Class-Modular, Class-Escape and Points-to Analysis for Object-Oriented Languages

Alexander Herz and Kalmer Apinis

Lehrstuhl für Informatik II, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
{herz,apinis}@in.tum.de

Abstract. We present a combined class-modular points-to and class-escape analysis that allows to analyze class declarations even if no information about the code that invokes the class's methods is available as is the case for e.g. shared libraries. Any standard whole-program or summary-based points-to analysis can be plugged into our framework and thus be transformed into a class-modular, class-escape and points-to analysis. The analysis framework uses the flow restrictions imposed by the access modifiers (e.g. *private*, *public* and *protected* in Java) to find all fields that may be modified by code that is not part of the class declaration. These fields escape the class. Unlike method-based summaries instantiated with an unknown context, our analysis framework can give detailed points-to information for non-escaping fields. In addition, the knowledge of which fields belong to the region that does not escape a class can be exploited to perform other analysis like class-modular object in-lining [6] more efficiently or enable the automatic inference of class invariants [10]. We prove the soundness of the analysis and present a set of benchmarks showing that the analysis is suitable to analyze real world code and that more than 75% of the fields from the benchmarked classes are identified as non-escaping.

1 Introduction

Accessibility of an object from different program parts is important information for optimizing compilers and verification tools. This information can be inferred using may points-to analyses. Often, not all code that uses a class declaration is available to analysis because program modules are compiled independently and linked dynamically at run time. In order to apply optimizations or verification in this scenario, points-to information for a class must be inferred in isolation from the code that uses the class. Such class-modular points-to information cannot be obtained from existing whole-program points-to analyses as these expect the complete program as input. In contrast to whole-program analyses, modular analyses can abstract different program parts independently. Commonly, individual methods are abstracted without calling-context. Later, these method-summaries are instantiated with calling-context information, so that eventually the context information and the summaries of the whole program are

A. Goodloe and S. Person (Eds.): NFM 2012, LNCS 7226, pp. 106–119, 2012.

combined. Instantiating method-summaries with unknown context information (e.g. for class methods without the code that calls the method) yields imprecise results for the *this*-pointer and all other method parameters. Therefore, it is not enough to use method-summary based analyses.

To solve this problem, we present a framework that transforms a common whole-program or summary-based points-to analysis into a class-modular points-to analysis. Given a sound plug-in analysis, the transformed analysis is sound and may be useful even if the whole program is available. The analysis time can be reduced by analyzing class declarations independently or in parallel, mostly without loosing much precision.

As a side-effect, the transformed analysis produces class-escape information. Escape analysis as presented by e.g. Blanchet [2] determines which local objects escape from a method as only local objects that do not escape can be considered truly local to that method and may be stack allocated. In contrast, our framework extends the scope from methods to classes. Local variables, *private* fields and locally used heap objects are considered class-local if and only if they can never become accessible from outside the class. If a local variable, a *private* field or a locally used heap object can possibly become accessible from outside the class (e.g. a pointer to it escapes through one of the *public* class methods) then it is considered *class-escaped*. This class-escape information can be used to improve other analysis (e.g. object in-lining), which depend on object accessibility information but commonly rely on a whole-program or summary-based analysis.

We have implemented an instance of the analysis in the Goblint [20] framework showing that it can handle large classes from industrial and open source C++ code in seconds. Our contributions in this paper are

- we present the combined class-modular, class-escape and points-to analysis based on encapsulation that is fully independent from the code that uses the analyzed class,
- we present a framework that allows to transform common points-to analyses into a class-modular, class-escape and points-to analysis,
- we prove the soundness of the transformed analysis,
- we present an implementation and a set of benchmarks applying an instance of the analysis to large, real world code in seconds.

Related Work. Many of the points-to analyses are based on work from Steensgaard [17] and Andersen [1] which are neither class-modular nor deal with class-escape information. Abstract interpretation based modular analyses in general is described by Cousot and Cousot [5], where program modules can be analyzed independently but a completely unknown (worst case) context is assumed and access modifiers are not taken into account.

Rountev [13], Cheng and Hwu [4], Horwitz and Shapiro [8] present pointer analyses that are modular on the function level but require additional information from the function's calling context. Whaley and Rinard [21] present a compositional pointer and (method-)escape analysis. They need information from the analyzed methods' calling context as their analysis is not on the class-level.

The precision of non-escaped objects cannot be as good if the class methods are analyzed separately, neglecting the object state information available through the access modifiers. Rountev and Ryder [14] present a similar approach, assuming worst case information on the function level.

The partitioning of fields into escaping and non-escaping fields performed by our analysis relates our analysis to region analysis where mutually unreachable heap regions are identified. The non-escaping field set represents a heap region that is unreachable from outside the class declaration and the resulting points-to information directly represents which of these fields can reach outside the region. Region analysis for C programs was recently investigated by Seidl and Vojdani [16]. These region analyses are not class-modular and often the pointer information is undirected and less precise.

Boyapati, Liskov, and Shrira [3] have applied ownership type-systems to verify encapsulation and alias protection properties of object-oriented programs. These type-systems heavily rely on annotations and restrict the programming language they can be applied to, as e.g. iterators are not easily incorporated. The ownership property verified in these systems is more restrictive than our class-escape property.

To the best of our knowledge, this is the first presentation of a class-modular, points-to and class-escape analysis. Class-level modular static analysis of classes and class methods that automatically infer class invariants have been proposed by Logozzo[10]. For the analysis to be sound it is required that accessibility of object internal state (to code that is outside the analyzed class) is detected by another static analysis. The suggested whole-program escape analysis from Blanchet[2] uses the different notion of method-escaping rather than class-escaping and cannot be applied when only the class declaration is given. As such, it does not provide the accessibility information required for Logozzo's analysis. Instead, the class-escape information provided by our analysis can be used.

Porat et al. [12] present a mutability analysis for Java that can handle missing class definitions and utilizes access modifiers. They give no details on their state accessibility analysis. It is not clear whether their state accessibility analysis is sound nor how it works.

Based on our analysis class-modular object in-lining[6] optimizations for garbage collected languages can be implemented. The life-time of fields which do not escape a class is limited to the life-time of the enclosing object. Since non-escaping fields are guaranteed to be non-accessible from outside a class it is sufficient to modify the code of the class itself to in-line an object, so method cloning is not required and the optimization is modular. JIT compilers could benefit from similar improvements [22].

Structure. First, we present a working example in Section 2. After describing the abstract semantics of our analysis in Section 3, we present our implementation of the analysis and benchmarks in Section 4. In Section 5 we summarize our findings. In the corresponding technical report [7] we define the language our analysis operates on and proof the soundness of the analysis.

2 Example

In this Section a C++ example is shown where pointer assignments in one method of a class have a non-local effect which is visible in another method and how the analysis handles this information.

The *private* fields of a class can become accessible from external code if a pointer to such a field escapes the class, for example as a return value as shown in line 27 in the example in Fig.1.

In the constructor of *Rect*, an instance of *Point* which we denote as Pt_b for convenience, is assigned to *lr*. Then the address of *lr* is assigned to *e* and *e*'s address is assigned to *p* in turn. This is denoted as edges leading from *p* to *e* to *lr* and finally to Pt_b in Fig.2, where an edge from node a to node b denotes that b is in the points-to set of a.

Within DoEscape from line 22 to line 26 Pt_b is escaped through various routes as noted in the comments of Fig.1. Especially interesting is line 23, here the pointer *pr* may be equal to the current instance *this* or another instance of the class. So Pt_b may be assigned to a field from *this* if *pr* equals *this*. Otherwise, Pt_b escapes because it is assigned to an external variable. In line 27 *e* is returned, so the content of *e* and everything reachable thereof may escape. Generally, an object escapes when its address may be stored in an externally accessible object.

Analysis Overview. In order to collect all escaped pointers in the points-to set of the variable *a_ext*, our analysis proceeds in three steps. First an instance *a_this* of the class is created and all public fields of the class are considered escaped. The created instance serves as representative object for this class. Then the effect of calling *any* constructor is over-approximated on the representative object *a_this*. Finally, all possible combinations of *public* method calls on *a_this* are simulated.

When applied to our example class *Rect*, the first step produces points-to information telling us that *a_ext* may point to itself, *a_this*, or *pub* — these are considered class-escaped. In the next step we need to apply the effect of any constructor to our abstract state. As our example only has one constructor, only the effect of that constructor is applied. Finally, we simulate all possible *public* method calls on *Rect*. Class *Rect* only has one (*public*) method DoEscape, but this method could be applied several times on the same object (while changing the escaped objects between each call). Therefore, the effect of $a\_ext = a\_this \rightarrow$ DoEscape(*a_ext*, *a_ext*) is computed until the smallest fix-point for *a_ext* and the fields of *a_this* is reached. In the first iteration, Pt_b class-escapes on line 22, 23 and 25, because a pointer to Pt_b is assigned to a potentially externally accessible variable. In line 24, Pt_b class-escapes because it is given as an argument to an unknown function. Eventually, *lr* and Pt_b class-escape in line 27, because they are returned. In the second iteration, Pt_b also escapes in line 26 because it is assigned to *lr*, which has escaped in the previous iteration. This last step reaches the fix-point and the result is shown in Fig. 2. At the end of each iteration step, all escaped pointers are modified so that any escaped object may point to any other escaped object.

```
1  class Point{ public: int x,y; };
2  extern void unknown(Rect* pr);
3  class Rect
4  {
5  private:  Point *ul,*lr;
6            Point **e,**l;
7            Point ***p;
8            Point *priv;
9  public:   Point *pub;//escapes
10
11 Rect(int x1,int y1, int x2, int y2)
12 {
13   ul=new Point();//Pt_a
14   lr=new Point();//Pt_b
15   p=&e;e=&lr;l=&ul;
16   ul->x=x1;ul->y=y1;
17   lr->x=x2;lr->y=y2;
18 }
19
20 Point** DoEscape(Point**v,Rect* pr)
21 { //pt_b escapes in the following:
22   pub=lr;//copied into public var
23   pr->priv=lr;//maybe copied into
         other instance
24   unknown(lr);//passed to unknown fun
25   *v=*e;//copied into external var
26   **p=lr;//copied into lr, becomes
         external in next line
27   return e; //lr, Pt_b escape (
         returned from public fun)
28 }
29 };
```

Fig. 1. C++ Example Code. Sound points-to information for class *Rect* in the absence of the code that uses class *Rect* is generated. The field e is assigned the address of lr in line 15, so when the content of e is escaped in line 27 then lr and the instance of Point which lr is pointing to are escaped. An object escapes when its address is stored in an externally accessible object. The points-to information of fields from *Rect* is a global property, points-to relations set up in one method are retained when another method is called.

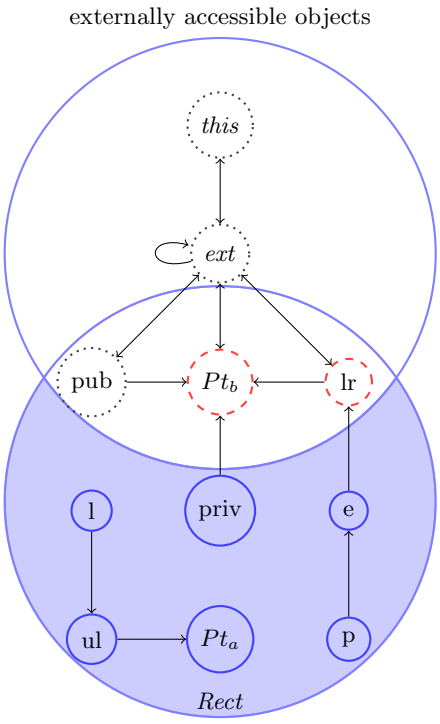

Fig. 2. Graph based representation of the final domain state for class *Rect* for the program on the left after the analysis has finished, dashed nodes represent escaped objects. Dotted nodes are externally accessible before any method from *Rect* is called. An edge from symbol a to symbol b means that b is in the points-to set of a. All nodes reachable from ext are also connected with each other, this is not shown for clarity. All addresses except *this* that are neither declared nor allocated inside Rect are abstracted to *ext*. *this* is an external instance of *Rect* for which the class-invariant is generated.

3 Abstract Semantics

Our framework transforms a given plug-in points-to analysis from whole-program or summary-based analysis to a class-modular class-escape analysis that can analyze a given class without any context information on how the class may be used. To achieve this, the domain of the plug-in analysis is extended by our own global domain $\mathbb{G}'$. The semantics of our analysis is defined by lifting the plug-in semantics to be able to handle the extended domain and the unknown context in which the class may be used in. A special address *ext* that abstracts all addresses which may exist outside the analyzed class definition is introduced. Furthermore, an address *this* is created that together with our global domain abstracts all possible instances of the analyzed class.

The concrete language our analysis operates on and its semantics is given in the corresponding technical report [7]. Any points-to analyses which adhere to the set of requirements given in this section can be used as plug-in analysis for our framework. Eventually, our analysis inherits the properties from the plug-in analysis while making the analysis class-modular and calculating sets of maybe class-escaping objects which are collected in the points-to set of *ext*.

This section is structured as follows. First, we list the requirements for the plug-in points-to analysis to be suitable for this framework. Afterwards, we define the necessary functions to lift the plug-in semantics $[\![s]\!]^\sharp$ to the abstract semantics $[\![s]\!]^{\sharp'}$ of our analysis. Finally, we give a set of initialization steps and a set of constraints that must be solved in order to perform the analysis.

Let $val^\sharp$ be the abstract values and $addr^\sharp \subseteq val^\sharp$ the abstract addresses used by the plug-in analysis.

Let $\mathcal{A} : \mathbb{D} \to lval \to \mathcal{P}(addr^\sharp)$ be a plug-in provided function that calculates the set of possible abstract addresses of a l-value given an abstract domain state $\rho^\sharp$.

Let $[\![s]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ be the abstract semantics of the plug-in points-to analysis for the statement s. The complete lattice $\mathbb{D}$ is the abstract domain used by the points-to analysis. From that we construct $\mathbb{D}' = \mathbb{D} \times \mathbb{G}'$ — the domain of our analysis, where $\mathbb{G}' : addr^\sharp \to \mathcal{P}(val^\sharp)$ extends the global (flow-insensitive) domain of the plug-in analysis.

Furthermore, let $q : \mathbb{D} \to addr^\sharp \to \mathcal{P}(val^\sharp)$ be a plug-in provided function that calculates the set of possible abstract values that may be contained by the memory at the given address when provided an abstract domain state and $\forall \rho^\sharp \in \mathbb{D} : q_{\rho^\sharp}\ \mathbf{null} = \emptyset$.

Intuitively, the plug-in analysis maintains some kind of mapping from abstract addresses to sets of abstract values for each program point abstracting the stack and the heap. How exactly this information is encoded inside the plug-in domain $\mathbb{D}$ is not relevant for our analysis. All addresses are initialized to **null**, so if an address a has not yet been written to in $\rho^\sharp$ then $q_{\rho^\sharp}\ a = \{\mathbf{null}\}$.

Finally, $\rho_1^\sharp = \rho_0^\sharp[x \to Y]$ denotes the weak update of $\rho_0^\sharp \in \mathbb{D}$ such that:

$$\forall z \in addr^\sharp : q_{\rho_1^\sharp}\ z \sqsupseteq \begin{cases} q_{\rho_0^\sharp}\ z \cup Y & : z = x \\ q_{\rho_0^\sharp}\ z & : \text{else} \end{cases}$$

Using this notation we can perform weak updates on the plug-in domain without knowing the details of $\mathbb{D}$.

For example the summary-based points-to and escape analysis from Whaley and Rinard [21], like virtually all other sound points-to analyses, fulfills all our requirements and can be plugged into our framework and thus become a class-modular, points-to and class-escape analysis.

To shorten the notation we also define a function $Q : \mathbb{D} \to \mathcal{P}(addr^\sharp) \to \mathcal{P}(val^\sharp)$ for sets of addresses.

$$Q_{\rho^\sharp}\ S \triangleq \bigcup_{x \in S} q_{\rho^\sharp}\ x$$

The function $Q^*_{\rho^\sharp} : \mathbb{D} \to \mathcal{P}(addr^\sharp) \to \mathcal{P}(val^\sharp)$ defines the abstract reachability using $Q_{\rho^\sharp}$:

$$\begin{aligned} Q^*_{\rho^\sharp}\ S \triangleq\ & F \cup Q^*_{\rho^\sharp}(F) \\ & \textbf{where } F = Q_{\rho^\sharp}\ S \cup \bigcup_{\substack{c \in Q_{\rho^\sharp} S \\ f_i \in public\ fields\ of\ c}} \mathcal{A}_{\rho^\sharp}(c \to f_i) \end{aligned}$$

The analysis is performed on a given class which we will call *Class*. Before starting the analysis an instance of *Class* is allocated and stored in the global variable *a_this* which we assume is not used in the analyzed code. Also, a global variable called *a_ext* of the most general pointer type (e.g. Object for Java or void* for C++) is created using the plug-in semantics (*a_ext* is also assumed not to be used in the analyzed code):

$$\rho_0^\sharp = (\llbracket a\_this := new\ \ Class \rrbracket^\sharp \circ \llbracket a\_ext := \textbf{null} \rrbracket^\sharp)\ d_0^\sharp$$

where $d_0^\sharp$ is the initial state of the plug-in domain, before any code has been analyzed. At this stage of the analysis *new* does not execute any constructors. As both *a_this* and *a_ext* are not used within the analyzed code, they do not change the semantics of the analyzed code and our lifted semantics can use these variables to communicate with the plug-in analysis and store special information as explained in the following.

The set *this* which contains all possible addresses of the allocated *Class* is defined as:

$$this = Q_{\rho_0^\sharp}(\mathcal{A}_{\rho_0^\sharp}(a\_this))\,.$$

The value *this* is meant to abstract all instances of *Class* that can exist (for the plug-in, *this* is an instance of *Class* that cannot be accessed from the program

unless our analysis provides its address). The plug-in analysis should be field sensitive at least for the *Class* instance addressed by *this* in order to exceed the precision of other points-to analyses when an unknown context is used.

The set *fields* contains all addresses of the *public* fields from the *Class* instance *a_this*:

$$fields = \bigcup_{f_i \in public\ fields\ of\ Class} \mathcal{A}_{\rho_0^\sharp}(a\_this \to f_i)\,.$$

Since the analysis is class-modular, only class declarations are analyzed. Hence, most of the program code is hidden from the analysis. We differentiate program segments which are visible to the analysis and the rest by defining external code:

Definition 1 (External Code)
External code with respect to a class C denotes all code that is not part of the class declaration of C. If no class C is stated explicitly, then class Class is assumed.

The points-to set of *ext* abstracts all addresses accessible from external code.

$$ext = \mathcal{A}_{\rho_0^\sharp}(a\_ext)$$

Initially, only *ext* itself, *this* and the *public* fields from *this* are reachable from external code, so *ext* must point to itself, *this* and the *public* fields, as an instance of *Class* may be allocated in code external to *Class*.

$$\rho_1^\sharp = \rho_0^\sharp[a \to ext \cup this \cup fields \mid a \in ext] \tag{1}$$

As the points-to set of *ext* contains multiple distinct objects, only weak updates can be performed on *ext* by the plug-in analysis.

Fields from *a_this* and their content become member of $Q_{\rho^\sharp}\ ext$ during the analysis if they may escape the *Class*. So after the analysis has finished, all possibly escaped memory locations are contained in $Q_{\rho^\sharp}\ ext$, all other memory locations do not escape the *Class* and are inaccessible from external code.

In the following we describe how the plug-in semantics $[\![s]\!]^\sharp$ is lifted to produce the abstract semantics $[\![s]\!]^{\sharp'}$ of our class-modular class-escape analysis:

The global addresses are constituted by *ext* and the fields of *this* since modifications of these addresses' values are observable inside different member methods of *this*, even if these methods do not call each other. For example, a method from *Class* may return an address to external code which was not previously accessible by external code. Later, external code may invoke a method from *Class* passing the newly accessible address (or something reachable thereof) as parameter to the method.

$$global : \mathcal{P}(val^\sharp)$$
$$global \triangleq ext \cup fields$$

Given the state of the plug-in domain, $globals : \mathbb{G}' \to \mathbb{D} \to \mathbb{G}'$ calculates the new state of the global domain $\mathbb{G}'$:

$$globals\ g^\sharp\ \rho^\sharp\ x \triangleq \begin{cases} q_{\rho^\sharp}\ x \cup g^\sharp\ x & : x \in global \\ \emptyset & : \text{else} \end{cases}$$

The global domain state tracks modifications to fields of *a_this* between different invocations of *Class*-methods from external code.

modify over-approximates the effects of code external to *Class*. In a single-threaded setting, these effects cannot occur inside code of *Class* so *modify* is applied when leaving code from *Class*. This happens either when returning from a *public* method to external code or when calling an unknown method. We assume that all methods from *Class* are executed sequentially. If other threads (that do not call methods from *Class*) exist, then *modify* must be applied after every atomic step a statement is composed of, as the other threads my perform modifications on escaped objects at any time. If no additional threads exit, then modifications of escaped objects can happen only before a method from *Class* is entered, when an external function is called and after a method from *Class* is exited. As external code may modify all values from addresses it can access to all values it can access, *modify* ensures all possible modifications are performed.

$$\begin{aligned} &\mathit{modify} \quad : \mathbb{D} \to \mathbb{D} \\ &\mathit{modify}\ \rho^\sharp \triangleq \rho^\sharp[x \to Q^*_{\rho^\sharp}\ \mathit{ext} \mid x \in Q^*_{\rho^\sharp}\ \mathit{ext}] \end{aligned}$$

The following semantic equation is inserted into the semantics $[\![s]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ of the plug-in analysis (or it replaces the existing version).

$$\begin{aligned} [\![l := e_0 \to m^{\text{extern}}(e_1, \ldots, e_n)]\!]^\sharp_{\rho^\sharp} \triangleq ([\![l := a\_ext]\!]^\sharp \circ \mathit{modify} \circ \\ [\![\mathbf{deref}(a\_ext) := e_0]\!]^\sharp \circ \ldots \circ [\![\mathbf{deref}(a\_ext) := e_n]\!]^\sharp)\ \rho^\sharp \end{aligned}$$

Methods m^{extern} are called from within *Class* but are not analyzed (e.g. because the code is not available). This makes the analysis modular with respect to missing methods in addition to its class-modularity. The procedure *unknown*, which is called in line 24 from our example in Fig. 1, represents such an external method where the above rule applies.

As shown in the proof [7], reading from *a_ext* and writing to **deref**(*a_ext*) correctly over-approximates reads from non-class-local r-values and writes to non-class-local l-values.

Finally, we give the abstract semantics $[\![s]\!]^{\sharp'} : \mathbb{D} \times \mathbb{G}' \to \mathbb{D} \times \mathbb{G}'$ of our analysis for a statement s.

$$\begin{aligned} [\![e_0 \to m(e_1, \ldots, e_n)]\!]^{\sharp'}_{(\rho^\sharp, g^\sharp)} \triangleq\ & (\rho^\sharp_2, \mathit{globals}\ g^\sharp\ \rho^\sharp_2) \\ & \textbf{where}\ \rho^\sharp_2 = \mathit{modify}([\![\mathbf{deref}(a\_ext) := e_0 \to m(e_1, \ldots, e_n)]\!]^\sharp_{\rho^\sharp_1}) \\ & \quad \textbf{and}\ \rho^\sharp_1 = \rho^\sharp[x \to g^\sharp\ x \mid x \in \mathit{global}] \end{aligned}$$

Our transfer function is invoked only for top-level methods when solving the constraint systems for the analysis (see Eq. 3,4). First, the current state of the flow-insensitive fields and *ext* is joined into the plug-in domain. Then the plug-in semantics (which now contains our patched rule for m^{extern}) is applied and

stores the return value of the method in *a_ext*. Afterwards, *modify* is applied to over-approximate the effects of external code that may execute after the top-level method is finished. Finally, the new global domain state is calculated using *globals*.

Before starting the actual analysis, the effects of external code that might have executed before a constructor from *Class* is called are simulated by applying *modify*:

$$(\rho_i^\sharp, g_i^\sharp) = modify(\rho_1^\sharp, globals\ g_0^\sharp\ \rho_1^\sharp) \tag{2}$$

Here, $g_0^\sharp$ is the bottom state of the global domain.

Then, the constructors are analyzed. Since we know that only one constructor is executed when a new object is created, it is sufficient to calculate the least upper bound of the effects of all available constructors:

$$(\rho_c^\sharp, g_c^\sharp) = \bigsqcup_{m \in public\ constructor\ of\ Class} [\![a\_this \to m(a\_ext, ..., a\_ext)]\!]^{\sharp'}(\rho_i^\sharp, g_i^\sharp) \tag{3}$$

Afterwards, the *public* methods from *Class* with all possible arguments and in all possible orders of execution are analyzed by calculating the solution [15] of the following constraint system,

$$(\rho_f^\sharp, g_f^\sharp) \sqsupseteq (\rho_c^\sharp, g_c^\sharp) \tag{4}$$

$$\forall m \in public\ method\ of\ Class:$$
$$(\rho_f^\sharp, g_f^\sharp) \sqsupseteq [\![a\_this \to m(a\_ext, ..., a\_ext)]\!]^{\sharp'}(\rho_f^\sharp, g_f^\sharp)$$

in order to collect all local and non-local effects on the *Class* until the global solution is reached. *a_ext* is passed for all parameters of the method as it contains all values that might be passed into the method. For non pointer-type arguments the plug-in's top value $\top_{type}$ for the respective argument type must be passed as argument to the top-level methods. If the target language supports function-pointers then all *private* methods for which function-pointers exist must be analyzed like *public* methods, if the corresponding function-pointer may escape the class.

When inheritance and *protected* fields are of interest, the complete class hierarchy must be analyzed. If the language allows to break encapsulation then additional measures must be taken to detect this. For example, C++ allows friends and *reinterpret_cast* to bypass access modifiers [18]. Friend declarations are part of the class declaration and as such easily detected. Usage of *reinterpret_casts* on the analyzed *Class* can be performed outside the class declaration, so additional code must be checked. Still, finding such casts is cheaper than doing a whole-program pointer analysis. In other languages, e.g. Java, such operations are not allowed and no additional verification is required.

4 Experimental Results

In this Section we present an implementation of the analysis and give a set of benchmark results that show how the analysis performs when it is applied to large sets of C++ code. As plug-in analysis a custom points-to analysis was implemented using the Goblint[20] framework.

The C++ code is transformed to semantically equivalent C using the LLVM [9] as the Goblint front-end is limited to C. Inheritance and access modifier information is also extracted and passed into the analysis. During this transformation, we verify that the access modifiers are not circumvented by casts or friends. No circumvention of access modifiers was found for the benchmarked code. Better analysis times can be expected from an analyzer that works directly with OO code as the LLVM introduces many temporary variables that have to be analyzed as well.

As additional input to the analysis a list of commonly used methods from the STL that were verified by hand was provided. Without this information more fields are flagged as escaping incorrectly by our implementation, because they are passed to an STL method which is considered external. This additional input is not required when using a different plug-in analysis that does not treat the STL methods as external.

The analysis is performed on two code-sets — the **Industrial** code is a collection of finite state machines that handle communication protocols in an embedded real-time setting whereas **Ogre**[11] is an open source 3d-engine. The results are given in Table 1.

Table 1. Benchmark results

Code	Classes	C++[loc]	C[loc]	Time[s]	Ext[%]	σ
Industrial	44	28566	1368112	282	23	24
Ogre	134	71886	1998910	42	62	36

The **C++** and **C** columns describe the size of the original code and its C code equivalent, respectively. More complex C++ code lines generate more C lines of code, so the ratio of code size is a measure for the complexity of the code that needs to be analyzed. The table shows that the industrial code is on average more complex than the Ogre code, requiring more time to analyze per line of code.

The time column represents the total time to analyze all the classes. The last two columns in the table show the mean and the standard deviation of the percentage of fields that the analysis identified as escaping. Fields that are identified as escaping limit the precision of subsequent analysis passes since they can be modified by external code. For the rest of the fields detailed information can be generated. A field f_i is escaping if and only if $\mathcal{A}_{\rho_f^\sharp}(a\_this{\rightarrow}f_i) \cap Q_{\rho_f^\sharp} ext \neq \emptyset$.

Only 23% of the fields are escaping for the industrial code. This is the case because most of the code processes the fields directly rather than passing the fields to methods outside the analyzed class, yielding good precision.

Inside the Ogre code most fields are classes themselves, so many operations on fields are not performed by code belonging to the class containing the field but by the class that corresponds to the field's type. Our plug-in analysis implementation handles the methods of these fields as unknown methods and assumes that the field escapes. By using a plug-in analysis that analyses into these methods from other classes (e.g. Whaley and Rinard [21]) the precision of the analysis for the Ogre code can be improved to about 25% escaping fields, as indicated by preliminary results. Our analysis provides an initial context to analyze deeper into code outside of the class declaration. Especially for libraries it is necessary to generate an initial context if the library is analyzed in isolation.

So for the presented examples, for more than 75% of the fields detailed information can be extracted without analyzing the code that instantiates and uses the initial class. The benchmark times are obtained by analyzing all classes sequentially on a 2.8 Ghz Intel Core I7 with 8GB RAM. Since the results for each class are independent from the results for all other classes, all classes could be analyzed in parallel.

5 Conclusion

We have presented a sound class-modular, class-escape and points-to analysis based on the encapsulation mechanisms available in OO-languages. The analysis can be applied to a set of classes independently without analyzing the code that uses the class thus reducing the amount of code that needs to be analyzed compared to whole-program and summary-based analysis.

In addition, we have presented an easy to apply, yet powerful transformation of non-class-modular points-to analyses into class-modular, points-to and class-escape analyses. Since our framework has very weak requirements on potential plug-in points-to analyses, it can be applied to virtually all existing points-to analyses. We have shown, that the transformation will produce a sound analysis, given that the whole-program plug-in analysis was sound. Moreover, the resulting class-modular, class-escape and points-to analysis will inherit the properties of the plug-in and therefore benefits from previous and future work on points-to analyses.

The presented benchmarks show that the analysis can be applied to large, real world code yielding good precision. Due to the modularity of the analysis, flow sensitive pointer analysis becomes viable for compiler optimization passes. Class files can be analyzed and optimized independently before they are linked to form a complete program. Hence, various compiler optimizations and static verifiers can benefit from a fast class-modular class-escape and pointer analysis. Especially in large OO software projects that enforce common coding standards[19] the usage of non-private fields is rare, so good results can be expected.

Acknowledgements. We would like to thank Prof. Helmut Seidl for his valuable input and support. In addition, we would like to thank Axel Simon for contributing his time and expertise on pointer analyses. Finally, we would like to thank Nokia Siemens Networks for providing their source code and additional funding.

References

1. Andersen, L.: Program analysis and specialization for the C programming language. Tech. rep., 94-19, University of Copenhagen (1994)
2. Blanchet, B.: Escape analysis for object-oriented languages: application to Java. SIGPLAN Not. 34, 20–34 (1999)
3. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. SIGPLAN Not. 38, 213–223 (2003)
4. Cheng, B.C., Hwu, W.M.W.: Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI 2000, pp. 57–69. ACM, New York (2000)
5. Cousot, P., Cousot, R.: Modular Static Program Analysis. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 159–179. Springer, Heidelberg (2002)
6. Dolby, J., Chien, A.: An automatic object inlining optimization and its evaluation. SIGPLAN Not 35(5), 345–357 (2000), aCM ID: 349344
7. Herz, A., Apinis, K.: Class-Modular, Class-Escape and Points-to Analysis (Proof). Tech. rep., TUM-I1202, Technische Universität München (2012)
8. Horwitz, S., Shapiro, M.: Modular Pointer Analysis. Tech. rep., 98-1378, University of Wisconsin–Madison (1998)
9. Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis transformation. In: International Symposium on Code Generation and Optimization, CGO 2004, pp. 75–86 (2004)
10. Logozzo, F.: Automatic Inference of Class Invariants. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 211–222. Springer, Heidelberg (2004)
11. Open Source 3D Graphics Engine OGRE, `http://www.ogre3d.org/`
12. Porat, S., Biberstein, M., Koved, L., Mendelson, B.: Automatic detection of immutable fields in Java. In: Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 2000, p. 10. IBM Press (2000)
13. Rountev, A.: Component-Level Dataflow Analysis. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Ren, X.-M., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 82–89. Springer, Heidelberg (2005)
14. Rountev, A., Ryder, B.G.: Points-to and Side-Effect Analyses for Programs Built with Precompiled Libraries. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 20–36. Springer, Heidelberg (2001)
15. Seidl, H., Vene, V., Müller-Olm, M.: Global invariants for analyzing multithreaded applications. Proc. of the Estonian Academy of Sciences: Phys., Math. 52(4), 413–436 (2003)
16. Seidl, H., Vojdani, V.: Region Analysis for Race Detection. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 171–187. Springer, Heidelberg (2009)
17. Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1996, pp. 32–41. ACM, New York (1996)

18. Stroustrup, B.: The C++ programming language, vol. 3. Addison-Wesley, Reading (1997)
19. Sutter, H., Alexandrescu, A.: C++ coding standards: 101 rules, guidelines, and best practices. Addison-Wesley Professional (2005)
20. Vojdani, V., Vene, V.: Goblint: Path-sensitive data race analysis. Annales Univ. Sci. Budapest., Sect. Comp. 30, 141–155 (2009)
21. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 1999, pp. 187–206. ACM, New York (1999)
22. Wimmer, C., Mössenböck, H.: Automatic feedback-directed object inlining in the java hotspot(tm) virtual machine. In: Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, pp. 12–21. ACM, New York (2007)