Mobius

Information Society Technologies

Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable 3.9

Report on Modular Verification

Due date of deliverable: 2009-08-31 (T0+48)

Actual submission date: 2009-09-05

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **ETH**

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	Contributed to Chapter
ETH	1, 2, 3, 4
IC	1, 2, 3

This document was written by Ádám Darvas (ETH), Sophia Drossopoulou (IC), Peter Müller (ETH), Arsenii Rudich (ETH), and Alexander Summers (IC).

Executive Summary:
Report on modular verification

This document summarises deliverable D3.9 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr`.

Following the goals of Task 3.4 to develop a practical verification technique for object invariants and frame properties that can handle interesting implementation patterns including inheritance, call-backs, recursive object structures, and concurrency, the Mobius consortium has made significant contributions to the area of modular verification. These contributions appear in diverse publication venues. This deliverable contains an overview of selected results and serves as a chart for Mobius contributions to this area, including pointers to specific publications for further details.

The following results are in the main focus of the deliverable:

- We have investigated the possible benefits of object invariants for software development, verification, and maintenance [22]. We found strong arguments that invariant protocols express programmers' intuitions, lead to better design, allow more succinct specifications and proofs, and allow the expression of properties which involve many objects in a localised manner. In particular, the resulting verification conditions can be made simpler and more modular through the use of invariant-based reasoning. We also found evidence that, even though encoding invariant protocols through method pre- and post-conditions is possible, such an encoding loses important information, and as a result makes specifications less explicit and program evolution more error-prone. Finally, we show that such encodings often cannot express properties over inaccessible objects, whereas an appropriate invariant protocol can handle them simply.

- We have developed the considerate specification/verification approach. It is a novel technique for the verification of object invariants, that explicitly mentions when object invariants are broken, and when these will be repaired [20]. The proposed technique was used to specify and verify the well known Composite design pattern. We obtained a succinct specification, which reflects the structure of the program. The verification was encoded in Boogie2 [12] and was relatively straightforward.

- We have developed the stereotypes-based approach to specification/verification of complex heap structures and their invariants [15]. The main specification primitive used in our approach for a heap structures description is a stereotype. On the one hand, stereotypes provide developers with a natural and lightweight way to describe heap structures, on the other hand, an automatic theorem prover can efficiently prove such specifications. To facilitate specification/verification of multi-object invariants on top of stereotypes, we introduced the notion of invariant state. Invariant states generalize the notions of object invariant and object frame. We used stereotypes and invariant states to verify a priority inheritance protocol [**?**].

- We have developed a technique to faithfully map modeling types to mathematical structures provided by the built-in libraries of theorem provers [7]. The technique formally checks if there is semantic correspondence between the specification of a modeling type and the structure to which the type is mapped. The approach enables reasoning about programs specified in terms of modeling types as well as facilitates writing better specifications for modeling types.

Contents

1 Introduction **5**

2 Object Invariants and Heap Verification **6**
2.1 Considerate programming 7
2.2 Stereotype-based verification 8
2.3 A Universe-Type-Based Verification Technique for Mutable Static Fields and Methods 10

3 Faithful Mapping of Model Classes to Mathematical Structures **11**

4 Conclusions **14**

Chapter 1

Introduction

This deliverable reports the final results on modular verification (Task 3.4). The deliverable summarizes publications and technical reports that have appeared elsewhere and includes contributions from the MOBIUS partners involved in Task 3.4, namely ETH and IC.

During the final stage of the project we concentrated our efforts on two aspects of modular verification. Firstly, we developed verification techniques for non-hierarchic object structures. Our aim was to develop a verification technique that provides the simplicity of ownership-based verification with the flexibility of dynamic frames and regions. Our work was inspired by four major existing verification techniques for object structures (dynamic frames [19], ownership [8], regions [1], separation logic [17]). Secondly, we continued our work on the well-definedness of specifications by extending it to modeling classes, which encode mathematical structures. Program verifiers map modeling classes to their underlying logics. Flaws in a modeling class or in the mapping easily lead to unsoundness and incompleteness. We developed a technique for finding such flaws.

Further development within the project. The following chapters describe various aspects of modular verification:

- Chapter 2 presents an overview of the considerate and stereotype-based approaches to specification/verification of complex heap structures and their invariants. Also, it briefly summarizes results of our investigation of possible benefits from object invariants for software development, verification, and maintenance.
- Chapter 3 gives an overview of the technique we developed for the faithful mapping of modeling classes to mathematical structures. A modeling class is an immutable class that contains only side-effect free methods. The technique allows one to specify the mapping of modeling classes and their methods to structures provided by theorem provers and their function symbols. The faithfulness of a mapping means semantical correspondence between the related classes and structures.

Chapter 2

Object Invariants and Heap Verification

The coupling of functionality and state is fundamental to the object-oriented programming paradigm. An object's state may change during program execution, and as a result the object's behavior may change. Objects behave properly if they are in a *consistent state*, and are *implicitly* expected by programmers to be in such a consistent state at most times of execution.

The pioneering work of Bertrand Meyer [**?**] suggested that the specification and verification of object-oriented programs should be based on the concept of *object invariant*, which describes the consistent states of an object. Object invariants provide a means of specification at the natural level of the object itself, typically in terms of its fields. Method pre- and post-conditions complement these object-level specifications by describing method behaviors in the traditional procedural programming style.

Conjoining invariants to pre- and post-conditions in method specifications can encode most of the invariant protocols, and allows for more flexibility: if it is intended that for some methods (e.g., helper methods) an invariant need not hold, then the predicate expressing the invariant need not be conjoined to the specification of that method. More generally, properties across several objects can be decomposed into weaker properties (via other predicate definitions), and so it is easy to express that an invariant partially holds. For these reasons, it has been suggested by Parkinson [**?**] that the object invariant is, as a fundamental principle for object-oriented verification, superfluous.

We argue that the object invariant plays an important role in the design, specification and verification of object-oriented programs.

We make three claims for the usefulness (and in some cases, necessity) of object invariants.

Invariant-based reasoning aids and improves software design. We believe that thinking in terms of invariants helps to come up with good software designs, and that a design can be made a more robust by following an invariant-based philosophy. Furthermore, we argue that if invariants are encoded into pre- and post-conditions, essential design information is lost, and its absence may lead to specification/design errors at the level of programmer specifications, and less practical specifications and proof obligations for automatic verification.

Software design can be exploited by invariant-based reasoning. We believe that for a verification approach to be both natural to a programmer and scalable to large programming projects, it is essential that the principles guiding the design of the code are reflected and exploited in specifying and verifying the code. Therefore, if code is designed with invariants in mind, then it is natural and useful to reflect this in the corresponding reasoning. In particular, invariants allow a verification analogue to the concept of delegation of responsibility, which is a fundamental aspect of object-oriented design philosophy.

Object invariants allow local reasoning about global properties. We argue that invariant protocols naturally express the code patterns where objects notify other objects whose properties they may have broken, and that object invariants allow the expression of these specification requirements in a local way. Furthermore, we argue that object invariants are the most natural way of maintaining properties

that depend on objects which are not accessible via field accesses, and supports a simpler and more practical verification of such properties.

2.1 Considerate programming

Invariants often describe not only constraints on the fields of a single object, but relationships between the states of collaborating objects. However, if an invariant depends on the state of some other objects as well as its own state, then the invariant may be broken while *other* objects are active. In order to handle this complexity, an *invariant protocol* is required, which determines at which points in a program the invariant of an object may be assumed to hold, and at which points it has to be established.

Müller et al. suggested an invariant protocol, whereby an object *owns* all objects on whose state its invariant depends, the ownership hierarchy is a tree, and whereby the invariant of an object can only be affected by its owner and its owned objects [14]. This protocol has subsequently been refined to deal with callbacks, visibility, dependency across objects with a common owner, and subclasses [14, **?**], dynamic ownership [**?**], or ownership types [**?**].

However, there are cases where the invariant dependency is not a tree, e.g., the Subject and Observer pattern, and where state-modifying methods do not navigate the tree from the top, e.g. the Composite, one of the classic design patterns in object-oriented languages [**?**]. It describes a tree-structure of objects in which a uniform interface is presented to the leaves of the tree and the *Composite* objects which can store a number of *components*. This example was used as the basis of the Challenge Problem at SAVCBS last year, and was also used as an example in the survey of open challenges for verification by Leavens et al. [**?**]. We give in Figure 2.1 a simplified version of the code presented there, in which we are concerned only with a *Composite* class, and not with issues regarding subclassing. We believe such issues to be orthogonal to the main challenge of this example, that regards the verification of the flexible updates which the data structure permits. We are still able to describe the intended tree-structure, since Composite objects need not have any components, and so can be used to represent the leaves of the tree as well as other nodes.

From the verification perspective, one of the main challenges is to preserve some property of the nodes over the whole data structure, which is historically a count of the number of nodes in the corresponding subtree. The major difficulty in verifying this property is that the data structure can be directly added to at any point, by simply calling a method on a *Composite* object. This is problematic for, e.g., ownership-based approaches to verification, since these typically guarantee encapsulation by requiring access to owned objects to be controlled in some way by the owning objects (which would correspond to a top-down traversal of the tree-structure before a modification could take place). Similarly, specifications of such patterns in separation logic typically involve the use of recursive predicates to describe properties over the data structure [16, 17], and such predicates are easier to unfold and fold from the root of the structure downwards.

To deal with this challenge considerate programming, a new approach to specification/verification of object invariant, was proposed [22, 20]. Considerate programming is an amalgamation of the ideas presented in [2, 13, **?**]. All these approaches support the notion of object invariant, whereby classes define invariants, and objects satisfy the invariants declared their class except for some specific points in a program.

When a field in object o is updated, it may affect the validity of invariants in the object o as well as in other objects. The notion of the set *vulnerable* to a field update is at the core of considerate programming. It describes the object invariants which are affected by a field update.

For example, according to Figure 2.1, the value o.total affects *invariant2* of o; therefore, we say that *invariant2* of o is *vulnerable* to updates of field total of object o. Notice, that the vulnerable set for updates of total of o also includes all objects which contain o in their field components. More formally, *invariant2* of o' is vulnerable to updates of field total of object o, for all objects o' such that o'.components[i] $= o$.

Considerate programming expects all objects to satisfy their invariants immediately before and immediately after all method calls, except if the invariant is explicitly indicated as broken. Thus a method which mentions *broken Inv1(o)* in its precondition but does not mention it in its postcondition, promises to repair the invariant *Inv1* on object o, while a method which mentions *broken Inv1(o)* in its postcondition but does

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

  // invariant1: 1 ≤ total ∧ 0 ≤ count
  // invariant2: total = 1 + ∑_{0≤i<count} components[i].total

  //@ requires: c.parent == null;
  public void addComponent(Component c) {
    // resize array if necessary
    components[count] = c;
    count++;
    c.parent = this;
    addToTotal(c.total);
  }

  //@ requires: 0≤p
  private void addToTotal(int p) {
    total += p;
    if(parent != null) {
      parent.addToTotal(p);
    }
  }
}
```

Figure 2.1: A single-class variant of the Composite pattern

not mention it in its precondition may break the invariant Inv1 on object o.

As in approach proposed by Middelkoop et al. [**?**], the identification of vulnerable sets precedes the verification of the code. At the end of a method body the verification has to establish the invariants from the corresponding vulnerable sets which are not included in the post-condition's broken set.

2.2 Stereotype-based verification

Another challenge for verification is complicated heap structures. Typically, the description of heap structures requires intensive usage of transitive closure or similar constructs, e.g., recursive functions or inductive definitions. Such constructs provide a transparent and natural way for heap structure description and reasoning. Unfortunately, an ATP (automatic theorem prover) cannot adequately handle such constructs. Usually it is possible to describe such constructs, but their applications fairly often results in infinite looping.

One possible way to address this problem is regional logic [1]. The main idea behind regional logic is the explicit maintenance of a finite set of objects of interest. Such sets of objects are called *regions* and maintained through ghost fields. For example, object o is an element of a tree structure, C is the set of children of o, and p is the parent of o. If, for specification purposes, we need the set of descendants and ancestors of o, we can introduce regions D and A. We use these regions instead of the transitive closure of C and p. An advantage of the approach is that we don't need transitive closure anymore, and thus the ATP can handle specifications automatically and avoid potential infinite looping.

The price that is paid for the extra flexibility is specification overhead. Whenever a tuple is added to or removed from a relation, the sets that encode the relation's transitive closure have to be updated explicitly. For example, if we want to add o' as a child of o, the following sequence of updates should be performed:

- $o'.p := o$
- $o.C := o.C \cup \{o'\}$
- $\forall o'' \in o.A \cup \{o\} \ : \ o''.D := o''.D \cup o'.D \cup \{o'\}$
- $\forall o'' \in o'.D \cup \{o'\} \ : \ o''.A := o''.A \cup o.A \cup \{o\}$

To deal with this specification overhead we developed the notion of stereotype. The main idea behind stereotypes is the approximation of a complicated property by a set of simple properties. If an approximation is precise enough then the ATP can be used for automatic verification of such properties. From the perspective of heap verification, a stereotype is an aspect of an objects' participation in heap structures. We formally define stereotype as triples of stereotype elements, stereotype invariants, and stereotype operations.

Stereotype elements define essential notions of the approximated heap structure. For example, the elements of the stereotype tree are parent, root of the tree, children, ancestors, and descendants. Stereotype invariants define the semantics of stereotype elements. For example, one of tree stereotype invariants is $\forall o, o' \ : \ o'.p = o \Leftrightarrow o' \in o.C$. It establishes a relation between the parent and children, namely o' is a child of o iff o is a parent of o'. Stereotype operations change the values of stereotype elements of objects participating in the stereotype in such a way that the stereotype invariants are preserved. Since stereotype operations change only stereotype elements, they act as ghost updates. Examples of tree stereotype operations are addition and removal of an element of a tree stereotype.

An advantage of the stereotype based approach is that it is a purely abstract mechanism and it doesn't restrict the real heap in any way. For example, if an object participates in the list stereotype, it doesn't mean that it has to participate in a real list heap structure, or that it has to have fields *next* or *previous*. Therefore the same stereotypes can be used for the specification of different heap structures. For example, the stereotype list can be used to specify singly linked or doubly linked, cyclic or acyclic lists. On the other hand, it is quite typical for the real applications that an object participates in several stereotypes simultaneously. In such a case, it is enough to provide an additional invariant to specify when and in which stereotypes an object participates. Existing stereotypes can be easily reused and combined together.

To facilitate stereotype-based verification, we generalized the notion of object invariant to a notion of invariant state. In ownership-based verification, it is assumed that an object invariant depends only on the part of the heap which is owned (encapsulated) by the object. This guarantees that the invariant can be broken only by the object. From the practical perspective this restriction is too strong. It is typical that the invariants depend on heap locations that can be changed by other objects. In this case, the invariant is broken and essential information about the object state is lost. It is not clear which methods need to be involved to restore the object invariant. One possible way to deal with this problem and make objects suitable for such external heap changes is to introduce invariant states. It is usually assumed that an invariant is either valid or invalid, which can be treated like that the invariants' state takes values from the Boolean domain. Instead, we assume that the class developer provides a description of the invariant states' values domain. Together with the domain description, the developer should provide and prove transitions rules. These rules describe how the objects' invariant state changes as a reaction to external heap changes. The rules formalize the developer's expectations about how the environment of the object can be changed during the program execution. If the provided rules are detailed enough, they can be used to restore the object invariant after external changes.

To evaluate the flexibility and efficiency of the stereotype based approach, we applied it for verification/specification of a priority inheritance protocol [**?**]. The resulting specifications have a clear structure and can be automatically verified. Most of the specifications in the heap structures of the priority inheritance protocol are a combination of three stereotypes: tree, acyclic list, and cyclic list. The combination and adjustment of these stereotypes requires only an insignificant amount of additional specifications.

2.3 A Universe-Type-Based Verification Technique for Mutable Static Fields and Methods

Another direction of our research in the area of object invariants verification is the extension of our verification technique for mutable static fields and methods [21]. We developed two extensions of our basic technique, aimed at improving usability [23].

Firstly, we introduce a new universe annotation *strong any* to allow safe callbacks between ownership trees, whereby trees may be visited not at the top, but at the point where a previous visit had left off. The intended semantics of *strong any* is that the object referred to is guaranteed to be safe to make a callback on; we permit all calls on *strong any* references. We consider *strong any* to be a specialisation of *any*, and a *strong any* reference can be upcast to *any* if desired (losing the special semantics associated with a *strong any* reference type), but downcasts are not permitted. Our *strong any* annotation expresses that an object's invariant may be assumed to hold without constraining the object's position in the ownership topology. In that sense, it is similar to a Spec# [**?**] specification stating that an object is peer-consistent. However, since we use a type system, we have to over-approximate the consistency of objects and enforce a stronger system invariant.

Secondly, we refined our heap topology with a notion of levels, which stratify the heap and provide modularity with regard to library classes and the required effects annotations. We aimed to abstract away from the details of previously written classes (such as library classes). The key observation is that library classes will never (directly) call static methods on newer classes being verified, and so the possibility of dangerous callbacks is naturally reduced when calling classes on a lower level. This extension allows for fewer (and more modular) effect annotations.

Chapter 3

Faithful Mapping of Model Classes to Mathematical Structures

A common way of writing abstract specifications is to specify implementations in terms of well-known mathematical structures, such as sets and relations. This technique is applied, for instance, in VDM, Larch, and OCL. While these approaches describe the mathematical structures in a language that is different from the underlying programming language, the one-tiered JML simplifies the development of specifications by describing the structures in an object-oriented manner through modeling types [4], commonly referred to as *modeling classes*.

A model class is immutable and contains only pure methods, that is, methods that are side-effect free. These methods provide an interface to the mathematical structure that the class represents. While model classes are useful for specification purposes, verifiers have to encode model classes in the underlying theorem prover.

Encoding the methods of a model class by uninterpreted functional symbols in the underlying logic is possible, but not optimal, mainly for the following three reasons. (1) The tactics of a theorem prover are optimized for theories that are part of the prover's theory-library, and not for the encoding of the methods of model classes. (2) It is difficult to ensure consistency of such encodings, in particular, in the presence of recursive specifications [18]. (3) The encoding technique can ensure consistency of specifications, but not their semantical correctness. While we cannot do better for pure methods that are written for a given domain, the situation is different for model classes. For instance, we have a good understanding about the semantics of the method `union` in the model class that represents mathematical sets.

Therefore, previous work [3, 10, 11] proposes to map model classes and their pure methods directly to theories of the underlying theorem prover. This is possible because model classes, are in fact, very similar to mathematical structures. Their objects are immutable, their operations are side-effect free, and equality is based on their state rather than object identity. Therefore, instances of model classes behave like mathematical values, rather than heap-allocated objects. This view greatly simplifies reasoning about model classes.

Figure 3.1 shows the use of model class `JMLObjectSet` for the specification of class `SingletonSet`. Model class `JMLObjectSet` is a prefabricated class that encodes a mathematical set of objects through its pure methods. In order to use the model class in the specification of class `SingletonSet`, model field `_set` is declared. The field represents the abstraction of an instance of type `SingletonSet` as specified by the `represents` clause: a singleton set containing the object referenced by field `value`. Method `setValue` is specified in terms of the model field and `JMLObjectSet`'s pure method `has`, which checks for set membership.

Let us see how one would prove the postcondition of method `setValue` in store h. Due to our semantical understanding of method `has`, expression `_set.has(o)` can be mapped to $o \in h(\mathbf{this}.\_set)$. To determine the value of location `this._set` in store h, we map the `represent` clause of field `_set` to the singleton set, yielding term $\{h(\texttt{this.value})\}$. Given the assignment in the body of the method, we obtain the following proof obligation for `setValue`: $o \in \{o\}$, which is trivial to prove.

```
class SingletonSet {
    Object value;
    model JMLObjectSet _set;
    represents _set <- new JMLObjectSet(value);

    void setValue(nullable Object o)
      ensures  _set.has(o);
    { value = o; }

    // other constructors and methods omitted
}
```

Figure 3.1: Specifying `SingletonSet` using model class `JMLObjectSet`

The proof obligation above is considerably better suited for verification than the proof obligation $\widehat{has}(\widehat{JMLObjectSet}(o,h),o,h)$ that one would obtain by encoding the expression with the use of uninterpreted function symbols.[1]

Previous work discusses only the mapping of method signatures, but ignores their contracts. With this approach, the meaning of `has` is given by the definition of symbol $\in$ of the underlying theorem prover, and not by the contract of `has`. This is problematic if there is a mismatch between the contract and the semantics of the operation given by the theorem prover. Static program verifiers might produce results that come unexpected for programmers who rely on the model class contract. The results may also vary between different theorem provers, which define certain operations slightly differently (for instance, division by zero yields 0 in Isabelle, while it is not admissible in PVS). Moreover, the result of runtime assertion checking might differ from that of static verification if the model class implementation used by the runtime assertion checker is based on the model class contract.

The main contribution of our work is a technique for proving that the mapping of a model class to a mathematical structure defined by the theorem prover is faithful, that is, the model class and the structure indeed correspond to each other in their properties.

Our technique allows one to prove faithfulness of a mapping in three steps. In the first step, one has to specify to what structure a given model class is mapped, and to which function symbols of the structure are the methods of the model class mapped. In the second step, one has to prove that everything that can be proved using the contracts of the model class can also be proved using the corresponding structure of the theorem prover. Completing this step successfully gives us the guarantee that the specification of the model class is consistent (that is, free from contradictions), provided that the structure is consistent. In the third step, one has to prove that the properties of the structure to which the model class is mapped can be derived from the properties of the model class. In this step, user-specified mappings need to be "reversed". Completing the third step successfully gives us the guarantee that the specification of the model class is complete relative to the target structure. Once faithfulness of a model class has been proven, the mapping can be used for program verification without worrying about semantic discrepancies.

Our approach leads to important results beyond semantical correspondence and simplified reasoning. Model class contracts are complex and can easily get inconsistent, which can lead to unsound reasoning. Showing that a model class can be mapped consistently to a mathematical structure proves that the model class contract itself is (relatively) consistent. In fact, one of our case studies discovered an inconsistent specification in class `JMLObjectSet`, which is one of the most basic model classes of JML's model library.

Proving that the specification of a model class is complete relative to a theory gives some level of confidence that the model class contains all important properties. Failing to prove completeness is typically

[1] Our encoding denotes a method call to method `m` by a function application $\widehat{m}$.

a sign that some properties are missing. Our case studies discovered such missing specifications.

These points show that proving faithfulness of mappings helps in writing better specifications for model classes by making them consistent and complete. Our approach can also be used to identify redundant parts of specifications as well as to check whether specifications marked as redundant are indeed derivable from non-redundant specifications. These capabilities further improve the quality of model-class specifications.

Chapter 4

Conclusions

This chapter summarises the results and briefly discusses their impact on modular verification.

Results There is rich evidence that we have achieved task goals :

- We presented a unified framework that describes verification techniques for object invariants in terms of seven parameters and separates verification concerns from those of the underlying type system. We identified sufficient conditions on the framework parameters that guarantee soundness, and we proved a universal soundness theorem. The result have been presented at the European Conference on Object-Oriented Programming (ECOO 2008) [9].

- We have outlined a verification technique based on VT, catering for static fields, methods, and invariants. In the process, we extended the usual heap topology of ownership types, and tackled potential callbacks through a combination of effects, levels, and the owner-as-modifier discipline. The result have been presented at the Workshop on Formal Techniques for Java-like Programs (FTfJP 2008) [21].

- We have investigated the usefulness of invariants for different aspects of object-oriented verification. Invariants can express the design intentions of a programmer, and provide an implicit contract for future subclasses which cannot be captured directly by method specifications. Furthermore, the ability to separate the concerns of the invariant from the method specifications themselves makes clear the division of responsibility in the verification, and avoids unnecessarily cluttering the client specifications with properties guaranteed by code they cannot see. We have also argued that invariants can help reflect properties inherent in software design, and that this can help keep the verification argument closer to the original intentions of the implementation.The results have been presented at the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO 2009) [22].

- We have developed the considerate specification/verification approach. The proposed technique was used to specify and verify the well-known Composite design pattern. We believe that provided specification of the Composite example is natural to an object-oriented programmer, and the approach is manageable from the verification point of view. Our encoding in Boogie2 sketches how it can be embedded into a verification tool. As future work we hope to develop further techniques to support verification efforts which can be flexible, natural, and appealing to programmers. To this end, we believe that a verification approach which closely reflects the design decisions of programmers is essential, and that as we have explained, invariants will play a vital role.

- Challenged by verification of complex design patterns, such as the Composite and the priority inheritance protocol, we developed stereotypes as a new specification primitive. Our goal was to develop a primitive, which on the one hand, provides developers with a natural and lightweight way for heap structure description, and, on the other hand, can be efficiently proven by an automatic theorem prover. To validate our approach we applied it for the specification and verification of the priority

inheritance protocol. Our specification of the priority inheritance protocol was automatically verified in Boogie2.

- We presented a new technique to check the well-formedness of specifications. We showed how to incrementally construct a model for the specification, which guarantees that the partiality constraints of operations are respected and that the axiomatisation of pure methods is consistent.The results have been presented at the International Symposium on Formal Methods (FM 2008) [18].
- We proposed a new procedure Y to generate well-definedness conditions. Y is complete and yields formulas that grow linearly with respect to the size of the input formula. Our procedure has been used to enforce well-formedness of invariants and method specifications. The results have been presented at the International Joint Conference on Automatic Reasoning (IJCAR 2008) [5].
- We proposed an approach to show that mappings of model classes to structures provided by theorem provers are faithful. The proposed approach improves on previous work in three ways. First, previous work that proposed the mapping of model class did not ensure any actual semantic relationship. This can easily lead to semantic mismatch between what was intended to be specified and what was actually verified. Second, our approach leads to better specifications for model classes by ensuring their (relative) consistency and completeness. Third, previous work for ensuring the consistency of recursive pure-method specifications either does not provide a satisfying solution [6], or proposes to explicitly prove well-foundedness [18]. The proposed solution solves this problem by proving that a certain mathematical structure is a model for the specifications of a model class, we get the guarantee that the specifications are consistent. This result is independent of the presence of recursion and requires no explicit proof for well-foundedness. The results have been published in IET Software Journal [7].

Impact The results of the task have been published in [22, 7, 20, 15, 23, 9, 21, 5, 18].

Bibliography

[1] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In J. Vitek, editor, *European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer-Verlag, 2008.

[2] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer-Verlag, 2004.

[3] J. Charles. Adding native specifications to JML. In *Workshop on Formal Techniques for Java Programs*, 2006.

[4] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, 35(6):583–599, 2005.

[5] Á. Darvas, F. Mehta, and A. Rudich. Efficient well-definedness checking. In A. Armando, P. Baumgartner, and G. Dowek, editors, *International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2008.

[6] Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[7] Á. Darvas and P. Müller. Faithful mapping of model classes to mathematical structures. *IET Software*, 2(6):477–499, December 2008.

[8] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[9] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In J. Vitek, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[10] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):185–205, 2005.

[11] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007. Accepted for publication.

[12] K. R. Leino. *This is Boogie 2*, June 2008. Manuscript KRML 178.

[13] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004. Available from `www.sct.inf.ethz.ch/publications/index.html`.

[14] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[15] P. Müller and A. Rudich. Using stereotypes to verify complex heap structures in regional logic. Technical report, ETH Zurich, 2009. Avaliable from http://www.sct.ethz.ch/publications/index.html.

[16] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Principles of Programming Languages*, pages 268–280. ACM Press, January 2004.

[17] M. Parkinson and G. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages*, pages 247–258. ACM Press, 2005.

[18] A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In J. Cuellar and T. Maibaum, editors, *Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 68–83. Springer-Verlag, 2008.

[19] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for java-like programs based on dynamic frames. In *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2008.

[20] A. J. Summers and S. Drossopoulou. Considerate reasoning and the composite design pattern. In *VMCAI 2010*, 2010.

[21] A. J. Summers, S. Drossopoulou, and P. Müller. A universe-type-based verification technique for mutable static fields and methods. In *Formal Techniques for Java-like Programs*, 2008.

[22] A. J. Summers, S. Drossopoulou, and P. Müller. The need for flexible object invariants. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2009.

[23] A. J. Summers, S. Drossopoulou, and P. Müller. Universe-type-based verification techniques for mutable static fields and methods. *Journal of Object Technology (JOT)*, 8(4), June 2009.