Universe Type System — Quick-Reference

Werner Dietl, Peter Müller, and Daniel Schregenberger

July 18, 2008

Contents

Abstract

The Universe type system is a lightweight ownership type system that hierarchically structures the object store. It enforces the owner-as-modifier discipline that allows the modular verification of object-oriented programs. The Universe type system is part of the Java Modeling Language (JML) and supported by several tools.

This document serves as a quick reference for the Universe type system, providing an overview and summarizing the type rules and the usage of the tools.

1 Introduction

The Universe type system (UTS) [3, 1] is designed to control aliasing and dependencies. It is integrated into the Java Modeling Language (JML) [4] and the Common JML2 tools [1]. The UTS enables the modular verification of object-oriented programs [6, 5, 7].

The Universe type system is a simple ownership type system. Every object is assigned at most one other object as the owner object. A context is the set of objects that share the same owner. Objects with no owner are said to be in the root context. The UTS enforces a hierarchical partitioning of the object store into contexts.

Normal read-write references are allowed only between objects in the same context or from the owner to objects in its context. Any other reference is read-only. Read-only references, as opposed to normal references in Java, do not allow any modification of the referenced object — neither by updating fields directly nor by calling non-pure methods on that object. Furthermore read-only references are transitive to prevent anyone from getting a read-write reference through a read-only reference. The idea behind this is to enforce representation encapsulation and avoid leaking, without completely disallowing aliasing.

This document is structured as follows. Section 2 gives an overview of the Universe type system and Section 3 summarizes the type rules. In Section 4 we discuss the tool support for the UTS. Section 5 gives an example and finally Section 6 gives references.

2 Universe Type System Overview

2.1 Annotations

2.1.1 Type Annotations — Ownership Modifiers

Alias control applies only to reference types. Primitive types like `int` or `boolean` are handled the same way as they are in standard Java and do not take an ownership modifier.

The classification of references is done by using an extended type system. There are three modifiers for types:

- `peer` denotes a reference to an object inside the same context. This is the default, unless otherwise noted below.

[1]Available from `http://www.jmlspecs.org/`

- `rep` denotes a reference from an object into the context it owns.
- `readonly` denotes a reference that is read-only and might point to objects in any context. The modifier `any` can be used as semantically equivalent alternative.

These keywords can be used in front of a standard Java type, separated by a whitespace. We call this prefix the ownership modifier(s).

For reference types and one-dimensional arrays of a primitive type, only a single ownership modifier is allowed. For arrays of reference types and multi-dimensional arrays of primitive types two modifiers are allowed, the first one for the array object itself and the second one for the references stored in the array. Ownership modifiers can equally be used for the type arguments of generic classes.

According to their (first) ownership modifier, we call reference types peer, rep, and read-only types, respectively.

The keywords can be used directly in Java code, but then the code can only be compiled with the JML compiler with enabled Universe parsing. Alternatively, they can be placed within JML specification comments, to allow standard Java tools to process the source. Within JML specifications the keywords `\peer`, `\rep`, and `\readonly` can also be used as alternatives, that are ignored by non Universe-aware JML tools.

Here some examples:

```
peer T a;      // as Java keyword, needs parse option
/*@ peer @*/ T b; // as JML specification, needs parse option
/*@ \peer @*/ T c; // as JML specification, always parses

// an array that is owned by this,
// the array elements are instances of class T in arbitrary contexts.
rep readonly T[] d;

// a List instance that is owned by this,
// the elements managed by the list are T instances also owned by this.
rep List<rep T> e;
```

In the examples in this document we will use the Java keyword syntax for brevity. In general, the JML comment syntax is recommended for backwards compatibility.

2.1.2 Method Annotations — Purity

Methods can be marked as pure, if they are side-effect free, that is, do not modify existing objects. The only methods that can be called on read-only references are pure methods.

```
public pure peer R m(readonly S s, readonly T t) { ... }
```

In this example method `m` is pure. The return value is of type `peer R`. Both parameters have the `readonly` modifier.

Purity can also be specified in a JML comment as `/*@ pure @*/`.

See [4], Section 7.1.1.3 [Pure Methods and Constructors], for more about pure methods.

2.2 Viewpoint Adaptation

The ownership modifiers are always relative to the current object. If the current object changes, viewpoint adaptation has to be applied to types. For example, to determine the owner of an object referenced by `x.f` — and, thus, the type of the field access `x.f` — one has to consider the ownership modifiers of both `x` and `f`:

1. If the types of both `x` and `f` are peer types, then we know (a) that the object referenced by `x` has the same owner as `this`, and (b) that the object referenced by `x.f` has the same owner as `x` and, thus, the same owner as `this`. Consequently, the type of `x.f` also has the modifier peer.

2. If the type of `f` is a rep type, then the type of `this.f` has the modifier rep, because the object referenced by `this.f` is owned by `this`.

3. If the type of `x` is a rep type and the type of `f` is a peer type, then the type of `x.f` has the modifier rep, because (a) the object referenced by `x` is owned by `this`, and (b) the object referenced by `x.f` has the same owner as `x`, that is, `this`.

4. In all other cases, we cannot determine statically that the object referenced by `x.f` has the same owner as `this` or is owned by `this`. Therefore, in these cases the type of `x.f` has the modifier read-only.

The same rules are used for method calls and can be expressed as a type combinator that takes two ownership modifiers and returns the resulting ownership modifier. This type combinator is used to determine the type of field accesses and method call parameters and results. It is defined by the following table (first argument: left-most cell of the rows, second argument: top-most cell of the columns)

*****	**peer**	**rep**	**readonly**
peer	`peer`	`readonly`	`readonly`
rep	`rep`	`readonly`	`readonly`
readonly	`readonly`	`readonly`	`readonly`

The `this` reference is always a peer reference. If the first parameter is a `this` reference, then the type combinator is not applied. Therefore `peer * rep` yields a rep reference if the first parameter is the `this` reference, allowing the owner object full access to its representation.

2.3 Subtyping

The subtype relation on Universe types follows the subtype relation in Java: Two peer, rep, or read-only types are subtypes if the corresponding classes or interfaces are subtypes in Java. In addition, every peer and rep type is a subtype of the read-only type with the same class, interface, or array element type, because it is more specific in terms of the context information it conveys.

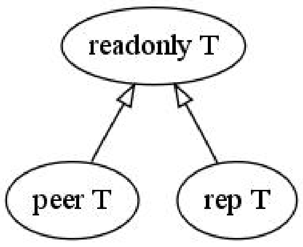

Figure 1: Type hierarchy.

Viewpoint adaptation and subtyping for generic types and arrays is discussed in the following subsections.

2.4 Generic Universe Types

For a detailed discussion of Generic Universe Types see [1].

- Ownership modifiers for type arguments are interpreted relative to the current object, not relative to the instance of the generic type.

  For example, consider the declaration `peer List<rep T> e`. Variable `e` will reference a `List` object that is in the current context and that manages `T` objects that are owned by `this`.

- Viewpoint adaptation of generic types is applied recursively on the type arguments. If a type argument is changed to readonly, also the enclosing arguments become readonly.

  For example, combining a `rep MyList` reference with a `peer List<peer Data>` reference results in a `rep List<rep Data>` reference.

- Upper bounds of type variables can have peer or read-only types; by default they are read-only.

- In general, subtyping for generic types follows the Java subtyping rules and is therefore invariant in the type arguments. However, we can relax this rule a bit and allow covariant changes in type arguments, if the corresponding main modifiers are read-only.

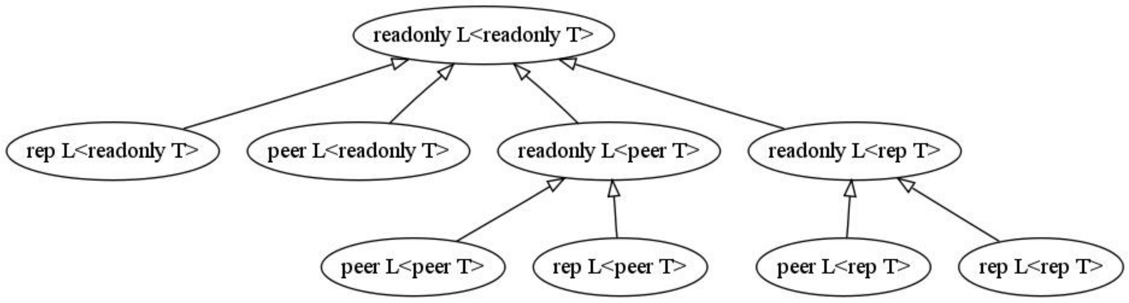

Figure 2: Type hierarchy for generic types.

2.5 Array Types

- Arrays of reference types need two ownership modifiers: one for the array object and one for the type of the references they store. All array objects of a multi-dimensional array of reference type are in the context specified by the first modifier.

- One-dimensional arrays of primitive types need only one modifier: the one for the array object. The elements stored in the array are primitive types and do not take ownership modifiers.

- Multi-dimensional arrays of primitive types need two modifiers: one for the array object and one for the one-dimensional array at the "lowest" level.

- We have covariant array subtyping: an array with peer or rep element type is subtype of a corresponding array with read-only element type. We need a runtime check for write access.

- Array types are interpreted like generic types, that is, the element type is also relative to the current object, not relative to the array object.

  For example, consider the declaration `peer rep T[] a`. This is interpreted like `peer List<rep T> l`. The only difference is the covariant subtyping of arrays.

- Note that the interpretation of array modifiers changed in a recent version. In the beginning, array accesses were basically interpreted like field accesses, that is, the type combinator was used to determine the type of the access expression. The ownership modifier of elements was restricted to be either peer or read-only — rep was not allowed — and was interpreted relative to the array object and not relative to the current object. This behavior was inconsistent with generic types and therefore unified. In the current version, ownership modifiers are always relative to the current object.

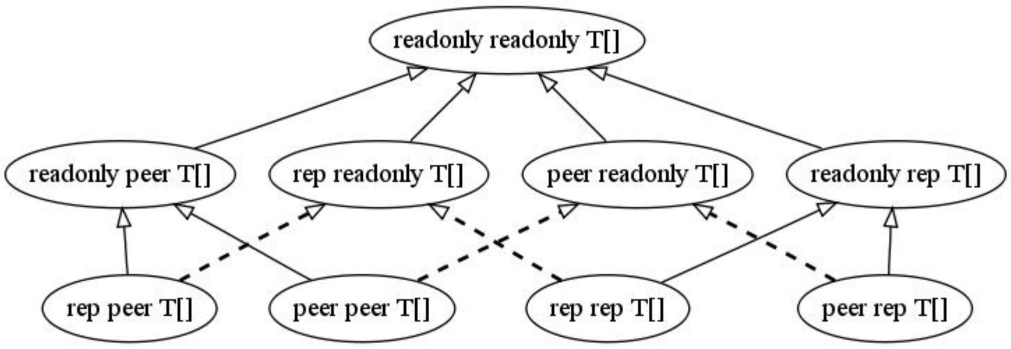

Figure 3: Type hierarchy for arrays. Dashed lines signify covariant relations that need runtime checks on writes.

2.6 Casts and Instanceof

A downcast can be used to change a read-only type into a read-write type. This is sometimes necessary if a read-write reference was passed out as read-only and is later returned back to the owner or a peer object.

As in standard Java, a downcast can fail, which causes a `ClassCastException` to be raised. To be sure that a cast will not fail at runtime, an `instanceof` can be used to check the type.

The two types of a cast or instanceof have to be comparable. This means the combination of peer and rep types is not allowed.

If the type in the cast or instanceof expression does not have an explicit ownership modifier, we take the ownership modifier of the static type of the expression being cast or tested. More explicitly, this means:

- For casts, we take the ownership modifier of the static type of the right hand side, i.e., the expression being cast, as the ownership modifier for the type of the cast.
- For instanceof, we take the ownership modifier of the static type of the left hand side, i.e., the object who's type is checked, as the ownership modifier for the type on the right hand side.

This eases the transition of existing Java code, by assuming that existing cast/instanceof operations do not want to test ownership information.

2.7 Dynamic checks

Dynamic checks are needed for the instanceof operator, downcasts of read-only reference types, and assignment to array components of reference types. The dynamic checks are implemented by inserting the checks into the bytecode generated by the JML compiler.

2.8 Defaults

Defaults are used to reduce the amount of ownership annotations that are needed.

- The ownership modifiers of local variable declarations are propagated from the initializer expression. If no initializer is present, the other defaults are applied.
- The ownership modifiers of field declarations are propagated from the initializer expression. If no initializer is present, the other defaults are applied. If a field type was already used to determine the ownership modifier of some other field, i.e. it was used in the initializer expression of some other field, then the type cannot be changed any more and the other defaults are applied.
- Reference types by default are peer types, with the following exceptions:

  - The ownership modifier of immutable types defaults to `readonly`. Currently, the set of immutable types only includes the Java wrapper types for primitive types (e.g. `java.lang.Integer` and `java.lang.Long`), `java.lang.String`, `java.lang.Class`, and `java.math.BigInteger`.
  - The default ownership modifier for a type in the `throws` clause of a method header, and in the declaration of a `catch` clause of a `try` statement is `readonly` [2]. Exception types default to `readonly`.
  - The default modifier for explicit formal parameters to a `pure` method (but not for the receiver, `this`) is `readonly`. Pure methods are required to have readonly parameters and explicitly using a different ownership modifier results in an error. (Note that this is not the case for pure constructors, however, which can have peer, rep, and readonly parameter types.)
  - In a cast expression of the form `(T) e` the ownership modifiers of `e` are propagated to the type of `T`.
  - In an instanceof expression of the form `e instanceof T` the ownership modifiers of `e` are propagated to the type of `T`.
  - Upper bounds of type variables default to `readonly`.
  - The same defaults are used for type arguments.

- Arrays of mutable types and multi-dimensional arrays of primitive type take two ownership modifiers. If only one ownership modifier is given, it is interpreted as element modifier and `peer` is used as array modifier.

  For example, `rep Object[]` is interpreted as `peer rep Object[]`.

- If an array of an immutable type takes only one ownership modifier, this modifier will be used as array modifier, not as element modifier, as the element modifier defaults to `readonly`.

  For example, `rep Integer[]` is interpreted as `rep readonly Integer[]`. A compiler warning will notify the user of this behavior.

3 Rules and Notes

This section is an informal collection of the rules of the Universe type system and notes about their implementation.

3.1 Instance Fields

- According to case 4 of the type combinator above, if an instance field with a rep type is read on a receiver different from `this`, the field access expression has a read-only type. To prevent even leaking of read-only references to representation objects, an appropriate access modifier must be used and leaking through getter methods must be prevented.

3.2 Static Fields

- Static fields can never have rep types, because there is no receiver object.

- Static fields with a peer type break type safety, because the type is interpreted relatively to whoever uses the static field.

  Therefore all static fields should be read-only.

  To ease the transition of existing programs we output a warning, but use the peer type as default.

3.3 Object Creation

- Only peer and rep types are allowed for new expressions, as objects always need to have an owner when created and read-only does not determine an owner.
- Constructors with a rep type as parameter cannot be used for an object creation, they can only be used for explicit `this` or `super` constructor calls.

3.4 Instance Method Declarations

- The `this` reference is a peer type of the current class.
- Argument and return types are relative to the receiver object.
- If any argument type has a rep ownership modifier, then the method can only be called on an explicit or implicit `this` as receiver.
- If a method with a rep return type is called on a receiver different from `this`, the call expression has a read-only type. To prevent leaking of representation objects, an appropriate access modifier must be used for the method.

3.5 Instance Method Calls

- The only methods that can be called on read-only references are pure methods.
- If any argument type has a rep ownership modifier, then the method can only be called on an explicit or implicit `this` as receiver.
- If a method with a rep return type is called on a receiver different from `this`, the call expression has a read-only type (case 4 of the type combinator above).

3.6 Static Method Declarations

- Cannot use rep types in their signature, for local variables or anywhere else, because there is no receiver object.

3.7 Static Method Calls

- Can be called as `T.m()` or `peer T.m()` relative to the current context.
- From instance methods they can be called as `rep T.m()` relative to the context owned by `this`.

3.8 Pure Method Declarations

Pure methods must not modify existing objects. Pure methods are declared using the `pure` modifier in front of the method.

- Also for pure method, the `this` reference is a peer type corresponding to the current class.
- All parameter types of pure methods (implicitly) have a read-only ownership modifier. The return type is relative to the context of the receiver, i.e. can be a peer, rep, or read-only type.
- Pure methods can only be overridden by pure methods. The purity modifier is implicitly added to overriding methods if it is missing.

Within pure methods,

- new objects can be created only by pure constructors;
- all field updates are forbidden;
- only pure methods can be called.

3.9 Overriding and Overloading

- To simplify program understanding and the implementation, we forbid overloading of methods, if the signatures only differ in their ownership modifiers.
- For overriding methods, we allow a non-pure method to become pure. Once a method is declared as pure all overriding methods are also (implicitly) pure.

3.10 Constructor Declarations

- In general behave like Instance Method Declarations.
- Constructors with rep types in their signature can only be called by explicit `this` or `super` constructor calls.

3.11 Pure Constructors

The same restrictions as for constructors and pure methods apply, except:

- The instance fields of `this` can be initialized.
- The parameters can be declared as peer or read-only types. The default is the corresponding peer type.

3.12 Exceptions

- The type of the parameter of the `throw` statement is `readonly Throwable`. This means that we put no additional constraint on the context of the exception object.
- The type of the `catch` clauses is also (implicitly) `readonly Throwable`. This ensures that exceptions that are thrown in any context can be caught, without considering the context information at runtime.
- The `throws` declaration of methods always implicitly denotes read-only types. At the moment no ownership modifier can be specified in the `throws` declaration.

3.13 Inner Classes (non-static nested classes)

- References to enclosing instances are `readonly`, because the inner class instance and the enclosing class instance may not be in the same context. A downcast can be used.
- Because of this, non-pure inner class constructors cannot initialize fields of the enclosing class object without using a downcast.

3.14 Miscellaneous

3.14.1 Relationship of `readonly` and `final`

`readonly` and `final` can be combined and express different things.

- A `final` reference cannot be assigned to after initialization. The reference can be used to modify the referenced object.
- A `readonly` reference cannot be used to modify the reachable objects. The reference itself can be assigned to.
- A `final readonly` reference cannot be assigned to after initialization and cannot be used for modifications. This combination should be used e.g. for constants of reference types.

4 Tool Usage

This section describes the usage of the tools that support the Universe type system. Section 4.1 explains the command-line options of the Common JML2 tools and how to compile and run programs with Universe types. Section 4.2 gives details about the runtime checks for the Universe type system. Finally, Section 4.3 introduces the Eclipse plug-in for JML2, which also supports the Universe type system.

4.1 Compiling and running programs with Universe types

To enable support for the Universe type system in JML2 and MultiJava, we added the following command line switches:

- The first option allows fine control over the different features of the compiler.

  ```
  --universesx <String>, -E <String>
       Specify the degree of support for the Universe type
       system (UTS).
  ```

  AVAILABLE OPTION STRINGS

  no UTS features are disabled and no keywords are reserved. Only the \xxx version of the keywords are allowed (all UTS keywords have to be prefixed by a backslash). This is the default.

  parse the UTS keywords are reserved and parsed.

  check UTS typechecking is performed.

  dynchecks code for UTS runtime checks (for downcasts and array updates) is generated. This also turns on the "check" option, because the runtime checks rely on a type-checked program.

  purity purity of methods is checked with a conservative method, which might forbid some methods that do not modify existing objects.

  xbytecode Universe type information is stored in special bytecode attributes. This also turns on the "check" option, because it is important that the stored information is typechecked.
  The resulting class-file is compatible with standard Java VMs.

  annotations Universe type information is stored in Java 5 annotations. This also turns on the "check" option, because it is important that the stored information is typechecked.
  The resulting class-file is compatible with Java 5 VMs.

  full all UTS features except "annotations" are enabled; this corresponds to the --universes flag below.

  The options "no" and "full" must be used alone.

  All other options can be combined by separating them with commas. First all options are turned off and then the given options (and the options implicitly turned on by the given options) are turned on.

- The second option is for turning all features off or on.

  ```
  --universes, -e
       Enable the default Universe type  system  features.
       This corresponds to the "--universesx full" flag.
       This option is disabled by default.
  ```

You can then run the compiled program like any other Java program:

```
java hello
```

If you compiled it with runtime check support (e.g. full Universe support), you need to make sure the `org.multijava.universes.rt` package containing the runtime classes is in the classpath.

4.2 Runtime checks

4.2.1 Using alternative implementations of the runtime classes

The runtime classes are divided in implementation and policy classes. At runtime, one instance of each is created. The implementation class is responsible for storing the ownership structure and performing the checks, while the policy class defines the behavior of the runtime checks. For example, whether invalid casts should generate an exception or just a warning.

Both, the implementation and the policy class, can be replaced by alternative versions independently.

4.2.1.1 Implementation classes Instead of using the default implementation `UrtDefaultImplementation`, it is possible to use an alternative version, either one that also comes with the JML tools or a newly created version.

JML is delivered with four implementation classes:

- `UrtDefaultImplementation` is the default implementation that uses a hashtable to store the ownership structure.
- `UrtDebug` prints out some debugging information to show what tests are done. Apart from that it behaves just like `UrtDefaultImplementation`.
- `UrtDummy` does not perform any checks. It is useful to "disable" the runtime checks for a program without recompiling it.
- `UrtVisualizer` writes a dot-file that can be used to visualize the object store. (dot is part of the Graphviz package (`http://www.graphviz.org/`) and is a tool to create “hierarchical” or layered drawings of directed graphs.) To use this implementation, you need to specify the filename for the output as an additional argument to the JVM on the command line:

  ```
  -DUrtOutfile=<filename>
  ```

To use an alternative implementation, simply specify it as argument to the JVM on the command line. To use `UrtDebug` for example you can call your program like this:

```
java -DUrtImplementation=org.multijava.universes.rt.impl.UrtDebug hello
```

If you specify an invalid implementation class, the program will print a warning and fall back to the default, which is `UrtDefaultImplementation`.

4.2.1.2 Policy classes Instead of using the default policy `UrtDefaultPolicy`, you may use an alternative version, either one that also comes with the JML tools or a newly created version.

JML is delivered with four policy classes:

- `UrtDefaultPolicy` is the default implementation that throws errors on illegal operations and handles external non-Universe objects as peer objects.
- `UrtDummy` does not throw errors and handles external non-Universe objects as peer objects.
- `UrtRelaxed` does not throw an exception on illegal operations but just displays an error message.
- `UrtStrict` handles external objects as read-only, which is type-safe with regards to the Universe type system, but bad for compatibility with legacy code like the Java SDK, since it pretty much disallows the use of any function in the SDK that takes parameters of reference type. This is because any variable, parameter or object that is not annotated is considered to have Universe type peer.

To use an alternative policy, simply specify it as argument to the JVM on the command line. To use `UrtStrict` for example you can call your program like this:

```
java -DUrtPolicy=org.multijava.universes.rt.policy.UrtStrict hello
```

If you specify an invalid runtime class, the program will print a warning and fall back to the default, which is `UrtDefaultPolicy`.

4.2.2 Writing runtime classes

All you need to do is implement the `UrtImplementation` interface if it is an implementation class or the `UrtPolicy` interface if it is a policy class.

4.2.3 Distributing Universe programs with runtime checks

If you distribute your program, you need to deliver it along with the runtime classes in the

```
org.multijava.universes.rt
```

package. The easiest way to ensure this is by using the JAR-file that can be generated with the `univer-se-rt-jar` target in the toplevel Makefile of the MultiJava compiler.

4.3 JML2 Eclipse Plug-In

4.3.1 Overview

This Eclipse plug-in provides a basic integration of the Common JML2 tools into the Eclipse IDE. Originally, this plug-in started as a small part of the Universe type system inference tools, but was spun off later.

4.3.2 Installation

Add the URL `http://www.sct.inf.ethz.ch/research/universes/tools/eclipse/` as an update site to Eclipse. Then search for new plug-ins and install the JML2 Eclipse plug-in.

4.3.3 Requirements

The plug-in was developed using Sun Java 6 and Eclipse 3.4 on Linux. However, the plug-in and the included JML2 were compiled to Java 5 bytecode and should therefore run with an older JVM. Also, Eclipse 3.3 was tested.

4.3.4 Basic Usage

After successful installation, you will find the following three changes: 1) a "JML2 Tools" context menu, 2) a "Run As -> JML Rac" configuration, and 3) "JML2 Plug-in" properties page in the project properties.

In the "Project Explorer", the context menu for files and folders contains a "JML2 Tools" sub-menu with three command: 1) "Run JML2 checker" executes the static type checker and 2) "Run JML2 compiler" checks the code and then compiles it with runtime assertion checking (RAC). 3) "Run JML2 exec" checks the code and then compiles it with the JML Exec tool.

Code that has been compiled with RACs can be executed with "Run As -> JML Rac". If a runtime check fails the exception is printed in the "Console" view.

The project properties contain a "JML2 Plug-in" page that contains the command line options for the checker and the compiler.

5 Example

The following example illustrates the use of the Universe type system for the implementation of a doubly-linked list with an iterator.

```
class Node< T extends readonly Object > {
  peer Node<T> prev, next;
  T elem;
}
```

```
public class LinkedList< T extends readonly Object > {
  /*@ spec_public @*/ rep Node<T> first;

  //@ requires np != null && np.owner == this;
  void set(readonly Node<T> np, T e) {
    rep Node<T> n = (rep Node<T>) np;
    n.elem = e;
  }

  public /*@ pure @*/ boolean equals(readonly Object l) {
    if (!(l instanceof readonly LinkedList<T>))
      return false;
    readonly Node<T> f1 = first;
    readonly Node<T> f2 = ((readonly LinkedList<T>)l).first;
    while (f1 != null && f2 != null && f1.elem == f2.elem) {
      f1 = f1.next;
      f2 = f2.next;
    }
    return f1 == null && f2 == null;
  }
  // constructors and other methods omitted
}
```

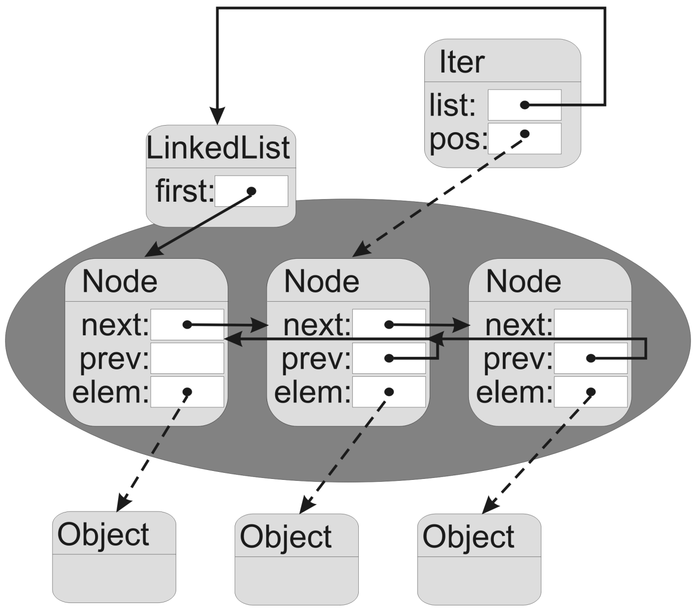

Figure 4: Object structure of the collection example. The LinkedList object owns the nodes of the doubly-linked list. The iterator is in the same context as the list head. It has a reference to the list head and a read-only reference to the Node object at the iterator position. The nodes have read-only references to their element objects.

```
public class Iter< T extends readonly Object > {
  /*@ spec_public @*/ peer LinkedList<T> list;
  /*@ spec_public @*/ readonly Node<T> pos;
  //@ invariant pos != null && pos.owner == list;

  public Iter(peer LinkedList<T> l) {
    list = l;
    pos = l.first;
  }

  public void setValue(T e) {
    list.set(pos, e);
  }
  // other methods omitted
}
```

6 References

- For current information see the project page:
  http://sct.inf.ethz.ch/research/universes/
- The research results are available from our publication page:
  http://sct.inf.ethz.ch/publications/index.html
- For general information about the Software Component Technology group see:
  http://sct.inf.ethz.ch/
- Homepage of Werner M. Dietl:
  http://sct.inf.ethz.ch/people/dietl/
- Homepage of Peter Müller:
  http://sct.inf.ethz.ch/people/mueller/

7 Bibliography

[1] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer-Verlag, 2007.

[2] W. Dietl and P. Müller. Exceptions in ownership type systems. In E. Poll, editor, *Formal Techniques for Java-like Programs*, pages 49–54, 2004.

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

[4] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. JML reference manual. Available from http://www.jmlspecs.org/, 2008.

[5] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[6] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

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

8 Project Reports

[8] Paolo Bazzi. *Integration of Universe Type System Tools into Eclipse*. Semester Project, Summer 2006.

[9] Andreas Fürer. *Combining Runtime and Static Universe Type Inference*. Master's Thesis, 09/06 – 03/07.

[10] Ovidio Mallo. *MultiJava, JML, and Generics*. Semester Project, Summer 2006.

[11] Mathias Ottiger. *Runtime Support for Generics and Transfer in Universe Types*. Master's Thesis, 02/07 – 08/07.

[12] Annetta Schaad. *Universe Type System for Eiffel*. Semester Project, Summer 2006.

[13] Daniel Schregenberger. *Universe Type System for Scala*. Master's Thesis, 09/06 – 06/07.

[14] Daniel Schregenberger. *Dynamic Typechecking in the Universe Type System*. Semester Project, Summer 2004.

[15] Manfred Stock. *Implementing a Universe Type Checker in Scala*. Master's Thesis, 08/07 – 02/08.

[16] Alex Suzuki. *Bytecode support for the Universe type system and compiler*. Semester Project, Winter 2004/05.

[17] Dirk Wellenzohn. *Implementation of a Universe type checker in ESC/Java2*. Semester Project, Summer 2005.

[18] Robin Züger. *Generic Universe Types in JML*. Master's Thesis, 01/07 – 07/07.