Modular Shape Analysis for Dynamically Encapsulated Programs

N. Rinetzky[1]*, A. Poetzsch-Heffter[2], G. Ramalingam[3]**, M. Sagiv[1], and E. Yahav[4]

[1] Tel Aviv University {maon,msagiv}@tau.ac.il
[2] University of Kaiserlautern poetzsch@informatik.uni-kl.de
[3] Microsoft Research India grama@microsoft.com
[4] IBM T.J. Watson Research Center eyahav@us.ibm.com

Abstract. We present a *modular* static analysis which identifies structural (shape) invariants for a subset of heap-manipulating programs. The subset is defined by means of a non-standard operational semantics which places certain restrictions on aliasing and sharing across modules. More specifically, we assume that live references (*i.e.*, used before set) between subheaps manipulated by different modules form a tree. We develop a conservative static analysis algorithm by abstract interpretation of our non-standard semantics. Our *modular* algorithm also ensures that the program obeys the above mentioned restrictions.

1 Introduction

Modern programs rely significantly on the use of heap-allocated linked data structures. In this paper, we present a novel method for automatically verifying properties of such programs in a modular fashion. We consider a program to be a collection of modules. We develop a shape (heap) analysis which treats each module separately. Modular analyses are attractive because they promise scalability and reuse.

Modular analysis [1], however, is particularly difficult in the presence of aliasing. The behavior of a module can depend on the aliasing created by clients of the module and vice versa. Analyzing a module making worst-case assumptions about the aliasing created by clients (or vice versa) can complicate the analysis and lead to imprecise results. Instead of analyzing arbitrary programs, we restrict our attention to certain "well-behaved" programs. The main idea behind our approach is to assume a modularly-checkable program-invariant concerning aliases of live intermodule references.

Motivating Example Fig. 1 shows the code of a module, m_{RP}, which serves as our running example. The code is written in a Java-like language. Module m_{RP} contains two classes: Class `R` is a class of resources to be used by clients of the module. A resource has a recursive field, `n`, which is used to link resources in an internal list. Class `RPool` is a pool of resources which stores resources using their internal list. We assume that the `n`-field is read or written only by `RPool`'s methods: `acquire`, which gets a resource out of the pool, and `release`, which stores a resource in the pool.

* Supported in part by the IBM Ph.D. Fellowship Program, and in part by a grant from the Israeli Academy of Science.
** Work done partly when the author was at IBM Research.

Typical properties we want to verify modularly are that for *any well behaved* program that uses m_{RP}, the methods of `RPool` never leak resources and never issue an acquired resource before it is released.[5] Note that these properties do not hold for arbitrary programs because of possible aliasing in the module induced by the client behavior: Consider an invocation of `p.release(r)` in a memory state in which `p` points to a non-empty resource pool. If `r` points to the head of a resource list containing more than one resource, then the tail of the list might be leaked. If, after being released into the pool that `p` points to, `r` is released into other pools, then these pools, along with the one pointed-to by `p` share (parts) of their resource lists. Note that after a shared resource is acquired from one pool, it can still be acquired from the other pools. Finally, if the resource that `r` points to is already in `p`'s pool, then `p`'s resource list becomes cyclic. A resource which is acquired from a pool whose list is cyclic, stays in the pool.

```
public class RPool {
 private R rs;
 // transferred: { e }
 public
 void release(R e){
   e.n=this.rs;
   this.rs=e;
 }
 // transferred: { }
 public R acquire(){
   R r = this.rs;
   if (r!=null) {
     this.rs=r.n;
     r.n = null; }
   else
     r = new R();
   return r;
 }}

public class R {
 R n; ... }
```

Fig. 1. Module m_{RP}.

Given a module, and the user specification for the other modules it uses, our analysis tries to verify that the given module is "well-behaved". If this verification is unsuccessful, the analysis gives up and reports that the module may not adhere to our constraints. Otherwise, the analysis computes invariants of the given module that hold in any "well-behaved" program containing the module. A program comprised only of successfully verified modules is guaranteed to be "well-behaved".

1.1 Overview

Non-standard semantics The basis for our approach is a *non-standard semantics* that captures the aliasing constraints mentioned above. In this paper, a module is a collection of type-definitions and procedures, and a component is a subheap. Our semantics represents the heap as an (evolving and changing) collection of (heap) *components*. Every component is comprised of objects whose types are defined in the same module. (We say that a component *belongs to* that module.) Note that multiple components belonging to the same module may co-exist. References between components belonging to different modules are allowed, however, the *internal structure* of a component can be accessed or modified only by the (procedures in the) module to which it belongs.[6] Components can be in two different states: *sealed* and *unsealed*. Sealed components represent encapsulated data returned by a module to its callers (and, hence, are expected to satisfy certain *module invariants*). In contrast, unsealed components are components that are currently being modified and may be in an unstable state.

At any point during program execution, the internal structure of only one component is "visible" and can be accessed or mutated, *i.e.*, only one unsealed component is

[5] Similarly, in the analysis of a client of m_{RP}, we would like to verify that the client does not use a dangling reference to a released resource. Our analysis can establish this property.

[6] A module m can manipulate a component of a module m' by an intermodule procedure call.

"visible". We refer to this component as the *current component*. The only way a sealed component can be *unsealed* (permitting its internal structure to be examined and modified) is to pass it as a parameter of an appropriate intermodule procedure call so that the component becomes part of the current component for the called procedure. Our semantics requires that all parameters and the return value(s) of intermodule procedure calls must be sealed components. For brevity, we do not consider primitive values here.

Constraints So far we have not really placed any constraints on the program. The above are standard "good modularity principles" and most programs will fit this model with minor adjustments. Before we describe the constraints we place on sharing across modules, we describe the two key issues that motivate these constraints:

1. How can we analyze a module M without using any information about the clients of M (*i.e.*, without using information about the usage context of M)?
2. When analyzing a client module C that makes use of another module M, how do we handle *intermodule* calls from C to M using only the analysis results for module M (*i.e.*, without analyzing module M again)?

We say that a component *owns* another component if it has a *live* reference (*i.e.*, used before set) to the other component. The most important constraint we place is that a component cannot be owned by two or more components. As a result, the heap (or the program state) may be seen as, effectively, a tree of components. Informally, this ensures that distinct components do not share (live) state. Furthermore, we require that all references to a component from its owner have the same target object. We call this object the component's *header*.[7] We refer to a program which satisfies these constraints as a *dynamically encapsulated* program. Recall that our analysis also verifies that a program is *dynamically encapsulated*.

In this paper, we require that the module dependency relation (see Sec. 2) be acyclic. This constraint simplifies our semantics (and analysis) as module reentrancy does not need to be considered: When a module is invoked *all* of its components are guaranteed to be sealed. We note that our techniques can be generalized to handle cyclic dependencies, provided that the ownership relation is required to be acyclic.

Benefits. The above constraints let us deal with the two issues mentioned above in a tractable way. The restriction on sharing between components simplifies dealing with intermodule calls as they cannot have unexpected side-effects: *e.g.*, an intermodule call on one component C_1 cannot affect the state of another component C_2 that is accessible to the caller. As for the first issue, *we conservatively identify all possible input states for an intermodule call by iteratively identifying all possible sealed components that can be generated by a module*.

Specification We now describe the extra specification a user must provide for the modular analysis. This specification consists of: (i) a *module specification* that partitions a program's types and procedures into modules; (ii) an annotation for every (public) procedure that indicates for every parameter whether it is intended to be "transferred" to the

[7] Note the slight difference in terminology: In ownership type systems, owners are objects and do not belong to their ownership contexts. In our approach, components are the owners; the component header belongs to the component that is dominated by the header.

callee or not; these annotations are only considered in intermodule procedure calls. A sealed component that is passed as a *transferred* parameter of an intermodule call cannot be subsequently used by the calling module (*e.g.*, to be passed as a parameter for a subsequent intermodule call). This constraint serves to directly enforce the requirement that the heap be a tree of components. For example, for `release` we specify that the caller transfer ownership only of the resource parameter.

Given the above specification, our modular analysis can automatically detect the boundaries of the heap-components and (conservatively) determine whether the program satisfies the constraints described above

Abstraction Our modular analysis is obtained as an abstract interpretation of our non-standard semantics. We use a 2-step successive abstraction. We first apply a novel *trimming abstraction* which abstracts away the contents of sealed components when analyzing a module. (Loosely speaking, only the heap structure of the current component, and the aliasing relationships between intermodule references leaving the current component, are tracked.) We then apply a *bounded* conservative abstraction of trimmed memory states. Rather than providing a new intraprocedural abstraction, we show how to *lift* existing *intra*procedural shape analyses, *e.g.*, [2–4], to obtain a modular shape abstraction (see Sec. 4). Our analysis is parametric in the abstraction of trimmed memory states and can use different (bounded) abstractions when analyzing different modules.

Analysis Our static analysis is conducted in an assume-guarantee manner allowing each module to be analyzed separately. The analysis, computes a conservative representation of every possible sealed components of the analyzed module in dynamically encapsulated programs. This process, in effect, identifies structural invariants of the sealed components of the analyzed module, *i.e.*, it infers module invariants (for dynamically encapsulated programs). Technically, the module is analyzed together with its *most-general-client* using a framework for interprocedural shape analysis, *e.g.*, [5, 6].

Extensions In this paper, we use a very conservative abstraction of sealed components and inter-component references (for simplicity). The abstraction, in effect, retains no information about the state of a sealed component (which typically belongs to other modules used by the analyzed module). This can lead to an undesirable loss in precision in the analysis (in general). We can refine the abstraction by using *component-digests* [7], which encode (hierarchical) properties of whole *components* in a typestate-like manner [8]. This, *e.g.*, can allow our analysis to distinguish between a reference to a pool of closed socket components from a reference to a pool of connected socket components.

1.2 Main Contributions

(i) We introduce an interesting class of dynamically encapsulated programs; (ii) We define a natural notion of *module invariant for dynamically encapsulated programs*; (iii) We show how to utilize dynamic encapsulation to enable modular shape analysis; and (iv) We present a modular shape analysis algorithm which (conservatively) verifies that a program is dynamically encapsulated and identifies its module invariants.

Due to space restrictions, many formal details and the possible extensions of our techniques are omitted and can be found in [9].

2 Program Model and Specification Language

Program model We analyze imperative object-based (*i.e.*, without subtyping) programs. A program consists of a collection of procedures and a distinguished `main` procedure. The programmer can also define her own types (à la `C` structs).
Syntactic domains. We assume the syntactic domains $x \in \mathcal{V}$ of variable identifiers, $f \in \mathcal{F}$ of field identifiers, $T \in \mathcal{T}$ of type identifiers, $p \in \mathcal{PID}$ of procedure identifiers, and $m \in \mathcal{M}$ of module identifiers. We assume that types, procedures, and modules have unique identifiers in every program.
Modules. We denote the module that a procedure p belongs to by $m(p)$ and the module that a type identifier T belongs to by $m(T)$. A module m_1 *depends* on module m_2 if $m_1 \neq m_2$ and one of the following holds: (i) a procedure of m_1 invokes a procedure of m_2; (ii) a procedure of m_1 has a local variable whose type belongs to m_2; or (iii) a type of m_1 has a field whose type belongs to m_2.
Procedures. A procedure p has local variables (V_p) and formal parameters (F_p), which are considered to be local variables, *i.e.*, $F_p \subseteq V_p$. Only local variables are allowed.

Specification language We expect to be given a partitioning of the program types and procedures into modules. Every procedure should have an ownership transfer specification given by a set $F_p^t \subseteq F_p$ of *transferred (formal) parameters*. (A formal parameter is a transferred parameter if it points to a transferred component in an intermodule call.) For example, `e` is `release`'s only transferred parameter, and `acquire` has none.

Simplifying assumptions We assume that procedure invocations should be *cutpoint-free* [5]. (We explain this assumption, and a possible relaxation, in Sec. 3.2.) In addition, to simplify the presentation, we make the following assumptions: (a) A program manipulates only pointer-valued fields and variables; (b) Formal parameters *cannot* be assigned to; (c) Objects of type T can be allocated and references to such objects can be *used as l-values* by a procedure p only if $m(p) = m(T)$; (d) Actual parameters to an intermodule procedure call should not be aliased and should point to a component owned by the caller. In particular, they should have a non-*null* value; and (e) The caller always becomes the owner of the return value of an intermodule procedure call.

3 Concrete Dynamic-Ownership Semantics

In this section, we define $\mathcal{DOS}$, a non-standard semantics which checks whether a program executes in conformance with the constraints imposed by the dynamic encapsulation model. ($\mathcal{DOS}$ stands for *dynamic-ownership semantics*.) $\mathcal{DOS}$ provides the execution traces that are the foundation of our analysis. For space reasons, we only discuss key aspects of the operational semantics, formally defined in [9].

$\mathcal{DOS}$ is a *store-based* semantics (see, *e.g.*, [10]). A traditional aspect of a store-based semantics is that a memory state represents a heap comprised of all the allocated objects. $\mathcal{DOS}$, on the other hand, is a *local heap* semantics [11]: A memory state which occurs during the execution of a procedure does not represent objects which, at the time of the invocation, were not reachable from the actual parameters.

$\mathcal{DOS}$ is a small-step operational semantics [12]. Instead of encoding a stack of activation records inside the memory state, as traditionally done, $\mathcal{DOS}$ maintains a *stack of program states* [9, 13]: Every program state contains a program point and a memory state. The program state of the *current procedure* is stored at the top of the stack, and it is the only one which can be manipulated by intraprocedural statements. When a procedure is invoked, the *entry memory state* of the callee is computed by a $Call$ operation according to the caller's current memory state, and pushed into the stack. When a procedure returns, the stack is popped, and the caller's *return memory state* is updated using a Ret operation according to its memory state before the invocation (the *call memory state*) and the callee's (popped) *exit memory state*.

The use of a stack of program states allows us to represent in every memory state the (values of) local variables and the local heap of just one procedure. An execution trace of a program P always begins with P's `main` procedure starts executing on an *initial memory state* in which all variables have a *null* value and the heap is empty. We say that a memory state is *reachable* in a program P if it occurs as the current memory state in an execution trace of P.

3.1 Memory States

Fig. 2 defines the concrete semantic domains and the meta-variables ranging over them. We assume Loc to be an unbounded set of locations. A value $v \in Val$ is either a location, *null*, or $\ominus$, the inaccessible value used to represent references which should not be accessed.

A memory state in the $\mathcal{DOS}$ semantics is a 5-tuple $\sigma = \langle \rho, L, h, t, m \rangle$. The first four components comprise, essentially, a 2-level store: $\rho \in \mathcal{E}$ is an environment assigning values for the variables of the *current* procedure. $L \subset Loc$ contains the locations of allocated objects. (An object is identified by its location. We interchangeably use the terms object and location.) $h \in \mathcal{H}$ assigns values to fields of allocated objects. $t \in \mathcal{TM}$ maps every allocated object to the type-identifier of its (immutable) type. Implicitly, t associates every allocated location to a module: The module that a location $l \in L$ *belongs to in memory state* σ, denoted by $m(t(l))$, is $m(t(l))$. The additional component, $m \in \mathcal{M}$, is the module of the current procedure. We refer to m as the *current module* of σ. (We denote the current module of a state σ by $m(\sigma)$.)

$l \in Loc$
$v \in Val = Loc \cup \{null\} \cup \{\ominus\}$
$\rho \in \mathcal{E} = \mathcal{V} \hookrightarrow Val$
$h \in \mathcal{H} = Loc \hookrightarrow \mathcal{F} \hookrightarrow Val$
$t \in \mathcal{TM} = Loc \hookrightarrow \mathcal{T}$
$\sigma \in \Sigma = \mathcal{E} \times 2^{Loc} \times \mathcal{H} \times \mathcal{TM} \times \mathcal{M}$

Fig. 2. Semantic domains.

Note that in $\mathcal{DOS}$, reachability, and thus domination,[8] are defined with respect to the *accessible heap*, *i.e.*, $\ominus$-valued references do not lead to any object.

[8] An object l_2 is *reachable from* (resp. *connected to*) an object l_1 in a memory state σ if there is a directed (resp. undirected) path in the heap of σ from l_1 to l_2. An object l is *reachable* in σ if it is reachable from a location which is pointed-to by some variable. An object l is a *dominator* if every access path pointing to an object reachable from l, must traverse through l.

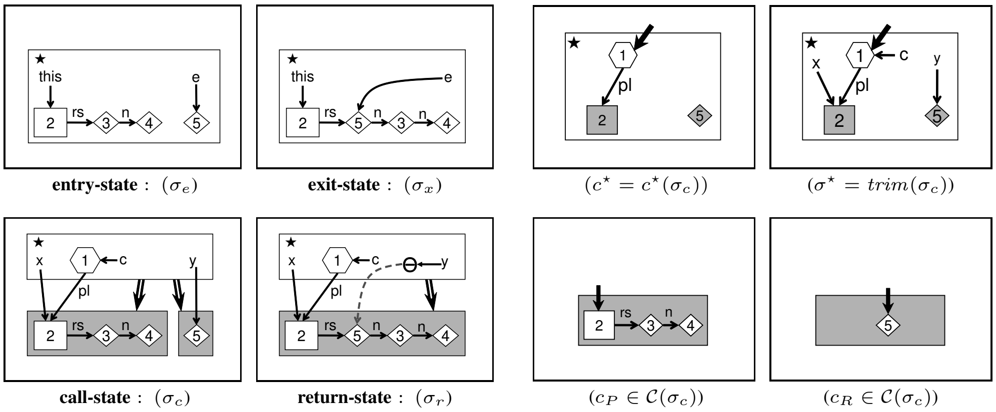

Fig. 3. $(\sigma_c, \sigma_e, \sigma_x, \sigma_r)$: $\mathcal{DOS}$ memory states occurring in an invocation of `x.release(y)` on σ_c. $(c^\star, c_P, c_R)$: The implicit components of σ_c. $(\sigma^\star)$: The trimmed memory state induced by σ_c.

Example 1. Fig. 3 (σ_c) depicts a possible $\mathcal{DOS}$ memory state that may arise in the execution of a program using the module m_{RP}. The state contains a *client* object (shown as an hexagon) pointed-to by variable `c` and having a `pl`-field pointing to a resource pool (shown as a rectangle). The resource pool, containing two resources (shown as diamonds) is also pointed-to by a variable `x`. In addition, a local variable `y` points to a resource outside the pool. (The numbers attached to nodes indicate the location of objects. The value of a (non-null) pointer variable is shown as an edge from a label consisting of the variable name to the object pointed-to by the variable. The value of a (non-null) field f of an object is shown as an f-labeled edge emanating from the object. Other graphical elements can be ignored for now.) The states σ_c and σ_e (also shown in Fig. 3), depict, respectively, the call- and the entry-memory states of an invocation of `x.release(y)` which we use as an example throughout this section. Note that σ_e represents only the values of the local variables of `release` and does not represent the (unreachable) client-object. In the return memory state of the invocation, depicted in Fig. 3 (σ_r), the dangling reference `y` has the $\ominus$-value, and the resource pool dominates the resources in its list. (The return state *does not* represent the value of `y` before the call, indicated by the dashed arrow.)

Components Intuitively, a component provides a partial view of a $\mathcal{DOS}$ memory state σ. A component of σ consists of a set of reachable objects in σ, which all belong to the same module, and records their types, their link structure, and their *spatial interface i.e.*, references to and from immediately connected objects and variables.

More formally, a component $c \in \mathcal{C} = 2^{Loc} \times 2^{Loc} \times 2^{Loc} \times \mathcal{H} \times \mathcal{TM} \times \mathcal{M}$ is a 6-tuple. A *component* $c = \langle I, L, R, h, t, m \rangle$ is a *component of* a $\mathcal{DOS}$ memory state σ if the following holds: L, the set of c's *internal objects*, contains only reachable objects in σ. $I \subseteq L$ and $R \subseteq Loc \setminus L$ constitute c's spatial interface: I records the *entry locations* into c. An object inside c is an *entry location* if it is pointed-to by a variable or by a field of a *reachable* object outside c. R is c's *rim*. An object outside c is in c's rim if it is pointed-to by a field of an object inside c. h defines the values of fields for objects inside c. We refer to a field pointing to an internal resp. rim object as an intra- resp. inter-component reference. h should be the restriction of σ's heap on L. t defines the types of the objects inside c and in its rim. t should be the restriction of σ's type map on $L \cup R$. m is c's *component module*. We say that component c belongs to m. The type of every object inside c must belong to m. (If L is empty then m must be the current module of σ.) Note that a component c records (among other things) all the aliasing information available in σ pertaining to fields of c's internal objects. For reasons explained below, we treat a variable pointing to a location outside the current component as an inter-component reference leaving the current component, and add that location to its rim (and relax the definition of a component accordingly).

Example 2. Memory state $\sigma_c = \langle \rho_c, L_c, h_c, t_c, m_c \rangle$, depicted in Fig. 3, is comprised of three components. A rectangular frame encompasses the internal objects of every component. The current component, marked with a star, belongs to m_c, the client's module. The sealed components, drawn shaded, belong to module m_{RP}. Fig. 3 ($c^\star$) depicts $c^\star = \langle I^\star, L^\star, R^\star, h^\star, t^\star, m_c \rangle$, the current component of σ_c, separately from σ_c. The client-object is the only object inside $c^\star$. It is also an entry location, *i.e.*, $I^\star = L^\star = \{1\}$. An entry location is drawn with a wide arrow pointing to it. The resource pool and the resource are rim objects, *i.e.*, $R^\star = \{2, 5\}$. Rim objects are drawn opaque. The `pl`-labeled edge depicts the only (inter-component) reference in $c^\star$. Note that $h^\star = h_c|_{\{1\}}$ and $t^\star = t_c|_{\{1,2,5\}}$. Fig. 3 ($c_P$) and ($c_R$) depict σ_c's sealed components.

The types of the reachable objects in a memory state σ induce a (unique) *implicit component decomposition* of σ: (i) a single *implicit current component*, denoted by $c^\star(\sigma)$, containing all the *reachable* objects in σ that belong to σ's current module and (ii) a set of *implicit sealed components*, denoted by $\mathcal{C}(\sigma)$, containing (disjoint subsets of) all the *other* reachable objects. Two objects *reside within* the same implicit sealed component if they belong to the same module $m_s \neq m(\sigma)$ and are connected in σ's heap via an *undirected heap path* which only goes through objects that belong to module m_s.

The component decomposition of a memory state σ induces an *implicit component (directed) graph*. The nodes of the graph are the implicit components of σ. The graph has an edge from c_1 to c_2 if there is a rim object in c_1 which is an entry location in c_2, *i.e.*, if there is a reference from an object in c_1 to an object in c_2. For simplicity, we assume that the graph is connected, and treat local variables in a way that ensures that.

Example 3. Component $c^\star$, c_P, and c_R are the implicit components of σ_c, *i.e.*, $c^\star = c^\star(\sigma_c)$ and $\{c_P, c_R\} = \mathcal{C}(\sigma_c)$. Double-line arrows depict the edges of the component graph. This graph is connected because $c^\star$'s rim contains the resource pointed-to by `y`.

From now on, whenever we refer to a component of a memory state σ, we mean an implicit component of σ, and use the term *implicit component* only for emphasis. (For formal definitions of components and of component graphs, see [9].)

Dynamically encapsulated memory state We define the constraints imposed on memory states by the dynamic encapsulation model by placing certain restrictions on the allowed implicit components and induced implicit component graphs.

Definition 1 (Dynamic encapsulation). *A $\mathcal{DOS}$ memory state $\sigma \in \Sigma$ is said to be* ***dynamically encapsulated****, if (i) the implicit component graph of σ is a directed tree and (ii) every (implicit) sealed component in σ has exactly one entry location.*

We refer to the parent (resp. child) of a component c in the component tree as the *owner* of c (resp. a subcomponent of c). We refer to the single entry location of a sealed component c in a dynamically encapsulated memory state σ as c's *header*, and denote it by $hdr(c)$. We denote the module of a component c by $m(c)$.

Invariant 1 *The following properties hold in every dynamically encapsulated $\mathcal{DOS}$ memory state $\sigma \in \Sigma$ and its implicit decomposition: (i) A local variable can only point to a location inside $c^\star(\sigma)$, the current component of σ, or to the header of one of $c^\star(\sigma)$'s subcomponents. (ii) For every component, every rim object is the header of a sealed component of σ. (iii) A field of an object in a component of σ can only point to an object inside c, or to the header of one of c's subcomponents. (iv) All the objects in a sealed component are reachable from the component's header. (v) A header dominates its reachable heap.*[8] *(vi) Every reachable object is inside exactly one component. (vii) If $c_1 \in \mathcal{C}(\sigma)$ owns $c_2 \in \mathcal{C}(\sigma)$ then $m(c_1)$* depends *on $m(c_2)$.*

$\mathcal{DOS}$ preserves dynamic encapsulation. Thus, from now on, whenever we refer to a $\mathcal{DOS}$ memory state, we mean a *dynamically encapsulated $\mathcal{DOS}$* memory state. As a consequence of our simplifying assumptions and the acyclicity of the module dependency relation, the following holds for every $\mathcal{DOS}$ memory state σ: (i) The internal objects of $c^\star(\sigma)$ are exactly those that the current procedure can manipulate without an (indirect) intermodule procedure call. (ii) The rim of $c^\star(\sigma)$ contains all the objects which the current procedure can pass as parameters to an intermodule procedure call.

3.2 Operational Semantics

Intraprocedural Statements Intraprocedural statements are handled as usual in a two-level store semantics for pointer programs (see, *e.g.*, [10]). The only unique aspect of $\mathcal{DOS}$, formalized in [9], is that it aborts if an inaccessible-valued pointer is accessed.

Interprocedural Statements $\mathcal{DOS}$ is a local-heap semantics [11]: when a procedure is invoked, it starts executing on an *input heap* containing only the set of *available objects for the invocation*. An object is *available for an invocation* if it is a *parameter object*, *i.e.*, pointed-to by an actual parameter, or if it is reachable from one. We refer to a component whose header is a parameter object as a *parameter component*.

A local-heap semantics and its abstractions benefit from not having to represent unavailable objects. However, in general, the semantics needs to take special care of available objects that are pointed-to by an access path which bypasses the parameters (*cutpoints* [11]). In this paper, we do not wish to handle the problem of analyzing programs with an unbounded number of cutpoints [11], which we consider a separate research problem. Thus, for simplicity, we require that *intramodule* procedure calls should be *cutpoint-free* [5], *i.e.*, the parameter objects should dominate[8] the available objects for the invocation. (In general, we can handle a *bounded* number of cutpoints.[9])

Fig. 4 defines the meaning of the $Call$ and Ret operations pertaining to an arbitrary procedure call $y = p(x_1, \ldots, x_k)$.

Procedure calls. The $Call$ operation computes the callee's *entry memory state* (σ_e). First, it checks whether the call satisfies our *simplifying* assumptions. In case of an intramodule procedure invocation, the caller's memory state (σ_c) is required to satisfy

[9] We can treat a bounded number of cutpoints as additional parameters: Every procedure is modified to have k additional (hidden) formal parameters (where k is the bound on the number of allowed cutpoints). When a procedure is invoked, the (modified) *semantics* binds the additional parameters with references to the cutpoints. This is the essence of [6]'s treatment of cutpoints.

$$
\begin{array}{ll}
\langle Call_{y=p(x_1,\ldots,x_k)}, \sigma_c \rangle \overset{D}{\rightsquigarrow} \sigma_e & m_c = m(p) \Rightarrow \textsc{cpf}\;\; D_{\rho_c,h_c}(dom(\rho_c), F_p) \\
\quad \sigma_e = \langle \rho_e, L_c, h_c|_{L_{rel}}, t_c|_{L_{rel}}, m(p) \rangle & m_c \neq m(p) \Rightarrow \textsc{dif}\;\; \forall 1 \leq i < j \leq k : \rho_c(x_i) \neq \rho_c(x_j) \\
\quad\quad \rho_e = [z_i \mapsto \rho_c(x_i) \mid 1 \leq i \leq k] & \qquad\qquad\qquad \textsc{loc}\;\; \forall 1 \leq i \leq k : \rho_c(x_i) \in Loc \\
\quad \text{where:}\;\; L_{rel} = R_{h_c}(\{\rho_c(x_i) \in Loc \mid 1 \leq i \leq k\}) & \\
\langle Ret_{y=p(x_1,\ldots,x_k)}, \sigma_c, \sigma_x \rangle \overset{D}{\rightsquigarrow} \sigma_r & m_c \neq m(p) \Rightarrow \textsc{own}\; \forall z \in F_p^{nt} : \rho_x(z) \in Loc \\
\quad \sigma_r = \langle \rho_r, L_x, h_r, t_r, m_c \rangle & \qquad\qquad\qquad \textsc{dom}\; \forall z \in F_p^{nt} : D_{\rho_x^{\ominus}, h_x}(F_p^{nt}, \{z\}) \\
\quad\quad \rho_r = (block \circ \rho_c)[y \mapsto \rho_x(ret)] & \\
\quad\quad h_r = (block \circ h_c|_{L_c \setminus L_{rel}}) \cup h_x & \\
\quad\quad t_r = t_c|_{L_c \setminus L_{rel}} \cup t_x & \\
\quad \text{where:}\;\; L_{rel} = R_{h_c}(\{\rho_c(x_i) \in Loc \mid 1 \leq i \leq k\}) & \\
\qquad\qquad \rho_x^{\ominus} = \rho_x[z \mapsto \ominus \mid m_c \neq m(p), z \in F_p^t] & \\
\qquad\qquad block = \lambda v \in Val. \begin{cases} \rho_x^{\ominus}(z_i) & v = \rho_c(x_i), 1 \leq i \leq k \\ v & \text{otherwise} \end{cases} &
\end{array}
$$

Fig. 4. $Call$ an Ret operations for an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ assuming p's formal variables are $z_1, \ldots, z_k$. $\sigma_c = \langle \rho_c, L_c, h_c, t_c, m_c \rangle$. $\sigma_x = \langle \rho_x, L_x, h_x, t_x, m_x \rangle$. $F_p^{nt} = \{ret\} \cup (F_p \setminus F_p^t)$. Variable `ret` is used to communicate the return value. We use the following functions and relations, formally defined in [9]: $R_h(L)$ computes the locations which are reachable in heap h from the set of locations L. The auxiliary relation $D_{\rho,h}(V_I, V_D)$ holds if the set of objects pointed-to by a variable in V_D, according to environment ρ, dominates the part of heap h reachable from them, with respect to the objects pointed-to by the variables in V_I.

the domination condition ($\textsc{cpf}$) ensuring cutpoint-freedom. Intermodule procedure calls are invoked under even stricter conditions which are fundamental to our approach: Every parameter object must dominate the subheap reachable from it. This ensures that distinct components are unshared. However, there is no need to check these conditions as they are invariants in our semantics: Inv. 1(i,iv,v) ensures that every parameter object to an intermodule procedure call is a header which dominates its reachable heap. (Note that Inv. 1(iv) can be exploited to check whether an object is a dominator by only inspecting access paths traversing through its component.) Thus, only our simplifying assumptions pertaining to non-nullness ($\textsc{loc}$) and non-aliasing of parameters ($\textsc{dif}$) need to be checked.

The entry memory state is computed by binding the values of the formal parameters in the callee's environment to the values of the corresponding actual parameters; projecting the caller's heap and type map on the available objects for the invocation; and setting the module of the entry memory state to be the module of the invoked procedure.

Note that in intermodule procedure calls, the change of the current module implicitly changes the component tree: all the available objects for the invocation which belong to the callee's module constitute the callee's current component. By Inv. 1 (vi,vii), these objects must come from parameter components.

Example 4. Fig. 3 (σ_e) shows the entry memory state resulting from applying the $Call$ operation pertaining to the procedure call `x.release(y)` on the call memory state σ_c, also shown in Fig. 3. All the objects in σ_e belong to m_{RP}, and thus, to its current component. Note that the latter is, essentially, a fusion of c_P and c_R, the sealed components in σ_c.

Note: The current component of a $\mathcal{DOS}$ memory state $\sigma \in \Sigma$ is the root of the component tree induced by the *local heap* represented in σ. In a *global heap*, this current component might have been one or more non-root subcomponents of a larger component-tree which is only partially visible to the current procedure. For example, the current component of the client procedure is not visible during the execution of `release`.

Procedure returns. The caller's return memory state (σ_r) is computed by a Ret operation. When an *intermodule* procedure invocation returns, Ret first checks that in the exit memory state (σ_x) every non-transferred formal parameter points to an object ($\textsc{own}$) which dominates its reachable subheap ($\textsc{dom}$). This ensures that returned components are disjoint and, in particular, that the procedure's execution respected its ownership transfer specification. (Here we exploit simplifying assumption (b) of Sec. 2.)

Ret updates the caller's memory state (which reflects the program's state at the time of the call) by carving out the input heap passed to the callee from the caller's heap and replacing it instead with the callee's (possibly) mutated heap. In $\mathcal{DOS}$, an object never changes its location and locations are never reallocated. Thus, any pointer to an available object in the caller's memory state (either by a field of an unavailable object or a variable) points after the replacement to an up-to-date version of the object.

Most importantly, the semantics ensures that any future attempt by the caller to access a transferred component is foiled: We say that a local variable of the caller is *dangling* if, at the time of the invocation, it points to (the header of) a component transferred to the callee. A pointer field of an object in the caller's memory state which was unavailable for the invocation is considered to be *dangling* under the same condition. The semantics enforces the transfer of ownership by *blocking*: assigning the special value $\ominus$ to every dangling reference in the caller's memory state. (Blocking also occurs when an *intra*module procedure invocation returns to propagate ownership transfers done by the callee.) Note that cutpoint-freedom ensures that the only object that separate the callee's heap from the caller's heap are parameter objects. Thus, in particular, the only references that might be blocked point to parameter objects.

When an intermodule call returns, and the current module changes, the component tree is changed too: The callee's current component may be split into different components whose headers are the parameter objects pointed-to by non-transferred parameters. These components may be different from the (input) parameter components.

Example 5. Fig. 3 (σ_r) depicts the memory state resulting from applying the Ret operation pertaining to the procedure call `x.release(y)` on the memory state σ_c and σ_x, also shown in Fig. 3. The insertion of the resource pointed-to by `y` at the call-site into the pool has (implicitly) fused the two m_{RP}-components. By the standard semantics, `y` should point to the first resource in the list (as indicated by the dashed arrow). This would violate dynamic encapsulation. $\mathcal{DOS}$, however, utilizes the *ownership specification* to block `y` thus preserving dynamic encapsulation.

3.3 Observational Soundness

We say that two values are *comparable* in $\mathcal{DOS}$ if neither one is $\ominus$. We say that a $\mathcal{DOS}$ memory state σ is *observationally sound* with respect to a standard semantics σ_G if every pair of access paths that have comparable values in σ, has equal values in σ iff they have equal values in σ_G. $\mathcal{DOS}$ *simulates* the standard 2-level store semantics:

Executing the same sequence of statements in the $\mathcal{DOS}$ semantics and in the standard semantics either results in a $\mathcal{DOS}$ memory states which is observationally sound with respect to the resulting standard memory state, or the $\mathcal{DOS}$ execution gets *stuck* due to a constraint breach (detected by $\mathcal{DOS}$). A program is *dynamically encapsulated* if it does not have have an execution trace which gets stuck. (Note that the initial state of an execution in $\mathcal{DOS}$ is observationally sound with respect to its standard counterpart).

Our goal is to detect structural invariants that are true according to the *standard semantics*. $\mathcal{DOS}$ acts like the standard semantics as long as the program's execution satisfies certain constraints. $\mathcal{DOS}$ enforces these restrictions by blocking references that a program should not access. Similarly, our analysis reports an invariant concerning equality of access paths only when these access paths have comparable values.

An invariant concerning equality of access paths in $\mathcal{DOS}$ for a dynamically encapsulated program is also an invariant in the standard semantics. This makes abstract interpretation algorithms of $\mathcal{DOS}$ suitable for verifying data structure invariants, for detecting memory error violations, and for performing compile-time garbage collection.

4 Modular Analysis

This section presents a conservative static analysis which identifies conservative *module invariants*. These invariants are true in *any* program according to the $\mathcal{DOS}$ semantics and in *any dynamically encapsulated* programs according to the standard semantics.

The analysis is derived by two (successive) abstractions of the $\mathcal{DOS}$ semantics: The *trimming semantics* provides the basis of our *modular* analysis by representing only components of the analyzed module. The *abstract trimming semantics* allows for an effective analysis by providing a *bounded* abstraction of trimmed memory states (utilizing existing *intra*procedural abstractions).

Module Invariants A *module invariant* of a module m is a property that holds for all the components that belong to m when they are not being used (*i.e.*, for sealed components). Our analysis finds module invariants by computing a conservative description of the set of all possible sealed components of the module. More formally, the *module invariant of module m for type T*, denoted by $[\![Inv_m\ T]\!] \subseteq 2^{\mathcal{C}}$, is a set of sealed components of module m whose header is of type T: a sealed component c is in $[\![Inv_m\ T]\!]$ iff there exists a reachable $\mathcal{DOS}$ memory state σ in some program such that $c \in \mathcal{C}(\sigma)$.

For example, the module invariant of module m_{RP} for type `RPoll` in our running example is the set containing all resource pools with a (possibly empty) *acyclic* finite list of resources. The module invariant of module m_{RP} for type `R` is the singleton set containing a single resource with a *nullified* `n`-field: An acquired resource always has a *null*-valued `n`-field and a released resource is inaccessible.

Trimming semantics The trimming semantics represents only the parts of the heap which belong to the current module. In particular, it abstracts away all information contained in sealed components and the shape of the component tree.

More formally, the *domain of trimmed states* is $\Sigma^{\star} = \mathcal{E} \times \mathcal{C}$. The *trimmed state induced by a $\mathcal{DOS}$ memory state $\sigma \in \Sigma$*, denoted by $trim(\sigma)$, is $\langle \rho, c^{\star}(\sigma) \rangle$. (For example, Fig. 3 ($\sigma^{\star}$) depicts the trimmed memory state induced by the $\mathcal{DOS}$ memory state shown in Fig. 3 (σ_c).) We say that two trimmed memory states are *isomorphic*, denoted by $\sigma_1^\star \sim \sigma_2^\star$, if $\sigma_1^\star$ can be obtained from $\sigma_2^\star$ by a consistent location renaming. A trimmed memory state $\sigma^\star$ *abstracts* a $\mathcal{DOS}$ memory state σ if $\sigma^\star \sim trim(\sigma)$.

A trimmed memory state contains enough information to determine the induced effect [14] under the trimming abstraction of intraprocedural statements and intramodule $Call$ and Ret operations by applying the statement to *any* memory state it represents. Intuitively, the reason for this uniform behavior is that the aforementioned statements are indifferent to the *contents* of sealed components: They only consider the values of fields of objects inside the current component (inter-component references included).

Analyzing intermodule procedure calls. The main challenge lies in the handling of intermodule procedure calls: Applying the induced effect of $Call$ is challenging because the *most important* information required to determine the input heap of an intermodule call is the contents of parameter components. However, this is exactly the information lost under the *trimming abstraction* of the call memory state. Applying the induced effect of Ret operations pertaining to intermodule procedure calls is challenging as it considers information about the contents of heap parts manipulated by *different* modules.

We overcome the challenge pertaining to $Call$ operations by utilizing the fact that $\mathcal{DOS}$ always changes components as a whole, *i.e.*, there is no sharing between components, thus changes to one component cannot affect *a part* of the internal structure of another component. In particular, we are *anticipating the possible entry memory states of an intermodule procedure call*: In the $\mathcal{DOS}$ semantics, the current component of an entry memory state to an intermodule procedure call is comprised, essentially, as a *necessarily* disjoint union of parameter components. Note that components are sealed only when an intermodule procedure call returns. Furthermore, the only way a sealed component can be mutated is to pass it back as a parameter to a procedure of its own module. Thus, a partial view of the execution trace, which considers only the executions of procedures that belong to the analyzed module, and collects the sealed components generated when an intermodule procedure invocation returns, can (conservatively) anticipate the possible input states for the next intermodule invocations. Specifically, *only* a combination of *already generated sealed components* of the module can be the component parameters in an intermodule procedure invocation.

We resolve Ret's need to consider components belonging to different modules utilizing the ownership transfer specification and the limited effect of intermodule procedure invocations on the caller's current component: The only effect an intermodule procedure call has on the current component of the caller is that (i) dangling references are blocked and (ii) the return value is assigned to a local variable. (By our simplifying assumptions, the return value must point either to a parameter object or to a component not previously owned by the caller. The latter case amounts to a new object in the rim of the caller's current component). Given a sound ownership specification for the invoked procedures we can apply this effect directly to the caller's memory state. This approach can be generalized (and made more precise) to handle richer specifications concerning, *e.g.*, nullness of parameters, aliasing of parameters (and return values), and digests.

Abstract trimming semantics We provide an effective conservative abstract interpretation [14] algorithm which determines module invariants by devising a bounded abstraction of trimmed memory states. Rather than providing a new intraprocedural abstraction and analyses, we show how to *lift* existing *intra*procedural shape analyses to obtain a modular shape abstraction. An abstraction of a trimmed memory state, being comprised of an environment of a single procedure and a subheap, is very similar to an abstraction of a standard two-level store. The additional elements that the abstraction needs to track is a bounded number of entry-locations and a distinction between internal objects and rim objects. In addition, the abstract domain, expected to support operations pertaining to basic pointer manipulating statements, should be extended to allow for: checking if a $\ominus$-valued reference is accessed; the operations required for cutpoint-free local-heap analysis: carving out subheaps reachable from variables and combining disjoint subheaps; and the ability to answer queries regarding domination by variables. The only additional operation required to implement our analysis is of *blocking*, *i.e.*, setting the values of all reference pointing to a given variable-pointed object to $\ominus$. The abstract domains of [2–4], which already support the operations required for performing standard local-heap cutpoint-free analysis, can be extended with these operations.

Modular analysis We conduct our modular static analysis by performing an interprocedural analysis of a module together with its *most-general-client*. The most-general-client simulates the behavior of an arbitrary dynamically encapsulated (*well behaved*) client. Essentially, it is a collection of non-deterministic procedures that execute arbitrary sequences of procedure calls to the analyzed module. The parameters passed to these calls also result from an arbitrary (possibly recursive) sequence of procedure calls. The most-general client exploits the fact that *different components are effectively disjoint* to separately create the value of every parameter passed to an intermodule procedure call. Thus, any conservative interprocedural analysis of the most-general client (which uses an extended abstract domain, as discussed above, and utilizes ownership specification to determine the effect of intermodule procedure calls made by the analyzed module) can modularly detect module invariants. In particular, the analysis can be performed by extending existing interprocedural frameworks for interprocedural shape analysis, *e.g.*, [5, 6]. Note that during the analysis process we also find conservative *module implementation invariants*: Properties that hold for all possible current components at different program points inside the component in every possible execution. [9] provides a scheme for constructing the most-general-client of a module. ([9] also provides a characterization of the module invariants based on a fixpoint equation system).

5 Related Work

A distinguishing aspect of our work is that we integrate a shape analysis with encapsulation constraints. Our work presents a nice interplay between encapsulation and modular shape analysis: it uses dynamic encapsulation to enable modular shape analysis, and uses shape analysis to determine that the program is dynamically encapsulated. In this section, we review some closely related work to both aspects of our approach. More discussion on related work can be found in [9].
Modular static analysis. [1] describes the fundamental techniques for modular static program analysis. These techniques allow to compose separate analyses of different program parts. We use their techniques, in particular, we use simple *user provided interfaces* to communicate the (limited) effect of mutations done by different modules.

Modular heap analysis. [15] presents a modular analysis which infers class invariants based on an abstraction of program traces. [16] is an extension which handles subtyping. The determined invariants concern values of atomic fields of objects of the analyzed class and of subobjects, provided that they are never leaked to the context, *e.g.*, passed as return values. [17] modularly determines invariants regarding the value of an integer field and the length of an array field of the *same* object. Our analysis, computes shape invariants of subheaps comprised of objects that may be passed as parameters
Interprocedural shape analysis. [18, 19] utilize user-specified pre- and post- conditions to achieve modular shape analysis which can handle a bounded number of flat set-like data structures. It allows objects to be placed in multiple sets. In our approach, an object can be placed only in a single separately-analyzed but arbitrarily-nested set. Other interprocedural shape analysis algorithms *e.g.*, [5, 6, 11, 20–22], compute procedure summaries, but are not modular. [22] tracks properties of single objects. The other algorithms abstract whole local heaps. Our abstraction, on the other hand, represent only a part of the local heap (*i.e.*, only the current component). We note that the aforementioned approaches do not require a user specification, which we require.
Encapsulation. Deep ownership models structure the heap into a tree of so-called *owner contexts* (see [23] for a survey). Our module-induced decomposition of a memory state into a tree of components is similar to the package-induced partitioning of a memory state into a tree of memory-regions in [24]. Our constraints are similar to external uniqueness [25], which requires that there be a *unique* reference pointing to an object from outside its (transitively) owned context. Our ownership specification is also in the spirit of [25]'s destructive reads and borrowing. [26] uses shape analysis to modularly verify (specified) uniqueness of a *live* reference to an *object*. Our use of sealed and unsealed components is close to the use of packed and unpacked owner contexts in Boogie [27, 28]. The latter, however, can handle reentrancy. The central difference between the approaches is that our techniques infer module invariants whereas Boogie verifies class invariants provided by the programmer.
Local reasoning. [29] and [30] allow to modularly conduct local reasoning [10] about abstract data structures and abstract data types with inheritance, respectively. The reasoning requires user-specified resource invariants and loop invariants. Our analysis automatically infers these invariants based on an ownership transfer specification (and an instance of the bounded parametric abstraction). [30], however, allows for more sharing than in our model. Our use of rim-objects (resp. abstract sealed components) is analogous to [30]'s use of *abstract predicates*' names (resp. resource invariants).

Conclusions Our long term research goal is to devise precise and efficient static shape analysis algorithms which are applicable to realistic programs. We see this work as an important step towards a modular shape analysis. While the ownership model is fairly restrictive with respect to the coupling between separate components, it is very permissive about what can happen inside a single component. This model is also sufficient to express several, natural, usage constraints that arise in practice. (In particular, when accompanied with digests.) We believe that our restrictions can be relaxed to help address a larger class of programs. We plan to pursue this line of research in future work.

Acknowledgments. We are grateful for the helpful comments of T. Lev-Ami, R. Manevich, S. Rajamani, J. Reineke, G. Yorsh, and the anonymous referees.

References

1. Cousot, P., Cousot, R.: Modular static program analysis, invited paper. In: CC. (2002)
2. Lev-Ami, T., Immerman, N., Sagiv, M.: Abstraction for shape analysis with fast and precise transformers. In: CAV. (2006)
3. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: VMCAI. (2005)
4. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS. (2006)
5. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: SAS. (2005)
6. Gotsman, A., Berdine, J., Cook., B.: Interprocedural shape analysis with separated heap abstractions. In: SAS. (2006)
7. Rinetzky, N., Ramalingam, G., Sagiv, M., Yahav, E.: Componentized heap abstractions. Tech. Rep. 164, Tel Aviv University (2006)
8. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. **12**(1) (1986) 157–171
9. Rinetzky, N., Poetzsch-Heffter, A., Ramalingam, G., Sagiv, M., Yahav, E.: Modular shape analysis for dynamically encapsulated programs. Tech. Rep. 107, Tel Aviv University (2006)
10. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: LICS. (2002)
11. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL. (2005)
12. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
13. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: CC. (1992)
14. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: POPL. (1977)
15. Logozzo, F.: Class-level modular analysis for object oriented languages. In: SAS. (2003)
16. Logozzo, F.: Automatic inference of class invariants. In: VMCAI. (2004)
17. Aggarwal, A., Randall, K.: Related field analysis. In: PLDI. (2001)
18. Lam, P., Kuncak, V., Rinard, M.: Hob: A tool for verifying data structure consistency. In: CC (tool demo). (2005)
19. Wies, T., Kuncak, V., Lam, P., Podelski, A., Rinard, M.: Field constraint analysis. In: VMCAI. (2006)
20. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. In: SAS. (2004)
21. Chong, S., Rugina, R.: Static analysis of accessed regions in recursive data structures. In: SAS. (2003)
22. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: POPL. (2005)
23. Noble, J., Biddle, R., Tempero, E., Potanin, A., Clarke, D.: Towards a model of encapsulation. In: IWACO. (2003)
24. Zhao, T., Noble, J., Vitek, J.: Scoped types for real-time java. In: RTSS. (2004)
25. Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: ECOOP. (2003)
26. Boyland, J.: Alias burying: unique variables without destructive reads. Softw. Pract. Exper. **31**(6) (2001) 533–553
27. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3**(6) (2004) 27–56
28. Leino, K.R.M., Müller, P.: A verification methodology for model fields. In: ESOP. (2006)
29. O'Hearn, P., Yang, H., Reynolds, J.: Separation and information hiding. In: POPL. (2004)
30. Bierman, G., Parkinson, M.: Separation logic and abstractions. In: POPL. (2005)