Confinement and Representation Encapsulation in Object Teams

Bericht-Nr. 2004/06

Stephan Herrmann
stephan@cs.tu-berlin.de

ISSN 14369915

Confinement and Representation Encapsulation in Object Teams

Stephan Herrmann

Abstract

The language ObjectTeams/Java introduced the concept of Teams as a new module construct which was inspired by aspectual components and family polymorphism. Research in the field of alias control, confinement and ownership tends to look at modularity and encapsulation from a quite different perspective. In this paper we present an enhancement to the type system of ObjectTeams/Java which allows to combine the given benefits of flexible aspect composition with additional encapsulation constraints thus providing a basis for modular reasoning and guaranteed security constraints. We adopt concepts from confined types and the universes model. The enhancement of the type system uses only two new keywords for a variety of encapsulation schemes. We demonstrate why family polymorphism is very appropriate as a basis for the desired encapsulation constraints.

1 Protection Goals

Research in the field of alias control and related techniques [2] strives for more safety and security in object oriented programs. Obviously, the flat structure of objects on the run-time heap imposes problems to locality that would not exist, if strict hierarchical modules would be used. Still, the known advantages of the object oriented model prevail and solutions are sought as additions to standard OO programming languages.

The diversity of proposed techniques is striking and at a closer look, some of the differences are founded in very different motivations for restricting and analyzing programs. The work presented in this paper is influenced by three approaches. We refer to the techniques of family polymorphism [3], confined types [16] and universes [10]. In our programming language ObjectTeams/Java [5] we combine capabilities of all three techniques.

Before presenting any details of specific techniques, we point out the very different goals of the three approaches. Only after understanding the goals, we can think about a smooth integration that should serve all three goals.

1.1 Object families

Family polymorphism[3] is about grouping a set of classes into a larger class, such that member classes, and hence their instances, are uniquely owned by the enclosing instance. While the notion of families is motivated quite intuitively as a means for conceptual modeling, the focus of the techniques involved lies on type safety in the traditional understanding: ensuring that each attribute or method accessed on a target object is in fact present, and that static type checking can guarantee all runtime values to conform to their respective declarations.

Two questions are answered by family polymorphism: (1) how can a group of (mutually recursive) classes be refined in a way that internal consistency is given, and (2) how can the refinement be hidden from external clients, i.e., how can the whole group be handled polymorphically.

Question (1) is answered by virtual classes, which help to refine relationships between classes along with the classes themselves. Refinement of relationships implies covariant redefinitions of signatures, which—if admitted unrestrictedly—is very hard to type check statically (see Eiffel's concept of "system level validity" which requires a global data flow analysis for type checking[9]!). Question (2) is about locality. It calls for a special type discipline in order to protect clients against problems resulting from covariant redefinitions. Family polymorphism employs instance dependent types in order to statically check that member instances are not confused between families, which would otherwise result in the mentioned problems of covariance. More specifically, each family instance has its own set of member types, which is incompatible with all member types of any other family instance.

Family polymorphism raises the level of reuse and adaptation from classes to groups of classes.

1.2 Package level confinement

Confined types [16] provide alias control in order to confine certain objects to a well-defined scope of visibility. In this approach, Java packages are used as the scope from which objects should not escape. At the level of static scoping, Java's visibility modifiers seam sufficient. However, at the instance level widening of a confined object to a super-type with different visibility would compromise the protection.

In [16] the authors propose a set of rules for confined classes that prevent explicit widening to a non-confined type. Additionally, the concept of "anonymous methods" is introduced in order to make implicit widening of `this` safe, which occurs when invoking an inherited method.

Motivating examples for confined types deal with security issues, where certain information must be a secret of a well-defined portion of the program. This prevents incidental or malicious programming errors which could otherwise compromise the security of the program.

1.3 Representation encapsulation

The universes approach [10] relates to issues of modular reasoning. Again, groups of objects are considered, but here the central question concerns the invariants the can be defined for such a group. Once the invariant for an object depends on the state of other objects, it is desirable to hide those element objects. This calls for a kind of alias control where element objects are owned by exactly one object, which has the exclusive right to modify its elements.

The universes model uses two modifiers to annotate a program for analysis. A *representation type* (`rep T`) denotes objects that are owned by the current object. A *readonly reference* (of type `readonly T`) allows access to its target object only if the target object will not be changed by the operation.

Furthermore, the set of objects that can be mentioned in an invariant is restricted. As a result, for any given invariant, it suffices to analyze the methods of this class and its representation objects in order to prove that all method calls will preserve the invariant. This is in contrast to unconstrained models, where external clients may invoke methods of element objects (supposed representation objects), which by some side-effect might break the invariant of the compound object without giving the compound object a chance to even detect this situation.

1.4 General purpose language support

In order for the presented approaches to be applied in praxis, we see two possible roads: either (1) protection properties of a program are to be described outside the program using a formalism separated from the programming language or (2) a general purpose programming language should unify existing techniques in order to provide choices to the programmer what and how to protect.

The first option provides greatest flexibility. Tools for checking protection properties can be developed independently of compilers. Care must be taken, however, to minimize the additional burden of specifying encapsulation. If separate property files were used, double maintenance and the danger of inconsistency would be considerable obstacles to these techniques. Embedding specification into program comments by the use of special tags is a road taken by JML [7] and others. This avoids the need for maintaining separate files, while allowing to use a standard compiler or a specialized tool on the same sources.

However, we believe that a deeper embedding of specification into the program is desirable. More specifically, we see the opportunity for synergetic effects between different language mechanisms, some of which contribute to the operational semantics of a language while others only constrain the set of legal programs. In this vein we propose an integration of confined types [16] and ideas from universes [10] into the programming language ObjectTeams/Java [5, 12] which intrinsically applies the concepts of family polymorphism [3].

2 Object Teams

The Object Teams model, which has its roots in Aspectual Components [8], has first been presented in [5]. Its central concept is a new kind of module, a *Team*, which combines properties of classes with those of a package. Teams realize the concepts of family polymorphism [3]. A Team corresponds to a family, the contained objects are called *roles*. Role classes are virtual classes.

In the context of Team-inheritance, furtherbinding of role classes is realized by so-called *implicit inheritance*, where a role class extends a role class inherited from the super-Team if both classes have the same name. Note, that implicit inheritance does not establish a sub-type relationship, which would otherwise be unsafe due to covariance. Polymorphism only applies to the Team as a whole. Within a Team, regular inheritance between roles is still available. In case of competing implementations of the same method, the implicit super-class has priority over the explicit super-class. The new dimension of inheritance also comes with a new keyword for super calls: `tsuper`, which invokes a corresponding method version from the super-Team.

Each role type depends on the enclosing Team instance (instance dependent types). Within a Team `T` and its roles the containing Team instance is implicitly available although it can still be explicitly referred to as `this` or `T.this`. Thus the instance dependency is usually invisible within Teams and roles. By contrast, external clients need to specify role types as `teamInstance.RoleType`. In this case we speak of *externalized roles.* Two role types `t1.R` and `t2.R` are only compatible if the instances `t1` and `t2` are provably the same instance. By requiring the value `t1` in `t1.R` to be an immutable variable (`final`), a simple flow analysis is supported.[1]

2.1 Further concepts

The concepts presented above suffice for the discussion at hand. However, the intention behind Object Teams can only be understood, when also looking at an additional dimension of composition:

A role class may be bound to a class outside the Team (`class R playedBy B`). In that case the language ensures, that each role instance always has a valid link to a base instance of the declared type. Data-flows which cross the boundary of a Team automatically translate base objects to roles and vice versa. The translations are called `lifting` and `lowering`. Substitutability involving lifting or lowering is called *translation polymorphism* [4].

Based on the role-base-binding two kinds of method bindings can be defined. Method bindings are denoted in a declarative style and are part of a role class. *Callout* bindings, define automatic forwarding from a role to its base. *Callin* bindings insert triggers into base methods that call into the role. Callin bindings can be interpreted as aspect weaving. Both kinds in combination setup a selective form of instance-based inheritance: a role inherits methods from its base via callout and it may override base methods using callin.

Object Teams finally provide dynamic aspects. Any callin binding is only effective with respect to a Team instance that has been activated. Team activation and deactivation happens at runtime and can be controlled within the program by different means.

3 Protection Mechanisms in Object Teams

We will now step by step introduce the mechanisms of ObjectTeams/Java that contribute to encapsulation in its different flavors. In Sect. 4 we will show how the approaches presented in Sect. 1 can be mapped to our model in order to show soundness of our model in relation to those existing approaches.

[1]See [14] for a formal analysis of Java's *definite assignment* rule.

3.1 Encapsulation boundary

Alias control and confinement strive for improved modularity by introducing boundaries that help to enforce different levels of information hiding. While some approaches use existing constructs like Java packages and try to give more meaning to those constructs, other approaches tend to super-impose such boundaries as a secondary structure. For encapsulating run-time instances the existing module constructs are in fact quite dis-satisfactory.

Investigating special forms of encapsulation with respect to Object Teams was motivated by the observation, that the new module construct `team` provides a natural boundary between items to be protected and the outside.

A Team instance as a natural boundary facilitates some tasks which in other approaches require additional effort. In ObjectTeams/Java we need no modifiers like `rep` (for representation) or `confined` to mark the elements that should be protected. Rather, each role class that is not declared `public` is subject to some kind of instance protection.

There is also a specific reason for choosing universes[10] and confined types [16] as a model for alias control in Object Teams. Both papers contain hints at the suitability of family polymorphism for the issues at hand. In universes "types have to be interpreted relative to the current `this` object X_{this}" [10]. Such type interpretation is exactly, how family polymorphism is founded in the type system of ObjectTeams/Java, where each role type is always anchored to the enclosing Team instance. This facilitates static type checking as we will see below.

Confined types suffer from lack of support for refinement. In that approach the encapsulation boundary is given by packages. While this is promising from a pragmatic point of view, the authors of [16] need laborious indirections when refining a package `RSA` for a general key-pair-infrastructure to a package `Secure`. They observe: "this suggests that virtual types might be a better fit for confined types, as they allow subclassing of a whole family of classes in such a way that use relationships between classes in the original family become use relationships between classes in the derived family" [16].

The desired capability of sub-classing a whole family of classes is given in Object Teams using Team inheritance and implicit inheritance between roles. Team inheritance is mainly orthogonal to the issues at hand. A sub-Team defines a completely new scope. For this scope some encapsulation constraints may need to be analyzed anew, because the confinement status of a role in a super-Team is also an obligation for all its sub-Teams.

In the following we will see in depth how roles are encapsulated by their enclosing Team instance.

3.2 Strict role confinement

We will start with the strongest form of encapsulation: complete confinement, whereby no reference to a contained role may ever escape from the enclosing Team instance. This level is useful for implementing secrecy. It can be achieved by combining type visibility with restricted inheritance along with further concealment constraints.

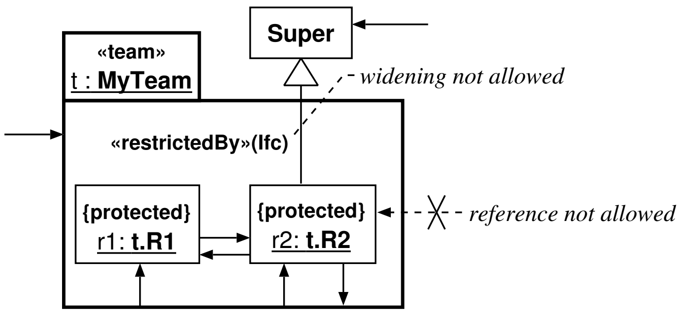

Figure 1: Roles as confined objects

Fig. 1 illustrates this situation. A Team instance `t` contains two role instances `r1` and `r2`, whose types depend on `t` (`t.R1` and `t.R2`). Confinement must achieve that no outside object ever references one of the roles `r1,r2`, whereas roles and Team may reference each other freely.

3.2.1 Type visibility.

In ObjectTeams/Java roles have only two levels of visibility: public and protected. Due to implicit inheritance private visibility would lead to inconsistencies. Since a Team class also has the function of a package for its contained roles, there is no point in distinguishing class and package level visibility. For roles the protected visibility is restricted to the enclosing Team instance and its roles.

A protected role type can not be used in declarations of the external interface of a Team, which includes all non-private Team features and all public features of public roles.

Given these rules a protected role would be completely invisible from the outside had it not been for polymorphism by which a role reference could still leak to the outside by widening to a non-restricted super-type (see class `Super` in Fig. 1). This super-type may be a class or an interface and may be a member of a Team (i.e., a role) or a regular type.

For the next paragraphs we will assume role classes that do not declare to implement any visible interfaces. It remains to be shown that the super-class of a role cannot be used for reference leakage. In Sect. 3.3 we will show, how interfaces can be used for a more specific control of accessibility.

3.2.2 Restricted inheritance.

Since every class in Java (except `Object`) has a super-class, widening to this super-class (commonly: widening to `Object`) would allow to view any object by a globally visible type. In order to restrict such widening we introduce restricted inheritance which uses the following syntax:

```
protected class Role
                extends Super restrictedBy SliceIfc
{
  /* class body */
}
```

The given declaration constructs a fresh super-class for `Role`, an imaginary class $\texttt{Super}_{SliceIfc}$, by extracting from `Super` only the slice of methods that are mentioned in `SliceIfc` and extending this set to its transitive closure with respect to self-calls (using `this` or `super`). Concerning the extends relation between the classes `Role` and `Super`, `SliceIfc` is called the *restricting interface*. This is the effect of restricted inheritance:

1. Internally, `Role` extends $\texttt{Super}_{SliceIfc}$.
2. `Role` is *not* a sub-type of `Super`, but only of `SliceIfc` (the imaginary class $\texttt{Super}_{SliceIfc}$ cannot be used explicitly).
3. Within inherited methods, `this` is internally re-typed to $\texttt{Super}_{SliceIfc}$.

Attributes are inherited without restrictions, since an attribute cannot leak the `this` reference (we'll come back to this later). By item (2), we disallow explicit widening from `Role` to `Super`. Accordingly, `(r instanceof Super)` yields `false` for any instance of `Role`.[2]

As a special case we support an interface `org.objectteams.None`, which is (contrary to Java's rules for interfaces) not a sub-type of `Object`. An interface designed as restricting interface will usually extend `None`. Using `None` directly as restricting interface we can construct role types with no usable super-type. In that case, the strictest form of encapsulation is given without further efforts.

For all cases where `SliceIfc` is non-empty — for the effect that certain methods of `Super` are available on `Role` instances — care must be taken that these methods don't compromise the desired encapsulation properties.

3.2.3 Concealment constraints.

The problem with inherited methods lies in the fact that they introduce implicit widening of the `this` reference to the declaring type of the method. While the intention of restricted inheritance is to disallow such widening the exception can be made safe by requiring the slice of inherited methods to conceal the callee's identity. Any implementation of a method in the restricting interface (either inherited or overridden in a role class) must be an anonymous method according to [16].[3] This means it may use the `this` and `super` references only for

[2]Only while executing a method from $\texttt{Super}_{SliceIfc}$, one might expect the type test to yield `true`, but note, that this affects only the reference `this`. We don't see reasonable use for the idiom `(this instanceof T)`. In order to avoid such confusion, the compiler should reject this pathological situation and consistently treat `Role` instances as *not* of type `Super`.

[3]In the sequel we use the notion of anonymous methods, although we consider the word a little misleading: a method may preserve the anonymity *of the callee instance*, i.e., anonymity is a property of the object, whereas the method should *preserve* this property.

1. accessing fields,
2. calling other anonymous methods,
3. reference comparison.

Anonymity of called methods will be analyzed transitively, i.e., the transitive closure $\texttt{Super}_{SliceIfc}$ mentioned above must contain anonymous methods only. However, only those methods explicitly mentioned in `SliceIfc` will be callable on the sub-class.

In order to include constructors in these constraints, restricting interfaces are allowed to mention constructors, too. Super-calls within methods and constructors of $\texttt{Super}_{SliceIfc}$ are also taken into account for the transitive closure. Within methods of `Role`, `super` may only be used to call methods that are mentioned in `SliceIfc`. [4]

Obviously, native methods are not allowed to be acquired using restricted inheritance.

These rules ensure that methods inherited via restricted inheritance will not reveal the object reference while viewing the object by the type of its super class.

3.2.4 Confinement example.

By means of figures 2 and 3 we briefly demonstrate the Object Teams syntax for confinement using an example from [16].

When evaluating line 7 in Fig. 2, the compiler checks the implementation of class `Random` with respect to the given constructor and method `nextDouble`. Both are required to be anonymous methods. By successful analysis, the instance `rnd` (line 26) is guaranteed not to escape from the context of its owning `RSA` instance.

Classes `Key` and `PrivateKey` have the same functionality but different visibility/protection.

Class `Secure.PrivateKey` in Fig. 3, line 6, refines `RSA.PrivateKey` using implicit inheritance. `RSA.PrivateKey` can still be widened to `Object`. The subtype relation is cut by the restricting interface `IKey`. Note, that in this case an *implicit* inheritance is restricted, which is the only case of using `restrictedBy` without `extends`. The same technique is used for class `KeyFactory`.

Recall that attributes are never restricted by a restricting interface, since attribute access cannot reveal the `this` reference: The concealment rules prevent storing `this` in an attribute in the first place.

Finally, a key-pair is created and initialized using the key factory. The public key can be passed to other modules of the system, while the private key, the key factory and its random generator are confined to their enclosing instance of type `Secure`.

[4]We will discuss modular type checking in Sect. 3.6.

```
public team class RSA {
   protected interface IRandom extends None {
      IRandom(long seed); // a constructor in an interface
      double nextDouble();
   }
   protected class ProtectedRandom
                 extends Random restrictedBy IRandom
   {
      protected ProtectedRandom (long seed) {
         super(seed);   // super constructor is imported via IRandom
      }
   }

   public class Key {
      public BigDecimal mod;
      public BigDecimal exp;
      public String crypt (String msg) {
         // encrypt msg using mod and exp.
      }
  }

   // Pre-planning: some keys will be confined.
   protected class PrivateKey extends Key {}

   public class KeyFactory {
      private ProtectedRandom rnd =
        new ProtectedRandom(System.currentTimeMillis());

      protected void genKeyPair(Key pub, PrivateKey priv)
      {
         // method nextDouble() is imported via IRandom:
         double d = rnd.nextDouble();
         // use the random value to compute and set the key components.
      }
   }
}
```

Figure 2: A general key-pair infrastructure (RSA) in ObjectTeams/Java

3.3 Representation encapsulation

The previous section has demonstrated an extreme form of encapsulating role objects by making those roles completely inaccessible from the outside. While this is desirable for some security issues, other applications require a more pragmatic approach. Work on modular reasoning suggests to define unique ownership for representation objects while allowing other external objects to access the representation as long as they are not allowed to modify representation objects. A typical example is a class `LinkedList` which owns the nodes that represent the list. Still an iterator should be given access to nodes in order to ef-

```
public team class Secure extends RSA {
   protected interface IKey extends None {
      protected String crypt (String msg);
   }
   // IKey restricts the implicit inheritance from RSA.PrivateKey.
   protected class PrivateKey restrictedBy IKey {}

   protected interface IFactory extends None {
      protected void genKeyPair(Key pub, PrivateKey priv)
                     ;
   }
   protected class KeyFactory restrictedBy IFactory {}

   private PrivateKey privateKey;
   public  Key        publicKey;

   public void test() {
      privateKey = new PrivateKey();
      publicKey = new Key();
      KeyFactory keyFactory = new KeyFactory();
      // method genKeyPair(..) is imported via IFactory:
      keyFactory.genKeyPair(publicKey, privateKey);
      // use keys for encryption and decryption...
   }
}
```

Figure 3: A refinement of the RSA Team

ficiently store its current position. In order to localize responsibility for the list's invariant in the `LinkedList` class, the iterator should not be allowed to modify the state of any node objects. This level of protection is called representation encapsulation.

Object Teams allow to define representation encapsulation by the techniques of externalized roles and readonly interfaces.

3.3.1 Externalized roles.

If external objects shall have access to role instances, the type rules of family polymorphism[3] require special considerations. In Object Teams the technique of *externalized roles* ensures type safety of mutually recursive role types in conjunction with Team inheritance. Externalized roles have been introduced in [5, 6]. Outside the Team the type of an externalized role must be given as `teamInstance.RoleType`, i.e., the role type is relative to the enclosing Team instance (instance dependent types).

The syntax `teamInstance.RoleType` requires the role type to be declared public. Note, that this also prohibits the case where a Team instance refers to protected roles of another Team even of the same type. Instance dependent types ensure instance based protection, which should be contrasted to Java's `private` modifier which defines a feature as local to its defining *class* not its

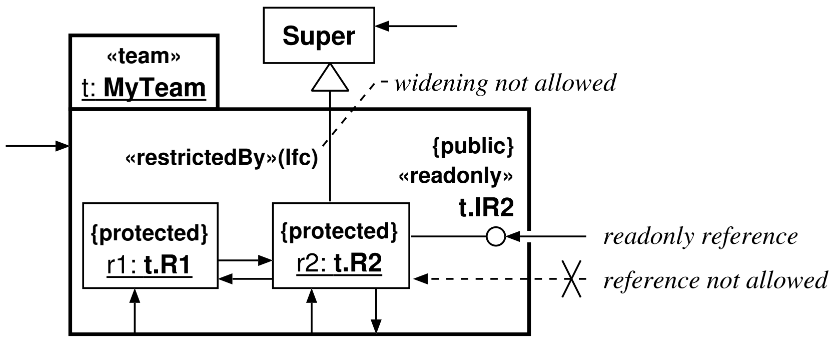

Figure 4: Owned roles accessed via a readonly interface

containing *object*.

In order to combine alias control of protected roles (using restricted inheritance) with externalized roles, a role class declares to implement a public interface. By this public interface the role is accessible from the outside. All that is needed in order to ensure representation encapsulation is a guarantee that no client may modify the role object via its public interface.

3.3.2 Readonly interfaces.

In ObjectTeams/Java an interface can be declared to be `readonly`. Such declaration is an obligation for each implementing class, to ensure that all methods of this interface are implemented as pure functional methods, i.e., methods without side effect.

This obligation results in the following checks for each method under consideration:

1. The method may not modify any fields of the current object.
2. Method calls to targets within the Team context may only use methods that are subject to the same readonly restriction. The Team context is defined as the enclosing Team instance and all its contained role instances (of course including the current object).

The default super-type for readonly interfaces is `None`, the type without properties. If a readonly interface extends another interface, the methods from that interface are included in the checks for readonly-ness. [5]

Representation encapsulation is realized by two views on roles: (1) The containing Team and its roles access the role by its exact type. This type allows full access. Role protection and restricted inheritance make sure, that no references by the exact role type or any un-protected super-type escape from the Team context. (2) External clients view the role only by its public interface which is declared as a readonly interface. The implementation of a role is checked for compliance to the readonly declaration.

Fig. 4 extends Fig. 1 by a new interface `t.IR2`, which is annotated as `readonly`. Using this interface external clients my refer to `r2` but this reference can only be used to call pure functional methods.

[5]We will discuss modular type checking in Sect. 3.6.

Note, that it is not a problem if the control flow of a readonly method leaves and re-enters the Team context, since outside objects can only refer to roles via their readonly interfaces. In other words, a readonly method does not prohibit state changes of outside objects, but only of protected roles.

In general, Team classes are subject to the normal Java mechanism for visibility control. Since a readonly role method may call methods of its enclosing Team instance, these Team-level methods need to be checked for readonly-ness, too, i.e., the Team must also declare to implement some readonly interface.

If by means of Team-nesting a Team class `Outer.Mid` is also a role in an even larger Team `Outer`, the readonly check for `Outer.Mid` concerns calls to the following instances:

- contained roles `Outer.Mid.Inner`
- `this` (of type `Outer.Mid`)
- sibling roles `Outer.Other`
- the enclosing Team `Outer`

3.4 Example for representation encapsulation

Figures 5 and 6 show how the linked list example from [10] could be implemented in ObjectTeams/Java.

In Fig. 5 three levels of encapsulation can be seen: lines 21–30 define the public interface of a linked list. Lines 7–18 define a protected role class and two variables of this type which cannot be accessed from the outside. Lines 2–5 define a public type for readonly access to the list's nodes. The return statement in line 21 uses widening from `Node` to `INode` in order to pass a protected role to the outside.

The iterator class is shown in Fig. 6. Each iterator instance is immutably tied to an instance of `LinkedList` (line 33). The variable `theList` is used as an anchor for all occurrences of type `INode` (lines 34, 48), ensuring that only nodes of this list will be handled by this iterator. Note, that this invariant is checked by the compiler based on family polymorphism.

On any instance of `INode` the iterator may only access methods `elem()` and `next()`. Removing the current node is supported by the Team-level method `void remove(INode n)` (lines 30, 49). Inside this method, casting `n` from `INode` to `Node` does not require runtime checks regarding the owner object, since this is already covered by the static rules for externalized roles.

For the given implementation, static analysis proves that outside the Team `LinkedList` any `Node` instance can only be accessed by the type `INode`, which does not allow any modification of the call target. An iterator may share the internal state with its associated list, but for any side effect on the list, it *must* use the public Team interface. Analysis of any invariant of `LinkedList` can thus be performed by only looking at this class and its contained elements.

3.5 Integrating existing classes

If we want to create a trivial variant of a non-role class such that its instances can be protected by the Team, this can be done by defining a new role class,

```
public team class LinkedList {
    public readonly interface INode {
        INode  next();
        Object elem();
    }

    protected class Node implements INode {
        protected Node   _next;
        protected Object _elem;

        protected Node (Object e)       {_elem = e;}
        protected void setNext (Node n) {_next = n;}

        public  INode  next () { return _next; }
        public  Object elem () { return _elem; }
    }

    protected Node _first, _last;


    public INode first  () { return _first;         }
    public Iter  getIter() { return new Iter(this); }


    public void add(Object o) {
        Node n = new Node(o);
        _last.setNext(n);
        _last = n;
    }
    public void remove (INode n) { /* body omitted */ }
}
```

Figure 5: Class `LinkedList` in ObjectTeams/Java

which inherits from the given class (subject to a restricting interface). Also a public readonly interface has to be defined which is implemented by the role class. As an example consider the frequently quoted class `Vector`. This is how a vector could be used as an encapsulated representation:

```
public class Iter {
    private final LinkedList    theList;
    private        theList.INode position;

    public Iter (LinkedList l) {
        theList = l;
        position = theList.first();
    }

    public Object next () {
        Object result = position.elem();
        position = position.next();
        return result;
    }

    public void remove () {
        theList.INode pos = position.next();
        theList.remove(position);
        position = pos;
    }

}
```

Figure 6: An iterator of linked lists

```
public readonly interface IVector {
   public Object elementAt(int i);
   public int size();
   public Iterator iterator();
}
team class T {
   protected class PVector
                    extends Vector restrictedBy ROVector
                    implements IVector
   { /* empty class body */ }
}
```

Interfaces like `IVector` and `ROVector` (not shown here) could be automatically derived from the library implementation and shipped as library enhancements. We are currently discussing to include a special syntax `restrictedBy *` in order to declare that all methods from a super-class are inherited and that all these methods must obey the concealment constraints of anonymous methods.

3.6 Modular type checking

All checks regarding anonymous and readonly methods must be repeated for each sub-class along both dimensions of inheritance (implicit and explicit). Compiler messages must provide help for understanding the connection between an obligation due to `restrictedBy` or `readonly` declarations and the implementation that has to comply to the obligation. We consider this tight coupling along inheritance quite normal.

It is an advantage of the presented model that use relationships always depend on declared interfaces only. As a result, the implementation of a Team and its roles can be checked in a modular way, providing protection guarantees with respect to all potential clients. Changing the implementation of a Team without changing its interface will never affect the validity of any client.

3.7 Summary so far

Protected role types are not visible outside a Team instance. By restricting inheritance, references to protected roles can be protected against escaping from the enclosing Team context. The concealing property of inherited or overridden methods is checked including transitive method calls.

The other extreme, a public role class, is only subject to the type rules of family polymorphism, which prohibit to confuse role instances of different Teams.

At a medium level of protection, a role class may, in addition to strict alias control, provide a readonly view using a corresponding readonly interface. The readonly property is checked for any control flow within the Team context.

All features can be freely combined, but only certain combinations succeed in providing well-defined protection. The following will result in compile-time warnings in order to alert the programmer of incomplete protection:

1. If a role class implements a readonly interface but extends a non-protected class using unrestricted inheritance.

2. If a role with restricted inheritance implements a public interface that is not `readonly`.

3. If a public role contains calls to update methods on protected roles.

In case (1), encapsulation is incomplete since a reference to the role may still leak by widening, which could result in updates outside the Team-scope. Case (2) would break ownership since an otherwise protected role could be accessed globally by a read-write interface. Case (3) would allow a client to use an externalized role in order to modify a protected role. This way, control flows that do not originate in the Team could break a Team invariant which depends on the state of a protected role. Still such control flow has to use a public interface (in this case of the public role), which simply means, that a Team shares responsibility for protected roles with all public roles it defines.

Without compile-time warnings these guarantees hold:

- A protected role class with restricted inheritance is *owned* by the enclosing Team instance, i.e., only the Team and its roles are allowed to modify the protected role. Any control flow that modifies an owned role must go through a Team-level method.

- A protected role with restricted inheritance that does not implement any public interfaces is *confined*, meaning that no reference to the role can escape the enclosing Team instance.

4 Comparison

In this section we relate encapsulation in Object Teams to existing approaches in order to demonstrate how we combine desirable properties without simply adding up all the language constructs which they introduce. Also, by mapping our concepts to those of existing approaches we reuse the work that has been done regarding the soundness of the proposed model.

Please note, that we do not intend to introduce any new mechanism. Only our `restrictedBy` keyword seems to be a new construct. It even reminds of the separation between code reuse and sub-typing as it is realized in Sather [15]. At a closer look, restricted inheritance is little more than a different style of denoting the anonymity constraints known from confined types. We see our contribution in the integration of existing techniques providing abstractions to the programmer that follow a consistent style of annotating classes and interfaces for controlling encapsulation.

4.1 Protection by Family Polymorphism

First of all, family polymorphism provides type safety in situations of covariant refinement of a set of mutually-recursive classes. However, the consistency invariant of family polymorphism is stronger than simply avoiding "method not understood" problems. By using outer *instances* instead of some exact–type mechanism, family polymorphism defines groups of objects that may interact with each other, whereas any member of a different group is considered incompatible.

This invariant may already be used to implement certain security constraints. Consider an outer class `User` containing a class `Key`. Even if `Key` is public and provides a public method `void copyValues(Key other)`, this method cannot be used to initialize the key of one user with the values of another user's key, since the parameter `other` must originate from the same outer instance as the call target.

While this protection is still incomplete it provides a very useful basis for implementing different styles of protection. The value of family polymorphism for the issues at hand lies in the connection between instance based scoping and static type analysis.

4.2 Strong Alias Control

Our restricting interfaces impose the same constraints on method implementations as the keyword `anon` in confined types. By separating declaration and implementation, existing classes can easily be fitted into our model. The super classes for protected roles need not contain the `anon` declarations but these are given at the inheritance relationship.

Confined types need additional rules to prohibit widening from a confined type to a non-confined type. In our model this follows naturally from restricted inheritance which does not define a sub-type relation.

From confined types we adopt another rule: protected roles may not be subtypes of `Throwable` and `Thread`.

The appendix lists the rules for confined types and how they are mapped in our model.

Comparing the implementation of the `RSA` and `Secure` packages in [16] with our version, we find indeed that by the use of virtual classes (here: implicit inheritance) the synthetic interface `KeyWriter` is not needed any more. This observation is in line with their assumption, that virtual classes might help to achieve better designs with confined types.

4.3 Representation Encapsulation

Readonly *references* and readonly *types* follow the same goal. The most obvious difference regards the propagation of the `readonly` property. In universes, reading fields or invoking functional methods via a readonly reference yields again a readonly reference. In Object Teams this needs no special mechanism, since a public readonly interface can only mention public role types, which in the case of protected roles will be their readonly interfaces.

Universes apply down-casts in order to convert a readonly reference back to the unconstrained reference. The run-time checks for these down-casts involve a comparison of owner objects, which fails if the cast happens in the wrong instance scope. Also Object Teams require down-casts, but the ownership check is performed statically using the instance dependent types of externalized roles. As a result, a down-cast from a public role interface to its protected implementation will never fail at run-time due to a wrong owner.

Universes include explicit dependency control by also restricting outgoing references. We only consider protection of roles in a Team. Thus, we allow parts of the program to be unrestricted with respect to ownership, which seems desirable for the purpose of incrementally introducing our mechanisms into traditional development.

Universes also have a potential problem with software evolution: if the implementation of a method being used via a readonly reference is ever changed to include a side effect, the new program is broken. In order to detect this situation the whole program needs to be analyzed, since all potential clients need to be found and all method invocations analyzed with respect to side-effects of the implementation. The ObjectTeams/Java compiler will easily detect this situation, since the new implementation fails the explicit readonly obligation of its readonly interface. This checking can be performed locally on the modified Team only.

4.4 Classification

A classification for encapsulation techniques has been proposed in [11]. By analyzing access graphs, they partition a system into the domains boundary $\mathcal{B}$, inside $\mathcal{I}$, external references $\mathcal{R}$ and outside $\mathcal{O}$. The partitioning function is called the *encapsulation function*. Different encapsulation functions define different encapsulation schemes enforcing different encapsulation constraints.

The encapsulation function for ObjectTeams/Java depends on which constructs are allowed in which combination. Only external references are not dealt with explicitly in our model. In addition to the comparison given above, the classification can for example be used, to show, that a Team containing no public roles ensures *external uniqueness*[1] regarding its roles: the Team is the only boundary object referring to a role.

5 Summary

We have presented, how different styles of alias control and encapsulation can be integrated into a modern programming language that already features family polymorphism. By the addition of only two new keywords — `restrictedBy` and `readonly` — ObjectTeams/Java supports static analysis which can be used to ensure different levels of protection. The integrated language involves three concepts for encapsulation: instance dependent types, restricted inheritance which separates method acquisition from sub-typing, and readonly interfaces. The annotations used for specifying encapsulation are explicit at the level of types (interfaces), rather than annotating specific references. This way, different possibilities of access are directly visible when looking at a role class. This is in contrast with approaches were any use-context of a given type may request different access modes. If an existing class shall be used in a protected manner, this can still be achieved by a sub-class which only alters the accessibility of the class.

5.1 Future work

We will need to further investigate the relationship between encapsulation and other concepts of Object Teams. In [6] we have given a first analysis of encapsulation in the presence of the role-base duality.

Based on the role-base relation, Object Teams support aspect oriented programming using `callin` method bindings. As a first approximation `callin`-bound methods need to be considered part of the public interface of the enclosing Team. This is not surprising since the purpose of a Team with callin bindings is to be invoked via these bindings at certain points of execution inside the base application. Interestingly, callin bindings define control flows entering a Team via a role *without* granting external access to the role. Leaking role references out off a Team can be prevented by the *lowering* mechanism, whereby any role instance leaving the scope of its Team is automatically translated to the corresponding base instance [4].

However, these are only preliminary considerations, which need to be refined in the future. We will hopefully show, how separating a conceptual entity into a base instance and a set of roles, may in fact help to protect parts of an object while sharing the basic concept.

The language ObjectTeams/Java is currently being evaluated with respect to real world applicability within a project funded by the German research ministry (BMBF). The case studies developed within that project will hopefully give further insight also regarding the encapsulation properties described in this paper.

Acknowledgements

The idea of combining Universes with Object Teams goes back to an inspiring discussion with Arnd Poetzsch-Heffter. Special thanks go to Erik Ernst and Rebekka Oeters, whose comments on early versions of this paper were of great help regarding both the concepts and their presentation.

References

[1] D. Clarke and T. Wrigstrad. External uniqueness is unique enough. In *Proc. ECOOP 2003*, number 2743 in LNCS, pages 176–200. Springer Verlag, 2003.

[2] D. Clarke (editor). Proc. International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO) at ECOOP 2003. Technical Report UU-CS-2003-030, Utrecht University, 2003.

[3] Erik Ernst. Family polymorphism. In *Proc. of ECOOP'01*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.

[4] S. Herrmann, C. Hundt, K. Mehner. Translation polymorphism in Object Teams. Technical Report 2004/05, Fak.IV, Technical University Berlin, 2004.

[5] Stephan Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In M. Aksit, M. Mezini, and R. Unland, editors, *Proc. Net Object Days 2002*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2002.

[6] Stephan Herrmann. Sustainable architectures by combining flexibility and strictness in Object Teams. *IEE Software*, to appear (Draft version available at: `http://objectteams.org/publications/IEE-Software.ps`).

[7] G. T. Leavens, A. L. Baker, and C. Ruby. *Behavioral Specifications for Businesses and Systems*, chapter JML: A Notation for Detailed Design,, pages 175–188. Kluwer Academic Publishers, 1999.

[8] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. In *Technical Report*, Northeastern University, April 1999.

[9] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall International, New York, 1992.

[10] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[11] J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *Proc. IWACO '03* [2].

[12] Object Teams home page. http://www.ObjectTeams.org.

[13] Rebekka Oeters. Formalisierung und Analyse eines Typsystems für ObjectTeams/Java mit dem Beweiswerkzeug Isabelle (in german). Master's thesis, Technical University Berlin, 2003.

[14] Norbert Schirmer. Java definite assignment in Isabelle/HOL. In *Proc. of ECOOP Workshop on Formal Techniques for Java-like Programs*, number 408 in Technical Report. ETH Zürich, 2003.

[15] C. Szyperski, S. Omohundro, and S. Murer. Engineering a programming language — the type and class system of Sather. In *Proc. International Conference on Programming Languages and System Architectures*, volume 782 of *LNCS*. Springer-Verlag, March 1994.

[16] J. Vitek and B. Bokowski. Confined types in Java. *Software– Practice and Experience*, 31(6):507–532, 2001.

A Rules for Confined Types

This is how the rules for confined types[16] map to the type system of Object-Teams/Java:

	Confined Types	**Object Teams**
$\mathcal{C}1$	A confined class or interface must not be declared public or protected, and must not belong to the unnamed global package.	A protected role is only visible within its enclosing Team instance.
$\mathcal{C}2$	Subtypes of a confined type must be confined and belong to the same package as their confined super-type.	Explicit subtypes of protected roles must be protected and belong to the same Team as their protected super-type. Protection must also be preserved along implicit inheritance.
$\mathcal{C}3$	Widening of references from a confined type to an unconfined type is forbidden in …	With restricted inheritance a protected role is not a sub-type of the class it extends.
$\mathcal{C}4$	Methods invoked on a confined object must either be defined in a confined class or be anonymous methods.	Methods inherited with restriction from a non-protected class to a protected role must be anonymous methods.
$\mathcal{C}5$	The same for constructors	
$\mathcal{C}6$	Subtypes of `java.lang.Throwable` and `java.lang.Thread` may not be confined/protected roles.	
$\mathcal{C}7$	The declared type of public and protected fields may not be confined.	The client interface of a Team may not use a protected role type.
$\mathcal{C}8$	The return type of public and protected methods may not be confined.	ditto