More Flexible Object Invariants with Less Specification Overhead

Stefan Huster, Patrick Heckeler, Hanno Eichelberger, Jürgen Ruf, Sebastian Burg, Thomas Kropf, and Wolfgang Rosenstiel

University of Tübingen, Department of Computer Science, Sand 14, 72076 Tübingen, Germany
{huster,weissensel,heckeler,eichelberger,ruf,burg, kropf,rosenstiel}@informatik.uni-tuebingen.de

Abstract. Object invariants are used to specify valid object states. They play a central role for reasoning about the correctness of object-oriented software. Current verification methodologies require additional specifications to support the flexibility of modern object oriented programming concepts. This increases the specification effort and represents a new source of error. The presented methodology reduces the currently required specification overhead. It is based on an automatic control flow analysis between code positions violating invariants and code positions requiring their validity. This analysis helps to prevent specification errors, possible in other approaches. Furthermore, the presented methodology distinguishes between valid and invalid invariants within one object. This allows a (more) flexible definition of invariants.

Keywords: Object Invariants, Dependency Analysis, Reduced Specification Overhead.

1 Introduction

Invariants specify relations on the program's data, which are expected to hold during the program execution. Besides other contract types, e.g. pre- and postconditions, invariants play a central role for reasoning about the correctness of object-oriented software [1–3]. It is generally accepted when pre- and postconditions must be valid. Preconditions must be valid at call time of a method and postconditions at a methods completion. But there exists no unique definition regarding the scope of an invariant, which specifies when an invariant must be valid. Different approaches exist regarding the definition of invariants and the supported scope. To handle the flexibility of object oriented concepts, current approaches introduce and require additional specifications. These additional specifications are used to define explicitly when an invariant or object must be valid or which methods are allowed to invalidate an invariant. But they also cause additional specification overhead and represent a new source of error.

This paper describes a new methodology to generate proof obligations for invariants. Figure 1 illustrates how the presented methodology is embedded in a

D. Giannakopoulou and G. Salaün (Eds.): SEFM 2014, LNCS 8702, pp. 302–316, 2014.

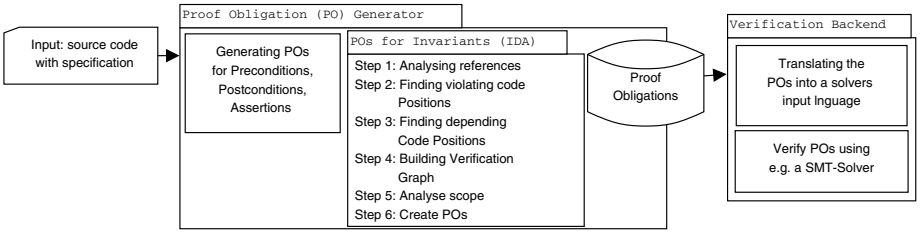

Fig. 1. Process flow diagram of our methodology

verification process. It is based on a static code analysis consisting of six steps. We begin in Step 1 to analyse all references of each invariant. In Step 2 we search code positions modifying the referenced values. These positions may invalidate the corresponding invariant. In Step 3 we search code positions depending on the validity of an invariant. In Step 4 we analyse backwards the possible call stack of each found code position and build a special call graph, called *Verification Graph*. In Step 5 we use the generated Verification Graph to analyse when invariants are invalidated and when they need to be re-established. We call this the *Scope* of an invariant. This information is used in Step 6 to generate proof obligations, ensuring the validity of an invariant whenever it is expected to hold. Therefore they are very similar to a Hoare-Triple [4]

The remainder of this paper is structured as follows: In Section 1.1 we introduce current methodologies for specifying and verifying invariants and detail their limitations. The contributions of the presented methodologies are listed in Section 1.2. A formal description of our methodology is given in Section 2. In Section 3 we show how this approach is applied to several examples. In Section 4 we conclude and introduce possible future work.

1.1 Related Work and Current Limitations

The classical Visible State Technique (VST) [5], as used in Eiffel [6] or Java Extended Static Checking [7], is the most restricted methodology verifying invariants. In this concept, invariants are allowed to reference only class fields of the same object. Each invariant must be valid in every public state. Therefore, one must show that each invariant of an object is valid before and after any public exported method has been executed. The limitations of the VST are well analysed in the literature [5, 8]. Due to its strictness the VST does not require any additional specifications. But it is also not flexible enough to support constructs like recursive methods, inheritance or invariants referring to values of multiple classes (multiclass invariants).

The ownership technique (OST) [9, 10, 5, 11] structures the set of objects in a acyclic hierarchical graph. Each object has at most one owner, which defines one context, containing all its (transitively) owned objects. Objects without any owner are part of the global context. The ownership model permits an object to

reference directly owned objects (*rep* references) and objects in the same context (*peer* references). Invariants may reference class fields of its own class and of all (transitively) owned objects. A method is allowed to violate the invariants of all objects within the ancestor context of its receiver object. Therefore, the ownership model allows the modification of an object only by methods of its owner. The OST introduces a new notation to specify the ownership relations between object references. Hierarchical references must be declared as *rep*-references. References to sibling objects must be marked as *peer*-reference. To verify an invariant, one must show that each exported method preserves the invariants of its receiver object. Besides the additional required specification effort, the OST represents a very strict verification model. This model does not support the verification of recursive data structures and limits the possible cooperation between different objects. Furthermore, the ownership technique prohibits invariants of two different objects to contain a reference to one shared instance.

Barnett et al. extends the ownership technique in [12], by introducing a friendship system (FSS). This system allows the specification of invariants beyond ownership boundaries. Friendship relations control the access to privately owned fields. This allows other classes to build their invariants on it. This is realised by two new specification statements *friend* and *read*. Another extension is introduced by Barnett et al. in [14]. We refer to this approach as Explicit State Technique (EST). In that methodology, whether or not an object invariant is known to hold is expressed within the objects state in a special class field, not accessible by normal program code. Therefore, objects have to be marked explicitly as "invalid", before their class fields are updated. This is done by two special statements: *unpack* and *pack*. The first one marks an object as invalid and opens a frame in which the object state may be changed. The second statement closes the frame and enforces all invariants to hold again. This extension is used by Leino et al. in [2] to express invariants in dynamic contexts. Their approach uses an additional *dependent* and *ownerdependent* clausal to mark recursive dependencies. Furthermore, it is the first approach able to reason separately about object invariants declared in different subclasses.

The visibility technique (VIS) is introduced by Müller et al. in [5] and is based on the ownership model. A declaration within a specific class is visible in a method, if the methods module imports the module of that class. An extended approach considering visibility modifiers is presented in [15]. Invariants may reference class fields from their own class and all classes, in which that invariant is visible. Therefore, an invariant might be violated only by methods in which the invariant is visible. To prove that an invariant holds, one must show that a method preserves the invariants of all its referenced objects. As a result, the visibility technique is powerful enough to define and handle among others specifications of recursive data structures. Because the VIS is based on the ownership model it supports only very strict invariants regarding the ownership hierarchy. The VIS requires that each invariant of all objects, relevant for a method's execution, hold before the corresponding method is called. Therefore, the VIS cannot be used to verify a gradual update on a not owned object.

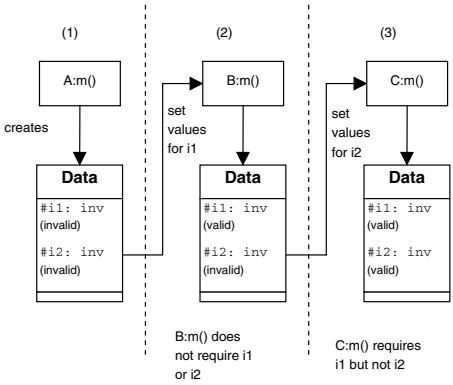

Fig. 2. Illustration of an inter-object based on partly valid objects

The Oval approach (OVL) [13] combines the ownership model and behavioural contracts. These contracts are defined by the additional specification sets *validity invariants* and *validity effects*: The first set contains objects which must be valid before and after a method is executed. The second set enumerates objects which might be violated during the execution of that method. Both sets are used to track which objects must be re-established and validated after a method was executed. This is very similar to the presented methodology which also is based on two sets, listing invariants expected to be valid and invariants which might be invalidated. However, we use a static code analysis to determine these sets automatically rather than requiring their definition manually by the programmer. Furthermore, the Oval approach allows only invariants based on its own fields or the fields of its (transitively) owned objects. As well as the pure ownership and the visibility technique, Oval cannot distinguish between valid and invalid invariants within one object. Additionally to the overhead caused by the underlying ownership model, the Oval approach requires a high specification overhead for defining the behavioural contracts. Consequently, a manual enumeration of required and effected invariants has to be performed for each method.

1.2 Contributions

Current approaches have two main limitations: (1) They require a high specification overhead to define object invariants. (2) They cannot distinguish between valid and invalid invariants within one object.

The first limitation has an additional drawback. The programmer must learn the semantic and syntax of these new specifications. Furthermore, they represent an additional source of error. This is because the programmer may define the expected scope of an invariant not correctly. This may cause the unrecognised invalidation of depending post- or preconditions, as shown in Section 3. The second limitation prevent current approaches to be used to verify method calls on partly valid objects, depending on a subset of invariants, as illustrated in Figure 2. In this example, the Data-object is invalid until the last method $C : m()$ establishes its valid state. But the operation $C : m()$ requires the validity of

invariant i_1, which has been established just one step before by the method $B : m()$.

The presented methodology addresses these limitations and has following contributions to the state of the art: **(1)** Reduced specification overhead, by using an automatic dependency analysis based on existing access modifiers **(2)** More flexibility, by distinguishing between valid and invalid invariants within one object.

2 Methodology

The presented methodology generates proof obligations ensuring the validity of invariants. The process of generating proof obligations uses static code analysis to analyse when invariants are expected to hold. Altogether, the presented methodology combines six different steps, as illustrated in Figure 1. These steps are applied for each invariant.

Step 1: Analysing References. To analyse when an invariant might be violated we need to know what values are referenced by a given invariant i. The set of class fields referenced by any statement s or invariant i is specified by the set $Ref^*(s)$. It combines directly ($Ref(s)$) and transitively referenced class fields. A direct reference is any direct access to a class field f_c in class c, by calling $o.f$ on an instance o of class c. Transitively referenced fields are accessed through method calls in s, e.g. by calling a get-method. The set of methods called in s is denoted by $CalledMethods(s)$.

$$Ref^*(s) = Ref(s) \cup \{Ref(m) \,|\, m \in CalledMethods(s)\} \tag{1}$$

The set of class fields, directly modified by a given statement s, is returned by the function $Ref!(s)$.

Step 2: Finding Invariant Violations. A statement might invalidate an invariant i, if it assigns a new value to any class field f_c, referenced in i. We call such statement s *Violating Code Position* and its method $m|s \in m$ *Violating Method*. Violating code positions are searched in each method body $S_m \in M_c$ of all methods M_c of each class $c \in C$ within the set of all classes C. The set of violating code positions for one invariant i is defined by:

$$ViolatingS(i) = \forall c \in C\, \forall m \in M_c\, \forall s \in S_m\, \{s \,|Ref!(s) \cap Ref^*(i) \neq \emptyset\} \tag{2}$$

For reasons of simplicity, we assume that every class field f has private accessibility. Therefore, direct access to a field f_c is possible only within its declaring class c. However, this assumption causes no limitations, because we make no further assumptions regarding the definition of set- and get-methods.

Step 3: Finding Depending Code Positions. To analyse the scope of an invariant i we must know when i might be violated and when i is expected to

be valid. A code position s is called *Depending Code Position* if it requires the validity of an invariant. The corresponding method $m|s \in m$ is called *Depending Method*. Depending code positions are searched in each method body $S_m \in M_c$ of each class c. A statement s accessing a field f, referred by the invariant i, depends on i in two cases: (1) If s has no (transitive) access to every $f_i \in Ref^*(i)$ (2) If there exists a different proof obligation ϱ, e.g. caused by a postcondition, containing s: $s \in S_\varrho$.

The first case is based on the idea that an invariant might be invalid, as long as its validity can be checked by the programmer. But this is only possible if all fields, referenced by an invariant are accessible. In this case we say *s can check i*. If an invariant cannot be checked, a code block may invalidate an invariant or access an invalid object. This is very similar to the task of checking manually the validity of preconditions, before calling the corresponding method. Therefore, preconditions are part of the public specification. In our concept, a programmer can use access modifiers to control the visibility of each invariant and therefore also its scope. Which code positions may access a defined class element is defined by its access modifier:

Definition 1 (Accessibility). *Each class element e_c of class c, which is either a class field f_c, a method m_c or an invariant i_c, has one access modifier $\alpha(e_c) \in AM$, while the set AM must be specified by the concrete programming language. An access modifier defines which code positions can access e_c. The predicate $IsAccessible\,(e_c, s)$ returns $True$, if the statement s can access the class element e_c. A class element e_c of the instance o of class c might be accessed either directly by $o.f$ or transitively through a get-method. A get-method get_{f_c}, for the class field f_c, is a method of class c, returning the unmodified value of f_c. In both cases, the predicate $IsAccessible(f_c, s)$ returns $True$. Furthermore, a statement can access a class element e_c transitively through references across several object instances. The predicate $IsAccessible^*(e_c, s)$ returns $True$, if the class element e_c can be accessed transitively by s.*

However, the syntax and semantic of each $\alpha \in AM$ depends on the selected programming language. Because we have exemplary implemented our methodology for Java, we support four different access modifiers[1]: *public*, *package*, *protected*, *private*. Elements e_c, declared as *public*, are accessible from any other method m_c in any other class $c \in C$. Each class $c_p \in C$ is member of one package $p \in \mathcal{P}$. Elements e_{c_p}, declared as *package*, within class c_p are accessible by all other methods m defined in class c'_p, declared in the same package p. Elements e_c, declared as *protected*, are accessible from methods defined in c and all classes extending c. Elements declared as *private* are accessible only from methods declared in the same class c.

The following predicate indicates, if a statement s can check an invariant i:

$$\begin{aligned} CanCheck(s, i) \Leftrightarrow\ & IsAccessible^*(i, s) \wedge \forall\, e \in Ref^*(i) \\ & \cup CalledMethods(i) : IsAccessible^*(e, s) \end{aligned} \tag{3}$$

[1] There exist no problem in adapting this methodology to a different semantic of access modifiers.

The second case considers dependencies between different proof obligations. For example, a postcondition might be verified only, if a corresponding invariant is valid. Therefore, a statement also depends on the validity of an invariant, if it refers to a proof obligation whose validity cannot be proven without assuming the correctness of that invariant. If a statement s requires the validity of an invariant i is given by the predicate $Requires(s, i)$.

$$Requires(s,i) \Leftrightarrow \exists\, \varrho = (P,S,Q) | s \in S_\varrho : \neg\Psi((P \setminus i, S, Q) \wedge \Psi(P \cup i, S, Q)) \quad (4)$$

In summary, a statement s depends on an invariant i if:

$$Depends(s,i) \Leftrightarrow \neg CanCheck(s,i) \vee Requires(s,i) \quad (5)$$

The set of depending code positions for one invariant i is defined by:

$$DependingS(i) = \forall c \in C\, \forall m \in M_c\, \forall s \in S_m\, \{s \,|Depends(s,i)\} \quad (6)$$

Step 4: Building The Verification Graph. The scope of an invariant defines when an invariant might be violated and when its validity is expected. We use a special call graph, called *Verification Graph (VG)*, to analyse the scope of each invariant.

Definition 2 (Verification Graph). *A Verification Graph $VG(i) = (V, E)$ is a tuple. Each vertex $v \in V$ has a reference m_v to a method m. Each edge $e = (v_i, v_j) \in E$ indicates a method call in m_{v_i} to the method referenced by its target m_{v_j}. Furthermore, each edge e has a reference s_e to the position of the method call in m_{v_i}. Within v_i, the edges are ordered by the position s_e in m_{v_i}. If $e_1 = (v_i, v_j) < e_2 = (v_i, v_k)$ the method m_{v_j} is called before the method m_{v_k} is called. Each method and method call is represented only once within the VG.*

The VG is built by analysing the possible call stack of each violating and depending method. We call this possible call stack *Context*.

Definition 3 (Context). *The context of any method $\hat{m}$ is defined as a set of pairs (m, s), where m refers to a method and $s \in S_m$ to a statement within the method m. Each pair corresponds to a method m, calling $\hat{m}$ with the statement $s = o.m'(\overrightarrow{p})$, and any list of parameters $\overrightarrow{p}$. If m is the source of the context, s might also be empty, denoted as $(m, \emptyset)$. The general n-order context, for $n \geq 0$, of a method m and a defined set of methods $M' \subset M$, of all methods M, is defined as:*

$$\mathcal{C}^n(m, M') = \left\{(\dot{m}, s)\,|s = o.m'(\overrightarrow{p}) \in S_{\dot{m}} \wedge m' \in \mathcal{C}^{n-1}(m) \wedge \dot{m} \notin M'\right\} \quad (7)$$

$$\mathcal{C}^0(m, M') = \{(m, \emptyset)\,|m \notin M'\} \quad (8)$$

The Verification Graph $VG(i) = (V, E)$ of a given invariant i is built as follows:

$$V = V_1 = \{(m)|\exists s \in m \wedge s \in ViolatingS(i)\} \cup \quad (9)$$

$$V_2 = \{(m)|\exists s \in m \wedge s \in DependingS(i)\} \cup \quad (10)$$

$$\{(m)|m \in \mathcal{C}(V_1, V_2) \cup \mathcal{C}(V_2, V_1)\} \quad (11)$$

$$E = \{(v_i, v_j)|v_i, v_j \in V \wedge m_{v_i}\ \texttt{calls}\ m_{v_j}\} \quad (12)$$

Data: Start vertex: v_j, Start statement: s, Searched type: $t = \{violating, depending\}$, Search behind s: b
Result: The closest vertex of type t wthin the context of v_j

```
1  if v_j marked then return ∅ // Skip visited vertexes ;
2  mark v_j ;                                          // Mark vertex as visited
3  if v_j is t then return v_j ;
   // Analysing all called methods in m_{v_j} before s is reached
   // by following edges e_{j,k} whose position s_{e_{j,k}} in m_{v_j} is smaller than s
4  foreach e_{j,k} = (v_j, v_k) | s_{e_{j,k}} < s | from s to 0 do
       // Searching recursively from begin on (m_{v_k}[0]) in each called method
5      v̂ = findNode(v_k, m_{v_k}[0], t, true)
6      if v̂ ≠ ∅ then return v̂ ;
   // Analysing all methods called after s is reached
   // by following edges e_{j,k} whose position s_{e_{j,k}} in m_{v_j} is greater than s.
   // This is used if we analyse methods called before the original s was
      reached.
7  if b then
8      foreach e_{j,k} = (v_j, v_k) | s_{e_{j,k}} > s | from s to n do
           // Searching recursively from begin on (m_{v_k}[0]) in each called
              method
9          v̂ = findNode(v_k, m_{v_k}[0], t, true)
10         if v̂ ≠ ∅ then return v̂ ;
   // Analysing all methods calling m_{v_j}
11 foreach e_{k,j} = (v_k, v_j) do
       // Searching from the position calling m_{v_k} to the begin of m_{v_j}.
12     v̂ = findNode(v_k, s_{e_{k,j}}, t, false)
13     if v̂ ≠ ∅ then return v̂ ;
14 return ∅ ;                                          // No vertex found
```

Algorithm 1. FindNode(v_j,s,t,b) : Searching closest node

Step 5: Analysing Invariant Scopes. The scope of an invariant defines when an invariant is expected to be valid and when it is allowed to be invalid. We must guarantee, that an invariant is valid whenever a depending code position is reached. In general, the invariant must be ensured by code positions modifying any value, referenced by that invariant. In combination with depending code positions, we are searching the last violating code position called before the depending code position is reached. We call the corresponding vertexes *Ensuring Vertexes*, because they must ensure the invariants validity. These positions are found by searching the shortest paths between each depending vertex and the closest violating vertex. For one depending code position $s \in m$, we analyse two categories of methods: (1) Methods (transitively) called in m before s is reached. (2) Methods (transitively) calling m. Algorithm 1 formalises this search for a given depending code position $s \in m_{v_j}$, by calling $FindNode(v_j, s, violating, false)$. A detailed walk through, based on an example, is presented in Section 3.

Step 6: Creating Proof Obligations. Proof obligations define an expected behaviour for a defined sequence of statements.

Definition 4. *Proof Obligation. A proof obligation $\varrho = (P, S, Q)$ is a triple. It combines a set of assumptions (P), an ordered list of statements (S) and a goal (Q). Each assumption and the goal are represented as boolean predicate. To prove a proof obligation, one must show, that each possible evaluation of S, validates Q, while assuming P. The predicate $\Psi(\varrho)$ is true if ϱ can be verified.*

Here, they are used to ensure that each ensuring method respects the corresponding invariant. But it is not sufficient to analyse the violating method. This is because the invariant might be ensured within the method calling the ensuring method. For example, a set-method is marked as ensuring method. In this case, the method calling the set-method must ensure, that the set value respects the invariant. Therefore, the code block s of the corresponding proof obligation $\varrho = (P, S, Q)$ must combine statements of both methods. In general, each code block, not depending on that invariant, within the call stack of the ensuring method may ensure its validity. For one ensuring node $\check{v}$, we analyse the undirected path $p(\hat{v}, \check{v})$ to the closest depending vertex $\hat{v}$. The closest depending node is found by using $FindNode$ defined by Algorithm 1.

In summary, we create the set of proof obligation $\rho(i)$ for each invariant i:

$$D = DependingS(i) \tag{13}$$

$$E = \bigcup_{d \in D} FindNode(v(d), d, violating, false) \tag{14}$$

$$P = \bigcup_{e \in E} p = (FindNode(v(e), e, depending, false), v(e)) \tag{15}$$

$$S(p) = \bigcup_{e=(v_j, v_k) \in p} m_{v_j}[0, e] \ \cup \ m_{\check{v}} \tag{16}$$

$$\rho(i) = \bigcup_{p(\hat{v}, \check{v}) \in P} (Preconditions(\hat{v}), S(p), i) \tag{17}$$

The predicate $Preconditions(\hat{v})$ refers to the set of preconditions defined for $\hat{v}$. We use the syntax $m_{v_j}[0, e]$ to refer to the subset of statements in S_m from the begin of m (position 0) to the position referenced by the edge s_e.

2.1 Soundness

We sketch the proof of soundness, by showing that the set of generated proof obligations is sufficient to guarantee an invariants validity whenever a depending code position is reached:

Theorem 1. *If all proof obligations could be verified, every invariant i is valid, whenever one of its referenced values $v \in Ref^*(i)$ is accessed by a statement $\hat{s}$ in method $\hat{m}$, with the set of valid assumptions P and defined condition Q such that $\neg\Psi(\varrho = (P \setminus i, m, Q)) \wedge \Psi(\varrho = (P \cup i, m, Q))$.*

Proof. If $\hat{s}$ fulfils the properties defined in the theorem, we know $\hat{s} \in DependingS(i)$ (Equation 6). If i is not valid when $\hat{s}$ is evaluated, there must be one statement $\check{s}$ within method $\check{m}$ which has invalidated i and which has been evaluated before $\hat{s}$. To invalidate i, $\check{s}$ must assign an invalid value to any field referred by i: $Ref!(\check{s}) \cup Ref^*(i) \neq \emptyset$. Regarding $\check{s}$, there exist two possibilities: (1) $\check{s}$ is the last code position modifying a value referenced by i, before reaching $\hat{s}$. (2) It exists a code position $\check{\check{s}}$ which modifies a value referenced by i, which is evaluated after $\check{s}$ and before $\hat{s}$ is reached. In the first case, $FindNode(v(\hat{s}), \hat{s}, violating, false) = \tilde{v} = v(\check{s})$. In the second case, $\check{\check{s}}$ must re-establish i and $FindNode(v(\hat{s}), \hat{s}, violating, false) = \tilde{v} = v(\check{\check{s}})$. The syntax $v(s)$ refers to the vertex v whose referenced method contains s. Step 6 generates a proof obligation $\varrho = (P, S, i)$ and the statement sequence S contains as last sequence the statements of $m_{\tilde{v}}$ (Equation 16). Therefore, if ϱ can be verified, i is re-established after $m_{\tilde{v}}$ has been evaluated and before $\hat{s}$ is reached. □

3 Case Studies

The presented methodology provides a higher flexibility in defining object invariants while requiring less specification overhead. This is shown by demonstrating the analysis process of different examples. The examples represent challenges and code examples which have been addressed by latest related work. Thereby we can compare the specification overhead of our methodology with the one required by related approaches. Furthermore, we present one example which cannot be verified using current approaches. All examples have been implemented in Java. Invariants were defined using the syntax of the Java Modelling Language, as descried in [7]. The defined access modifiers are interpreted as described in Step 3 within Section 2.

Challenge 1: Gradual Updates. Invariants may refer to multiple class elements. A gradual update of referenced values may invalidate an invariant temporary. This enables access to an invalid object. Current methodologies use additional specification elements to define when an invariant is valid. The approach presented in [14] uses *unpack* and *pack* statements. These statements mark the begin and end of an interval, in which an object is allowed to be invalid. Their usage is indicated in Listing 1. The example is based on a data type, representing a numerical interval by storing a min and max value. It provides a method $getSize()$, which guarantees a positive return value. Thereby, $getSize()$ requires the validity of the invariant $getMin() <= getMax()$.

In Listing 1.2 *pack* and *unpack* is correctly used, because *getSize* is called, after the object invariant has been ensured by calling *pack*. In Listing 2 the *pack* statement is located after $getSize()$ and has been called on the invalid object. This is a specification error, because the programmer declares the scope of the invariant incorrectly. This causes a violated postcondition of $getSize()$. We want to demonstrate two points in this example: (1) How the presented methodology analyses the code without using additional specification elements like *pack* and

```
1 class Interval {
2  //@public invariant getMin() <= getMax();
3  private int min,max;
4  // For reasons of compactness we do not display
5  // the constructor and the analog set/get methods for min
6  public void setMax(int max){ this.max=max;}
7  public int getMax(){ return this.max; }
8  //@ensures \return >= 0;
9  public void getSize() { return this.max-this.min; }
10 }
11 class UseInterval {
12  //@ensures \return >= 0;
13  public int main() {
14   Interval inter = new Interval(5,7);
15   //unpack inter as Interval
16   inter.setMin(8);
17   inter.setMax(9);
18   //pack  inter as Interval
19   return inter.getSize();
20  }
21 }
```

Listing 1.1. Gradual update of an invariant

unpack. (2) How the automatic code analysis of the presented methodology detects the described error of Listing 1.3.

The VGs (VG_a,VG_b) for both examples are illustrated in Figure 3.

```
1 //@ensures \return >= 0;
2  public int main2() {
3   Interval inter = new Interval(5,7);
4   //unpack inter as Interval
5   inter.setMin(8);
6   int dist = inter.getSize();
7   inter.setMax(9);
8   //pack  inter as Interval
9   return dist;
10  }
```

Listing 1.2. Specification error

In the following, we use the vertex labels as reference to the different methods. In Step 1 we analyse the references of the public invariant i: $Ref^*(i) = \{min, max\}$. In Step 2 we analyse which code positions modify any referenced value. These are the methods $setMin()$ and $setMax()$: $ViolatingS = \{s_{v_3}, s_{v_4}\}$. In Step 3 we analyse which code positions depend on i. The method $getSize()$ is the only depending code position: $DependingS = \{s_{v_5}\}$. This is because the

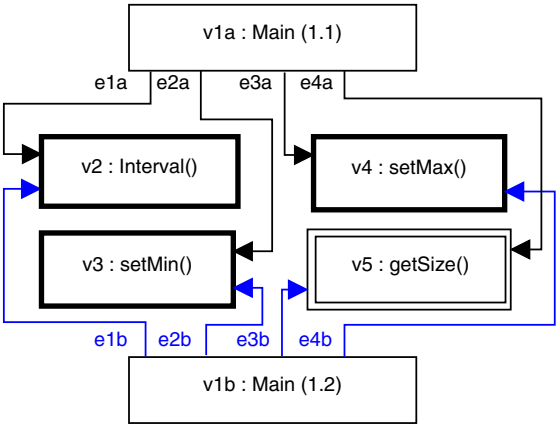

Fig. 3. The VGs for the Listings (a) 1.1 and (b) 1.2

postcondition $\varrho_{v_5} = (i, (max - min), (max - min) >= 0)$ cannot be verified without assuming i. In Step 4, we build the VG, illustrated in Figure 3. In Step 5, we search the ensuring methods of each depending code position. In example (a) the analysis follows first edge $e4a$ and next edge $e3a$. The first edge is followed by lines 11-13 of Algorithm 1 and the second edge by lines 3-6, within the first recursive call. Thereby we reach vertex $v4$ and we find the closest violating method. In example (b) the analysis follows first edge $e3b$ and next edge $e2b$. Here, vertex $v3$ is the closest violating method. In Step 6, we create the proof obligations ensuring i. Therefore, we search for each ensuring vertex the closest depending vertex. In these examples, there is no depending vertex within the context of both ensuring vertexes. Therefore, the context covers the full program until each ensuring vertex is reached. The proof obligations are: $\varrho_A = (\emptyset, m_{v_2} \cup m[14] \cup m_{v_3} \cup m_{v_4}, i)$, $\varrho_B = (\emptyset, m_{v_2} \cup m[14] \cup m_{v_3}, i)$. Using Z3 as verification back-end, we can prove the validity of ϱ_A and ϱ_B.

Challenge 2: Recursive Data-Structures. Figure 4 contains an example for a recursive data structure width following private invariant: $inv1 = ((val >= 0) \wedge (prev.val < 0)) \vee ((val < 0) \wedge (prev.val >= 0))$. Recursive data structures have been addressed by Leino et al. in [2]. They require the additional specification statements *peer*, *dependent* and *owner dependent*. The presented methodology does not require any additional specification elements. Our analysis recognises that the public *setPrev*-method (m_1) and the constructor (m_2) are the only two methods modifying the value *val*. Because the invariant is private, both public methods must ensure the invariant. The two proof obligations are $\varrho_1 = (\emptyset, m_1, i)$ and $\varrho_2 = (\emptyset, m_2, i)$.

Challenge 3: Dependencies on an Invariant Subset. Current methodologies do not distinguish between valid and invalid invariants within one object. Methods cannot require the validity of an invariant subset only. Current methodologies cannot be used to verify the example in Figure 5 or in Listing 1.3.

```
1 class Starter {
2  // Post: Return Value > 0;
3  static main(int a) {
4   if(a < 2)
5   { throw new Exception(); }
6   Data d = new Data();
7   d.setA(a); d.setB(0);
8   return DataProcessorA.
     process(d);
9 } }
10 class DataProcessorA {
11  static int process(Data d) {
12   assert(d.getA()>2);
13   int b = 2 * sqrt(d.getA());
14   d.setB(b);
15   return DataProccessorB.
     process(d);
16 } }
17 class DataProcessorB {
18  // Post: Return Value > 0;
19  static int process(Data d)
20  { return d.getC()/d.getB();}
21 }
```

Listing 1.3. Partly valid objects

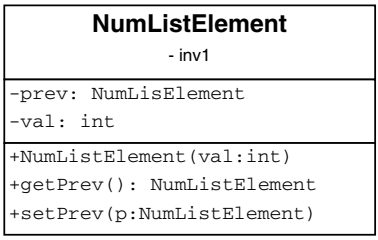

Fig. 4. Recursive data structure

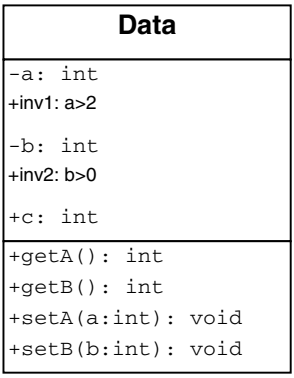

Fig. 5. Data model of Listing 3

This Listing uses the data structure shown in Figure 5, which contains two invariants ($i_1 = inv1$) and ($i_2 = inv2$). The *Data*-object is passed as an argument to the $DataProcesorA : process$-method and later further to the $DataProcessorB : process$-method. The $DataProcessorA : process$-method uses the square root of the a-field-value to calculate a new b-field-value. Because of the defined assertion, it relies on the invariant i_1. But at call time of $DataProcessorA : process$, the invariant i_2 is invalid. The invariant i_2 is required not until $DataProcessorB : process$ is called. This method uses the b-field-value as divisor in Line 19 of Listing 1.3. This implies the obligation $b! = 0$, which cannot be assured without assuming i_2. In summary, both methods $DataProcessorA : process$ and $DataProcessorB : process$ rely on a different subset of invariants defined within the *Data*-object. We use this example to demonstrate how the presented methodology distinguishes between valid and invalid invariants within one object. This example causes two Verification Graphs $VG(i_1)$ and $VG(i_2)$, one for each invariant. They are both illustrated in Figure 6. Again, we start by analysing the references of each invariant: $Ref^*(i_1) = \{a\}$, $Ref(i_2) = \{b\}$. These values may be modified by following code position: $Violating(i_1) = \{s_{v_4}\}$, $Violating(i_2) = \{s_{v_3}\}$. The depending code positions are $DependingS(i_1) = \{m_{v_5}[12]\}$ and $DependingS(i_2) = \{m_{v_6}[20]\}$. We use the syntax $[\bullet]$ to identify the corresponding code position by their

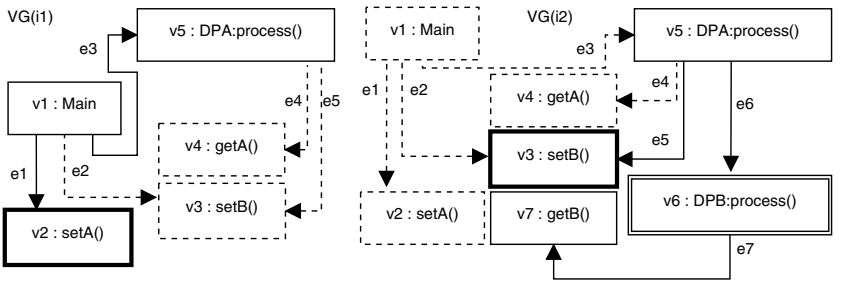

Fig. 6. The VG for the example of Goal 6

line number in Listing 1.3. The ensuring method for $m_{v_5}[12]$ is m_{v_2}, following the edges $e3$,$e2$,$e1$. For m_{v_6} the ensuring method is m_{v_3}, following the edges $e6$ and $e5$. There is no depending code position within the context of m_{v_2} and m_{v_3}. Therefore, we add the all statements to S, until $setA()$ respectively $setB()$. The corresponding proof obligations are: $\varrho_{i_1} = (\emptyset, main[4-6] \cup m_{v_2}, i_1)$, $\varrho_{i_2} = (\emptyset, main[4-6] \cup m_{v_2} \cup m_{v_3} \cup m_{v_2}[12-13] \cup m_{v_3}, i_2)$. Both proof obligations can be verified, using the Z3 as verification back-end.

Results. The analysis of Challenge 1, 2, and 3 shows that the presented methodology does not require specification overhead like current state of the art methods. Furthermore, the automatic analysis prevents errors caused by the wrong usage of additional specification elements, as shown in Challenge 1. The presented methodology recognised the violated postcondition in Listing 1.3, caused by the access to an invalid object. The determination between valid and invalid invariants within one object enables the verification of Challenge 3, which is not possible with current state of the art approaches.

4 Conclusion and Future Work

We have introduced a new methodology to specify and verify object invariants. This methodology uses access modifiers to control the scope of an object invariant. An automatic control flow analysis is used to analyse when invariants may be invalidated and when they must be re-established. This reduces the specification overhead and helps to prevent errors through the usage of specification statements, as we have shown in Challenge 1 in Section 3. The presented methodology distinguishes between invalid and valid invariants within one object. Thereby, it supports more flexible scopes of invariants, as we have shown in Challenge 3. The computational overhead does highly depend on the analysed program structure. In general, public invariants cause a larger number of paths that need to be considered to validate an invariant, because more code positions may invalidate an public invariant. The same applies to invariants with a large number of references. Therefore, the higher flexibility may cause a higher number of proof obligations. The reduced specification overhead causes a higher computational overhead. We have implemented the presented methodology within a tool

analysing single threaded Java programs using the Z3 solver as formal verification back-end. Currently we do not support the full Java language specification (e.g. method overloading). Current limitations are caused by high implementation efforts but should not influence the completeness of the presented methods. At the moment we must apply all six steps to each source file after every change in order to validate defined invariants. Future work may address the integration of a change review to analyse only code fragments, affected by latest changes.

References

1. Summers, A.J., Drossopoulou, S., Müller, P.: The need for flexible object invariants. In: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, pp. 1–9. ACM (2009)
2. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
3. Barnett, M., Fähndrich, M., Müller, P., Leino, K.R.M., Schulte, W., Venter, H.: Specification and verification: The spec# experience (2009)
4. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
5. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Sci. Comput. Program. 62(3), 253–286 (2006)
6. Meyer, B.: Eiffel: The Language. Prentice-Hall, Inc., Upper Saddle River (1992)
7. Leavens, G.T., Baker, A.L., Ruby, C.: Jml: a java modeling language. In: Formal Underpinnings of Java Workshop, at OOPSLA 1998 (1998)
8. Huizing, K., Kuiper, R.: Verification of object oriented programs using class invariants. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 208–221. Springer, Heidelberg (2000)
9. Dietl, W., Müller, P.: Object ownership in program verification. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 289–318. Springer, Heidelberg (2013)
10. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
11. Müller, P.: Reasoning about object structures using ownership. In: Meyer, B., Woodcock, J. (eds.) Verified Software. LNCS, vol. 4171, pp. 93–104. Springer, Heidelberg (2008)
12. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
13. Lu, Y., Potter, J., Xue, J.: Validity invariants and effects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 202–226. Springer, Heidelberg (2007)
14. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3, 2004 (2004)
15. Leavens, G.T., Muller, P.: Information hiding and visibility in interface specifications. In: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 385–395. IEEE Computer Society, Washington, DC (2007)