \emptyset), (\emptyset, \{Y\}), (\emptyset, \{Z\}), (\{q\}, \{Y\}), (\{q\}, \{Y, Z\})\} \tag{7}$$

Then the migration bug of Fig. 15 will be found by the abstract pair $(\emptyset, \{Y\})$, representing the concrete pair $(\emptyset, \{B\})$.

Our complete abstract model checking problem, then, consists of the abstract model M_a, the abstract constraint C_a, and the property shown in Fig. 13c.

5 Experimental Results

We implemented our method in a tool to formally verify migration of benefit code from ICD-9 to ICD-10 as part of an IBM engagement with NASCO. The result is being used to verify migration of tens of millions of lines of benefit code, consisting of millions of relatively small functions. As part of our testing process we gathered statistics on a subset of the real code, consisting of some 90,000 functions, comparing them to an ad hoc migration developed specially for testing purposes. We used GEM files published by the Centers for Medicare and Medicaid Services [2], in which J consists of approximately 175,000 pairs, mapping some 17,000 ICD-9 diagnosis and procedure codes to some 141,000 ICD-10 codes. In this section we present some practical details regarding the implementation, and information on the performance of our tool on its realistic test base.

As we have seen, some functions are fundamental mismatches with respect to J, thus not migratable, and so correct-by-construction migration is not possible

Table 2. Size, compile and run time (in seconds)

Plan	# f's	Lines of Code				Compile Time				Run Time			
		Total	Ave	Med	Max	Total	Ave	Med	Max	Total	Ave	Med	Max
A	149	4241	28.5	22	160	56	0.4	0.1	9.9	86	0.6	0.5	11.3
B	13,673	344,184	25.2	21	632	21,521	1.6	0.2	11.4	158,197	11.6	6.0	445.3
C	20,395	538,281	26.4	26	632	38,115	1.9	0.2	12.4	128,421	6.3	5.5	459.2
D	4,727	135,746	28.7	23	428	1,966	0.4	0.5	12.2	2,187	0.5	0.5	509.5
E	49,523	395,781	8.0	7	232	7,497	0.2	0.1	11.6	44,825	0.9	0.6	999.0
F	1,607	77,347	48.1	32	330	3,655	2.3	0.3	11.2	156,049	97.1	1.5	965.3

Table 3. Comparison results

Plan	Count			%		
	Migration Correct	Fundamental Mismatch	Other Mismatch	Migration Correct	Fundamental Mismatch	Other Mismatch
A	138	11	0	92.6	7.4	0.0
B	13,673	0	0	100.0	0.0	0.0
C	13,141	7,200	54	64.4	35.3	0.3
D	4,584	139	4	97.0	2.9	0.1
E	49,248	169	106	99.4	0.3	0.2
F	1,400	175	32	87.1	10.9	2.0

without a check for migratability. Even for migratable functions, the process is not so simple: Not every expression is expressible in the proprietary language used in the benefit code, so some expressions need to be split into sequential or nested conditional statements in such a way that the migrated function f' might be syntactically far from the original function f; in particular, the control flow of f and f' may be quite different. For these and other reasons, the migration process consists of two steps. First a heuristic migration is performed that produces correct results in most but not all cases, then a verification step checks if the migration is correct. If it is not, a counterexample is produced and the migration is fixed manually and re-verified.

Our experiment consisted of applying our method to 90,000 real benefit code functions belonging to six different insurance plans, comparing them to an ad hoc migration developed specially for testing purposes. Our industrial strength model checker RuleBase PE [8], designed for very large model checking problems, incurs unnecessary overhead when applied to small problems. Instead, we run directly on Discovery, RuleBase PE's BDD-based model checking engine. Table 2 shows, for each plan, the total, average, median and maximum lines of code per function and compile and run time in seconds on a 2×2.4 GHz Intel Xeon processor with 2 GB RAM running Red Hat 5.6. For testing purposes, we timed out at 1,000 seconds run time. Out of just over 90,000 test cases, 140 timed out and are not included in the numbers shown in Table 2.

Table 3 shows the model checking results. In most cases, our ad hoc migration was correct. Some cases were identified as suspected fundamental

mismatches at compile time and those that were confirmed at run time are listed in the column labeled "Fundamental Mismatch". We find suspected fundamental mismatches at compile time when we discover during predicate abstraction that some atomic proposition should be used by f but is not. For example, if $(\{a, b\}, \{A\}) \in J$ and there is no other $(v, v') \in J$ such that $a \in v$, then an f that uses a but not b cannot be correctly migrated unless the use of a is in dead code. Run time distinguishes between real and spurious suspected fundamental mismatches. Note that the 7,200 fundamental mismatches found for Plan C most likely result from a smaller number of errors in some table used by multiple functions.

The other mismatches shown in Table 3 represent either fundamental mismatches not identified at compile time (e.g., in the above example, if both a and b are used in the code but in the wrong context) or bugs in our ad hoc migration.

6 Conclusion

We have formalized the notion of relative equivalence and characterized the set of cases R sufficient to check it. We have shown how predicate abstraction interacts with constraint generation and presented a solution that avoids overrefinement. We have implemented our solution in a tool currently being used to migrate tens of millions of lines of insurance claims processing code from from ICD-9 to ICD-10, two versions of the International Statistical Classification of Diseases and Related Health Problems. We have presented experimental results for the migration of 90,000 real functions from this code, using a crosswalk consisting of approximately 175,000 subset pairs, that maps 17,000 ICD-9 codes to 141,000 ICD-10 codes and is ambiguous in both directions. Future work is to explore the applicativity of our work beyond ICD, for instance in the context of databases.

Acknowledgements. Thank you to Gadi Aleksandrowicz, Elena Guralnik, Alexander Ivrii, Shiri Moran, Ziv Nevo, Avigail Orni, Julia Rubin, Karen Yorav and anonymous reviewers for important comments on early versions of this work.

References

1. Centers for Disease Control and Prevention, `http://www.cdc.gov/nchs/icd.htm`
2. Centers for Medicare and Medicaid Services, `https://www.cms.gov/Medicare/Coding/ICD10/index.html`
3. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. ACM Transactions on Programming Languages and Systems 16(5), 1512–1542 (1994)

4. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2001)
5. Eisner, C.: Model Checking the Garbage Collection Mechanism of SMV. Electronic Notes in Theoretical Computer Science 55(3), 289–303 (2001)
6. Eisner, C.: Formal Verification of Software Source Code through Semi-automatic Modeling. Software and System Modeling 4(1), 14–31 (2005)
7. McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
8. RuleBase Parallel Edition, `https://www.research.ibm.com/haifa/projects/verification/RB_Homepage/`