

Bogor Software Model Checking Framework: User Manual

[Next](#)



Bogor Software Model Checking Framework: User Manual

[Robby](#)

Matt Hoosier

Edwin Rodriguez

Copyright © 2004 [SAnToS Laboratory, Kansas State University](#)

Table of Contents

[1. Introduction](#)

[Bogor Overview](#)

[Installing Bogor](#)

[Using Bogor](#)

[Model Checking BIR Models](#)

[Counter-example Display](#)

[Random Simulation Mode](#)

[User-guided Simulation Mode](#)

[Configuring Bogor](#)

[Bogor Command-line Interface](#)

[2. BIR: Syntax](#)

[Systems](#)

[Identifiers](#)

[Types](#)

[Literals](#)

[Constants](#)

[Enumerations](#)

[Records](#)

[Extensions](#)

[Global Variables](#)

[Threads and Functions](#)

[Virtual Tables](#)

[Functional Expressions](#)

[Expressions](#)

[3. Bogor Architecture](#)

[ISearcher: State Space Exploration](#)

[4. Extensions](#)

[Language Extension](#)

[Syntax Extension](#)

[Semantics Implementation](#)

[Module Extension](#)

[A. BIR Reference](#)

[Syntax](#)

[Semantics](#)

[Keywords](#)

[B. Bogor API](#)

[Next](#)

Chapter 1. Introduction

Chapter 4. Extensions

[Prev](#)[Next](#)

Chapter 4. Extensions

Table of Contents

[Language Extension](#)[Syntax Extension](#)[Semantics Implementation](#)[Module Extension](#)

Language Extension

In this section, we will use the design, specification, and implementation of *set* native to the BIR language as a running example of enhancing Bogor's input language. This simple, yet illustrative, example highlights both the reasons why one would wish to add direct support for an abstract datatype to BIR and the Bogor infrastructure used to achieve this effect at the cost of little developer time.

Why, one may ask, is the ability to implement complex datatypes as native BIR constructs so important? Occasionally, the model writer wishes to peek "under the hood" of the model checking engine and inspect some property of the system state. In the best case, this is extremely ungainly to do directly in BIR (and usually requires instrumenting the model with auxiliary variables, thereby inflating the state space). Often, the property at hand is simply inexpressible in BIR. For example, one may wish to query if a heap-allocated object is reachable from references beginning at some distinguished object. There is no reasonable way to do this directly in the BIR language; yet, an extremely simple *language extension* operation implemented with perhaps 20-30 lines of Java code can easily decide this reachability query.

More importantly, language extension *types* and *operations* enable a tremendous amount of abstraction. To take our running example (a mathematical set), the elements must be kept inside of some container type. Absent the language extension mechanism, this could be done directly using BIR arrays. But then the modeler must confront the usual problems associated with implementing set container types: *ensuring element uniqueness* and *expanding the underlying storage* (here, the BIR array) as the set grows are but the most obvious ones. Last, demands unique to model checking impose themselves. Mathematical sets are unordered, but arrays in model checking input languages are not. If the ordering of two array elements is reversed, a model checker will consider these states to be distinct, and in both cases the entire state space descending from these semantically equal, but syntactically different, set encodings will be traversed. This forces the model writer to use some technique for imposing a canonical ordering on the elements. All this is possible in the BIR language, but at the prices of steep runtime penalty (both in memory requirements for the now-larger seen state set, and in time to interpret the added command sequences which manipulate the set data structures) and great confusion of the code expressing the actual domain problem to be checked. By introducing new, encapsulated BIR native datatypes and operations to manipulate them, this code and complexity can be pushed into the model checker runtime environment.

Syntax Extension

In our example, we will define a set abstract datatype which supports the following operations:

- insertion of elements,
- removal of elements,
- emptiness test,
- automated evaluation of predicates across all elements, and
- nondeterministic selection of a contained element.

Informing the Bogor language recognizer about the new datatypes is quite easy. In contrast to the input mechanism of some model checkers (e.g., SPIN) that require modification of the parser itself if new native types must be added, the BIR language allows use of a simple *extension declaration* block which specifies both the identifiers used for the new types/operations and strong typing information about the arguments and return values of said operations. We show the declaration of new primitives for the set abstract datatype in [Figure 4.1, "Declaration of set ADT and operations"](#).

Figure 4.1. Declaration of set ADT and operations

```
extension Set for edu.ksu.cis.projects.bogor.module.set.SetModule
{
  typedef type<'a>;
  expdef Set.type<'a> create<'a>('a ...);
  expdef 'a choose<'a>(Set.type<'a>);
  expdef boolean isEmpty<'a>(Set.type<'a>);
  actiondef add<'a>(Set.type<'a>, 'a);
  actiondef remove<'a>(Set.type<'a>, 'a);
  expdef boolean forAll<'a>('a -> boolean, Set.type<'a>);
}
```

The first line declares the extension namespace, `Set`. We do so by using the keyword `extension` followed by the name of the extension, `Set`. Then, we use the keyword `for` to specify the Java class which provides the semantics implementation. We will return to a discussion of their implementation in [the section called "Semantics Implementation"](#).

Next comes the declaration of the new native type itself. This identifier, `Set.type`, is the type of any set ADT variable which will be declared later. Note that the set type (and all operations to follow) are parameterized by a *single type variable* (here, `'a`) set between angle brackets. This allows us to write our set implementation in a completely generic manner so that it is not hard-wired to contain only a specific type of elements, e.g., `int`. Users familiar with the C++ template mechanism or SML's polymorphic type system will quickly recognize the syntax for polymorphism in BIR.

When a Bogor abstract type is declared using placeholder type variables (`'a`), it can be instantiated in

a model with any other legal type used as elements. For instance, the following BIR fragment declares two sets with mutually incompatible element types:

```
Set.type<int> anIntSet;
Set.type<string> aStringSet;
```

If we were not interested in making the `Set` type and operations generalized, we could instead just declare the type as follows:

```
extension Set for edu.ksu.cis.projects.bogor.module.set.SetModule
{
  typedef type;

  expdef Set.type create(int ...);

  expdef int choose(Set.type);

  ...
}
```

This would in turn force us to hard-wire the parameter types of the operators `create`, `add`, `remove`, etc. (we have shown this done with `int`). As we will see later, supporting polymorphism of extensions requires some additional code in the Java modules which implement the set semantics. The generality and robustness gained, though, are usually worth the extra time spent up-front.

The remaining lines in [Figure 4.1, “Declaration of set ADT and operations”](#) are used to declare the operations allowed on `Set.type` ADT. These come in two types, split along the same lines as regular BIR constructs. *Expressions* are mandated to be side-effect-free; operators which serve as expressions are declared using the `expdef` keyword. *Actions* transform the state of a BIR system; we declare the operations which perform state changed using the `actiondef` keyword.

Two operations in particular deserve special mention. First, the expression `choose` nondeterministically retrieves an element from the set. We will see later how Bogor's scheduler can be made to exhaustively retrieve each element from the set so that the state space traversal explores every possible outcome from a `choose` expression. Second, the `forAll` expression demonstrates Bogor's first-class support for functions. The first argument (whose signature is `'a -> boolean`) is the name of a function which accepts the set's contained element type and returns a yes/no answer. We will see that the implementation of `forAll` applies the function named as this argument is applied to each element of the set as a predicate; the operation returns `true` if and only if the predicate holds on every element.

At this point, we have shown all the syntax necessary to declare extension datatypes in Bogor. The markup from [Figure 4.1, “Declaration of set ADT and operations”](#) is enough to enable Bogor's type-checking mechanism to verify that the extensions have been used safely. We have, though, only given an informal description of the intended behavior of the set operations. In the next section, we will describe the process of writing Java implementations of these operations to fix their semantics.

Semantics Implementation

Once the syntax for the extension has been defined and included in the model, we now shift our attention to implementing the module which provides the extension's functionality. Recall that, as shown in [Figure 4.1, “Declaration of set ADT and operations”](#), we name the Java class which implements a BIR language extension's operations:

```
extension Set for edu.ksu.cis.projects.bogor.module.set.SetModule
{
  ...
}
```

We are responsible, then, for providing a Java class `SetModule` which exposes one method for each operation (both the `actiondef` and `expdef` varieties) named in the extension. The Bogor framework includes a Java interface, `IModule`, which any class providing a language extension module must implement. We thus begin by creating a class `SetModule` which is tagged as conforming to the `IModule` interface:

```
package edu.ksu.cis.projects.bogor.module.set;

public class SetModule implements IModule
{
  ...
}
```

Because we have tagged our Bogor module class as implementing `IModule`, we must provide an implementation for each of its required methods (shown in [Figure 4.2, “Required methods for a Bogor extension module \(IModule.java\)”](#)). Of the three methods shown, only `connect` usually has a non-empty body. Intuitively, this is because `connect` establishes the module's links to the main Bogor model checking components, while `getCopyrightNotice` and `setOptions` are sometimes used to display legal messages and configure advanced options.

Figure 4.2. Required methods for a Bogor extension module (`IModule.java`)

```
public interface IModule extends Disposable
{
  String getCopyrightNotice();

  void setOptions(
    String key,
    Properties options);

  void connect(IBogorConfiguration bc);
}
```

We will follow suit, making `getCopyrightNotice` return the null string and `setOptions` do nothing. We will define a `connect` method that acquires references to some of the Bogor runtime components, and a `dispose` method required by the fact our `IModule` implementation is transitively required to implement Bogor's `Disposable` interface too:

```

// acquire references to runtime modules
public void connect(IBogorConfiguration bc)
{
    tf = bc.getSymbolTable().getTypeFactory();
    ee = bc.getExpEvaluator();
    ss = bc.getSchedulingStrategist();
    vf = bc.getValueFactory();
}

// release references to runtime modules
public void dispose()
{
    tf = null;
    ee = null;
    ss = null;
    vf = null;
}

```

In our case, we have chosen to store references to Bogor's type factory, main expression interpreter, scheduling policy, and value creation mechanism. This selection is dictated by the subset of functionality we'll need to maintain the set representation. In principle, an extension module can connect to any or all of the nine core modules ([Figure 3.1, "Core model-checking components"](#)). If in doubt, we suggest that extension writers initially not save any references to Bogor modules. These can always be added later during development as needed.

Operations

The code shown so far is a minimal (i.e., zero functionality) Bogor extension. It can be loaded into the runtime and acquire model checker component references, but because no additional methods beyond those required by `IModule` have been defined, it cannot manipulate any extension types. We must add one Java method into the `SetModule` class for each operation (e.g., `create`, `choose`, and `add`) shown in [Figure 4.1, "Declaration of set ADT and operations"](#).

Recall that every extension operation is either an `expdef` or an `actiondef`. The Java method implementing each of these two varieties of operations takes a fixed signature. In [Figure 4.3, "Method signatures for extension operator implementations"](#) we see the pattern for writing both action and expression operator implementations.

Figure 4.3. Method signatures for extension operator implementations

```

// expdef variety
public IValue expOperationName(IExtArguments args)
{
    ...
}

// actiondef variety
public IBacktrackingInfo[] actionOperationName(IExtArguments args)
{
    ...
}

```

Note

The sole parameter to any operation implementation is an `IExtArguments` instance. This data structure is a generic mechanism used to encapsulate all arguments, polymorphic type variables, and even syntax tree nodes from the modeling language. The Bogor runtime passes this information down to individual extension modules as a bridge between high-level language constructs and implementation level code.

Note

All expression implementations return an `IValue` instance. This is the interface type which all Bogor values implement. Because expressions are always side-effects-free and always evaluate to a value, every expression operation *will* produce an `IValue` to return. All action operations, on the other hand, return a sequence of `IBacktrackingInfo` instances. These function as the "undo" operations that allow Bogor to reverse its path on the depth-first stack when backtracking is needed. Because actions *never* evaluate to a value, an action operation never needs to return an `IValue`.

We begin with the `create` expression operation. It functions as a pseudo-constructor, building a set value to contain zero or more initial elements. The Java method will also be named `create`, and it has the same form as the expression method implementation in [Figure 4.3, "Method signatures for extension operator implementations"](#):

```

public IValue create(IExtArguments args)
{
    ....
}

```

This provides the signature of the method. But how is the value which is stored in `BIRSet.type<'a>` variables produced? We will discuss strategies for writing Java code to represent this value type later in [the section called "Value Types"](#) and for now just note that there are significant differences between the methods one uses to handle `IValue` objects representing Bogor primitives types (e.g., numerical data and enumerated types) and Bogor reference types (e.g., strings, records). The bulk of `create`'s body is devoted to detecting which type of element the set will contain and adjusting appropriately. We show here the entire method body except code used to detect and adapt to primitive element types (it has been elided as noted with an ellipsis):

```

Type argType = (Type) arg.getTypeVariableArgument(0);
Type setType = arg.getExpType();

if (argType instanceof NonPrimitiveType)
{
    // builds an empty set
    ISetValue result = new ReferenceElementSetValue((NonPrimitiveExtType)
        vf.newReferenceId());

    // add the arguments to the set
    int size = arg.getArgumentCount();
}

```

```

    for (int i = 0; i < size; i++)
    {
        result.add(arg.getArgument(i));
    }

    return result;
}
else if (argType instanceof PrimitiveType)
{
    ...
}
else
{
    assert false;
    throw new RuntimeException("Unimplemented extension");
}

```

We inspect the type of elements which will populate the set by retrieving the first *type variable* (our operation uses only one: 'a'). The inheritance hierarchy for Bogor types (rooted at `edu.ksu.cis.projects.bogor.type.Type`) shows that every value-bearing types descends from either `PrimitiveType` or `NonPrimitiveType`. Because all non-primitive types are allocated on the heap, and thus variables of these types are references, we group them together and use a particular set `IValue` implementation suited to manipulating contained elements as reference types. In the code we have elided, a different set implementation able to cope with primitive-typed elements is constructed and returned instead.

Note

Pleasantly, we are able to localize the tedious inspection of element types to only the `create` method. The block of code dealing with `PrimitiveType` is able to configure its dedicated `ISetValue` implementation to autonomously deal with all the different Bogor primitive types. So, the rest of our operation implementations are much tidier to list.

After the reference-type-specific set value is created, we just iterate over all the arguments (initial elements of the set) and add them one by one to the set implementation. The `ISetValue.add(IValue)` method is sufficient to do this. Finally the set value is returned as an `IValue` to complete the operation.

Next, consider the `choose` expression. This leverages the functionality of Bogor's scheduling module to non-deterministically pick and return an object from the set. Recall the listing of its type and arguments from [Figure 4.1, "Declaration of set ADT and operations"](#):

```
expdef 'a choose<'a>(Set.type<'a>);
```

As an expression operator, then, `choose` will be implemented by a Java method in `SetModule.java` with the following signature:

```

public IValue choose(IExtArguments arg)
{
    ...
}

```

The procedure for nondeterministically picking an element from set turns out to be surprisingly simple. Here is the body of `choose`:

```

// gets the elements of the set
ISetValue set = (ISetValue) arg.getArgument(0);
IValue[] elements = set.elements(); ❶

int size = elements.length;
int index = 0;

if (size > 1) ❷
{
    // ask the scheduler which one should be picked now
    index = ss.advise(❸
        arg.getExtDesc(),
        arg.getNode(),
        elements,
        arg.getSchedulingStrategyInfo());
}

// returns the one picked by the scheduler
return elements[index];

```

- ❶ The set's `elements` method returns the contained objects in some canonical order, so that the i^{th} element of the array is the same each time that the transition is executed.
- ❷ If the set only contains one element, nondeterminism is not needed.
- ❸ This seems a little like "magic." When the `advise` call is made on the scheduler `ss`, Bogor interprets this as a request to branch the state space exploration. Thus `advise` will successively return the values 0, 1, ... `elements.length - 1`.

The first time `advise` is executed, the scheduler returns 0 and makes a note that once the DFS bottoms out and backtracks, the current transition (from which `choose` was evaluated) should be re-played. The next time that `advise` is consulted, it returns 1. Again the ensuing DFS again bottoms out and backtracks to the current transition. This continues, until `advise` returns `elements.length - 1`. At this point, the scheduler notes that all possible members of the `elements` array have been explored. When the backtracking following the `elements.length - 1` response moves back past the transition using `choose`, the scheduler simply declines to order an additional DFS. The non-deterministic exploration is complete.

The implementation of the next operation, `isEmpty`, is extremely easy. The novel technique to observe is the use of the value factory (`IValueFactory`) to create a BIR value wrapping the boolean result. After extracting the set from the `IExtArguments` passed down from Bogor's runtime, the helper method `getBooleanValue` uses the value factory to create a new boolean `IValue`:

```
public IValue isEmpty(IExtArguments arg)
```

```

{
    // gets the set
    ISetValue set = (ISetValue) arg.getArgument(0);

    // returns a boolean value depending on the emptiness of the set
    return getBooleanValue(set.isEmpty());
}

protected IValue getBooleanValue(boolean b)
{
    return vf.newIntValue(
        tf.getBooleanType(),
        b ? 1
          : 0);
}

```

Next is the most complicated expression we'll see: `forall`, which applies a predicate to all elements of the set. The signature for the BIR expression from [Figure 4.1, "Declaration of set ADT and operations"](#) was this:

```
expdef boolean forall<'a>('a -> boolean, Set.type<'a>);
```

While the description of `forall`'s semantics sounds intimidating, we are able to delegate most of the work to Bogor's expression evaluator module to determine if the predicate holds on each element. Our job then, is just to iterate across all the elements contained in the set:

```

public IValue forall(IExtArguments arg)
{
    // get the fun name of the function
    String predFunId = ((IStringValue)
        arg.getArgument(0)).getString(); ❶

    // get the set elements
    ISetValue set = (ISetValue) arg.getArgument(1);
    IValue[] elements = set.elements();

    // assume all true
    boolean result = true;

    // for each element, apply the function
    // if it returns false for at least one element, then there exists
    // an element that does not satisfy the condition
    for (int i = 0; i < elements.length; i++)
    {
        IValue element = elements[i];
        IIntValue val = (IIntValue) ee.evaluateApply(
            predFunId,
            new IValue[] { element });

        if (val.getInteger() != 1) ❷
        {
            result = false;
            break;
        }
    }
}

```

```

}

return getBooleanValue(result); ❸
}

```

- ❶ What's going on here? The function's type is listed as 'a -> boolean, not string! Because functions are not simply values which can be copied around, the predicate is transmitted to the Java implementation of `forall` by sending the name of the function's identifier. Once "under the hood," we can use the expression evaluator module to evaluate the function just by passing its name and the argument(s).
- ❷ Bogor, like the Java Virtual Machine, encodes boolean values as 0- or 1-valued integers. So the result of the boolean predicate is communicated back as an `IIntValue`.
- ❸ Another use of the value factory to create a wrapped primitive value suitable for Bogor's interpreter engine.

This completes the complement of expression operators we have specified on the set extension. Next, we take a look at the techniques for implementing destructive-update operations (in BIR terms, `actiondefs`). The Java code providing a BIR action is fundamentally different than its expression cousins: rather than returning a value, the method must supply "undo" objects which can be executed to return the BIR system to the state just before the action was taken.

Our set extension to the BIR language only includes two side-effecting operations: `add` and `remove`. Conceptually, both are extremely simple and very nearly mirror operations, so we will only examine the former, `add`.

Our basic strategy for implementing the set insertion operator is to case-split. If the "new" element is already in the set, then execution aborts immediately and no undo information is needed (we have made no change to the state). If, on the other hand, the proposed addition is valid, then we insert the element into the set and return an `IBacktrackingInfo` object which, when executed, will remove the new element from the set. The code is given below.

```

public IBacktrackingInfo[] add(IExtArguments arg)
{
    // get the set
    ISetValue set = (ISetValue) arg.getArgument(0);

    // get the element to be added
    IValue element = (IValue) arg.getArgument(1);

    if (!set.contains(element)) ❶
    {
        // add the element
        set.add(element);

        ISchedulingStrategyContext ssc =
            arg.getSchedulingStrategyContext();

        // create the backtracking infos
        return new IBacktrackingInfo[]
        {
            createAddBacktrackingInfo(❷
                set,
                element,

```

```

        arg.getNode(),
        ssc.getStateId(),
        ssc.getThreadId(),
        arg.getSchedulingStrategyInfo())
    };
}
else
{
    // do nothing, so empty backtracking info
    return new IBacktrackingInfo[0];
}
}

```

- ❶ The exact procedure for testing whether an element already belong to the set depends on the static type of the element (e.g., `int` versus a reference type). Each `ISetValue` implementation will define its `contains` method in a way meaningful to its domain.
- ❷ The real work of creating the undo operation object is delegated to a helper function.

While it demonstrates how to add elements to a set, this code listing doesn't really help us to understand the key problem: how to create the `IBacktrackingInfo` undo operations. To see this, we have to look behind the implementation of the helper function. Here, then, is the complete body of `createAddBacktrackingInfo` in full glory:

```

protected IBacktrackingInfo createAddBacktrackingInfo(
    final ISetValue set,
    final IValue element,
    final Node node,
    final int stateId,
    final int threadId,
    final ISchedulingStrategyInfo ssi)
{
    return new IBacktrackingInfo()
    {
        public Node getNode()
        {
            return node;
        }

        public ISchedulingStrategyInfo
        getSchedulingStrategyInfo()
        {
            return ssi;
        }

        public int getStateId()
        {
            return stateId;
        }

        public int getThreadId()
        {
            return threadId;
        }

        public void backtrack(IState state)
        {
            set.remove(element);
        }
    };
}

```

```

    }

    public IBacktrackingInfo clone(Map cloneMap)
    {
        return createAddBacktrackingInfo(
            (ISetValue) cloneMap.get(set),
            (IValue) ((element instanceof
                INonPrimitiveValue)
                ? cloneMap.get(element)
                : element),
            node,
            stateId,
            threadId,
            ssi.clone(cloneMap));
    }

    public void dispose()
    {
    }
}
};
}

```

Most of the `IBacktrackingInfo` methods just pass along simple data such as (1) the syntax tree node for the `actiondef` executed (2) a process descriptor recording which BIR thread executed the code (3) and the state vector ID before the action executed. Only two methods need special attention. First, `backtrack` is the actual "undo" undo operation. In this case, it just needs to remove the element inserted by `add`. Second, the `clone` method; this one's purpose is a little more indirect. Although this method is not used in the current Bogor codebase, it would be used if a future version of Bogor cloned the system state (`IState`); because the object references in a deep state clone would change, a clone of the backtracking operation would be necessary also in order to operate on this new forest of references.

Value Types

We have, so far, engaged in vigorous hand-waving about the nature of the value types in BIR language extensions. Most notably, while describing the implementation of the `create` expression operation, we hinted at the fact that any one of several different classes may implement the same abstract `Set.type<'a>` type. In this section we will examine Bogor's mechanism for grafting new nodes onto the internal type hierarchy. We shall also see how to a bridge between (1) low-level storage of values and (2) high-level model checking requirements such as state vector creation, heap ordering, and garbage collection.

Figure 4.4. Required methods for any Bogor ADT extension type (`INonPrimitiveExtValue.java`)

```

public interface INonPrimitiveExtValue
    extends INonPrimitiveValue, Serializable
{
    // Methods required directly

    void externalize(
        PrintWriter pw,
        INonPrimitiveValueIdTracker nvpIdTracker);
}

```

```

byte[][] linearize(
    int bitsPerNonPrimitiveValue,
    ObjectIntTable nonPrimitiveValueIdMapd,
    int bitsPerThreadId,
    IntIntTable threadOrderMap);

void visit(
    IValueComparator vc,
    boolean depthFirst,
    Set seen,
    LinkedList workList,
    IValueVisitorAction vva);

// Methods required by INonPrimitiveValue

int getReferenceId();

// Methods required by IValue

Type getType();

int getTypeId();

IValue clone(Map cloneMap);

void validate(IBogorConfiguration bc);
}

```

Figure 4.5. Required operations for all set types (ISetValue.java)

```

public interface ISetValue extends INonPrimitiveExtValue
{
    void add(IValue v);

    boolean contains(IValue v);

    IValue[] elements();

    boolean isEmpty();

    void remove(IValue v);
}

```

All value classes in Bogor must implement the `IValue` interface. Usually, this is done indirectly by implementing one of its descendants. There are two value interfaces in particular which extension types may choose to implement: `IPrimitiveExtValue` and `INonPrimitiveExtValue`. In our case, a set is certainly not a primitive (atomic) type. Rather, it is an abstract datatype that contains other objects. As such, our Bogor value class must implement `INonPrimitiveExtValue` (the required methods are shown in [Figure 4.4, “Required methods for any Bogor ADT extension type \(INonPrimitiveExtValue.java\)”](#)). The interface requires three methods: `linearize`, `visit`, and `externalize`, as well as those that its parent interfaces require. We will study the example implementations of these methods in `ReferenceElementSetValue.java`.

As an exercise in good design methodology, we have also created a new Java interface, `ISetValue`, which specializes the `INonPrimitiveExtValue` type. See [Figure 4.5, “Required operations for all set types \(ISetValue.java\)”](#) for a listing of its required operations. Our running example, `ReferenceElementSetValue.java` is an implementation of `ISetValue`.

The first of these, `linearize`, is the critical bridge from the Java world of objects, scalar types, and hierarchical structure to the model checker's *state vector*. Recall that model checkers use a *seen-before set* to record which states in a computation tree have already been visited. This is vital to the depth-first space exploration, because it provides a means to prove termination when the state space is finite. Algorithms listings usually abstract away from the implementation of this set. In Bogor, the seen-before set is a collection of bit sequences which encode states. A state is said to be in the seen-before set if and only if the seen-before set contains a bit sequence equal to the bit sequence which encodes *s*. The `linearize` method is tasked with walking the internal structure of an extension type and producing a sequence of bits which uniquely encodes the state information about the datatype. Because a typical state space exploration will accumulate tens of thousands or more distinct states' bit-vectors in memory simultaneously, it is imperative that the bitwise encoding produced by `linearize` is efficient. Let's glance at the signature:

```

byte[][] linearize(
    int bitsPerNonPrimitiveValue,
    ObjectIntTable nonPrimitiveValueIdMapd,
    int bitsPerThreadId,
    IntIntTable threadOrderMap);

```

The return value is a sequence of byte arrays (normally, the outer dimension is only one deep). The first and third arguments give the minimum number of bits needed to encode process descriptors and heap location identifiers. The second argument gives a mapping from the `IValue` objects representing heap-allocated types a unique identifier for each such object. We will use this shortly. In the `ReferenceElementSetValue` version of this method (its body is shown next), note how the unique object ID is fetched for each set element. After sorting these ID's (because sets are unordered and we wish our linearization to be a symmetric set), the bit-vector is constructed by just concatenating the binary representation of all these object ID's.

```

public byte[][] linearize(
    int bitsPerNonPrimitiveValue,
    ObjectIntTable nonPrimitiveValueIdMap,
    int bitsPerThreadId,
    IntIntTable threadOrderMap)
{
    IValue[] elements = elements();

    BitBuffer bb = new BitBuffer();

    int[] elementIds = new int[elements.length];

    for (int i = 0; i < elements.length; i++)
    {
        elementIds[i] = nonPrimitiveValueIdMap
            .get(elements[i]);
    }
}

```

```

Arrays.sort(elementIds);

for (int i = 0; i < elements.length; i++)
{
    bb.append( ❷
        elementIds[i],
        bitsPerNonPrimitiveValue); ❸
}

return new byte[][] { bb.toByteArray() };
}

```

- ❶ The unique ID of each set element object is retrieved. It is this value which gets written to the state vector.
- ❷ Note the use of the `BitBuffer` to handle construction of bitwise encodings. Using its convenience methods is the usual idiom when writing a linearization method.
- ❸ We must specify the number of bits to use when encoding the element ID into the bit vector. It happens that Bogor has pre-calculated this for our particular element type, but usually this must be manually calculated with `Util.widthInBits()`.

The contents of the two-dimensional byte array which we return are appended to the overall bit-vector, and a linearized representation of the overall system state is created. One factor important to correctly implement a datatype linearization is to always impose an outside ordering on the elements if their order of occurrence in the container is not relevant. If we implemented a `Queue` extension type, the ordering of elements is important to the semantics; it is after all a first-in, first-out data structure! But the semantics of a set care not whether the most recently inserted element is stored at the end of the underlying container or at the beginning. To prevent these two conditions from artificially producing different bit vectors, we first impose a total ordering on the elements by sorting according to their object ID's.

Next we move to the `visit` method. This method is called as part of an overall traversal that collects all `IValues` in the Bogor system. This is needed for several analytical purposes: garbage collection, obtaining canonical representations of the heap, etc. Its signature is given by the following listing:

```

public void visit(
    final IValueComparator vc,
    boolean depthFirst,
    Set seen,
    LinkedList workList,
    IValueVisitorAction vva)
{
    ...
}

```

The interesting bit about `visit` is that the traversal which it helps to implement is *ordered*. First, the search can be either a pre- or post-order traversal, as controlled by the `depthFirst` flag. A pre-order traversal will set this flag to `true`; this means that all `IValues` contained inside the our ADT should be prepended to the *beginning* of the `workList` queue. If, on the other hand, the flag is `false`, this indicates a post-order traversal; all contained values should be appended to the *end* of the `workList`

queue.

Second, the elements themselves should be inserted in a particular order. If our set contained BIR primitive values as elements, then we could use the `IValueComparator` argument as a means to sort them. Because the set implementation we consider here stores references values, though, we must sort them in an ad hoc manner. For our purposes, the ordering by heap ID already done by the `elements()` method call is good enough to ensure a symmetric representation. This done, we simply add the elements to the work queue according to whether the traversal is depth-first or breadth-first:

```

public void visit(
    final IValueComparator vc,
    boolean depthFirst,
    Set seen,
    LinkedList workList,
    IValueVisitorAction vva)
{
    IValue[] elements = elements();

    if (depthFirst)
    {
        for (int i = 0; i < elements.length; i++)
        {
            workList.addFirst(elements[i]);
        }
    }
    else
    {
        for (int i = 0; i < elements.length; i++)
        {
            workList.add(elements[i]);
        }
    }
}

```

The last of the methods required for all nonprimitive extension types (as shown in [Figure 4.4](#), “Required methods for any Bogor ADT extension type (`INonPrimitiveExtValue.java`)” is `externalize`. Here, the abstract contents of the data structure are written to an output stream (usually in the form of XML). The implementation of this method is not interesting; the reader can consult `ReferenceElementSetValue.java` for a representative example.

Finally, we still must implement the methods required by the set interface itself (shown in [Figure 4.5](#), “Required operations for all set types (`ISetValue.java`)”). Our set implementation uses a standard Java Collections `java.util.HashSet` to store its elements, because by happenstance there is a one-to-one correspondence between BIR heap-allocated objects and the Java object instances which implement them. This means that the default `java.lang.Object.equals()` method that compares Java object references is sufficient to preserve uniqueness of BIR objects, and we can use a Collections set as a convenient container. Thus we add a set container as a field to our `ReferenceElementSetValue` class:

```
protected HashSet set = new HashSet();
```

Of the five required `ISetValue` methods, four are now reduced to one-liners: `add`, `contains`, `isEmpty`, and `remove`. They just delegate the work to the `HashSet` container. Because all four are substantially similar, we just show the implementation of `add`:

```
public void add(IValue v)
{
    set.add(v);
}
```

Only the `elements` method has an interesting implementation. Here, we must retrieve the BIR set's elements from the delegated container and sort them by increasing order of their heap identifier:

```
public IValue[] elements()
{
    Object[] elements = set.toArray();
    orderValues(elements);

    IValue[] result = new IValue[elements.length];
    System.arraycopy(
        elements,
        0,
        result,
        0,
        elements.length);

    return result;
}

protected void orderValues(Object[] values)
{
    Arrays.sort(
        values,
        new Comparator()
        {
            public int compare(
                Object o1,
                Object o2)
            {
                int refId1 = ((INonPrimitiveValue) o1)
                    .getReferenceId();
                int refId2 = ((INonPrimitiveValue) o2)
                    .getReferenceId();

                return Util.compare(
                    refId1,
                    refId2);
            }

            public boolean equals(Object obj)
            {
                return this == obj;
            }
        });
}
```

This completes our walkthrough of the procedure and concerns involved when adding a new native datatype to BIR. The complete sources (`SetModule.java`, `ISetValue.java`, `ReferenceElementSetValue.java`, and `PrimitiveElementSetValue.java`) are available in the public [GudangBogor repository](http://gudangbogor.projects.cis.ksu.edu) reachable from <http://gudangbogor.projects.cis.ksu.edu> in addition to being available bundled here:

- [SetModule.java](#)
- [ISetValue.java](#)
- [ReferenceElementSetValue.java](#)
- [PrimitiveElementSetValue.java](#)

[Prev](#)

Chapter 3. Bogor Architecture

[Up](#)

[Home](#)

[Next](#)

Module Extension