Efficient Runtime Assertion Checking of Assignable Clauses with Datagroups

Hermann Lehner and Peter Müller

ETH Zurich, Switzerland
{hermann.lehner,peter.mueller}@inf.ethz.ch

Abstract. Runtime assertion checking is useful for debugging programs and specifications. Existing tools check invariants as well as method pre- and postconditions, but mostly ignore `assignable` (or `modifies`) clauses, which specify the heap locations a method is allowed to assign to. A way to abstract from implementation details is to specify `assignable` clauses using datagroups, which represent sets of concrete memory locations. Efficient runtime checking of `assignable` clauses with datagroups is difficult because the members of a datagroup may change over time and because datagroups may get very large, especially for recursive data structures. We present the first algorithm to check `assignable` clauses in the presence of datagroups. The key idea is to compute the set of locations in a datagroup lazily, which requires data structures that reflect when the contents of a datagroup change during the execution of a method. We implemented our approach in a prototypical runtime assertion checker for the Java Modeling Language (JML); our experiments show that the runtime overhead is moderately small.

1 Introduction

To verify interesting program properties, it is important to know the side effects of a method. To this end, frame properties define which heap locations a method may modify, and, more importantly, that everything else in the heap stays unchanged. In JML, a method specification expresses such frame properties by the use of the `assignable` clause. This clause declares the heap locations that may be updated during method execution.

To achieve information hiding, we can mention *datagroups* in `assignable` clauses to abstract away from concrete locations [7,8]. For any field of an object, we can specify which datagroup(s) it belongs to. A datagroup is *static* if it only contains fields of the same object. Otherwise, the datagroup is *dynamic*.

Dynamic datagroups are crucial to specify frame properties for aggregate or recursive data structures. In our example in Fig. 1, we introduce a class `Store` that manages items. Dynamic datagroups allow us to specify that method `add` changes at most the internal data structure of the store.

To check a program against its specification, we can either use a static verification tool or we equip the code with runtime assertion checks that fail if an

```
class Item {
  JMLDataGroup footprint;                    // 0

  boolean selected;                          // 1

  String name; //@ in footprint;             // 2

  int price; //@ in footprint;               // 3

  /*@ assignable this.footprint, other.selected; */
  void copy(Item other){
    this.name = other.name;
    this.price = other.price;
    other.selected = false;
  }
}

class Node {
  JMLDataGroup struct;                       // 0

  Node left; //@ in struct;                  // 1
  /*@ maps left.struct into struct; */

  Node right; //@ in struct;                 // 2
  /*@ maps right.struct into struct; */

  Item data;                                 // 3

  /*@ assignable this.struct; */
  void replace(Node old, Node new){ [...] }
}

class Store {
  JMLDataGroup struct;                       // 0

  Node root; //@ in struct;                  // 1
  /*@ maps root.struct into struct; */

  /*@ assignable this.struct; */
  void add(Item i){ [...] }
}
```

Fig. 1. A store that contains items, using a tree as internal data structure. `Store` and `Node` objects contain a field `struct`, whose datagroup contains the fields `left` and `right` of the same object, and the `struct` datagroup of the children. The `struct` datagroup allows us to refer to all locations of the data structure without exposing implementation details. The number behind each field declaration will be used later when we explain our algorithm.

illegal operation is about to happen. Both approaches have already been taken to check `assignable` clauses, however datagroups pose a problem on both sides.

Many static verification tools [1,3,5,9,10,11] support `assignable` clause to some extent; some partly support static datagroups, but no static verification tool currently supports dynamic datagroups. To precisely reason about dynamic datagroups, a verification environment produces proof obligations that have to be discharged manually, as checking the containment in a dynamic datagroup is essentially a reachability problem, which is not handled well by SMT solvers. Existing static analyses can only provide an over-approximation that is too imprecise to be useful.

The situation for runtime assertion checkers (RAC) is similar: The RAC for JML presented in Cheon's dissertation [4] does not provide checks for `assignable` clauses. Ye [12] adds limited support for static datagroups only. JML's semantics is to determine upon method invocation the set of locations in the datagroups. The number of locations in a dynamic datagroup is unknown at compile time and can grow as fast as the heap itself. Therefore, a naïve implementation of the semantics would lead to a large memory and time overhead.

We present an algorithm to efficiently check `assignable` clauses at runtime. The motivation for such checks is twofold: First, we can use a RAC to check a program's validity with little effort and small annotation overhead before starting to prove its correctness in an interactive theorem prover. In this way, we find bugs early and reduce the risk of getting stuck in an expensive manual proof. Second, if we use an automatic verification tool, we often get spurious error messages because of under-specification or deficiencies of the prover. In this case, we can use RAC to see if the program really violates the specification for the given input values. In order to achieve our goal we attack the problem from three sides.

(1) We provide efficient implementations of two operations that are heavily used in our algorithm: checking if an `assignable` clause mentions a certain location or datagroup, and collecting all static datagroups that contain a location. We introduce new data structures for assignable maps and for static datagroups based on bitset operations to achieve this goal.

(2) We reduce memory consumption by introducing the concept of *lazy unfolding* of dynamic datagroups to avoid unnecessary overhead. Instead of unfolding the datagroups of an `assignable` clause in the pre-state of the method, we track the changes to dynamic datagroups during method execution and only store the difference between the pre-state and the current state. We can decide at compile time, which operations trigger a change to the dynamic datagroups and instrument the code at that point to store the changes.

(3) We optimize time complexity by caching the result of checking whether a location is assignable, as this information can be reused within the same method.

2 Prerequisites

In this section, we introduce the notations and semantics of locations, method call stacks, `assignable` clauses, and datagroups.

Locations. At runtime, a field of an object is called a *location*. For convenience, we define a function $obj(\cdot)$ that yields the object of a location. For example, $obj(o.f)$ yields o.

Method Call Stack. We introduce the binary relation $m_1 \hookrightarrow m_2$ which states that method m_2 is called by m_1 at runtime. We also introduce the reflexive transitive closure: $m_1 \hookrightarrow^* m_2$, meaning, m_1 is m_2 or m_1 is a direct or transitive caller of m_2.

Assignable Clauses. We can specify the frame of a method using the clause `assignable` $l_1, \ldots, l_n$;, where l_i has the form $o.f$ to refer to a field of an object. JML provides several other forms to specify assignable locations, but these are not relevant for this paper.

The semantics of an `assignable` clauses is defined as follows. The fields mentioned in the clause are evaluated to a set of locations. This evaluation is performed in the pre-state of the method, that is, upon method invocation. The `assignable` clause only restricts assignment to locations that already existed in the pre-state of the method.

Let $\mathcal{A}_m$ be the set of locations from the `assignable` clause of method m. Furthermore, let $\mathcal{F}_m^{\triangleleft}$ be the set of locations that have been freshly allocated *during* the execution of m. The little triangle $^{\triangleleft}$ indicates that this set contains the locations that have been freshly allocated in m and all methods directly or transitively called by m.

Let's assume a method m that is called by m' (i.e., $m' \hookrightarrow m$). According to the JML semantics, a location is assignable in m if it is either freshly allocated or it is in the set of locations evaluated from the `assignable` clause of m and it was already assignable in m'. We can write this condition as follows:

$$\mathcal{A}_m^{\mathsf{eff}} = \mathcal{F}_m^{\triangleleft} \cup (\mathcal{A}_m \cap \mathcal{A}_{m'}^{\mathsf{eff}}).$$

An important consequence of JML's semantics is that a runtime assertion checker needs to consider the `assignable` clauses of all methods on the call stack to determine whether a location is assignable. An alternative to this expensive check would be to enforce for each call that the `assignable` clause of the callee denotes a subset of the locations that are assignable in the caller. However, such a requirement would be overly conservative since it rejects certain calls based on the `assignable` clause of the callee rather than its actual behavior. Therefore, our checker actually inspects the call stack when necessary.

Datagroups. Datagroups are sets of locations. Every field of a program defines its own datagroup that initially contains only the field itself.

If we are not interested in the value of the field but only its datagroup, JML provides a special type `JMLDataGroup` to indicate that the field just serves as a declaration of the corresponding datagroup.

To add all locations in the datagroup of a location $o.f$ to a datagroup of the same object o, JML uses the **`in`** clause at the field declaration. In the class `Item` in Fig. 1, the two fields `name` and `price` are declared to be in the datagroup of the field `footprint`. We declare `footprint` of type `JMLDataGroup`, since we are interested in its datagroup, but not its value.

To add all locations in the datagroup of a location $o.f$ to a datagroup of another object p, JML uses the **`maps`** ... **`into`** clause. For instance in class `Node` in Fig. 1, we add `left.struct` to the datagroup of `struct`. The field `left` is called a *pivot field*, as an update of `left` changes the contents of datagroup `struct`. Since `left.struct` also has a datagroup itself, we essentially nest datagroups in our example. Adding locations from other objects makes a datagroup *dynamic*; the set of locations in the datagroup now depends on the program state.

Upon evaluation of an **`assignable`** clause in method m, the semantics states that each datagroup mentioned in the **`assignable`** clause is evaluated to a set of locations. We call this process *unfolding* of the datagroup. Datagroups that contain nested datagroups do not evaluate to nested sets of locations, but result in one single set of locations which is added to the set $\mathcal{A}_m$.

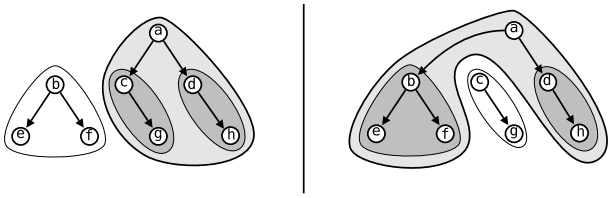

Fig. 2. A set of `Node` objects. Shapes depict the `struct` datagroups. Objects within the shape contain fields that the datagroup contains. Arrows depict references. Left: The situation in the pre-state of a call to `a.replace(c,b)`. Right: The post-state of the call.

Fig. 2 shows the dynamic datagroup of `a.struct` in light gray. One can see that `a.struct` also contains all locations that are mentioned in nested datagroups, depicted by a darker gray. The left picture shows the initial state where the pivot field `a.left` points to node `c`, the right picture depicts the datagroup of `a.struct` after executing the statement `a.left = b`. White shapes depict datagroups that are not in `a.struct`. This example illustrates that dynamic datagroups may contain different locations in different program states.

3 Checking Assignable Clauses with Static Datagroups

In a first step, we present an algorithm to check **`assignable`** clauses in the presence of *static* datagroups only. This part of the algorithm serves as a basis for checking **`assignable`** clauses with dynamic datagroups, as presented in the next section.

3.1 Data Structures

Our goal is to check **`assignable`** clauses in the presence of static datagroups in constant time. The following operations are involved in the check for field updates and we want them to perform in constant time: (1) lookup of all assignable fields of a given object without unfolding datagroups, (2) lookup of all datagroups that contain a location, and (3) the decision if the intersection between two sets of fields of an object is empty.

Field Identifiers. We assign a number to each field of a class such that at runtime, in the presence of inheritance, every field of an object has a unique number. In Fig. 1 we show these numbers in comments behind field declarations.

Assignable Locations. As explained in Sec. 2, the evaluation of **`assignable`** clauses leads to a set of locations. We give this set some structure and represent it as a map from objects to bitsets, in which the keys of the map are the objects of the locations, and each bit of the bitset correspond to a field of that object. Furthermore, as opposed to the semantical description of datagroups in Sec. 2, we do not unfold the static datagroups mentioned in such sets and instead deal with unfolding of datagroups on demand.

For method `copy` of class `Item` in Fig. 1, we represent the set $\mathcal{A}_{\texttt{copy}}$ by $\{\texttt{this} \mapsto [\checkmark\ \cdot\ \cdot\ \cdot\ ], \texttt{other} \mapsto [\ \cdot\ \checkmark\ \cdot\ \cdot\ ]\}$, where the fields receive their indices in the bitset in order of presence, as shown in the code, i.e., the first bit in the bitset of object `this` represents field `footprint` and the second bit in the bitset of object `other` represents field `selected`.

To retrieve the bitset of a given object o in a set $\mathcal{A}$, we write $\mathcal{A}[o]$. For example, $\mathcal{A}_{\texttt{copy}}[\texttt{other}]$ yields $[\ \cdot\ \checkmark\ \cdot\ \cdot\ ]$. If the map does not contain o, $\mathcal{A}[o]$ yields an empty bitset. To store object o with bitset bs in $\mathcal{A}$, we write $\mathcal{A}[o] \leftarrow bs$.

This design allows us to perform the operation (1) in constant time, as we can use `HashMap`s as the underlying data structure.

Fresh Locations. If an object is newly created, all locations of that object are *fresh*. To represent the set of fresh locations $\mathcal{F}^{\triangleleft}$, we need to save only the set of newly allocated objects, which implicitly gives us the set of fresh locations. The query $\mathcal{F}^{\triangleleft}[o]$ simply yields *true* if the object o is freshly allocated in the current method execution, and *false* otherwise.

Static Datagroups. For each field, we use a bitset to represent the datagroup(s) the field belongs to. That is, we equip every class with an array of bitsets. For class `Item`, we represent the static datagroups by the following array.

$$\begin{array}{ll} \texttt{footprint} & \\ \texttt{selected} & \\ \texttt{name} & \\ \texttt{price} & \end{array} \quad \begin{bmatrix} [\checkmark\ \cdot\ \cdot\ \cdot\ ] \\ [\ \cdot\ \checkmark\ \cdot\ \cdot\ ] \\ [\checkmark\ \cdot\ \checkmark\ \cdot\ ] \\ [\checkmark\ \cdot\ \cdot\ \checkmark] \end{bmatrix}$$

To access the datagroups that *statically* contain field f of class c, we write $\mathcal{D}^{\text{st}}[c@f]$. For instance, $\mathcal{D}^{\text{st}}[\texttt{Item@name}]$ yields $[\checkmark \cdot \checkmark \cdot]$, which means that `name` is in the datagroup of `footprint` (the first bit) and of course in its own datagroup (the third bit). For simplicity, we may also write $\mathcal{D}^{\text{st}}[o.f]$ to get the static datagroups of field f of class c, where o is of type c.

We set up the data structures for static datagroups such that we can perform the second and third operation described at the beginning of this section in constant time. Operation (2) involves an array access, and operation (3) involves computing the intersection of two bitsets, which is possible in constant time.

3.2 Code Instrumentation

An **`assignable`** clause restricts assignments throughout a method execution. This implies that checks of the **`assignable`** clause need to be performed throughout a method execution and not only in pre- and post-states. In the following, we present the code instrumentation to build up the necessary data structures and to check the validity of a field update. The relevant statements are: field updates (as these might violate the **`assignable`** clause), object creation (to track fresh locations), method invocation (to evaluate **`assignable`** clauses in the pre-state of a method and merge assignable sets), and method return (to update the assignable sets from the caller).

Field Update. Updating a field is the only way to violate an **`assignable`** clause. Before an update of a location $o.f$ in a method m called by m', we need to check if $o.f$ (hereafter referred to as loc) is in the set $\mathcal{A}^{\text{eff}}_m$. According to the semantics defined in Sec. 2, this is the case if either (1) the object of the location o has been freshly allocated during the execution of m and therefore is a member of the set $\mathcal{F}^{\triangleleft}_m$, or (2) loc is assignable in m' and the **`assignable`** clause of m either mentions the location itself or at least one datagroup that contains loc. Therefore, we need to check:

$$loc \in \mathcal{F}^{\triangleleft}_m \;\vee\; loc \in (\mathcal{A}_m \cap \mathcal{A}^{\text{eff}}_{m'}).$$

As explained in Sec. 3.1, we use maps of bitsets to represent sets of assignable locations and we do not unfold the datagroups. Checking if loc is in a set of fresh locations $\mathcal{F}^{\triangleleft}_m$ is performed by $\mathcal{F}^{\triangleleft}_m[o]$. Checking if loc is in a set of assignable locations $\mathcal{A}_m$ is performed by $\mathcal{A}_m[o] \cap \mathcal{D}^{\text{st}}[loc] \neq \emptyset$. That is, we get the bitset representing the fields of object o in $\mathcal{A}_m$ and intersect it with the bitset representing the datagroups that contain loc. If the intersection of the two bitsets is not empty, either $\mathcal{A}_m$ contains loc or it contains at least one datagroup that contains loc. We maintain the set $\mathcal{A}^{\text{eff}}_{m'}$ explicitly, as we explain below. This gives us the the following assertion that needs to hold at runtime:

$$\mathcal{F}^{\triangleleft}_m[o] \;\vee\; (\mathcal{A}_m \cap \mathcal{A}^{\text{eff}}_{m'})[o] \cap \mathcal{D}^{\text{st}}[loc] \neq \emptyset$$

All of these operations can be performed in constant time, which means that we can check the `assignable` clauses for field updates in constant time in the presence of static datagroups only.

Object Creation. On creation of a new object o in method m, all locations of o are fresh in m and in every transitive caller m_i of m. According to the semantics of fresh sets, we would have to add all fields of o to $\mathcal{F}^{\triangleleft}_{m}$ as well as to each $\mathcal{F}^{\triangleleft}_{m_i}$. Since any caller of m can observe newly allocated locations only *after* m returns, we add o only to $\mathcal{F}^{\triangleleft}_{m}$ and update the callers later.

Because of this simplification, the instrumentation of object creation can be performed in constant time and produces a memory overhead linear in the number of newly allocated objects.

Method Invocation. On invocation of method m, we evaluate the `assignable` clause of m to the set $\mathcal{A}_m$. For all location $o.f$ in the `assignable` clause, we enable the bit that represents f in the bitset $\mathcal{A}_m[o]$. To do this, we perform the following update: $\mathcal{A}_m[o] \leftarrow \mathcal{A}_m[o] \cup \mathcal{B}(f)$ where $\mathcal{B}(f)$ is the bitset in which only the bit for field f is enabled. Furthermore, we compute the intersection $\mathcal{A} = (\mathcal{A}_m \cap \mathcal{A}^{\mathsf{eff}}_{m'})$ because we need this set at every field update within the method and because this set does not change during method execution. We call this computation *merging* of assignable locations because we merge caller and callee. Our data structures allow efficient merging as follows: for each object that is a key in the maps of both $\mathcal{A}_m$ and $\mathcal{A}^{\mathsf{eff}}_{m'}$, we compute the intersection of the corresponding bitsets, $bs = \mathcal{A}_m[o] \cap \mathcal{A}^{\mathsf{eff}}_{m'}[o]$. If $bs \neq \emptyset$, we add it to the resulting set $\mathcal{A}[o] \leftarrow bs$, otherwise we just drop it.

The time and memory overhead at method invocation is linear in the number of objects that contain assignable fields.

Method Return. Before a method m may return to its caller m' the set of fresh locations $\mathcal{F}^{\triangleleft}_{m}$ needs to be added to $\mathcal{F}^{\triangleleft}_{m'}$.

This operation can be done with time overhead linear in the number of objects in $\mathcal{F}^{\triangleleft}_{m}$ and does not increase the memory overhead as $\mathcal{F}^{\triangleleft}_{m}$ will be consumed by the garbage collector.

4 Checking Assignable Clauses with Dynamic Datagroups

We extend the algorithm for checking `assignable` clauses to deal with *dynamic* datagroups, that is, datagroups that contain fields from other objects and therefore depend on the heap. We optimize our algorithm to cope well with situations that match the following two observations we made.

(1) A dynamic datagroup typically contains many locations through nested datagroups in recursive or aggregate data structures, whereas a location is typically only in a few datagroups of other objects. In our example in Fig. 1, the datagroup `struct` in class `Store` contains the field `root` and the fields `struct`,

`left`, and `right` of all n nodes in the store, that is, $3 \times n$ locations. By contrast, the field `struct` of a node is dynamically contained in no more than $log_2(n)$ datagroups, namely in the datagroups `struct` of all ancestors, assuming that the tree is balanced.

(2) **`assignable`** clauses are often quite unspecific, yet useful. This implies that the set of assignable locations is often very large although only a few locations actually get assigned to. An example supporting this observation is method `add` in class `Store`. As we do not want to reveal the internal data structure of the store, we specify that `add` is only allowed to assign to the datagroup `struct` of the store, i.e., the `root` field of the store and all `struct`, `left`, and `right` fields of the nodes. In other words, method `add` cannot change the content of any existing item, but may for instance balance the tree.

Because of these two observations, we do not unfold datagroups into sets of locations in the pre-state of a method as described in the semantics, which is potentially very expensive in both time and space. This decision raises three issues:

(1) We have to spent more effort to check if a field is assignable, as the information is not directly available.

(2) We can no longer merge sets of assignable locations of callers and callees upon method invocations. If we had to merge two sets that contain partially overlapping dynamic datagroups we would have to unfold the datagroups to find out which locations are in the intersection. Since we decide not to unfold datagroups, we cannot merge anymore.

(3) As the content of dynamic datagroups may change over time, we need to keep track of all changes in dynamic datagroups in order to reconstruct the assignable locations as of the pre-state of the method.

In the following sections, we explain how we can efficiently cope with these issues.

4.1 Data Structures

We do not change any of the existing data structures for checking assignable clauses, but add data structures to represent dynamic datagroups. We design our data structures such that it is possible to quickly find all datagroups that dynamically contain a location.

Dynamic Datagroups. To represent dynamic datagroups, we add an array of sets of locations to each object to store for each field of the object a set of datagroups that dynamically contain the field. We call these *back-links*, from the location back to the datagroup. For the object `g` of class `Node` (see Fig. 2), we therefore represent the dynamic datagroups by the following array, in which only the entry for field `struct` contains a back-link.

$$\begin{array}{ll} \texttt{struct} & \left[\begin{array}{l} \{\texttt{c.struct}\} \\ \{\} \\ \vdots \end{array}\right] \\ \texttt{left} & \\ \vdots & \end{array}$$

To access the set of datagroups that *dynamically* contain location *loc* in heap h over one pivot field, we write $\mathcal{D}_h^{\text{dyn}}[loc]$. Furthermore, we write $\mathcal{D}_h^{*\text{dyn}}[loc]$ for the reflexive transitive closure of $\mathcal{D}_h^{\text{dyn}}[loc]$. Implicitly, we also unfold static datagroups to calculate those sets. Since we evaluate dynamic datagroups in the pre-state of a method, we introduce the notation $h_0(m)$ to refer to the pre-heap of method m.

In our example in Fig. 2, $\mathcal{D}^{\text{dyn}}_{h_0(\texttt{replace})}[\texttt{g.struct}]$ yields the set $\{\texttt{c.struct}\}$, whereas $\mathcal{D}^{*\text{dyn}}_{h_0(\texttt{replace})}[\texttt{g.struct}]$ yields the set $\{\texttt{g.struct}, \texttt{c.struct}, \texttt{a.struct}\}$.

Assignable Stack. Since we no longer merge **`assignable`** clauses, we now have to check for each field update whether the updated location is assignable in each method on the call stack. To enable this check, we provide access to the sets of assignable and fresh locations for all methods on the call stack by passing a stack of assignable maps to the callee (rather than one merged assignable map).

Using stacks results in a memory footprint for storing assignable locations that grows linearly in the number of methods on the call stack.

4.2 Code Instrumentation

In order to support dynamic datagroups, we need to change the code instrumentation for field updates and method invocations, whereas object creation and method return stay unchanged.

Field Update. We reuse the efficient check of field updates for **`assignable`** clauses with static datagroups, but have to do additional work. Again, we need to check before updating a location $o.f$ (referred to as *loc*) in method m, if it is in $\mathcal{A}_m^{\text{eff}}$. Without merging assignable sets of locations, we do the following: for every method m_i that is in the call stack of m, we check that the location *loc* is either fresh in m_i or contained in the assignable set of locations of m_i. More formally:

$$\forall m_i \cdot m_i \hookrightarrow^* m \implies loc \in \mathcal{F}^{\triangleleft}_{m_i} \ \vee \ loc \in \mathcal{A}_{m_i}.$$

Since we do not update the set of fresh locations for all transitive callers of m, we need to add some extra logic to find out if *loc* is fresh in m_i. This is the case if *loc* has been freshly allocated during execution of m_i, that is, either in m_i itself or some callee of m_i. We can express this by $\exists m_k \cdot m_i \hookrightarrow^* m_k \wedge \mathcal{F}^{\triangleleft}_{m_k}[o]$. Although this looks more complicated, it actually allows us to simplify the implementation considerably.

Since we do not unfold dynamic datagroups, we need to perform some computation to check if *loc* is assignable. *loc* is in $\mathcal{A}_{m_i}$, if we find a datagroup *dg* that both dynamically contains the location *loc*, and is mentioned in the **`assignable`** clause. We write this as $\exists dg \cdot dg \in \mathcal{D}^{*\text{dyn}}_{h_0(m_i)}[loc] \wedge dg \in \mathcal{A}_{m_i}$. We reuse our technique from the static datagroups to replace $dg \in \mathcal{A}_{m_i}$ by $\mathcal{A}_{m_i}[obj(dg)] \cap \mathcal{D}^{\text{st}}[dg] \neq \emptyset$, see Sec. 3.2.

The time complexity for finding a datagroup that dynamically contains *loc* is linear in the size of $\mathcal{D}^{*\text{dyn}}[loc]$ multiplied by the number of methods on the call stack. However, we can dramatically speed up this lookup by introducing caches for finding dynamic datagroups, see Sec. 5. So in summary, we check the following assertion at runtime:

$$\begin{aligned}&\forall m_{\text{i}} \cdot m_{\text{i}} \hookrightarrow^* m \implies \\ &\quad \exists m_{\text{k}} \cdot m_{\text{i}} \hookrightarrow^* m_{\text{k}} \wedge \mathcal{F}^{\lhd}_{m_{\text{k}}}[o] \quad \vee \\ &\quad \exists dg \cdot dg \in \mathcal{D}^{*\text{dyn}}_{h_0(m_{\text{i}})}[loc] \wedge \mathcal{A}_{m_{\text{i}}}[obj(dg)] \cap \mathcal{D}^{\text{st}}[dg] \neq \emptyset\end{aligned}$$

Updating a Pivot Field. Whenever we update a pivot field of a datagroup, we change the content of the datagroup. This is a problem because upon a method call, we do not unfold the datagroups mentioned in the **`assignable`** clause of the callee, even though the semantics of **`assignable`** clauses prescribes that the set of assignable locations is to be determined in the pre-state of the method. Consequently, any change to a datagroup mentioned in an **`assignable`** clause needs to be tracked in order to be able to reconstruct the situation in the pre-state of a method.

We apply a technique that we call *lazy unfolding*. If we update a pivot field of a datagroup that is contained in an assignable map, we perform two operations. (1) We add the old location that was contained in the datagroup via the pivot field before the update directly to the assignable map. By doing this, the location stays assignable although it is not in the datagroup anymore. (2) We add additional information to the assignable map, stating that the back-link of the new location that is contained in the datagroup via the pivot field after the update should not be considered when we check whether a location is assignable. By doing this, we can cut away parts of datagroups in assignable maps.

Note that this information needs to be stored per assignable map and not per datagroup as every assignable map is evaluated in a different state, and thus has a different set of assignable locations for the same datagroup.

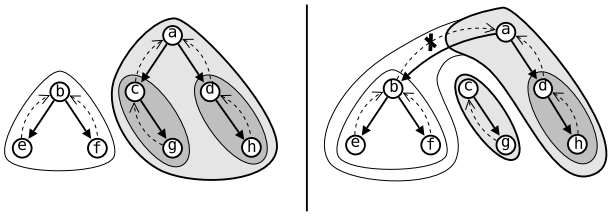

Fig. 3. The same situation as in Fig. 2. Dashed arrows depict the back-links from locations to datagroups. The cross depicts the back-link that has been invalidated in the assignable map of method `replace`.

Fig. 3 shows how the lazy unfolding works in our running example. On the left side, we see again the situation in the pre-state, the locations in the

gray shapes are assignable. On the right side, we see the datagroups after the update `a.left = b`. To preserve the assignable locations of the pre-state, we add `c.struct` explicitly to the assignable map, which preserves the assignability of the locations of objects `c` and `d`. Furthermore, we mark the back-link from `b.struct` to `a.struct` in the assignable map as invalid, which essentially renders the locations of objects `b`, `e`, and `f` not assignable. Looking at the locations in the gray shapes, one can see that we effectively preserved the assignability of locations in the pre-state on the left-hand side although the datagroup changed in the meantime.

Method Invocation. On invocation of method m, we evaluate the **`assignable`** clause of m to the set $\mathcal{A}_m$ as described in Sec. 3.2, but we do not merge the assignable sets of the caller and the callee. Instead, we add the sets of assignable and fresh locations from method m to the stack of assignable maps.

5 Introducing Caches

As we have shown in the last section, the time overhead to check if a location *loc* is assignable depends on the height of the call stack and on the size of the set of dynamic datagroups $\mathcal{D}^{*\mathsf{dyn}}[loc]$. For every update of location *loc*, we check if *loc* is assignable in all assignable maps on the stack. This involves to compute the set $\mathcal{D}^{*\mathsf{dyn}}[loc]$ and to check for each datagroup in that set whether it is mentioned in the assignable map or not. That is, we spend a considerable effort to check if a location is assignable.

In this process of checking, we gain a lot of additional information. We learn which datagroups in $\mathcal{D}^{*\mathsf{dyn}}[loc]$ are mentioned in what assignable map, and we also learn which datagroups in $\mathcal{D}^{*\mathsf{dyn}}[loc]$ are not assignable. We can reuse all this information since the set of locations of an **`assignable`** clauses is computed in the pre-state and does not change during method execution.

We equip each assignable map with a cache that stores all the additional information from the queries since the method invocation. The information in the cache is valid for this assignable map as long as the corresponding method executes. Caches become especially useful if we assign to the same set of locations several times in a method, for instance when doing a computation in a loop.

6 Evaluation

6.1 Experimental Results

As a proof of concept, we implemented the algorithm described in this paper to check **`assignable`** clauses in Java programs.

To test the efficiency of our algorithm, we chose a doubly-linked list, where the nesting of datagroups is as deep as the number of nodes in the list: every node is equipped with a `struct` datagroup that contains the `next` and `previous` fields and dynamically contains the `struct` field of the successor node.

We performed experiments with different list operations to measure the performance of our algorithm[1]. The most interesting experiment has been to reverse large doubly-linked list, which involves operations on every node of the list and changes the structure of the dynamic datagroups completely. In fact, every pivot field gets assigned to, which leads to a complete unfolding of the datagroup. This is the worst case scenario for our algorithm, which tries to avoid unfolding as much as possible.

Surprisingly, we need only a bit more than one seconds to add 10'000 nodes to a list and reverse it with the runtime assertion checker enabled. We spend around 80% of the time to add the nodes, and 20% of the time to reverse the list. The memory footprint is around 20MB before reversing the list and grows to 25MB during reversing because of the caches that get filled in the process. If we switch off runtime assertion checking for the same example, the program terminates within half a second and uses around 2.5MB. When repeating the experiment with 20'000 nodes, time and memory consumption doubles for both versions.

For the doubly-linked list, the runtime overhead of our checker is a factor of 2 and the memory overhead is a factor of 10. For the main applications of runtime assertion checking (to prepare static verification and to reproduce possibly spurious verification errors), we consider this overhead acceptable, especially for recursive data structures such as our doubly-linked list. We expect the overhead to be significantly smaller for non-recursive aggregate structures, where dynamic datagroups are not nested as deeply.

6.2 Theoretical Results

Our algorithm depends mainly on the following factors: the size of the set of dynamic datagroups that contain a location ($|\mathcal{D}^{*\mathsf{dyn}}|$), the size of the assignable sets ($|\mathcal{A}|$), and the size of the call stack ($|cs|$),

Time Complexity. Field update is the only operation that may generate a significant time overhead. The check if a location is assignable has a time complexity of $\mathcal{O}(|\mathcal{D}^{*\mathsf{dyn}}| \times |cs|)$ if the result is not cached in any assignable map, and $\mathcal{O}(|cs|)$ if the result is cached in all assignable maps. That is, the caches have a big impact on the performance if we have a deep nesting of `assignable` clauses. We also see that we do not have a good solution for recursive method calls, where $|cs|$ gets big.

In our running example, $|\mathcal{D}^{*\mathsf{dyn}}|$ is logarithmic to the number of nodes in the tree, which leads to a very good performance.

Memory Overhead. The data structures that produce a significant memory overhead are the ones for storing the sets of assignable locations, including caches. That is, the memory overhead depends on the number of assignable locations

[1] On a desktop computer with a single core 3.4 GHz CPU

mentioned in the **assignable** clauses, the amount of lazy unfolding and of course the number of methods on the call stack. We get an overhead of $\mathcal{O}(|\mathcal{A}| \times |cs|)$, where the size of $\mathcal{A}$ depends on how much unfolding happened already.

In our running example, if we have a method with an **assignable** clause stating '`a.struct`' $|\mathcal{A}|$ initially contains only the location `a.struct` and our memory overhead is very small. For each `left` or `right` pointer that we assign to in a method, we add one more location to that set, and, if we completely reorder the whole tree, end up in a complete unfolding of the datagroup.

7 Related Work

Cheon's runtime assertion checker for JML [4] provides data structures to represent **assignable** clauses and datagroups but does not generate checks for it. Ye uses those data structures in his thesis [12] to implement an **assignable** clause checker in the presence of static datagroups only. The checks have a time overhead linear in the size of the set of locations from the **assignable** clause, whereas our algorithm for static datagroups works in constant time.

The CHASE tool [3] provides a simple means to discover common specification mistakes, but is not designed to be sound. It performs a purely syntactic check on **assignable** clauses, ignores aliasing, and does not support datagroups.

Spoto and Poll [10] formalized a trace semantics for a sound reasoning on **assignable** clauses. Their approach takes aliasing into account, but datagroups are not supported. They conclude that JML's **assignable** clause may be unsuited for a precise and correct static analysis.

The LOOP tool [11] generates PVS proof obligations for a given JML annotated Java program. It is mainly used to prove non-trivial properties of JavaCard applications. LOOP can deal with **assignable** clauses, but datagroups are not taken into account.

KRAKATOA [9] is a verification tool for Java. The specification language of KRAKATOA is similar to JML and contains an *assigns* clause to specify a list of locations that can be assigned. Again, it is not possible to apply information hiding by using datagroups.

The KEY system [1] allows one to verify Java programs against JML specifications. KEY handles **assignable** clauses, but not datagroups.

ESC/JAVA2 supports most JML annotations, including **assignable** clauses and datagroups. However, ESC/JAVA2 fails to give a precise and correct answer on **assignable** clauses that mention datagroups.

Spec#[2] does not provide datagroups, but instead uses a hierarchical heap model to provide abstraction; if a *modifies* clause allows modification of an object o then all (committed) objects that have o as (transitive) owner can be modified as well. This is similar to declaring a datagroup in each object that contains the locations of that object and all (transitively) owned objects. Therefore, we expect that our algorithm, especially the idea of lazy unfolding, can also be applied to Spec#.

8 Conclusion

We presented an algorithm to check **`assignable`** clauses in the presence of static and dynamic datagroups. Our algorithm performs well, in particular, on recursive data structures with large and deeply nested dynamic datagroups by introducing the concept of lazy unfolding of datagroups. We provide the foundation to close a big gap in the runtime assertion checker of JML. The algorithm has been tested against recursive data structures with a prototypical implementation of a runtime assertion checker.

We plan to prove correctness of our algorithm by adding an operational semantics to our JML formalization in Coq [6] that includes the runtime assertion checks and show that the algorithm enforces the semantics of **`assignable`** clauses. Moreover, we intend to contribute our algorithm to the OpenJML project and to use that implementation for larger experiments.

Acknowledgments. We are grateful to Alex J. Summers and the anonymous reviewers for helpful comments.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *SS*, 2004.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.
3. N. Cataño and M. Huisman. Chase: A static checker for JML's assignable clause. In *VMCAI*, volume 2575 of *LNCS*, pages 26–40. Springer, 2003.
4. Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, 2003.
5. ESC/Java2. http://secure.ucd.ie/products/opensource/ESCJava2.
6. A. Kägi, H. Lehner, and P. Müller. A formalization of JML in the Coq proof system. Technical report, ETH Zurich, 2009. Available at http://www.pm.inf.ethz.ch/people/lehnerh/jmlcoq.
7. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA*, pages 144–153, 1998.
8. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI*, pages 246–257, 2002.
9. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *JLAP*, 58:89–106, 2004.
10. F. Spoto and E. Poll. Static analysis for JML's assignable clauses. In G. Ghelli, editor, *FOOL*, 2003.
11. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *TACAS*, pages 299–312. Springer, 2001.
12. C. Ye. Improving JML's assignable clause analysis. Technical report, Iowa State University, 2006.