Static Analysis for JML's `assignable` Clauses*

Fausto Spoto[1] and Erik Poll[2]

[1] Dipartimento di Informatica, Verona, Italy
`spoto@sci.univr.it`
[2] University of Nijmegen, The Netherlands
`erikpoll@cs.kun.nl`

Abstract The specification language JML (Java Modelling Language) includes so-called `assignable` clauses, also known as `modifies` clauses, for specifying which fields may change their value as side-effect of a method. This paper uses abstract interpretation over a trace semantics for a simple object-oriented language to define a correct static analysis for checking the correctness of `assignable` clauses.

1 Introduction

JML (for Java Modeling Language) [4,5] is a specification language for Java. It allows assertions to be included in Java code, specifying pre- and postconditions and invariants in the style of Eiffel and the well-established *design by contract* approach [9], but JML is much more expressive. For instance, a specification of a method can also include a so-called `assignable` (or `modifies`) clause. It specifies which locations may be changed by the method (a *frame condition*), in a style similar to [6]. These locations are described through a set of fields. JML offers a rich syntax for expressing `assignable` clauses. We will not concern ourselves with this here. An example of an `assignable` clause for the method `update` of the class `myclass` in Figure 1 would be

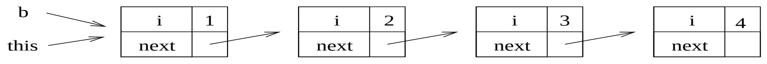
```
        assignable this.i, this.next, b.next.next;
```

The information conveyed by an `assignable` clause is essential for reasoning about methods. For instance, it is used in the LOOP verification tool [8].

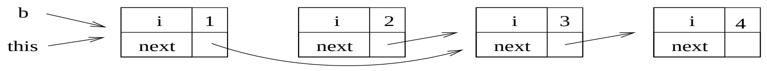
At first sight, it seems that checking the correctness of `assignable` clauses (or, better still, generating correct `assignable` clauses) is something that could easily be automated. The Chase tool [1] performs a syntactical analysis to automatically check `assignable` clauses. Basically, the tool checks if every assignment in a method is to a variable listed in its `assignable` clause. The assignments for a method call are those listed in its `assignable` specification. The full syntax of `assignable` specifications is allowed. Unfortunately, as the developers of the Chase tool are well aware, the syntactic analysis it performs has its limitations. Because of aliasing, `assignable` clauses are trickier than they may seem at first sight. For example, the assignable clause given above for the method

* This work has been partially funded by MURST grant *Abstract Interpretation, Type Systems and Control-Flow Analysis* and EPSRC grant GR/R53401.

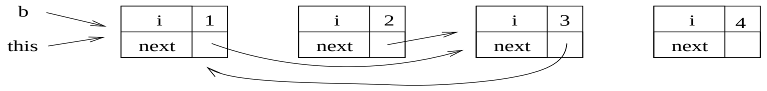
`update` of the class `myclass` in Figure 1 is incorrect, although the Chase tool (and the average reader?) will not spot this. Consider, indeed, what happens during the execution of the method `update` if `this` and `b` are *aliases* (*i.e.* refer to the same object) when it is invoked. We can represent this situation as



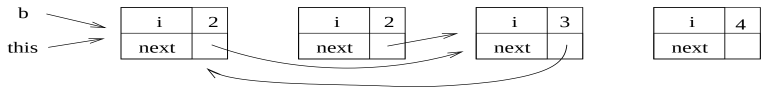
The assignment `this.next:=b.next.next` modifies the state as follows:



The assignment `b.next.next:=b` results in the following state:



The assignments `out:=b.next; this.i:=this.i+1` result in the state (`out` is not shown)



You can see that the modified locations are `this.i`, `this.next` (aka `b.next`, and `this.next.next.next` (aka `b.next.next.next`). But the `assignable` clause mentioned above does not allow `this.next.next.next` to be modified.

Although the syntactical analysis performed by Chase is neither sound nor complete, it is very useful to quickly spot many potential mistakes in `assignable` clauses. We go here one step beyond, by developing a sound static analysis for checking `assignable` clauses. Namely, these are our contributions.

- We formalise the `assignable` clause of a method as an abstract interpretation $\mathcal{A}$ of its trace semantics.
- We show that the abstract domain $\mathcal{A}$ is not useful for static analysis, since it lacks good compositionality properties. Hence we *refine* it into a domain $\mathcal{AR}$ which features better compositionality results. The idea here is to keep track during the analysis of which locations have been stored in each variable. In this way, we can safely approximate the set of locations modified by an assignment. For instance, the command `a:=b` copies the location l of `b` to `a`. Since $\mathcal{AR}$ keeps track of this (while $\mathcal{A}$ does not), it is able to conclude that a subsequent assignment `a.x=6` actually changes the location of the field `x` of l, and not that of the field `x` of the location that `a` originally pointed to at the beginning of the method.
- We show that a static analysis over $\mathcal{AR}$ spots the erroneous `assignable` clause mentioned above and accepts/computes the correct assignable clause for `update`, namely
  ```
  assignable this.i, this.next, b.next.next, b.next.next.next;
  ```

Proofs of the theoretical results can be found in [13].

```
class myclass :                                  class subclass extends myclass :
field i : int                                    field j : int
field next : myclass                             method update(b : myclass) : subclass is
method update(b : myclass) : myclass is            this.i := this.j + 1;
  this.next := b.next.next;                        out := this
  b.next.next := b;
  out := b.next;
  this.i := this.i + 1
```

$$\mathcal{K} = \{\texttt{myclass}, \texttt{subclass}\} \quad \texttt{subclass} \le \texttt{myclass}$$

$$F(\texttt{myclass}) = [\texttt{i} \mapsto int, \texttt{next} \mapsto \texttt{myclass}] \quad F(\texttt{subclass}) = [\texttt{i} \mapsto int, \texttt{j} \mapsto int, \texttt{next} \mapsto \texttt{myclass}]$$

Figure 1. A program and its static information.

2 Preliminaries

The powerset of a set S is $\wp(S)$. A total (partial) map f is denoted by $\mapsto$ ($\rightarrow$). Its *domain* (*codomain*) is $\mathsf{dom}(f)$ ($\mathsf{rng}(f)$). We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ the map f where $dom(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its *update* is $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain may be enlarged.

The two components of a *pair* are separated by $\star$. A definition of S such as $S = a\star b$, with a and b meta-variables, silently defines the pair selectors $s.a$ and $s.b$ for $s \in S$. For instance, Definition 5 implicitly defines $o.\kappa$ and $o.\phi$ for $o \in Obj$.

We recall now the basics of abstract interpretation [2]. Let $C\star\le$ and $A\star\preceq$ be two partially ordered sets (or *posets*, the *concrete* and the *abstract* domain). A *Galois connection* is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ such that $\gamma\alpha$ is extensive and $\alpha\gamma$ is reductive. An abstract operator $\hat{f} : A^n \rightarrow A$ is *correct w.r.t.* $f : C^n \rightarrow C$ if $\alpha f \gamma \preceq \hat{f}$ (here f is applied pointwise).

3 The Framework of Analysis

We build on a denotational trace semantics which interprets every expression or command as a map from input states to traces of states (see [12,11]).

A *type environment* assigns types to a finite set of variables. From now on, every τ will implicitly denote a type environment.

Definition 1. *Let Id be a set of* identifiers*,* $\mathcal{K}$ *a finite set of* classes *ordered by a* subclass relation $\le$ *such that* $\mathcal{K}\star\le$ *is a poset. Let Type be the set* $\{int\} + \mathcal{K}$*. We extend* $\le$ *to Type by defining* $int \le int$*. Let* $Vars \subset Id$ *be a set of* variables *such that* $\{\texttt{out}, \texttt{this}\} \subseteq Vars$*. We define the set of* type environments

$$TypEnv = \{\tau : Vars \rightarrow Type \mid \mathsf{dom}(\tau) \textit{ is finite, if } \texttt{this} \in \mathsf{dom}(\tau) \textit{ then } \tau(\texttt{this}) \in \mathcal{K}\}.$$

Expressions and commands are given in Definition 2. They specify a simple object-oriented language (used for instance in Figure 1) which can be considered as the kernel of real-world object-oriented languages.

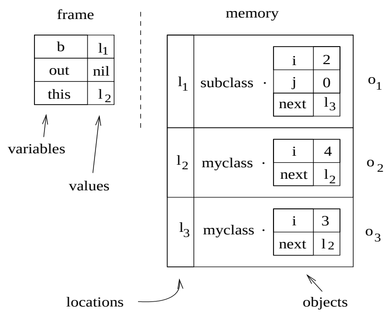


Figure 2. State $\phi_1 \star \mu_1$ for $\tau = [\mathtt{b} \mapsto \mathtt{myclass}, \mathtt{out} \mapsto \mathtt{myclass}, \mathtt{this} \mapsto \mathtt{myclass}]$.

Definition 2. Expressions $\mathcal{E}$ *and* commands $\mathcal{C}$ *are defined by the grammar*

$$e ::= i \mid v \mid \mathtt{nil} \mid e \ bop \ e \mid \mathtt{is\_nil}(e) \mid \mathtt{new} \ \kappa \mid e.f \mid e.m(v_1, \dots, v_n)$$
$$c ::= (v' := e) \mid (e.f := e) \mid c; c \mid \mathtt{let} \ v' {:} t \ \mathtt{in} \ c \mid \mathtt{if} \ e \ \mathtt{then} \ c \ \mathtt{else} \ c$$

with bop is a generic binary operation, $t \in$ Type, $\kappa \in \mathcal{K}$, $i \in \mathbb{Z}$, $f, m \in$ Id, $v, v', v_1, \dots, v_n \in$ Vars and $v' \neq$ `this`. The operators $=$ and $+$ work over integer expressions only. Note that we distinguish between variables*, which are identifiers local to a method, and* fields*, which are identifiers local to an object.*

A class contains *fields* and functions called *methods*. A method has a set of input/output variables called *parameters*, including a special parameter `out`, which holds the result of the method, and `this`, which is the object on which the method has been called. *Fields* is a set of maps which bind each class to the type environment of its fields. The variable `this` cannot be a field.

Definition 3. *Fields* $= \{F : \mathcal{K} \mapsto \mathit{TypEnv} \mid \mathtt{this} \notin \mathsf{dom}(F(\kappa))$ *for every* $\kappa \in \mathcal{K}\}$.

The *static information* of a program is used by a static analyser.

Definition 4. *The* static information *of a program consists of a poset* $\mathcal{K} \star \leq$ *and of a map* $F \in$ *Fields.*

An example program and its static information are shown in Figure 1.

We define a *frame* as a map that assigns values to variables. These *values* can be integers, locations or *nil* where a *location* is a memory cell. The value assigned to a variable must be consistent with its type. For instance, a class variable should be assigned to a location or to *nil*. A *memory* is a map from locations to objects where an *object* is characterized by its class and the frame of its fields. Figure 2 illustrates these different concepts.

Definition 5. *Let Loc be an infinite set of* locations *and* $Value = \mathbb{Z} + Loc + \{nil\}$*. We define* frames*,* objects *and* memories *as*

$$\begin{aligned} Frame_\tau &= \left\{ \phi \in \mathsf{dom}(\tau) \mapsto Value \;\middle|\; \begin{array}{l} \textit{for every } v \in \mathsf{dom}(\tau) \\ \textit{ if } \tau(v) = int \textit{ then } \phi(v) \in \mathbb{Z} \\ \textit{ if } \tau(v) \in \mathcal{K} \textit{ then } \phi(v) \in \{nil\} \cup Loc \end{array} \right\} \\ Obj &= \{\kappa \star \phi \mid \kappa \in \mathcal{K},\ \phi \in Frame_{F(\kappa)}\} \\ Memory &= \{\mu \in Loc \to Obj \mid \mathsf{dom}(\mu) \textit{ finite}\}\ . \end{aligned}$$

Example 1. Let $\tau = [\mathtt{b} \mapsto \mathtt{myclass}, \mathtt{out} \mapsto \mathtt{myclass}, \mathtt{this} \mapsto \mathtt{myclass}]$ be the type environment inside the method `update` of the class `myclass` (Figure 1). Let $l_1, l_2 \in Loc$. $Frame_\tau$ contains $\phi_1 = [\mathtt{b} \mapsto l_1, \mathtt{out} \mapsto nil, \mathtt{this} \mapsto l_2]$ (Figure 2).

Example 2. Objects of class `myclass` (consistent with $F(\mathtt{myclass})$ in Figure 1), are $o_2 = \mathtt{myclass}\star[\mathtt{i} \mapsto 4, \mathtt{next} \mapsto l_2]$ and $o_3 = \mathtt{myclass}\star[\mathtt{i} \mapsto 3, \mathtt{next} \mapsto l_2]$. An object of class `subclass` (consistent with $F(\mathtt{subclass})$) is $o_1 = \mathtt{subclass}\star[\mathtt{i} \mapsto 2, \mathtt{j} \mapsto 0, \mathtt{next} \mapsto l_3]$. *Memory* contains $\mu_1 = [l_1 \mapsto o_1, l_2 \mapsto o_2, l_3 \mapsto o_3]$ (Figure 2).

A notion of *type correctness* $\phi\star\mu : \tau$, with $\phi \in Frame_\tau$ and $\mu \in Memory$, constrains locations to contain objects allowed by τ (see [11,12] for details). Note that we require `this` to be bound.

Definition 6. *We define the* states $\Sigma = \cup\{\Sigma_\tau \mid \tau \in TypEnv\}$*, where*

$$\Sigma_\tau = \left\{ \phi\star\mu \;\middle|\; \begin{array}{l} \phi \in Frame_\tau,\ \mu \in Memory,\ \phi\star\mu : \tau \\ \mathtt{this} \in \mathsf{dom}(\tau) \textit{ entails } \phi(\mathtt{this}) \neq nil \end{array} \right\}\ .$$

A state $\phi_1\star\mu_1 \in \Sigma_\tau$ is shown in Figure 2. We define now the *traces* of states. This definition will be refined in Definition 13, in order to ban traces which do not represent the execution of any expression or command. This will be needed in the correctness proofs for Section 6.

Definition 7. *The set $\mathcal{T}$ of* traces *over Σ is the set of non-empty sequences in Σ. In particular,*

- *a* convergent *trace $\sigma_1 \to \cdots \to \sigma_n$ represents a terminated computation,*
- *a* divergent *trace $\sigma_1 \to \cdots \to \sigma_n \to \cdots$ represents a divergent computation.*

The first state of $t \in \mathcal{T}$ is $\mathsf{fst}(t)$*. By* $\mathsf{div}(t)$ *we mean that t is divergent. If* $\neg\,\mathsf{div}(t)$*, the last state of t is* $\mathsf{lst}(t)$*. We define the set $\mathcal{T}_\tau$ of traces which, if they are not divergent, end with a state in Σ_τ. Namely,* $\mathcal{T}_\tau = \{t \in \mathcal{T} \mid \textit{if } \neg\,\mathsf{div}(t) \textit{ then } \mathsf{lst}(t) \in \Sigma_\tau\}$*.*

Expressions and commands are *denoted* by a map from an initial state to a trace t. If the execution terminates then t is *convergent*. Otherwise it is *divergent*. A special variable *res* holds the value v of an expression *i.e.,* if $\neg\,\mathsf{div}(t)$ then $\mathsf{lst}(t)(res) = v$. We will often identify a command or expression with its denotation (see for instance Example 8).

Definition 8. *We define the* denotations

$$D_{\tau,\tau'} = \{d \in \Sigma_\tau \mapsto \mathcal{T}_{\tau'} \mid \textit{for every } \sigma \in \Sigma_\tau \textit{ we have } \mathsf{fst}(d(\sigma)) = \sigma\} \ .$$

Here τ refers to the beginning of the computation, and τ' to its end.

A denotational semantics over the denotations of Definition 8 is defined in [12,11]. For reasons of space, we just say here that its main operations are an operator $\otimes$ for sequential composition of denotations *i.e.,* the denotation of the command $c_1; c_2$ is the application of $\otimes$ to the denotations of c_1 and c_2; an operator $\oplus$ for the disjunctive composition of denotations (at the end of a conditional or among the different targets of a virtual method call). Note that a `while` loop can be interpreted as a fixpoint. Although a bytecode granularity is considered in [12,11], for simplicity we consider a high-level approach here, and we do not introduce the explicit construction of the semantics.

4 The Property $\mathcal{A}$

We define here a formal semantics of `assignable` clauses as a property (an abstract interpretation) of denotations.

A *path* is a period-separated non-empty sequence of identifiers.

Definition 9. *A* path *for τ of length 1 is every $v \in \mathsf{dom}(\tau)$. We let $type_\tau(v) = \tau(v)$. One of length $i \geq 2$ is $p.f$, where p is a path of length $i-1$, $type_\tau(p) = \kappa \in \mathcal{K}$ and $f \in \mathsf{dom}(F(\kappa))$. We let $type_\tau(p.f) = F(\kappa)(f)$. The paths for τ of positive length are denoted by $Paths_\tau$. Those of length at least 2 by $Paths'_\tau$.*

Example 3. Let $\tau = [\mathtt{b} \mapsto \mathtt{myclass}, \mathtt{out} \mapsto \mathtt{myclass}, \mathtt{this} \mapsto \mathtt{myclass}]$ (Example 1). Then $Paths_\tau \supseteq \{\mathtt{b}, \mathtt{out}, \mathtt{this}, \mathtt{b.i}, \mathtt{b.next}, \mathtt{out.i}, \mathtt{b.next.i}, \mathtt{this.next.next}\}$ and $Paths'_\tau \supseteq \{\mathtt{b.i}, \mathtt{out.i}, \mathtt{out.next}, \mathtt{b.next.i}, \mathtt{this.next.next}\}$ but $\mathtt{b} \notin Paths'_\tau$.

A *handle* specifies a position in the memory or the frame where the value of a field or variable is stored. It is a pair consisting of the location of an object o and of an identifier. If the location is *nil*, the *value* of the handle is that of a variable in the frame, otherwise it is that of a field of o.

Definition 10. *A* handle *is an element of the set $H = (Loc \cup \{nil\}) \times Id$. The* value *$\nu(h, \sigma)$ of a handle $h \in H$ in a state $\phi\star\mu \in \Sigma_\tau$ is defined as*

$$\nu(\langle nil, v\rangle, \phi\star\mu) = \phi(v) \quad \textit{(undefined if } v \notin \mathsf{dom}(\phi))$$
$$\nu(\langle l, f\rangle, \phi\star\mu) = \mu(l).\phi(f) \quad \textit{(undefined if } l \notin \mathsf{dom}(\mu) \textit{ or } f \notin \mathsf{dom}(\mu(l).\phi)).$$

Example 4. In Figure 2, the value 2 of the field `i` of o_1 is accessible through the handle $\langle l_1, \mathtt{i}\rangle$. The value l_2 of the variable `this` through the handle $\langle nil, \mathtt{this}\rangle$.

We define a handle which allows us to access the value of a path in a state.

Definition 11. *Let $p \in Paths_\tau$ and $\sigma \in \Sigma_\tau$. The* handle $[\![p]\!]_\sigma$ of p in σ *is*

$$[\![v]\!]_\sigma = \langle nil, v \rangle, \quad [\![p.f]\!]_\sigma = \begin{cases} \langle \nu([\![p]\!]_\sigma, \sigma), f \rangle & \text{if } [\![p]\!]_\sigma \text{ is defined, } \nu([\![p]\!]_\sigma, \sigma) \in Loc \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The map $[\![]\!]$ is pointwise extended to $\wp(Paths_\tau)$. Given $p_1, p_2 \in Paths_\tau$, if $[\![p_1]\!]_\sigma = [\![p_2]\!]_\sigma$ then we say that p_1 and p_2 are aliases *in σ (they refer to the same handle).*

Example 5. Consider again the state σ_1 in Figure 2. We have $[\![\texttt{b}]\!]_{\sigma_1} = \langle nil, \texttt{b} \rangle$, $[\![\texttt{b.i}]\!]_{\sigma_1} = \langle l_1, \texttt{i} \rangle$ while $[\![\texttt{out.next}]\!]_{\sigma_1}$ is undefined. Moreover, $[\![\texttt{b.next.next.next}]\!]_{\sigma_1} = \langle l_2, \texttt{next} \rangle = [\![\texttt{this.next}]\!]_{\sigma_1}$, hence `b.next.next.next` and `this.next` are aliases.

A location is *reachable* from $P \in \wp(Paths_\tau)$ if it is stored in a state at a position that a path in P points to.

Definition 12. *Let $P \in \wp(Paths_\tau)$, $\sigma \in \Sigma_\tau$ and $l \in Loc$. We say that l is* reachable in σ from P *if there exists $p \in P$ such that $\nu([\![p]\!]_\sigma, \sigma) = l$.*

We restrict the set of denotations to ban meaningless cases. For instance, only reachable locations can be updated. This is essential for Definition 14.

Definition 13. *We say that $\phi_1 \star \mu_1 \in \Sigma_{\tau_1}$* follows *$\phi_2 \star \mu_2 \in \Sigma_{\tau_2}$ if*

1. *$\mathsf{dom}(\mu_1) \subseteq \mathsf{dom}(\mu_2)$ and for every $l \in \mathsf{dom}(\mu_1)$ we have $\mu_1(l).\kappa = \mu_2(l).\kappa$ (locations do not disappear and objects cannot change class);*
2. *if $l \in \mathsf{dom}(\mu_1)$ and l is reachable in $\phi_2 \star \mu_2$ then l is reachable in $\phi_1 \star \mu_1$ (reachability of non-fresh variables cannot be forged);*
3. *if $l \in \mathsf{dom}(\mu_1)$ is not reachable in $\phi_1 \star \mu_1$ then $\mu_1(l) = \mu_2(l)$ (unreachable objects are not updated).*

We modify Definition 8 by requiring that $\sigma' \in c(\sigma)$ follows σ for every $\sigma \in \Sigma_\tau$.

To formalise the notion of *update along a trace t*, we say that the value of a given handle, read in different states of t, is different. A "benevolent" or "temporary" change is considered as an update by this definition. If we do not like this, we should compare the value of the handle in the first and the last states of t only.

Definition 14. *Let $h \in H$, $\sigma_1 \in \Sigma_\tau$ and $\sigma_2 \in \Sigma_{\tau'}$ be such that σ_2 follows σ_1. We define $\mathsf{assigned}(h, \sigma_1, \sigma_2)$ if and only if $\nu(h, \sigma_1)$ is defined and $\nu(h, \sigma_1) \neq \nu(h, \sigma_2)$. $\mathsf{assigned}$ is extended to $t \in \mathcal{T}$ as the set of handles that change along t i.e., $\mathsf{assigned}(t) = \{h \in H \mid$ there exists $\sigma_2 \in t$ such that $\mathsf{assigned}(h, \mathsf{fst}(t), \sigma_2)\}$.*

We want to approximate every command c with a set $P \subseteq Paths'_\tau$ whose locations are allowed to be assigned in the traces of the denotation of c.

Definition 15. *We define $\gamma_{\tau,\tau'} : \wp(Paths'_\tau) \mapsto \wp(D_{\tau,\tau'})$ such that $\gamma_{\tau,\tau'}(P)$ contains the denotations that modify only handles in P i.e., $\gamma_{\tau,\tau'}(P) = \{d \in D_{\tau,\tau'} \mid$ for every $\sigma \in \Sigma_\tau$ we have $\mathsf{assigned}(d(\sigma)) \subseteq [\![P]\!]_\sigma\}$.*

Example 6. Consider $c = (\texttt{this.next} := \texttt{b.next})$ and its denotation d. The execution of c from the state σ_1 in Figure 2 is the trace $d(\sigma_1) = \sigma_1 \to \sigma_2$ where $\sigma_2 = \phi_1 \star [l_1 \mapsto o_1, l_2 \mapsto \texttt{myclass}\star[\texttt{i} \mapsto 4, \texttt{next} \mapsto l_3], l_3 \mapsto o_3]$. Then $\mathsf{assigned}(\langle l_2, \texttt{next}\rangle, \sigma_1, \sigma_2)$ since the field `next` of `this` has been updated. Since c changes only this handle, we have $d \in \gamma_{\tau,\tau}(\{\texttt{this.next}\})$.This does not mean that `this.next` is the *only* field which can be modified by c, but that it *covers* all the handles whose values can be modified by c. For instance, the execution of c from a state where `b` and `this` are aliases also modifies `b.next`, but it is an alias of `this.next`.

Aliasing allows different paths to have the same handle. Hence there is not always a best (canonical) $P \in \wp(\mathit{Paths}')$ which approximates a given command.

Example 7. Let $c = (\texttt{if this} = \texttt{b then this.i} := 3)$. Every trace in the denotation d of c can change only a location pointed by both `this` and `b`. Hence $d \in \gamma_{\tau,\tau}(\{\texttt{this.i}\})$ and $d \in \gamma_{\tau,\tau}(\{\texttt{b.i}\})$. There is no motivation for choosing `this.i` instead of `b.i` as the property we are looking for.

Thus some commands do not have a *best* approximation in $\wp(\mathit{Paths}')$. To solve this problem, we consider $\wp\wp(\mathit{Paths}')$ as the property we are looking for.

Definition 16. *Let $\mathcal{A}_{\tau,\tau'} = \wp\wp(\mathit{Paths}'_\tau)$ ordered as $A_1 \subseteq A_2$ if for every $P_2 \in A_2$ there is $P_1 \in A_1$ such that $P_1 \subseteq P_2$. Moreover, let $\gamma_{\tau,\tau'} : \mathcal{A}_{\tau,\tau'} \mapsto \wp(D_{\tau,\tau'})$ be such that $\gamma_{\tau,\tau'}(A) = \cap\{\gamma_{\tau,\tau'}(P) \mid P \in A\}$. So $\gamma_{\tau,\tau'}(A)$ contains the (denotations of) those commands that, for every $P \in A$, modify only handles in P.*

The $\subseteq$ ordering in Definition 16 is just a preorder. Hence we implicitly consider an element of $\mathcal{A}$ as standing for its $\subseteq$ equivalence class.

Example 8. Let a command stand for its denotation. In Example 6 we have $c \in \gamma_{\tau,\tau}(\{\{\texttt{this.next}\}\})$. In Example 7 we have $c \in \gamma_{\tau,\tau}(\{\{\texttt{this.i}\}, \{\texttt{b.i}\}\})$.

Proposition 1. *The map $\gamma_{\tau,\tau'} : \mathcal{A}_{\tau,\tau'} \mapsto \wp(D_{\tau,\tau'})$ of Definition 16 is the concretisation map of a Galois connection from $\wp(D_{\tau,\tau'})$ to $\mathcal{A}_{\tau,\tau'}$.*

$\mathcal{A}$ is theoretically interesting since it is *the* reference for comparing static analyses for `assignable` clauses. But it is useless for a real analysis, since it induces a very imprecise abstract composition of commands, as we show now.

Example 9. Consider the method `update` of the class `myclass` in Figure 1. Let α be the abstraction map induced by the map γ of Definition 16. The abstraction $\alpha(\texttt{this.next} := \texttt{b}(\texttt{.next})^i)$ does not depend on $i \geq 0$, since only and always `this.next` is modified. Let us denote the optimal abstract counterpart of $\otimes$ by $\otimes$ itself. We would like that $\alpha(\texttt{this.next} := \texttt{b.next.next}) \otimes \alpha(\texttt{b.next.next} := \texttt{b})$ correctly approximates $A = \alpha(\texttt{this.next} := \texttt{b.next.next}; \texttt{b.next.next} := \texttt{b}) = \{\{\texttt{this.next}, \texttt{b.next.next}, \texttt{b.next.next.next}\}\}$ (Section 1). We have

$$\begin{aligned}
&\alpha(\texttt{this.next} := \texttt{b.next.next}) \otimes \alpha(\texttt{b.next.next} := \texttt{b})\\
&= \alpha(\texttt{this.next} := \texttt{b}(\texttt{.next})^i) \otimes \alpha(\texttt{b.next.next} := \texttt{b})\\
(*) &\supseteq \alpha(\texttt{this.next} := \texttt{b}(\texttt{.next}^i); \texttt{b.next.next} := \texttt{b})\\
(**) &\supseteq \{\{\texttt{this.next}, \texttt{b}(\texttt{.next})^{i+1}\}\}
\end{aligned}$$

for every $i \geq 0$. Point $*$ follows by the correctness of the abstract $\otimes$. Point $**$ by considering a starting state where `b` and `this` are bound to the same arbitrarily long list of distinct objects (see the pictures in Section 1). Hence $\alpha(\texttt{this.next} := \texttt{b.next.next}) \otimes \alpha(\texttt{b.next.next} := \texttt{b}) \supseteq \{\{\texttt{this.next}\} \cup \{\texttt{b}(\texttt{.next})^{i+1} \mid i \geq 0\}\}$, a correct but very imprecise approximation of A.

The problem in Example 9 is that $\mathcal{A}$ says which fields have been modified, but it does not say anything about the variables nor about *what has been put inside* those fields. This information is vital for a precise abstract $\otimes$. For instance, if in Example 9 we knew that `this.next` has been modified with `b.next.next`, we could conclude that $\texttt{b.next.next} := \texttt{b}$ can only modify the fields `b.next.next` and `b.next.next.next` (when `this` and `b` are aliases). From another perspective, we can say that Example 9 shows that in some cases the property $\mathcal{A}$ is too weak to allow for its modular verification. It is not easy to tell if this means that $\mathcal{A}$ (*i.e.,* the `assignable` specifications that JML currently provides) are not powerful enough for their modular verification. A practical evaluation of the precision of $\mathcal{A}$ is needed here. Anyway, we want to solve the problem shown by Example 9. Hence, in Section 5, we consider a *refinement* $\mathcal{AR}$ of $\mathcal{A}$ which contains the information that $\mathcal{A}$ is missing.

5 The Refined Domain $\mathcal{AR}$

We add to $\mathcal{A}$ information about how each variable and field of the last state of a trace t can be *covered* (Definition 18) by the values of some paths in the first state of t. This allows us to define a precise abstract $\otimes$ (Definition 22).

Definition 17. *Let $\overline{\tau} = \cup\{F(\kappa) \mid \kappa \in \mathcal{K}\}$ be the type environment of all the fields. This definition is sensible if we assume that fields are distinguished through their fully-qualified name. We define the domain $\mathcal{COV}_{\tau,\tau'}$ of* coverings *as*

$$\mathcal{COV}_{\tau,\tau'} = \{E\star M \mid E : \mathsf{dom}(\tau') \mapsto \wp\wp(Paths_\tau),\ M : \mathsf{dom}(\overline{\tau}) \mapsto \wp\wp(Paths_\tau)\}$$

and the refinement $\mathcal{AR}_{\tau,\tau'} = \mathcal{A}_{\tau,\tau'} \times \mathcal{COV}_{\tau,\tau'}$ *ordered by pointwise $\subseteq$ (Definition 16).*

Example 10. In Figure 1 we have $\overline{\tau} = [\texttt{i} \mapsto \mathit{int}, \texttt{j} \mapsto \mathit{int}, \texttt{next} \mapsto \texttt{myclass}]$. Let $\tau = [\texttt{b} \mapsto \texttt{myclass}, \texttt{out} \mapsto \texttt{myclass}, \texttt{this} \mapsto \texttt{myclass}]$. An element of $\mathcal{COV}_{\tau,\tau}$ is

$$C = \begin{bmatrix} \texttt{b} \mapsto \{\{\texttt{b}\}\},\ \texttt{out} \mapsto \{\{\texttt{b.next}\}\} \\ \texttt{this} \mapsto \{\{\texttt{this}\}\} \end{bmatrix} \star \begin{bmatrix} \texttt{i} \mapsto \{\varnothing\},\ \texttt{j} \mapsto \{\varnothing\} \\ \texttt{next} \mapsto \{\{\texttt{b.next.next}\}\} \end{bmatrix} .$$

We want C to mean that, at the end of the execution, `b` and `this` did not change their binding *or* are bound to fresh locations (the result of a `new` command). The variable `out`, instead, at the end of the execution must be bound to the location `b.next` was bound to at its beginning, *or* to a fresh location. Moreover, at the end of the execution, the fields `next` of *every* object did not change their binding *or* are bound to fresh locations *or* to the location `b.next.next` was bound to at the beginning of the execution. We formalise this idea now.

Definition 18. *Let $A \in \wp\wp(Paths_\tau)$, $\sigma \in \Sigma_\tau$ and $l \in Loc$. We say that A* covers *l in σ if, whenever l is reachable in σ from $Paths_\tau$ (Definition 12), l is reachable in σ from every $P \in A$.*

Example 11. Let σ_1 be the state in Figure 2. Then $A = \{\{\texttt{this}\}, \{\texttt{b.next.next}, \texttt{b}\}\}$ covers l_2 in σ_1 since l_2 is reachable in σ_1 from `this` and `b.next.next`. Let $l_4 \in Loc$ be such that $l_4 \neq l_i$ for $i = 1, 2, 3$ (a *fresh* variable *w.r.t.* σ_1). Then A covers l_4 in σ_1 since l_4 is not reachable in σ_1 from $Paths_\tau$. Similarly, $\{\varnothing\}$ covers l_4 in σ_1.

We extend Definition 18 to $E\star M \in \mathcal{COV}$, which says how variables and fields are covered. Variables and fields are treated asymmetrically, since a variable v stands for a single value, while a field f stands for every field f in all objects. Then, we require that v is covered by E while f is covered by M *or* has not changed.

Definition 19. *We say that $E\star M \in \mathcal{COV}_{\tau,\tau'}$* covers *$\sigma' \in \Sigma_{\tau'}$ in $\sigma \in \Sigma_\tau$ if*

- *$\forall v \in Paths_\tau$ such that $l = \nu(\llbracket v \rrbracket_{\sigma'}, \sigma') \in Loc$, $E(v)$ covers l in σ;*
- *$\forall p.f \in Paths_\tau$ such that $l = \nu(\llbracket p.f \rrbracket_{\sigma'}, \sigma') \in Loc$, $M(f)$ covers l in σ or if $l' = \nu(\llbracket p \rrbracket_{\sigma'}, \sigma')$ is reachable in σ then $\nu(\langle l', f\rangle, \sigma) = \nu(\langle l', f\rangle, \sigma')$.*

Definition 20. *We define $\gamma_{\tau,\tau'} : \mathcal{COV}_{\tau,\tau'} \mapsto \wp(D_{\tau,\tau'})$ as*

$$\gamma_{\tau,\tau'}(E\star M) = \left\{ d \in D_{\tau,\tau'} \,\middle|\, \begin{array}{l} \forall \sigma \in \Sigma_\tau \text{ s.t. } \neg\,\mathsf{div}(d(\sigma)) \\ E\star M \text{ covers } \mathsf{lst}(d(\sigma)) \text{ in } \sigma \end{array} \right\}$$

and $\gamma_{\tau,\tau'} : \mathcal{AR}_{\tau,\tau'} \mapsto \wp(D_{\tau,\tau'})$ as $\gamma_{\tau,\tau'}(a\star E\star M) = \gamma_{\tau,\tau'}(a) \cap \gamma_{\tau,\tau'}(E\star M)$.

Example 12. Let $c_1 = (\texttt{this.next} := \texttt{b.next.next})$, $c_2 = (\texttt{b.next.next} := \texttt{b})$, $c_3 = (\texttt{out} := \texttt{b.next})$ and $c_4 = (\texttt{this.i} := \texttt{this.i} + \texttt{1})$ be the four commands of the method `update` of `myclass` in Figure 1. They are approximated, respectively, by (*i.e.,* their denotations belong to γ of)

$$\{\{\texttt{this.next}\}\}\star \begin{bmatrix} \texttt{b} \mapsto \{\{\texttt{b}\}\},\ \texttt{out} \mapsto \{\{\texttt{out}\}\} \\ \texttt{this} \mapsto \{\{\texttt{this}\}\} \end{bmatrix} \star \begin{bmatrix} \texttt{i} \mapsto \{\varnothing\},\ \texttt{j} \mapsto \{\varnothing\} \\ \texttt{next} \mapsto \{\{\texttt{b.next.next}\}\} \end{bmatrix}$$

$$\{\{\texttt{b.next.next}\}\}\star \begin{bmatrix} \texttt{b} \mapsto \{\{\texttt{b}\}\},\ \texttt{out} \mapsto \{\{\texttt{out}\}\} \\ \texttt{this} \mapsto \{\{\texttt{this}\}\} \end{bmatrix} \star \begin{bmatrix} \texttt{i} \mapsto \{\varnothing\},\ \texttt{j} \mapsto \{\varnothing\} \\ \texttt{next} \mapsto \{\{\texttt{b}\}\} \end{bmatrix}$$

$$\{\varnothing\}\star \begin{bmatrix} \texttt{b} \mapsto \{\{\texttt{b}\}\},\ \texttt{out} \mapsto \{\{\texttt{b.next}\}\} \\ \texttt{this} \mapsto \{\{\texttt{this}\}\} \end{bmatrix} \star \begin{bmatrix} \texttt{i} \mapsto \{\varnothing\},\ \texttt{j} \mapsto \{\varnothing\} \\ \texttt{next} \mapsto \{\varnothing\} \end{bmatrix}$$

$$\{\{\texttt{this.i}\}\}\star \begin{bmatrix} \texttt{b} \mapsto \{\{\texttt{b}\}\},\ \texttt{out} \mapsto \{\{\texttt{out}\}\} \\ \texttt{this} \mapsto \{\{\texttt{this}\}\} \end{bmatrix} \star \begin{bmatrix} \texttt{i} \mapsto \{\varnothing\},\ \texttt{j} \mapsto \{\varnothing\} \\ \texttt{next} \mapsto \{\varnothing\} \end{bmatrix} .$$

Proposition 2. *The map $\gamma_{\tau,\tau'} : \mathcal{AR}_{\tau,\tau'} \mapsto \wp(D_{\tau,\tau'})$ of Definition 20 is the concretisation map of a Galois connection from $\wp(D_{\tau,\tau'})$ to $\mathcal{AR}_{\tau,\tau'}$.*

We could define now the approximation of every bytecode defined in [12,11] and of the sequential and disjunctive composition of bytecodes $\otimes$ and $\oplus$. However, since we plan to use our analysis for high-level source codes, we prefer to approximate every high-level constructs in Definition 2. Hence the next section explains how approximations like those in Example 12 can be automatically constructed for the commands and expressions in Definition 2.

6 The Analysis

We discuss here a static analysis which uses the domain $\mathcal{AR}$ *i.e.,* we explain how to mechanically construct the approximation of the denotations of single commands (like those in Example 12) and how to combine them into an approximation of their sequential ($\otimes$) or disjunctive ($\oplus$) composition.

The analysis we are going to define always uses a singleton set of sets of paths to represent how some value can be covered. For instance, for Example 7 it computes $\{\{\texttt{this.i}, \texttt{b.i}\}\}$ instead of the more precise information $\{\{\texttt{this.i}\}, \{\texttt{b.i}\}\}$. Hence, we simplify the notation from now on, by removing one level of bracketing. Note that this does not contradict our reasonings in Section 4. We just observe here, *a posteriori*, that some theoretically possible precision does not come out from the analysis. This does not mean that the property we are looking for (*i.e.,* $\mathcal{A}$) must be defined differently.

Assume that a value is covered by a path p in σ (Definition 18). How can that value be covered if we do some computation, covered by some $E\star M \in \mathcal{COV}$, before σ? To answer this question, we use the following operation $\bullet$.

Definition 21. *Let $p \in Paths_\tau$ and $E\star M \in \mathcal{COV}_{\tau,\tau'}$. We define the* update *$(E\star M) \bullet p$ of p w.r.t. $E\star M$ as*

$$\begin{aligned}(E\star M) \bullet v &= E(v)\\ (E\star M) \bullet p.f &= \{p'.f \mid p' \in (E\star M) \bullet p\} \cup M(f)\ .\end{aligned}$$

This operation is pointwise extended to sets and then to functions into sets.

Example 13. Let $E\star M$ be the approximation of s_1 given in Example 12 (remember that we forget about a level of bracketing now). We have

$$\begin{aligned}(E\star M) \bullet \texttt{b.next} &= \{p'.\texttt{next} \mid p' \in (E\star M) \bullet \texttt{b}\} \cup \{\texttt{b.next.next}\}\\ &= \{p'.\texttt{next} \mid p' \in \{\texttt{b}\}\} \cup \{\texttt{b.next.next}\}\\ &= \{\texttt{b.next}, \texttt{b.next.next}\}\ .\end{aligned}$$

We can now define the abstract sequential and disjunctive composition.

Definition 22. *Let $a_1\star E_1\star M_1 \in \mathcal{AR}_{\tau,\tau'}$ and $a_2\star E_2\star M_2 \in \mathcal{AR}_{\tau',\tau''}$. We define $(a_1\star E_1\star M_1) \otimes (a_2\star E_2\star M_2) \in \mathcal{AR}_{\tau,\tau''}$ as*

$$(a_1 \cup ((E_1\star M_1) \bullet a_2))\star((E_1\star M_1) \bullet E_2)\star(M_1 \cup ((E_1\star M_1) \bullet M_2))\ .$$

Example 14. Let $a_i\star E_i\star M_i$ be the denotation of the command c_i in Example 12, for $i = 1, 2, 3, 4$. An approximation of $c_1; c_2$ is $(a_1\star E_1\star M_1) \otimes (a_2\star E_2\star M_2)$ *i.e.,*

$$X = \left\{\begin{array}{l}\texttt{this.next}\\ \texttt{b.next.next}\\ \texttt{b.next.next.next}\end{array}\right\} \star \left[\begin{array}{l}\texttt{b} \mapsto \{\texttt{b}\}\\ \texttt{out} \mapsto \{\texttt{out}\}\\ \texttt{this} \mapsto \{\texttt{this}\}\end{array}\right] \star \left[\begin{array}{l}\texttt{i} \mapsto \varnothing\\ \texttt{j} \mapsto \varnothing\\ \texttt{next} \mapsto \{\texttt{b}, \texttt{b.next.next}\}\end{array}\right].$$

An approximation of $c_1; c_2; c_3$ is $X \otimes (a_3 \star E_3 \star M_3)$ *i.e.,*

$$Y = \left\{\begin{array}{l}\texttt{this.next}\\ \texttt{b.next.next}\\ \texttt{b.next.next.next}\end{array}\right\} \star \left[\begin{array}{l}\texttt{b} \mapsto \{\texttt{b}\}\\ \texttt{out} \mapsto \left\{\begin{array}{l}\texttt{b, b.next}\\ \texttt{b.next.next}\end{array}\right\}\\ \texttt{this} \mapsto \{\texttt{this}\}\end{array}\right] \star \left[\begin{array}{l}\texttt{i} \mapsto \varnothing\\ \texttt{j} \mapsto \varnothing\\ \texttt{next} \mapsto \{\texttt{b}, \texttt{b.next.next}\}\end{array}\right] .$$

An approximation of $c_1; c_2; c_3; c_4$ is $Y \otimes (a_4 \star E_4 \star M_4)$ *i.e.,*

$$\underbrace{\left\{\begin{array}{l}\texttt{this.i}\\ \texttt{this.next}\\ \texttt{b.next.next}\\ \texttt{b.next.next.next}\end{array}\right\}}_{A} \star \left[\begin{array}{l}\texttt{b} \mapsto \{\texttt{b}\}\\ \texttt{out} \mapsto \left\{\begin{array}{l}\texttt{b, b.next}\\ \texttt{b.next.next}\end{array}\right\}\\ \texttt{this} \mapsto \{\texttt{this}\}\end{array}\right] \star \left[\begin{array}{l}\texttt{i} \mapsto \varnothing\\ \texttt{j} \mapsto \varnothing\\ \texttt{next} \mapsto \{\texttt{b}, \texttt{b.next.next}\}\end{array}\right] .$$

The set A in Example 14 shows that the domain $\mathcal{AR}$ does not suffer of the imprecision problem shown for $\mathcal{A}$ (Example 9). Namely, our analysis computes the correct `assignable` A clause for the method shown in the introduction. As we already said, the Chase tool gives an incorrect answer for that method.

Definition 23. *Let* $a_1 \star E_1 \star M_1$ *and* $a_2 \star E_2 \star M_2$ *in* $\mathcal{AR}_{\tau,\tau'}$. *We define* $(a_1 \star E_1 \star M_1) \oplus (a_2 \star E_2 \star M_2) \in \mathcal{AR}_{\tau,\tau'}$ *as (the operation* $\cup$ *is applied pointwise to functions here)*

$$(a_1 \cup a_2) \star (E_1 \cup E_2) \star (M_1 \cup M_2) .$$

Example 15. Example 14 shows an approximation of the method `update` of the class `myclass` in Figure 1. An approximation of the method `update` of the class `subclass` can be computed similarly. It is

$$\{\texttt{this.i}\} \star \left[\begin{array}{l}\texttt{b} \mapsto \{\texttt{b}\},\ \texttt{out} \mapsto \{\texttt{this}\}\\ \texttt{this} \mapsto \{\texttt{this}\}\end{array}\right] \star \left[\begin{array}{l}\texttt{i} \mapsto \varnothing,\ \texttt{j} \mapsto \varnothing\\ \texttt{next} \mapsto \varnothing\end{array}\right] .$$

If we want an approximation of a call to the method `update`, but we do not know which of the two alternatives the late binding mechanism will choose, we can use $\oplus$ over the approximation of the two alternatives and obtain

$$\left\{\begin{array}{l}\texttt{this.i}\\ \texttt{this.next}\\ \texttt{b.next.next}\\ \texttt{b.next.next.next}\end{array}\right\} \star \left[\begin{array}{l}\texttt{b} \mapsto \{\texttt{b}\},\ \texttt{this} \mapsto \{\texttt{this}\}\\ \texttt{out} \mapsto \left\{\begin{array}{l}\texttt{this}\\ \texttt{b, b.next}\\ \texttt{b.next.next}\end{array}\right\}\end{array}\right] \star \left[\begin{array}{l}\texttt{i} \mapsto \varnothing\\ \texttt{j} \mapsto \varnothing\\ \texttt{next} \mapsto \{\texttt{b}, \texttt{b.next.next}\}\end{array}\right] .$$

We can now define an approximation of expressions and commands. Recall that the special variable *res* holds the value of an expression (Section 3) and that method call is an expression in our language. We first define the *empty* covering $E^\perp \star M^\perp$. It expresses the fact that no variable or field has changed. We also need to auxiliary maps that remove the result (*res*) of an expression and store a covering set into a variable, respectively.

Definition 24. *Let $v \in \mathsf{dom}(\tau)$, $f \in \mathsf{dom}(\overline{\tau})$, $a{\star}E{\star}M \in \mathcal{AR}$ and*

$$E^{\perp}(v) = \begin{cases} \varnothing & \text{if } \tau(v) = int \\ \{v\} & \text{otherwise} \end{cases} \qquad M^{\perp}(f) = \varnothing$$

$$\llcorner a{\star}E{\star}M \lrcorner = a{\star}E|_{-res}{\star}M \qquad (a{\star}E{\star}M)^{r}_{v} = a{\star}E[v \mapsto r]{\star}M \ .$$

We define ι_τ as (τ will be usually omitted)

$$\begin{aligned}
\iota(i) = \iota(\mathtt{nil}) = \iota(\mathtt{new}\ \kappa) &= (\varnothing{\star}E^{\perp}{\star}M^{\perp})^{\varnothing}_{res} \\
\iota(v) &= (\varnothing{\star}E^{\perp}{\star}M^{\perp})^{\{v\}}_{res} \\
\iota(e_1\ bop\ e_2) &= \llcorner \iota(e_1) \lrcorner \otimes \iota(e_2) \\
\iota(\mathtt{is\_nil}(e)) &= \iota(e)^{\varnothing}_{res} \\
\iota(e.f) &= \iota(e)^{\{p.f \mid p \in E(res)\} \cup M(f)}_{res} \quad \text{with } \iota(e) = a{\star}E{\star}M \\
\iota(e.m(v_1,\ldots,v_n)) &= \left(A \otimes \left(\oplus \left\{ d_{m'} \,\middle|\, \begin{array}{l} m'(w_1:t_1,\ldots,w_n:t_n):t \\ \text{can be called here and} \\ a{\star}E{\star}M \text{ is its denotation} \end{array} \right\} \right) \right) \\
\text{where } A &= \llcorner \iota(e)^{E'(res)}_{\mathtt{this}} \lrcorner \quad \text{with } \iota(e) = a' \cdot E' \cdot M' \\
\text{and } d_{m'} &= (a{\star}E^{\perp}{\star}M)^{E(\mathtt{out})}_{res}[w_1 \mapsto v_1,\ldots,w_n \mapsto v_n] \ .
\end{aligned}$$

The above renaming operation at method call substitutes actual arguments for formal ones. The information about the covering for the value of e flows inside the method call through the `this` variable. At the end, the variable *res* takes the covering information of `out`, hence the result of the method is available as the value of the method call expression. We use $E^{\perp}$ there, so that changes to the parameters of the method are not observable outside it.

The denotation considered for method call can be that computed at the previous iteration step or one provided by the user. The first choice can be used to actually *compute* `assignable` clauses for a program through an iteration to the fixpoint, while the second one can be used for *checking* that given `assignable` clauses are correct. In this second case it is enough to check that the new denotation is $\subseteq$ (Definition 17) than that given by the user. In such a case, the user has provided a post-fixpoint of the abstract immediate consequence operator induced by ι, and post-fixpoints are a correct approximation of the minimal fixpoint [14]. Note that, in this case, the user must specify an `assignable` clause (an element of $\mathcal{A}$) together with the extra *covering* information (Definition 17)!

Example 16. Consider the expression `b.next.next` from the method `update` of the class `myclass` in Figure 1. We have

$$a{\star}E{\star}M = \iota(\mathtt{b}) = \varnothing{\star} \begin{bmatrix} \mathtt{b} \mapsto \{\mathtt{b}\},\ \mathtt{out} \mapsto \{\mathtt{out}\} \\ res \mapsto \{\mathtt{b}\},\ \mathtt{this} \mapsto \{\mathtt{this}\} \end{bmatrix} {\star} \begin{bmatrix} \mathtt{i} \mapsto \varnothing,\ \mathtt{j} \mapsto \varnothing \\ \mathtt{next} \mapsto \varnothing \end{bmatrix} .$$

Hence

$$a'{\star}E'{\star}M' = \iota(\mathtt{b.next}) = \iota(\mathtt{b})^{\{p.\mathtt{next} | p \in E(res)\} \cup M(\mathtt{next})}$$
$$= \varnothing{\star}\begin{bmatrix} \mathtt{b} \mapsto \{\mathtt{b}\},\ \mathtt{out} \mapsto \{\mathtt{out}\} \\ res \mapsto \{\mathtt{b.next}\},\ \mathtt{this} \mapsto \{\mathtt{this}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{i} \mapsto \varnothing,\ \mathtt{j} \mapsto \varnothing \\ \mathtt{next} \mapsto \varnothing \end{bmatrix}$$

and $\iota(\mathtt{b.next.next})$ is

$$\iota(\mathtt{b.next})^{\{p.\mathtt{next} | p \in E'(res)\} \cup M'(\mathtt{next})}$$
$$= \varnothing{\star}\begin{bmatrix} \mathtt{b} \mapsto \{\mathtt{b}\},\ \mathtt{out} \mapsto \{\mathtt{out}\} \\ res \mapsto \{\mathtt{b.next.next}\},\ \mathtt{this} \mapsto \{\mathtt{this}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{i} \mapsto \varnothing,\ \mathtt{j} \mapsto \varnothing \\ \mathtt{next} \mapsto \varnothing \end{bmatrix} .$$

We define the approximation of the execution of a command now.

Definition 25. *We define*

$$\begin{aligned}
\iota(v{:=}e) &= \llcorner \iota(e)_v^{E(res)} \lrcorner \quad \textit{with } \iota(e) = a{\star}E{\star}M \\
\iota(e_1.f{:=}e_2) &= \llcorner \iota(e_1) \otimes (a \cup \{res.f\}) \cdot E \cdot M[f \mapsto M(f) \cup E(res)] \lrcorner \\
&\quad \textit{with } a{\star}E{\star}M = \iota(e_2) \\
\iota(c_1; c_2) &= \iota(c_1) \otimes \iota(c_2) \\
\iota(\mathtt{let}\ v{:}t\ \mathtt{in}\ c) &= \iota_{\tau[v \mapsto t]}(c) \quad \textit{where every path } v \textit{ or } v.p \textit{ is removed} \\
\iota(\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2) &= \llcorner \iota(e) \lrcorner \otimes (\iota(c_1) \oplus \iota(c_2)) \ .
\end{aligned}$$

Example 17. Example 16 shows an approximation of `b.next.next`. An approximation of `this` is

$$\iota(\mathtt{this}) = \varnothing{\star}\begin{bmatrix} \mathtt{b} \mapsto \{\mathtt{b}\},\ \mathtt{out} \mapsto \{\mathtt{out}\} \\ res \mapsto \{\mathtt{this}\},\ \mathtt{this} \mapsto \{\mathtt{this}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{i} \mapsto \varnothing,\ \mathtt{j} \mapsto \varnothing \\ \mathtt{next} \mapsto \varnothing \end{bmatrix} .$$

Hence $\iota(\mathtt{this.next} := \mathtt{b.next.next})$ is

$$\llcorner \iota(\mathtt{this}) \otimes \{res.\mathtt{next}\} \cdot \begin{bmatrix} \mathtt{b} \mapsto \{\mathtt{b}\},\ \mathtt{out} \mapsto \{\mathtt{out}\} \\ res \mapsto \{\mathtt{b.next.next}\} \\ \mathtt{this} \mapsto \{\mathtt{this}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{i} \mapsto \varnothing,\ \mathtt{j} \mapsto \varnothing \\ \mathtt{next} \mapsto \{\mathtt{b.next.next}\} \end{bmatrix} \lrcorner$$
$$= \{\mathtt{this.next}\}{\star}\begin{bmatrix} \mathtt{b} \mapsto \{\mathtt{b}\},\ \mathtt{out} \mapsto \{\mathtt{out}\} \\ \mathtt{this} \mapsto \{\mathtt{this}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{i} \mapsto \varnothing,\ \mathtt{j} \mapsto \varnothing \\ \mathtt{next} \mapsto \{\mathtt{b.next.next}\} \end{bmatrix} .$$

Note that this is consistent with Example 12, where the same result were obtained by following our intuition about the abstract domain.

Proposition 3. *The map ι of Definitions 24 and 25 correctly approximates the concrete denotation of expressions and commands.*

Since paths are potentially infinite, if we want to *compute* `assignable` clauses for a program we must *cut* the paths to a maximum length. Longer paths can be approximated by introducing the JML `reach` clause.

7 Conclusion

We have formalised the semantics of the `assignable` clauses of the specification language JML as an abstract interpretation $\mathcal{A}$ of trace semantics. We have shown that a static analysis based on $\mathcal{A}$ can only be very imprecise, since $\mathcal{A}$ lacks information useful for the definition of a precise sequential composition operator. For the same reason, modular verification over $\mathcal{A}$ seems impractical. Therefore we have refined $\mathcal{A}$ into a more precise property $\mathcal{AR}$ which does not suffer of the same problem. We have then defined a static analysis to check, in a modular way, the correctness of $\mathcal{AR}$ annotations.

To the best of our knowledge, this is the first correct static analysis to check or compute JML `assignable` clauses. We have shown that it works correctly in some cases for which the Chase tool fails. Although the analysis has not been implemented yet, we have described in every detail its algorithmic definition (Section 6).

As pointed out at the end of Section 4, the problem with $\mathcal{A}$ is that it says which fields may be modified, but not what may be assigned to these fields. Note that $\mathcal{A}$ is similar to the assignable clauses in JML in this respect! The refinement $\mathcal{AR}$ remedies the problem by keeping track of the additional "covering" information *i.e.,* of *what* may be assigned to fields. This means that modular checking for $\mathcal{AR}$ does require the user to supply the extra covering information in addition to the assignable clauses. Note that this suggests that assignable clauses as currently available in JML may be fundamentally unsuited for a good – *i.e.,* accurate and correct – static analysis. Indeed, it seems that the last word on best way to specify side effects, by assignable clauses or other means, has not been said, e.g. see [3], [7], or [10].

References

1. N. Cataño and M. Huisman. A Static Checker for JML's `assignable` Clause. Available from `www-sop.inria.fr/lemme/Nestor.Catano/`, 2002.
2. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
3. A. Greenhouse and J. Boyland. An Object-Oriented Effects System. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, number 1628 in LNCS, pages 205–229. Springer-Verlag, 1999.
4. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
5. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical report, Dept. of Comp. Sci., Iowa State University, 1999. Tech. Rep. 98-06.
6. K.R.M. Leino. Data Groups: Specifying the Modification of Extended State. In *OOPSLA'98*, pages 144–153. ACM, 1998.
7. K.R.M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using Data Groups to Specify and Check Side Effects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37(5), pages 246–257, June 2002.

8. The LOOP Project. `www.cs.kun.nl/∼bart/LOOP/index.html`.
9. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2^{nd} rev. edition, 1997.
10. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular Specification of Frame Properties in JML. *Concurrency and Computation: Practice and Experience*, 2002. To appear.
11. F. Spoto. Watchpoint Semantics: A Tool for Compositional and Focussed Static Analyses. In P. Cousot, editor, *Proc. of the Static Analysis Symposium, SAS'01*, volume 2126 of *LNCS*, pages 127–145, Paris, July 2001.
12. F. Spoto and T. Jensen. Focused Class Analyses through Abstract Interpretation of Trace Semantics. Available from `www.sci.univr.it/∼spoto/papers.html`.
13. F. Spoto and E. Poll. Static Analysis for JML's `assignable` Clauses. Extended version. Available from `www.sci.univr.it/∼spoto/papers.html`.
14. A. Tarski. A Lattice-theoretical Fixpoint Theorem and its Applications. *Pacific J. Math.*, 5:285–309, 1955.