Considerate Reasoning and the Composite Design Pattern

Alexander J. Summers[1,2] and Sophia Drossopoulou[1]

[1]Imperial College London [2] ETH Zürich

Abstract. We propose *Considerate Reasoning*, a novel specification and verification technique based on object invariants. This technique supports succinct specifications of implementations which follow the pattern of breaking properties of other objects and then notifying them appropriately. It allows the specification to be concerned only with the properties directly relevant to the current method call, with no need to explicitly mention the concerns of subcalls. In this way, the specification reflects the division of responsibility present in the implementation, and reflects what we regard as the natural argument behind the design.
We specify and prove the well-known Composite design pattern using Considerate Reasoning. We show how to encode our approach in Boogie2. The resulting specification verifies automatically within a few seconds; no manual guidance is required beyond the careful representation of the invariants themselves.

1 Introduction

Verification for imperative object-oriented languages is challenging. The arbitrarily complicated heap structures which can arise out of even quite short programs, and the potential for *aliasing* make it difficult to structure the verification argument in an organised fashion, or to predict the effects of code fragments.

Some approaches to these challenges use specification languages which reflect the heap structure explicitly, describing the intended topology of objects and references in a logic which includes customised assertions for the purpose. Such approaches include separation logic [20, 18], dynamic frames [10], implicit dynamic frames [23] and regional logic [2].

Other approaches build on the concept of *object invariant*, and usually support some variation of *visible states semantics* (with the notable exception of the Boogie methodology [3]). In visible states semantics, object invariants should hold at the pre- and post-states of method calls, but may be temporarily broken during method execution. Various refinements have been proposed, usually based on some notion of *ownership* - a way of imposing structure on the heap by requiring that one object is encapsulated within another. This idea neatly supports client-provider implementations in which the encapsulated object is only modified via its owner; but it cannot support another programming pattern, whereby methods may break other objects' invariants and then notify them, ie call other methods to fix them. This kind of pattern is prevalent, e.g., in the

Marriage example, the Subject-Observer and Composite patterns [7], and the Priority Inheritance Protocol [22].

```
class Composite {
  private Composite parent;
  private Composite[] comps;
  private int count = 0;
  private int total  = 1;

  // Inv1:  $1 \leq \text{total} \wedge 0 \leq \text{count}$
  // Inv2:  total  $= 1 + \sum_{0 \leq i < \text{count}} \text{comps[i].total}$

  // requires :  c $\neq$ null && c.parent = null;
  public void add(Composite c) {
    // resize  array  if  necessary
    comps[count] = c;
    count++;
    c.parent  = this;
    addToTotal(c.total);
  }

  private void addToTotal(int p) {
     total  += p;
     if(parent != null) parent.addToTotal(p);
  }
}
```

Fig. 1. A single-class variant of the Composite pattern

The Composite pattern, recently proposed as a verification challenge in [11], was the 2008 challenge problem at the SAVCBS workshop. It describes a tree-structure, and allows addition of subtrees in any part of the tree. Figure 1 contains a simplified version of the code from [11]. A Composite node has fields comps which contain all its direct descendants, parent which points to its parent, and integer total. The code has to preserve the invariant that the total field of an object is equal to the size of the subtree rooted at that object.

The major difficulty in verifying this invariant is that the data structure can be directly modified at any point, by calling add on any Composite object. This is problematic for, e.g., ownership-based approaches, since these typically require modification of owned objects to be controlled by the owning object (thus modification would be preceeded by a top-down traversal of the tree-structure). Similarly, separation logic specifications of such patterns typically require recursive predicates describing properties over the data structure [17, 18]; such predicates are easier to fold/unfold from the root of the structure downwards.

In this paper we propose *Considerate Reasoning*, a novel approach to verification, and apply it to the Composite problem. Considerate Reasoning was briefly outlined in [24]; it extends the work of Middelkoop et. al. [15], and is related to [4, 12]. It is based on visible states semantics; in order to support methods meant to fix invariants, it introduces the specification construct broken. Invariants declared "broken" in a method specification are *not* expected to hold before calls to the corresponding methods, but are expected to be re-established by these methods. All invariants are expected to hold at the end of a method execution. Thus, the specification of method addToTotal contains broken : Inv2(this), cf. Fig. 2.

In Fig. 2 we give a specification of the Composite in Considerate Reasoning. This specification is concerned only with properties directly relevant to the current method call, without needing to explicitly mention the concerns of subcalls. In this way, the specification reflects the division of responsibility present in the implementation, and reflects what we regard as the natural argument behind the design.[1] This is the specification we ultimately expect the user to have to write (up to a couple of additional keywords whose use will become apparent).

Considerate Reasoning also introduces *concerns-descriptions*, which describe which invariants may be broken by a field update. These are used to determine which invariants may be broken (ie are *vulnerable*) at each code point, and therefore must be re-established at the end of a method body. Because no tool directly supports Considerate Reasoning, we have encoded our approach in Boogie2 [13], using explicit assume and assert statements to describe our handling of invariants. We developed refinements which allow for simplifications of the required proof obligations. The resulting specification is natural and succinct, and verifies automatically in approximately six seconds. In section 3.5 we outline how a tool could infer concerns-descriptions and other internal concepts.

Conventions To simplify the presentation, we make the following simplifying assumptions: The names of fields declared in different classes should be distinct. The names of invariants declared in different classes should be distinct. Type-incorrect expressions in the specifications are considered false. The predicate describing the meaning of an invariant I is called P_I. Invariants only depend on path expressions containing field accesses, and in particular do not feature predicates. [2]

[1] Our specification does not express framing, which we left to further work. Note however, that in the Composite example, we believe that the client naturally should not depend on the value of total remaining unmodified.

[2] The last assumption is the only one to represent a true restriction. Note, however, that all invariants we require for the Composite pattern have definitions we permit, even though other specifications of the Composite used recursive predicates. We expect recursively defined predicates to be expressible through explicit invariants in a semantics where the invariants of all objects are expected to hold by default.

2 A Considerate Specification of the Composite

We first identify what we believe to be the intuitive argument underlying the implementation. By making this argument precise, we are able to identify and incorporate invariants and conditions which are necessary for soundness but missing from the original code.

Had we only been interested in the preservation of $Inv2$(this), then the following would have been an adequate implementation for adding a component:

```
public void addWeak(Composite c) {
  // resize array if necessary
  comps[count] = c;
  count++; // breaks Inv2(this)
  c.parent = this;
  total += c.total; // fixes Inv2(this),
                    // breaks Inv2(o), where this ∈ o.comps
}
```

This simpler implementation does preserve the invariant of the receiver, but in turn it breaks Inv2 of any object with the receiver in its comps. The real implementation takes account of this fact: the method addToTotal performs the role not only of fixing the invariant of the receiver, but also of being *considerate* of the invariants of other objects. In particular, after the total field of the receiver is updated, the parent of the receiver is notified of the change by another call to addToTotal, in order to ensure that their invariant can also be maintained.

How do we know that this implementation is indeed correctly considering *exactly the* concerned invariants? In particular, why is it correct for the addToTotal method to recursively call the parent of the current receiver? The intuitive argument here depends on the assumption that the comps of any object are exactly those objects which point to it via parent fields. This assumption is *implicit* in the design pattern, but was missing in [11]. We add two further invariants:

Inv3(o): $\forall 0 \leq i < \mathsf{o.count} : \mathsf{o.comps[i].parent} = \mathsf{o}$

Inv4(o): $\mathsf{o.parent} \neq \mathsf{null} \Rightarrow \exists 0 \leq i < \mathsf{o.parent.count} : \mathsf{o.parent.comps[i]} = \mathsf{o}$

A further subtle problem arises if the comps of an object are not distinct. If an object is in the comps of another object *twice*, then the implementation of addToTotal would be incorrect. We add a further invariant:

Inv5(o): $\forall 0 \leq i \neq j < \mathsf{count} : \mathsf{o.comps[i]} \neq \mathsf{comps[j]}$

This invariant may seem redundant, since it is *preserved* by the methods of the class Composite; however there is no guarantee that the heap structure is *already* a tree; this is indispensable in the proof that the information propagated upwards through addToTotal is correct. Note that the *combination* of the invariants Inv3, Inv4 and Inv5 guarantee that the whole Composite structure is a tree.

We now give a specification of the Composite in Figure 2. We include in the specification of addToTotal the declaration broken : Inv2(this), reflecting that this method fixes a broken invariant. The precise semantics of this construct will be made clear in the following section. In order to make it possible for addToTotal to guarantee to fix the declared invariant, we add a pre-condition requiring that the value of total is "out" by exactly the value passed as argument to the method.

```
class Composite {
  private Composite parent;
  private Composite[] comps;
  private int count = 0;
  private int total  = 1;

  // Inv1(o):  $1 \leq$ o.total $\wedge\ 0 \leq$ o.count
  // Inv2(o):  o. total  $= 1 + \sum_{0 \leq i < \text{o.count}}$ o.comps[i].total
  // Inv3(o):  $\forall 0 \leq i <$ o.count : o.comps[i].parent = o
  // Inv4(o):  o.parent $\neq$ null $\Rightarrow \exists 0 \leq i <$ o.parent.count : o.parent.comps[i] = o
  // Inv5(o):  $\forall 0 \leq i \neq j <$ o.count : o.comps[i] $\neq$ o.comps[j]

  // requires :  c $\neq$ null;
  // requires :  c. parent  = null;
  public void add(Composite c) {
     comps[count] = c;
     count++;
     c. parent  = this;
     addToTotal(c. total );
  }

  // broken:  Inv2( this )
  // requires :  this.total $+ p = 1 + \sum_{0 \leq i < \text{count}}$ comps[i].total
  private void addToTotal(int p) {
     total  += p;
     if  (parent  != null)  parent . addToTotal(p);
  }
}
```

Fig. 2. A considerate-style specification in Java

3 Considerate Reasoning

Our proposed methodology, once fully supported by tools, requires the user to:

1. Define the invariants.
2. Declare certain invariants as *structural* (Definition 5 below).
3. Define the broken declarations along with method specifications.

We will now explain the workings of our methodology, and then in Section 3.5 we will outline how it can be automated. Our methodology consists of the following:

1. An *invariant semantics*, specifying which invariants must hold at which points in execution.
2. The concept of a *concerns-description*, which describes which objects' invariants are concerned with field updates in a program.

3. The derivation of *vulnerable invariants* at all intermediate program points, computed from the code and concerns-description.
4. A *verification technique*, defining sufficient proof obligations to guarantee soundness with respect to the invariant semantics.

3.1 Invariant Semantics

Visible states semantics [19, 16] requires all invariants of all objects to hold immediately before and immediately after any method calls. For simplicity of the presentation, we base the work in this paper on this simple visible states semantics, but our work could be applied to weaker variants of the semantics, in which only the invariants of certain objects need hold in the visible states. programming patterns which involve calling certain methods to *fix* broken invariants (e.g method addToTotal in our example), the visible states semantics requirement is too restrictive. In the Considerate Reasoning methodology we add the necessary flexibility with the extra specification construct broken : to explicitly declare exceptions to the visible states semantics. Invariants declared "broken" in a method specification are *not* required to hold before calls to the corresponding methods, but are expected to be fixed by these methods.

Definition 1 (Broken Declarations and Invariant Semantics). *A method specification may contain a declaration* broken : $\mathsf{I}_1(\mathsf{e}_1), \mathsf{I}_2(\mathsf{e}_2), .., \mathsf{I}_n(\mathsf{e}_n)$.
A verification methodology is sound *if for any method* m *whose specification contains* broken : $\mathsf{I}_1(\mathsf{e}_1), \mathsf{I}_2(\mathsf{e}_2), .., \mathsf{I}_n(\mathsf{e}_n)$*, it guarantees that:*

1. *At the beginning of execution of* m*, all invariants of all objects must hold, except for* I_i *for those objects denoted by an expressions* e_i*, for* $i \in \{1, ..., n\}$.
2. *At the end of method execution, all invariants of all objects must hold.*

3.2 Concerns-Descriptions

Updating objects' fields may break invariants of other objects. For example, updating total of this, may break *Inv2* of this.parent. We say that objects whose invariants may be broken when a field of another object is updated, are *concerned* with the field. Obviously, concern is naturally a dynamic notion. For a static approximation of this notion, we define *concerns-descriptions* which associate with each field name f (the field to be updated) and invariant name I (the invariant under consideration), a description of the set of concerned objects. This set usually depends on the identity of the object being updated, therefore the set description may mention a special variable mod, which denotes the object being modified. Thus, the variable mod has a special meaning for concerns-descriptions, similar to the way the variable this has a special meaning for methods. Furthermore, we allow additional flexibility to the descriptions by also including a (possibly empty) list of invariant names, which we call *supporting invariants.* Their intuitive meaning is that the set described is only guaranteed to be conservative at program points where the supporting invariants are guaranteed to hold (for all objects). This allows us to use more-refined definitions of the sets of concerned objects, which depend on the guarantees that other invariants provide - the use of this feature will become clear shortly.

Definition 2 (Concerns Descriptions). *A* concerns-description $\mathcal{D}$ *is a mapping from a field name and an invariant name to a pair consisting of a* set description*, and a (possibly-empty) set of invariant names - the* supporting invariants*.* [3] *A* set description *is a description of a set of references, parameterised by a special variable* mod*; it may be described using usual set-theoretical operations, including comprehensions.*

For the Composite pattern, a possible concerns-description would determine $\mathcal{D}(\mathsf{count}, \mathsf{Inv1}) = (\{\mathsf{mod}\}, \emptyset)$, specifying that when the field count of any object mod is modified, at most the single object mod has invariant Inv1 broken. To obtain a sound verification methodology, the concerns-descriptions should be "big enough", i.e., any object whose invariant could be violated by a field update should fall within the corresponding described set. In fact, we make the weaker requirement that the set must be guaranteed "big enough" so long as the supporting invariants hold for all objects.

Definition 3 (Admissible Descriptions). *A concerns-description* $\mathcal{D}$ *is* admissible *if, for all invariants* $\mathsf{I}, \mathsf{I}_1, \mathsf{I}_2, \ldots, \mathsf{I}_\mathsf{m}$*, such that* $\mathcal{D}(\mathsf{f}, \mathsf{I}){\downarrow_2} = \{\mathsf{I}_1, \mathsf{I}_2, \ldots, \mathsf{I}_\mathsf{m}\}$ *and for any (sub-)expression* $o.\mathsf{f}_1.\mathsf{f}_2 \ldots .\mathsf{f}_n.\mathsf{f}$ *(with* $n \geq 0$*) occurring in* $P_\mathsf{I}(o)$*, we can prove for arbitrary* o *that:*

$$(\forall o', P_{\mathsf{I}_1}(o') \wedge P_{\mathsf{I}_2}(o') \ldots P_{\mathsf{I}_m}(o')) \wedge P_\mathsf{I}(o) \Rightarrow o \in \mathcal{D}(\mathsf{f}, \mathsf{I}){\downarrow_1}[o.\mathsf{f}_1.\mathsf{f}_2 \ldots .\mathsf{f}_n/\mathsf{mod}]$$

Consider the simple case of no supporting invariants being specified (i.e., $\mathcal{D}(\mathsf{f}, \mathsf{I}) = (S, \emptyset)$ for some set description S, and $m = 0$ in the definition above). Then admissibility guarantees that whenever we modify the field f of an object mod and the invariant I of an object o can become broken as a result, it must be the case that o$\in S$.

Note that we only need to show that o is in the described set if the invariant of o actually held - since we are trying to predict the invariants which *get* broken by a particular field update, we are only interested in the case where an invariant held prior to the update. There is a simple, mechanical way of deriving one such admissible concerns-description directly from the definitions of the invariants:

Definition 4 (Simplest Concerns-Descriptions). *For expressions* e_1 *and* e_2 *we write* $e_1 \sqsubseteq e_2$ *to mean* e_1 *is a syntactic subexpression of* e_2*. We then define the* simplest concerns-description $\mathcal{D}_S$ *for any field* f *and invariant* I *as follows:*

$$\mathcal{D}_S(\mathsf{f}, I) = \left(\bigcup_{n \geq 0,\ \mathsf{this}.\mathsf{f}_1.\mathsf{f}_2 \ldots .\mathsf{f}_n.\mathsf{f} \sqsubseteq P_\mathsf{I}(this)} \{o \mid o.\mathsf{f}_1.\mathsf{f}_2 \ldots .\mathsf{f}_n = \mathsf{mod}\} \quad , \quad \emptyset \right)$$

[3] We write $Q{\downarrow_1}$ and $Q{\downarrow_2}$ for the first and second projections of pair Q, respectively.

We treat array accesses analogously to field accesses, except that if any quantified variables occur in an array index expression, we additionally include existential quantifiers (with the same bounds) around the equality generated in the set comprehension.

For the Composite, we derive the following simplest concerns-description, in which the shorthand $\mathsf{o} {\in} \mathsf{o}'.\mathsf{comps}$ stands for $\exists 0 {\leq} i {<} \mathsf{o}'.\mathsf{count} :: \mathsf{o}'.\mathsf{comps}[\mathsf{i}] = \mathsf{o}$:

$$\begin{array}{llll}
\mathcal{D}_S(\mathsf{parent}, \mathsf{Inv3}) = (\{o \mid \mathsf{mod} \in o.\mathsf{comps}\}, \emptyset) & & \mathcal{D}_S(\mathsf{parent}, \mathsf{Inv4}) = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}_S(\mathsf{comps}, \mathsf{Inv2}) = (\{\mathsf{mod}\}, \emptyset) & & \mathcal{D}_S(\mathsf{comps}, \mathsf{Inv3}) = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}_S(\mathsf{comps}, \mathsf{Inv4}) = (\{o \mid o.\mathsf{parent} = \mathsf{mod}\}, \emptyset) & & \mathcal{D}_S(\mathsf{comps}, \mathsf{Inv5}) = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}_S(\mathsf{count}, \mathsf{Inv1}) \;\; = (\{\mathsf{mod}\}, \emptyset) & & \mathcal{D}_S(\mathsf{count}, \mathsf{Inv2}) \;\; = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}_S(\mathsf{count}, \mathsf{Inv3}) \;\; = (\{\mathsf{mod}\}, \emptyset) & & \mathcal{D}_S(\mathsf{count}, \mathsf{Inv4}) = (\{o \mid o.\mathsf{parent} = \mathsf{mod}\}, \emptyset) \\
\mathcal{D}_S(\mathsf{count}, \mathsf{Inv5}) \;\; = (\{\mathsf{mod}\}, \emptyset) & & \mathcal{D}_S(\mathsf{total}, \mathsf{Inv1}) \;\;\; = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}_S(\mathsf{total}, \mathsf{Inv2}) \;\;\; = (\{\mathsf{mod}\} \cup \{o \mid \mathsf{mod} \in o.\mathsf{comps}\}, \emptyset) & & \\
\mathcal{D}_S(\mathsf{f}, \mathsf{I}) \qquad\quad\;\; = (\emptyset, \emptyset) \;\; \textit{otherwise} & &
\end{array}$$

Admissibility is trivially satisfied by the simplest concerns-description:

Proposition 1. *The simplest concerns-description $\mathcal{D}_S$ is admissible.*

Observe that $\mathcal{D}_S$ given above uses sets $\{\mathsf{mod}\}$, $\{o \mid o.\mathsf{parent} = \mathsf{mod}\}$, and $\{o \mid \mathsf{mod} \in o.\mathsf{comps}\}$. We call a set description *direct*, if any field access paths start at mod, and *indirect* otherwise. Thus, the set description $\{\mathsf{mod}\}$ is direct, and the other two above are indirect. Indirect set descriptions turn out to be undesirable in practice, since they give rise to proof obligations concerning indirectly described objects, which are often too difficult for the automated theorem prover.

We shall attempt to transform the four cases of indirect sets in our example into direct ones. We start with $\mathcal{D}_S(\mathsf{parent}, \mathsf{Inv3})$, which specifies that modification of parent of an object mod may break invariant Inv3 for those objects which contain mod in their comps. Recall however that the "structural" invariant Inv3 guarantees that for any o', if $o' \in o.\mathsf{comps}$ then $o'.\mathsf{parent} = o$. Therefore, we can conclude that *if* an object o satisfies Inv3, then $o \in \{o \mid \mathsf{mod} \in o.\mathsf{comps}\} \Rightarrow o \in \{\mathsf{mod}.\mathsf{parent}\}$; the latter set is direct. Since the definition of admissibility allows us to assume that the concerned invariant (in this case Inv3) holds, applying the invariant's definition to the set description does not affect admissibility:

Proposition 2. *Suppose $\mathcal{D}$ and $\mathcal{D}'$ are concerns-descriptions, and that for all fields* f *and invariants* I*, it holds that $\mathcal{D}(\mathsf{f}, \mathsf{I}){\downarrow_2} = \mathcal{D}'(\mathsf{f}, \mathsf{I}){\downarrow_2}$ and we can prove for arbitrary o that if $P_{\mathsf{I}}(o)$ holds and $o \in \mathcal{D}(\mathsf{f}, \mathsf{I}){\downarrow_1}$ then $o \in \mathcal{D}'(\mathsf{f}, \mathsf{I}){\downarrow_1}$. Then, if $\mathcal{D}$ is admissible then $\mathcal{D}'$ is admissible.*

Using this proposition, we can take a set description from a concerns-description known to be admissible, and rewrite it using the definition of the invariant it is concerned with. Admissibility of the resulting concerns description is guaranteed to be preserved. In particular, for the Composite we can replace the concerns-description for parent and Inv3 as follows:

$\mathcal{D}(\mathsf{parent}, \mathsf{Inv3}) = (\{\mathsf{mod.parent}\}, \emptyset)$
Similarly, we can replace the next two indirect set descriptions with the following:
$\mathcal{D}(\mathsf{comps}, \mathsf{Inv4}) = (\{o \in \mathsf{mod.comps}\}, \emptyset)$
$\mathcal{D}(\mathsf{count}, \mathsf{Inv4}) = (\{o \in \mathsf{mod.comps}\}, \emptyset)$

This leaves us now with one remaining indirect set description:
$\mathcal{D}(\mathsf{total}, \mathsf{Inv2}) = (\{\mathsf{mod}\} \cup \{o \mid \mathsf{mod} \in o.\mathsf{comps}\}, \emptyset)$
This set comprehension is the same as for the first case, therefore if we could assume that $\mathsf{Inv3}$ held for all objects in the set, we could rewrite the set into the direct form, $\{\mathsf{mod.parent}\}$ as before. In this case, Proposition 2 does not apply, because we wish to use a different invariant $\mathsf{Inv3}$ to rewrite the set description for $\mathsf{Inv2}$. In order to justify the desired rewriting of the set comprehension, we can instead make use of the supporting invariants, and explicitly mark that the correctness of the new set description depends on $\mathsf{Inv3}$ holding, i.e., we define
$\mathcal{D}(\mathsf{total}, \mathsf{Inv2}) = (\{\mathsf{mod}\} \cup \{o \mid \mathsf{mod.parent}\}, \{\mathsf{Inv3}\})$
This can be understood at an intuitive level also: We can be sure that the objects whose invariants $\mathsf{Inv2}$ are affected by modifying the total of mod are (at most) the objects mod and $\mathsf{mod.parent}$ *only if* we can be sure that mod cannot be in the comps of any other object. This is what invariant $\mathsf{Inv3}$ guarantees. We can generalise the process we applied above with the following result:

Proposition 3. *Suppose $\mathcal{D}$ is an admissible concerns-description, and $\mathcal{D}'$ is a concerns-description identical to $\mathcal{D}$ except for the definition of $\mathcal{D}(\mathsf{f}, \mathsf{I})$ for some particular f and I. Suppose further that for some invariant J we have $\mathcal{D}'(\mathsf{f}, \mathsf{I})\downarrow_2 = \mathcal{D}(\mathsf{f}, \mathsf{I})\downarrow_2 \cup \{\mathsf{J}\}$. Then, if by assuming that $\forall o, P_{\mathsf{J}}(o)$ holds, we can prove that $\mathcal{D}(\mathsf{f}, \mathsf{I})\downarrow_1 \subseteq \mathcal{D}'(\mathsf{f}, \mathsf{I})\downarrow_1$, then $\mathcal{D}'$ is admissible.*

Using this proposition, we can take a set description from a concerns description known to be admissible, and rewrite it using the definition of any invariant we like (adding the invariant to the supporting invariants). Admissibility of the resulting concerns-description is guaranteed to be preserved. However, in order for a verification technique based on the resulting concerns-description to be sound, we require a mechanism for guaranteeing that supporting invariants will hold when required. For example, in case of the $\mathsf{Composite}$, we require some way of ensuring that whenever we wish to use $\mathcal{D}_S(\mathsf{total}, \mathsf{Inv2})\downarrow_1$, the condition $\forall o :: P_{\mathsf{Inv3}}(o)$ holds. For this reason, we need a way of treating the invariant $\mathsf{Inv3}$ in some special fashion. We recall that the invariants $\mathsf{Inv3}$ and $\mathsf{Inv4}$ were introduced to make explicit the inverse relationship between $\mathsf{components}$ and parent, which is implicitly intended in the implementation. As such, we expect these invariants to hold *almost* all of the time. The only reason the invariants ever need to be broken is that it is impossible to simultaneously update the necessary fields to keep the implementation of this relationship consistent. For this reason, the invariant semantics of Definition 1 seems too coarse-grained for these invariants, since it allows them to be broken for arbitrarily long code fragments (so long as no method boundaries are reached), whereas in fact they are only required to be broken for a handful of consecutive statements at a time.

Using this observation, we introduce a refinement of our treatment of invariants. The idea is to allow some invariants to be declared as more fundamental, and to only allow these invariants to be broken for short and prescribed sections of the code. A scoped declaration unreliable is used to specify that certain named invariants may possibly be violated for the duration of the scope (which is expected to enclose only a brief fragment of the code). This follows the intuition behind why these "structural invariants" are broken at all - it is just while the necessary field updates can all be made, to modify the intended parent-components relation.

Definition 5 (Structural Invariants and Unreliable Declarations). *The keyword* structural *may be placed before an invariant declaration, to mark the invariant as a* structural invariant. *By default, invariants are not structural.*

1. *Concerns-descriptions $\mathcal{D}$ are restricted to only allow structural invariants to be mentioned in the supporting invariants.*
2. *A scoped construct* unreliable: $\mathsf{I}_1, \mathsf{I}_2, \ldots, \mathsf{I}_n\{..\}$*, may be placed around any sequence of statements which do not contain any method calls (specifying which structural invariants may possibly be broken within the scope).*
3. *Programs are restricted as follows: for any field update* $\mathsf{e.f} = \mathsf{e}'$*, and for any invariant* I*, if* $\mathcal{D}(\mathsf{f},\mathsf{I})\downarrow_1[\mathsf{e}/\mathsf{mod}]$ *is non-empty, then the field update* must not *occur within an* unreliable *declaration which names any (structural) invariants* I' *in* $\mathcal{D}(\mathsf{f},\mathsf{I})\downarrow_2$*. Additionally, if* I *is itself a* structural *invariant, then the field update* must *occur within an* unreliable *block declaring* I*.*
4. *Structural invariants may not be mentioned in* broken *declarations.*

Note that the restrictions in the latter two points above do not introduce extra proof obligations for the verification process, since they can be guaranteed by syntactic checks on the program code.

Intuitively, this approach guarantees that structural invariants can only be violated within unreliable blocks which explicitly declare that they might be, while structural invariants may be *depended* on to accurately predict the concerns of a field update only outside the scope of such blocks. Furthermore, any structural invariants violated within an unreliable block should be re-established by the end of the block[4]. From a practical perspective, the burden of determining which objects' invariants are "concerned" with a field update can be completely lifted from the prover - not only are supporting invariants used to precisely identify which objects should be considered, but the validity of the supporting invariants is guaranteed by purely syntactic means.

3.3 Verification Technique

We say that an object's invariant is *vulnerable* at some point in the code, if we have no guarantee that it holds at that point. We calculate vulnerable invariants

[4] Our unreliable blocks described are similar to the expose blocks used in the Spec♯ methodology [3], but are simpler since they only mention invariants by name, rather than distinguishing them for particular objects.

based on the concerns-descriptions $\mathcal{D}$. Namely, for an update to $\mathsf{r.f}$, and invariant I, the set $\mathcal{D}(\mathsf{f}, \mathsf{I})\downarrow_1[r/\mathsf{mod}]$ gives a conservative approximation of the vulnerable invariants. For sequences of statements we accumulate the vulnerable invariants for each statement, in a similar fashion to standard static code analysis techniques. [5] For conditional branches we accumulate the effects of each branch.[6] Finally, according to the invariant semantics, after a method finishes executing all invariants must hold, and so it is justified after a call to "reset" the vulnerable set to empty.

Definition 6 (Vulnerable Invariants). *At any program point, the* vulnerable invariants *are represented by a map $\mathcal{V}$ from invariant names to descriptions of sets of references (denoting which objects may possibly not satisfy the invariant). It is computed as follows.*

1. *At the start of a method body, the vulnerable invariants are exactly those declared by the method's* broken *constructs (if any).*
2. *After a field assignment* $\mathsf{e.f} = \mathsf{e}'$*, if $\mathcal{V}$ describes the vulnerable invariants* before *the assignment, then the vulnerable invariants* after *the assignment, $\mathcal{V}'$ are defined for each invariant* I*, by:* $\mathcal{V}'(\mathsf{I}) = \mathcal{V}(\mathsf{I}) \cup \mathcal{D}(\mathsf{f}, \mathsf{I})\downarrow_1[\mathsf{e}/\mathsf{mod}]$.[7]
3. *After a conditional statement, the vulnerable invariants for each invariant is the union of those at the end of each branch.*
4. *After the end of an* unreliable *block, for each (structural) invariant* I *named by the block and not named by a further enclosing* unreliable *block,* $\mathcal{V}(\mathsf{I})$ *is empty. For all other invariants,* $\mathcal{V}(\mathsf{I})$ *is as it was at the end of the block.*
5. *After a method call,* $\mathcal{V}(\mathsf{I})$ *is empty for all invariants* I.

Note that we allow for the possibility of nesting unreliable blocks within each other. While we don't require this feature for our specification of the Composite, it could add extra flexibility in a setting where several structural invariants are mutually dependent - in this case it may be useful to accurately reflect the situation when some structural invariants are re-established before others by closing one block and leaving another open.

Our verification technique allows us to make assumptions about the validity of invariants and imposes proof obligations for invariants as follows:

Definition 7 (Considerate Verification Technique). *Given a program annotated with specifications, invariants, an (admissible) concern-description $\mathcal{D}$ and* unreliable *blocks, our methodology handles invariants as follows:*

1. *At the start of a method body, all invariants of all objects may be assumed to hold, except those explicitly declared as* broken *in the method specification.*

[5] In fact, the meaning of the set descriptions may be affected by subsequent field updates. We cater for this by recording copies of the symbolic heap, and writing assert and assume statements in terms of these copies, cf. [1]

[6] For simplicity we do not handle loops here, but believe that they can be handled by suitably extending the usual loop-invariant-based approach from Hoare Logic.

[7] Recall that we are eliding details of how to handle field updates which change the meaning of the vulnerable invariants recorded so far.

2. *Before call to a method* m*, for every invariant* I*, if S is the set of expressions e for which* $\mathsf{I}(e)$ *is mentioned in a* broken *declaration of* m*, then* $\forall o, o \in \mathcal{V}(\mathsf{I}) \wedge o \notin S \Rightarrow P_{\mathsf{I}}(o)$ *must be proven.*
3. *After a method call, all the invariants of all objects may be assumed to hold.*
4. *At the end-point of an* unreliable *block, for every invariant* I *declared in the block but not in an enclosing such block,* $\forall o, o \in \mathcal{V}(\mathsf{I}) \Rightarrow P_{\mathsf{I}}(o)$ *must be proved.*
5. *At the end of a method body, for every invariant* I*,* $\forall o, o \in \mathcal{V}(\mathsf{I}) \Rightarrow P_{\mathsf{I}}(o)$ *must be proven.*

Proposition 4. *The Considerate Verification Technique is sound.*

Proof sketch. We first show as an easy lemma that the vulnerable set for a structural invariant I is empty at all program points which are not inside an unreliable block declaring I.

This allows us to prove that any invariants which are broken by any field assignment fall within the described vulnerable set, as follows. Because we assume invariants only depend on the heap via field accesses, we know that if $\mathsf{I}(o)$ holds in heap h, but does not hold in heap h', and h' differs from h only in the value of $o'.f$, then there exist fields f_1, ... f_n, such that $o.\mathsf{f}_1...\mathsf{f}_n.\mathsf{f}$ appears in $P_I(o)$, and $o.\mathsf{f}_1...\mathsf{f}_n = o'$ in h. By the previous lemma, we know that for any structural invariant $\mathsf{I}' \in \mathcal{D}(\mathsf{f}, I)\downarrow_2$, it is safe to assume $\forall o', P_{\mathsf{I}'} o'$ holds. By definition 3, we obtain that $o \in \mathcal{D}(\mathsf{f}, I)\downarrow_1[o.\mathsf{f}_1...\mathsf{f}_n/\mathsf{mod}]$. The latter set corresponds in h to $\mathcal{D}(\mathsf{f}, I)[o'/\mathsf{mod}]$, which is the set added to the vulnerable invariants.

We can now show that assuming that all methods have been checked according to Def. 7, then execution preserves the property that any invariants which do not hold are within those calculated to be vulnerable according to Def. 6. This can be shown by induction on the execution.

At all point where our invariant semantics (Def. 1) specifies that invariants must hold, our technique imposes proof obligations to show that all required invariants which are also vulnerable, are shown to hold. Therefore, by the above, no required invariants can be false at these points.

3.4 Verification of the Composite Pattern

For the Composite code, we use the improved concerns-description developed earlier in the paper, which we recall here in full, for reference:

$$\begin{array}{llll}
\mathcal{D}(\mathsf{parent}, \mathsf{Inv3}) & = (\{\mathsf{mod.parent}\}, \emptyset) & \mathcal{D}(\mathsf{parent}, \mathsf{Inv4}) & = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}(\mathsf{comps}, \mathsf{Inv2}) & = (\{\mathsf{mod}\}, \emptyset) & \mathcal{D}(\mathsf{comps}, \mathsf{Inv3}) & = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}(\mathsf{comps}, \mathsf{Inv4}) & = (\{o \mid o \in \mathsf{mod.comps}\}, \emptyset) & \mathcal{D}(\mathsf{comps}, \mathsf{Inv5}) & = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}(\mathsf{count}, \mathsf{Inv1}) & = (\{\mathsf{mod}\}, \emptyset) & \mathcal{D}(\mathsf{count}, \mathsf{Inv2}) & = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}(\mathsf{count}, \mathsf{Inv3}) & = (\{\mathsf{mod}\}, \emptyset) & \mathcal{D}(\mathsf{count}, \mathsf{Inv4}) & = (\{o \mid o \in \mathsf{mod.comps}\}, \emptyset) \\
\mathcal{D}(\mathsf{count}, \mathsf{Inv5}) & = (\{\mathsf{mod}\}, \emptyset) & \mathcal{D}(\mathsf{total}, \mathsf{Inv1}) & = (\{\mathsf{mod}\}, \emptyset) \\
\mathcal{D}(\mathsf{total}, \mathsf{Inv2}) & = (\{\mathsf{mod}, \mathsf{mod.parent}\}, \{\mathsf{Inv3}\}) & & \\
\mathcal{D}(\mathsf{f}, \mathsf{I}) & = (\emptyset, \emptyset)\ \textit{otherwise} & &
\end{array}$$

We consider the invariants Inv3,Inv4 and Inv5 to be structural, and place an unreliable block around the three assignment statements in the add method which temporarily violate these invariants. Def. 5 requires that a total field is not modified within such a block - this is indeed the case. Using the concerns-description, we analyse the code to predict vulnerable invariants at each point, and generate proof obligations according to Def. 7. Figure 3 shows the complete code, including assume/assert statements which encode the proof obligations. Note that these statements are exactly as specified by Def. 7 - no additional manual assertions are required, and no further assert/assume statements need to be provided. We map this specification to Boogie2, which then passes the proof obligations to the Z3 automatic theorem prover for verification.

Verification of the Boogie2 code opens up a low-level problem concerning the prover's treatment of quantifiers. In particular, some control needs to be imposed to stop the prover from taking arbitrarily many (mostly irrelevant) instances of a quantifier formula it "knows", and thus looping forever. The very strong assumptions made by our methodology at the beginning of a method body and after a method call, can actually *negatively* impact the performance of the prover, if not controlled. This problem is generic to the use of quantifiers with the Z3 prover, and can be tackled by using *triggers* [6, 13, 14], a mechanism which restricts the situations under which the prover instantiates quantified formulae. We do not go into detail here; however, we have developed a methodology for defining triggers for the formulae concerned with our methodology, which we will describe in future work. Our Boogie2 code [1] uses triggers.

Verification of the Boogie2 code succeeds, in approximately six seconds. Interestingly, if one takes the simplest concerns-description instead (which still employs indirect set comprehensions), the resulting specification does not verify. Therefore, the improvements introduced by Propositions 2 and 3 are essential for our approach to be practical. However, the current need to annotate the code with unreliable declarations and concerns-descriptions seems to place an extra burden to the user; we next consider how to alleviate it.

3.5 Automation of our Technique

We now explain how the various aspects of our methodology could be supported by automatic tools.

Determine Concerns-Description: A tool can straightforwardly derive the simplest concerns-description $\mathcal{D}'$ (Def. 4). Next, any declared structural invariants expressing "inverse" relationships (e.g., $\mathsf{o.components[i].parent} = \mathsf{o}$) can be used to rewrite any indirect set descriptions. Given a set description of the form $\{o \mid o.\mathsf{f}_1 \ldots .\mathsf{f}_{m-1}.\mathsf{f}_\mathsf{m} = \mathsf{mod}.\mathsf{g}_1 \ldots .\mathsf{g}_\mathsf{n}\}$ a structural invariant of the form $\mathsf{o}'.\mathsf{f}_\mathsf{m}.\mathsf{h} = o'$ (i.e., declaring an inverse relationship for the field f_m) should be sought. The set description can then be rewritten to $\{o \mid o.\mathsf{f}_1 \ldots .\mathsf{f}_{m-1} = \mathsf{mod}.\mathsf{g}_1 \ldots .\mathsf{g}_\mathsf{n}.\mathsf{h}\}$ in which the length of the "indirect" field access from o has been reduced. To preserve admissibility, a structural invariant used to rewrite the set must be recorded in the supporting invariants (cf. Prop. 3), unless it is the same invariant as the one being described by $\mathcal{D}$ (c.f. Prop. 2). This process of rewriting the set can be repeated until the length of the indirect field access is zero, at which point the set comprehension describes precisely one object, and the set can be

```
// requires : c ≠ null;
// requires : c.parent = null;
public void add(Composite c) {
  assume ∀ o :: Inv1(o) ∧ Inv2(o) ∧ Inv3(o) ∧ Inv4(o) ∧ Inv5(o);

  // unreliable : Inv3, Inv4, Inv5 {
  this.comps[this.count] = c;
  this.count = this.count + 1;
  c.parent = this;
  assert Inv3(this) ∧ Inv3(c.parent);
  assert ∀ o :: o=c ∨ (∃_{0≤i≤this.count} i:int :: this.comps[i] = o) ⇒ Inv4(o);
  assert Inv5(this);
  // }

  assert Inv1(this);
  assert this ≠ this ⇒ Inv2(this); // trivial − by "broken" declaration
  addToTotal(this,c.total);
  assume ∀ o :: Inv1(o)∧Inv2(o)∧Inv3(o)∧Inv4(o)∧Inv5(o);

}

// broken: Inv2(this)
// requires : this.total + p = 1 + Σ_{0≤i<count} comps[i].total
private void addToTotal(int p) {
{
  assume ∀ o :: Inv1(o)∧Inv3(o)∧Inv4(o)∧Inv5(o);
  assume ∀ o :: o≠ this ⇒ Inv2(o);

  this.total = this.total + p;
  if(parent != null) {

     assert Inv1(this);
     assert ∀ o :: (o=this ∨ o=this.parent) ∧ o≠ this.parent ⇒ Inv2(o);
     parent.addToTotal(p);
     assume ∀ o :: Inv1(o)∧Inv2(o)∧Inv3(o)∧Inv4(o)∧Inv5(o);
  }

  assert Inv1(this);
  assert ∀ o :: (o=this ∨ o=this.parent) ⇒ Inv2(o);
}
```

Fig. 3. Proof Obligations for the Composite

made direct. In practice, invariants of the desired kind tend to exist in "considerate" implementations, since the inverse field references are required for the implementation to be able to notify objects appropriately (e.g., the parent field in the Composite). However, if at any point a suitable structural invariant cannot be found, either an error can be reported to the user (suggesting that further structural invariants may need to be specified), or a warning could be given, and the verification optimistically continued using the indirect description.

Introduce unreliable blocks: One can automatically infer when an unreliable block needs to begin, and which invariants need to be named, by using $\mathcal{D}$ to identify the points in the code at which structural invariants may be invalidated. Inferring where to *end* the unreliable blocks is more challenging, since we need to "guess" how soon we re-establish these invariants. The simplest solution is to be lazy, and leave the block open until these invariants are required to hold again, either because a method call, end of method or conditional block is reached, or because they appear in the supporting invariants of a concerns-description for a field update. In practice, this typically doesn't leave much scope for "laziness", and showing that the structural invariants are re-established at the derived point is not problematic. For example, in the Composite add method, the structural invariants must be re-established before addToTotal can be called.

Calculate proof obligations: The vulnerable invariants can be calculated (Def. 6) and corresponding assume/assert statements derived (Defs. 7 and 5).

4 Conclusions, Related and Future Work

We have proposed Considerate Reasoning, a specification/verification methodology based on object invariants, which, we claim, neatly reflects the natural argument of the implementation, and leads to succinct specifications. We have outlined soundness of the technique, described how its support could be automated, and applied it to specify the Composite pattern.

Our work is based on, and extends, that of Middelkoop et. al. [15]. Our concerns-descriptions add to their "coop-sets" the concept of supporting invariants; we introduced inference of admissible concerns-descriptions, structural invariants, unreliable blocks, and the application to Boogie2.

Several specifications of the Composite were proposed for SAVCBS 2008. For example, Jacobs et. al. [9] give a specification in separation logic, which expresses the decomposition of a tree-structure into different context-tree views from the viewpoint of the current receiver. The specification is not able to enforce invariants for all objects, and thus cannot guarantee preservation of the main invariant, Inv2, for *all* objects in the heap. It was machine-verified using VeriFast [8]. The verification was interactive, and required the manual addition of lemmas, and open/close and assert statements.

Bierhoff and Aldrich [5] present a specification using data groups, fractional permissions, type states, and explicitly marking the violation/re-establishing of invariants through unpack/pack statements. Permissions control state dependencies in invariants - essentially each object depending on certain state for its invariant must carry some permission to that state. The authors outline a manual verification, and discuss how a tool could infer unpack/pack statements.

More recently, Rosenberg et. al. [21] give a specification of the Composite using regional logic [2]. They express an invariant semantics similar to ours, whereby they explicitly quantify over the set of all allocated objects, and require in the pre- and post- conditions of the methods the invariants of all objects to hold, except for those objects belonging to a further region (this corresponds to our broken declarations). They mapped the specification into Boogie2 and verified it in approx 6 secs. However, because their handling of invariants is explicit, rather than with a prescribed methodology, some guidance is needed for the verified, which takes the form of several lemmas, and manually annotating the Boogie code with several assume/assert statements.

All these specifications required significant technical development; this is reflected in their length. Conversely, we have tried to retain in the specification only those details which are essential and intuitive from the point of view of the programmer. Furthermore, verification of these specifications requires further work from the programmer, in that he needs to provide lemmas and insert further annotations into the code. Conversely, our methodology can be automated as we discussed earlier on; with our hypothetical tool, the programmer will only need to provide the 25 lines of code and specification shown in Fig. 2. On the other hand, our methodology does not deal with framing, whereas the above approaches address this issue.

In future work we will formalise and prove soundness of Considerate Reasoning, and will combine it with other methodologies supporting complementary programming patterns, as e.g., ownership-based methodologies. We will also address the framing problem, and investigate extending our work to more-general kinds of invariants and patterns in which *collections* of objects may be broken at a time.

We have considered the extension of our approach to concurrency. We propose a locking discipline based on the calculated vulnerable invariants (calculated per thread). Any object in the vulnerable invariants should be locked by the current thread. Correspondingly, objects can only be unlocked if all of their invariants which are vulnerable, can be shown to have been re-established. When applied to the Composite, this idea allows a hand-over-hand locking discipline which can handle many threads updating the tree structure concurrently. Formalising this idea and its extensions will be interesting future work.

Acknowledgments This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. We thank the anonymous reviewers for many insightful suggestions, and Arsenii Rudich, Claude Marché, Frank Piessens, Jan Smans, Ronald Middelkoop, Joseph Ruskiewicz and most particularly and specially Peter Müller, for many stimulating discussions.

References

1. Boogie 2 code. Available: `http://people.inf.ethz.ch/summersa/wiki/lib/exe/fetch.php?media=papers:boogie-composite.zip`.

2. Anindya Banerjee, David Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP'08*. Springer-Verlag, 2008.
3. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6):27–56, 2004.
4. M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, volume 3125 of *LNCS*. Springer, 2004.
5. Kevin Bierhof and Jonathan Aldrich. Permissions to specify the Composite Design Pattern. In *SAVCBS*, 2008.
6. David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, (52):2005.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
8. Bart Jacobs and Frank Piessens. The verifast program verifier. Technical report, Katholieke Universiteit Leuven, August 2008.
9. Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the Composite Pattern using Separation Logic. In *SAVCBS*, 2008.
10. Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, 2006.
11. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *FAC*, 19(2):159–189, 2007.
12. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP'04*, LNCS. Springer-Verlag, 2004.
13. K. Rustan M. Leino. This is Boogie 2. Available from `http://research.microsoft.com/en-us/um/people/leino/papers.html`.
14. K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order smt solvers. In *SAC'09*. ACM, 2009.
15. R. Middelkoop, C. Huizing, R. Kuiper, and E. J. Luit. Invariants for non-hierarchical object structures. *ENTCS*, 195:211–229, 2008.
16. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
17. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*. ACM Press, 2004.
18. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*. ACM Press, 2005.
19. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.
20. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE Computer Society, 2002.
21. Stan Rosenberg, Anindyia Banerjee, and David Naumann. Local Reasoning and Dynamic Framing for the Composite Pattern and its Clients. `http://www.cs.stevens.edu/~naumann/publications/RosenbergBanerjeeNaumann09a.pdf`.
22. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comp.*, 39(9):1175–1185, 1990.
23. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009*. Springer-Verlag, July 2009.
24. A. J. Summers, S. Drossopoulou, and P. Müller. The need for flexible Object Invariants. In *IWACO'09*, June 2009.