Bogor Extensions Tutorial

Edwin Rodríguez
SAnToS Laboratory
Department of Computing and Information Sciences
Kansas State University
edwin@cis.ksu.edu

30th January 2004

Introduction

Bogor is a fully extensible and highly customizable model checker developed at the Laboratory for Specification, Analysis and Transformation of Software (SAnToS) at Kansas State University. It is designed so that both its input language and checking engine can be extended so as to exploit domain dependent characteristics to reduce space and time requirements during model checking. In this document we explore Bogor's input language extensions.

Bogor Input Language Extension

Bogor's input language name is BIR. One of the most interesting features of BIR is its extension facility. It allows the designer to add new semantic primitives to the language which then are implemented as Bogor modules. Let us consider a simple example.

We will model the allocation/deallocation of resources in a resource pool where we have two types of resources: disks and displays. A straigthforward representation for a resource pool is a set. But BIR primitives do not include sets. We could model the resource pool as a record. If we do this then we have to encode all the constraints that define the properties of a set as part of the model. All this adds to the model complexity and increases the properties that need to be checked. But in this example our interests is not the behavior of a set, but instead we want to model and check the dynamics of the resource pool. What we can do instead is extend the language with a new primitive: a set.

A complete model for this example is included in the file *resourcePool.bir* along with this document. The advantage of modeling the set as an extension to the language is that now the set properties won't add to the model complexity and the model checker can focus in the real problem instead of the representation. Let us see how we would

make this extension in Bogor:

```
extension Set for myPackage.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>);
  expdef boolean forAll<'a>('a -> boolean,
                            Set.type<'a>);
  actiondef add<'a>(Set.type<'a>, 'a);
  actiondef remove<'a>(Set.type<'a>, 'a);
}
```

As can be seen, it is really easy to write an extension. The first line declares the name of the extension. We do so by using the keyword **extension** followed by the name of the extension (in this case, *Set*). Then, we use the keyword **for** to specify the java class that implements the extension semantics. For now, we won't worry about the implementation.

Next, we declare the type of the extension. BIR type definition is really flexible and allows to extend primitive types over new extensions. Since our goal is to make this extension really flexible, we will allow the set to be polymorphic, i.e., depending on the context, this set type can hold objects of different types. If we didn't want to do this, we'd just declare the type of the set the following way:

```
typedef type;
```

saying that the extension type is just *Set* (actually, the type is *Set.type*). If we did this, we would be forces to fix the type of the set's object so that it's function definitions are unambiguous. Instead we can parameterize the Set's type by using Bogor's polymorphic type *'a*, as done in the example:

```
typedef type<'a>;
```

The declaration above acts as a template. As we will see later, we can instantiate the *Set* with any type by substituting *'a* with the appropriate type. For example, the following line:

```
Set.type<int> theSet;
```

declares a set called *theSet* that can only contain integers (if that's the semantic interpretation we specify upon implementation).

Finally, we must specify the operations that can be performed upon objects of the new declared extension. We do so with either **expdef** or **actiondef**. The former defines an operation wihtout side-effects, that is, operations that do not alter the state of the environment. The latter defines operations with side-effects, that is, operations that change some state variable. In this way we define four non-side-effects operations: *create, choose, isEmpty* and *forAll*. These operations will create a new set, pick an element non-deterministically, tell if the set is empty and check a predicate over all the elements of the set, respectively. Seemingly, we define two side-effects operations: *add* and *remove*. These operations will add/remove an element to/from the set.

The functionality of all the operations described above is just the intended meaning for each one, but it is not so until we implement them. The final semantics of all the operations will be those that we specify in the module implementation. In the next section we will see how to implement the java module for the *Set* extension.

Implementation of Bogor Extensions

Once the syntax for the extension has been defined and it has been included in the model, the next step is implementing the module that carries out the extension's functionality. The Bogor framework provides an interface called *IModule* that must be implemented by all Bogor extension classes. So we create a class for our extensions that implements this interface. We will call it *SetModule*. Here is where the **for** clause comes into play. This is the class we mentioned above that must be specified in the extension declaration. Provided we implement the extension with the following class:

```
package myPackage;

public class SetModule implements IModule {

        // class implementation ...

}
```

then we must use the following extension declaration in the model file:

```
extension Set for myPackage.SetModule
```

just as it is done in the example. The complete implementation class is given in given in the file *SetModule.java* distributed together with this document. Let us examine the interface:

```
package edu.ksu.cis.santos.bandera.bogor.module;

import edu.ksu.cis.santos.bandera.bogor.IBogorConfiguration;
```

```
import edu.ksu.cis.santos.bandera.bogor.util.Disposable;

import java.util.Properties;

public interface IModule
    extends Disposable
{
    //~ Methods ................................................................

    String getCopyrightNotice();

    void setOptions(String key, Properties configuration);

    /**
     * Connects this modules to a model checker configuration.
     */
    void connect(IBogorConfiguration bc);
}
```

The interface declares three methods that must be implemented: *getCopyrightNotice*, *setOptions* and *connect*. For the purposes of this example only the last one must be implemented, the other two can be left empty, as shown in the file *SetModule.java*. In the class module we must declare four specific fields:

```
protected TypeFactory tf;
protected IExpEvaluator ee;
protected IValueFactory vf;
protected ISchedulingStrategist ss;
```

This fields link the module to the current model checker. So, in the connect method we basically make this linking:

```
public void connect(IBogorConfiguration bc)
{
    tf = bc.getSymbolTable().getTypeFactory();
    ee = bc.getExpEvaluator();
    ss = bc.getSchedulingStrategist();
    vf = bc.getValueFactory();
}
```

Now that we are done with the interface implementation, we must proceed to implement the extension operations. Let us first start with *create*. Recall that the purpose of this function is to create a new set. Let us start with the signature of the method. So, our function here must return a set, which is a value. In Bogor, all value objects extend the interface *IValue*, so the method signature should be:

```
public IValue create(IExtArguments arg) {

  // Implementation of the method ...

}
```

So, the object returned must be of type *IValue*. Later on we will see how to implement this object, that is, the class that will represent the set internally. The formal parameter of the method must be of the type *IExtArguments*. This interface represents an object that holds an array with the arguments that are passed to the operation, so that they can be accessed internally by the method. All methods that implement extension operations either take this one formal parameter or none, depending on whether the operations take or not any arguments.

Let us now get into the details of the method implementation. Recall that our operation, *create*, takes a list of polymorphic arguments, that is, they can be of any type. So, we need to handle this in a case by case fashion. In bogor there are two kinds of types: Primitive types and non-Promitive types. The primitive types are: *IntType*, *BooleanType*, *IntRangeType*, *EnumType*, *ThreadIdType*, *DoubleType*, *FloatType*, *LongType* and *LongRangeType*. These are the names of the classes that represent internally the BIR corresponding types. The first five are descendants of the class *IntType*, and therefore, their value is represented internally by an integer. Thereby, we handle those five in the same case:

```
if(arg.getTypeVariableArgument(0) instanceof IntType ||
  arg.getTypeVariableArgument(0) instanceof BooleanType ||
  arg.getTypeVariableArgument(0) instanceof IntRangeType ||
  arg.getTypeVariableArgument(0) instanceof EnumType ||
  arg.getTypeVariableArgument(0) instanceof ThreadIdType) {

  // Some code to handle this case

}
```

Since we need to know the type of the arguments, what we do is check the type of the first one because we know all the arguments are of the same type. Similarly for the rest of the cases:

```
else if(arg.getTypeVariableArgument(0) instanceof DoubleType) {

  // ...

} else if(arg.getTypeVariableArgument(0) instanceof FloatType) {

  // ...

} else if(arg.getTypeVariableArgument(0) instanceof LongType ||
  arg.getTypeVariableArgument(0) instanceof LongRangeType) {

  // ...

} else {

  // Handle non-Primitive types

}
```

As can be seen, *LongType* and *LongRangeType* are also treated as a single case. At the end we handle the non-Primitive types which are treated all as a single case too. In each of this cases what we must do is create a set and populate it with the arguments. In this tutorial we will limit ourselves to the first case: handling integers. The rest of the cases is very similar. So let's take a look at the implementation of the first case:

```
if(arg.getTypeVariableArgument(0) instanceof IntType ||
        arg.getTypeVariableArgument(0) instanceof BooleanType ||
        arg.getTypeVariableArgument(0) instanceof IntRangeType ||
        arg.getTypeVariableArgument(0) instanceof EnumType ||
        arg.getTypeVariableArgument(0) instanceof ThreadIdType) {

        // builds an empty set
        IntSetValue result = new IntSetValue(arg.getExpType().getTypeId(), vf,
                                        arg.getTypeVariableArgument(0));

        int size = arg.getArgumentCount();

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

        return result;

}
```

In the first line we instantiated the set object. This object is of type *IntSetValue* which is an special purpose class that implements the interface *IValue* (more specifically, it implements *INontPrimitiveExtValue*). We will see how to write this class later in the tutorial. The full source code for this class is included with this document with the name *IntSetValue.java*.

Once we have created the set object we have to populate it with arguments. To do this, we get the argument count and use a **for** loop to add the elements to the set using the *add* method from the *IntSetValue* class, which we will see later how to implement.

So, that was it for *create*. Let us now move on to the next operation: *choose*. This is an interesting operation because it picks and returns, non-deterministically, an object from the set. To refresh our minds, we will look again at the operation's signature:

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

Hence, the method that implements this operation must return a value, that is, an *IValue* object. Because it takes an argument, we know the method has a formal parameter of type *IExtArguments* with only one element in its array. So, the method signature is:

```
public IValue choose(IExtArguments arg)
```

Again, we must handle all the cases that correspond to the different types. As we said before, we will focus on the integer case as the rest of the cases are handled similarly. Here is the code for the integer case:

```
if(arg.getArgument(0) instanceof IntSetValue) {

        // get the elements of the set
        IntSetValue set = (IntSetValue) arg.getArgument(0);
        int[] elements = set.elements();

        // create array of IValues
        IValue[] elementsValueArray = new IValue[elements.length];
        for(int i=0; i < elements.length; ++i) {
                elementsValueArray[i] = vf.newIntValue(elements[i]);
        }

        // order the values so that the choices are consistent
        orderValues(elementsValueArray);

        // ask the scheduler which one should be picked now
        int index = ss.advise(
                        arg.getExtDesc(),
                        arg.getNode(),
                        elementsValueArray,
                        arg.getSchedulingStrategyInfo());

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

}
```

The first lines is just getting the set object out of the arguments object, extracting the elements and then wrapping them with *IValue* objects. The wrapping is needed for the next part which is the interesting one. Because we need a non-deterministic choice, we use the model checker's scheduler to take the pick. Since we already connected the module to the model checker, we can ask for the advice of the *ISchedulingStrategist* object. Once we have the index, the selected value is returned. The complete implementation of this method is provided in the file *SetModule.java* along with this document.

The next operation is *isEmpty*. The operation's signature is:

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

So, the operation returns a value and takes an argument. Then, the method's signature is straightforward as we have seen in the other cases:

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

Again, we focus our attention to the integer case:

```
if(arg.getArgument(0) instanceof IntSetValue) {

        IntSetValue set = (IntSetValue) arg.getArgument(0);
        return getBooleanValue(set.isEmpty());

}
```

This is a really easy one. We just get the set and then invoke the *isEmpty* method of the *IntSetValue* class, which we will see later how is implemented. Finally the result must be wrapped with an *IValue* object because this is the only type that module methods that implement extension operations can return.

Continuing with the operations we arrive to *forAll*. Here is the operation's signature:

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

Following our methodology, the method returns a value and takes arguments, therefore the java method's signature is:

```
public IValue forAll(IExtArguments arg)
```

The implementation of this operation is very easy: we just apply the function to every element of the set and verify that it returns **true** in each case. Here is the code for the integer case:

```
if(arg.getArgument(1) instanceof IntSetValue) {

        // get the fun name of the function
        String predFunId = ((IStringValue) arg.getArgument(0)).getString();

        // get the set elements
        IntSetValue set = (IntSetValue) arg.getArgument(1);
        int[] 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 = vf.newIntValue(elements[i]);
                IIntValue val = (IIntValue) ee.evaluateApply(
                                predFunId,
                                new IValue[] { element });
```

```
                if (val.getInteger() != 1)
                {
                        result = false;

                        break;
                }
        }

        return getBooleanValue(result);

}
```

In the first line, we get the name of the function we must apply to the set elements. Having done this, we start applying the function to the elements of the set and check the result: if it's **true** we continue with the next element, otherwise we break the loop and return false as the result. If the function is true *for all* the elements, we return **true**.

Now that we're done with all the non-side-effect operations, we proceed to implement the side-effect operations. We defined only two of this kind: *add* and *remove*. We will only look at the implementation of *add* since *remove* is defined analogously. The complete implementation of both operations is provided in the file *SetModule.java* along with this document. Let us review the operation's signature:

```
actiondef add<'a>(Set.type<'a>, 'a);
```

Here, the things change a little bit. Because this is a side-effect operation it changes the state as it executes, therefore we need to provide a way to restore the state variables to the values they were at this point when the model checker needs to backtrack. Hence, all the methods implementing a side-effect operation must return an *IBacktrackingInfo* object, which holds the necessary backtracking information. This object is held in memory and is used by the model checker at backtracking time. For the rest, it is the same as in the non-side-effect operations:

```
public IBacktrackingInfo[] add(IExtArguments arg)
```

The implementation of this method, as might be expected, is very easy. Here we show the code fragment for the integer case. The whole method implementation is shown in the file *SetModule.java* provided with this document.

```
if(arg.getArgument(0) instanceof IntSetValue) {

        // get the set
        IntSetValue set = (IntSetValue) arg.getArgument(0);

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

        // get the value of the element
```

```
        int elementValue = ((IIntValue)element).getInteger();

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

                // create the backtracking infos
                return new IBacktrackingInfo[] {
                        createAddBacktrackingInfo(set, element, arg)
                };
        } else {
                // do nothing, so empty backtracking info
                return new IBacktrackingInfo[0];
        }

}
```

In the first lines we get the two arguments out of the *IExtArguments* object and extract the integer value out of the second argument. This is because, as we will see later, *IntSetValue* holds primitive integers to make it more efficient. Following, we check that the element is not in the set, add the element to the set and generate the backtracking information. Next, we will see the details of *createAddBacktrackingInfo*.

The last method we will discuss in this section is *createAddBacktrackingInfo*. This method returns an *IBacktrackingInfo* object by means of an anonymous class definition. In this way we implement all the methods declared by the interface *IBacktrackingInfo*, in particular we will take a look at the implementation of the method *backtrack* for the integer case. The rest of the details for this and the other methods are provided in the file *SetModule.java* along with this document.

```
// create add backtracking info
    protected IBacktrackingInfo createAddBacktrackingInfo(
        final SetValue set,
        final IValue element,
        IExtArguments arg)
    {
        return new IBacktrackingInfo()
            {

                // ...

                public void backtrack(IState state)
                {

                        // ...

                        if(set instanceof IntSetValue) {
                                ((IntSetValue)set).remove(((IIntValue)element).getInteger());
                        }

                        // ...

                }

                // ...

            };
    }
```

Very simple, in the method *backtrack* we simply provide the necessary steps to restore the original state, in this case, remove from the set the element that was added to it.

In the next section we will review the implementation of *IntSetValue*, the class that provides the actual container for the set abstraction in the case when the elements are integers.

Implementation of Bogor Value Classes

When implementing Bogor extensions usually we have to implementent special value objects for the extended types. For instance, in our example, we need to implement a class that represents the values that the objects of the new type *Set* can take. Because this type is *generic* (by means of the parameterization that we introduced with the help of the polymorphic type *'a*) we need several classes representing sets of integers, longs, doubles, etc. In this section we will focus in the class that represents a set of integers. The whole implementation of this class is provided in a file called *IntSetValue.java* along with this document.

All value classes in Bogor must implement the interface *IValue*. Usually, this interface is implemented through one of its descendants. There are two value interfaces for extended values: *IPrimitiveExtValue* and *INonPrimitiveExtValue*. A data structure (in this case a set), is not a primitive value; therefore, we must implement the interface *INonPrimitiveExtValue*. This interface declares three methods: *externalize*, *linearize* and *visit*. The first method must provide a way to externalize this object (usually in the form of XML). The implementation of this method is not interesting and code can be found in the file *IntSetValue.java*, provided with this document. We will place our attention instead in the last two methods: *linearize* and *visit*.

The first method, *linearize*, is a really important one. This method must provide a way to transform the current object into an efficient linear representation. This is for storage purposes. Since the object will be kept in memory during checking, we must provide a memory efficient representation to save memory resources. Let us first look at the signature of the method:

```
public void linearize(
  BitBuffer bb,
  int bitsPerNonPrimitiveValue,
  ObjectIntTable nonPrimitiveValueIdMapd,
  int bitsPerThreadId,
  IntIntTable threadOrderMap)
```

The first argument, the *BitBuffer* object, is where we have to linearize this object in. Since this set can hold objects of different types which are represented as integers,

then we do a case-based processing. For the *BooleanType* case we have:

```
if(type instanceof BooleanType) {

        for(int i=0; i < elements.length; ++i) {

                bb.append(elements[i] != 0);

        }

        return;

}
```

The method *append(*boolean) from *BitBuffer* just sets the next bit in the buffer if its argument is **true**. Because a boolean can be represented efficiently with only one bit, this is all we need to linearize the set: represent it as a bit string where each bit represents an element from the set. Now, let us see the *IntRangeType* case:

```
if(type instanceof IntRangeType) {

        IntRangeType rangeType = (IntRangeType) type;
        int rangeMin = rangeType.getLowLimit();
        int rangeMax = rangeType.getHighLimit();
        int offset = 0;

        if(rangeMin < 0) {

                offset = -rangeMin;
                rangeMin = 0;
                rangeMax += offset;

        }

        int bitLength = Util.widthInBits(rangeMax);

        for(int i=0; i < elements.length; ++i) {

                int element = elements[i];
                element += offset;

                // Better be between the limits
                assert element <= rangeMax && element >= rangeMin;

                bb.append(element, bitLength);

        }

        return;

}
```

In this case we need more than one bit and to be efficient we want to use the least possible bits. So, at the beginning we get the low and high limit to calculate the minimum amount of bits necessary to represent any number in the range. For this latter purpose we use the Bogor utility function *Util.widthInBits* which gives the amount of bits needed to represent a given number. Then we simply do the same as before: we set the corresponding bits. In this case we need to set more bits per element, so we use the

function *append*(**long**, **int**) which lets us specify the number of bits to set. An interesting aspect here is that we shift the range to the right if it spans negative numbers. That is, we make it range over non-negative numbers only. This is because it is more efficient to linearize positive numbers because they require less bits. The rest of the cases is handled similarly. To take a full look to the code, check the file *IntSetValue.java* provided with this document.

Now we move to the *visit* function. This function is used for garbage collection and heap symmetry purposes. A full description of its functionality is out of the scope of this tutorial, but let us just say that it gives a way to keep track of the values that have been visited in the current verification path. The signature of the method is:

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

For the purposes of this example we will only use the *LinkedList* work list object. Basically, what we have to do is add the set elements to this list:

```
int[] elements = set.toArray();
Arrays.sort(elements);

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

return;
```

As seen in the code, if the visit mode is depth first search, we add the elements to the front of the list with the method *addFirst* from the *LinkedList* object.

Now we start with the implementation of the set functions themselves. To implement the set structure we use the GNU Trove container classes which are efficient implementations of most types of containers (including sets). So we have the class field:

```
protected IntSet set;
```

IntSet is just a wrapper around the class *gnu.trove.TIntHashSet*. Implementation of the functions is now straightforward because most of the set operations are provided in the GNU Trove set implementation. For instance, let us take a look at *isEmpty*:

```
public boolean isEmpty() {
  return set.isEmpty();
}
```

And that is it. The same applies for the rest of the operations so we don't include them here. For a complete listing of the code, check the file *IntSetValue.java* provided with this document.

The completion of this tutorial should give enough understanding of Bogor's extension functionality so as to be able to implement new extensions from scratch. However, we warn that there's more to Bogor's extension than what is showed in this tutorial and the expertise can only be acquired through practice. For any comment regarding this tutorial or Bogor's extensions, please contact the author or any member of the SAnToS laboratory.