Modular Checking of Confinement for Object-Oriented Components using Abstract Interpretation

Kathrin Geilmann Arnd Poetzsch-Heffter
University of Kaiserslautern, Germany
[geilmann,poetzsch]@cs.uni-kl.de

Abstract

The ability to express confinement properties of objects is crucial in the context of component-oriented software development. Recently, several techniques based on type systems have been developed to express and check confinement properties for object-oriented programs. Unfortunately, learning these often quite complex type systems is a burden to programmers. Furthermore, without inference techniques, they require nontrivial code annotations. In this paper, we propose a very lightweight approach to add encapsulation support to a Java-like language. We annotate class declarations and `new` statements to express which classes instantiate new components and which objects are confined to their surrounding component. To check encapsulation we developed a modular static analysis based on an abstract interpretation of single components. Successfully analyzed components will never break confinement regardless of the program they are used in.

1. Introduction

Combining components to form complex software systems is a very challenging task: components may interfere with each other, cause unexpected side-effects and introduce bugs that are hard to find, because they only appear in the complete system but not when the components are considered individually. Especially if the components come from different origins e.g. third party libraries, for which one may not even have the sources, the effects of sticking them together are hard to predict. The main reason for these problems is that current object-oriented programming languages work with classes, objects and references but have a weak notion of components (or none at all). A programming language with a component model which defines clear component boundaries and allows the confinement of the internal representation, eases the development of component-oriented software.

Recently different solutions to get confinement and aliasing under control have been proposed. The most promising ones are ownership types and ownership domains [2, 3, 6, 11, 13, 14, 16, 31], which use static type checking to control aliasing of objects in the internal representation. Type systems that are expressive enough to support common programming patterns are quite complex and require the normal type information to be extended with parametric ownership annotations, which is very challenging for the average programmer. To reduce the annotation burden, type inference has been used [1, 3, 6, 7, 19, 25, 27], but the programmer still has to understand the complex type systems.

In this paper, we propose a very lightweight approach to ensure confinement, which lets the programmer express the desired confinement when creating objects. Our approach is the first one for ensuring confinement in the box model, which is not based on type systems but on abstract interpretation of components. We use a small Java-like language with a lightweight component model which is a slightly simplified version of the box model [27–29]. We developed a static, modular analysis based on abstract interpretation of boxes, to ensure the confinement of components. The analysis takes the implementation of a box and executes it together with its most-general client. The most-general client is an abstraction of all possible clients and creates all possible traces through the box. If the execution succeeds, the box never exposes any confined object regardless of the program that uses the box.

The remainder of the paper is structured as follows. In Section 2 we describe syntax, semantics and confinement properties of our language. In Section 3 we present the abstract interpretation and abstract semantics. The abstract interpretation is used to define a confinement analysis for components (Section 4). Section 5 discusses related work and Section 6 contains our conclusion.

2. Language and Concrete Semantics

We present a confinement analysis for components based on a simplified version of the box model [28]. The box model extends the object-oriented programming world of classes, objects, object-local state and methods with the notion of a box. A new component instance, called a *box*, is implicitly created when a new object of a so-called *box class* is instantiated (a class is called a box class if annotated by `box`). The object created along with the box is called the box owner. A box is essentially a dynamically created group of objects containing its owner and all objects created by objects in the box. An object can be created as *confined* to a box b adding the annotation `confined` to the creation expression. It is not allowed to reference confined objects from outside of b. The owner and the other non-confined objects of the box are called *boundary* objects. Boundary objects may be referenced from everywhere.

The most important property of the box model is that confined objects stay confined to the box and no reference to these objects will ever leave the box. This guarantees that all accesses to the box are done via methods of the boundary objects and therefore their effects are under the control of the box instance. Such boxes are called encapsulated:

DEFINITION 1 (Encapsulated box).
A box is called encapsulated *iff confined objects cannot be referenced from the outside.*

Figure 1 presents a linked list class with iterators as a code example. Figure 2 shows a box of this class, after a client added two elements and created and moved an iterator. The code differs from Standard Java by two annotations. First, the list class is annotated with `box`. This means that each instantiation of this class also creates a new box. Second, the creation of `Node` objects in the add method is annotated with `confined`. This ensures, that the objects created here are confined to the box and these objects are never exposed.

```
box class List {
   Node head;
   void add(Object o) {
      Node t = head;
      head = new confined Node(t,o);
   }
   Iterator iter() {
      Iterator i = new Iterator(head);
      return i;
   }
}

class Node {
   Node next;
   Object value;
   Node (Node n, Object o) {
      next = n;
      value = o;
   }
   Node next() {...}
   Object value() {...}
}

class Iterator {
   Node current;
   Iterator (Node n) {
      current = n;
   }
   Object next() {
      Node t = current;
      current = current.next();
      return t.value();
   }
}
```

Figure 1. An implementation of List component

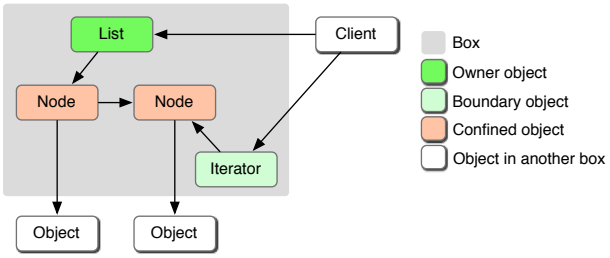

Figure 2. Box structure of a program using the list component

2.1 Syntax

For the formal definition of the analysis, we use a small object-oriented Java-like language, whose syntax is shown in Figure 3. Programs consist of class declarations and a main statement that instantiates a box class. Class declarations are annotated with `box` for box classes and `helper` for normal classes. Statements in our language are labeled. The `new` statement takes an annotation about the confinement of the new object and the `return` statement is only allowed as the last statement of a method body. We currently do not support nested boxes, i.e., a new box is always created outside the current box. Owners may not be confined. Fields are object private, i.e. they can only be accessed on `this`. For the rest of the paper we assume that all labels, variables (except `this`), fields and formal parameters are unique. For readability, we drop the keywords `helper` and `boundary` as well as the labels, if they are not explicitly needed.

$$
\begin{array}{lcl}
P & := & \overline{CD}\ \{\texttt{C x; x = new C(); x.m(}\overline{\texttt{null}}\texttt{)}\} \\
CD & := & cm\ \texttt{class C [extends C]}\{\overline{\texttt{C f;}}\ \overline{M}\} \\
M & := & \texttt{C m (}\overline{\texttt{C x}}\texttt{)}\ \{\overline{\texttt{C x;}}\ S\} \\
S & := & [S\ S]^l \\
 & | & [\texttt{x = } e\texttt{;}]^l \\
 & | & [\texttt{x = new } nm\ \texttt{C();}]^l \\
 & | & [\texttt{x = x.m(}\overline{\texttt{x}}\texttt{);}]^{l_1,l_2} \\
 & | & [\texttt{x = this.f;}]^l \\
 & | & [\texttt{this.f = } e\texttt{;}]^l \\
 & | & [\texttt{if (x) }\{S\}\texttt{ else }\{S\}]^l \\
 & | & [\texttt{return x;}]^l \\
 & | & \epsilon \quad \text{(empty statement)} \\
e & := & \texttt{x} \mid \texttt{null}
\end{array}
$$

Annotations

$$
\begin{array}{lcl}
cm & := & \texttt{box} \mid \texttt{helper} \\
nm & := & \texttt{confined} \mid \texttt{boundary}
\end{array}
$$

$$
\begin{array}{lcl}
l & \in & L \text{ (unique labels)} \\
\texttt{C} & \in & \text{class names} \\
\texttt{x} & \in & \text{variable names} \\
\texttt{f} & \in & \text{field names} \\
\texttt{m} & \in & \text{method names}
\end{array}
$$

Figure 3. Abstract syntax of our language

2.2 Semantics

In this section, we sketch the concrete semantics of our language. We use a small step semantics, which aborts the program execution whenever encapsulation is broken. We use the following notations, $\overline{x}$ denotes sets with elements x_i, $\mathcal{S}(\texttt{x})$ looks up the value of `x` in the topmost stack frame of the stack $\mathcal{S}$ and $\mathcal{H}(\o_{id})$ looks up objects in the heap.

2.2.1 Program State

A program state (cf. Figure 4) is either a pair of heap and stack or the abortion state. Heaps are mappings from object identifiers to objects and stacks consist of frames mapping variables to values.

To track confinement of objects, each object has a reference to the owner of the box, it belongs to. The owner is set during creation of the object and does not change. Furthermore, each object stores a boolean flag, describing whether it is confined. With this flag and the owner, we can decide if passing an object reference is allowed. Additionally, each object stores its exposed status. An exposed object is a boundary object, which has been passed as parameter or return value to an object in a different box, i.e., a reference left the owning box. This flag is write-only in the concrete semantics, but it simplifies the definition of the abstraction for the analysis. Values are either object identifiers or `null`.

We say that a program state is *valid*, if it is a reachable state in some program.

2.2.2 Semantic Rules

The concrete semantics checks confinement dynamically such that a program can never violate encapsulation. Execution terminates in state $\top_{abort}$ whenever a violation would occur.

Most semantic rules are standard and modify heap and stack as usual. The interesting rules are the ones that guarantee the confinement. These are the rules for calls and returns (Figure 5). For calls, we have to distinguish between *internal* calls (the `this`-object and the receiver are in the same box) and *external* calls (the call passes a box border). The distinction is done in the function samebox by comparing the owners of the receiver object and `this`.

$$
\begin{array}{lcll}
\sigma & ::= & (\mathcal{H}, \mathcal{S}) \mid \top_{abort} & \text{program state} \\
\mathcal{H} & ::= & o_{id} \mapsto o & \text{heap} \\
\mathcal{S} & ::= & \epsilon \mid [\texttt{this} \mapsto o_{id}, \overline{\texttt{x}} \mapsto \overline{v}] \cdot \mathcal{S} & \text{stack} \\
o & ::= & (o_{id}, \texttt{C}, \text{conf}, \text{exp}, \overline{\texttt{f}} \mapsto \overline{v}) & \text{object} \\
v & ::= & o_{id} \mid \texttt{null} & \text{values}
\end{array}
$$

o_{id}	object identifiers
f	fields of an object
x	local variables and method parameters
C	type of the object
conf, exp	confinement, exposure status

Figure 4. Concrete program states

CALL-INTERNAL

$$
\frac{\begin{array}{c}\text{samebox}((\mathcal{H},\mathcal{S}),\texttt{this},\texttt{x}_2) \qquad \texttt{C} = \text{type}(\mathcal{H}(\mathcal{S}(\texttt{x}_2))) \\ \overline{\texttt{p}} = \text{params}(\texttt{C},\texttt{m}) \qquad \overline{\texttt{y}} = \text{localVars}(\texttt{C},\texttt{m}) \\ \mathcal{S}' = [\overline{\texttt{p}} \mapsto \mathcal{S}(\overline{\texttt{x}}), \overline{\texttt{y}} \mapsto \texttt{null}, \texttt{this} \mapsto \mathcal{S}(\texttt{x}_2)] \cdot \mathcal{S}\end{array}}{\begin{array}{c}(\mathcal{H},\mathcal{S}),\ \texttt{x}_1 \texttt{ = x}_2\texttt{.m(}\overline{\texttt{x}}\texttt{);} \Rightarrow \\ (\mathcal{H},\mathcal{S}'),\ \text{body}(\texttt{C},\texttt{m})\ \texttt{x}_1\texttt{=retVal}_{\texttt{Cm}}\texttt{;}\end{array}}
$$

CALL-EXTERNAL

$$
\frac{\begin{array}{c}\neg\text{samebox}((\mathcal{H},\mathcal{S}),\texttt{this},\texttt{x}_2) \\ \text{confined}(\mathcal{H}(\mathcal{S}(\texttt{x}_i))) = \text{false for all } \texttt{x}_i \in \overline{\texttt{x}} \\ \mathcal{H}' = \text{expose}(\overline{\texttt{x}}, (\mathcal{H},\mathcal{S})) \qquad \texttt{C} = \text{type}(\mathcal{H}(\mathcal{S}(\texttt{x}_2))) \\ \overline{\texttt{p}} = \text{params}(\texttt{C},\texttt{m}) \qquad \overline{\texttt{y}} = \text{localVars}(\texttt{C},\texttt{m}) \\ \mathcal{S}' = [\overline{\texttt{p}} \mapsto \mathcal{S}(\overline{\texttt{x}_i}), \overline{\texttt{y}} \mapsto \texttt{null}, \texttt{this} \mapsto \mathcal{S}(\texttt{x}_2)] \cdot \mathcal{S}\end{array}}{\begin{array}{c}(\mathcal{H},\mathcal{S}),\ \texttt{x}_1 \texttt{ = x}_2\texttt{.m(}\overline{\texttt{x}}\texttt{);} \Rightarrow \\ (\mathcal{H}',\mathcal{S}'),\ \text{body}(\texttt{C},\texttt{m})\ \texttt{x}_1\texttt{=retVal}_{\texttt{Cm}}\texttt{;}\end{array}}
$$

CALL-EXTERNAL-ABORT

$$
\frac{\begin{array}{c}\neg\text{samebox}((\mathcal{H},\mathcal{S}),\texttt{this},\texttt{x}_2) \\ \text{confined}(\mathcal{H}(\mathcal{S}(\texttt{x}_i))) = \text{true for some } \texttt{x}_i \in \overline{\texttt{x}}\end{array}}{(\mathcal{H},\mathcal{S}),\ \texttt{x}_1 \texttt{ = x}_2\texttt{.m(}\overline{\texttt{x}}\texttt{);} \Rightarrow \top_{abort}}
$$

Figure 5. Call-rules of the concrete semantics

For internal calls, a new stack frame with the actual parameters and `this` set to the receiving object is created and pushed on top of the existing stack. The execution continues with the method body. When executing the return statement at the end of the method body, the return value is stored in the special variable $\texttt{retVal}_{\texttt{Cm}}$, which is then copied into the variable $\texttt{x}_1$. Like all other variables $\texttt{retVal}_{\texttt{Cm}}$ has to be unique, therefore we have a variable $\texttt{retVal}_{\texttt{Cm}}$ for each method `m` in each class `C`.

For external calls, the rules additionally check confinement. If some of the objects that are passed as actual parameters have the confined flag set, i.e., have been created with the `confined` annotation, the execution is aborted (CALL-EXTERNAL-ABORT). Otherwise, the function expose is used to set exposed flag of the all actual parameters to true and the execution continues like in the internal case.

For the execution of return statements across box borders, we do the same checks for the return value (not shown).

2.2.3 Characterizing Encapsulation

Based on the semantics, we can state precisely when a box is encapsulated. Let `B` be a box class. A *codebase* $\overline{CD}$ of `B` is a set of classes such that

1. $\texttt{B} \in \overline{CD}$
2. $\overline{CD}$ is declaration closed, i.e., all classes used in $\overline{CD}$ are declared in $\overline{CD}$.
3. There is no proper subset of $\overline{CD}$ satisfying (1.) and (2.).

In the following, we assume that a box class is always given with a particular codebase. A program P *comprises* a box class `B`, if P is a context correct extension of $\overline{CD}$ where $\overline{CD}$ is the assumed codebase of `B`.

DEFINITION 2 (Encapsulated box class). *A box class `B` is called* encapsulated *if the boxes created from `B` are encapsulated.*

COROLLARY 1 (Encapsulated box class). *A box class `B` is encapsulated iff there is no program P comprising `B` such that $\top_{abort}$ is reachable from P's initial state.*

3. Abstract Semantics

The defined language enforces confinement at runtime. In order to detect bugs as early as possible, and to guarantee that programs will not abort due to a confinement error, a static approach to enforce confinement is needed. Additionally, in the context of component-oriented software development, it is useful to give guarantees not only about the behavior of whole programs, but also about single components. This simplifies the reuse of components, because combining components does not have side-effects for the confinement.

The idea is to statically guarantee that a box class `B` does not break the confinement property in any context. To achieve this, we put `B` into a context that first creates an instance b of `B` and then generates all possible traces through it. We call this context the *most-general client* (MGC). The MGC is a non-deterministic program that can call any method on any of the exposed objects of b. As parameters for the method calls, the MGC uses either objects that have been exposed by the box, or objects instantiated from classes of MGC. In the classes of the MGC, all method implementations behave non-deterministically as well.

The states of the program MGC + `B` can be partitioned into a part that belongs to b, the *box state*, and the rest. The box state contains all objects belonging to b and all stack frames, in which `this` references an object in b.

The last problem for static analysis is that the space of possible box states is infinite, because the most general client can generate infinitely many different traces with an unbounded number of objects and call stacks of an unbounded size. In order to make the analysis computable, we have to abstract this infinite state space to a finite one.

This leads to an analysis based on an abstract interpretation of the MGC + `B` program. The analysis takes as input the codebase of `B` and is implicitly parameterized with the box class `B`. The component is executed with an abstract semantics on the abstract state space. If the abstract execution does not reach the abortion state, we can conclude that for every program the box will never violate encapsulation.

In the following, we describe the abstraction and the abstract semantics in more detail. A comprehensive presentation is contained in [20].

3.1 Abstract State

To deal with the infinite concrete state space, we have to abstract concrete program states as defined in Figure 4 to abstract box states. The definition of an abstract box state is given in Figure 6. The main idea of the abstraction is to consider only one (abstract) object per creation site, i.e., one per `new`-statement.[1] This means that e.g. a

[1] This is a common approach for static analysis of heap manipulating programs, cf. [10, 24, 33, 34].

$\mathcal{B}$	::=	$(\mathcal{H}_a, \mathcal{S}_a) \mid \top_{abort}$	abstract box state
$\mathcal{H}_a$	::=	$l \mapsto o_a$	abstract box heap
$\mathcal{S}_a$	::=	$\texttt{this} \mapsto v_t, \overline{\texttt{x}} \mapsto \overline{\overline{v_a}}$	abstract stack
o_a	::=	$(\texttt{C}, \text{conf}, \text{exp}, \overline{\texttt{f}} \mapsto \overline{\overline{v_a}})$	abstract object
v_a	::=	$l \mid \texttt{null} \mid \text{extRef}$	abstract values
v_t	::=	$l \mid \text{extRef} \mid \bot$	this values
l	::=	$l_o \mid l_n$	labels

l_n	labels of new statements
l_o	label of the owner object
f	fields of an object
x	local variables and method parameters
C	type of the object
conf, exp	confinement, exposure status

Figure 6. Abstract box states

list with several node objects which are all created by the same statement, has only one abstract node in the abstract state (cf. Figure 7). This leads to a finite number of so-called *abstract* objects. As object identifier, we use the label of the creation statement. Because the owner object is always created by some code outside the box, we do not have a new-statement for this object. Therefore we use the special label l_o for the owner. Besides these labels, an abstract value is either null or the special reference extRef, which represents references to objects outside the box. This allows to abstract all objects of the program context and is necessary to make the analysis modular. In consequence of this abstraction, we have a finite set of values.

The second idea to approximate possible concrete behavior is to use sets of abstract values for fields, parameters and local variables. We write $\overline{\overline{v_a}}$ to denote sequences of sets of values. This is needed, because in the concrete states the field f of objects created at the same statement can reference different objects. Similarly in the abstract stack, the value of an abstract variable can correspond to several concrete values. An exception is made for this, to allow the precise tracking in which object a method is executed. If this has the value extRef, the control flow is currently in an object outside the box. The value $\bot$ never occurs during the execution of a box, but it simplifies the definition of the abstraction functions (see below).

Abstract objects are defined as tuples of a confinement and an exposure status, the object's type and a mapping of fields to sets of abstract values. Abstract heaps are mappings from labels to abstract objects. To avoid infinite call stacks, we use a flat mapping of local variables (parameters) to sets of values as abstraction for the concrete stack. We use $\mathcal{S}_a(\texttt{x})$ and $\mathcal{H}_a(l)$ to look up values in the stack and heap respectively and $[\texttt{x} \mapsto \overline{v_a}]\mathcal{S}_a$ and $[l \mapsto o]\mathcal{H}_a$ for updates of the stack and the heap.

Galois Connection

To use abstract interpretation we have to relate valid concrete program states and abstract box states. Because our analysis is modular, i.e., it analyzes the codebase of a single component, all functions defined below are implicitly parametrized with the box class B of the component. First, we define a representation function β, which maps a valid concrete program state to an abstract box state and then use β to construct an abstraction function α and a concretization function γ. With α and γ we then define a Galois connection, which relates sets of concrete valid program states to sets of abstract box states.

We first need some auxiliary definitions. The set $\mathcal{I}_\sigma$ contains the object identifiers of objects in the concrete heap of the state σ, whose owners are of type B, i.e. all identifiers of objects belonging to a box of box class B. We assume a function cs that takes a concrete object identifier and returns the label of the new statement, at which the corresponding object has been instantiated[2]. Concrete values are abstracted by the function $abstract_v$. References to objects that are not part of some box of the box class B are abstracted to extRef, other references are abstracted to the corresponding label returned by cs, and null stays null.

For heap abstraction, all concrete objects of a heap $\mathcal{H}$, which have been created with the same label and are part of a box of box class B are merged into a single abstract object. This is done by the function $abstract_o$, which takes a set of concrete objects and returns an abstract object, which is exposed if at least one concrete object was exposed and whose fields store the union of the abstraction of the concrete values.

$$abstract_o(\sigma, \overline{o}) = (\texttt{C}, c, e_a, \overline{\texttt{f}} \mapsto \overline{v_a})$$

where $\overline{o} = \overline{(o_{id}, \texttt{C}, c, e, \overline{\texttt{f}} \mapsto \overline{v})}$, $e_a = \bigvee_{o_i \in \overline{o}} \text{exposed}(o_i)$ and $v_a = \cup_{o_i \in \overline{o}} abstract_v(\sigma, o_i(\texttt{f}))$. Using these functions the abstraction of a heap can be defined as

$$abstract_H(\sigma) = \overline{l} \mapsto \begin{cases} abstract_o(\sigma, \overline{o}) & l \in cs(\mathcal{I}_\sigma) \\ (\texttt{C}, c, \text{false}, \overline{\texttt{f}} \mapsto \{\}) & \text{otherwise} \end{cases}$$

where $cs(\mathcal{I}_\sigma) = \{cs(o_{id}) | o_{id} \in \mathcal{I}_\sigma\}$, $\overline{o}$ are all concrete objects in σ created at label l, C and c are given by the new statement with label l. Objects $(\texttt{C}, c, \text{false}, \overline{\texttt{f}} \mapsto \{\})$ can be seen as placeholders for objects that have not (yet) been created.

The concrete stack is abstracted by the function $abstract_S$. Let $\mathcal{F}_\sigma$ denote the set that contains all stack frames of the state σ, which correspond to some execution inside the box, i.e., frames with a value for this contained in $\mathcal{I}_\sigma$. In the abstract stack the value of a variable x is the union of the abstraction of all values of x in all stack frames of $\mathcal{F}_\sigma$. An exception is made for this, which gets the abstract value of the topmost stack frame, if the frame is contained in $\mathcal{F}_\sigma$ and extRef otherwise.

The representation function β is defined as

$$\begin{array}{lcl} \beta(\top_{abort}) & = & \top_{abort} \\ \beta(\sigma) & = & (abstract_H(\sigma), abstract_S(\sigma)) \end{array}$$

Note that concrete program states may contain multiple instances of the same box, which will be abstracted to a single abstract box.

For the Galois connection we choose $(\mathcal{P}_\sigma, \subseteq)$ where $\mathcal{P}_\sigma$ is the power set over all valid concrete program states as the complete lattice. To define a complete lattice on the abstract state space, we define a partial order $\leq$ on box states. $\mathcal{B}_1 \leq \mathcal{B}_2$ means that $\mathcal{B}_2$ is less precise about the values of fields and variables (parameters) and about the exposed status of objects.

$$\begin{array}{lll} (\mathcal{H}_a, \mathcal{S}_a) \leq (\mathcal{H}'_a, \mathcal{S}'_a) & \Leftrightarrow & \mathcal{H}_a \leq \mathcal{H}'_a \wedge \mathcal{S}_a \leq \mathcal{S}'_a \\ \mathcal{B} \leq \top_{abort} & & \text{for all } \mathcal{B} \end{array}$$

$$\begin{array}{lll} \mathcal{H}^1_a \leq \mathcal{H}^2_a & \Leftrightarrow & \mathcal{H}^1_a(l) \leq \mathcal{H}^2_a(l) \\ \mathcal{S}^1_a \leq \mathcal{S}^2_a & \Leftrightarrow & \mathcal{S}^1_a(\texttt{x}) \subseteq \mathcal{S}^2_a(\texttt{x}) \text{ for all variables x}, \\ & & \mathcal{S}^1_a(\texttt{this}) = \mathcal{S}^2_a(\texttt{this}) \text{ or } \mathcal{S}^1_a(\texttt{this}) = \bot \\ o_1 \leq o_2 & \Leftrightarrow & e_1 \Rightarrow e_2, \\ & & o_1(\texttt{f}) \subseteq o_2(\texttt{f}) \text{ or extRef} \in o_2(\texttt{f}) \end{array}$$

where $o_i = (\texttt{C}, c, e, \overline{\texttt{f}} \mapsto \overline{\overline{v_{ai}}})$ for $i = 1, 2$. Using $\leq$ we define a subset relation on sets of box states that takes the precision into account.

$$\overline{\mathcal{B}} \subseteq' \overline{\mathcal{B}'} \Leftrightarrow \forall \mathcal{B}_i \in \overline{\mathcal{B}}, \exists \mathcal{B}_j \in \overline{\mathcal{B}'} : \mathcal{B}_i \leq \mathcal{B}_j$$

[2] The creation site could be added as an additional component to the representation of concrete objects.

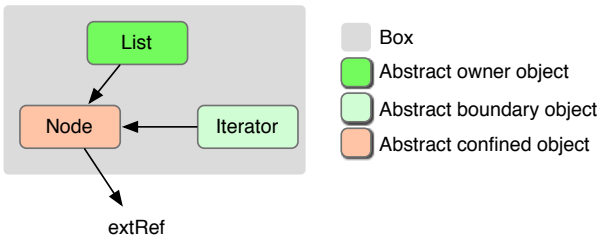

Figure 7. Abstract state of the list box in Figure 2

It is straightforward to show that $(\mathcal{P}_\mathcal{B}, \subseteq')$ with $\mathcal{P}_\mathcal{B}$ being the power set over the abstract state space is a complete lattice.

We define the abstraction and concretization functions in terms of the representation function β.

$$\begin{array}{l} \alpha : \mathcal{P}_\sigma \to \mathcal{P}_\mathcal{B} \\ \gamma : \mathcal{P}_\mathcal{B} \to \mathcal{P}_\sigma \\ \alpha(\overline{\sigma}) \quad = \quad \{\beta(\sigma_i) \mid \sigma_i \in \overline{\sigma}\} \\ \gamma(\overline{\mathcal{B}}) \quad = \quad \{\sigma \mid \beta(\sigma) \in \overline{\mathcal{B}}\} \end{array}$$

This gives us the Galois connection $(\mathcal{P}_\sigma, \alpha, \gamma, \mathcal{P}_\mathcal{B})$ over $(\mathcal{P}_\sigma, \subseteq)$ and $(\mathcal{P}_\mathcal{B}, \subseteq')$.

3.2 Abstract Semantics

The abstract reduction function $\Rightarrow_a^*$ takes a set of abstract box states and returns a set of pairs each consisting of an abstract state and a (potentially empty) statement, describing the remaining execution. $\Rightarrow_a^*$ is defined in terms of an abstract transformer function $\Rightarrow_a$ which executes a statement S on a single abstract state.

$$\overline{\mathcal{B}}, S \Rightarrow_a^* \bigcup_i \overline{(\mathcal{B}_i', S_i')}$$

where $\mathcal{B}_i, S \Rightarrow_a \overline{(\mathcal{B}_i', S_i')}$ for all $\mathcal{B}_i \in \overline{\mathcal{B}}$.

The abstract transformers are flow-insensitive. Like the concrete semantics, the abstract semantics aborts the execution whenever the confinement of an object would be violated.

The most important rules are the rules for handling calls and returns that result in leaving a box and the rules for entering a box. Call and return rules are shown in Figure 8.

To execute a call to a method `m` (A-CALL), the call is executed on each possible receiver.

If the receiver is a label, the call stays inside the box (A-CALL-INTERNAL). The receiver becomes the new value for `this` and the actual parameters are passed on the stack; the execution continues with the method body. Note that we always add new parameter values to already existing ones. This is required, because in the abstract semantics we merge different invocations of a method.

Whenever a call is directed at the outside of the box, the receiver of the call is extRef. This leads to the execution of the rule (A-CALL-EXTERNAL). Like in the concrete semantics we check that the parameter objects are not confined, and we expose all abstract objects that are referenced by the actual parameters. We then enter a state $(\mathcal{H}_a', \mathcal{S}_a')$ with extRef as `this` object, which signals that the control flow has left the box, i.e., the control passed to the MGC. If some of the actual parameters are confined objects, the execution is aborted (A-CALL-EXTERNAL-ABORT). As we do not know anything about the return value of an external call and because it is possible that the return value is a reference to an object in a different box, the rule uses `x = extRef` as the next statement. We do not have to consider other return values from the MGC, because we defined the lattice for box states such that a set containing extRef is always greater than one without. External references are the cause for breaking the confinement. Thus, extRef is the worst case scenario that may happen to the executing component. Because our analysis has to guarantee the absence of confinement errors, this is a conservative approximation, which reduces the state space the analysis has to handle.

Whenever the control flow is in the MGC (A-CALLBACK), the MGC can initiate a callback to any method of any exposed object or the MGC executes a `return` statement that returns back to the component. The function exposed returns the labels of all exposed objects, and the function methods looks up method bodies and parameters of a class. For each possible call we add extRef to all parameters.

The handling of the return statement is slightly more complicated, as we do not know to which object the control passes, because the abstract stack does not record the different method invocations. Therefore we return to all objects that could have initiated the call. The return of an internal call (A–RETURN-INTERNAL) puts the return value on the stack and the execution continues in all objects, which contain a call to the returning method somewhere in the code. It is obvious that this behavior produces more traces through the component than the concrete semantics. This is one of the points at which the analysis can be made more precise.

In case of a return from a call by the MGC (A-RETURN-EXTERNAL), i.e., the control flow leaves the box after the return, we have to ensure that the return value is not confined and we have to expose it. Like in the case for calling a method on an external object, the outside world may continue the execution with arbitrary callbacks or directly return into the component again. If the return value is confined, returning it would break the confinement, therefore we enter the abortion state (A-RETURN-EXTERNAL-ABORT).

The initial state $\mathcal{B}_{init}$ of a box maps the label l_o to the owner object $(\mathtt{B}, \text{false}, \text{true}, \mathtt{f} \mapsto \overline{\mathtt{null}})$, all other labels to placeholders and all variables of the stack to an empty set of values, and `this` has the value extRef. Thus, the callback rule can be executed as the first reduction. $\mathcal{B}_{init}$ is the abstraction of all valid concrete states in which the owner has been created, but no method has been called on it yet.

3.3 Properties of the abstract semantics

We can show that this abstract semantics is consistent with the concrete semantics, meaning that whenever a concrete program state σ can be reduced with a statement S to a state σ', the corresponding abstract state $\mathcal{B}$ can be abstractly reduced with the same statement to a set of abstract box states $\mathcal{B}'$ and the resulting abstract states are abstractions of the resulting concrete states. This gives us the well-known picture of abstract interpretations:

$$\begin{array}{ccc} \overline{\sigma} & \xrightarrow{\Rightarrow} & \overline{\sigma'} \\ \alpha \downarrow & & \mid\cap \\ & & \uparrow \gamma \\ \overline{\mathcal{B}} & \xrightarrow[\Rightarrow_a^*]{} & \overline{\mathcal{B}'} \end{array}$$

THEOREM 1 (Consistency). *Let $(\mathcal{P}_\sigma, \alpha, \gamma, \mathcal{P}_\mathcal{B})$ be the Galois connection from 3.1. The abstract transition function $\Rightarrow_a^*$ is* consistent *with the concrete transition function $\Rightarrow$, i.e. for all $\overline{\sigma} \in \mathcal{P}_\sigma$ and statements S*

$$\gamma(\overline{(\mathcal{B}')} \supseteq' \overline{\sigma}' \text{ and } \alpha(\overline{\sigma}), S \Rightarrow_a^* \overline{(\mathcal{B}', S')}$$

holds, where $\overline{\sigma}'$ are the resulting states of executing S on $\overline{\sigma}$

$$\overline{\sigma}' = \{\sigma_i' \mid \forall \sigma_i \in \overline{\sigma}, \sigma_i, S \Rightarrow \sigma_i', S'\}$$

A-CALL

$$\frac{\overline{v_a} = \mathcal{S}_a(\mathtt{y}) \setminus \{\mathtt{null}\}}{((\mathcal{H}_a, \mathcal{S}_a),\ \mathtt{x = y.m(\overline{x});}) \Rightarrow_a \cup_i \{(\mathcal{H}_a, \mathcal{S}_a),\ \mathtt{x} = v_{ai}\mathtt{.m(\overline{x});}\}}$$

A-CALL-INTERNAL

$$\frac{v_a \neq \text{extRef} \qquad \mathtt{C} = \text{type}(\mathcal{H}(v_a)) \qquad \overline{\mathtt{p}} = \text{params}(\mathtt{C}, \mathtt{m}) \qquad \mathcal{S}'_a = [\mathtt{this} \mapsto v_a, \overline{\mathtt{p}} \mapsto \mathcal{S}_a(\overline{\mathtt{p}}) \cup \mathcal{S}_a(\overline{\mathtt{x}})]\mathcal{S}_a}{((\mathcal{H}_a, \mathcal{S}_a),\ \mathtt{x} = v_a\mathtt{.m(\overline{x});}) \Rightarrow_a \{((\mathcal{H}_a, \mathcal{S}'_a),\ \text{body}(\mathtt{C}, \mathtt{m})\ \ \mathtt{x = retVal_{Cm};})\}}$$

A-CALL-EXTERNAL

$$\frac{\text{confined}((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}_j) = \text{false for all } \mathtt{x}_j \in \overline{\mathtt{x}} \qquad \mathcal{H}'_a = \text{expose}((\mathcal{H}_a, \mathcal{S}_a), \overline{\mathtt{x}}) \qquad \mathcal{S}'_a = [\mathtt{this} \mapsto \text{extRef}]\mathcal{S}_a}{((\mathcal{H}_a, \mathcal{S}_a),\ \mathtt{x = extRef.m(\overline{x});}) \Rightarrow_a \{(\mathcal{H}'_a, \mathcal{S}'_a),\ \mathtt{x = extRef;}\}}$$

A-CALL-EXTERNAL-ABORT

$$\frac{\text{confined}((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}_j) = \text{true for some } \mathtt{x}_j \in \overline{\mathtt{x}}}{((\mathcal{H}_a, \mathcal{S}_a),\ \mathtt{x = extRef.m(\overline{x});}) \Rightarrow_a \top_{abort}}$$

A-CALLBACK

$$\frac{\mathcal{S}_a(\mathtt{this}) = \text{extRef} \qquad \bar{l} = \text{exposed}(\mathcal{H}_a) \qquad \overline{(S_j, \overline{\mathtt{p}}_j)} = \text{methods}(\text{type}(\mathcal{H}_a(l_i))) \qquad \mathcal{S}^j_a = [\mathtt{this} \mapsto l_i, \overline{\mathtt{p}}_j \mapsto \text{extRef} \cup \mathcal{S}_a(\overline{\mathtt{p}}_j)]\mathcal{S}_a \text{ for all } j}{((\mathcal{H}_a, \mathcal{S}_a),\ \mathtt{x = extRef;}) \Rightarrow_a \cup_{l_i \in \bar{l}} \{((\mathcal{H}_a, \mathcal{S}^j_a),\ S_j\ \ \mathtt{x = extRef;})\} \cup \{((\mathcal{H}_a, \mathcal{S}_a),\ \mathtt{return\ extRef;\ x = extRef;})\}}$$

A-RETURN-INTERNAL

$$\frac{\mathcal{S}^i_a = [\mathtt{this} \mapsto \mathtt{this}_i, \mathtt{retVal_{Cm}} \mapsto \mathcal{S}_a(\mathtt{x}_1)]\mathcal{S}_a \qquad \mathtt{m} = \text{inMethod}(j) \qquad \overline{\mathtt{this}} = \{l \mid \mathcal{H}_a(l) \text{ not a placeholder object and } \mathtt{m} \in \text{methods}(\text{type}(\mathcal{H}_a(l)))\}}{((\mathcal{H}_a, \mathcal{S}_a),\ [\mathtt{return\ x_1;}]^j\ \ \mathtt{x_2 = retVal_{Cm};}) \Rightarrow_a \cup_{\mathtt{this}_i \in \overline{\mathtt{this}}} \{((\mathcal{H}_a, \mathcal{S}^i_a),\ \mathtt{x_2 = retVal_{Cm};})\}}$$

A-RETURN-EXTERNAL

$$\frac{\text{confined}((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}_1) = \text{false} \qquad \mathcal{H}'_a = \text{expose}((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}_1) \qquad \mathcal{S}'_a = [\mathtt{this} \mapsto \text{extRef}]\mathcal{S}_a}{((\mathcal{H}_a, \mathcal{S}_a),\ \mathtt{return\ x_1;\ x_2 = extRef;}) \Rightarrow_a \{((\mathcal{H}'_a, \mathcal{S}'_a),\ \mathtt{x_2 = extRef;})\}}$$

A-RETURN-EXTERNAL-ABORT

$$\frac{\text{confined}((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}_1) = \text{true}}{((\mathcal{H}_a, \mathcal{S}_a),\ \mathtt{return\ x_1;\ x_2 = extRef;}) \Rightarrow_a \top_{abort}}$$

Figure 8. Call and Return Rules of the Abstract Semantics

Knowing that $\Rightarrow^*_a$ is consistent with the concrete semantics, we can directly derive the confinement criterion, which enables a modular analysis of confinement.

COROLLARY 2 (Confinement Criterion). *If the abortion state $\top_{abort}$ is not reachable from the initial state $\mathcal{B}_{init}$, a box class is* encapsulated.

The following section describes an analysis based on the abstract interpretation to check for confinement of boxes.

4. Reachable State Analysis and Implementation

To check if a box is encapsulated, we use a reachable state analysis based on the presented abstract semantics. For the analysis, we use a dynamic flow graph representation of the box class and its codebase. This simplifies the definition of the analysis, because we do not have to explicitly deal with configurations containing statements. The relation *flow* contains the pair of labels (l, l') if the statement at label l' can be executed after the statement at l. The label 0 is the label for the MGC. Due to exposing objects and subtyping, new pairs can be added to the relation during the analysis.

Using *flow* we can define for each label a set of entry and exit states. Exit states are the results of executing a reduction step on the entry states and entry states are the exit states of the previous statements. This gives us the following equation system to solve:

$$\begin{array}{lcl} entry(0) & = & \{\mathcal{B}_{init}\} \bigcup \{exit(l') | (l', 0) \in \textit{flow}\} \\ exit(0) & = & entry(0) \\ entry(l) & = & \bigcup \{exit(l') | (l', l) \in \textit{flow}\} \\ exit(l) & = & \overline{\mathcal{B}} \text{ with } entry(l), \text{statement at } l \Rightarrow^*_a \overline{\mathcal{B}} \end{array}$$

Because the state space is finite and the abstract semantics never removes values from fields or variables, a least fixpoint for these sets exists.

To check the confinement, we calculate all entry and exit sets. If the abortion state $\top_{abort}$ is not part of any set, the box is encapsulated. If some entry or exit set contains $\top_{abort}$, the component is either not encapsulated or the abstraction is not detailed enough to produce a precise result.

We have written a prototypical implementation of the analysis and experimented with different examples. The implementation is a straightforward translation of the abstract reduction function and a fix point iteration for the equation system into about 1200 lines of Haskell. Although the implementation has not been optimized and does not use all possibilities given by the definition of $\subseteq'$ to reduce the state space, it runs quite fast on our examples. In the future we will study bigger components and see how far the implementation scales. The tool can write graph representations of all states in the entry and exit sets to files. The graphs are automatically laid out by graphviz and can help the programmer to find bugs which prevent encapsulation. Figure 9 shows an example of such a graph, to fit on the paper an extra line break in the stack has been inserted into

the automatically generated layout. Currently, programs have to be translated into abstract syntax by hand before the analysis can handle them, but we are thinking of an implementation for Java bytecode.

The language supported by the tool contains an additional confinement feature. Methods can be declared box local. Such local methods cannot be called across box boundaries. This restricts the set of callable methods on exposed objects, and allows e.g. to hide methods that may break box invariants.

5. Related Work

To statically guarantee the confinement of objects at runtime, ownership types ([3, 6, 11, 13, 14] and others) have been developed. Ownership type systems impose the owners-as-dominators discipline, i.e. all accesses to encapsulated objects are done via the owner object, and in contrast to our approach do not support multi-object interfaces for ownership contexts like the iterator pattern. To handle this problem, extensions to the ownership types have been proposed. One is to use an owners-as-mutators discipline like the Universe type system [18] does. This discipline allows to expose multiple objects for each ownership context, but all modifications of the owned objects have to be done via the owner. This approaches cannot handle patterns, in which boundary objects directly modify the representation objects. Other proposals allow to store dynamic aliases to owned objects on the stack [12] or allow Java's inner member classes to access encapsulated objects of their parent classes [6].

Ownership domains (OD) [2] are a generalization of the ownership types systems. Objects are not directly owned by other objects, but by domains, which in turn are owned by objects. Each object can have an arbitrary number of domains, which can be either public or private. Objects in public domains can be accessed by other domains. This is similar to our approach with boundary and confined objects, which can be seen as two domains with a fixed accessibility relation between them, whereas in OD the programmer can use link declarations to control accessibility. OD gives the programmer a very fine-grained control to express the confinement, but needs a lot of annotations. We think, that for most components distinguishing confined and boundary objects together with a fixed accessibility relation is powerful enough.

To lower the annotation burden, type inference algorithms for ownership type systems and OD have been developed. One kind of these algorithms works on un-annotated code and infers the ownership structure by static analysis [3, 17] or by observing the runtime behavior [19]. We think, while this approach helps understanding un-annotated code, it is not well suited for new code, because the programmer does not express the desired ownership structure directly, but has to check if the inferred one matches his desire. Other inference algorithms take partially annotated code and propagate the annotations through the rest of the system [1, 7, 25, 27]. While this approach lowers the annotation burden significantly, the programmer still has to understand the underlying type system, which is a challenging task for average programmers. We think that our approach is easier to employ.

Several works have extended the generic type system of Java to express ownership information [16, 30, 31]. This approach integrates well with the existing type system, but gives the programmer an even more complex type system to handle than the generic one already is.

An annotation free ownership system based on virtual classes is presented in [8], it uses the lexically nesting of virtual classes as ownership hierarchy. While this approach looks interesting for languages with virtual classes, which use nested inner classes to describe families of classes, because it comes with very low cost, it is not suited for languages like Java with a nearly flat class structure.

The box model presented in this paper is a simpler version of the ones in [27–29]. These papers describe box models that support hierarchical box structures and the confinement is guaranteed by an ownership domain type system, whose annotations can partially be inferred.

The fundamental techniques for modular static analysis and a classification for different kinds of analysis are described in [15]. Our analysis fits into the category of worst-case analysis. The idea of abstracting the context of a component to the most general client has been used in [32] before. The allocation site abstraction for heaps [9, 23] has been used in numerous points-to-analysis [10, 33, 34] and others.

Close to our work are the ideas in [26]. This work presents a framework using abstract interpretation to infer class invariants. The used abstraction and the resulting semantics are similar to our setting if all classes are box classes. It also shows how the abstract semantics can be split into the context part and the part inside the object and approximates the interactions between an object and its context by regular expressions, whereas we only support the most general client. It would be interesting to refine the most general client in a similar way, because this would certainly lead to a more precise analysis.

The encapsulation property allows to control the escaping of references to confined objects. This is similar to the goals of escape analysis, which try to detect objects that do not leave a certain scope. In [21, 22] an escape analysis based on abstract interpretation is given. The analysis guarantees that an object does not escape from a method or a thread. The difference to our work is that our confinement border is given by the structure of the heap and evolves during program execution, whereas theirs is determined mainly by the stack. Like in our work, they use an object allocation abstraction for the heap. Similar works have been presented in [5].

6. Conclusion and Future Work

In this paper we presented a modular, static analysis for checking the confinement property of the box model. Using the defined abstraction we get a reachable state analysis for a box in the context of the most general client. If no box used in a program reaches the abortion state in the analysis, the whole program fulfills the confinement property.

Even if the abstraction removes a lot of information about object identity and the most general client creates a lot of impossible traces through the box, we think that this analysis is powerful enough for a lot of components. We implemented a first prototype of the analysis, which substantiates this assumption. In the future we will explore bigger examples. It will be interesting to study, if e.g. recency-abstractions for heaps [4] or the techniques described in [24] give significantly better results for components as they appear in the real code.

In the future, we will extend the language with hierarchical boxes, such that patterns like sets using a list component as internal representation can be expressed in a more natural way. Furthermore, we may extend the language with annotations for fields and variables. This would enable us to provide better feedback to the programmer by comparing the annotations with the calculated states, or we could use the annotations to get a more precise analysis, which is sound under the assumption that the invariants imposed by the annotations are satisfied.

References

[1] R. Agarwal and S.D. Stoller. Type inference for parameterized race-free java. In *Verification, Model Checking, and Abstract Interpreta-*

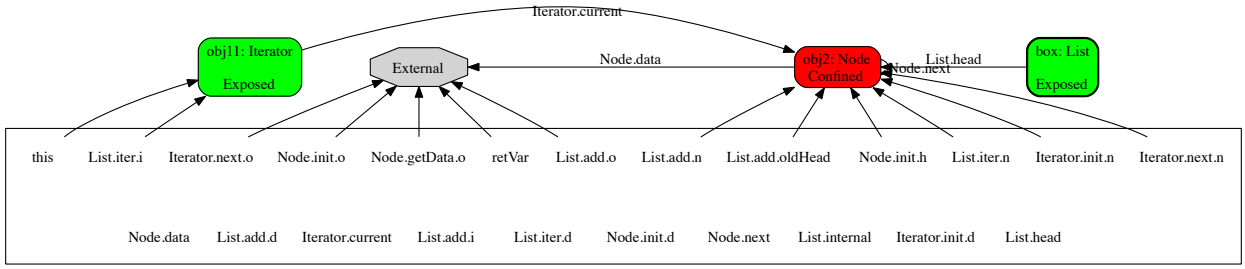

Figure 9. A state of the exit set at the end of `Iterator.next()` of a list component

tion, pages 77–108. Springer, 2003.

[2] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. *ECOOP 2004*, pages 1–25, 2004.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA '02*, pages 311–330, 2002.

[4] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. *Static Analysis*, pages 221–239, 2006.

[5] R. Barbuti and St. Cataudella. Abstract interpretation of an object calculus for synchronization optimizations. *Fundam. Inf.*, 67(1-3):1–12, 2005.

[6] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. *ACM SIGPLAN Notices*, 38(1):213–223, 2003.

[7] C. Boyapati and M. C. Rinard. A parameterized type system for race-free java programs. In *OOPSLA '01*, pages 56–69, 2001.

[8] N. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *OOPSLA '10*, pages 618–633. ACM, 2010.

[9] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI '90*, pages 296–310. ACM, 1990.

[10] B.C. Cheng and W.M.W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *ACM SIGPLAN Notices*, volume 35, pages 57–69. ACM, 2000.

[11] D. Clarke. *Object ownership and containment*. PhD thesis, School of Computer Science and Engineering University of New South Wales, Australia, 2001.

[12] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM SIGPLAN Notices*, volume 37, pages 292–310. ACM, 2002.

[13] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. *ECOOP 2007*, pages 53–76, 2001.

[14] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 33(10):48–64, 1998.

[15] P. Cousot and R. Cousot. Modular static program analysis. In *CC '02*, volume 2304 of *LNCS*, pages 159 – 178, London, UK, 2002. Springer-Verlag.

[16] W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. *ECOOP 2007*, pages 28–53, 2007.

[17] W. Dietl, M. Ernst, and P. Müller. Tunable universe type inference. Technical Report 659, Department of Computer Science, ETH Zurich, 2009.

[18] W. Dietl and P. Müller. Universes: Lightweight ownership for jml. *Journal of Object Technology*, 4(8):5–32, 2005.

[19] W. Dietl and P. Müller. Runtime Universe Type Inference. In *IWACO '07*, July 2007.

[20] Kathrin Geilmann. Ensuring confinement of object-oriented components using abstract interpretation. Available at `http://softech.cs.uni-kl.de/Homepage/PublikationsDetail?id=154`.

[21] P. M. Hill and F. Spoto. A foundation of escape analysis*. *Algebraic Methodology and Software Technology*, pages 5–11, 2002.

[22] P. M. Hill and F. Spoto. Deriving escape analysis by abstract interpretation: Proofs of results. *CoRR*, abs/cs/0607101, 2006.

[23] N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL '82*, pages 66–74. ACM, 1982.

[24] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA '10*, pages 411–427. ACM, 2010.

[25] Y.D. Liu and S. Smith. Pedigree types. In *IWACO '08*, 2008.

[26] F. Logozzo. *Modular static analysis of object-oriented languages*. PhD thesis, École polytechnique, 2004.

[27] A. Poetzsch-Heffter, K. Geilmann, and J. Schäfer. Infering ownership types for encapsulated object-oriented program components. In *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, LNCS, pages 120–144. Springer, 2007.

[28] A. Poetzsch-Heffter and J. Schäfer. Modular specification of encapsulated object-oriented components. *Lecture Notes in Computer Science*, 4111:313, 2006.

[29] A. Poetzsch-Heffter and J. Schäfer. A representation-independent behavioral semantics for object-oriented components. In M. Bonsangue and E. Johnsen, editors, *FMOODS '07*, volume 4468 of *Lecture Notes in Computer Science*, pages 157 – 173. Springer, 2007.

[30] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Featherweight generic ownership. In *FTfJP '05*, 2005.

[31] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic java. In *OOPSLA '06*, pages 311–324. ACM, 2006.

[32] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *ESOP'07*, LNCS. Springer, March 2007.

[33] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[34] J. Whaley and M.S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04*, pages 131–144. ACM, 2004.