Reconstruction of Class Hierarchies for Decompilation of C++ Programs

A. Fokin
Moscow State University
Computational Math. and Cybernetics Dept.
Leninskie Gory, Moscow, Russia
apfokin@gmail.com

K. Troshina
Institute for System Programming
Russian Academy of Sciences
25, Alexander Solzhenitsyn st., Moscow, Russia
katerina@ispras.ru

A. Chernov
Moscow State University
Computational Math. and Cybernetics Dept.
Leninskie Gory, Moscow, Russia
cher@unicorn.cmc.msu.ru

Abstract

This paper presents a method for automatic reconstruction of polymorphic class hierarchies from assembly code obtained by compiling a C++ program. If the program was compiled with run-time type information (RTTI), class hierarchy is reconstructed via analysis of RTTI structures.

In case RTTI structures are missing in the assembly, a technique based on analysis of virtual tables and virtual destructors is used. An inheritance relation on a set of classes induces an inheritance relation on a set of virtual tables. If virtual inheritance is not used, then induced relation on a set of virtual tables is a single inheritance relation. The presented technique is used for reconstruction of this single inheritance relation. Information on virtual function tables belonging to classes that use multiple inheritance is also gathered during reconstruction.

1 Introduction

Decompilation from low-level machine languages to high-level languages (especially C) has gained fair amount of attention recently. Decompilers have been improving in quality and range of accepted low-level programs. The C programming languages was chosen as the target language for decompilers essentially because it is simple enough yet powerful and widely used. However, a part of programs subject for decompilation were actually written in C++ or C/C++ mix. Decompilation of C++ programs into C results in undesirable artefacts like non-decompiled assembly fragments in place of C++ exception handling operators, or mess of C types instead of C++ inheritance hierarchy. Therefore recovering of C++ specific language features is important for quality decompilation. This work is an step on this way.

In this work we present a method for reconstruction of hierarchies of polymorphic classes from assembly code obtained by compiling a C++ program. We presume that no modifications of the assembly code were performed after compilation (such as assembly-level obfuscation).

If the program was compiled with run-time type information (RTTI) enabled, hierarchy of polymorphic classes is reconstructed exactly as it was in the source C++ program. Multiple and virtual inheritance are handled correctly. As run-time type information structures store class names, class names are also recovered.

If the program was compiled without run-time type information, we consider an induced inheritance relation on a set of virtual function tables. This relation is reconstructed via analysis of virtual function tables and virtual destructors. Multiple inheritance is also partially handled by gathering information on virtual function tables.

For a prototype implementation we considered C++ ABI (application binary interface) of Microsoft Visual Studio compiler on Windows platform and C++ ABI of GNU C++ compiler on Windows. We use MSVC 9.1 and G++ 4.2.4 for experimental study, but the prototype tool also work for other versions of these compilers provided they use the same C++ ABI.

The paper is organized as follows. Section 2 discusses related work. Class hierarchy reconstruction via analysis of RTTI structures is presented in Section 3. Section 4 presents methods for class hierarchy reconstruction without RTTI structures. The experimental results are discussed in Sec-

tion 5. Our conlusions and directions for future work are presented in the last section.

2 Related work

There have been a relatively small amount of work on automatic reconstruction of class hierarchies from assembly code.

Sabanal and Yason [6] have proposed a technique based on the analysis of constructors and destructors. In case RTTI is present, information on class hierarchy is extracted from RTTI structures. If RTTI is not present, classes are identified by searching for constructors, destructors and virtual tables. This approach also allows detecting non-polymorphic classes in some cases. Class relationship inference for polymorphic classes is done via constructor analysis.

Skochinsky [7] has given a detailed description of RTTI structures used by MSVC, along with implementation details of some of the C++ concepts, such as constructors and destructors. He has successfully used RTTI structure analysis for polymorphic class hierarchy reconstruction.

Paper [3] describes the experience gained from applying a native executable decompiler, assisted by a commercial disassembler and hand editing, to a real-world Windows-based application. Authors were able to recover almost all original class names and the complete class hierarchy via analysis of RTTI structures.

3 Analysis of RTTI structures

3.1 Run-time type information in C++

Run-time type information in C++ is used for implementing the following operators:

- The **dynamic_cast**<> operator. This operator performs type conversions at run time. The **dynamic_cast**<> operator guarantees the conversion of a pointer to a base class to a pointer to a derived class, or the conversion of an lvalue referring to a base class to a reference to a derived class. A program can thereby use a class hierarchy safely.

- **typeid** operator. This operator provides a program with the ability to retrieve the actual type of an object referred to by a pointer or a reference.

The **typeid** operator can be applied to non-polymorphic types. In this case it is evaluated at compile time. On the contrary, the **dynamic_cast**<> operator works with polymorphic types only, and results in undefined behavior in case the program was compiled without RTTI.

The layout of RTTI structures used for implementing the **typeid** and the **dynamic_cast**<> operators is defined by the application binary interface (ABI) used by the C++ compiler. The layout varies from compiler to compiler and from platform to platform. For example, Microsoft uses an undocumented format, which, however, has been successfully reverse-engineered, and can be looked up in source code of Wine. The layout of RTTI structures used in GCC is defined in the <cxxabi.h> header file, which is supplied with the compiler. In this paper we presume that the format of RTTI structures used by the compiler is known and can be parsed.

3.2 Class hierarchy reconstruction

In case RTTI structures are present in assembly, the process of class hierarchy reconstruction can be split into the following steps:

1. Locate RTTI structures.
2. Parse RTTI structures.
3. Use obtained information to reconstruct a polymorphic class hierarchy.

The layout of the RTTI structures in assembly is determined by the ABI used. As there are quite few different RTTI layouts, it is practical to repeat the above steps several times, once for each layout, and then choose the most plausible variant manually.

In certain cases it is possible to determine the compiler used. Firstly, it can be determined by analyzing the standard library the program uses. If the standard library is linked dynamically, the compiler can easily be determined given the shared library file. If the standard library is linked statically, its vendor can be determined via function signature comparison.

Given the layout and position of the RTTI structures in assembly, parsing them imposes no difficulties. Then full polymorphic class hierarchy can be reconstructed by merging all partial class hierarchies obtained from RTTI structures.

3.3 Locating the RTTI structures

In all ABIs known to the moment the pointer to the RTTI structure of a class always precedes the corresponding virtual function table, so it can be accessed like `vptr[-1]`. Therefore, the problem of finding the RTTI structures can be reduced to a problem of locating virtual function tables. For each virtual function table the following statements hold:

- Each virtual function table is an array of pointers to functions.

- Only the first element of the virtual function table is referenced from program code — its address is used in constructors and destructor of the corresponding class.

Algorithm *LocateVTables* makes use of these statements for finding virtual function tables. It uses the following functions:

isReferenced — determines whether the given address is referenced from the program code.

isPointerToFunction — determines whether the given pointer is a pointer to a function.

Algorithm *LocateVTables*
Output: Set of pairs (virtual table position, virtual table size)
1. *vTables* $\leftarrow \emptyset$
2. $p \leftarrow$ *DataSegment*$_{start}$
3. **while** $p <$ *DataSegment*$_{end}$ **do**
4. **if** *isReferenced*(p) **and** *isPointerToFunction*$(*p)$ **then**
5. *vTableSize* $\leftarrow 0$
6. $p' \leftarrow p$
7. **repeat**
8. $p \leftarrow p +$ **sizeof** (**void**$*$)
9. *vTableSize* $\leftarrow$ *vTableSize* $+ 1$
10. **until** *isReferenced*(p) **or not** *isPointerToFunction*$(*p)$
11. *vTables* $\leftarrow$ *vTables* $\cup \{(p', \textit{vTableSize})\}$
12. **else**
13. $p \leftarrow p +$ **sizeof** (**void**$*$)
14. **return** *vTables*

4 Absence of RTTI structures

Run-time type information in C++ programs is considered frequently misused [8], and some modern applications written in C++ refrain from using it. The virtual function tables are still generated for each polymorphic class, however, uniqueness of virtual function tables for each class is not guaranteed, i.e. the same virtual function table can be shared by several different classes.

As it was stated in the introduction, we assume that the source C++ program does not use virtual inheritance. That means that the induced inheritance relation on a set of virtual functions is a single inheritance relation.

A set of all virtual function tables $\mathfrak{C}$ can be built using algorithm *LocateVTables*. The induced inheritance relation $\leftarrow \subseteq \mathfrak{C} \times \mathfrak{C}$ is defined as $\forall B, D \in \mathfrak{C} : B \leftarrow D \iff$ one of the classes corresponding to the virtual function table B is a direct base of one of the classes corresponding to the virtual function table D. In this case we also say that virtual function table B is a direct base of the virtual function table D. We work with inheritance relation $\lhd = \leftarrow^+$, which is a transitive closure of the direct inheritance relation $\leftarrow$. This relation defines a strict partial order on $\mathfrak{C}$.

Relation $\lhd$ can be decomposed into two relations $\not\rhd$ and $\sim$:

$$\begin{aligned} \not\rhd &= \mathfrak{C}^2 \setminus \rhd, \\ \sim &= \lhd \cup \rhd, \end{aligned} \tag{1}$$

i.e. $B \not\rhd D$ means that D is not a direct or indirect base of B, and $A \sim C$ means that A is a direct or indirect base of C, or C is a direct or indirect base of A. Relation $\lhd$ can be reconstructed from relations $\not\lhd$ and $\sim$ as follows:

$$\lhd = \not\rhd \cap \sim \tag{2}$$

The following notation is used:

$|A|$ — the size of virtual function table $A \in \mathfrak{C}$.

A_i — the i-th entry in a virtual function table A. Pointer in a virtual function table may point to an *adjuster thunk* that adjusts the passed **this** pointer and then transfers control to the actual method body. In case there is no adjuster thunk, we presume that zero adjustment is used. Each entry A_i is a pair of **this** adjustment value A_i^{adj} and a pointer to an actual method body A_i^{func}.

$|\textit{params}(f)|$ — the size of the parameters of function f, in bytes.

pure — the address of the pure virtual function call handler.

executes — a relation between a set of functions and a set of expressions. In case f *executes* e and g *executes* $f(/*\ \ldots\ */)$ for some functions f and g and an expression e we consider g *executes* e to be true.

4.1 Retrieving inheritance relation

Statement 1. *Let $B, D \in \mathfrak{C} : |B| < |D|$. Then restriction $\mathcal{R}_1 = B \not\rhd D$ is satisfied.*

In compliance with the C++ inheritance rules [2], if there are more virtual functions in D than in B, then D cannot be a base of B.

Statement 2. *Let $B, D \in \mathfrak{C}, i \geq 0 : B_i^{func} =$ pure and $D_i^{func} \neq$ pure. Then restriction $\mathcal{R}_2 = B \not\rhd D$ is satisfied.*

In C++ it is impossible to override a virtual function that is not pure with a pure one [2].

Statement 3. *Let $A, C \in \mathfrak{C}, i \geq 0 : A_i = C_i$ and $A_i^{func} \neq$ pure. If during compilation for each definition of a function in the source file a distinct function in the resulting assembly was generated, then restriction $\mathcal{R}_3 = A \sim^+ C$ is satisfied, where $\cdot^+$ is a transitive closure operation.*

If the same function appears on the same position in two virtual function tables, this function is inherited by one of these tables from another, or it is inherited from some base, or it is a result of optimization.

Experiments with GCC and MSVC have shown that neither GCC nor MSVC perform pairwise function comparison during compilation to eliminate identical functions.

However, during compilation of a single compilation unit, MSVC eliminates identical functions with simple bodies like **return** */* const */*. Further experiments have shown that there is a limit on the length of the assembly code of a function that can be eliminated this way. Let *isPrimitive*(f) return true if the assembly code of the function f is shorter than this limit, i.e. f could have been generated as a result of identical function elimination.

Consider the following refinement of statement 3:

Statement 4. *Let* $\mathfrak{f} \subseteq \mathfrak{C}, i \geq 0$, *and* $\forall A, C \in \mathfrak{f} : A_i = C_i \wedge \neg$*isPrimitive*$(A_i^{func}) \wedge A_i^{func} \neq$ *pure, and* $\forall A \in \mathfrak{C}, C \in \mathfrak{C} \setminus \mathfrak{f} : A_i \neq C_i$. *Then restriction* $\mathcal{R}_4 = (\mathfrak{f}, \leftarrow[\mathfrak{f}]) \in \mathbb{T}(\mathfrak{f})$ *is satisfied.*

Here $\cdot[\cdot]$ is a reduction operation, i.e. for $R \subseteq X \times X$, $R[Y] = \{(a,b) | (a,b) \in R \wedge a, b \in Y\}$, and $\mathbb{T}(X)$ is a set of all trees with nodes from X. This statement says that reduction of an inheritance hierarchy defined by a relation $\leftarrow$ on set $\mathfrak{f}$ is a tree.

There are several cases when this statement fails. Consider the following program:

```
class A {
public:
    virtual void f() { /* ... */ }
};

class B: public A {
public:
    virtual void f() { /* ... */ }
};

class C: public B {
public:
    virtual void f() { A::f(); }
};

class D {
    virtual void f() {
        ((A*)this)->A::f();
    }
}
```

Due to optimizations, function A::f can be stored directly in the virtual function table of class D, which is unrelated to A, without the body of the function D::f being generated. But such cases are rare in real code. While the function call in function D::f can be considered a "dangerous" C++, function call in C::f is perfectly safe. Due to optimizations function A::f can also be stored directly in the virtual function table of class C, and that would also break statement 4. But such cases are also rare in real code. Corresponding class hierarchy for this case is presented on the fig. 1(a). Reduction of inheritance tree on the set $\{A, C\}$ is not a tree, and therefore statement 4 cannot be applied.

Under the assumption that the above-mentioned situations do not occur in the source C++ code, statement 4 is true. Fig. 1(b) illustrates its application. On this figure virtual function tables A, B and D have the same function A::f on the first position, and therefore reduction of the hierarchy on the set $\{A, B, D\}$ is a tree. The same is true for the set $\{B, C, E\}$.

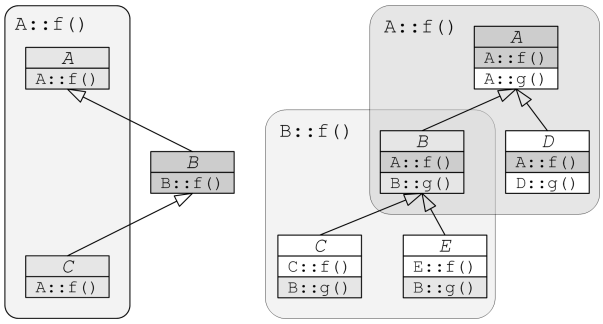

Figure 1. Example for statement 4.

If it is possible to reliably determine the size of the parameters of a virtual function (for example, if the function uses a calling convention where the callee clears the stack), then the following statement holds.

Statement 5. *Let* $A, C \in \mathfrak{C}, i \geq 0 : |$*params*$(A_i^{func})| \neq |$*params*$(C_i^{func})|$. *Then restriction* $\mathcal{R}_5 = \neg(A \sim C)$ *is satisfied.*

In compliance with the C++ inheritance rules [2], a virtual function cannot be overridden by a function with different parameters.

Some information on relation $\leftarrow$ can also be gathered using an analysis of destructors and constructors. A constructor of a class performs the following sequence of operations [2, 4]:

1. Calls constructors of direct base classes.
2. Calls constructors of data members.
3. Initializes virtual function table pointer field(s).
4. Performs user-specified initialization code in the body of the constructor.

So, in case of a "deep" inheritance hierarchy, a construction of an object may require many successive initializations of the class's virtual function table pointer. Where appropriate, these assignments are optimized away by the compiler, but the last assignment is never optimized away. Conversely, a destructor must deinitialize the object in the exact reverse order to how it was initialized:

1. Initializes the virtual function table pointer field(s).
2. Performs user-specified destruction code in the body of the destructor.
3. Calls destructors of data members.

4. Calls destructors of direct bases.

Destruction of an object may also require many successive initializations of the class's virtual function table pointer. Experiments have shown that neither GCC nor MSVC optimize away the last initialization. That means that the destructor of the object always overwrites the virtual function table pointer field with a pointer to a "most-base" virtual table. The inheritance reconstruction method presented in this paper highly relies on this fact.

Constructors and a destructor are the only functions that access the virtual function table pointer field of a class. By analyzing these accesses, it is possible to find bases of a given virtual table, so the following statement holds.

Statement 6. *Let $B, D \in \mathfrak{C} : B \neq D$ and $i \geq 0$: D_i^{func} executes* ****(void **)((char*)this** $- D_i^{adj}$**)** *= $\&B$. Then restriction $\mathcal{R}_6 = B \lhd D$ is satisfied.*

In compliance with the ABIs used by GCC and MSVC compilers [1, 4], **this** pointer passed to a virtual function always points to a field containing pointer to the virtual function table corresponding to this function. Therefore the statement $*$**(void**$**$**)((char**$*$**)this** $- D_i^{adj}$) = $\&B$ means that the virtual function table pointer corresponding to virtual table D is overwritten with a pointer to a virtual table B. According to the actions that must be performed by a destructor, in this case $B \lhd D$. Moreover, since virtual function table pointer can be overwritten only in a call to destructor, the order in which it is overwritten with different virtual function table pointers actually defines the inheritance relation $\lhd$ between all these virtual tables.

Since most of the polymorphic class hierarchies use virtual destructors, in most cases statement 6 will produce at least one relation $\lhd$ — between the virtual function table being analyzed and the "most-base" one.

The virtual function table pointer assignments can be detected via data flow analysis. In case calling conventions of a virtual function are known, the way **this** pointer is passed to a function is also known, and data flow analysis can be used to find such assignments.

4.2 Inheritance hierarchy reconstruction

As it was denoted in chapter 4.1, for each virtual table D with a virtual destructor, a $B \lhd D$ relation can be reconstructed using statement 6, where B is the "most-base" virtual table of D. Therefore, using statement 6, it is possible to construct a set of all virtual tables in a single connected inheritance hierarchy with virtual destructors. In a rare case when virtual destructors are not used, this set can still be restored in most cases using statement 4. This makes possible to subdivide set $\mathfrak{C}$ into several sets of virtual tables, so that all virtual tables in a single set form a connected inheritance hierarchy. Since we do not consider virtual inheritance, such hierarchy is a rooted tree. Further in this paper we presume that $\mathfrak{C}$ is one of such sets, i.e. virtual tables in $\mathfrak{C}$ form an inheritance tree.

Let $\mathfrak{R} \subset \mathbb{T}^R(\mathfrak{C}) \to \{0, 1\}$ be a set of all restrictions on the inheritance tree, obtained by applying statements 1, 2, 4, 5, and 6, where $\mathbb{T}^R(X)$ is a set of all rooted trees with nodes from X. Let the function *isSolution* for the given rooted tree with nodes from $\mathfrak{C}$ and a set of restrictions return whether this tree satisfies all the given restrictions. Then a problem of virtual table hierarchy reconstruction can be considered as follows:

$$\text{Find } T \in \mathbb{T}^R(\mathfrak{C}) : \textit{isSolution}(T, \mathfrak{R}). \tag{3}$$

Since we have made some assumption on the structure of the source C++ program, it may be possible that all restrictions from $\mathfrak{R}$ cannot be jointly satisfied. That's why we will be solving the relaxed problem:

$$\text{Find } \mathfrak{R}' \subseteq \mathfrak{R}, T \in \mathbb{T}^R(\mathfrak{C}) : \textit{isSolution}(T, \mathfrak{R}') \wedge \forall \mathcal{R} \in \mathfrak{R} \setminus \mathfrak{R}' : \nexists T \in \mathbb{T}^R(\mathfrak{C}) : \textit{isSolution}(T, \mathfrak{R}' \cup \{\mathcal{R}\}) \tag{4}$$

4.3 Building a partial solution

We assume restrictions $\mathcal{R}_4$ to be jointly satisfiable. If they are not, which is a rare case, additional manual analysis will be required.

Let $\mathfrak{F}$ be the set of all sets $\mathfrak{f}$ from restrictions $\mathcal{R}_4$. Then $\forall \mathfrak{f} \in \mathfrak{F} : \textit{isSolution}(T, \mathfrak{R}) \Longrightarrow T[\mathfrak{f}] \in \mathbb{T}(\mathfrak{f})$, i.e. a reduction of a solution tree on any of the sets $\mathfrak{f} \in \mathfrak{F}$ is a tree. Let the function *isPartSolution* for the given rooted tree with nodes from $\mathfrak{C}$ and a set of subsets of $\mathfrak{C}$ return whether the reduction of this tree on all of these subsets is a tree. Then $\textit{isSolution}(T, \mathfrak{R}) \Longrightarrow \textit{isPartSolution}(T, \mathfrak{F})$. Let

$$\mathfrak{F}^* = \bigcup_{\mathfrak{F}' \subseteq \mathfrak{F}} \left\{ \bigcap_{\mathfrak{f} \in \mathfrak{F}'} \mathfrak{f} \right\} \cup \{\mathfrak{C}\} \cup \bigcup_{C \in \mathfrak{C}} \{\{C\}\} \setminus \emptyset, \tag{5}$$

i.e. $\mathfrak{F}^*$ is an intersection closure of set $\mathfrak{F}$ with some additional elements. For $\mathfrak{F}^*$ the following condition is satisfied:

$$\forall T \in \mathbb{T}(\mathfrak{C}) : \textit{isPartSolution}(T, \mathfrak{F}) \Longrightarrow \textit{isPartSolution}(T, \mathfrak{F}^*). \tag{6}$$

Consider a $\subset$ relation on set $\mathfrak{F}^*$. This relation defines a strict partial order on $\mathfrak{F}^*$, and a Hasse diagram can be constructed for it. A Hasse diagram is a directed graph representation of a partially ordered set in which each element is represented by a node. Immediate successors of each element are connected to the corresponding node by a directed edge [5]. Let $\mathfrak{H}$ be a Hasse diagram for $(\mathfrak{F}^*, \subset)$, and $\mathfrak{H}_E$ be

Algorithm *BuildPartialSolution*
Output: Tree that is a partial solution of problem (4)
1. $T_E \leftarrow \emptyset$
2. **for all** $\mathfrak{f} \in \mathfrak{F}^*$ **do**
3. $\quad \mathfrak{D} \leftarrow \{\mathfrak{f}' \in \mathfrak{F}^* | (\mathfrak{f}', \mathfrak{f}) \in \mathfrak{H}_E\}$
4. $\quad \mathfrak{K} \leftarrow \mathfrak{D}$
5. $\quad$ **while** $\exists \mathfrak{f}_1, \mathfrak{f}_2 \in \mathfrak{K} : \mathfrak{f}_1 \cap \mathfrak{f}_2 \neq \emptyset$ **do**
6. $\quad\quad \mathfrak{K} \leftarrow \mathfrak{K} \setminus \{\mathfrak{f}_1, \mathfrak{f}_2\} \cup \{\mathfrak{f}_1 \cup \mathfrak{f}_2\}$
7. $\quad t \leftarrow$ *any*$(\mathbb{T}(\mathfrak{K}))$
8. $\quad$ **for all** $\{\mathfrak{f}_1, \mathfrak{f}_2\} \in t_E$, where t_E is a set of edges of t **do**
9. $\quad\quad T_E \leftarrow T_E \cup \{$*any*$(\mathfrak{f}_1),$ *any*$(\mathfrak{f}_2)\}$
10. **return** tree $(\mathfrak{C}, T_E)$

a set of all edges of $\mathfrak{H}$, i.e. if $(\mathfrak{f}_1, \mathfrak{f}_2) \in \mathfrak{H}_E$ then $\mathfrak{f}_1 \subset \mathfrak{f}_2$. We consider nodes of Hasse diagram $\mathfrak{H}$ to be sets $\mathfrak{f} \in \mathfrak{F}^*$.

Algorithm *BuildPartialSolution* builds a partial solution of a relaxed problem (4). For this algorithm the following theorem can be proven.

Theorem 1. *If* $\exists T' \in \mathbb{T}(\mathfrak{C})$: *isPartSolution*$(T', \mathfrak{F}^*)$, *then regardless of the values returned by functions any, algorithm BuildPartialSolution builds a tree* $T \in \mathbb{T}(\mathfrak{C})$: *isPartSolution*$(T, \mathfrak{F}^*)$.

Proof. This theorem can be proven by induction. Let T be some tree built using algorithm *BuildPartialSolution*. If for all sets $\mathfrak{f}_i \in \mathfrak{F}^* : \mathfrak{f}_i \subseteq \mathfrak{f}$, condition $T[\mathfrak{f}_i] \in \mathbb{T}(\mathfrak{f}_i)$ is satisfied, then it is also satisfied for $\mathfrak{f}$, i.e. $T[\mathfrak{f}] \in \mathbb{T}(\mathfrak{f})$. Since $\mathfrak{C} \in \mathfrak{F}^*$, condition *isPartSolution*$(T, \mathfrak{F}^*)$ will also be met.

The base of induction is $|\mathfrak{f}| = 1$. The induction step can be proven from the contrary, assuming that there is a cycle in $T[\mathfrak{f}]$. In this case cycle will also be present in T', which contradicts the hypothesis of the theorem. □

A consequence of this theorem is that under the additional requirement that all the restrictions $\mathcal{R}_4$ must be jointly satisfied, any solution of problem (4) can be generated using algorithm *BuildPartialSolution*.

4.4 Building a complete solution

For each node $\mathfrak{f}$ of Hasse diagram $\mathfrak{H}$ algorithm *BuildPartialSolution* builds set $\mathfrak{D}$. We call this set a set of *child nodes* for node $\mathfrak{f}$. For each node $\mathfrak{f}$ algorithm *BuildPartialSolution* also builds set $\mathfrak{K}$. Let $\mathfrak{F}^\cup$ be a union of sets $\mathfrak{K} \setminus \mathfrak{D}$ for all nodes of Hasse diagram $\mathfrak{H}$. Let $\mathfrak{G}$ be a Hasse diagram for $(\mathfrak{F}^* \cup \mathfrak{F}^\cup, \subset)$, and let all the nodes of this graph be divided into two types — *any*-nodes and *fixed*-nodes. Node $\mathfrak{f}$ is a *fixed*-node, if $\mathfrak{f} \in \mathfrak{F}^\cup$, and an *any*-node, if $\mathfrak{f} \in \mathfrak{F}^*$. Note that each two child nodes of a *any*-node have empty intersection, and each two child nodes of a *fixed*-node have non-empty intersection. That means that if tree T is a partial solution for problem (4), then for any *fixed*-node $\mathfrak{f}$ the structure of the tree $T[\mathfrak{f}]$ is fully determined by the structure of corresponding trees of its children, while for each *any*-node $\mathfrak{f}'$ the structure of the tree $T[\mathfrak{f}']$ is defined by the return values of functions *any* in algorithm *BuildPartialSolution*.

Fig. 2 illustrates the difference between graphs $\mathfrak{H}$ and $\mathfrak{G}$. On Fig. 2(b) a fragment of graph $\mathfrak{G}$ is presented that corresponds to the fragment of graph $\mathfrak{H}$ presented on Fig. 2(a).

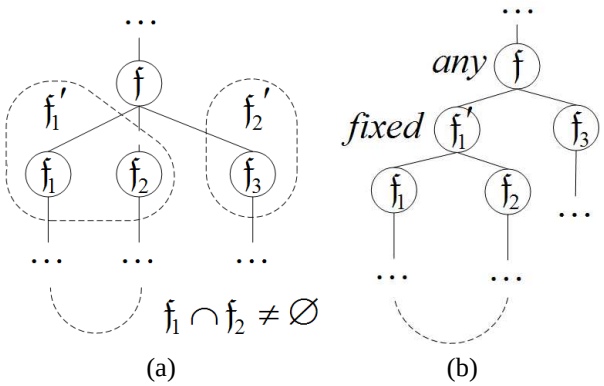

Figure 2. Fragments of graphs $\mathfrak{H}$ and $\mathfrak{G}$.

Solution of the problem (4) is constructed using graph $\mathfrak{G}$. The following functions will be used:

children — returns the set of child nodes for the given node $\mathfrak{f}$.

type — returns the type of the given node, *any* or *fixed*.

vtable — if for the given node $\mathfrak{f}$ condition $|\mathfrak{f}| = 1$ is satisfied, then returns $\mathfrak{f}$, else returns $\emptyset$.

adjacent — for the given node returns the set of adjacent nodes in graph $\mathfrak{G}$.

parents — returns the set of adjacent non-child nodes for the given node, i.e. *parents*$(\mathfrak{f})$ = *adjacent*$(\mathfrak{f})$ \ *children*$(\mathfrak{f})$. Note that if *type*$(\mathfrak{f})$ = *fixed*, then there is only one node in *parents*$(\mathfrak{f})$.

Graph $\mathfrak{G}$ is constructed so that there are no cycles of *any*-nodes in $\mathfrak{G}$. Also for each *any*-node:

$$|\textit{children}(\mathfrak{f})| \leq |\mathfrak{C}|, \tag{7}$$

Let set $\mathfrak{C}'_{(\mathfrak{f}_1,\mathfrak{f}_2)}$, where $\mathfrak{f}_1$ and $\mathfrak{f}_2$ are two adjacent nodes in $\mathfrak{G}$, be the set of all reachable virtual tables in the direction from $\mathfrak{f}_1$ to $\mathfrak{f}_2$. It is defined as follows:

$$\begin{gathered}\textit{type}(\mathfrak{f}_1) = \textit{type}(\mathfrak{f}_2) = \textit{any} \wedge \textit{vtable}(\mathfrak{f}_2) \neq \emptyset \\ \Longrightarrow \mathfrak{C}'_{(\mathfrak{f}_1,\mathfrak{f}_2)} = \textit{vtable}(\mathfrak{f}_2).\end{gathered} \tag{8}$$

Else, if $\mathfrak{f}_1$ and $\mathfrak{f}_2$ are *any*-nodes, or one of the nodes $\mathfrak{f}_1$ and $\mathfrak{f}_2$ is a *fixed*-node, and the other one is an *any*-node, and *fixed*-node is a child of *any*-node, then:

$$\mathfrak{C}'_{(\mathfrak{f}_1,\mathfrak{f}_2)} = \bigcup_{\mathfrak{f} \in \textit{adjacent}(\mathfrak{f}_2) \setminus \mathfrak{f}_1} \mathfrak{C}'_{(\mathfrak{f}_2,\mathfrak{f})}. \tag{9}$$

In case neither of the two conditions are met, $\mathfrak{C}'_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ is empty. Since there are no cycles of *any*-nodes in $\mathfrak{F}$, recursive definition (9) is correct.

For any two nodes $\mathfrak{f}_1$ and $\mathfrak{f}_2$ let $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)} = \mathfrak{C}'_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ with the additional condition that if $\mathfrak{f}_1$ is an *any*-node, and $\mathfrak{f}_2$ is the closest to $\mathfrak{f}_1$ direct or indirect parent *fixed*-node, then:

$$\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)} = \mathfrak{C} \setminus \left(\mathit{vtable}(\mathfrak{f}_1) \cup \bigcup_{\mathfrak{f} \in \mathfrak{C}} \mathfrak{C}'_{(\mathfrak{f}_1,\mathfrak{f})} \right). \quad (10)$$

Then the following conditions are met:

$$\begin{aligned} &\forall \mathfrak{f} \in \mathfrak{G}_V : \bigcup_{\mathfrak{f}' \in \mathfrak{G}_V} \mathfrak{C}_{(\mathfrak{f},\mathfrak{f}')} = \mathfrak{C} \setminus \mathit{vtable}(\mathfrak{f}), \\ &\forall \mathfrak{f}_1, \mathfrak{f}_2 \in \mathfrak{G}_V : \mathfrak{C}_{(\mathfrak{f},\mathfrak{f}_1)} \cap \mathfrak{C}_{(\mathfrak{f},\mathfrak{f}_2)} = \emptyset, \end{aligned} \quad (11)$$

where $\mathfrak{G}_V$ is the set of all nodes of graph $\mathfrak{G}$.

The presented algorithm iteratively builds the following sets for each two child nodes $\mathfrak{f}_1$ and $\mathfrak{f}_2$ of each *any*-node $\mathfrak{f} \in \mathfrak{G}$:

$\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ — the set of virtual tables, that must be inherited by the root of $T[\mathfrak{f}_1]$ in case its base lies in $\mathfrak{f}_2$, where T is a solution of problem 4.

$\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ — the set of virtual tables, that must not be inherited by the root of $T[\mathfrak{f}_1]$ in case its base lies in $\mathfrak{f}_2$.

For these sets the following condition is satisfied:

$$\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}, \mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \subseteq \mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \quad (12)$$

The presented algorithm also works with an $\lhd$ relation defined on the set of child nodes of each *any*-node $\mathfrak{f}$. In case $\mathfrak{f}_1$ and $\mathfrak{f}_2$ are child nodes of $\mathfrak{f}$, $\mathfrak{f}_1 \lhd \mathfrak{f}_2$ means that the base of the root of $T[\mathfrak{f}_2]$ lies in $\mathfrak{f}_1$. For each *any*-node $\mathfrak{f}$ relation $\lhd$ must define a rooted tree $T_{\mathfrak{f}} \in \mathbb{T}^R(\mathit{children}(\mathfrak{f}))$. Let this relation be described with a set of restrictions $\mathfrak{R}_{\mathfrak{f}}$ of the following form:

$$\begin{aligned} &\mathcal{R} = b \not\rhd d, \text{ or} \\ &\mathcal{R} = a \sim c. \end{aligned} \quad (13)$$

Restrictions $\mathcal{R} = \neg(a \sim c)$ can be represented as two restrictions $\mathcal{R}_{\not\rhd} = a \not\rhd c$ and $\mathcal{R}_{\not\lhd} = a \not\lhd c$. Restrictions $\mathcal{R} = a \lhd c$ can also be represented as two restrictions $\mathcal{R}_{\not\rhd} = a \not\rhd c$ and $\mathcal{R}_{\sim} = a \sim c$.

All restrictions from $\mathfrak{R}$, except for restrictions $\mathcal{R}_4$, can also be represented in the form (13). Let $\mathfrak{R}_s \subseteq \mathfrak{R}$ — the set of all restrictions from $\mathfrak{R}$, except for restrictions $\mathcal{R}_4$.

As it was denoted at the end of chapter 4.2, not all of the restrictions from $\mathfrak{R}$ may be jointly satisfiable. Therefore, solution tree T is built by considering restrictions from $\mathfrak{R}_s$ one by one. In case the restriction results in a conflict and construction of the solution tree become impossible, the last step of the algorithm is undone, and the restriction under consideration is added to set $\mathfrak{R}_\times$. After processing all restrictions, set $\mathfrak{R} \setminus \mathfrak{R}_\times$ corresponds to set $\mathfrak{R}'$ from definition of the relaxed problem (4).

Algorithm *ProcessSim*
Input: Virtual tables $A, C \in \mathfrak{C} : A \sim C \in \mathfrak{R}$
1. **let** $\mathfrak{f}_1 : \mathit{vtable}(\mathfrak{f}_1) = \{C\}$
2. **while** $\mathit{vtable}(\mathfrak{f}_1) \neq \{A\}$ **do**
3. ($*$ Due to (11) there exists only one set $\mathfrak{f}_2$ $*$)
4. **let** $\mathfrak{f}_2 \in \mathit{adjacent}(\mathfrak{f}_1) : A \in \mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$
5. **if** $\mathit{type}(\mathfrak{f}_2) = \mathit{any} \wedge \mathfrak{f}_1 \in \mathit{children}(\mathfrak{f}_2) \wedge \exists \mathfrak{f}_3 \in \mathit{children}(\mathfrak{f}_2) : B \in \mathfrak{C}_{(\mathfrak{f}_2,\mathfrak{f}_3)}$ **then**
6. $\mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_1,\mathfrak{f}_3)} \leftarrow \mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_1,\mathfrak{f}_3)} \cup \{C\}$
7. $\mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_3,\mathfrak{f}_1)} \leftarrow \mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_3,\mathfrak{f}_1)} \cup \{A\}$
8. **elseif** $\mathit{type}(\mathfrak{f}_2) = \mathit{fixed} \wedge \mathfrak{f}_2 \in \mathit{children}(\mathfrak{f}_1)$ **then**
9. **break**
10. $\mathfrak{f}_1 \leftarrow \mathfrak{f}_2$

Algorithm *ProcessNotDerived*
Input: Virtual tables $B, D \in \mathfrak{C} : B \not\rhd D \in \mathfrak{R}$
1. **let** $\mathfrak{f}_1 : \mathit{vtable}(\mathfrak{f}_1) = \{D\}$
2. **while** $\mathit{vtable}(\mathfrak{f}_1) \neq \{B\}$ **do**
3. ($*$ Due to (11) there exists only one set $\mathfrak{f}_2$ $*$)
4. **let** $\mathfrak{f}_2 \in \mathit{adjacent}(\mathfrak{f}_1) : B \in \mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$
5. **if** $\mathit{type}(\mathfrak{f}_2) = \mathit{any} \wedge \mathfrak{f}_1 \in \mathit{children}(\mathfrak{f}_2) \wedge \exists \mathfrak{f}_3 \in \mathit{children}(\mathfrak{f}_2) : B \in \mathfrak{C}_{(\mathfrak{f}_2,\mathfrak{f}_3)}$ **then**
6. $\mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_1,\mathfrak{f}_3)} \leftarrow \mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_1,\mathfrak{f}_3)} \cup \{B\}$
7. $\mathfrak{P}^{\mathfrak{f}_2}_{(\mathfrak{f}_3,\mathfrak{f}_1)} \leftarrow \mathfrak{P}^{\mathfrak{f}_2}_{(\mathfrak{f}_3,\mathfrak{f}_1)} \cup \{D\}$
8. **elseif** $\mathit{type}(\mathfrak{f}_2) = \mathit{fixed} \wedge \mathfrak{f}_2 \in \mathit{children}(\mathfrak{f}_1)$ **then**
9. **break**
10. $\mathfrak{f}_1 \leftarrow \mathfrak{f}_2$

The reconstruction algorithm starts with all sets $\mathfrak{R}_{\mathfrak{f}}$, $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ and $\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ being empty. It continuously processes restrictions from $\mathfrak{R}_s$ with one of algorithms *ProcessSim* and *ProcessNotDerived*, and then modifies the sets to be consistent. Following are consistency maintenance rules.

Let $\mathfrak{f}$ be an *any*-node, and $\mathfrak{f}_1$, $\mathfrak{f}_2$ and $\mathfrak{f}_3$ be child nodes of $\mathfrak{f}$. Then from the definition of $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ and $\sim$:

$$\begin{aligned} &\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \neq \emptyset \wedge \mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_2,\mathfrak{f}_1)} \neq \emptyset \\ &\quad \Longrightarrow (\mathfrak{f}_1 \sim \mathfrak{f}_2) \in \mathfrak{R}_{\mathfrak{f}} \end{aligned} \quad (14)$$

Sets $\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ and $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ must not conflict:

$$\begin{aligned} &\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \cap \mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \neq \emptyset \\ &\quad \Longrightarrow (\mathfrak{f}_1 \not\rhd \mathfrak{f}_2) \in \mathfrak{R}_{\mathfrak{f}} \end{aligned} \quad (15)$$

After applying rules (14) and (15), sets $\mathfrak{R}_{\mathfrak{f}}$ may have been modified. For elements of these sets the following consistency maintenance rules are used:

$$\begin{aligned} &b \lhd d \Longleftrightarrow b \not\rhd d \wedge b \sim d \\ &b \lhd c \wedge c \lhd d \Longrightarrow b \lhd d \\ &d_1 \sim d_2, \ldots, d_{n-1} \sim d_n, d_n \sim d_1 \Longrightarrow \\ &\quad \forall a, c \in \{d_1, \ldots, d_n\} : a \sim c \end{aligned} \quad (16)$$

These rules are based on the properties of relations $\not\rhd$, $\lhd$ and $\sim$, and on the fact that they must define a tree on *children*($\mathfrak{f}$). Also there must be no conflicts in $\mathfrak{R}_\mathfrak{f}$, i.e. the following condition must be met:

$$\neg(b \sim d \wedge b \not\lhd d \wedge b \not\rhd d) \tag{17}$$

After applying rules (16), additional consistency maintenance rules for sets $\mathfrak{R}_\mathfrak{f}$ are applied. Two unrelated virtual tables cannot be base for a single virtual table, and therefore:

$$\begin{aligned}&\{\mathfrak{f}_1 \rhd \mathfrak{f}_2\} \subseteq \mathfrak{R}_\mathfrak{f}, A, C \in \mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)} : A \neq C \\ &\quad\Longrightarrow A \sim C\end{aligned} \tag{18}$$

There are no cycles of *any*-nodes in $\mathfrak{G}$, and therefore:

$$\begin{aligned}&\{\mathfrak{f}_1 \rhd \mathfrak{f}_2\} \subseteq \mathfrak{R}_\mathfrak{f}, B \in \mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}, D \in \mathfrak{C}_{(\mathfrak{f},\mathfrak{f}_1)} \\ &\quad\Longrightarrow D \rhd B\end{aligned} \tag{19}$$

In case $\{\mathfrak{f}_1 \rhd \mathfrak{f}_2\} \subseteq \mathfrak{R}_\mathfrak{f}$, it must be possible to inherit from $T[\mathfrak{f}_2]$ all virtual tables $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ without inheriting any of virtual tables $\mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$:

$$\begin{aligned}&\{\mathfrak{f}_1 \rhd \mathfrak{f}_2\} \subseteq \mathfrak{R}_\mathfrak{f}, B \in \mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}, D \in \mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \\ &\quad\Longrightarrow D \not\lhd B\end{aligned} \tag{20}$$

The outline of the algorithm is as follows:

1. A new restriction $\mathcal{R}$ from $\mathfrak{R}_s$ is selected. It is processed by algorithm *ProcessSim* or *ProcessNotDerived*.
2. The consistency maintenance rules are applied.
   2.1. Algorithms *ProcessSim* and *ProcessNotDerived* modify sets $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_2,\mathfrak{f}_1)}$ and $\mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$. For modified sets consistency maintenance rules (14) and (15) are applied.
   2.2. These consistency maintenance rules modify $\mathfrak{R}_\mathfrak{f}$ sets. For these sets consistency maintenance rules (16) are applied. If consistency criteria (17) is not met, then $\mathcal{R}$ is added to $\mathfrak{R}_\times$, all structures are returned to their state on the previous iteration, and the algorithm restarts from step 1.
   2.3. For modified sets $\mathfrak{R}_\mathfrak{f}$, consistency maintenance rules (18), (19), and (20) are applied. These rules add new restrictions on virtual tables from $\mathfrak{C}$. These restrictions are processed with algorithms *ProcessSim* and *ProcessNotDerived*, and for these restrictions steps 2.1-2.3 are repeated. Since all sets being processed are finite, this recursive process will stop.
3. It is checked whether for each node $\mathfrak{f}$ there exists a rooted tree $T_\mathfrak{f} \in \mathbb{T}^R(\textit{children}(\mathfrak{f}))$ satisfying $\mathfrak{R}_\mathfrak{f}$. If such tree does not exist, then $\mathcal{R}$ is added to $\mathfrak{R}_\times$, all structures are returned to their state on the previous iteration, and the algorithm restarts from step 1.
4. In case there are unprocessed restrictions in $\mathfrak{R}_s$, algorithm restarts from step 1.
5. For each *any*-node $\mathfrak{f}$, rooted tree $T_\mathfrak{f} \in \mathbb{T}^R(\textit{children}(\mathfrak{f}))$ satisfying $\mathfrak{R}_\mathfrak{f}$ is constructed. Existence of such tree was checked on step 3.
6. Trees $T[\mathfrak{f}] \in \mathbb{T}^R(\mathfrak{f})$ are constructed for *any*-nodes in an order so that tree for node $\mathfrak{f}$ is constructed only when trees for each child node of $\mathfrak{f}$ have already been constructed. Each tree $T[\mathfrak{f}]$ and $T_\mathfrak{f}$ can be seen as a directed graph with edges directed towards the root of this tree.
   6.1. For each directed edge $(\mathfrak{f}_1, \mathfrak{f}_2) \in T_\mathfrak{f}$ an edge connecting root R of $T[\mathfrak{f}_1]$ and a node from $\mathfrak{f}_2$ is added to $T[\mathfrak{f}]$, so that all nodes from $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ are reachable from R via directed edge walk, and all nodes from $\mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ are not. Existence of such edge is ensured by rule (20).
   6.2. If $\mathfrak{f}_R$ is the root of $T_\mathfrak{f}$, then the root of $T[\mathfrak{f}_R]$ is selected as the root of $T[\mathfrak{f}]$.

The way the solution tree is built in this algorithm corresponds to the way it is build in algorithm *BuildPartialSolution*, and according to theorem 1, this tree satisfies all the restrictions $\mathcal{R}_4$. Therefore, this tree is the solution of the relaxed problem (4).

4.5 Complexity analysis

Let $n = |\mathfrak{C}|$, and $N = |\mathfrak{F}^*|$. By definition (5), $N \geq n$. The number of *any*-nodes in $\mathfrak{G}$ equals $|\mathfrak{F}^*|$, and due to (11), the total size of all $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ sets does not exceed nN.

Every *fixed*-node in $\mathfrak{G}$ can only have *any*-nodes as children. Considering the fact that each *any*-node can have at most one parent *fixed*-node, the number of *fixed*-nodes in $\mathfrak{G}$ does not exceed the number of *any*-nodes. It means that:

$$|\mathfrak{G}_V| \leq 2N. \tag{21}$$

There are no cycles of *any*-nodes in $\mathfrak{G}$, and therefore the number of edges connecting *any*-nodes in $\mathfrak{G}$ does not exceed N. Since each *fixed*-node has exactly one parent *any*-node, the total number of child nodes of all *any*-nodes does not exceed $2N$:

$$\sum_{\mathfrak{f} \in \mathfrak{G}_V : \textit{type}(\mathfrak{f}) = \textit{any}} |\textit{children}(\mathfrak{f})| \leq 2N. \tag{22}$$

The implication of (21) is that the total size of all sets corresponding to the vertices of $\mathfrak{G}$ does not exceed $2nN$. Also (7) and (22) imply that the total number of restrictions in sets $\mathfrak{R}_{\mathfrak{f}}$ does not exceed $4nN$. Implication of (11), (12), and (22) is that the total size of all sets $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ and $\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ does not exceed $4nN$. The number of different restrictions of type (13) on $\mathfrak{C}$ does not exceed $2n^2$. Therefore, memory requirements of the presented algorithm are $O(nN)$.

Estimation of computational complexity of the presented algorithm is given in an assumption that it rarely performs restarts with modification of set $\mathfrak{R}_{\times}$, and these restarts do not affect the total computational complexity. Set $\mathfrak{F}^*$ can be constructed in $O(nN^2)$ operations. Conditions (14)–(15) are checked every time a new element is added to one of sets $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ and $\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$. Since the added element is known, these conditions can be checked in $O(n^2)$ operations. Conditions (16), (17), (18), (19), and (20) are checked every time a new element is added to one of sets $\mathfrak{R}_{\mathfrak{f}}$. Since the added element is known, these conditions can also be checked in $O(n^2)$.

Each *any*-node in $\mathfrak{G}$ can have a single child node only if this node is a *fixed*-node. Therefore, since each time the algorithms *ProcessSim* or *ProcessNotDerived* step down from a parent to a child node, they either step into a *fixed*-node and stop, or step into an *any*-node, and that makes the size of set $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ decrease on the next iteration of the algorithm. Due to the fact that the longest path in $\mathfrak{G}$, where each next node is the parent of the previous one, is no longer than n, and (11), the number of iterations of algorithms *ProcessSim* and *ProcessNotDerived* is no more than $2n$. Using appropriate data structures, each step of algorithms *ProcessSim* and *ProcessNotDerived* can be performed in $O(\log n)$, and therefore the computational complexity of these algorithms is $O(n \log n)$. Since the total number of possible restrictions of type (13) on set $\mathfrak{C}$ does not exceed $2n^2$, the total number of restrictions in sets $\mathfrak{R}_{\mathfrak{f}}$ does not exceed $4nN$, and the total size of sets $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ and $\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ does not exceed $4nN$, the total computational complexity of steps 1–2 is $O(n^3N)$.

The check on step 3 for a single node can be performed in $O(n^2)$. The total number of restrictions in sets $\mathfrak{R}_{\mathfrak{f}}$ does not exceed $4nN$, and therefore the total computational complexity of step 3 is $O(n^3N)$.

Trees on step 6 can be constructed in $O(n^2)$ operations. The total of N of such trees must be constructed. The total of $n-1$ edges will be added to tree T on step 6.1. Since each edge can be found in $O(n^2)$ operations, the total computational complexity of step 6 is $O(n^2N)$

Therefore, the total computational complexity of the presented algorithm is $O(n^3N)$. Even though the size of $\mathfrak{F}^*$ seem to be growing exponentially of n, experiments have shown that in most cases $N = |\mathfrak{F}^*| < n^2$. Therefore, in most cases the algorithm requires $O(n^3)$ memory and runs in $O(n^5)$ time.

4.6 Multiple inheritance

According to the ABIs used by GCC and MSVC compilers [1, 4], the virtual function table pointer is always the first field in a class, and every virtual function expects the passed **this** pointer to point at the beginning of an embedded instance of the class that first introduced this virtual function. This means that in case a class defines a function that must be present in several virtual tables belonging to this class, only one function body will be generated, and in all but one of the virtual tables pointers to adjuster thunks will be stored instead of a pointer to function body. These adjuster thunks will perform adjustment of the passed **this** pointer, and then pass control to the function body.

Therefore, the following statement holds.

Statement 7. *Let $A, C \in \mathfrak{C}, i \geq 0 : A_i^{func} = C_i^{func} \wedge A_i^{adj} \neq C_i^{adj}$. Then virtual tables A and C belong to two (not necessarily different) classes that use multiple inheritance and $A_i^{adj} - C_i^{adj}$ is an offset between corresponding virtual table pointer fields in these classes.*

If all bases of class SomeClass use virtual destructors, then the virtual destructor of this class must be present in all virtual tables of this class, and statement 7 can be applied. Most polymorphic class hierarchies use virtual destructors, and that's why the use of multiple inheritance can be detected in most cases by using statement 7. Since destructor of class SomeClass can be reused in one of derived classes, statement 7 cannot be used to reliably determine which virtual tables belong to the same class.

5 Experimental results

The presented techniques for inheritance hierarchy reconstruction were implemented in an application, a part of which was written in C++, and another part — in IDC, a programming language of the IDA Pro interactive disassembler. This application was tested on several open source projects, including doxygen and mkvmerge . Each of these projects was compiled twice — with and without run-time type information. The created application was used to process the resulting binaries, and results were compared with the results of source code analysis.

In case run-time type information was used, the class hierarchy was reconstructed in exactly the same form as it was present in the source code. If run-time type information was not present in the assembly, a set of virtual tables was reconstructed with some function pointers present in the assembly being mistakenly treated as virtual function tables. Using the algorithm presented in chapter 4.4, the inheritance relation was correctly reconstructed on the set of

virtual tables that contained virtual destructors. On most of the virtual tables that did not contain virtual destructors the inheritance relation was also reconstructed correctly.

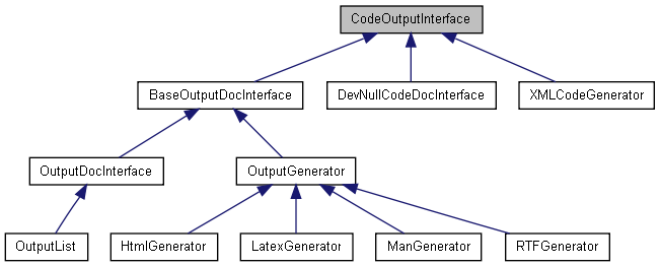

Figure 3. Fragment of class hierarchy of doxygen.

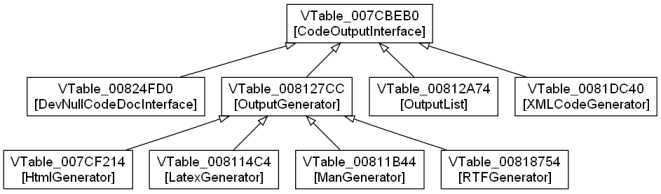

Figure 4. Fragment of reconstructed virtual table hierarchy.

Fig. 4 demonstrates the inheritance relation on the set of virtual tables reconstructed using the presented method for a fragment of the class hierarchy of doxygen, presented on Fig. 3. The virtual tables on Fig. 4 are labeled with the corresponding class names for convenience. The virtual tables for classes BaseOutputDocInterface and OutputDocInterface were not generated by the compiler, and that's why they are not present on Fig. 4. The inheritance relation between all other virtual tables is reconstructed correctly.

6 Conclusion and further work

A method for automatic reconstruction of polymorphic class hierarchies from assembly code obtained by compiling a C++ program is presented. The class hierarchy is reconstructed exactly in the same form as it was in the source code provided that the source C++ program was compiled with run-time type information (RTTI) enabled. If the RTTI is not compiled into the assembly program, our reconstruction method is based on inducing the inheritance relation from the set of virtual tables and virtual class destructors. The method is applicable if no virtual inheritance is used in the source program, and still reconstruct inheritance relation correctly in most cases if virtual destructors are not used. We have also presented a method for gathering information on virtual tables belonging to classes that use multiple inheritance.

A prototype implementation of the proposed methods use the IDA Pro interactive disassembler for extracting the virtual tables from an executable file (implemented as an IDA Pro plug-in) and a standalone tool for reconstructing and visualizing the inheritance relation.

Directions of future work include deeper analysis of class constructors to determine automatically which virtual tables belong to the same class in case of multiple inheritance.

References

[1] Itanium c++ abi, http://www.codesourcery.com/public/cxx-abi/abi.html.

[2] *ISO/IEC Standard 14882:2003. Programming languages - C++*. American National Standards Institute, New York, 2003.

[3] M. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. *Proceedings of 11th Working Conference on Reverse Engineering*, pages 27–36, November 2004.

[4] J. Gray. C++: Under the hood. 1994.

[5] M. Lerma. *Notes on Discrete Mathematics*. 2004.

[6] P. Sabanal and M. Yason. Reversing c++. *Black Hat DC*, February 2007.

[7] I. Skochinsky. Reversing microsoft visual c++ part 2: Classes, methods and rtti. 2006.

[8] B. Stroustrup. A history of c++: 1979-1991. *Proceedings of the second ACM SIGPLAN conference on History of programming languages*, pages 271–297, April 1993.