# Modular Verification with Shared Abstractions

Uri Juhasz *
Tel Aviv University
urijuhasz@tau.ac.il

Noam Rinetzky †
Queen Mary University of London
maon@dcs.qmul.ac.uk

Arnd Poetzsch-Heffter
Kaiserslautern University
poetzsch@informatik.uni-kl.de

Mooly Sagiv
Tel Aviv University
msagiv@tau.ac.il

Eran Yahav
IBM T.J. Watson Research Center
eyahav@us.ibm.com

## Abstract

Modular verification of shared data structures is a challenging problem: Side-effects in one module that are observable in another module make it hard to analyze each module separately. We present a novel approach for modular verification of shared data structures. Our main idea is to verify that the inter-module sharing is restricted to a user-provided specification which also enables the analysis to handle side-effects.

## 1. Introduction

In object-based programming languages, it is natural to implement abstract data types using pointers. Pointers permit situations in which the representations of apparently distinct data structures share objects in a way which is observable to their clients. For example, a client may perform operations on a *wrapper* [10] object, yet have access to, and operate directly on, the underlying object of the wrapper. This form of sharing drastically complicates the task of (modular) reasoning. In particular, naive reasoning in terms of the wrapper's specification may be unsound because of updates to the underlying object that can change the abstract state as specified by the wrapper. Systems for modular reasoning based on hierarchical ownership (see, e.g., [22, 23, 9], for surveys), do not apply to programs in which internal data structures that come from different modules are shared. However such programs are common in practice.

In this paper, we present a novel *modular approach* for verifying partial correctness of *shared* data structures for a restricted class of programs. Historically, our starting point is Hoare's seminal work on modular verification of abstract data types [11]. Hoare's approach for modular verification is centered around hiding the implementation details of data structures using an *abstract value* which records the data structure's abstract (i.e., client-observable) state. The abstract value is computed using an implementation-dependant *representation function*. Our main insight is that we can allow for modular reasoning involving shared data structures, which Hoare forbids, by combining the following two ideas:

(i) a *controlled exposure of the inter-module sharing patterns*: The user specifies the permitted inter-module sharing, i.e., the allowed sharing of internal data structures that come from different modules. This information allows to determine which (abstract value of which) data structures may be affected when another data structure is modified, and

(ii) a user provided *aggregate model function* which allows to update the abstract value of a data structure due to an indirect change. The key idea behind the aggregate model function is that it can (conservatively) compute the new abstract value of a data structure using its old abstract values and the abstract values of its internal data structures. This compositional way for computing abstract values is possible thanks to the information provided by the (controlled) exposure of the inter-module sharing.

***Main Contributions.*** The main contributions of this paper can be summarized as follows:

1. We present a novel approach for manual and automatic modular verification of shared composite data structures. The additional proof burden required in our approach is proportional to the expected "degree of sharing".
2. We define a form of specification and enumerate the proof obligations which suffice for the modular verification of a module. The specification allows to control the permitted inter-module sharing patterns.
3. Based on our approach, we develop a non-standard semantics specifically designed as a foundation for modular analysis: The semantics allows to execute a module $m$ using the *specification* of the modules used by $m$, and aborts when $m$ violates its specification.
4. We develop a modular static analysis by abstract interpretation of our non-standard semantics.

***Outline*** The rest of the paper is organized as follows: Sec. 2 and 3, present an extended informal overview of our approach. Sec. 4 describes our programming model. Sec. 5 lists the main restrictions on the class of programs that our approach can handle. Sec. 6 defines the required specification and the proof obligations

* Supported by the German-Israeli Foundation for Scientic Research and Development (G.I.F.).

† This work was done partly when Noam Rinetzky was at Tel Aviv University and supported by the German-Israeli Foundation for Scientic Research and Development (G.I.F.).

Copyright is held by the author/owner(s).
*FOOL '09* 24 January, Savannah, Georgia, USA.
ACM .

in our approach. Sec. 7 presents our non standard semantics. Sec. 8 shortly discusses a static analysis based on an abstract interpretation of our non-standard semantics. Sec. 9 discusses possible ways to overcome some of our limitations. Sec. 10 reviews related works and Sec. 11 concludes.

For clarity, we omit some of the technical details, which can be found in [12].

## 2. Running Example

Fig 1 shows our running example. The code is written in a Java-like language. The figures shows three modules: a `Client` module, a `KeySet` module, and a `Map` module. Our running example, uses a single instance of each module: a client using a key-set and a map. We refer to a data structure instance as a *component* and to an internal data structure as a *subcomponent*. We refer to data structures sharing an internal data structure as *siblings*. In this example, the key-set and the client are sibling data structures: the map is a *shared subcomponent* of both of them.

The client initialization procedure, `Client`, shown in Fig 1(a), uses the *map* module to associate keys (integers) to values (floats). The set reflects the domain of the map: In the spirit of the `keySet()` method of a Java's maps, when the set is constructed, it is linked to a map and provides a view of the keys contained in this map. For example, the third invocation of `insert` associating key `9` to value `1.0`, has a *side-effect* on the key-set, it makes `9` a member of the key-set.

The `Keyset` class, shown in Fig 1(b), stores in field `map` a reference to the set's underlying map. This field is initialized upon construction and utilized to delegate methods invoked on the set to its map. Note that after the key-set is initialized, the client keeps– and uses–a reference to its underlying map.

Fig 1(c) shows the signature of the methods in the `Map` class, whose actual implementation, using a binary search tree, is irrelevant for the purpose of this paper, and thus omitted.

***Verification Goal*** Our goal is to verify each module separately. In the following, we focus on verifying that all the instances of the `Keyset` class comply with their specification (explained below). In our approach, we also modularly extracts information about the dependencies between `Keyset` and `Map` in a way that allows to verify that all the instances of the client satisfy the assertions. The main challenge we face is the propagation of side-effects due to methods invoked on shared subcomponents. For example, verifying the first assert statement stating that `9` is a member of the key-set requires propagating the side-effects of the third invocation of `insert` on the map to the key-set.

## 3. Our Approach

In this section, we provide an informal overview of our approach. Our approach builds on the seminal work of Hoare [11] which allows for manual modular verification of abstract data structures. Hoare proposes the use of a *representation function* which maps the representation of the data structure's state to an *abstract value*. The specification of the data structure's procedures is given in terms of abstract values. This allows a client of the data structure to be independent of the data structure's actual implementation. Hoare's work does not support pointers and dynamically allocated memory. However, it can be extended naturally to nested (encapsulated) data structures, i.e., where subcomponents can be organized only in the form of a tree.

### 3.1 Controlled exposure of sharing

In this work, we provide an approach for manual and automatic modular verification of composite data structures in the presence of shared subcomponents. In this setting, any sound analysis needs to consider side-effects due to sharing. *Our main idea is to have a controlled exposure of the sharing of subcomponents instead of forbidding it or ignoring it.*

We support shared subcomponents with the aid of a user specification that: (i) exposes subcomponents that can be referenced by other data structures; (ii) provides the (lack of) effect of every operation invoked on the data structure not only on the abstract value of the data structure, but also *on the abstract value of any subcomponent that it may share with others*. A key feature of this specification is that it exposes the effect of a procedure call on shared *subcomponents*, but it need *not* specify the effect of the procedure on sibling data structures, thus enabling modular reasoning.

### 3.2 The aggregate model function

The main challenge that we face is the tracking of indirect changes to the abstract values of sibling data structures. Such changes may occur, e.g., when a (shared) subcomponent of two data structures is modified.

We address the above challenge using a user-specified *aggregate model function*. The aggregate model function allows us to compute a conservative approximation of the (side-affected) data structure's abstract value after an indirect change. It computes the (updated) data structure's abstract value based on the abstract values of its internal representation (including unshared subcomponents) and the abstract values of its shared subcomponents. When analyzing sibling data structures the analysis tracks which subcomponents are shared and by whom. When the abstract value of a shared subcomponent is modified, the analysis *updates* the abstract values of the affected data structures using their *aggregate model functions*.

Our approach uses two kinds of additional fields to record the abstract value of a data structure and to define the aggregate model function. *Model fields* are used to record the abstract values of data structures. *Model pivot fields* are used to record the inter-component reference structure.

**Example 3.1.** Fig 2(c) shows the (user-provided) interface specification of the `Map` class. A map implements a partial function from integers to reals. The abstract value of a map is a function from integers to floats. We use the model field *map* to record this value. The map is an example of a *fully-encapsulated* data structure, i.e., one which does not expose or share its internal data structures.

The `Keyset` data structure, whose interface specification is shown in Fig 2(a), is an example of a data structure implementation which allows a controlled exposure of subcomponents. The abstract value of the `Keyset` is a set of integers. The model field *keys* records this value. In addition, the `Keyset` exposes a *model pivot field*. The model pivot field is a special sort of a model field whose value is a reference to a subcomponent. It is initialized with the map parameter when a key-set is constructed and allows the exposure of the effect of the key-set's method's on this (possibly) shared component. For example, the specification of the `Keyset`'s `remove` method exposes the fact that removing a key from a key-set also removes it from the (abstract value of the) map.

The aggregate model function of the *keys* model field of the `Keyset` class is $keys_m$. It is shown in Fig 2(a). It specifies how to compute the value of the model field *keys* when the abstract value of its underlying map changes. Specifically, it specifies that the value of the key-set's model field *keys* should be updated to the domain of the abstract value of the map referenced by the key-set's model pivot field *amap*. Recall that the client of the map (and of the key-set) is aware of the connection between the key-set and the map because the key-set exposes its pivot field *amap*.

```
Module Client
  class Client{
    Map m;
    Keyset s;
   Client(){
    m = new Map();
    s = new Keyset(m);
    m.insert(3,4.0);
    m.insert(2,5.0);
    m.insert(9,1.0);
    assert s.isMember(9);
    s.remove(9);
    assert m.find(9) == NOTFOUND;
   }
  }
```

(a) Client program

```
Module Keyset
  class Keyset {
   Map map;
   Keyset(Map smap){
    map=smap;
   }
   bool isMember(int k){
    return map.find(k)
        != NOTFOUND;
   }
   void remove(int k){
    map.remove(k);
   }
  }
```

(b) Keyset class

```
Module Map
  class Map {
   static int NOTFOUND=-1;
   ...
   Map(){...}
   void insert(int k, float val) {
    ...
   }
   float find(int k) {
    ...
   }
   void remove(int k) {
    ...
   }
  }
```

(c) A Map class

**Figure 1.** Running Example (code).

**model** $keys \in 2^{\mathbb{N}}$

**model pivot** `Map` *amap*;
**model function** $keys_m \equiv keys = \mathbf{dom}(amap.map)$

**pre** $m \neq null$
**post** $ret\ \ fresh$
$ret.amap = m$
$ret.keys = \mathbf{dom}(m.map)$
`Keyset(Map m)`

**post** $ret = k \in keys$
`bool isMember(int k)`

**post** $keys' = keys \setminus \{k\}$
$amap.map' = amap.map|_{\mathbf{dom}(amap.map)\setminus\{k\}}$

`void remove(int k)`

(a) `Keyset` interface specification

**rep pivot**
$amap = map$

**rep function**
$keys = \mathbf{dom}(map.map)$

(b) `Keyset` internal specification

**model** $map \in \mathbb{N} \hookrightarrow_{fin} \mathbb{R}$

**post** $ret.map = \emptyset$
$ret\ \ fresh$
`Map()`

**pre** $k \geq 0$
**post** $map' = map[k \mapsto v]$
`void insert(int k, float v)`

**post** $ret = k \in dom(map)\ ?\ map(k) : \mathrm{NOTFOUND}$
`float find(int k)`

**post** $map' = map|_{\mathbf{dom}(map)\setminus\{k\}}$

`void remove(int k)`

(c) `Map` interface specification

**Figure 2.** Running example (specification). Fields which are not mentioned, are assumed not to be modified.

### 3.3 Non-standard semantics

We use our approach to design a static analysis which verifies that a program is correct according to its specification. The analysis is based on an abstract interpretation [6] of $\mathcal{LHM}$, a non-standard *componentized local-heap hybrid modular semantics*. $\mathcal{LHM}$ abstracts the standard semantics (instrumented to also track the values of model fields and model pivot fields) by abstracting away the inner structure of subcomponents: $\mathcal{LHM}$ records only the abstract values of the subcomponents and their inter-component reference structure. Technically, $\mathcal{LHM}$ keeps track only of the model fields of subcomponents and of their pivot model fields.

**Example 3.2.** ***(Instrumented Standard Memory States).*** Fig 3 ($s_c$) depicts a possible memory state that may arise in the execution of the running example before the third invocation of `insert` according to an (instrumented) standard semantics which is instrumented to track also the values of model fields. The state contains a client object (depicted as a diamond). The client object has two fields, depicted as labeled edges: An `s`-field pointing to a key-set, depicted as a rectangle, and an `m`-field pointing to a map, depicted as a hexagon. The key-set's `map`-field also points to the map object. The map associates 2 to 5.0 and 3 to 4.0. Recall that the map is implemented as a binary search tree. Each tree node records an association of a key to a value. Tree nodes are depicted as an oval. Each node is annotated with the key and the value which are stored in the node ($d = \ldots$ and $v = \ldots$, respectively). For example, the node at the root of the tree (i.e., the node pointed to by the `root` field of the map object) records the association of key 3 with value 4.0. The rectangular frames depict the *component-decomposition* of the memory state. We say that two objects *reside within* the same implicit component if they belong to the same module and are connected in the heap via an *undirected heap path* which only goes through objects that belong to the same module. We say that a component is an *unsealed component* if it is the current component (its head is pointed to by this) and a *sealed component* otherwise. The memory state $s_c$ is comprised of three components. The keyset component and the map component are sealed.

The (instrumented) standard semantics records the values of model fields and model pivot fields only for object in sealed components. The value of the model pivot field *amap* is depicted as a edge emanating from the set object to the map object annotated with name of the pivot field written in italics. The values of the model fields *keys* and *map* are depicted inside the key-set and map

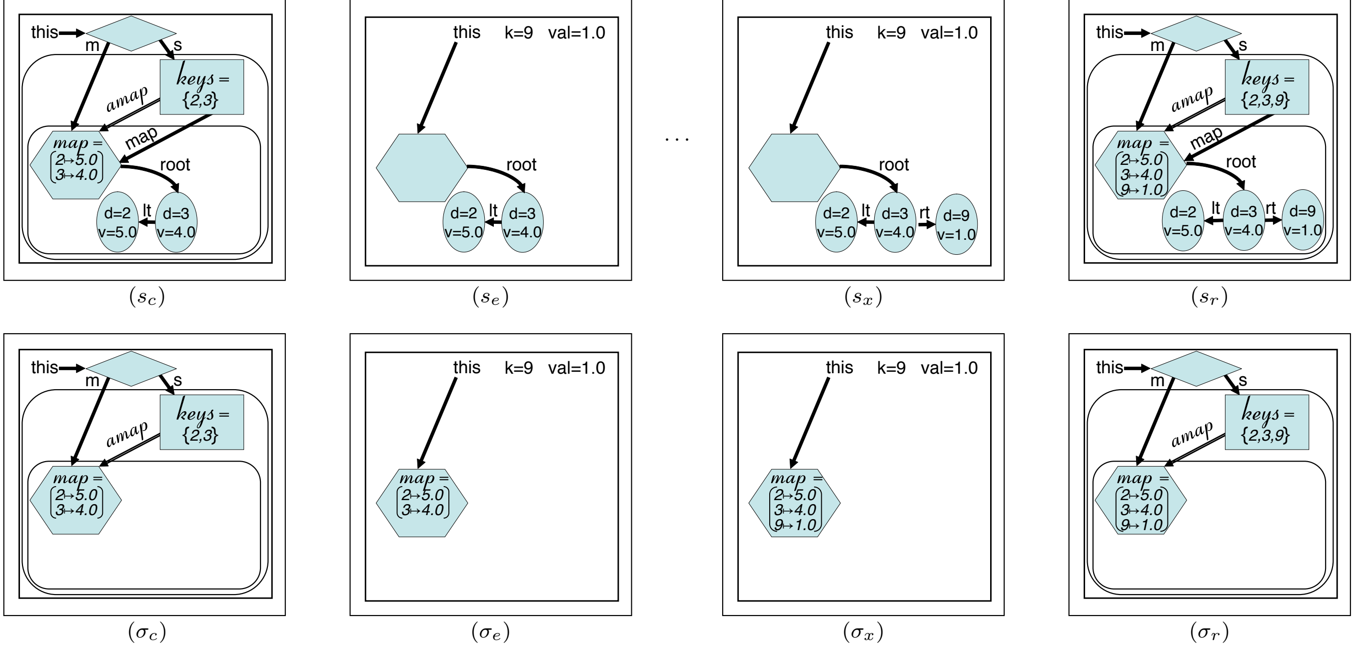


**Figure 3.** Possible memory states. $(s_c, s_e, s_x, s_r)$ memory states occurring in an invocation of `m.insert(9,1.0)` on $s_c$ according to the instrumented standard semantics. $(\sigma_c, \sigma_e, \sigma_x, \sigma_r)$ memory states occurring in an invocation of `m.insert(9,1.0)` on $\sigma_c$ according to the $\mathcal{LHM}$ semantics.

object, respectively. For example, the (local) model field *map* of the map is the partial function $[2 \mapsto 5.0, 3 \mapsto 4.0]$.

Fig 3 $(s_e)$ depicts the part of the memory state which the `insert` procedure can reach after being invoked for the third time. This part of the memory state is transformed into the one shown in Fig 3 $(s_x)$ by the execution of the `insert` procedure. The standard memory state shown in Fig 3 $(s_r)$ results when the call to `insert` returns to the `Client`. Note that while `insert` could not directly modify fields in object which it could not reach, it does have a side effect on the abstract value of the key-set's *keys* model field.

**Example 3.3.** ***($\mathcal{LHM}$ Memory States).*** The $\mathcal{LHM}$ memory states depicted in Fig 3 $(\sigma_c)$, Fig 3 $(\sigma_e)$, Fig 3 $(\sigma_x)$, and Fig 3 $(\sigma_r)$, abstract the standard memory states depicted in Fig 3 $(s_c)$, Fig 3 $(s_e)$, Fig 3 $(s_x)$, and Fig 3 $(s_r)$, respectively. Note that the $\mathcal{LHM}$ memory states represents the abstract values of sealed components, but abstract away their representation. Also note that the inter-component reference structure is maintained by the model pivot fields.

The $\mathcal{LHM}$ memory state shown in Fig 3 $(\sigma_x)$, representing the memory state when the `insert` procedure terminates, is computed directly from the memory state shown in Fig 3 $(\sigma_e)$, representing the memory state when the `insert` is invoked using the specification of `insert`. Furthermore, the model pivot field *amap* exposing the dependency of the value of the key-set's *keys* model field on the abstract value of the map allows computing the updated value of *keys* using its aggregate model function.

### 3.4 Static Analysis

We define a modular static analysis based on an abstract interpretation of $\mathcal{LHM}$ using a *bounded* conservative abstraction. Our analysis is parametric in the bounded abstraction and can use different (bounded) abstractions when analyzing different modules. In our implementation, we use canonical abstraction [29].

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 input state to an intermodule procedure call. This process, in effect, identifies structural *module invariants*. Our analysis verifies conservatively that a module satisfies its specification and respects the restrictions we impose.

Technically, our analysis computes in a conservative manner all possible input states of intermodule procedure calls by exercising the analyzed module using its *most-general-intrusive-client* (*MGIC*): a program that simulates all possible procedure invocations on the analyzed component (thus simulating arbitrary usage contexts) and on its exposed subcomponents (thus simulating all possible side-effects).

## 4. Program Model

We reason about 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 `Java` classes). We expect to be given a partitioning of the program types and procedures into modules.

***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 and formal parameters, which are considered to be local variables. Only local variables are allowed. We assume that the target of a procedure invocation is bound to an implicit `this` parameter. For each field access x.f=e or y = x.f by a procedure $p$ we require that the type $T$ of the object pointed to by x satisfies $m(p) = m(T)$.

# 5. Limitations and Simplifying Assumptions

In this section we list the main limitations on the class of programs for which the work presented in this paper can be applied. The goal of some of the limitations is to simplify our approach. Other limitations are more inherent in our approach. See Sec. 9 for a discussion.

1. We assume that the programming language is object-based, i.e., with dynamic memory allocation and procedures, but without inheritance or subtyping.
2. We expect the module dependency relation to be acyclic. (See Sec. 9 for a possible alleviation of this restriction.)
3. Every component has a single object (the component's *header*) which dominates the entire component, meaning that any heap path reaching the objects comprising the component passes through the component's header, *except* for heap paths reaching the headers of subcomponents. (Note that these heap paths must come from a different module).
4. Every component has a bounded number of exposed (possibly shared) subcomponents.
5. We do not handle deallocation or garbage collection.

In addition, to simplify the presentation, we make the following assumptions:

6. Every module defines a single data structure which can be used by other modules.
7. Every module has a specially-designated *initialization procedure* which allocates new instances of its data structure and initializes them.

# 6. Specification and Proof Obligations

In this section, we describe the required user-provided specification and the proof obligations. Any approach, be it manual or automatic, which can establish the proof obligations with the given specification verifies soundly that the module respects its specification.

## 6.1 Standard Module Specification

In this section, we discuss the more-standard Hoare-style specification that we use.

### 6.1.1 Interface specification

We expect to have a user-supplied Hoare-style *"public" specification* of the program's modules:

- Every module may have *model fields*. The model fields represent the abstract value of the data structure implemented by the module. A model field should range over a value domain. For example, the (only) model field of the `Keyset` module is *keys* and its possible values are sets of integers. The model field of the `Map` module is *map* and its possible values are maps of integers to floats.
- Every procedure should have a pre-post specification which describes the effect of the procedure on the model fields of its parameters. A pre-post specification is *admissible* if it refers to a relation over expressions over the callee's local model heap — i.e. *access paths* starting at the procedure parameters and following a sequence of model pivot fields or to the values of model fields that can be reached by traversing such access paths. For example, the specification of the map's `insert` procedure indicates the abstract value of a map after the invocation of `insert(k,v)`, indicated by a $map'$, is the same as the map's value before the insertion except that the key $k$ is now associated with the value $v$. As a negative (inadmissible) example — if Map's specification were to mention the keys field of a Keyset object referring to it, it would be inadmissible as this Keyset is not reachable from the Map. In the following, we assume that procedure specifications are admissible.

### 6.1.2 Internal specification

We expect an *internal module specification* which provides a *representation function* [11] mapping the data structure's concrete representation to its abstract value. We allow the representation function to be defined only as a function of (i) the values of objects reachable from the component's header via fields defined in the module and (ii) the values of model fields of subcomponents.

## 6.2 Additional Specification Burden

In this section, we describe the additional specification burden required by our approach. More specifically, the additional specification burden required by our approach (compared, e.g., to [11]) is the specification of the *model pivot* fields and the *aggregate model functions*.

### 6.2.1 Interface specification

The model pivot fields specify the data structure's subcomponents that can be shared with other data structures. Any such subcomponent is given an externally-visible name and any procedure which has a side-effect on the abstract value of such a subcomponent is required to expose it in its specification. For example, the `Keyset` module has a (single) model pivot field *amap*. The set's initialization function, `Keyset` specifies that *amap* names the map passed as parameter. The set's `remove` function specifies that the removed key is extracted not only from the set's model field, but also from the map which *amap* refers to. We emphasize that a model field is allowed to have reference (location) value only if it is a model *pivot* field.

We refer to a model field whose value does *not* depend on model fields of externally-visible subcomponents as a *local model field*. An aggregate model function is *admissible* if it is defined as a function of the values of the component's local model fields and the model fields of its subcomponents. For example, the aggregate model function associated with the model field *keys* of the `Keyset` module specifies that the value of this field is the domain of the abstract value of the key-set's map. If we change the key-set to be a *filtered* key-set which contains only keys whose values are bigger than a given threshold, then the threshold value could be exposed by a local model field and the aggregate model function of the filtered key-set would specify that the key-set contains all elements in the domain of its map that are smaller than the (exposed) threshold.

**Remark 6.1.** *In our running example, the model function and the representation function agree. In general, the model function can be less precise than the representation function. This can happen in cases where the representation function can compute a more precise value than the model function thanks to internal knowledge of the data structure implementation.*

### 6.2.2 Internal specification

The internal specification is used only during the verification of the module itself. It links the model pivot fields and model fields with the actual implementation of the module's data structure.

Every local model field is associated with a *representation function* that specifies its value. The representation function can depend only on properties of objects that are reachable from the header of the data structure through fields defined in the module.

The representation function for every model pivot field is further limited to only access fields inside the current component (so it depends only on objects which resides also in the current component) and must evaluate to an outgoing pointer. Note that as a result of the above restriction, changes made to one component, cannot have side effects on the values of pivot model fields of another component.

For example, the model pivot field *amap* is associated with the Keyset's `map`-field which points to a `Map`.

**Remark 6.2.** *We note that the goal of our analysis, described in Sec. 8, is to find* module invariants *(properties which are expected to hold when the data structure is not being manipulated) in an automatic manner. For example, a module invariant of the* `Keyset` *module which our analysis can find is that the field* `m` *never has a* `null` *value.*

### 6.3 Proof Obligations

In this section, we list the required proof obligations. When verifying a module, we need to verify, as usual, that the post-condition of every procedure $p$ and the module invariant are implied from executing $p$'s body in any state which satisfies $p$'s precondition and the module invariant. (When verifying an initialization function only the precondition can be assumed.) In addition, we need to establish the following properties:

**Model function consistency** evaluating the aggregate model function of a model field *mf* of component $c$ using the abstract value (of $c$'s model fields) is conservative with respect to evaluating the representation function of *mf* over any component that can be represented by $c$.

**Model pivot consistency** in any state which satisfies the module invariant, every reachable subcomponent which is not dominated by the data structure's header (i.e., possibly exposed) is named (pointed to) by a model pivot field.

**Single header** every component has a single header.

**Model function dependency is locally acyclic** there is no local dependency cycle between model functions of the same component.

This admissibility of the aggregate model function in addition to the single header requirement and the acyclicity of the module dependency relation ensure that these functions are well defined, i.e., there is no cyclic definition of aggregate model functions.

## 7. Non-Standard Semantics

In this section we define $\mathcal{LHM}$, a non standard concrete semantics which serve as a foundation for a modular analysis. Specifically, the semantics aborts if a module violates its specification. We note that our analysis establishes that the proof obligations are satisfied according to the $\mathcal{LHM}$ semantics. Our restrictions on the program model and on the specification ensure that even though the proof obligations are shown to be satisfied in $\mathcal{LHM}$, the properties that they assert also hold in the standard semantics.

$\mathcal{LHM}$ is a *store-based* semantics (see, *e.g.*, [26]). A traditional aspect of a store-based semantics is that a memory state represents a heap comprised of all the allocated objects. $\mathcal{LHM}$, on the other hand, is a *local heap* semantics [27]: 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.

| | | | |
|---|---|---|---|
| $l$ | $\in$ | $Loc$ | Location |
| $var$ | $\in$ | $VarId$ | Variable id |
| $\varepsilon$ | $\in$ | $Env = \mathcal{V} \hookrightarrow Val$ | Local variables |
| $v$ | $\in$ | $Val = Loc \cup \{null\} \cup \mathbb{N} \cup \mathbb{F}$ | Concrete value |
| $h$ | $\in$ | $\mathcal{H} = Loc \hookrightarrow \mathcal{F} \hookrightarrow Val$ | Concrete heap |
| $t$ | $\in$ | $Type$ | Type |
| $t$ | $\in$ | $\mathcal{TM} = Loc \hookrightarrow \mathcal{T}$ | Type heap |
| $a$ | $\in$ | $AVal$ | Model value |
| $ah$ | $\in$ | $\mathcal{AH} = Loc \hookrightarrow \mathcal{F} \hookrightarrow AVal$ | Model heap |
| $ph$ | $\in$ | $\mathcal{PH} = Loc \hookrightarrow \mathcal{F} \hookrightarrow Val$ | Pivot heap |
| $\sigma$ | $\in$ | $\Sigma = Env \times 2^{Loc} \times \mathcal{H} \times \mathcal{TM} \times \mathcal{M} \times \mathcal{AH} \times \mathcal{PH} \times \mathcal{AH}$ | Memory states |

**Figure 4.** Semantic domains.

$\mathcal{LHM}$ is a hybrid semantics: It represents both objects and components as well as concrete fields and model fields, however it uses different representations for the current component and for the sealed components (thus the name hybrid): $\mathcal{LHM}$ represents the allocated objects inside the current component and their *concrete* fields. However, it does not represent the model fields of the current component. In contrast, $\mathcal{LHM}$ does not represent either the objects inside sealed components or record the values of their concrete fields. Instead, it represents every sealed components using its header and only records its model fields and model pivot fields. Technically, $\mathcal{LHM}$ associates the model and model pivot fields of a component with its header.

$\mathcal{LHM}$ is a "mixed-step" semantics: When executing intraprocedural statements and *intra-module* procedure calls, it acts as small-step operational semantics [25]. However, instead of encoding a stack of activation records inside the memory state, as traditionally done, $\mathcal{LHM}$ maintains a *stack of program states* [15]. The use of a stack of program states allows us to represent in every memory state the (values of) local variables and the partial heap of just one (the *current*) procedure. When executing an *inter-module* procedure call, $\mathcal{LHM}$ acts as large-step operational semantics [13]: it computes the effect of a procedure invocation in "one step" using the procedure's specification.

$\mathcal{LHM}$ checks that the program memory states satisfy certain *admissibility conditions* (listed below). Thus, $\mathcal{LHM}$ may abort whereas standard heap semantics would not abort. Such an abort means that a module does not satisfy our restrictions. (Our analysis conservatively detects such aborts.) For brevity, we only informally discuss the relation between $\mathcal{LHM}$ and the standard heap semantics and describe key aspects of the operational semantics. In [12], we formally define $\mathcal{LHM}$ and relate it to the standard semantics using Galois connections.

### 7.1 Memory States

Fig 4 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, the special *null* value, an integer or a float.

A memory state in the $\mathcal{LHM}$ semantics is an 8-tuple:

$\sigma = \langle \varepsilon, L, h, t, m, alh, ph, ah \rangle$. The first four elements in the tuple comprise a 2-level store:

(i) $\varepsilon \in Env$ is an environment assigning values for the variables of the *current* procedure.

(ii) $L \subset Loc$ contains the locations of allocated objects. (An object is identified by its location. We interchangeably use the terms object and location.)

(iii) The *heap* $h \in \mathcal{H}$ assigns values to fields of allocated objects.

(iv) $t \in \mathcal{TM}$ 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* $\sigma$ is $m(t(l))$.

The last four elements of the tuple are specific to $\mathcal{LHM}$, and are used to record the module of the current procedure and the fields of local model fields, model fields, and model pivot fields:[1]

(v) The fifth component, $m \in \mathcal{M}$, is the module of the current procedure. We refer to $m$ as the *current module* of $\sigma$.

(vi) The sixth component, $alh \in \mathcal{AH}$, is the *local abstract value map*. It records the abstract values of *local model fields*.

(vii) The seventh component, $ph \in \mathcal{PH}$, is the *model pivot heap*, recording the reference values of pivot model fields.

(viii) The eighth component, $ah \in \mathcal{AH}$, is the *abstract value map*. It records the abstract values of non-local model fields.

To exclude states that cannot arise in any program, we now define the notion of admissible states. We note that $\mathcal{LHM}$ preserves the admissibility of memory states.

An $\mathcal{LHM}$ memory state $\sigma = \langle \varepsilon, L, h, t, m, alh, ph, ah \rangle \in \Sigma$ is **admissible** if

(i) The domain of the heap and the local abstract value map, the model pivot heap, and the abstract value map are disjoint, i.e., let $A = dom(alh) \cup dom(ph) \cup dom(ah)$, then $dom(h) \cap A = \emptyset$;

(ii) Every object in the domain of the heap belongs to the current module, i.e., for every $l \in dom(h)$, $m(t(l)) = m$;

(iii) The type of every object in the (local) abstract values map and the model pivot heap does not come from the current module, i.e., for every $l \in A$, $m(t(l)) \neq m$, where $A$ is as defined above;

(iv) Fields and model fields can point only to allocated locations, i.e., $\{h(l)f \in Loc, ph(l)f \mid l \in L, f \in \mathcal{F}\} \subseteq L$; and

(v) The pivot heap is acyclic.

**Example 7.1.** Figures 3($\sigma_c$) and 3($\sigma_r$) depict the (admissible) $\mathcal{LHM}$ memory states that arise in the execution of the running example before and after the third invocation of `insert`, respectively.

### 7.2 Operational Semantics

We only discuss the key aspects of the operational semantics, formally defined in [12]. For simplicity, we assume that the procedure of the module are partitioned into *public procedures*, which can be invoked only by procedures of other modules and *private procedures*, which can be invoked only be procedures of the module. (Note that, in particular, a public procedure cannot be invoked by another procedure of the module.)

#### 7.2.1 Intraprocedural statements.

Intraprocedural statements are handled, essentially, as usual in a two-level store semantics for pointer programs (see, *e.g.*, [26]). The main difference from the standard semantics is that the semantics checks that the program accesses only fields of objects that belong to the current component.

We note that we simplify the memory management mechanism in our semantics by exploiting the fifth simplifying assumption (i.e., the assumption that memory locations are not reused): Whenever a new location is allocated, the semantics chooses an arbitrary location which was not allocated before. Technically, it utilizes the set of allocated location (i.e., the second component of a memory state, see Fig 4) to accumulate all allocated locations.

#### 7.2.2 Intra-module interprocedural statements.

$\mathcal{LHM}$ is a local-heap semantics [27] which maintains a *stack of program states* to handle intra-module procedure calls [28]. 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.

Note that only a intra-module procedure call can invoke only a private procedure.

#### 7.2.3 Inter-module interprocedural statements.

Fig 5 defines the axioms for intermodule procedural calls. When an intermodule procedure call is invoked, $\mathcal{LHM}$ computes the return state $\sigma_r$ in three steps, as described below.

***Computing the callee's entry state.*** First, the rule computes an intermediate memory state, $\sigma_e$, which represents the callee's (input) local heap but with all components remaining sealed. It does this by restricting the heap to $L_{rel}$, the part reachable from the actual parameter. Because the module dependency is acyclic, only headers of sealed components (reachable via the pivot heap) can be passed as the values of actual parameters. Thus, $L_{rel}$ is computed as the set of locations that are reachable from the parameters in the pivot heap. Note that every location in $L_{rel}$ is the header of a sealed component. Also note that the restriction of the caller's heap ($h_c$) to $L_{rel}$ necessarily returns an empty heap.

***Computing the callee's exit state.*** $\mathcal{LHM}$ applies the effect of the invoked procedure in one step: it selects (non-deterministically) any exit state ($\sigma_x$) which is a possible outcome of the invoked procedure *according to its specification* on the computed entry state ($\sigma_e$).

**Remark 7.2.** *The notation $\sigma_e \overset{[\![p]\!]}{\rightsquigarrow} \sigma_x$ is used to express that according to the specification of procedure $p$, its invocation on an entry state $\sigma_e$ may result in memory state $\sigma_x$. In particular, it denotes that $\sigma_e$ satisfies $p$'s preconditions (and that the semantics aborts otherwise.)*

*Note that this notation also makes it clear that the specification cannot refer to the current component, as it should hold in any (arbitrary) calling context in which $p$ is invoked on $\sigma_e$.*

***Computing the caller's return state.*** $\mathcal{LHM}$ computes the return state ($\sigma_r$) of the invocation $y = p(x_1, \ldots, x_k)$ by, essentially, carving out the local heap passed to the callee from the call heap and replacing it with the callee's heap at the exit site. The replacement is mostly straightforward:

(i) Updating the caller's environment amounts to assigning the return value of the procedure ($\varepsilon_x(ret)$) to $y$.

(ii) The module of the caller's function at the return state is the same as in the call state ($m_c$), and thus does not change.

(iii) The caller's set of allocated locations is set to that of the callee ($L_x$) to account for the locations which were allocated during the procedure invocation. (Recall that $L_c \subseteq L_x$.)

(iv) The caller's heap at the return state is the same heap as at the call state ($h_c$) because the callee could not have modified it. (Recall the $h_c$ contains only the objects inside the caller's component, and that the callee could not have reached them.)

(v-vii) The caller's type map at the return memory state is defined to be the same as the call state, except for location which are in the callee's local heap, whose type is taken from the type map of the callee at the exit state ($t_x$). We note that the callee could not have allocated a location which appears in the caller's heap because these locations are accumulated in the set of allocated locations passed to the callee in its input state. The caller's local abstract value map ($alh_r$) and model

[1] Recall that model fields are used to record the abstract values of sealed components and that pivot model fields are used to record the inter-component reference structure between sealed components. Specifically, the *model pivot fields* of a component specify which of its subcomponents is externally-visible and thus can be shared with other components. We remind the reader that we refer to a model field whose value does *not* depend on model fields of externally-visible subcomponents as a *local model field*. See Sec. 6.2.1.

$$\langle Call_{y=p(x_1,\ldots,x_k)}, \sigma_c\rangle \overset{LHM}{\longrightarrow} \sigma_r$$

where:

$$\begin{array}{l}
\sigma_e = \langle \varepsilon_e, L_c, \emptyset, t_c|_{L_{rel}}, m_c, alh_c|_{L_{rel}}, ph_c|_{L_{rel}}, alh_c|_{L_{rel}}\rangle \\
\quad \varepsilon_e = [z_i \mapsto \varepsilon_c(x_i) \mid 1 \leq i \leq k] \\
\quad L_{rel} = R(\{\varepsilon_c(x_i) \in Loc \mid 1 \leq i \leq k\}, ph_c) \\
\sigma_e \overset{[\![p]\!]}{\rightsquigarrow} \sigma_x \\
\sigma_r = \langle \varepsilon_c[y \mapsto \varepsilon_x(ret)], m_c, L_x, h_c, t_r, alh_r, ph_r, ah_r\rangle \\
\quad t_r = t_c[t_x] \\
\quad alh_r = alh_c[alh_x] \\
\quad ph_r = ph_c[ph_x] \\
\quad ah_r = ah_x \cup \bigcup_{l \in dom(ah_c) \setminus dom(ah_e)} update(l, ah_c, alh_r, ph_r, t_r, ah_x)
\end{array}$$

$$update(l, ah_c, alh_r, ph_r, t_r, ah_x) =$$

$$\begin{cases}
ah_x & l \in dom(ah_x) \\
ah_{sub} \cup \{l \mapsto f \mapsto m_f(l, alh_r, ah_{sub}, ph_r) \mid f \in model(t_r(l))\} & \text{otherwise} \\
\quad ah_{sub} = \bigcup_{l' \in \text{subcomponents}} update(l', ah_c|_{L'_{rel}}, alh_r|_{L'_{rel}} ph_r|_{L'_{rel}}, t_r|_{L'_{rel}}, ah_x|_{L'_{rel}}) & \\
\quad\quad \text{subcomponents} = \{l' \in \{ph_r(l,p) | p \in pivot(t_r(l))\} & \\
\quad\quad L'_{rel} = R(\{l'\}, ph_r) &
\end{cases}$$

$$R(L, ph) = \{l \in dom(ph) \mid \text{ there is a heap path in pivot heap } ph \text{ which starts at a location in } L \text{ and reaches } l\}$$

**Figure 5.** The axioms for an arbitrary intermodule procedure call $y = p(x_1, \ldots, x_k)$ assuming $p$'s formal parameters are $z_1, \ldots, z_k$ and $p$ returns its value by assigning it to a specially designated variable $ret$. The call state is $\sigma_c = \langle \varepsilon_c, L_c, h_c, t_c, m_c, alh_c, ph_c, ah_c\rangle$ The exit state is $\sigma_x = \langle \varepsilon_x, L_x, h_x, t_x, m_x, alh_x, ph_x, ah_x\rangle$ is a possible outcome of $p$ when invoked on memory state $\sigma_c$ according to $p$'s specification, $[\![p]\!]$. For a type $t \in \mathcal{T}$, $model(t)$ denotes the model fields of $t$, $pivot(t)$ denotes the model pivot fields of $t$. The aggregate model function associated with a model field $f$ is denoted by $m_f$.

pivot map ($ph_r$) are constructed in a similar way. We note the straightforward update of these maps is possible thanks to (a) the restrictions that we impose on these maps (See Sec. 6.2.2 and Sec. 6.2.2) and (b) the fact that the callee could not have reached the components which were not passed to it in its input state.

The most challenging aspect of our semantics is the computation of the caller's abstract value map ($ah_r$): The computation has to account for *operations by the callee that might have change the abstract value of components which the callee could not have reached.*

To correctly propagate the side effect of the invoked procedure on the (aggregate) model fields the semantics updates the value of every model field pertaining to an object which was not passed to the callee using the $update$ procedure. The invocation of $update(l, ah_c, alh_r, ph_r, t_r, ah_x)$ updates the model fields of $l$ by first (recursively) computing $ah_{sub}$, an abstract value map containing the updated values of model fields of every subcomponent of $l$. (Note that to determine the subcomponents, the semantics uses the updated pivot heap.) $\mathcal{LHM}$ then computes the value of every model field $f$ of $l$ using $m_f$, the aggregate model function of $f$. Note that this computation is well defined because of the acyclicity of model field dependencies promised by our assumptions.

**Remark 7.3.** *We note that the above definition recomputes values of fields that cannot be modified. In [12], we present a more complicated version of the axioms for intermodule procedural calls. The latter updates only fields that depend on potentially modified fields, and behaves better under abstraction. The version in [12] also handle some delicate issues that may occur when the model function is less precise than the representation function.*

The procedure call rule for *public* procedures also checks that usual Hoare requirement: i.e., it executes a procedure only if the entry state satisfies its precondition and checks that its exit state satisfies the post condition.[2] The semantics computes the abstract value of the current component using the representation functions defined in the module internal representation. For example, Fig 2(b) shows the internal specification of the KeySet module. For brevity, we omit this standard part from Fig 5. For details, see [12, App. C.].

### 7.3 Observational Soundness

Our goal is to verify (modularly) properties of modules according to the *standard* semantics. The admissibility of $\mathcal{LHM}$ memory states, our programming model, and the conditions checked by the operational semantics, ensure that if the $\mathcal{LHM}$ semantics never aborts when executing a module, then for any memory state $s$ that can arise according to the *standard* semantics (in a program comprised only of such "well-behaved" modules) there exists a $\mathcal{LHM}$ memory state $\sigma$ which arises during the execution of the program in the $\mathcal{LHM}$ semantics which abstracts $s$.

An immediate consequence of the above is that any assertion regarding properties of objects reachable from component headers which is shown to hold with respect to $\mathcal{LHM}$ semantics also holds with respect to the standard semantics. For a formal definition of the observational soundness theorem, see [12].

## 8. Static Analysis

This section presents key aspects of our (conservative) modular static analysis. The analysis is obtained as an abstract interpretation of $\mathcal{LHM}$ using a *bounded* conservative abstraction. Our analysis is parametric in the bounded abstraction and can use different (bounded) abstractions when analyzing different modules. In our implementation, we use canonical abstraction [29].

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 input state to an intermodule procedure call. This process, in effect, identifies

[2] In addition, the rule checks that every component of the module has a single header. This check is similar to the one done in [28].

structural *module invariants*. Our analysis verifies conservatively that a module satisfies its specification and respects the restrictions we impose.

The main challenge in our analysis lies in finding all the possible input states to an intermodule procedure calls. We overcome this challenge by utilizing the fact that in $\mathcal{LHM}$ whenever a program passes a component of the analyzed module a parameter to an intermodule procedure call, it must be a sealed component which was previously generated by the program. (This is derived from the fact that every component has a single header and that a component can be manipulated only by the module which generated it). In particular, we can *anticipate the possible entry memory states of an intermodule procedure call*: 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 anticipate (conservatively) 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 conservatively compute the effect of procedure calls on subcomponents using the user-provided specification. The procedure's pre-post specification allows us to find the effect of the procedure on the components passed to it as parameters (and their subcomponents) and the aggregate model function allows us to propagate side effects to sibling components.

### 8.1 Modularity

Our analysis is modular in two aspects. First, it is modular in the program code: When verifying one module, we only require the code implementing that module and only the specification of the other modules. (Specifically, we determine the effect of an intermodule procedure call on a subcomponent using the procedure's specification.) Second, it is modular in the program state: when reasoning about a module, we

(i) reason about the *concrete* representation of data structures manipulated by the module,

(ii) represent the *abstract* values and the topologies (*sharing patterns*) of subcomponents that come from other modules, and

(iii) ignore the data structure context containing the analyzed data structures.

(In comparison, Hoare's approach, by being targeted to verify only fully encapsulated data structures, can avoid reasoning about sharing patterns.) A key reason for the modularity of our analysis in the program state is the heterogeneous memory representation of the *hybrid* states. This allows our analysis to require only the specification of dependant modules and not their implementation (as is required, e.g., in [31]).

### 8.2 Most-General-Intrusive Client

We conduct modular static analysis by performing an interprocedural analysis of a module together with its *most-general-intrusive client* (MGIC). The module's *MGIC* is defined outside the module and invokes a sequence of arbitrary (intermodule) procedures calls to the module using arbitrary input arguments. (In this respect, the most-general-intrusive-client is similar to the most-general-client of a class [28].) However, it also invokes procedures directly on the exposed subcomponents of the analyzed module. (In this respect, the client is intrusive as it bypasses the component and directly interacts with its subcomponents.) This allows to simulate conservatively any arbitrary context in which components of the module can be used. In our analysis, the MGIC takes the role of the $\mathcal{LHM}$ semantics in verifying that the preconditions and post-conditions (specified using abstract values) are respected. This allows our analyzer to verify that every intermodule procedure call made by the MGIC respects the procedure's specification.

**Example 8.1.** The MGIC for the Keyset of the running example is shown in Fig 6. The MGIC first allocated a map, and invokes an arbitrary sequence of operations on the map, thus generating any possible map that can be passed to the KeySet when the latter is initialized. The MGIC executes a loop in which it invokes the KeySet methods (cases 2 and 3) acting as a client of the KeySet. It also invokes, "intrusively", methods on the underlying map (cases 0 and 1), simulating indirect changes to the underlying map.

We note that the analysis considers the actual implementation of the KeySet, but only the abstract view (i.e., the model fields) of the underlying map.

Pre and post conditions are checked before/after each call (in the model space). The last assertion checks that the model function over-approximates the representation function.

### 8.3 Bounded abstraction

We provide an effective conservative abstract interpretation [6] algorithm which determines *module invariants* by devising a bounded abstraction of $\mathcal{LHM}$ memory states. An abstraction of a $\mathcal{LHM}$ memory state, being comprised essentially 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 tracks are the model pivot fields (which can be abstracted in a similar manner to standard concrete fields) and model fields representing abstract values of data structures. Thus, the abstract domain is expected to be able to (conservatively) represent the abstract values used in the specification.

Abstracting a $\mathcal{LHM}$ memory state is simpler than abstracting standard memory states: Instead of abstracting the representation of a data structure (e.g., the representation of a map as a tree) the abstraction needs to record the essential properties of the data structure, e.g., the association between keys and values, the elements in the domain of the map, etc. Furthermore, we believe that a key reason for the success of our analyzer is the fact that while a procedure call might have a complicated effect on the concrete heap, e.g., inserting a node to a tree, its effect on the abstract value of the data structure can be much more limited, e.g., adding an association to a map.

### 8.4 Experimental evaluation

We have experimented with expressing several examples in our system. We realized our system by developing a proof-of-concept modular analyzer using canonical abstraction [29] within the TVLA system [19]. Our analyzer was able to verify the `Keyset` module in 10 seconds running on a machine with a 2.66 Ghz Core 2 Duo processor and 2 Gb memory. The verification proved that any Keyset component in any context using any map that conforms to the Map specification would behave as in its specification.

## 9. Future Work

In our work we chose to focus on one of the main challenges in object-invariant-based modular verification of object-oriented programs: invariants that span multiple data structures where sharing is allowed [9]. Our approach can handle programs that employ certain popular programming idioms, e.g., the *wrapper* [10], *decorator* [10], and *passive-model-view-controller* [4] design patterns. However, other popular design patterns, such as iterators and the

```
MGIC(){
   Map m;
   Keyset s;
   //Initialize an arbitrary map
   m = new Map();
   while (?){
    switch (?) {
      case 0: m.insert(?,?); break;
      case 1: m.remove(?); break;
    } //switch
   } //while

   //Exercise all use-cases of Keyset
   s = new Keyset(m);
   while (true){
     int key = ?;
     float val = ?;
    switch (?) {
      case 0:
       assume(true);
       m.insert(key,val);
       assert(m.map' = m.map ∪ (key, val));
      case 1:
       assume(true);
       m.remove(key);
       assert(m.map' = m.map \ {key});
      case 2:
       assume(true);
       bool b = s.isMember(key);
       assert(b == key ∈ s.keys);
       assert(m.map' = m.map);
       assert(s.keys' = s.keys);
       break;
      case 3:
       assume(true);
       s.remove(key);
       assert(m.map' = m.map \ {key});
       assert(s.keys' = s.keys \ {key});
       break;
    } //switch
     assert(s.keys = keys_m(s));
   } //while
  }
```

**Figure 6.** MGIC for the Keyset component. The question marks represent non-deterministic selections.

*active-model-view-controller* [4] cannot be handled.[3] In this section, we discuss certain ways to extend the class of programs for which our approach is applicable by alleviating some of the limitations described in Sec. 5. For additional discussion, see [12].

***Cyclic module dependency relation..*** Our requirement that the module dependency relation be acyclic is used to simplify the presentation. The fundamental requirement is model field dependency acyclicity — that can be enforced modularly. A simple and practical way to achieve this is to require that model field dependency is acyclic by type (meaning the dependency relation for model field types is acyclic) however this disallows some existing programming patterns such as alternating lists. A more general way is to require local acyclicity (among model fields of the same type) and require acyclicity whenever an object is abstracted (e.g. $this$ on return from function) as a proof obligation. This requires more verification work but allows more programs.

A particular challenging issue with cyclic import is that $this$ can be passed (e.g., in callbacks) as an argument. In this situation, any knowledge of the current component (including local variables pointing into it) is lost upon return unless it is passed as immutable (even if the specification does not specify any modification to its model fields).

***Multiple headers.*** Our restriction that every component should have a single header simplifies our semantics and analysis as it prevents us from tracking the relation between different entry points into the same component. However, is quite severe as it disallows, e.g., iterators. We believe it can be removed if all headers of the component (e.g. List and its Iterators) are defined in the same module and verified with knowledge of each other. Specifically, each should have a pivot to other one. (This means pivots and representation functions in general will no longer only reference reachable heap as, e.g., an iterator does not have to have a pointer to a List's header but only to a Node). We hope it is possible to allow multiple headers per component with the above mentioned restrictions and some additional modifications to the formulation. In particular, we believe that this is doable if we restrict the number of headers.

***Subtyping*** Subtyping is challenging because an upcast loses specification information. The main problem that our approach faces because of subtyping is that information pertaining model fields which appear only in the subtype is lost on upcast (e.g. when calling a function that receives a supertype as a parameter) and hence has to be reconstructed on return.

We believe that with some restrictions on subtyping and modifications to the call rule, our system can support subtyping. In particular, we suggest to consider a stricter subtyping model where downcasts are forbidden, subtype-added methods are the only ones allowed to modify subtype-added model fields. By a stricter subtyping model we mean that subtypes may have additional constructs (concrete functions and specification model fields) and stricter specification for constructs: stricter pre-post for functions, stricter specification for model fields, i.e., *less* dependencies, and stricter model functions, i.e., more deterministic ones.

## 10. Related Work

In this section, we review closely related works.

***Modular verification.*** In [2], Barnett and Naumann presented a verification approach that supports state dependencies across ownership boundaries. A component can grant "friends" the right to depend on their state. Thus, the dependency is publicly visible and treated in a modular way. The downside of this technique is that the granting component needs to know its clients. To overcome this restriction, Leino and Schulte developed a technique based on history invariants [18] which allow component invariants to depend on other non-encapsulated components, which resembles our method. However, the expressible dependencies are weaker than the ones allowed by our method because they need to be captured by a history constraint.

A modular solution for model fields and non-shared abstractions is developed by Müller in [21] and by Müller and Leino in [17]. The focus of their work is the verification of object oriented programs using object invariants. They face a similar challenge to the one faced in this work: maintaining consistency between the abstract values of component and their concrete state. Indeed, These works inspired some of our definitions for when an abstraction has to hold and the ideas of pivots and dependencies. However, the methods shown in these works are more suitable for manual verification or theorem provers rather than analysis.

[3] The passive model can be employed when one controller manipulates the model exclusively. The controller modifies the model and then informs the view that the model has changed and should be refreshed. Specifically, there is no means for the model to report changes in its state. In the active model, the model notifies the views to refresh the display [1].

In [14], Kassios presents a framework for modular reasoning supporting abstraction, data-hiding, and subtyping. The framework allows asserting properties such as disjointness and inclusion between the domains of different representation functions and to prove *non-modification* of model fields in a modular fashion. Classes can have "public" invariants which encode similar information to our "public" model function, however, in a less restrictive way. We give explicit rules for pivots and rely on pivot equality while Kassios uses frames which are sets of pointers and relies on set operations (inclusion, disjointness etc) which we believe are harder to use in practice. Our restrictions give us simpler proof obligations with simpler propagation of side-effects which can be verified automatically.

Cameron et al. [5] describe an ownership system that support multiple owners. Their work supports programs with concurrency, subtyping, inheritance and cyclic module import work supports. Our work is similar to theirs in the sense that in both works the ownership graph is required to be a dag. However, they focus on verifying non-modification, whereas we focus on verifying more general partial correctness assertions.

***Local reasoning*** O'Hearn et al. [24] and Bierman and Parkisnon [3] allow to conduct modularly (manual) local reasoning [26] 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 user provided interface specification and representation functions (and an instance of the bounded shape abstraction). Bierman and Parkisnon [3], however, allows for more sharing than in our model. Their system can encode "public" invariants of aggregates and model pivots, but does not have an inherent notion of aggregate representation function. Lately, a verification tool [8] based on abstract predicates [3] was developed by Distefano and Parkinson.

***Modular static analysis*** Cousot and Cousot [7] describe 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 *user provided specification* to communicate the effect and *side effects* of mutations done by different modules.

Lam et al. [16] and Wies et al. [30] also utilize user-specified pre- and post- conditions to achieve modular shape analysis which can handle a bounded number of flat set-like data structures. Our approach, allows for separately-analyzed arbitrarily-nested and possibly shared sets.

Logozzo [20] presents a modular analysis which infers class invariants. The determined invariants do *not* concern properties of *shared* sub-objects.

Yorsh et al. [31] provide a method for computing the effect of a procedure call which is modular in the program code — but not in the program state: A theorem prover is used to propagate the effect of a procedure call on the abstract field of the caller by inferring it from the call's effect on the concrete fields of the callee. Intuitively, their approach requires maintaining the values of concrete fields for subcomponents.

Rinetzky et al. [28] present a *modular* shape analysis which identifies structural (shape) invariants for dynamically encapsulated programs: heap-manipulating programs which forbids sharing between components via live (i.e., used before set) references. This paper allows shared components but requires that every shared subcomponent be named by a model pivot field. This may restrict our approach from handling data structures which hold object data which are transferred as parameters. One way we can overcome this restriction is by incorporating in our approach dynamic encapsulation for certain kinds of object parameters.

# 11. Conclusions

We present a novel approach for modular verification of programs with shared data structures. The essence of our work is the propagation of the side-effects on the abstract values of sibling data structures. The aggregate model function provides a conservative approximation of the data structure's abstract value based on the abstract values of its internal, unshared parts and the abstract values of its shared subcomponents.

The additional proof burden that we place (beyond the one imposed, e.g., by Hoare's approach for verifying abstract data types [11]) is proportional to (i) the *allowed sharing*, i.e., to the number of subcomponents of the verified data structure which are allowed to be shared externally and the interconnection between them, and (ii) the *actual sharing*, i.e., the sharing between the data structure subcomponents.

# References


[1] MSDN library; model-view-controller. available at *http://msdn.microsoft .com/en-us/library/ms978748.aspx*.

[2] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Lecture Notes in Compute Science*, number 3125, 2004.

[3] G. Bierman and M. Parkinson. Separation logic and abstractions. In *Principles of Programming Languages (POPL)*, 2005.

[4] S Burbeck. Application programming in smalltalk-80: How to use model-view-controller (mvc). available at *http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html*, 1992.

[5] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2007.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Principles of Programming Languages (POPL)*, 1977.

[7] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *Compiler Construction (CC)*, 2002.

[8] D. Distefano and M. Parkinson. jstar: Towards practical verification for java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2008. to appear.

[9] S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. A Unified Framework for Verification Techniques for Object Invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, July 2008.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[11] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[12] U. Juhasz. Modular verification with shared abstractions. Master's thesis, Tel-Aviv University, School of Computer Science, Tel-Aviv, Israel, July 2008. available at *www.cs.tau.ac.il/∼urijuhas/thesis.pdf*.

[13] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39. Springer-Verlag, 1987.

[14] I. T. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, University of Toronto, 2006.

[15] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Compiler Construction (CC)*, 1992.

[16] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *Compiler Construction (CC)*, 2005. (tool demo).

[17] K. R. M. Leino and Peter Müller. A verification methodology for

model fields. In *European Symposium on Programming (ESOP)*, 2006.

[18] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *European Symposium on Programming (ESOP)*, 2007.

[19] T. Lev-Ami and M. Sagiv. Tvla: A framework for kleene based static analysis. In *Static Analysis Symposium (SAS)*. Springer, 2000.

[20] F. Logozzo. Automatic inference of class invariants. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2004.

[21] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[22] Peter Muller. Modular specification and verification of object oriented programs. In *Lecture Notes in Compute Science*, number 2262, 2002.

[23] J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.

[24] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *Principles of Programming Languages (POPL)*, 2004.

[25] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[26] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, 2002.

[27] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Principles of Programming Languages (POPL)*, 2005.

[28] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *European Symposium on Programming (ESOP)*, 2007.

[29] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Transactions on Programming Languages and Systems (TOPLAS)*, 2002.

[30] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2006.

[31] G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manupilating programs. In *International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL)*, 2005.