Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing

Paolo Tonella, Giuliano Antoniol, Roberto Fiutem
I.R.S.T.
via Alla Cascata
I-38050 Povo (Trento), Italy
461.314444
{tonella, antoniol, fiutem}@irst.itc.it

ABSTRACT
Large software systems are difficult to understand and maintain. Code analysis tools can provide programmers with different views of the software which may help their understanding activity. To be applicable to real programs written in modern programming languages, these tools need to efficiently handle pointers. In the case of C++ analysis, object oriented peculiarities (like, e.g., polymorphism) have to be accounted for as well.

We propose a flow insensitive, context insensitive points-to analysis capable of dealing with the features of the object oriented code. It is extremely promising because of the positive trade-off between complexity and accuracy. The integration of the points-to results with other analyses, such as reaching definitions and slicing, is also discussed in the context of our program understanding environment.

Keywords
Points-to analysis, slicing, program understanding, C++, polymorphism, flow analysis, reverse engineering.

INTRODUCTION
Many flow analyses have been applied to software engineering. For example reaching definitions analysis has been used in data flow testing [10]. Slicing has been used in maintenance, debugging, reuse, testing, reverse engineering, metrics and architectural understanding [4, 5, 6, 8, 9, 11, 12, 22]. Constant propagation has been used in user interface reengineering [2, 14].

Modern programming languages contain features like pointers and objects. Object-Oriented (OO) characteristics such as classes, inheritance, polymorphism and dynamic binding augment language expressivity and offer new possibilities to programmers. On the other hand, they represent new challenges to flow analyses.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.

ICSE 97 Boston MA USA
Copyright 1997 ACM 0-89791-914-M97/84/..$3.50

had originally been developed for languages not including such concepts.

Different flow and context sensitive algorithms have been developed to obtain accurate results on pointers analysis, data dependences and slicing, at the expense of time and memory complexity [7, 13, 15, 16]. However, when the objective is to analyze large industrial software systems, efficiency becomes crucial. The importance of controlling the time complexity has been investigated in [3], where a solution based on a fine grained context sensitivity specification is proposed. In [20], we have presented a variable precision reaching definitions analysis with similar objectives of scalability and control. To make pointer analysis tractable for large software systems, a flow and context insensitive Points-To Analysis (PTA) has been recently presented in [17, 18] for the C language. It has an almost linear time worst case complexity and is also very fast in practice.

The purpose of the present work is to extend the mentioned flow insensitive points-to algorithms to handle C++ peculiarities (class attributes, polymorphism, templates, dynamic binding). Flow insensitive points-to results have been compared with those obtained with a flow sensitive technique [15], to assess the practical accuracy of the proposed extensions.

The results of the OO PTA are integrated in the IRST program understanding environment which contains different flow analyses for program and architectural understanding. Some examples of program slicing based on flow insensitive points-to analysis are presented.

The paper is organized as follows: next section is an overview of the flow insensitive points-to analysis of C code. It is followed by the description of the proposed flow insensitive points-to algorithm for C++. Then our results are compared with those obtained through a flow and context sensitive analysis on a test suite. The following section discusses the experimental results. Finally the integration of the PTA with reaching definitions and slicing is presented. The last section presents conclusions and future work.
FLOW INSENSITIVE POINTS-TO ANALYSIS OF C CODE: AN OVERVIEW

The PTA purpose is to determine the points-to set of every location, i.e., the set of locations which may be pointed to by a given location (by location we mean stack and heap allocated objects, data members and variables). PTA can be performed adopting the non standard set of types defined in [17, 18], and denoted as \( \tau \)-types.

\( \tau \)-types are not related to concrete types: they serve to model how storage is used in a program. As types may be recursive, type variables are introduced to describe mutually referencing objects. In the following type variables are denoted as \( \tau_i \) where \( i \) is an integer identifying the type. Each location may contain actual data (i.e., it does not reference other locations), such in a case its type is \( \tau_i = \text{ref}(\bot) \), or may contain a location address, this is indicated as \( \tau_i = \text{ref}(\top) \).

Once the PTA has been performed, each location is associated with a type variable, say \( \tau_i = \text{ref}(\tau_j) \) or \( \tau_i = \text{ref}(\bot) \). \( \tau_j \), referenced by \( \tau_i \), models the points-to relation in that all locations of type \( \tau_j \) are possibly pointed to by the locations of type \( \tau_i \). The points-to set of a location \( x \) is thus:

\[
\text{points-to}(x) = \{ y \mid y : \tau_j \}
\]

The points-to set of a location with \( \tau_i = \text{ref}(\bot) \) is obviously empty. The points-to relation between locations is conveniently represented by the storage shape graph (SSG) [17], where each node is a \( \tau \)-type (and is labelled with all locations of that type), and a directed edge exists between \( \tau_i \) and \( \tau_j \) if \( \tau_i = \text{ref}(\tau_j) \). Initially all locations are associated with different type variables, \( \tau_i \), each initialized as \( \tau_i = \text{ref}(\bot) \). Then the instructions of the program are analyzed in an arbitrary order (the analysis is both flow and context insensitive), and each time a pointer affecting statement is encountered, types are merged and referenced types are updated by means of the join() or c-join() functions. When different locations are pointed to by the same pointer, their types have to be merged. The join() function is used to unify the types to one single type, and is recursively applied to the referenced types. When the pointed to locations have not yet been encountered, the pointer is a reference to bottom, and type merging is delayed. The c-join() function inserts the type into a pending list, and a real unification is performed only if a successive PTA step transforms the reference to bottom into a reference to a non bottom type [17].

The points-to algorithm infers a typing environment under which the program is well-typed, i.e., the SSG dedicated by the types is a safe (conservative) description of all possible dynamic storage configurations [17]. Well-typedness can be expressed in terms of typing constraints [17], the satisfaction of which is obtained through the application of join() or c-join() as described above.

FLOW INSENSITIVE POINTS-TO ANALYSIS OF C++ CODE

A flow insensitive PTA method, which covers the peculiarities of modern programming languages, is presented. It is an extension of Steensgaard's work on C [17, 18]. Although the approach is not limited to a particular OO language, to illustrate results on real software developed with a widely used programming language, this paper is focused on C++. All OO features are considered: pointers to objects, dynamic object allocation, single and multiple inheritance, recursive data structures, recursive methods, virtual functions, dynamic binding and pointers to methods.

Since both flow insensitivity and context insensitivity are preserved in our extensions, the language control structures and functions (or methods) calling sequences do not affect the analysis and therefore are not taken into consideration.

\[
A ::= (\text{type-id}) A \\
* A \\
A[\text{EXPR}] \\
A.\text{id} \\
A->\text{id} \\
\text{id} \\
(A) \\
\text{new CLAS}
\]

Figure 1: Grammar of the access paths. Dereference operators are: *, -> and [EXPR].

The approach presented here integrates the notion of access paths [15] and that of non-standard set of \( \tau \)-types [17] with a new type inference scheme, which has been defined first to take into account C++ object members under the access path framework and second to be compatible with the low cost PTA presented in [17]. To deal with polymorphism, we integrate the \( \tau \)-types with the standard type information (denoted as concrete types), which can be determined by syntactic analysis of C++ code.

An access path is a unary expression [19] described by the grammar of Figure 1. It contains an arbitrary sequence of dereferences and accesses to data members. Dereferences and member accesses may be combined through the \( \to \) operator. Array components accesses
are considered as a particular kind of dereference, since they can be obtained through normal dereferences as well. An access path represents a fixed location if it does not contain dereference operators (*, -> and [EXPR]) [15]. Otherwise it represents a variable location. Variable locations, depending on the possible values of the involved pointers, may be associated with different fixed locations during the execution. A dynamic allocation is represented as a fixed location object heap_n created for each line, n, where a new operator is encountered (i.e., heap identification by memory allocation sites). Array components are fixed location objects denoted as id_comp (while a variable location leading to an array component has the form id[EXPR]), representing a generic component of the array. A fixed location can therefore be expressed by the following regular expression:

\[(id([ ])\ast \ast \ast \ast \ast )\] (1)

where star (\ast) stands for zero or more occurrences, while dot (\.) and square brackets ([ ] ) are not metacharacters.

### INSTRUCTIONS

**FIXED LOCATIONS**

<table>
<thead>
<tr>
<th>INSTRUCTIONS</th>
<th>FIXED LOCATIONS</th>
<th>VARIABLE LOCATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>class A {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x x;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y y;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a;</td>
<td>(\Rightarrow)</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a.x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a.y</td>
<td></td>
</tr>
<tr>
<td>n: *p = new A;</td>
<td>(\Rightarrow)</td>
<td>heap_n *p</td>
</tr>
<tr>
<td></td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>b[10] = 1;</td>
<td>(\Rightarrow)</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>b[10]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Examples of fixed and variable locations

In fig. 2 the declaration of an object of class A generates one fixed location for the object, plus as many fixed locations as there are data members in the class. The dynamic allocation of an object at generic line n involves the fixed location identified by the site of the dynamic allocation and the fixed location of the pointer p. While the pointer itself is a fixed location, its content, *p, is a variable location. Also an array assignment involves two fixed locations, namely the array name, b, and the generic array element, b[10], accessed through the variable location b[10].

It can be shown [15] that every access path which includes \(n > 0\) dereferences is aliased to at least one fixed location object. Hence it is possible to statically associate every variable location with the set of fixed locations that may be dynamically accessed through it.

The extensions to C++ involve the computation of appropriate \(\tau\)-types associated with access paths. The \(\tau\)-type of an access path is represented as an attribute \((A.\text{type})\), which can be determined using the syntax directed definitions of Figure 3. Explicit and implicit pointer dereferencing and array element accesses perform a deref operation on the \(\tau\)-type attribute. The deref operation simply returns the referenced type of a given type, i.e., if \(\tau_1 = \text{ref}(\tau_2)\), deref(\(\tau_1\)) returns \(\tau_2\). Creating (with the new operator) and assigning a dynamic object to an access path, produce the construction of the relation (ref) between the access path \(\tau\)-type and the heap \(\tau\)-type corresponding to the allocation site. All the other semantic rules propagate the \(\tau\)-types of member attributes, and will be discussed in detail in the next section.

PTA consists of examining in turn all the instructions of the system under analysis and for each assignment instruction which affects some pointers a \(\tau\)-type update action is taken. Two basic kinds of instructions involving access paths may be encountered during the PTA: assignments without or with the use of the & operator, which takes the address of a location. The corresponding update actions, are defined in Figure 4.

### Figure 3: Syntax directed definitions to compute the type attribute of an access path. The type of a member attribute of a type is denoted as: \(\text{type.id.type}\).

Instruction (1) of Figure 4 is an assignment between two access paths. Types referenced by left and right hand sides are merged according to the c-join (conditional join) procedure described in [17]. In the second kind of assignment the left hand side referenced type is merged into the right hand side type according to the join procedure.

### Object Data Members Extension

Each type variable has a (possibly empty) set of member attributes, used to model data members of class objects
The constraint is applied to the type referenced by variable \( p \), i.e., \( \tau_1 \), whose \( x \) member is made to point to \( \tau_2 \). When the last statement is analyzed (step 4), variable \( c \) will inherit the typing constraint of the member attribute \( x \), which is associated with its new type \( \tau_1 \).

**Effects of interprocedurality on pointers**

Since the proposed PTA is context insensitive, only the points-to relations involving the passed parameters have to be taken into consideration.

Parameters are handled by applying the rules of Figure 4, where each actual parameter of a method (or a function) invocation is assigned to the corresponding formal parameter of the called method.

A method may induce a typing constraint on data members of its class. As the constraint holds for every object, it is applied to a generic object \( \text{any} \) of that class. The last step of the points-to analysis propagates the \( \text{any} \) constraints to all the fixed locations of that class.

```c++
class A {
    public:
        B* p;
    }

    f(B* q) { p = q; }
    };

    A a, b;
    B c, d;
    a.f(s);  // Accesses member attribute id.
    b.f(s);  // Accesses member attribute id.

    ... any.p: \( \tau_1 = \text{ref}(\tau_2) \)
    q: \( \tau_2 = \text{ref}(\tau_2) \)
    c: \( \tau_2 = \text{ref}(\tau_2) \)
    a: \( \tau_1 = \text{ref}(\tau_1) \)
    b: \( \tau_1 = \text{ref}(\tau_1) \)
    \( \text{any}.p: \tau_1 = \text{ref}(\tau_2) \)
    \( \text{any}.p: \tau_1 = \text{ref}(\tau_2) \)
    
    Figure 5: Example including the declaration of class A, objects a, c and four statements. The resulting typing environment after steps 3 and 4 is shown.

    Figure 6: Example comprising the declaration of class A, the declaration of objects a, b, and two method invocations. The resulting typing environment is shown.
```

An alternative choice in C++ is to consider a data member access as an access through the \text{this} pointer [10], and to consequently apply member constraints to the type referenced by \text{this}. With this alternative choice all the objects of a given class are forced to have the same type, because their address is assigned to \text{this}, yielding a lower accuracy of the points-to results. This overly restrictive constraint is avoided by using the \( \text{any} \) fictitious object. Actually, in Figure 6, where the two
method invocations modify the points-to set of the data member p, after the application of the _any constraints, a and b are not forced to be of the same type.

A ::= (type-id) A_1[A .name = A_1.name]
    * A_1    {A .name =
                  points-to(A_1.name)}
A_1[EXPR] {A .name =
              points-to(A_1.name)}
A_1.id {A .name = A_1.name.id}
A_1->id {A .name =
          points-to(A_1.name).id}
id {A .name = {id}}
(A_1) {A .name = A_1.name}

Figure 7: Syntax directed definitions to compute the name attribute of an access path, which is the set of fixed locations reachable through the access path.

Inheritance and Polymorphism

A derived class is defined through inheritance, a mechanism which allows the derived class to access data members and use methods from its parent class, as if they were of its own. New fixed locations are available for objects of the derived class, if data members are added. Redefinition of methods is also possible, and needs special treatment only when virtual functions are involved, and therefore the calls become polymorphic. Points-to results can be used to analyze polymorphism, i.e., to associate polymorphic function calls with the set of invocable methods. The following structures are required:

- A symbol table, associating each fixed location with its concrete type.

- The points-to sets, associating each fixed location x to the possibly referenced locations (points-to(x)).

Given a polymorphic method invocation of the form:

ap->f(a1,...,an);(*ap).f(a1,...,an);

where *ap, a1,...,an are access paths, the syntax directed definitions of Figure 7 are used to compute the name attribute, i.e., set of fixed locations which are possibly referenced through a particular access path. Each fixed location has a concrete type representing the class of such a location and the combination of all the possible concrete types of the possibly referenced fixed locations determines the set of invocable methods (the polymorphic call set).

1 Shape *p;
2 Circle a;
4 if (c)
5    p = new Rectangle();
6 else
7    p = &a;
8    p->plot();

Fixed locations (and corresponding concrete type):
a: Circle
heap_5: Rectangle
p: Shape*

Points-to set of p (after processing statements 4,..., 8): points-to(p) = {heap_5, a}

Virtual calls resolution:
p->plot()  ==>  points-to(p).plot()
           ==>  heap_5.plot(), a.plot()
           ==>  Rectangle::plot(), Circle::plot()

Figure 8: Example including a polymorphic call (line 8). Its resolution, based on concrete types of fixed locations and points to sets, is shown at the bottom.

The example of Figure 8 includes a polymorphic call at line 8. Concrete types and points to sets are shown, as they are required to perform the steps necessary for polymorphism resolution. To solve the virtual call (p->plot()), which is equivalent to (*p).plot(), the name of the access path *p is computed according to the second definition of Figure 7, resulting in {heap_5, a}. The call is therefore either heap_5.plot() or a.plot(), and the concrete types of the involved fixed locations are Rectangle and Circle respectively, resulting in the invocable methods Rectangle::plot(), Circle::plot().

Algorithm Properties

The proposed algorithm has several interesting properties. It is safe: the points-to sets and the polymorphic call sets are conservative approximations. This means that if a points-to relation holds, it is surely included in the points-to set reported by the approximate analysis, and if a method is invocable from a polymorphic call, it is surely reported in the approximate polymorphic call set.

As the algorithm is both flow and context insensitive, it does not require any knowledge about the call graph, while other flow and context sensitive approaches require the construction of the call graph, or its abstraction, during the analysis. However, in flow sensitive
analyses and in presence of polymorphism the call graph depends on the results of the ongoing analysis. Therefore while flow sensitive algorithms iteratively use the available points-to information for call graph construction and vice versa, the proposed approach is able to completely separate the points-to analysis from polymorphism resolution and no call graph construction is required.

Recursive data structures and recursive function calls are inherently handled. The SSG may contain cycles, hence it is not necessary to introduce $k$-limiting techniques to represent self-referential data structures. Recursive data structures are properly modeled by type variables, the member attributes of which reference the type variable itself. As the analysis is context insensitive, the call graph is not taken into consideration, i.e., it is the presence of cycles in it. Therefore methods may be recursive or indirectly recursive.

Being flow and context insensitive, the algorithm has a low cost in terms of both space and time complexity. The worst case complexity is theoretically exponential with the program size $N$, as it is possible to build a program with $O(2^N)$ distinguishable locations [18]. Even if theoretically correct, it is practically meaningless to express the complexity in terms of $N$. A more convenient metric is the count $S$ of all objects and variables in the program. If all classes have $R$ or fewer data members, the time complexity results $O(RS\alpha(S)), \alpha$ is the inverse Ackerman’s function, i.e., the algorithm has a quadratic worst-case running time complexity [18]. The term $R$ is the maximum number of data members of a class, and thus depends on the depth of inheritance and on the use of multiple inheritance. Common programming practices indicate that both of these factors are limited, and do not linearly increase with program size. Consequently $R$ turns out to be a constant and the almost linear complexity $O(S\alpha(S))$ presented in [17] is preserved in practice.

EXPERIMENTAL SETUP AND RESULTS

The framework for the development of the Points-To Analysis Tool (PTAT) is the IRST program understanding environment. It comprises the FLOW ANALYSIS Tool (FLANT), the Architectural Recovery Tool (ART), which uses flow analysis results, and the PTAT.

FLANT provides many flow sensitive interprocedural analyses among which reaching definitions, reachable uses, slicing and impact. It handles recursive function calls and allows variable precision for reaching definitions [20]. Its output is textual, graphical (when results can be represented as graphs) or interactively provided to the user through a customized version of the text editor emacs. ART [9] is a reverse engineering tool supporting the discovery of architectural information.

It reports information about architectural components and connectors found in the source code according to different views (system, module, task and code).

The IRST program understanding environment is multi language and modular. Information exchange among modules is based on an intermediate representation for both inputs and outputs. PTAT is crucial in the environment, because FLANT analyses depend on the points-to results, and FLANT results are used by ART. The contribution of PTAT in the analysis framework is in its capability to solve polymorphic calls, and to provide a points-to set for each fixed location.

Points-to sets are used by FLANT to decide which variables are possibly defined or used in a statement. This is essential for some of the FLANT analyses, such as data dependences analysis, where taking pointers into considerations is fundamental.

PTAT is written in C++ and uses the LEDA library. Its input is an intermediate representation of the source code produced by the front-end, and its output can be textual or graphical. The latter results in the visualization of the SSG.

The experiments performed to test the PTAT were aimed at measuring the number of extra points-to relations, possibly, introduced by the context insensitive approximation, with respect to the results from flow sensitive C++ pointer and polymorphism analysis. For this purpose the public domain test suite collected by Pande [15] was used. The test suite consists of 19 programs containing virtual calls, for which polymorphism resolution is required. For each polymorphic call our PTA provides a conservative polymorphic call set.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Methods</th>
<th>Virtual Methods</th>
<th>Virtual Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. greed</td>
<td>968</td>
<td>47</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>2. garage</td>
<td>149</td>
<td>19</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3. vcircle</td>
<td>142</td>
<td>16</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4. office</td>
<td>213</td>
<td>12</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5. family</td>
<td>199</td>
<td>22</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6. FSM</td>
<td>98</td>
<td>15</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7. deriv2</td>
<td>313</td>
<td>34</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>8. ship0</td>
<td>257</td>
<td>24</td>
<td>12</td>
<td>22</td>
</tr>
<tr>
<td>9. deriv1</td>
<td>192</td>
<td>31</td>
<td>13</td>
<td>28</td>
</tr>
<tr>
<td>10. objects</td>
<td>465</td>
<td>50</td>
<td>31</td>
<td>39</td>
</tr>
<tr>
<td>11. aimal</td>
<td>336</td>
<td>54</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>12. primes</td>
<td>46</td>
<td>11</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>13. ocean</td>
<td>444</td>
<td>64</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>14. NP</td>
<td>31</td>
<td>7</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>15. city</td>
<td>519</td>
<td>67</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>16. trees</td>
<td>217</td>
<td>26</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>17. employ</td>
<td>804</td>
<td>58</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>18. life</td>
<td>178</td>
<td>21</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>19. chess</td>
<td>392</td>
<td>43</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Some characteristics of the test suite by Pande.

\(^1\)LEDA is an efficient general purpose class library developed at the Max Planck Institut für Informatik, Saarbrücken, Germany
DISCUSSION OF RESULTS

The last column contains CPU times in milliseconds for the flow insensitive PTA. This program computes a function derivative with respect to a variable. The function is given as a combination of operators (+, -, *, /) and operands (constants and variables). As the functions, used as test case, are hard coded in the main, and they do not exploit the full range of operators, some classes are never instantiated, giving rise to unreachable methods. These methods are discarded by the context sensitive analysis, while their effect is taken into consideration by the flow insensitive algorithm. Flow and context insensitivity has some effect on three other programs (numbered 8, 9 and 13 in Table 2), and does not affect at all the remaining 15 test cases.

An explanation for the high precision obtained could be that it is common programming practice to not reuse pointer variables for different purposes, according to different control flows or calling contexts: a code where a pointer variable has a unique meaning (possibly suggested by the name chosen for it) is definitely more readable and maintainable. A consequence is that a flow and context insensitive points-to analysis exhibits an accuracy similar to that of the flow sensitive counterpart, since the results scarcely depend on flow and context.
On the other hand, because of flow insensitivity, the proposed algorithm has low time complexity. The trade-off between accuracy and performance is in our opinion satisfactory, at least on the test suite under investigation. This becomes important when industrial size software is taken into consideration for analysis, and scalability has to be taken into account.

**SLICING C++ CODE**

Slicing is a technique originally developed by [21] and subsequently investigated by [11], [1]. The extensions of the system dependence graph to the OO software, described in [13], make it possible to slice OO code. The importance of efficiently handling pointers and polymorphism is also stressed, as a prerequisite in a code analysis framework for the C++ language. PTA is also helpful in improving the accuracy of data dependences analysis used in slicing.

In this section two examples of slicing produced using C++ flow insensitive PTA are presented. The first is taken from [13] and serves the purpose to show that, in the presented example, flow insensitive points-to results lead to a slice which is identical to the reported one. The second example serves the purpose of further illustrating slicing with the use of flow insensitive PTA on a public domain C++ code.

PTA provides the set of invocable methods for each polymorphic call statement and also the points-to set of every fixed location. All this information are used by IRST data dependences analyzer. The set of invocable methods is used to determine the possible call-graph thus allowing inter-procedural analysis. The points-to set of every fixed location is essential to determine the DEF and USE sets of every instruction, which are used during reaching definitions computation. The DEF (USE) set of a statement is the set of fixed locations which are defined (used) by such a statement. As a statement uses access paths to reference the locations that are being defined (used), it is necessary to transform an access path to the set of fixed locations reachable through it. For this purpose the syntax directed definitions of Figure 7 and Figure 10 are respectively used.

When corresponding to variable locations, the fixed locations inserted in the DEF set are marked as non-killing definitions, i.e., definitions which do not override previous definitions, but are simply added during the analysis. The reason for their non-killing nature is that the points-to relation used to obtain them is a possible points-to, and not a definite points-to. A discussion about the impact of definite points-to and possible points-to on the kill set can be found in [7].

Figure 9 shows an example of DEF set computation for an assignment statement where the left hand side accesses a data member through a pointer.

The USE set computation requires different syntax directed definitions, since the USE set can be considered as the collection of the referenced fixed locations together with the fixed locations used in the dereference chain. For this reason, in Figure 10 the union between referenced and referencing fixed locations is computed, at every production that includes a dereference operator.

Figure 11 describes an example of USE set computation for a statement comprising the access to a data member through a pointer.

Once the DEF and USE sets have been determined for each instruction, reaching definitions and data dependences (use-definition chains) can be computed. Data dependences are subsequently used by the slicer. A detailed description of the analyses performed by FLANT can be found in [20]. Data dependences determination,
class A {
    int x;
    void f(int i);
}

void A::f(int i) {
    x = i;
}

main() {
    A a;
    a.f(1);
}

void A::f(int i, int& x) {
    x = i;
}

main() {
    A a;
    a.f(1, a.x);
}

Figure 12: Example of reaching definitions computation. The necessary signature extension is shown under the horizontal line, along with the resulting reaching definitions.

based on PTA results, is safe, as a consequence of the safety of the points-to and reaching definitions analyses: if the points-to set is larger than needed, so is the resulting data dependencies set, but every valid data dependence is surely reported.

A method may induce some data dependences on the data members of its class. To map these dependences to the data members of the actual object which invokes the method, the method signature is extended and every class attribute can be considered in our approach as a parameter passed by reference. The object invoking the method passes by reference its own data members, and correspondingly receives back on them the effects of the call.

In the example of Figure 12, reaching definitions are indicated as a set of pairs <fixed location, definition identifier>. Note that, for sake of simplicity, the definitions identifier has been confused with the line number in which it appears. The signature of method f has been extended with the passed by reference parameter x to propagate the dependences to the calling object data member (a.x).

This approach is similar to that proposed in [13], where global variables, instead of parameters passed by reference, are used.

FLANT code slicer uses data dependences, control dependences and invocation graph to provide the user with a slice of the program on a slicing criterion <p, x>, where p is a statement and x is a variable of interest in the program. The result is the set of statements which directly or indirectly contribute to the value of x at the statement p, and is computed as the transitive closure of data dependences and control dependences from p on x. In object oriented programming, variable references are often substituted by method calls (simply returning a value), so that an extension of the slicing criterion is required: x may be both a variable or a method [13]. If it is a variable, it must be defined or used at statement p, and if it is a method it must be invoked at p.

If data and control dependences are forward (instead of backward) traversed, the result is a forward slice on the given slicing criterion <p, x>. Forward slices include all instructions directly or indirectly affected by the value of x at p (the impact set). Once data and control dependences are available for a program, forward slices are computable as well.

Figure 13 presents an example of C++ code taken from [13]. The points-to analysis performed by PTAT gives the following results on variable e_ptr (they hold all over the program, since the analysis is flow insensitive):

points-to(e_ptr) = {heap_36, heap_37}

and consequently the polymorphic call of line 38 is associated with the following invocable methods:

e_ptr->go() => points-to(e_ptr).go()
    ==> heap_36.go(), heap_37.go()
    ==> AlarmElevator::go(), Elevator::go()

On the following criteria (taken from [13]), FLANT produced the following slices:

slice<39, which_floor> = {3,4,5,12,16,17,18, 19,20,22,25,26,32,33,35,36,37,38}
slice<20, current_floor> = {3,4,5,12,16,19, 20,22,25,26,32,33,35,36,37,38}

FLANT slices are identical to those presented in [13]. Again, on this example, flow insensitive points-to produced results identical to flow sensitive ones and therefore the slices turn out to be the same. Figure 13 shows an example of an interactive session with FLANT. Slicing criteria can be introduced by clicking on a
variable name at the desired program point (<20, current_floor> in this case). FLANT computes the slice and visualizes it in reverse (or in color when possible) on the screen. Line numbers in Figure 13 are the same as in [13], so that they can be easily compared.

FLANT slicer has been applied to a second example, HMM, previously analyzed with PTAT. HMM is a C++ public domain package, about 800 LOC (excluding comments and empty lines), developed by R. Myers and J. Whitson at the University of California, Irvine, implementing a recognizer based on Hidden Markov Models.

Figure 14 shows a portion of the SSG of the program as computed by PTAT. Heap locations are identified by a suffix which comprises a file identifier and the line number (e.g., heap_hmm.cc_502 corresponds to the allocation at line 502 of file hmm.cc). The points-to information was used to determine the DEF and USE sets of each instruction in the program, and subsequently to compute reaching definitions and data dependences. Once data dependences were determined, it was possible to interactively query FLANT for slices. As an example in table 3 three slices are summarized. The third one on heap_hmm.cc_506 is actually on a generic component of the array scaling_factors, which is a relevant HMM package data structure.

CONCLUSION AND FUTURE WORK
We have presented a flow insensitive context insensitive points-to algorithm for C++ code which is an extension of the corresponding one for C-language described in [17, 18].

A comparison of the flow insensitive points-to results with flow sensitive ones on a C++ test suite reported in the literature shows that accuracy is high.

Under some assumptions on data members and depth of inheritance limited growth with program size, the presented extensions for C++ do not affect the original almost linear running time complexity, so the proposed approach should be easily scalable to large industrial size software.

Table 3: Slices of HMM. Line numbers in the slicing criterions refer to file hmm.cc. The absolute and relative (to the program size) extracted lines of code are given.
We have also described the integration of the points-to analysis with data dependences and slicing. Some examples have been presented and discussed.

Our future work will be devoted to further evaluating the trade-off between accuracy and performance of the flow insensitive points-to analysis with respect to the flow sensitive one on a larger industrial benchmark and in the context of slicing.

ACKNOWLEDGMENT
We are grateful to Laurie Hendren for the interesting discussions and to Hemant Pande for providing the C++ test suite.

REFERENCES