Chase: a Static Checker for JML's *Assignable* Clause

Néstor Cataño and Marieke Huisman

{Nestor.Catano, Marieke.Huisman}@sophia.inria.fr
INRIA Sophia-Antipolis, France

Abstract. This paper presents a syntactic method to check so-called *assignable* clauses of annotated JAVA programs. Assignable clauses describe which variables may be assigned by a method. Their correctness is crucial for reasoning about class specifications. The method that we propose is incomplete, as it only makes a syntactic check and it does not take aliasing or expression evaluation into account, but it provides efficient means to find the most common errors in assignable clauses. This is demonstrated by applying the method to the specification of an industrial case study.

1 Introduction

Recently, several lightweight approaches to formal methods have demonstrated that the use of formal methods in practice is actually feasible. Up to now, formal methods merely were considered as a theoretical issue, which could not be applied to "real" problems: the complicated logical notations and the minimal tool support could not compete with the software development methods and tools used in industry. But recently, specification languages have been proposed that closer resemble programming languages, and powerful tools have been developed which are tailored to these specification languages, while at the same time, industry has started to feel the need for certified software, in particular after the discovery of bugs such as in the INTEL PENTIUM II chips.

Traditional techniques for program verification are model checking and theorem proving. Drawbacks of both are that they require a good insight in the particular verification techniques and tools and that they are often time and memory consuming. Recent tools try to overcome these drawbacks, by using efficient checking techniques that work directly on the level of the program and its specification. Two of such techniques are run-time checking and static checking. These are what could be called lightweight formal techniques, which do not result in a full correctness proof, but which are able to find the most common errors efficiently. Such techniques are useful to increase the trust in the correctness of an implementation *w.r.t.* a specification.

Run-time checking is a technique where safety conditions are checked while running the program, *e.g.* when making a method call it is checked that the precondition of the method is not violated for this particular call, otherwise an

error message is returned. Typically, run-time checking is implemented by transforming the program into one that explicitly executes all these tests. Examples of run-time checkers are the EIFFEL-compiler [16] and JML [9] and JASS [1] (both working on JAVA programs). JML is not only a run-time checker, but it also provides a full-fledged specification language for JAVA.

Complementary to run-time checking is static checking, where one tries to find problems in a program by applying standard program verification techniques automatically. Static checking can be considered as an extended form of type checking, because it finds potential run-time problems already at compile-time. Typically, static checking gives quick feedback on a program and can find many common errors, so more expensive formal verification techniques can concentrate on the essential parts of the program. An example of a static checker for JAVA is ESC/JAVA (EXTENDED STATIC CHECKER/JAVA) [12], which generates proof obligations from an annotated program and sends these proof obligations to an automatic, dedicated theorem prover. ESC/JAVA is especially tuned to find errors such as indexing an array out of bounds, null pointer dereferencing, deadlocks and race conditions, but it also can check arbitrary functional specifications (given in a simplified version of the JML specification language). If it cannot establish a certain specification, it issues a warning. As ESC/JAVA is neither sound nor complete, a warning does not necessarily mean that the program is incorrect, and when it produces no warnings this does not necessarily mean that the program is correct.

The checking of methods in ESC/JAVA is done in a modular way, on a method per method basis. When one encounters a method call in the body that one is checking, it is not known which method implementation actually will be used – due to dynamic binding. Therefore, following the behavioral subtype approach [13], the method specification in the static class type of the receiver of the method call is used[1]. When reasoning in such a way, pre- and postconditions of methods alone are not sufficient, one also needs to know which variables may have been changed by a method. Consider for example the following annotated program fragment.

```
public class C {
  int[] arr;

  //@ ensures arr.length >= 4;
  public void m(){arr = new int[5]; n();}

  //@ ensures true;
  public void n(){arr = new int[2];}
}
```

To establish whether `m` satisfies its postcondition, the functional specification of `n` alone does not give enough information, one also needs to know whether `n`

[1] Additionally, one has to show that for each overriding method in a subclass, it satisfies the specification of the superclass.

might change the variable `arr`. Therefore, JML and ESC/JAVA allow the user to specify a so-called assignable clause – using the keyword `modifies`[2] – specifying a set of locations that may be modified by a method. An assignable clause can contain variable names, but also more complicated expressions, *e.g.* denoting all the elements in an array or all the fields of an object. By default, when a method specification does not contain an assignable clause, the method is supposed to leave all variables unchanged.

The current version of ESC/JAVA does not check assignable clauses [12, 11], it only uses them when checking method calls. As specifying assignable clauses is error-prone and the information in the assignable clause is crucial in the verification of other methods, we think this is a significant omission. When specifying a real-life application it is easy to forget a variable in the assignable clause – see for example our experiences with the specification and static checking of an industrial case study [5] in Sect. 5.2. Therefore, in this paper we propose a method for static checking of assignable clauses (implemented in our tool CHASE).

As ESC/JAVA, our method is neither sound nor complete, since it does not handle aliasing and it does not keep track of variable updates (see Sect. 4.3). However, our experiments show that in practice our method is very useful, because it finds immediately the most common mistakes in an assignable clause: simply forgetting to mention that a variable may be modified. We feel that when specifying a method in which variables are modified in a complicated way, a specifier will be careful when writing the assignable clause and will be more willing to do full verification. However, for simple methods it is easy to forget to mention a variable and preferably this is detected without using full verification.

Related work The term *frame condition* finds its origins in the field of Artificial Intelligence [15]. Borgida *et al.* [3] revealed the relationship with formal specifications, where one needs to know which part of the state may be modified. They discuss several approaches, and finally propose to use so-called change axioms which can describe the possible changes of the state under a certain condition. They also give a syntactic method which can be used to generate a first attempt for the change axioms. A limitation of their approach is that they only look at single variables, while we can also handle expressions describing a set of memory locations.

Recently, Poll and Spoto [18] have proposed a correct static analysis method for assignable clauses. However, their method is not supported by a tool.

Leino [10] and Müller [17] both studied how to specify and verify assignable clauses in subclasses and/or using abstract variables. In this paper we only look at assignable clauses containing concrete variables. When extending our work to abstract variables, their work would form the basis to deal with dependencies.

Within the JASS project [8] the non-violation of assignable clauses can be checked at run-time, but this is restricted to simple variables only, more complex expressions are not allowed in the assignable clauses.

[2] JML also allows the alternative keywords `assignable` and `modifiable`.

Finally, within the LOOP project [14, 2] a *semantic* definition for assignable clauses is defined. When verifying a method body, one has to show that each variable not mentioned in the assignable clause is unchanged. This is a heavy proof burden, because it involves quantifications over the whole state. Therefore, it is useful to first check automatically whether the assignable clause is likely to be correct, before diving into the full verification.

Organization The rest of this paper is organized as follows. Section 2 briefly presents the most important assertion constructs of JML and ESC/JAVA. Section 3 discusses the meaning of the assignable clause. Section 4 presents the rules to check assignable clauses, and Sect. 5 presents the implementation of CHASE on top of the JML parser and discusses practical experiences. Finally, Sect. 6 gives conclusions and presents future work.

2 JML and ESC/Java Specifications

The JML specification language is designed to be easily usable for JAVA programmers. In particular, JML expressions are side-effect-free boolean JAVA expressions, extended with specification-specific constructs. The specification language for ESC/JAVA uses the same design principles. Initially there were several differences between the two languages, but to enable the use of different tools on the same specification, effort has been put in making the two languages converge [6]. In this paper, we do not really distinguish between the two languages[3], although we take the more general JML format for assignable clauses.

We briefly present the main JML and ESC/JAVA constructs, followed by a method specification example. A complete description can be found in [9, 12]. Method behaviors can be specified using preconditions (`requires P`), postconditions (`ensures Q`), and exceptional postconditions (`exsures (E) R`), describing under which conditions a certain exception can occur. Also frame conditions (`modifies L`) can be part of a method specification, denoting which modifications may be made by a method (see Sect. 3). The behavior of a class can be restricted by an `invariant` clause.

There are several specification-specific constructs, *e.g.* logical implication `==>`, universal and existential quantifiers ("`forall T V; E` and "`exists T V; E`), `\old(E)`, to denote the value of expression `E` in pre-state of method, and `\result` to denote a method return value.

As an example of a method specification, we give the specification of a method `addCurrency` from an electronic purse case study [5]. This specification has been checked with ESC/JAVA.

```
/*@ modifies nb, data[nb];
    ensures (\old(nb) >= MAX_DATA) ? (nb == \old(nb)) :
                (nb == \old(nb) + 1 && data[\old(nb)] == cur); */
void addCurrency(byte cur){if(nb<MAX_DATA){data[nb]=cur; nb++;}}
```

[3] In particular, we do not consider JML's model variables.

```
<Assignable-exp> ::= <Identifier>
                   | <Identifier>[Exp]
                   | <Identifier>[Exp. . . Exp]
                   | <Identifier>[*]
                   | \fields_of(<Field-exp>)
                   | \nothing
                   | \everything

     <Field-exp> ::= <Point-identif>
                   | this
                   | \reach(<Point-identif>)
                   | \reach(this)

<Point-identif> ::= <Identifier>.<Point-identif>
                   | <Identifier>
```

Fig. 1. Syntax for assignable expressions

The method `addCurrency` belongs to a class `Currencies`, storing all currencies supported by this purse application. It adds a new currency to the list of valid currencies (the array `data`). The `modifies` clause specifies that this method may modify only `nb` and `data` in the position `nb`. The postcondition of the method `addCurrency` expresses that if `nb` has not yet reached the threshold value `MAX_DATA`, `nb` increases its value by one and the value of the formal parameter `cur` is assigned to `data[\old(nb)]`, otherwise `nb` is unchanged. In the postcondition, the expression `\old(nb)` refers to the value of `nb` before the method invocation.

3 JML's Assignable Clauses

An assignable clause specifies which variables may be modified by a method; all other variables *should* remain unchanged. Within an assignable clause, a list of so-called assignable expressions is specified, describing which memory locations may be modified.

An assignable expression can be an identifier or an array indexing expression denoting a single location, but it can also denote a set of locations. An expression `a[i...j]` denotes the set of locations $\{\texttt{a}[k] \mid \texttt{i} \leq k \leq \texttt{j}\}$, while `a[*]` denotes the same set as `a[0...a.length-1]`. An assignable expression `\fields_of(`e`)` denotes the set of *all fields* of the objects represented by the expression e. The expression e can be a single object (possibly `this`), an array, or the `\reach` of an object or array f (again possibly `this`). The fields of an array are its elements. The expression `\reach(`f`)` denotes the minimal set containing f, the fields of f and all objects reachable from the fields of f. Finally, there are special keywords `\nothing` and `\everything`, denoting the empty set of locations and the full set of locations, respectively. Figure 1 lists the complete grammar of assignable expressions.

Given the set of locations characterized by the assignable expressions, the semantics of a JML assignable clause is defined as follows[4].

Definition 1 (Assignable clause). *An* assignable clause *only allows a method to modify a location* loc *if:*

- loc *is mentioned in the method's assignable clause;*
- loc *is not allocated when the method starts execution; or*
- loc *is local to the method (*i.e.*, a local variable or a formal parameter).*

Notice that this is a syntactic definition, thus an assignment `x = x;` only is permitted if the variable `x` is mentioned in the appropriate assignable clause or if `x` is a local variable.

For a normal, non-overriding, method the assignable clause is exactly the assignable clause as given in the method specification. When overriding a method in a subclass, JML and ESC/JAVA allow one only to extend the assignable clause of the overridden method[5]. Therefore, the *complete assignable clause* of a method that is overriding a method in a superclass is the union of its specified assignable clause and the complete assignable clause of the method it is overriding. To reduce the possibilities of unsoundness in modular verification, the assignable clause in the subclass preferably only adds variables (or expressions) that are not visible in the scope of the overridden method [12, 10]. Finally, for a constructor, its body may freely modify (the reach of) all instance fields of the newly created object – as they were not allocated before method execution. Therefore, for a constructor body one has to check whether it obeys the union of the assignable clause as specified and the assignable expression `\fields_of(\reach(this))`.

4 A Syntactical Method to Check Assignable Clauses

This section presents a syntactic method to decide for each method whether it violates its complete assignable clause. For every method we check its assignable clause by checking for every assignment and for every method call encountered in the body, whether it agrees with the assignable clause of the method that is checked. For each language construct s we define a rule which allows us to derive $s \,\mathsf{mod}\, A$, with the intuitive meaning: statement s only assigns to variables which are mentioned in assignable clause A. Together, these rules define mod. In some cases we need to lift mod over a set of instructions, defined as:

$$S \,\overrightarrow{\mathsf{mod}}\, A \mathrel{\hat{=}} \forall s.\, s \in S \Rightarrow s \,\mathsf{mod}\, A.$$

As mentioned in the previous section, the assignable clause can contain expressions as `\fields_of(this)`, thus checking syntactically whether a variable

[4] This definition differs from [9] in that it does not consider dependencies, as we do not consider model variables.

[5] Both in JML and in ESC/JAVA such method specifications can be recognized by the keyword `also`.

<Full-exp> ::= <Full-exp>[<Exp>$^+$]
| <Full-exp>(<Exp>*)
| <Full-exp>.<Suf-exp>
| <Suf-exp>

<Suf-exp> ::= <Identifier>
| `this`
| `super`
| Literal
| `new` <Type>[<Exp>]
| `new` <Type>(<Exp>*)

Fig. 2. Syntax for full expressions and suffix expressions

occurs in the assignable clause cannot be done by using the standard set membership $\in$. Therefore, we use an extended notion of set membership, subsuming the standard $\in$, denoted by $\Subset$. Below, in Sect. 4.2, this notion is defined formally. Intuitively, $e \Subset A$ means that the expression e appears literally in A, or A contains an expression such as `\fields_of(`f`)`, and the set that is described by the expression `\fields_of(`f`)` contains e.

We lift the notion of extended set membership to extended subsets:

$$A_1 \sqsubseteq A_2 \mathrel{\hat{=}} \forall v.\ v \Subset A_1 \Rightarrow v \Subset A_2.$$

We also use variant notions modFE and modSuf, with a similar intuitive meaning, but defined only on so-called *full expressions* and *suffix expressions*, respectively, whose syntax is defined in Fig. 2. A full expression is an array indexing expression, a method call, a qualified expression or a suffix expression. A suffix expression can be an identifier, a reference to `super` or `this`, a literal or an initialization of an object or array. This syntax is a simplified version of the JML and JAVA grammar [9, 7], because we assume that the program is accepted by the JAVA compiler and/or the JML tool. We assume that we have predicates full? and suffix? to decide whether an expression is a full expression or a suffix expression, respectively.

4.1 The Rules Defining mod, modFE and modSuf

For each language construct, appropriate rules are given to define mod, modFE and modSuf in a compositional way. In several cases, the JAVA and JML syntax prescribe that a certain subexpression of an arbitrary expression is a so-called full expression or that a subexpression of a full expression is a suffix expression. Therefore, mod is defined in terms of modFE, and modFE is defined in terms of modSuf.

Method declaration rules The first rule that we define deals with method and constructor declarations. For each method and constructor we check that the complete assignable clause is respected by its body – assuming that the assignable clauses for all methods called within this body are correct. Following Def. 1, it is sufficient to check that a body only modifies the formal parameters or the variables in the complete assignable clause A.

$$\text{(Meth-Dec)}\quad \frac{body\ \mathsf{mod}\ (A \cup \{\overrightarrow{par}\})}{m(\overrightarrow{par})\{body\}\ \mathsf{mod}\ A}$$

Local variable declarations Note that in the rule for method declarations the method's local variables are not added to the assignable clause. This is done because local variables are only visible in the program text below their declaration. For example, a method fragment {`x = 3; int x; x = 4;`} actually changes variable `x` (assuming that `x` is *e.g.* an instance variable) to 3. To handle this appropriately, we decompose statement compositions into a single statement and a composed statement, and we have a special rule for when this single statement is a local variable declaration (of some type T). In this case we add the local variable to the assignable clause of the composed statement. In addition, we also check that the initialization expression e (if present) does not have unwanted side-effects.

$$\text{(Var-Decl)}\quad \frac{e \bmod A \quad t \bmod (A \cup \texttt{x})}{T\ \texttt{x = } e\texttt{;}t \bmod A}$$

Assignment rules For assignments $e_1 \texttt{ = } e_2$ we must check that e_1 belongs to the set of locations that the method may modify ($e_1 \subseteqq A$). Also we must check that e_1 and e_2 do not have unwanted side-effects. Following the JAVA syntax, we know that e_1 must be a full expression, thus we check e_1 modFE A.

$$\text{(Assg)}\quad \frac{e_1 \subseteqq A \quad e_1 \text{ modFE } A \quad e_2 \bmod A}{e_1 \texttt{ = } e_2 \bmod A}$$

This rule generalizes to all JAVA's assignment operators (`+=`, `*=` *etc.*) and to JAVA's pre- and postfix operators (*e.g.* e`++` and `++`e) in the obvious way.

Statement rules For all JAVA statements, *e.g.* $s\texttt{;}t$, `if(`c`){`s`}` and `while(`c`){`s`}` the rules pass on the check for side-effects to the components of the statement. Remember that statement compositions $s\texttt{;}t$ we decompose so that s is a single statement, and if s is a local variable declaration the rule Var-Decl above applies.

$$\text{(Comp)}\quad \frac{s \bmod A \quad t \bmod A}{s\texttt{;}t \bmod A}\ s \neq T\ \texttt{x = } e \qquad \text{(If-Then)}\quad \frac{c \bmod A \quad s \bmod A}{\texttt{if(}c\texttt{)\{}s\texttt{\}} \bmod A}$$

Expression rules For most expressions the same applies as for statements: the rules pass on the check for side-effects to the arguments of the expressions. We present the rule BinOp, which applies to all binary operators. Similar rules are defined for *e.g.* the unary operators, the `instanceof` expression, the casting operator, and the conditional expression.

$$\text{(BinOp)}\ \frac{e_1 \bmod A \quad e_2 \bmod A}{e_1 \oplus e_2 \bmod A}\ \oplus \in \left\{ \begin{array}{l} \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}, \texttt{||}, \\ \texttt{\&\&}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\textbackslash}, \texttt{\&}, \texttt{\^{}}, \texttt{|} \end{array} \right\}$$

If none of these rules apply to an expression, it means that the expression is a full expression and we should check for side-effects using modFE.

$$\text{(To-Fe)}\quad \frac{e \text{ modFE } A}{e \bmod A}\ \text{full?}(e)$$

Full expression rules Following the grammar for full expressions, we define modFE. Full expressions can be method invocations, array indexing expressions, qualified expressions or suffix expressions.

The assignable clause of a method invocation should be an extended subset of the assignable clause A of the method body that is currently checked – with appropriate substitutions – to ensure that no other variables than the ones mentioned in A are changed. A method invocation is of the form $fe(\overrightarrow{par})$ where fe is a full expression again. Given that the expression is accepted by the JAVA compiler, fe is a qualified expression <Full-exp>.<Suf-exp>, or it is a suffix expression only. The suffix expression must be a method name, `this` or `super`. The appropriate assignable clause can be found by looking at the static type of the receiving object of the method call (possibly `this`). This is sufficient, provided that additional assignable clauses in overriding methods do not name locations that are visible in the scope of the overridden method [12, 10]. Additionally it should be checked that the full expression fe and the actual parameters do not violate the assignable clause A.

$$\text{(Meth-Inv)}\ \frac{fe(\overrightarrow{par})\text{.assignable}\,[\overrightarrow{act}/\overrightarrow{par}, fe/\texttt{this}] \sqsubseteq A \quad fe\ \text{modFE}\ A \quad \overrightarrow{act}\ \overrightarrow{\text{mod}}\ A}{fe(\overrightarrow{act})\ \text{modFE}\ A}$$

Otherwise, if the full expression is not a method invocation, it must be an array indexing expression, a qualified expression or a suffix expression. Both for array indexing expressions and for qualified expressions, the rules pass on the check for side-effects to the components of the expression. In case the full expression is a suffix expression, modSuf is used to check for side-effects.

$$\text{(Array)}\ \frac{fe\ \text{modFE}\ A \quad \overrightarrow{e}\ \overrightarrow{\text{mod}}\ A}{fe\,[\overrightarrow{e}]\ \text{modFE}\ A} \qquad \text{(Qualified)}\ \frac{fe\ \text{modFE}\ A \quad s\ \text{modSuf}\ A}{fe.s\ \text{modFE}\ A}$$

$$\text{(To-Suf)}\ \frac{fe\ \text{modSuf}\ A}{fe\ \text{modFE}\ A}\ \text{suffix?}(fe)$$

Suffix expression rules Most suffix expressions are constant values (*e.g.* `this` and all the literals) which do not have side-effects, thus the rules for these constructs are straightforward. Also the access to an identifier does not modify the state – if the identifier is the target of an assignment, the rule Assg takes care that it is mentioned in the assignable clause. Thus, the only suffix expressions which are of interest are the `new` expressions. Here, the possible arguments have to be checked for unwanted side-effects. The creation of a new object or array does not violate the rules for the assignable clause (*cf.* Def. 1).

$$\text{(This)}\ \frac{\text{true}}{\texttt{this}\ \text{mod}\ A} \qquad \text{(New-Exp)}\ \frac{\overrightarrow{e}\ \overrightarrow{\text{mod}}\ A}{\texttt{new T(}\overrightarrow{e}\texttt{)}\ \text{modSuf}\ A}$$

4.2 The Extended Membership Relation

Finally, we define the extended membership relation. For all assignable clause expressions we give appropriate syntactic rules. First we consider the assignable

expressions `\nothing` and `\everything`. In the first case the assignable clause is the empty set, in the second case it is the complete set of locations. The rules for these constructs are straightforward.

$$\text{(In-Nothing)}\ \frac{\mathsf{false}}{e \in \texttt{\textbackslash nothing}} \qquad \text{(In-Everything)}\ \frac{\mathsf{true}}{e \in \texttt{\textbackslash everything}}$$

For non-trivial assignable clauses, the basic case is where a variable or array index expression is mentioned literally[6]. In this case, we fall back directly on the standard definition of set membership to check whether an assignment target occurs in A. Here we give the rules for unqualified expressions, we have similar rules for qualified expressions.

$$\text{(In-Var)}\ \frac{\texttt{x} \in A}{\texttt{x} \in A} \qquad \text{(In-Arr)}\ \frac{\texttt{a[}e\texttt{]} \in A}{\texttt{a[}e\texttt{]} \in A}$$

Also expressions `a[*]` or `a[i..j]` can occur in A. These expressions allow array index expressions to be the assignment target[7]. Again, there are similar rules for qualified expressions.

$$\text{(Global-Arr)}\ \frac{\texttt{a[*]} \in A}{\texttt{a[}e\texttt{]} \in A} \qquad \text{(Interv-Arr)}\ \frac{\texttt{a[i..j]} \in A \quad \texttt{i} \le e \le \texttt{j}}{\texttt{a[}e\texttt{]} \in A}$$

Finally, expressions of the form `\fields_of(`e`)` might occur in A. Now we have to distinguish between two cases: e is a single object (possibly `this`) or array[8], or it is a `\reach` expression. In the first case, any field of the object e or any array index expression of the array e is allowed as assignment target.

$$\text{(In-Fld-Exp-Var)}\ \frac{\texttt{\textbackslash fields\_of(}e\texttt{)} \in A}{e\texttt{.x} \in A} \qquad \text{(In-Fld-Arr)}\ \frac{\texttt{\textbackslash fields\_of(a)} \in A}{\texttt{a[}e'\texttt{]} \in A}$$

When e is a reach expression `\reach(`e'`)`, any variable access or array index expression is allowed as assignment target for which the receiving object (possibly `this`) or the array is an element of the set determined by the `\reach` expression.

$$\text{(In-Reach-Exp)}\ \frac{\texttt{\textbackslash fields\_of(\textbackslash reach(}e'\texttt{))} \in A \quad f \in \texttt{\textbackslash reach(}e'\texttt{)}}{f\texttt{.x} \in A}$$

$$\text{(In-Reach-Arr)}\ \frac{\texttt{\textbackslash fields\_of(\textbackslash reach(}e'\texttt{))} \in A \quad \texttt{a} \in \texttt{\textbackslash reach(}e'\texttt{)}}{\texttt{a[}f\texttt{]} \in A}$$

As explained in Sect. 3, the expression `\reach(`e'`)` denotes the minimal set containing e', the fields of e' and all objects reachable from the fields of e'. To

[6] We do not distinguish between `this.x` and `x`.
[7] Provided that `i` $\le e \le$ `j` can be checked syntactically, which is not always the case.
[8] For arrays, `\fields_of(`e`)` is equivalent to e`[*]` [9].

determine whether an object or array is part of `\reach(`e'`)`, we state the two following rules: an element e is in the reach of an object, if it is the object itself (Reach-Base), or if it is in the reach of one of the fields of this object (Reach-Rec).

$$\text{(Reach-Base)}\ \frac{e = e'}{e \in \texttt{\textbackslash reach(}e'\texttt{)}}$$

$$\text{(Reach-Rec)}\ \frac{e'' \in \texttt{\textbackslash fields\_of(}e'\texttt{)} \quad e \in \texttt{\textbackslash reach(}e''\texttt{)}}{e \in \texttt{\textbackslash reach(}e'\texttt{)}}$$

4.3 Limitations of the Method

Using our method, many specification mistakes in assignable clauses can be found (see the next section for experimental results). Nevertheless it has important limitations, because it works purely on a syntactical basis.

Firstly, our method does not handle aliasing. When two references point to the same object and a field of such an object is changed via one reference, the tool does not require that the assignable clause also reflects the changes via the other reference. For example, given a class `O` with an integer field `i`, we do not detect any problem in the specification of method `p` – although implicitly `y.i` also will be changed within `p`.

```
public class C {
  O y = new O();
  O x = new O();

  //@ modifies x, x.i;
  public void p(){x = y; x.i = 7;}
}
```

Secondly, our method does not take earlier variable updates into account. For example, for a program fragment {`i++; a[i] = 3;`}, it accepts an assignable clause containing the expression `a[i]` (and it rejects `a[i+1]`). Similarly, it will reject changes to the fields of a newly allocated object.

Overall, we think that these limitations do not severely restrict the usability of our method. Our method should be considered as a quick check to get a reasonable trust in the correctness of the specified assignable clauses. In our experience, when specifying the assignable clause of a method, it is more likely that one forgets a variable than to overlook complicated modification structures. Finding these small specification mistakes before doing formal verification significantly can help to improve the verification speed. In particular, our method is a good purity checker, because it always issue a warning if a method can have side-effects, but has an empty assignable clause.

5 Chase: a Checker for Assignable Expressions

As mentioned above, we have implemented our method in a tool called Chase, which is freely available [4]. This section presents its implementation and dis-

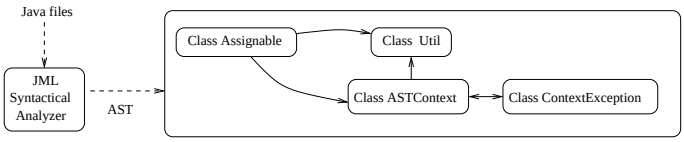

Fig. 3. Overall structure of CHASE

cusses our experiences using CHASE on an earlier specification case study [5]. When applying CHASE to this case study, it found several incorrect assignable clauses. Further, CHASE enables us to determine the set of side-effect free methods of this application.

5.1 The Implementation of Chase

CHASE is constructed by reusing the syntactical analyzer of JML [9] and by implementing – as static methods – the rules presented in Sect. 4. Figure 3 presents the overall structure of CHASE. The input for CHASE is an annotated JAVA file. First the JML syntactical analyzer constructs the corresponding abstract syntax tree (AST) for this file. The class `Util` then transforms the AST to make it usable for the class `Assignable`, which implements the rules presented in Sect. 4, in methods `_mod`, `_modFE`, *etc.*. Each method is parameterized by an instruction and a context. The context of an instruction is the set of locations it may modify: the union of its parameters and its assignable locations. Class `Util` also calculates these contexts, such that whenever it cannot establish the context of an instruction, it throws a `ContextException`.

As an example, the rule (Assg) is implemented as a fragment of method `_mod`.

```
public static boolean _mod(AST e, ASTContext currentContext)
    throws assignable.exception.ContextException
    { if(e.getType() == JavaTokenTypes.ASSIGN)
          return in_PRIME(e.getFirstChild(),currentContext) &&
                 _modFE(e.getFirstChild(),currentContext) &&
                 _mod(e.getFirstChild().getNextSibling(),
                      currentContext);
  ... // other cases
}
```

Whenever `e` is a `JavaTokenTypes.ASSIGN` expression, CHASE checks that its left expression appears in the list of assignable locations, (method `in_PRIME`) and recursively calls the methods `_modFE` and `_mod`, with the left and right trees of `e`, respectively.

When CHASE detects a possible violation of an assignable clause, it issues a warning containing the AST of the instruction that might violate the assignable clause.

5.2 Experiences using Chase

In an earlier case study [5] we specified a JavaCard purse application, describing class invariants and the functional behavior (pre- and postconditions) and frame conditions of each method. We used Esc/Java to check the specifications, but naturally we could not check the assignable clauses. Therefore, we applied Chase to this case study. We checked the 3 packages of the purse application, containing 27 classes and ± 800 methods, finding 43 incorrect assignable clauses (see the Chase web site [4] for a full compendium).

We present one example of a missing assignable clause. The method `setValue` sets the fields of the class `Decimal`, which represents a decimal number by means of an integer part, `intPart`, and a decimal part, `decPart`. Initially, our assignable clause only included the variables `intPart` and `decPart`, which are directly assigned in the body of the method.

```
//@ modifies intPart, decPart;
public Decimal setValue(short i, short d) throws DecimalException{
  if(i<0 || d<0 || d>=PRECISION || (i==MAX_DEC_NUMBER && d!=0))
    decimal_exception.throwIt(decimal_exception.DEC_OVERFLOW);
  intPart = i; decPart = d; return this;
}
```

When Chase is applied to this method, it issues a warning that the expression `decimal_exception.throwIt(decimal_exception.DEC_OVERFLOW)` may cause a problem. Inspection of the method `throwIt` reveals that the instance variables `decimal_exception.instance` and `decimal_exception.instance.type` also may be modified, and thus that these must be mentioned in the assignable clause of `setValue`.

6 Conclusion and Future Work

This paper presents a method to do an efficient check on assignable clauses. It has been implemented in a tool called Chase, which is freely available [4]. In particular, Chase can be used to do a quick check on side-effect-freeness.

The method works on a syntactic basis: for each Java construct a rule is defined which checks that every assignment or method call only modifies variables that are declared as *assignable*. This approach is neither sound nor complete, but experiences have shown that it is useful in practice: it efficiently finds common specification mistakes – such as forgetting to mention a variable that may be modified. Before completely verifying an annotated Java program, it is good practice to use Chase to check the assignable clauses, so that simple specification mistakes do not disturb the verification process.

There are limitations to our approach, in particular it does not work in a context with aliasing. In these cases, full formal verification is necessary. However, as said, applying our method first, one finds the simpler mistakes quickly, thus allowing to concentrate on the essentials when verifying.

As future work, we plan to improve the tool to overcome (at least partly) the limitations of the tool, *e.g.* better handling of updates and newly allocated memory, and returning a warning if a possibly aliased variable has changed.

It is also future work to extend the tool to appropriately handle so-called model variables, which allow to define specification-only variables and relate them to concrete variables by using `represents` or `depends` clauses, following [10].

References

1. D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In K. Havelund and G. Roşu, editors, *ENTCS*, volume 55(2). Elsevier Publishing, 2001.
2. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, number 2031 in LNCS, pages 299–312. Springer, 2001.
3. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.
4. N. Cataño and M. Huisman. Assignable specifications for the Electronic Purse case study, 2002. `http://www-sop.inria.fr/lemme/verificard/modifSpec`.
5. N. Cataño and M. Huisman. Formal specification and static checking of Gemplus's electronic purse using ESC/Java. In L.-H. Eriksson and P.A. Lindsay, editors, *Formal Methods Europe (FME'02)*, number 2391 in LNCS, pages 272–289. Springer, 2002.
6. Differences between Esc/Java and JML, 2000. Comes with JML distribution, in file `esc-jml-diffs.txt`.
7. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.
8. The JASS project. `http://semantik.informatik.uni-oldenburg.de/~jass/`.
9. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary Design of JML: a Behavioral Interface Specification Language for Java. Technical Report 98-06, Iowa State University, Department of Computer Science, 1998. `http://www.cs.iastate.edu/~leavens/JML/prelimdesign/`.
10. K.R.M. Leino. Data groups: specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, pages 144–153. ACM Press, 1998.
11. K.R.M. Leino. Applications of Extended Static Checking. In P. Cousot, editor, *Static Analysis (SAS 2001)*, number 2126 in LNCS, pages 185–193. Springer, 2001.
12. K.R.M. Leino, G. Nelson, and J.B. Saxe. ESC/Java user's manual. Technical Report SRC 2000-002, Compaq System Research Center, 2000.
13. B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Trans. on Progr. Lang. and Systems*, 16(1):1811–1841, 1994.
14. The LOOP project. `http://www.cs.kun.nl/~bart/LOOP/`.
15. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh Univ. Press, 1969.
16. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd rev. edition, 1997.

17. P. Müller. *Modular Specification and Verification of Object Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001.
18. E. Poll and F. Spoto. Static analysis for JML's assignable clauses, 2002. Manuscript.