Ensuring Confinement of Object-Oriented Components using Abstract Interpretation

Kathrin Geilmann

1 Introduction

Component-based software systems are build out of a set of program components, which are linked together or from which new components are build. A key issue in component-based programming is to guarantee that components are used as intended by their developer. Usually, there are two assumptions in component-based programming:

- The component developer assumes that the internal component state is not modified by the other components.
- The component user assumes that he can freely use every reference or value the component returns.

If one of them is disregarded, the behavior of the component is not defined anymore. Current object-oriented programming languages only provide a very limited support, if one wants to guarantee these assumptions. Therefore we developed the box model, a component model for the object-oriented world. The box model defines an encapsulation property, which corresponds to the assumptions above.

In this paper we will present an analysis, which decides wether a set of classes implements a component according to the box model and therefore ensures the encapsulation property or not. As we are in the component-oriented programming world, the analysis will work on single components. This has the advantage that the component developer can give guarantees about his component and that the user can put together components to systems and does not need to take care of preserving internal invariants of the used components.

2 Language

2.1 The Box Model

In this paper we use a simplified version of the box model presented in [7]. The box model extends the object-oriented programming world of classes, objects, references to objects, object-local state and methods with the notion of the box.

Similar to an object, a box is a runtime entity which is created, has an identity and a state. In general, it groups several objects together. Its state is composed of the states of the contained objects. Together with a box an owner object for the box is created, such that a box always contains at least one object. The objects in a box are either confined to it or are part of the box boundary. The access to a confined object is only allowed for objects in the same box, whereas boundary objects may be accessed from everywhere. The owner is always part of the box boundary.

The most important property of the model is, that confined objects stay confined to the box and no reference to these objects will ever leave the box. We call this the *encapsulation property*. Additionally, the model allows to restrict the methods that can be called on an object. On objects of a different box only methods declared as public can be called, the local methods are callable only inside the box. This guarantees that all accesses to the box state are done via public methods of the boundary objects and therefore their effects are under the control of the box instance.

2.2 Language Description

For our analysis we use a small object-oriented Java-like language, its abstract syntax is given in Figure 1 and an example program for a component with a client using this component in Figure 2. Note for readability Figure 2 uses a more convenient syntax than the given abstract syntax.

A program consists of a set of class declarations and a main-statement, which creates the first object and calls a method `m` on it. We only allow box classes to be instantiated in the main statement. As instantiating a box class always instantiates a box, we therefore always have at least one box. The parameters of the method `m` are considered to be the program inputs.

A class declaration consists of field and method declarations. To simplify the language, we support fields accesses only on `this`. Methods can be public or local. Local methods can be called by objects of the same box, whereas public methods may be called by any object. The class declaration also contains a modifier, which is either `box` or `helper`. If it is `box`, the instantiation of this class results in the creation of a new box and of a new object. The newly created object is the owner of the new box. Like in Java we have `Object` as the superclass of every class, but our `Object` class does not have any methods or fields. Subclasses inherit the methods from their superclasses and they can override them, but for simplicity we do not have `super`.

Simple statements and the return statements are labeled. These labels have to be unique over the whole program. We will need them later, in order to create the flow

P	:=	$\overline{CD}$ `{C x; x = new C(); x.m(`$\overline{c}$`)}`	programs, `C` has to be box class
CD	:=	cm `class C [extends C]{`$\overline{T}$ $\overline{\texttt{f}}$`;` $\overline{M}$`}`	class declaration
M	:=	mm T `m (`$\overline{T}$ $\overline{\texttt{x}}$`) {`$\overline{T}$ $\overline{\texttt{x}}$`;` S`; [return x]`^{l}`}`	method declaration
S	:=	S`;`S	
	|	`[x =` e`]`^{l}	assignment to local variable
	|	`[x = new` nm `C()]`^{l}	object creation
	|	`[x = x.m(`$\overline{\texttt{x}}$`)]`^{l_1,l_2}	method call
	|	`[x = this.f]`^{l}	field read
	|	`[this.f =` e`]`^{l}	field update
	|	`[if (x) {`S`} else {`S`}]`^{l}	
e	:=	`x`	variable read
	|	`null`	`null` constant
T	:=	`C`	types
cm	:=	`box` | `helper`	class modifier
nm	:=	`confined` | `boundary`	creation modifier
mm	:=	`public` | `local`	method modifier
l	$\in$	L (unique labels)	
`C`	$\in$	class names	
`x`	$\in$	variable names	
`f`	$\in$	field names	
`m`	$\in$	method names	

Figure 1: Abstract syntax of our language

```
box class List {
    Node head;

    void add(Object o) {
        Node t = head;
        head = new confined Node();
        head.init(t,o);
    }

    Iterator iter() {
        Iterator i = new Iterator();
        i.init(head);
        return i;
    }
}

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

class Iterator {
    Node current;

    void init (Node n) {
        current = n;
    }

    Object next() {
        Node t = current;
        current = current.next;
        return t.value;
    }
}

box class ListClient {
    List all;
    List even;

    void main() {
        all = new List();
        all.add(new Data(1));
        ...
        all.add(new Data(382));
        even = filter(all);
        // the following call is not allowed,
        // the created object is confined to
        // the list client and cannot leak
        // into the List Box
        even.add(new confined Data(16));

    }

    List filter(List l) {
        List l2 = new List();
        Iterator i = l.iter();
        Data d = i.next();
        while (d != null) {
            if (d.x() % 2 == 0) l2.add(d);
        }
    }
}


class Data {
    int x;

    void init(int x, Object o) {
        this.x = x;
    }

    int x() {
        return x;
    }
}

// main statement
ListClient lc = new ListClient(); lc.main();
```

Figure 2: An implementation of a multiple components

σ	::=	$(\mathcal{H}, \mathcal{S}) \mid \top_{abort}$	program state
$\mathcal{H}$	::=	$o_{id} \mapsto o$	heap
$\mathcal{S}$	::=	$\overline{(\texttt{this} \mapsto o_{id}, \overline{\mathrm{x}} \mapsto \overline{v})}$	stack
o	::=	$(o_{id}, \mathtt{C}, \text{confined}, \text{exposed}, \bar{\mathrm{f}} \mapsto \overline{v})$	object
v	::=	$o_{id} \mid \texttt{null}$	values

o_{id}	object identifiers
f	fields of an object
x	local variables and method parameters
C	type of the object
confined	boolean flag for the confinement status
exposed	boolean flag for the exposed status

Figure 3: Concrete program states

graph of a program, which will be used for the encapsulation analysis. The new statement takes a modifier defining whether a reference to the new object is allowed to leave the box (nm = boundary) or not (nm = confined). In contrast to [7], we do not support nested boxes, so the modifier has to be boundary, if a box class is instantiated. This restriction simplifies the analysis later on.

The other statements and expressions work as usual, except that

- calling a local method on an object in another box, or
- trying to return references to confined objects, or passing references of confined object to objects in other boxes

aborts the program execution, because this would break the encapsulation properties of the box model.

In the following parts of the paper we will develop an analysis, which statically guarantees, that the program execution will not abort due to a violation of the encapsulation properties.

3 Concrete Semantics

In this chapter, we will present the concrete small step semantics of our language. The semantics guarantee, that the encapsulation property given in Section 2.1 is always fulfilled. To check the encapsulation property, we need to keep track of which object belongs to which box. For readability we omit the labels of the statements most of the time.

3.1 Concrete State

For the semantics we first define the concrete program state σ (see Figure 3). The program state is a pair of a heap and a stack. The heap is a mapping from unique object identifiers to objects, each of them being a tuple of an object identifier referencing the owner of the box, to which the object belongs, the class of the object, a boolean flag for the confinement status and a mapping of the object fields to values. Objects of box classes reference themselves as box owners. The reference to the box owner allows to decide if two objects are part of the same box. The stack consists of a list of stack frames which store a reference to the current `this`-object and a mapping of all local variables and parameters to values. Values in our language can be either a reference to an object or the `null`-reference. To access values on the topmost stack frame f of a stack we use the following notation: $\mathcal{S}(\mathtt{x}) := f(\mathtt{x})$ if $\mathcal{S} = f \cdot \mathcal{S}'$ and to update values we use $[\mathtt{x} \mapsto v]\mathcal{S} := [\mathtt{x} \mapsto v]f \cdot \mathcal{S}'$ if $\mathcal{S} = f \cdot \mathcal{S}'$.

Additionally we define an abortion state $\top_{abort}$. This state is used to signal the occurrence of an error during program execution. It is impossible to leave the abortion state.

A concrete program state is called *valid*, if it is a reachable state in some program. The set of all valid concrete program states is called S_σ.

The initial program state for a program $\overline{CD}$ `{C x; x = new C(); x.m(`$\overline{c}$`)}` consists of an empty heap and an empty stack.

3.2 Auxiliary Functions

For the definition of the concrete semantics we need first define some sets and auxililary functions.

Expressions are evaluated by the function eval. It returns `null` for the expression `null` and for a variable it returns the value of the variable in the topmost stackframe.

The function boxclass returns whether a class is declared as a box class or not.

$$\frac{\ldots\ \texttt{box class C}\ \ldots}{\text{boxclass}(\texttt{C}) = \text{true}} \qquad \frac{\ldots\ \texttt{helper class C}\ \ldots}{\text{boxclass}(\texttt{C}) = \text{false}}$$

The function local returns whether a methods is declared local or not.

$$\frac{\texttt{local}\ T\ \texttt{m}\ (\overline{T}\ \overline{\texttt{x}})\ldots}{\text{local}(\texttt{m}) = \text{true}} \qquad \frac{\texttt{public}\ T\ \texttt{m}\ (\overline{T}\ \overline{\texttt{x}})\ldots}{\text{local}(\texttt{m}) = \text{false}}$$

The function body gives us the statements in the body of a method `m`. It gets as parameters the type of the object, on which `m` shall be called.

$$\frac{\ldots\ \texttt{class C}\ \ldots\ mm\ T\ \texttt{m}\ (\overline{T}\ \overline{\texttt{x}})\ \{\overline{T}\ \overline{\texttt{x}};\ S;\ \texttt{return x}\}\ldots}{\text{body}(\texttt{C}, \texttt{m}) = S;\ \texttt{return x}}$$

$$\frac{\ldots\ \texttt{class B}\ \ldots\ mm\ T\ \texttt{m}\ (\overline{T}\ \overline{\texttt{x}})\ \{\overline{T}\ \overline{\texttt{x}};\ S;\ \texttt{return x}\}\ldots \quad B \text{ is the smallest supertype of C, which defines m}}{\text{body}(\texttt{C}, \texttt{m}) = S;\ \texttt{return x}}$$

To manage the creation of object identifiers we assume two functions. The function fresh takes the label of the new-statement and returns a new unique object identifier and the function cs takes an object identifier and returns the label of the creation statement.

We define some functions to access the different parts of an object. Be $o = (o_{id}, \mathsf{C}, b, e, \overline{\mathsf{f}} \mapsto \overline{v})$ an object, then

$$\begin{array}{ll} \text{owner}(o) & = o_{id} \\ \text{type}(o) & = \mathsf{C} \\ \text{confined}(o) & = b \\ \text{isExposed}(o) & = e \\ o(\mathsf{f}) & = v \end{array}$$

To access the current this-object in a state $\sigma = (\mathcal{H}, \mathcal{S})$ we use the function currentThis$(\sigma) = \mathcal{H}(\mathcal{S}(\mathtt{this}))$.

To determine if two variables reference objects in the same box we define the function samebox.

$$\frac{owner(\mathcal{H}(\mathcal{S}(\mathsf{x}_1))) = owner(\mathcal{H}(\mathcal{S}(\mathsf{x}_2))}{\text{samebox}((\mathcal{H}, \mathcal{S}), \mathsf{x}_1, \mathsf{x}_2) = \text{true}} \qquad \frac{owner(\mathcal{H}(\mathcal{S}(\mathsf{x}_1))) \neq owner(\mathcal{H}(\mathcal{S}(\mathsf{x}_2))}{\text{samebox}((\mathcal{H}, \mathcal{S}), \mathsf{x}_1, \mathsf{x}_2) = \text{false}}$$

Whenever a call crosses the boundaries of boxes we have to expose the objects. This is done by the function expose, which takes a set of variables and a state and returns a heap, in which all objects references by the variables have their exposed flag set.

3.3 Concrete Transition Relation

Program execution is controlled by the transition relation $\Rightarrow$, which is quite standard except for the checks of the encapsulation. The occurence of an encapsulation error leads to the abortion state. Normal program errors like dereferencing null let the execution get stuck. The relation has the following forms:

- $\sigma, S \Rightarrow \sigma', S'$: the execution of the statement S in the state σ is not yet completed and σ', S' expresses the remaining execution.
- $\sigma, S \Rightarrow \sigma'$: the execution of statement S in σ has terminated and σ' is the final state.

In the following, the rules for the statements are presented. Sequences of statements are executed one after the other.

$$\text{SEQ-1}\ \frac{\sigma, S_1 \Rightarrow \sigma'}{\sigma, S_1; S_2 \Rightarrow \sigma', S_2} \qquad \text{SEQ-2}\ \frac{\sigma, S_1 \Rightarrow \sigma', S_1'}{\sigma, S_1; S_2 \Rightarrow \sigma', S_1'; S_2}$$

An assignment to a variable changes only the topmost stack frame.

$$\text{ASSIGN}\ \frac{\mathcal{S}' = [\mathsf{x} \mapsto \text{eval}(e, \mathcal{S})] \cdot \mathcal{S}}{(\mathcal{H}, \mathcal{S}), \mathsf{x}\ \mathtt{=}\ e \Rightarrow (\mathcal{H}, \mathcal{S}')}$$

New helper objects are created on the heap and their reference is stored in a variable of the stack. The fields of the new object are set to `null`. New helper objects are always created in the same box as the current `this`-object.

NEW

$$\frac{\neg\text{boxclass}(\texttt{C}) \qquad b_{id} = \text{owner}(\text{currentThis}((\mathcal{H},\mathcal{S})) \qquad o_{id} = \text{fresh}(l) \\ o_{new} = (b_{id}, \texttt{C}, \text{false}, \bar{\texttt{f}} \mapsto \overline{\texttt{null}}) \qquad \mathcal{H}' = [o_{id} \mapsto o_{new}] \cdot \mathcal{H} \qquad \mathcal{S}' = [\texttt{x} \mapsto o_{id}] \cdot \mathcal{S}}{(\mathcal{H},\mathcal{S}), [\texttt{x = new C()}]^l \Rightarrow (\mathcal{H}',\mathcal{S}')}$$

NEW-CONF

$$\frac{\neg\text{boxclass}(\texttt{C}) \qquad b_{id} = \text{owner}(\text{currentThis}((\mathcal{H},\mathcal{S})) \qquad o_{id} = \text{fresh}(l) \\ o_{new} = (b_{id}, \texttt{C}, \text{true}, \bar{\texttt{f}} \mapsto \overline{\texttt{null}}) \qquad \mathcal{H}' = [o_{id} \mapsto o_{new}] \cdot \mathcal{H} \qquad \mathcal{S}' = [\texttt{x} \mapsto o_{id}] \cdot \mathcal{S}}{(\mathcal{H},\mathcal{S}), [\texttt{x = new confined C()}]^l \Rightarrow (\mathcal{H}',\mathcal{S}')}$$

Creating box objects is similar to creating non-box objects, but the owner is set to the newly created object, because the instantiation of box objects always creates a corresponding box owned by the new object.

NEW-BOX

$$\frac{\text{boxclass}(\texttt{C}) \qquad o_{id} = \text{fresh}(l) \\ o_{new} = (o_{id}, \texttt{C}, \text{false}, \bar{\texttt{f}} \mapsto \overline{\texttt{null}}) \qquad \mathcal{H}' = [o_{id} \mapsto o_{new}] \cdot \mathcal{H} \qquad \mathcal{S}' = [\texttt{x} \mapsto o_{id}] \cdot \mathcal{S}}{(\mathcal{H},\mathcal{S}), [\texttt{x = new C()}]^l \Rightarrow (\mathcal{H}',\mathcal{S}')}$$

Reading and updating fields are standard.

READ

$$\frac{\mathcal{S}' = [\texttt{x} \mapsto \mathcal{H}(\mathcal{S}(\texttt{this}))(\texttt{f})] \cdot \mathcal{S}}{(\mathcal{H},\mathcal{S}), \texttt{x = this.f} \Rightarrow (\mathcal{H},\mathcal{S}')}$$

UPDATE

$$\frac{v = \text{eval}(e) \qquad o' = [\texttt{f} \mapsto v] \cdot \mathcal{H}(\mathcal{S}(\texttt{this})) \qquad \mathcal{H}' = [\mathcal{S}(\texttt{this}) \mapsto o'] \cdot \mathcal{H}}{(\mathcal{H},\mathcal{S}), \texttt{this.f = } e \Rightarrow (\mathcal{H}',\mathcal{S})}$$

The `if`-statement just continues with the correct branch.

IF-TRUE

$$\frac{\text{eval}(\mathcal{S}(\texttt{x})) \neq \texttt{null}}{(\mathcal{H},\mathcal{S}), \texttt{if (x)}\{S_1\}\texttt{else }\{S_2\} \Rightarrow (\mathcal{H},\mathcal{S}), S_1}$$

IF-FALSE

$$\frac{\text{eval}(\mathcal{S}(\texttt{x})) = \texttt{null}}{(\mathcal{H},\mathcal{S}), \texttt{if (x)}\{S_1\}\texttt{else }\{S_2\} \Rightarrow (\mathcal{H},\mathcal{S}), S_2}$$

To handle calls we assume a special variable `retVal`, that serves as a placeholder for the return value of a method. Calls to objects in the same box are not restricted, so we

just construct a new stack frame and put onto the stack. The new frame contains the values for the parameters $\mathtt{p}_i$, and all local variables $\mathtt{y}_k$ are set to `null`. The new `this` is the receiver of the call. We continue the execution with the method body and upon return we assign the return value.

CALL-INTERNAL

$$\frac{\text{samebox}((\mathcal{H},\mathcal{S}),\mathtt{this},\mathtt{x}_2) \qquad \mathcal{S}' = [\overline{\mathtt{p}}_i \mapsto \overline{\mathcal{S}(\mathtt{x}_i)}, \overline{\mathtt{y}}_k \mapsto \overline{\mathtt{null}}, \mathtt{this} \mapsto \mathcal{S}(\mathtt{x}_2)] \cdot \mathcal{S}}{(\mathcal{H},\mathcal{S}), \mathtt{x}_1 \texttt{ = } \mathtt{x}_2\texttt{.m(}\overline{\mathtt{x}}_i\texttt{)} \Rightarrow (\mathcal{H},\mathcal{S}'), \text{body}(\mathtt{m},\mathcal{S}(\mathtt{x}_2)); \mathtt{x}_1 \texttt{ = retVal}}$$

When calling a method on an external object, we have to check that it is not a local method and that no confined object is used as parameters, because this would reveal it to objects outside its own box.

CALL-EXTERNAL

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

CALL-EXTERNAL-ABORT

$$\frac{\neg\text{samebox}((\mathcal{H},\mathcal{S}),\mathtt{x}_1,\mathtt{x}_2) \qquad \text{local}(\mathtt{m}) \vee \text{confined}(\mathcal{H}(\mathcal{S}(\mathtt{x}_i))) = \text{true for some } \mathtt{x}_i}{(\mathcal{H},\mathcal{S}), \mathtt{x}_1 \texttt{ = } \mathtt{x}_2\texttt{.m(}\overline{\mathtt{x}}_i\texttt{)} \Rightarrow \top_{abort}}$$

When returning from a call, put the return value in a variable named `retVal` on the stack.

RETURN-INTERNAL

$$\frac{\text{owner}(\mathcal{H}(f_2(\mathtt{this}))) = \text{owner}(\mathcal{H}(f_1(\mathtt{this}))) \qquad \mathcal{S}' = [\mathtt{retVal} \mapsto f_1(\mathtt{x})]f_2 \cdot \mathcal{S}}{(\mathcal{H}, f_1 \cdot f_2 \cdot \mathcal{S}), \mathtt{return\ x} \Rightarrow (\mathcal{H},\mathcal{S}')}$$

When the execution after the return will continue in a different box, ensure that the return value is not a confined object.

RETURN-EXTERNAL

$$\frac{\begin{array}{c}\text{owner}(\mathcal{H}(f_2(\mathtt{this}))) \neq \text{owner}(\mathcal{H}(f_1(\mathtt{this}))) \qquad \text{confined}(\mathcal{H}(f_1(\mathtt{x}))) = \text{false} \\ \mathcal{H}' = \text{expose}(\overline{\mathtt{x}}, (\mathcal{H}, f_1 \cdot f_2 \cdot \mathcal{S}))\mathcal{S}' = [\mathtt{retVal} \mapsto f_1(\mathtt{x})]f_2 \cdot \mathcal{S}\end{array}}{(\mathcal{H}, f_1 \cdot f_2 \cdot \mathcal{S}), \mathtt{return\ x} \Rightarrow (\mathcal{H}',\mathcal{S}')}$$

RETURN-EXTERNAL-ABORT

$$\frac{\text{owner}(\mathcal{H}(f_2(\mathtt{this}))) \neq \text{owner}(\mathcal{H}(f_1(\mathtt{this}))) \qquad \text{confined}(\mathcal{H}(f_1(\mathtt{x}))) = \text{true}}{(\mathcal{H}, f_1 \cdot f_2 \cdot \mathcal{S}), \mathtt{return\ x} \Rightarrow \top_{abort}}$$

3.4 Encapsulation Properties

First we define the accessibility invariant that holds during the complete execution of the program.

Definition 1 (Accessibility Invariant) *1. Object Access. Objects can only access other objects which are in the same box, or boundary objects.*

2. Methods Calls. For call to local methods, caller and callee belong to the same box.

Using the accessibility invariant we can formulate the encapsulation property as a property about the reachable states of a program.

Corollary 1 (Encapsulation Property) *The accessibility invariant holds iff the state* $\top_{abort}$ *is not reachable from the initial state.*

The proof is trivial, by the construction of the semantics.

4 Analysing Encapsulation Properties

The defined language enforces the encapsulation property at runtime. But for the development of component-based systems, it is desirable to be able to check the property already at compile time. And even further, it is desirable to do modular checks for single components, such that a developer can guarantee, that his component will not break encapsulation regardless of the program it is used in.

In this section, we will develop a modular static analysis, which checks single components for their conformance to the box model. The analysis is done under the assumption, that all other components of the program also conform to the box model.

Conformance to the box model model means, that

- the component will never pass object references of confined objects across the box boundaries, and
- the component will never call local methods on objects not contained in itself.

i.e. the accessibility invariant holds for all objects of the component. We call such a component *boxed*. Therefore a boxed component never tries to compromise other components and it never leaks any confined information. It is easy to see, that for any program consisting only of boxed components the encapsulation property holds.

In the following, we present our analysis for the introduced component model. The analysis is based on an abstract interpretation of the source code of a single component. We use the abstract interpretation and the equation system defined by it to calculate for each program point a set of reachable abstract states. If these sets do not contain the abortion state, which signals a violation of the encapsulation property, the component is boxed. The resulting analysis is sound but incomplete, because we loose information about object identity in order to get finite state spaces.

This section is structured as follows. After the definition of the input of the analysis, we give the abstract state of a box and the Galois connection between the abstract and the concrete states. We then show an abstract semantics for components and prove that it is a safe abstraction. The last subsection describes the equation system defined by the abstract interpretation.

$$
\begin{array}{rcll}
\mathcal{B} & ::= & (\mathcal{H}_a, \mathcal{S}_a) | \top_{abort} & \text{abstract box state} \\
\mathcal{H}_a & ::= & l \mapsto o^a & \text{abstract box heap} \\
\mathcal{S}_a & ::= & \overline{\mathtt{x}} \mapsto \overline{\{v^a\}} & \text{abstract stack} \\
o^a & ::= & (\mathtt{C}, \text{confined}, \text{exposed}, \overline{\mathtt{f}} \mapsto \overline{\{v^a\}}) & \text{abstract object} \\
v^a & ::= & l \mid \texttt{null} \mid \text{extRef} & \text{abstract values}
\end{array}
$$

l	creation sites
f	fields of an object
x	local variables and method parameters
C	Classes of the codebase of the components
$\mathcal{B}^{ex}$	marked abstract box state

Figure 4: Abstract box states

4.1 Analysis Input

The analysis works on the *codebase* of a box. The codebase consists of the box class under analysis and all other classes needed to implement the box i.e. it consists of all sources that can be executed on objects of the component.

The codebase can be build by starting with the box class and transitively adding, the superclasses of the box class, and all helper classes, which occur in `new` statements and their superclasses.

4.2 Abstract State

When a programmer thinks about the state of a component, he usually thinks only about the states of objects belonging to the component. He ignores all the objects not part of it. Our encapsulation analysis does the same by abstracting away all parts of a program state, that do not belong to the currently analyzed component.

For the analysis we have to formally define the abstract state of a box. The abstract heap contains only objects which belong to one box, additionally we keep only one object per creation site. The stack is abstracted to a finite mapping between variable names and set of abstract values. This gives us a finite state space and therefore automatically guarantees the existence of fixpoints and termination of the calculation.

An abstract object is a tuple of the type of the object, a boolean value for the confinement status, a boolean value for the exposed flag, and a mapping of fields to a set of abstract values. We will abstract all concrete objects, that are created at the same code location into a single abstract object. All of these concrete objects have the same type and confinement status. The values of the abstract fields are sets, containing all the values of the corresponding concrete fields. The exposed flag is true, if a reference to the abstract object has left the box. The flag is used to determine the set of methods that can be called on a box in some state. Because every abstract object may stand for multiple concrete objects, the value false for the exposed flag means, that none of the

concrete objects has been exposed, and true means, that one or more of the concrete objects might have been exposed. Obviously, a confined object can never have the exposed flag set to true. As we have at most one abstract object per creation site we can use the creation site as object identifier in the heap and because we now work on a single box the box owner is the same for every abstract object, so we do not need to keep track of it anymore.

The abstract stack is represented as a map of variable names to sets of possible values. We assume that all variables in the codebase of the component have unique names (except `this`). The set of values for `this` always contains only a single abstract object, such that we always know which abstract object currently executes code, otherwise one would have to deal with several possible `this` objects at once, making the execution of calls more difficult and less precise.

The values used in the abstract state can either be a creation site l, `null` or extRef. extRef is used to refer to objects outside of the box. The default value for an abstract field or variable is denoted with $v^a_{default}$ and is `null` for references and v^a_c in all other cases. If the value set of a variable contains extRef, the variable may reference some object not part of the box. Thus, calling local methods on it is no longer safe and will lead to abortion.

An abstract box state is a pair of heap and stack or the special state $\top_{abort}$. Like the conrete abortion state, $\top_{abort}$ is used to signal a violation of the encapsulation properties. The initial state is the state with a stack, that maps all variables to the set {`null`} and an empty heap, we denote it as $\mathcal{B}_{init} = (\emptyset, \emptyset)$. Box states can be marked with ex. This signals, that the component is in the state $\mathcal{B}$, but the control flow is currently outside the component. Marked states are used in the semantics to allow an easy handling of callbacks and returns between components.

The set of all box states is called $B_{\mathcal{B}}$.

4.3 Abstraction- and Concretization-Functions

In this subsection we define a pair of functions (α, γ), which relates the abstract and the concrete states. First, we define a representation function β, which maps a concrete program state to an abstract box state. We will use β to construct α and γ and then show that $(\mathcal{P}(S_\sigma), \alpha, \gamma, \mathcal{P}(B_{\mathcal{B}}))$ is a Galois-Connection, where $\mathcal{P}(S_\sigma)$ is the powerset over the set of valid concrete program states S_σ and $\mathcal{P}(B_{\mathcal{B}})$ is the powerset over the set of abstract box states $B_{\mathcal{B}}$. Note, that we relate concrete program states with abstract box states, so we do not need to define the notion of a box state in the concrete world.

In the concrete program state, multiple instances of the same component may exist. These different instances are abstracted into a single abstract box.

4.3.1 Representation Function β

For each box class `B` we get different functions, therefore each function is implicitly marked with `B`.

We assume, that there exists a function cs, which takes a concrete object identifier and returns the creation site, i.e. the label of the `new` statement, where the corresponding object has been created.

The set $\mathcal{I}_\mathcal{H}$ contains all object identifiers, that describe objects that are part of some component with the box class `B`. It is defined as $\mathcal{I}_\mathcal{H} = \{o_{id} | \text{type}(\text{owner}(\mathcal{H}(o_{id}))) = \texttt{B}\}$.

Concrete values are abstracted with the function $abstract_v$. Object identifiers of a heap $\mathcal{H}$, that belong to the component are abstracted to their creation site, if not, they are considered to reference some external object.

$$abstract_v(\mathcal{B}, v) = \begin{cases} \texttt{null} & \text{if } v = \texttt{null} \\ \text{extRef} & \text{if } v \notin \mathcal{I}_\mathcal{H} \\ \text{cs}(v) & \text{otherwise} \end{cases}$$

Heap Abstraction To abstract the heap we remove all objects that do not belong to the component and merge the remaining objects that are created at the same creation site into a single abstract object.

We define a function $abstract_o$ that takes an object identifier and the concrete object for this identifier and returns an abstract object.

$$abstract_o(\mathcal{B}, o_{id}, (o_b, \texttt{C}, c, \bar{\text{f}} \mapsto \overline{v})) = (\texttt{C}, c, \text{isExposed}(o_{id}), \bar{\text{f}} \mapsto \overline{\{abstract_v(\mathcal{B}, v)\}})$$

We define the merge of two abstract objects, by merging the values of the fields and by setting the exposed status to true if one of the objects is exposed.

$$(\texttt{C}, c, e_1, \bar{\text{f}} \mapsto \overline{v_f^a}) \cup (\texttt{C}, c, e_2, \bar{\text{f}} \mapsto \overline{v_f^{a\prime}}) = (\texttt{C}, c, e_1 \vee e_2, \bar{\text{f}} \mapsto \overline{v_f^a \cup v_f^{a\prime}})$$

The set of all object identifiers that reference objects with the same creation site is called $\mathcal{L}_l$ and defined as $\mathcal{L}_l = \{o_{id} | cs(o_{id}) = l\}$

We can now define the abstraction for a heap as

$$abstract_H((\mathcal{H}, \mathcal{S})) = [l \mapsto \bigcup_{o'_{id} \in \mathcal{L}_l} abstract_o(o'_{id})] \text{ for all } l = \text{cs}(o_{id}), o_{id} \in \mathcal{I}_\mathcal{H}$$

Note, that as we only consider valid concrete program states, we know that objects created by the same line of code have the same type and the same confinement status. Both values cannot change during the execution of a program, thus we can conclude that $abstract_H$ is a well-defined function.

Stack Abstraction Abstracting the stack means to merge all stackframes corresponding to some execution inside the box into an abstract stack vector.

First we define for a concrete state $(\mathcal{H}, stack)$ the set $\mathcal{F}$ which contains all stackframes that correspond to some execution inside the box, i.e. frames which have a `this`-object which is part of the component.

$$\mathcal{F}_{\mathcal{H},\mathcal{S}} = \{f_i | f_i \in \mathcal{S}, f_i(\texttt{this}) \in \mathcal{I}_\mathcal{H}\}$$

The merge of two stackframes is defined as follows:

$$f_1 \cup f_2 = \overline{\mathtt{x}} \mapsto \overline{f_1(x) \cup f_2(x)} \text{ for all } \mathtt{x} \in f_1 \text{ or } f_2, \mathtt{x} \neq \mathtt{this}$$

The `this` variable is not merged because, we always want to have a single value stored in it, such that we exactly know on which object we are currently executing code.

The complete abstraction function for the stack looks as follows

$$abstract_S(\mathcal{H}, \mathcal{S}) = [\mathtt{this} \mapsto abstract_v(\mathcal{S}(\mathtt{this}))] \cdot \bigcup_{f \in \mathcal{F}_{\mathcal{H},\mathcal{S}}} f$$

Representation function β Using these previously defined functions we construct the representation function β, that maps concrete program states to abstract box states. Because $abstract_H$ and $abstract_S$ are well-defined functions, β is well-defined.

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

4.3.2 The Galois Connection $(\mathcal{P}(S_\sigma), \alpha, \gamma, \mathcal{P}(B_\mathcal{B}))$

For the Galois connection between valid concrete program states and abstract box states, we need some complete lattices. For the concrete domain we choose $(\mathcal{P}(S_\sigma), \subseteq)$, i.e. the powerset over the set of valid concrete program states together with the subset relation. See [6] A.1 for the proof of $(\mathcal{P}(M), \subseteq)$ being a complete lattice for any set M.

On the abstract domain, we define a partial order $\leq$ on abstract objects, $o_1 \leq o_2$ if o_2 contains less precise information about the values of fields and the exposed status.
$o_i = (\mathtt{C}, \text{confined}, \text{exposed}_i, \overline{\mathtt{f}} \mapsto \overline{\{v_i^a\}})$ for $i = 1, 2$

$$\begin{array}{lcl} o_1 \leq o_2 & \Leftrightarrow & \text{exposed}_1 \Rightarrow \text{exposed}_2, \\ & & o_1(\mathtt{f}) \subseteq o_2(\mathtt{f}) \text{ or extRef} \in o_2(\mathtt{f}) \end{array}$$

We use this order to define a partial order on box states.

$$\begin{array}{lcl} \mathcal{H}_a^1 \leq \mathcal{H}_a^2 & \Leftrightarrow & \mathcal{H}_a^1(l) \leq \mathcal{H}_a^2(l) \text{ for all defined } \mathcal{H}_a^1(l) \\ \mathcal{S}_a^1 \leq \mathcal{S}_a^2 & \Leftrightarrow & \mathcal{S}_a^1(\mathtt{x}) \subseteq \mathcal{S}_a^2(\mathtt{x}) \text{ for all variables } \mathtt{x}, \\ & & \mathcal{S}_a^1(\mathtt{this}) = \mathcal{S}_a^2(\mathtt{this}) \\ (\mathcal{H}_a, \mathcal{S}_a) \leq (\mathcal{H}'_a, \mathcal{S}'_a) & \Leftrightarrow & \mathcal{H}_a \leq \mathcal{H}'_a \wedge \mathcal{S}_a \leq \mathcal{S}'_a \\ (\emptyset, \emptyset) \leq \mathcal{B} \leq \top_{abort} & & \text{for all } \mathcal{B} \end{array}$$

With this order we can define a special subset relation, that takes the abstract objects into account. $\subseteq'$ is defined as

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

The complete lattice for the abstract site of the Galois connection is choosen as $(\mathcal{P}(B_\mathcal{B}), \subseteq')$. Proving $(\mathcal{P}(B_\mathcal{B}), \subseteq')$ being a complete lattice can be done by applying total functions and building cartesian product on the lattice $(\mathcal{P}(O), \subseteq)$ with O being the set of abstract objects. (See [6] for details).

We define the abstraction and concretization functions as usual out of the representation function.

$$\begin{array}{lcl} \alpha : \mathcal{P}(S_\sigma) \to \mathcal{P}(B_\mathcal{B}) & & \\ \gamma : \mathcal{P}(B_\mathcal{B}) \to \mathcal{P}(S_\sigma) & & \\ \alpha({S_\sigma}') & = & \{\beta(\sigma) \mid \sigma \in {S_\sigma}'\} \\ \gamma({B_\mathcal{B}}') & = & \{\sigma \mid \beta(\sigma) \in {B_\mathcal{B}}'\} \end{array}$$

for ${S_\sigma}' \subseteq S_\sigma$ and ${B_\mathcal{B}}' \subseteq' B_\mathcal{B}$.

β gives rise to the Galois connection

$$(\mathcal{P}(S_\sigma), \alpha, \gamma, \mathcal{P}(B_\mathcal{B}))$$

between $\mathcal{P}(S_\sigma)$ and $\mathcal{P}(B_\mathcal{B})$.

Proof. Due to the restriction of S_σ to valid concrete program states, it is clear, that β is well-defined. Therefore α and γ are well-defined too. We have,

$$\begin{array}{rcl} \alpha({S_\sigma}') \subseteq' {B_\mathcal{B}}' & \Leftrightarrow & \{\beta(\sigma) \mid \sigma \in {S_\sigma}'\} \subseteq' {B_\mathcal{B}}' \\ & \Leftrightarrow & \forall \sigma \in {S_\sigma}' : \{\beta(\sigma)\} \subseteq' {B_\mathcal{B}}' \\ & \Leftrightarrow & {S_\sigma}' \subseteq' \gamma({B_\mathcal{B}}') \end{array}$$

This means, that $(\mathcal{P}(S_\sigma), \alpha, \gamma, \mathcal{P}(B_\mathcal{B}))$ is an adjunction, which is equivalent to being a Galois connection (See [6] Proposition 4.20).

An example how the abstraction of a program state looks like is given in Figure 5. The picture shows the heap of a concrete state of a program, which has two list boxes, and one of them has exposed an iterator object. Instead of drawing the reference from each object to its owner object, all objects with the same owner are grouped in dashed boxes. In Figure 6 the abstraction of the heap is shown.

4.4 Abstract Transformers

In this section we give an abstract semantics similar to the concrete semantics in the last chapter. As we are developing a modular analysis, we have to take into account, that the component can be used in very different program contexts and in each context it has to fulfill the encapsulation property. The idea to handle this, is to execute the code of the component in an artificial program that can create all traces through the component. Instead of constructing such a program explicitly we simulate its behaviour in the abstract semantics. Whenever the control flow leaves the box, the semantics construct all possible method calls that the components context could initiate and executes them. We assume that the context itself conforms to the box model, i.e. it will not call local methods on the analyzed component.

4.4.1 Auxiliary Functions

Like in the concrete semantics, expressions are evaluated by the function $eval_a$, which returns the values of the variable stored in the stack or `null` for the `null` constant.

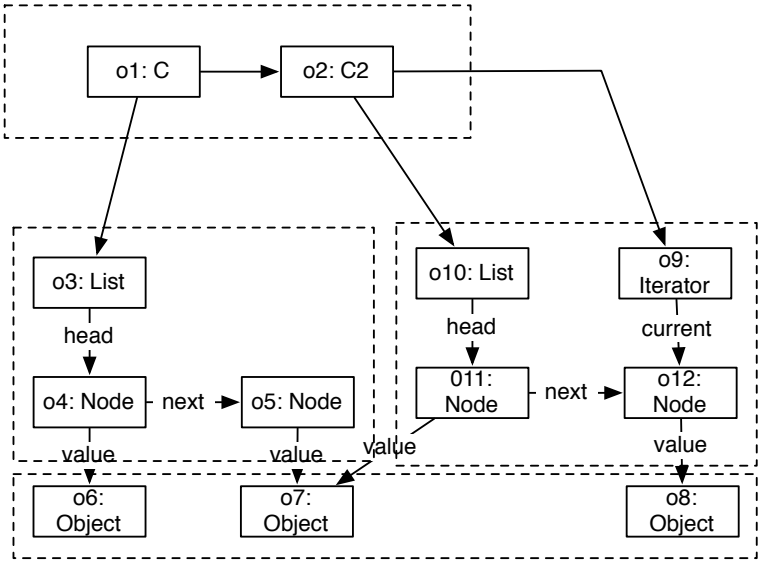

Figure 5: Concrete state of a program using the list box

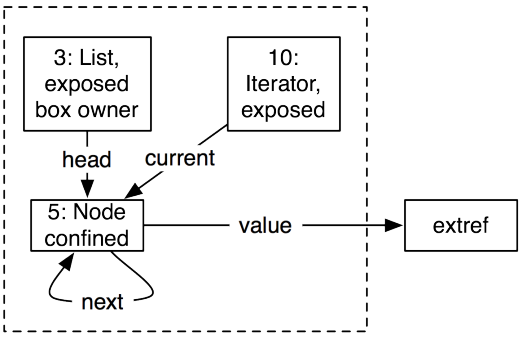

Figure 6: Abstraction of the state from figure 5

The auxiliary functions of the concrete semantics are analogously defined for the abstract semantics. Additionally, we define for an abstract object $(\mathtt{C}, \text{confined}, \text{exposed}, \overline{\mathtt{f} \mapsto \{v^a\}})$ the function isExposed, which returns the exposed flag and the function expose with sets the flag to true.

The function methods is used to determine for a given abstract box state which methods can be called on objects of the box by an object in another box. These are all public methods on all exposed objects. The function returns a set of tuples, each one consisting of the label of the exposed object on which a method can be called, the parameters and local variables of the method and the method body. L is the set of creation sites of the component's codebase.

$$\begin{aligned} \mathit{methods}(\mathcal{H}_a) = \quad & \{(l_i, \text{body}(\mathtt{m}_j), \text{params}(\mathtt{C}, \mathtt{m}_j), \text{variables}(\mathtt{C}, \mathtt{m}_j) \mid \\ & \mathtt{C} = \text{type}(\mathcal{H}_a(l_i)) \\ & l_i \in L, \text{exposed}(\mathcal{H}_a(l_i)), \\ & \mathtt{m}_j \in \text{publicMethods}(\mathtt{C})\} \end{aligned}$$

$$\frac{\ldots \texttt{ class C extends C'}\{\ldots \overline{M_i} \ldots\} \qquad M_i = \texttt{public } T\ \mathtt{m}_i\ (\overline{T}\ \overline{\mathrm{p}}) \ldots}{\text{publicMethods}(\mathtt{C}) = \{\mathtt{m}_i\} \cup \text{publicMethods}(\mathtt{C}')}$$

$$\frac{\ldots \texttt{ class C } \{\ldots \overline{M_i} \ldots\} \qquad M_i = \texttt{public } T\ \mathtt{m}_i\ (\overline{T}\ \overline{\mathrm{p}}) \ldots}{\text{publicMethods}(\mathtt{C}) = \{\mathtt{m}_i\}}$$

For a class declaration of the form $\ldots$ `class B` $\ldots\ T\ \mathtt{m}\ (\overline{T}\ \overline{\mathrm{p}})\{\overline{T}\ \overline{\mathrm{x}}; S; \texttt{return x}\} \ldots$, with `B` being the smallest supertype of `C` defining `m`, we define

$$\text{params}(\mathtt{C}, \mathtt{m}) = \overline{\mathrm{p}}$$

$$\text{variables}(\mathtt{C}, \mathtt{m}) = \overline{\mathrm{x}}$$

The function inMethod is used to find the name of the method in which the statement with some label is written.

$$\frac{\ldots \texttt{ class C } \ldots\ T\ \mathtt{m}\ (\overline{T}\ \overline{\mathrm{p}})\{\overline{T}\ \overline{\mathrm{x}}; \ldots [\mathtt{S}]^l \ldots\} \ldots}{\text{inMethod}(l) = \mathtt{m}}$$

A variable can only be used as actual parameter or return value, if it does not reference any confined object, i.e. if its value set does not contain extRef. This check is done by the function *noneConfined*.

$$\frac{\neg\text{confined}(\mathcal{H}_a(l)) \text{ for all } l \in \mathcal{S}_a(\mathtt{x})}{\mathit{noneConfined}((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}) = \text{true}} \qquad \frac{\neg\text{confined}(\mathcal{H}_a(l)) \text{ for some } l \in \mathcal{S}_a(\mathtt{x})}{\mathit{noneConfined}((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}) = \text{false}}$$

4.4.2 Abstract Transition Function

The transition function $\Rightarrow_a^*$ of the abstract semantics takes a set of pairs, consisting of an abstract box state and a statement and returns a set of other pairs and final box states. On each pair the function $\Rightarrow_a$ is executed. And the result is the union of the $\Rightarrow_a$'s results.

The function $\Rightarrow_a$ has the following forms:

- $(\mathcal{B}, S) \Rightarrow_a \{(\mathcal{B}', S')\}$: the execution of the statement S in the state $\mathcal{B}$ is not yet completed and $\{(\mathcal{B}', S')\}$ expresses the remaining executions.
- $(\mathcal{B}, S) \Rightarrow_a \{\mathcal{B}'\}$: the execution of statement S in $\mathcal{B}$ has terminated and $\{\mathcal{B}'\}$ are the final states.

The transition functions ignore flowsensitive information, such as the condition in `if` statements.

For readability we omit the labels of statements in the definition of the transition function except if they are really needed.

Sequences of statements are executed one after the other.

$$\text{A-SEQ-1}\quad \frac{(\mathcal{B}, S_1) \Rightarrow_a \{\mathcal{B}_i\}}{(\mathcal{B}, S_1; S_2) \Rightarrow_a \cup_i \{(\mathcal{B}_i, S_2)\}} \qquad \text{A-SEQ-2}\quad \frac{(\mathcal{B}, S_1) \Rightarrow_a \{(\mathcal{B}_i, S_i)\}}{(\mathcal{B}, S_1; S_2) \Rightarrow_a \cup_i \{(\mathcal{B}_i, S_i; S_2)\}}$$

Assigning a value to a variable just adds the new value to the already existing ones. Thus, the variables on the stack never forget any value. This is needed because one abstract variable may be related to several concrete variables and in the abstraction we do not know, which instance we manipulate.

$$\text{A-ASSIGN}\quad \frac{\mathcal{S}'_a = [\mathtt{x} \mapsto \{\text{eval}^a(e)\} \cup \mathcal{S}_a(\mathtt{x})]\mathcal{S}_a}{((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x = } e) \Rightarrow_a (\mathcal{H}_a, \mathcal{S}'_a)}$$

When executing a new statement for a non-box object two scenarios are possible. Either the statement has been executed before, then an abstract object with the object identifier already exists or it is the first time then the object has to be created. If a new object is created, it has not been exposed and its fields are `null`.

$$\text{A-NEW-OBJ-1}\quad \frac{\neg\text{boxclass}(\mathtt{C}) \qquad \mathcal{H}_a(l) \text{ is defined} \qquad \mathcal{S}'_a = [\mathtt{x} \mapsto \mathcal{S}_a(\mathtt{x}) \cup \{l\}]\mathcal{S}_a}{((\mathcal{H}_a, \mathcal{S}_a), [\mathtt{x = new\ boundary\ C()}]^l) \Rightarrow_a \{(\mathcal{H}_a, \mathcal{S}'_a)\}}$$

$$\text{A-NEW-OBJ-2}\quad \frac{\neg\text{boxclass}(\mathtt{C}) \qquad \mathcal{H}_a(l) \text{ is not defined} \qquad \mathcal{S}'_a = [\mathtt{x} \mapsto \mathcal{S}_a(\mathtt{x}) \cup \{l\}]\mathcal{S}_a \qquad o_{new} = (\mathtt{C}, \text{false}, \text{false}, \overline{\mathtt{f}} \mapsto \overline{\mathtt{null}}) \qquad \mathcal{H}'_a = [l \mapsto o_{new}]\mathcal{H}_a}{((\mathcal{H}_a, \mathcal{S}_a), [\mathtt{x = new\ boundary\ C()}]^l) \Rightarrow_a \{(\mathcal{H}'_a, \mathcal{S}'_a)\}}$$

For the creation of confined objects we have the same rules as for the non-confined object, except that the confined field of the new object is set to true.

$$\text{A-NEW-OBJ-1-CONF}\quad \frac{\neg\text{boxclass}(\texttt{C}) \qquad \mathcal{H}_a(l) \text{ is defined} \qquad \mathcal{S}'_a = [\texttt{x} \mapsto \mathcal{S}_a(\texttt{x}) \cup \{l\}]\mathcal{S}_a}{((\mathcal{H}_a, \mathcal{S}_a), [\texttt{x = new confined C()}]^l) \Rightarrow_a \{(\mathcal{H}_a, \mathcal{S}'_a)\}}$$

$$\text{A-NEW-OBJ-2-CONF}\quad \frac{\neg\text{boxclass}(\texttt{C}) \qquad \mathcal{H}_a(l) \text{ is not defined} \qquad \mathcal{S}'_a = [\texttt{x} \mapsto \mathcal{S}_a(\texttt{x}) \cup \{l\}]\mathcal{S}_a \qquad o_{new} = (\texttt{C}, \text{true}, \text{false}, \overline{\texttt{f}} \mapsto \overline{\texttt{null}}) \qquad \mathcal{H}'_a = [l \mapsto o_{new}]\mathcal{H}_a}{((\mathcal{H}_a, \mathcal{S}_a), [\texttt{x = new confined C()}]^l) \Rightarrow_a \{(\mathcal{H}'_a, \mathcal{S}'_a)\}}$$

The instantiation of a box class always leads to the creation of a new box. Therefore the new object is not part of the current box and so we only have to assign the variable x the value extRef for a reference outside of the box.

$$\text{A-NEW-BOX}\quad \frac{\text{boxclass}(\texttt{C}) \qquad \mathcal{S}'_a = [\texttt{x} \mapsto \mathcal{S}_a(\texttt{x}) \cup \{\text{extRef}\}]\mathcal{S}_a}{((\mathcal{H}_a, \mathcal{S}_a), \texttt{x = new C()}) \Rightarrow_a \{(\mathcal{H}_a, \mathcal{S}'_a)\}}$$

Reading and updating a field merges the right hand side values with the already stored ones.

$$\text{A-FIELD-READ}\quad \frac{\mathcal{S}'_a = [\texttt{x} \mapsto \mathcal{H}(\mathcal{S}(\texttt{this}))(\texttt{f}) \cup \mathcal{S}(\texttt{x})]\mathcal{S}_a}{((\mathcal{H}_a, \mathcal{S}_a), \texttt{x = this.f}) \Rightarrow_a \{(\mathcal{H}_a, \mathcal{S}'_a)\}}$$

$$\text{A-FIELD-UP}\quad \frac{o_i = \mathcal{H}(\mathcal{S}(\texttt{this})) \qquad o_i^{up} = [\texttt{f} \mapsto o_i(\texttt{f}) \cup \text{eval}^a(e)]o_i \qquad \mathcal{H}'_a = [l_i \mapsto o_i^{up}]\mathcal{H}_a}{((\mathcal{H}_a, \mathcal{S}_a), \texttt{this.f = } e) \Rightarrow_a \{(\mathcal{H}'_a, \mathcal{S}_a)\}}$$

Due to flow-insensitivity the execution of a conditional expression leads to two different states.

$$\text{A-IF}\quad \frac{}{(\mathcal{B}, \texttt{if (x) } S_1 \texttt{ else } S_2) \Rightarrow_a \{(\mathcal{B}, S_1), (\mathcal{B}, S_2)\}}$$

For calls we have to distinguish between inner calls and calls to the outside. To handle calls we start with resolving the receiver of the call. We generate a state-statement pair for all possible receivers, such that the next rules only have to handle exactly one receiver, which can be some local object or extRef. The parameters are pushed onto the stack and the variable `this` is set to the receiver.

A-CALL

$$\frac{\bar{l} = \mathcal{S}_a(\mathtt{y}) \setminus \{\mathtt{null}\}}{((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x = y.m(x}_j\mathtt{)}) \Rightarrow_a \cup_i \{(\mathcal{H}_a, \mathcal{S}_a), \mathtt{x = } \, l_i\mathtt{.m(x}_j\mathtt{)}\}}$$

For internal calls we put the actual parameters on the stack, add the default value to the values for the local variables and set the `this`-object to the callee and continue with the body of the called method. The assignment of the return value is added to the sequence of statements to be executed, like in the concrete semantics.

A-CALL-INTERNAL

$$\frac{l \neq \text{extRef} \qquad \mathtt{C} = \text{type}(\mathcal{H}(l)) \qquad \mathcal{S}'_a = [\mathtt{this} \mapsto \{l\}, \mathtt{y}_k \mapsto \{\mathtt{null}\} \cup \mathcal{S}_a(\mathtt{y})_k, \mathtt{p}_j \mapsto \mathcal{S}_a(\mathtt{p}_j) \cup \mathcal{S}_a(\mathtt{x}_j)]\mathcal{S}_a}{((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x = } \, l\mathtt{.m(x}_j\mathtt{)}) \Rightarrow_a \{((\mathcal{H}_a, \mathcal{S}'_a), \text{body}(\mathtt{C}, \mathtt{m}); \ \mathtt{x = retVal})\}}$$

If the receiver of a call is extRef then the call is directed outside the box. In that case we have to ensure, a) the called method is public and b) no actual parameter is confined. Furthermore, we have to expose the actual parameters. We then go into the state $\mathcal{B}^{ex}$, which signals, that the component state is $\mathcal{B}$, but the control flow left the component. As we do not know anything about the return value, and it is therefore possible that the return value is a reference to an object in a different box, we set `x` to extRef in the stack. We do not have to deal with all possible return values, as our lattice is defined, such that a set containing extRef is always greater that one without.

A-CALL-EXTERNAL

$$\frac{\neg\text{local}(\mathtt{m}) \qquad \begin{array}{c} noneConfined((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}_j) \\ \mathcal{H}'_a = [l \mapsto \text{expose}(\mathcal{H}_a(l))]\mathcal{H} \text{ for all } l \in \mathcal{S}_a(\mathtt{x}_j) \end{array}}{((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x = extRef.m(x}_j\mathtt{)}) \Rightarrow_a \{(\mathcal{H}'_a, \mathcal{S}_a)^{ex}, \mathtt{x = extRef}\}}$$

A-CALL-EXTERNAL-ABORT

$$\frac{\text{local}(\mathtt{m}) \vee \neg noneConfined((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}_j)}{((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x = extRef.m(x}_j\mathtt{)}) \Rightarrow_a \top_{abort}}$$

Whenever the control flow is outside the box, the component's context can initiate a callback to any of the public methods of the exposed objects or the context can execute a `return` statement, that returns the control flow back into the component. We use extRef as value for the parameters of callbacks.

External references are the cause for violations of the encapsulation properties, so passing extRef as return value or as a parameter to a callback imposes the most restrictions on the remaining execution. Thus, extRef is the worst scenario that may happen to the executing component. Because our analysis has to guarantee, the absence of encapsulation errors, we can limit ourselves to the worst case. This reduces the state space, the analysis has to handle.

A-CALLBACKS

$$\frac{\{(l_i, b_i, \overline{\mathtt{p}_i}, \overline{\mathtt{y}}_i)\} = \text{methods}(\mathcal{H}_a) \quad \mathcal{S}_a^i = [\mathtt{this} \mapsto \{l_i\}, \overline{\mathtt{y}}_i \mapsto \{\mathtt{null}\} \cup \mathcal{S}_a(\overline{\mathtt{y}})_i, \overline{\mathtt{p}}_i \mapsto \text{extRef} \cup \mathcal{S}_a(\overline{\mathtt{p}}_i)]\mathcal{S}_a \text{ for all } i}{((\mathcal{H}_a, \mathcal{S}_a)^{ex}, \mathtt{x = extRef}) \Rightarrow_a \cup_i\{((\mathcal{H}_a, \mathcal{S}_a^i), b_i\mathtt{;\ x = extRef})\} \cup \{((\mathcal{H}_a, \mathcal{S}_a^i), \mathtt{return\ extRef;\ x = extRef})\}}$$

When executing a return statement, we have to distinguish between returning from an internal or from an external call. We can do this, based on the statement following the return. When returning from an internal call, the return statement is always followed by an assignment of the return value to a variable. For a return from an external call the return statement is followed by an assignment of extRef to a variable.

The return of an internal call puts the return value into the stack vector and the execution continues in all objects, that could have originated the call, i.e. that 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, where the analysis can be improved and tailored to specific components.

A-RETURN-INTERNAL

$$\frac{\mathcal{S}_a^i = [\mathtt{this} \mapsto \{\mathtt{this}_i\}, \mathtt{retVal} \mapsto \mathcal{S}_a(\mathtt{x}_1)]\mathcal{S}_a \quad \mathtt{m} = \text{inMethod}(j) \quad \mathtt{this}_i = \{l_i \mid l_i \in L, \mathtt{m} \in \text{declared methods of type}(\mathcal{H}_a(l_i))\}}{((\mathcal{H}_a, \mathcal{S}_a), [\mathtt{return\ x}_1]^j\mathtt{;\ x}_2\mathtt{\ = retVal}) \Rightarrow_a \cup_i\{((\mathcal{H}_a, \mathcal{S}_a^i), \mathtt{x}_2\mathtt{\ = retVal})\}}$$

In case of a return from an external call, i.e. the controlflow will leave 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. We enter the external state then. If the return value is confined, returning it would break the encapsulation, therefore we enter the abortion state.

A-RETURN-EXTERNAL

$$\frac{noneConfined((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}_1) \quad \mathcal{H}'_a = [l \mapsto \text{expose}(\mathcal{H}_a(l))]\mathcal{H}_a \text{ for all } l \in \mathcal{S}_a(\mathtt{x}_1)}{((\mathcal{H}_a, \mathcal{S}_a), \mathtt{return\ x}_1\mathtt{;\ x}_2\mathtt{\ = extRef}) \Rightarrow_a \{((\mathcal{H}'_a, \mathcal{S}_a)^{ex},\ \mathtt{x}_2\mathtt{\ = extRef})\}}$$

A-RETURN-EXTERNAL-ABORT

$$\frac{\neg noneConfined((\mathcal{H}_a, \mathcal{S}_a), \mathtt{x}_1)}{((\mathcal{H}_a, \mathcal{S}_a), \mathtt{return\ x}_1\mathtt{;\ x}_2\mathtt{\ = extRef}) \Rightarrow_a \top_{abort}}$$

We need two rules for the initial state, which is $(\emptyset, \emptyset)$. The execution of the box `B` starts with its creation and the possible calls to the box owner object. The rule is a combination of the rules for creating new objects and doing call on it, like in the call rules. Note, that the assignment of the variable `x` is executed outside the box, therefore it is not stored on the stack. The label of the new box owner object is always considered

to be l_0. The second rule catches executions of all other statements on the initial state. As we ignore everything before the creation of the box, we do not need the source for this part the program, that is, for the analysis it is enough to consider the codebase of a box.

$$\text{A-INIT-BOX}\quad \frac{\texttt{B} \text{ is the box class under analysis} \qquad \mathcal{H}'_a = [l_0 \mapsto (\texttt{C}, \text{false}, \text{true}, \bar{\text{f}} \mapsto \overline{\texttt{null}})]}{((\emptyset,\emptyset)^{ex}, \texttt{x = new B()}) \Rightarrow_a \{((\mathcal{H}'_a, \mathcal{S}_a)^{ex}, \texttt{x = extRef})\}}$$

$$\text{A-INIT-IGNORE}\quad \frac{S \neq \texttt{x = new B()}}{((\emptyset,\emptyset)^{ex}, S) \Rightarrow_a \{(\emptyset,\emptyset)^{ex}\}}$$

4.4.3 Abstract Encapsulation Properties

Like for the concrete semantics, we define an abstract accessibility invariant.

Definition 2 (Abstract Accessibility Invariant) *1. Object Access. References to confined abstract objects are never passed to the receiver extRef, thus objects not part of the box never get access to a confined object.*

2. Method Calls. Abstract objects never call local methods on extRef references.

This leads to the abstract encapsulation property.

Corollary 2 (Abstract Encapsulation Property) *If the abstract state $\top_{abort}$ is not reachable from the initial state $(\emptyset,\emptyset)$, then the box* B *is boxed, i.e. it fulfills the abstract accessibility invariant.*

The proof is clear by the definition of the transition function.

We now have to relate the abstract and concrete encapsulation properties. We will show, that the abstract encapsulation property implies the concrete one. Thus, we can use the set of reachable abstract states to determine encapsulation. Note, that the concrete invariant does not imply the abstract one. This is due to the loss of object identities and because we ignore parts of the controlflow.

We constructed the abstract transition function, such that we can use it to compute an over approximation of the reachable concrete states. In Figure 7 the relations between the sets of concrete and abstracts states are depicted, but we still have to show, that we really have this relation between $\Rightarrow$ and $\Rightarrow_a^*$, i.e. we still have to show that the abstraction of the semantics is safe.

Theorem 1 (Safe Abstraction) *$(\mathcal{P}(S_\sigma), \alpha, \gamma, \mathcal{P}(B_\mathcal{B}))$ is the Galois connection of section 4.3.2, ${S_\sigma}' \in \mathcal{P}(S_\sigma)$, S is a statement. For the transition function $\Rightarrow_a^*$ holds the following*

$$\gamma(\{(\mathcal{B}'_j, S_j)\} \cup \{\mathcal{B}'_{j'}\}) \supseteq' \{\sigma'_i \mid \sigma_i \in {S_\sigma}', \sigma_i, S \Rightarrow \sigma'_i, S' \text{ or } \sigma_i, S \Rightarrow \sigma'_i\}$$

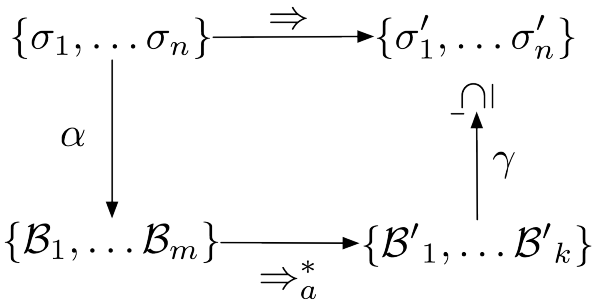

Figure 7: The relations between concrete and abstract states

with

$$\{(\beta(\sigma_i), S) \mid \sigma_i \in S_\sigma{}'\} \Rightarrow^*_a \{(\mathcal{B}'_j, S_j)\} \cup \{\mathcal{B}'_{j'}\}$$

We call the transition function $\Rightarrow^*_a$ *a* safe abstraction *of the concrete transition function* $\Rightarrow$*.*

Proof. Proof by induction. Let `B` be the boxclass used for the Galois connection.

Base Case The initial concrete state is $\sigma_{init} = (\emptyset, \emptyset)$. The main statement executed by a concrete program is S: `x = new C(); x.m(`$\overline{c}$`)`, $\alpha(\{\sigma_{init}\}) = \{(\emptyset, \emptyset)\}$ by the definition of α.

If `C` $\neq$ `B`: $\sigma_{init}, S \Rightarrow \sigma, S'$. $\alpha(\{\sigma\}) = \{(\emptyset, \emptyset)\}$, because there is no instance of box `B`. With the rule A-Init-Ignore we get $\{((\emptyset, \emptyset), S)\} \Rightarrow^*_a \{((\emptyset, \emptyset), S')\}$. The abstract transition function is safe.

If $C = B$: $\sigma_{init}, S \Rightarrow (\mathcal{H}, \mathcal{S}), S'$ with $\mathcal{H} = [o_{id} \mapsto= (b_{id}, \mathtt{B}, \text{false}, \bar{\mathtt{f}} \mapsto \overline{v_{default}})]$, $\mathcal{S} = [\mathtt{x} \mapsto o_{id}]$. Applying the abstraction leads to the abstract state $\mathcal{B} = ([\text{cs}(o_{id}) \mapsto (\mathtt{B}, \text{false}, \text{true}, \bar{\mathtt{f}} \mapsto \overline{v^a_{default}})], \emptyset)$. With the rule A-Init-Box we get $\{((\emptyset, \emptyset), \mathtt{S})\} \Rightarrow^*_a \{((\mathcal{H}_a, \emptyset), \mathtt{S'}_i)\}$ with $\mathcal{H}_a = [l \mapsto (\mathtt{B}, \text{false}, \text{true}, \bar{\mathtt{f}} \mapsto \overline{v^a_{default}})]$. By the definition of cs and fresh we know that $\text{cs}(o_{id}) = l$. The abstract transition function is safe.

Inductive Step For simplicity we write S_σ, $S \Rightarrow S_\sigma$', and omit the potential continuations. Be $\sigma = (\mathcal{H}, \mathcal{S})$, $S_\sigma = \{\sigma_1, \ldots, \sigma_n\}$.

- $S_1; S_2$: Follows directly from the induction hypotheses.
- `x = e`: $\beta(\sigma'_i) = \beta((\mathcal{H}, [\mathtt{x} \mapsto v] \cdot \mathcal{S}) = (\mathcal{H}_a, [\mathtt{x} \mapsto \{v^a\} \cup \mathcal{S}_a(\mathtt{x})]\mathcal{S}_a)$. Abstracting S_σ and executing one step leads to the following abstract state set $\{(\mathcal{H}_a, [\mathtt{x} \mapsto \{v^a_i\} \cup \mathcal{H}_a(\mathtt{x})]\mathcal{S}_a)_j\}$ where $j = 1 \ldots n$. With the induction hypotheses we get $\alpha(S_\sigma) = \{(\mathcal{H}_a, [\mathtt{x} \mapsto \{v^a_i\} \cup \mathcal{S}_a(x)]\mathcal{S}_a)\} \subseteq' \{(\mathcal{H}_a, [\mathtt{x} \mapsto \{v^a_i\} \cup \mathcal{H}_a(\mathtt{x})]\mathcal{S}_a)_j\}$. So $\Rightarrow^*_a$ is safe.
- `x = this.f` $\sigma_i, S \Rightarrow (\mathcal{H}, [\mathtt{x} \mapsto \mathcal{H}(\mathcal{S}(\mathtt{this}))(\mathtt{f})] \cdot \mathcal{S}])$. Thus, $\beta(\sigma_i) = (\mathcal{H}_a, [\mathtt{x} \mapsto \{v^a\} \cup \mathcal{H}_a(\mathcal{S}_a(\mathtt{this}))(\mathtt{f})]\mathcal{S}_a)$ with $\{v^a\} = \mathcal{H}_a(\mathcal{S}_a(\mathtt{this}))(\mathtt{f})$ Abstracting S_σ and executing one step leads to the following abstract state set $\{(\mathcal{H}_a, [\mathtt{x} \mapsto \mathcal{H}_a(\mathcal{S}_a(\mathtt{this}))(\mathtt{f})]\mathcal{S}_a\}$. Obviously, $\{v^a\} \cup \mathcal{H}_a(\mathcal{S}_a(\mathtt{this}))(\mathtt{f}) = \mathcal{H}_a(\mathcal{S}_a(\mathtt{this}))(\mathtt{f})$. By the induction hypothesis we get, $\Rightarrow^*_a$ is safe.

- `this.f =` e Similar.
- `x = new boundary C()`: $\sigma_i, S \Rightarrow ([o_{id} \mapsto (b_{id}, \mathtt{C}, \text{false}, \bar{\mathtt{f}} \mapsto \overline{\mathtt{null}})\mathcal{H}, [\mathtt{x} \mapsto o_{id}]\mathcal{S})$. With the definition of β we get $\beta(\sigma_i) = ([\text{cs}(o_{id}) \mapsto (\mathtt{C}, \text{false}, \text{false}, \bar{\mathtt{f}} \mapsto \overline{v^a_{default}})\mathcal{H}_a, [\mathtt{x} \mapsto \text{cs}(o_{id})]\mathcal{S}_a)$. Executing the statement on $\alpha(S_\sigma)$ leads to $\{(\mathcal{H}'_a, [\mathtt{x} \mapsto \mathcal{S}_a(\mathtt{x}) \cup l)\ \mathcal{S}_a\}$. If the object has not been in the heap before, $\mathcal{H}'_a = [l \mapsto (\mathtt{C}, \text{false}, \text{false}, \bar{\mathtt{f}} \mapsto \overline{v^a_{default}})]\mathcal{H}_a$. In this case it is clear that $\Rightarrow^*_a$ is safe. If the object has already been in the heap, $\mathcal{H}'_a(l) = (\mathtt{C}, \text{false}, \text{exposed}, \bar{\mathtt{f}} \mapsto \overline{v^a})$. Note, that the default value of an field is always part of the value set, because the default value is written into the field upon creation. With definition of $\subseteq'$ and the induction hypotheses we know that the abstract transition function is safe.
- `x = new confined C()` Similar.
- `x = new BC()` A: If $\mathtt{BC} \neq \mathtt{B}$: the newly created object will be removed by the abstraction, such that $\beta(\sigma'_i) = (\mathcal{H}'_a, [\mathtt{x} \mapsto \text{extRef}]\mathcal{S}_a)$. The execution on the abstract state $\alpha(S_\sigma)$ leads to $\{(\mathcal{H}'_a, [\mathtt{x} \mapsto \text{extRef}]\mathcal{S}_a)\}$. Thus $\Rightarrow^*_a$ is safe. B: If $\mathtt{BC} = \mathtt{B}$, we get $\sigma_i = ([o_{id} \mapsto (b_{id}, \mathtt{C}, \text{false}, \bar{\mathtt{f}} \mapsto \overline{\mathtt{null}})\mathcal{H}, [\mathtt{x} \mapsto o_{id}]\mathcal{S})$. Because the object is of the box type `B` it will get merged during the abstraction with the already existing ones. As the existing box object also contains $v^a_{default}$ in all fields the result of the merge is equal to the existing object. Thus, $\alpha(S_\sigma{}') = \mathcal{H}_a, [\mathtt{x} \mapsto \text{extRef} \cup \mathcal{S}_a(\mathtt{x})]\mathcal{S}_a$, which is the same as when executing the statement on $\alpha(S_\sigma)$. Therefore, we know that the abstract transition function is safe.
- $\mathtt{x}_1$ `= x2.m(`$\overline{\mathtt{p}}$`)` $\sigma_i, S \Rightarrow (\mathcal{H}, \mathcal{S}')_i$ with $\mathcal{S}' = [\overline{\mathtt{p}}_i \mapsto \overline{\mathcal{S}(\mathtt{x}_i)}, \overline{\mathtt{y}_k} \mapsto \overline{\mathtt{null}}, \mathtt{this} \mapsto \mathcal{S}(\mathtt{x}_2)] \cdot \mathcal{S}$. $\beta((\mathcal{H}, \mathcal{S}')_i) = (\mathcal{H}'_a, [\mathtt{p}_i \mapsto \mathcal{S}_a(\mathtt{x}_i) \cup \mathcal{S}_a(\mathtt{p}_i), \mathtt{y}_k \mapsto \{v^a_{default}\} \cup \mathcal{S}_a(\mathtt{y}_k), \mathtt{this} \mapsto \text{cs}(\mathcal{S}(\mathtt{x}_2))] \cdot \mathcal{S}_a)$ and $\mathcal{H}'_a = [\overline{\mathcal{S}_a(\mathtt{x}_1)} \mapsto \text{expose}(\mathcal{H}_a(\overline{\mathcal{S}_a(\mathtt{x}_1)}))]\mathcal{H}_a$. When executing the statement on $\alpha(S_\sigma)$, we execute the statement on several objects. We have $\text{cs}(\mathcal{S}(\mathtt{x}_2)) = l \in \mathcal{S}_a(\mathtt{x}_2)$, therefore we know, the call is executed on the abstract object corresponding to the called concrete object and on some more objects, that we can ignore because the $\subseteq'$ relation can already been shown without them. We also know, that if the concrete call is internal (external), the call on the corresponding abstract object is internal (external) as well. We distinguish between internal and external calls.

  Internal call: $\alpha(S_\sigma) \Rightarrow^*_a \{(\mathcal{H}_a, \mathcal{S}'_a)\}$ with $\mathcal{S}'_a = [\mathtt{this} \mapsto \{l\}, \mathtt{p}_i \mapsto \mathcal{S}_a(\mathtt{p}_i) \cup \mathcal{S}_a(\mathtt{x}_i)]\mathcal{S}_a$. It is easy to see that $\cup_i \beta((\mathcal{H}, \mathcal{S}')_i) \subseteq' \{(\mathcal{H}_a, \mathcal{S}'_a)\}$, therefore the transition function is safe.

  External call: If the method `m` is declared local or one of the parameters is confined, the execution leads to the abortion state. As the abstraction of $\top_{abort}$ leads to $\top_{abort}$, this is safe. If no parameter is confined, we have $\alpha(S_\sigma), S \Rightarrow^*_a \cup_i \{((\mathcal{H}'_a, \mathcal{S}^i_a)\}$ with $\mathcal{H}'_a = [l \mapsto \text{expose}(\mathcal{H}_a(l))]\mathcal{H}_a$ for all $l \in \mathcal{S}_a(\mathtt{x}_1)$ and $\mathcal{S}^i_a = [\mathtt{this} \mapsto \{l_i\}, \overline{\mathtt{p}_i} \mapsto \text{extRef} \cup \mathcal{S}_a(\overline{\mathtt{p}_i})]\mathcal{S}_a$ for all i. It now easy to see, that the transition function is safe.

- `return x` Similar to calls, but arguing about the return value instead of the parameters.
- `if (x) then` S_1 `else` S_2 Clear.

□

Knowing that $\Rightarrow_a^*$ is safe, we can directly derive the encapsulation criterion, which is the basis of the analysis.

Corollary 3 (Encapsulation Criterion) *If all components in a program are boxed, the encapsulation invariant holds for the whole program.*

4.5 Defining the equation system

To be able to use the abstract interpretation to define an equation system, which allows us to calculate the set of possible states at each program point, we need a relation flow. flow defines the controlflow graph of a component implementation. It relates the labels of statements to each other. If a pair (l_1, l_2) is part of flow, this means, that after the execution of the statement at l_1 the execution can continue with the statement at l_2. To define flow we first define the functions init and final. For a statement or a method body, init gives us the entry label, i.e. the label of the first statement and final the set of exit labels, i.e. the labels of the last executed statements.

Statement	init	final	flow
$\{\overline{T}\,\overline{\mathtt{x}}; S; [\mathtt{return\ x}]^l\}$	$\text{init}(S)$	$\{l\}$	$\{(l', l) \mid l' \in \text{final}(S)\}$
$S_1; S_2$	$\text{init}(S_1)$	$\text{final}(S_2)$	$\text{flow}(\mathtt{S_2})$
$[\mathtt{x = }\ e]^l$	l	$\{l\}$	$\{\}$
$[\mathtt{x = new}\ nm\ \mathtt{C()}]^l$	l	$\{l\}$	$\{\}$
$[\mathtt{x = x.m(}\overline{\mathtt{x}}\mathtt{)}]^{l1,l2}$	l_1	$\{l_2\}$	$\{(l_1, \text{init}(\text{body}(T, m))),$ $(\text{final}(\text{body}(T, m)), l_2)\}$ T: type of the receiver
$[\mathtt{x = this.f}]^l$	l	$\{l\}$	$\{\}$
$[\mathtt{this.f = }\ e]^l$	l	$\{l\}$	$\{\}$
$[\mathtt{if\ (x)\ \{}S_1\mathtt{\}\ else\ \{}S_2\mathtt{\}}]^l$	l	$\text{final}(S_1) \cup \text{final}(S_2)$	$\text{flow}(S_1) \cup \text{flow}(S_2) \cup$ $\{(l, \text{init}(S_{1)}), (l, \text{init}(S_2))\}$

We now define an equation system, which relates the reachable box states before executing a statement at label l with the box states after its execution and the box states of the previous and the next statements. So we handle two sets of states for each label, one at the entry and on at the exit. We consider 0 as the label for the whole box implementation.

The equation system to solve now looks like

$$
\begin{array}{lcl}
entry(0) & = & \{(\mathcal{B}_{init}, S_i)\} \text{ where } \mathrm{S}_i \text{ are the bodies of the methods of the box object} \\
entry(l) & = & \bigcup\{exit(l') | (l', l) \in \text{flow}\} \\
exit(l) & = & \Rightarrow_a^* (entry(l))
\end{array}
$$

As we are working on a finite lattice, we are able to calculate the fixpoints for these sets. If none of these sets contains the abortion state, $\top_{abort}$ is not reachable and by corollary 2 we know, that the component is boxed, i.e. it will never violate the encapsulation properties. So when several components with positive analysis results are put together into a whole program, the program fulfills the properties of the box model.

In Figure 8 the codebase of the list component from Figure 2 is given. For this example the calculation of flow returns

$$(1,2),(2,3),(3,9),(9,4),(4,5),(6,7),(7,12),(13,18),$$
$$(18,8),(9,10),(10,11),(12,13),(14,15),(15,16),(16,17)$$

When resolving the resulting equation system, we get two sets of possible boxstates for each label and none of them contains the abort state, so our list box implementation follows the box model, so it never exposes confined objects or call local methods on external objects.

5 Related Work

The box model of this work is a simpler version of the ones presented in [9], [7] and [8]. These papers describe box models that support the hierarchy of boxes. [9] defines a type system, that guarantees encapsulation of boxes. The type system needs a lot of annotations in the code, which can partly be infered. Whereas our approach only needs annotations for the `new` statement and the class declarations.

Close to our work are the ideas in [5]. This work defines a compositional separate analysis to show encapsulation for single objects. The idea there is to split the semantics of an object into the context part and the part inside the object. This is similar to our work. [5] uses regular expressions to approximate the interactions between the object and its context, whereas we use the worst case client.

The accessibility invariant allows to control the escaping of references to confined object. This is similar to the escape analysis, which try detect objects that do not leave a certain context. In [3], [4] such an analysis based on an abstract interpretation is given. The authors show that an object does not escape a certain context like a method or a thread. That is different to our work, where the encapsulation border is given in the heap, i.e. we do not handle escaping from a static scope. Like in our work they use an object allocation abstraction for the heap. Similar works have been presented in [1].

[2] gives an overview of different kinds of modular static analysis using abstract interpretations. Our analysis resembles the symbolic relational separate analysis but also uses parts of the worst case analysis.

```
box class List {
    Node head;
    Object add(Object o) {
        [Node t = head]_1;
        [head = new confined Node()]_2;
        [head.init(t,o)]_{3,4};
        [return null]_5;
    }
    Iterator iter() {
        [Iterator i = new Iterator()]_6;
        [i.init(head)]_{7,18};
        [return i]_8;
    }
}

class Node {
    Node next;
    Object value;
    Object init (Node n, Object o) {
        [next = n]_9;
        [value = o]_10;
        [return null]_11;
    }
}

class Iterator {
    Node current;
    Object init (Node n) {
        [current = n]_12;
        [return null]_13;
    }
    Object next() {
        [Node t = current]_14;
        [current = current.next]_15;
        [Object v = t.value]_16;
        [return v]_17;
    }
}
```

Figure 8: Codebase of a list component

6 Conclusion

We presented a modular, static analysis for checking the encapsulation property of the box model. Our analysis abstracts the infinite concrete state space to a finite one by projecting on the relevant parts of the heap, merging similar objects together and flatten the stack. Using this abstraction, which corresponds to an upper approximation of the call graph, we get an equation system, describing the reachable abstract states for each program point. Even if the abstraction throws away a lot of information about the objects and the call graph contains a huge number of edges, which represent an impossible flow of control in the concrete semantics, we think that it is powerful enough for a lot of components. To be able to examine bigger examples, we will implement the analysis and see where we have to improve it to be able to check as many components as possible. There are several points, where the analysis can be adapted to the component to analyze. First of all, the abstraction function can be refined, such that it distinguishes more different object or parts of the stack. The second big improvement could be done on the handling of returns. It is possible to restrict the set of objects, in which the execution can continue after executing a return instead of returning to all of them. These improvements will be part of further work.

References

[1] Roberto Barbuti and Stefano Cataudella. Abstract interpretation of an object calculus for synchronization optimizations. *Fundam. Inf.*, 67(1-3):1–12, 2005. ISSN 0169-2968.

[2] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *LNCS*, pages 159 – 178, London, UK, 2002. Springer-Verlag.

[3] Patricia M. Hill and Fausto Spoto. A foundation of escape analysis*. *Algebraic Methodology and Software Technology*, pages 5–11, 2002.

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

[5] Francesco Logozzo. Separate compositional analysis of class-based object-oriented languages. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 334–348. Springer, 2004. ISBN 3-540-22381-9.

[6] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer-Verlag New York Inc, 1999.

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

[8] Arnd Poetzsch-Heffter and Jan Schäfer. A representation-independent behavioral semantics for object-oriented components. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4468 of *Lecture Notes in Computer Science*, pages 157 – 173. Springer, 2007. ISBN 978-3-540-72919-8.

[9] Arnd Poetzsch-Heffter, Kathrin Geilmann, and Jan 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.