User's Guide for BDDs in Jython

Mark Greenstreet, January 29, 2013

1 Overview

I'll add something here about downloading the package, running it on UBC CS department machines, etc.

2 Class `SymBool`

Class `SymBool` provides symbolic boolean operations using BDDs. It is build on the JavaBDD package and provides symbolic operations in Jython.

2.1 Using the Class

To import the class, give the Jython statement:

```
import SymBool
```

This assumes that everything is set up properly on your `CLASSPATH` which hopefully I'll describe in Section 1. To create a new symbolic boolean variable, use the Jython statement:

a = SymBool("a")

The parameter to SymBool is the name that will be used when printing symbolic expressions. To make sure that I can "make-up" names for array elements, anonymous variables, etc., I've restricted the names for symbolic variables to consist only of upper and lower case letters, digits, and underscores, and not start with a digit. Of course, there's no requirement that the Jython variable have the same name as the symbolic variable that you create. You could say

cat = SymBool("dog_42")

Of course, if you get yourself confused by doing this, that's your problem, not mine.

For the rest of this section, I'll assume that you've created three symbolic boolean variables:

```
>>> import SymBool
>>> a = SymBool("a")
>>> b = SymBool("b")
>>> c = SymBool("c")
```

2.2 Overloading of Jython Operators

The Jython logical operators are overloaded. In particular:

Bitwise-logical operators: `&` (and), `|` (or), ∼ (not), and ∧ `exclusive-or`.

These do what you would expect. For example,

```
>>> a & b
a&b
>>> a ∧ b
∼a&b | a&∼b
>>> a | ∼a
True
```

Furthermore, Jython booleans and integers can be used as operands as well. Booleans are handled in the obvious way. An integer is treated as `True` if its least significant bit is a 1 and as `False` if its least significant bit is a 0. I'll give my rationale for how integers are treated in Section 3 about symbolic integers. Some more examples:

```
>>> a & True
a
>>> a ^ True
~a
>>> a | True
True
```

Comparisons: ==, <, <=, !=, >=, and >.

These are defined using the inequality that `False < True`. Note that these are evaluated symbolically: they produce the symbolic expression that describes *when* the comparison holds. For example:

```
>>> a|b == a^b
~a | ~b
```

Because `a|b` is equal to `a^b` when either `a` or `b` (or both) is false, and otherwise they are different. If you want to determine whether or not two expressions are logically the same expression, then use the `equals` method as described in Section 2.3. For now, here's a quick example:

```
>>> (a|b).equals(a^b)
False
>>> ((a&b) | (b&c) | (a&c)).equals((a|b) & (b|c) & (a|c))
True
```

Arithmetic: +, −, <<, and >>.

A `SymBool` acts like a one-bit, unsigned, binary integer – in other words, it has a value of 0 or 1. These operations produce the least significant bit of the result:

Binary + and − are equivalent to exclusive or: `a + b` = `a-b` = `a ^ b`.

unary − is the identity operation: `-a` = `a`.

`a << b` and `a >> b` are both equivalent to `a & ~b`.

These probably aren't very useful, but there versions for multi-bit symbolic integers, `SymUInt`s, are very useful. I've included the `SymBool` versions for completeness so there won't be unnecessary `UnsupportedOperationException`s.

2.3 Additional Symbolic Operations

2.3.1 Methods for additional binary operations

The JavaBDD provides a few other operations that aren't part of Jython/Python. These are available as methods for `SymBool` objects:

`biimp` is "bi-implication" (if and only iff). For `SymBool` objects, `a.biimp(b)` is equivalent to `a == b`.

`equals` tests to see if two symbolic expressions are logically equivalent. It returns either `True` or `False`.

`imp` is implication. Note that `a.imp(b)`, `~a | b`, and `a <= b` are all logically equivalent, but `a.impb` is probably easier to read if what you intended was implication.

`ite` is an if-then-else. The expression `a.ite(t, e)` means "if `a` then `t` else `e`".

2.3.2 The methods that overload Jython operations

`SymBool` objects provide the following methods which overload Jython operators as described in Section 2.2 in the (hopefully) obvious way:

`and`, `eq`, `ge`, `gt`, `le`, `lt`, `ne`, `neg`, `not`, `shl`, `shr`, and `xor`.

2.3.3 Quantifiers, composition, and relations

`compose(oldVars, newExprs)` function composition.
`oldVars` is a symbolic boolean variable, a symbolic a list of `SymBool` objects. Likewise, `newExprs` is a symbolic boolean expression, a `SymUInt` For example,

```
>>> d = SymBool("d")
>>> e = SymBool("e")
>>> m = a&b | b&c | a&c
>>> m.compose([a,b], [d&e, ~d&~e])
c&~d&~e | c&d&e
```

Note that all of the substitutions are done "simultanously". For example,

```
>>> f = a & (b | c)
>>> f
a&b | a&c
>>> f.compose([a,b,c], [b,c,a]) b&c | b&a
```

but

```
>>> f.compose(a,b).compose(b,c).compose(c,a)
a
```

Try the `compose` operations one at a time if this isn't clear.

`exist(vars)` existential quantification.
`x.exists(v)` evaluates to $\exists$v. x. v can be a `SymBool` variable, an array of `SymBool` variables, or a `SymUInt` variable. To make this clear, consider

```
>>> m.exist(a)
b | c
>>> m.exist([a,b])
True
>>> f = a&d
>>> m.exist(f)
Traceback (most recent call last):
...
...vars[0] is not a variable
```

If x is a `SymUInt`, then each element of x is existentially quantified over the variable of v.

`forAll(vars)` universal quantification.
`x.forAll(v)` evaluates to $\forall$v. x. Same comments about v being a `SymBool` variable, an array of `SymBool` variables or a `SymUInt` variable apply here as described above for existential quantification. Likewise, if x is a `SymUInt`, then each element of x is universally quantified over the variables of v. Continuing our previous example:

```
>>> m.forall(a)
b&c
```

`relprod(r2, vars)` relational product.
`r1.relProd(r2, v)` is logically equivalent to `(r1 & r2).exist(v)` but apparently executes much faster. Now, a few words about relations. Consider domains X, Y, and Z each represented by four-bit `SymUInts`:

```
>>> import SymUInt
>>> x = SymUInt("x", 4)
>>> y = SymUInt("y", 4)
>>> z = SymUInt("z", 4)
```

To make a simple example, I using the `SymUInt` type. If you want, skip ahead and read Section 3 and then come back to these examples. To express a relation over these domains (i.e. $R_1 \subseteq X \times Y$) let `r1` be the predicate that is true iff the values of `x` and `y` are such that $(\mathtt{x}, \mathtt{y}) \in R_1$. Writing it this way makes it sound much more complicated than it really is. For example, I could define

$$R_1 \quad = \quad \{(\mathtt{x}, \mathtt{y}) \mid ((\mathtt{x} \text{ is even}) \,\&\, (\mathtt{x} > \mathtt{y}))|((\mathtt{x} \text{ is odd}) \,\&\, (\mathtt{x} < \mathtt{y}))\}$$

In Jython, I would write

```
>>> r1 = (~x[0] & (x > y)) | (x[0] & (x < y))
```

Now, let $R_2 \subseteq Y \times Z$ be the relation with

$$R_2 \quad = \quad \{(\mathtt{y}, \mathtt{z}) \mid (\mathtt{z} = \mathtt{y} + 4) \,\&\, (\mathtt{z} \geq 4)\}$$

(the $z \geq 4$ clause is to avoid wrap-around issues because $+$ is modulo 16 because `x`, `y`, and `z` are 4-bit unsigned integers). The relational product of R_1 and R_2 is written mathematically as $R_1 \circ R_2$ where

$$R_1 \circ R_2 \quad = \quad \{(x, z) \mid \exists y.\, ((x, y) \in R_1) \,\&\, ((y, z) \in R_2)\}$$

In Jython, we write this as

```
>>> r2 = (z == y+4) & (z >= 4)
>>> r12 = r1.relProd(r2, y)
```

Now, we can do various membership checks in these relations:

```
>>> r1.compose(x, 4).compose(y,2)
True
```

This says that $(4, 2) \in R_1$ which is true because 4 is even and $4 > 2$. We could be tempted to try

```
>>> r1.compose([x, y], [4, 2])
Traceback (most recent call last):
...
...vars[0] is not a variable
```

This is because I didn't allow the variable to be a list of `SymUInt` variables. Basically, this is because I couldn't think of a tidy way to match up the widths of the variables and the widths of expressions. Of course, this could be a nuisance if you need *simultaneous* composition over several `SymUInt` variables. I'll describe how to do this in Section 3.2.4.

Continuing with our example,

```
>>> r1.compose(x, 0)
False                  # x is even an no y is < 0
>>> r12.compose(x, 5).compose(z, 6)
True                   # consider y = 2
```

2.3.4 Operations that are specific to BDDs

`high()` return the "high" (i.e. `topVar() == True`) descendant of this BDD.

`low()` return the "low" (i.e. `topVar() == False`) descendant of this BDD.

`topVar()` return a `SymBool` object for the variable associated with the top-node of this BDD. If the BDD represents the constant `True` or `False`, then `topVar` returns `None`.

2.4 Other Operations

`logSatCount()` returns the logarithm (base 10) of the number of satisfying assignments for this BDD. If this BDD is unsatisfiable, `-inf` is returned.
See also: `satCount()`.

`nodeCount()` return the number of BDD nodes for this BDD. Note that if `x` and `y`, then the total number of BDD nodes needed to represent both `x` and `y` can be less than `x.nodeCount() + y.nodeCount()` because they may share nodes. See `SymUInt.nodeCount()` for a way to get totals that take into account the effects of node sharing.

`satCount()` return the number of satisfying assignments for this BDD. Note that this number includes all variables that you have declared, even if they aren't involved in your BDD. For example (from a fresh Jython session):

```
>>> import SymBool
>>> a = SymBool("a")
>>> b = SymBool("b")
>>> (a & b).satCount()
1.0                    # a&b
>>> c = SymBool("b")
>>> (a & b).satCount()
2.0                    # a&b&~c and a&b&c
```

Also, note that `satCount()` returns a double. This is because a BDD may have a large number of satisfying assignments, more than can be represented by a Java `int` or `long`. Of course, Jython integers can be arbitrarily large, but to exploit that, I'd have to rewrite some portions of the JavaBDD code, and I don't want to do that.
See also: `logSatCount()`.

`satOne()` return one satisfying assignment. The return value is a BDD that is a conjunction of variables and complemented variables. For example:

```
>>> m = a&b | b&c | a&c
>>> m.satOne()
~a&b&c
>>> (a & b).satOne()
a&b
```

The return values reflect the structure of the BDD. `m.satOne()` included ∼`a` in the conjunction even though `a&b&c` is a satisfying assignment, and thus `b&c` is a conjunction that implies `m`. The ∼`a` term was included because the top node of `m` is `a`

```
>>> m.topVar()
a
```

The code for `satOne()` chose the low branch and continued. In contrast, `(a & b).satOne()` doesn't include a conjunct for `c` or ∼`c` along the path that `satOne()` traversed from the top node to `True` (for that example, there is only one such path).

`toString(), toString(maxCubes)` write a string representation of a `SymBool`. This is what is printed by Jython for an expression that evaluates to a `SymBool` value.

- `toString()` prints the boolean expression as a "sum-of-products" (a.k.a. disjunctive normal form). This is *not* the internal BDD representation. In particular, the printed form can be exponentially larger than the BDD.
- To prevent an output that goes on for centuries, or longer, `toString()` prints at most 100 "cubes" where a cube is a conjunction of variables or their negations.
- If you want to change the limit, use the form `toString(maxCubes)` which will print at most `maxCubes` cubes.
- I try to simplify the expression. For example, if the traversal of the BDD produces the cubes `a&b&c` and `a&`∼`b&c`, then they will be merged to the simpler cube `a&c`. The simplified form is not the smallest possible sum-of-products form in all cases. Furthermore, if the `maxCubes` limit is reached, then the simplification may be incomplete.
- I've thought of adding options to provide other simplifications, for example, allowing product-of-sums form for subtrees, using exclusive-OR, etc., but that's a low-priority (for now).
- Likewise, `SymUInt` values are printed by giving values for the bits of the integer, and these are bits are interleaved over the `SymUInt` variables of the BDD. I've thought of gathering the bits for a single `SymUInt` together and printing the integer value (or perhaps a Jython/Python style "slice"). I may do that eventually, but not soon.

2.5 Missing Features

- I should exclude `True` and `False` as variable names, but I haven't done that (yet). Nothing breaks in the code if you use those names, but the printed versions of symbolic expressions will be confusing.
- I should provide a way for users to specify the variable ordering. Most of the hooks for this are actually lurking in the code, but there are still a few technical and interface issues that I haven't worked out.
- I don't provide `div` and `mod` operations. They're pretty useless for one-bit quantities, and I haven't figured out how to sensibly handle `div` by a symbolic expression that can evaluate to zero.
- The `ite` method for `SymBool` objects should be extended to support `SymUInt`, `boolean`, and `int` operands.
- I didn't overload the Jython/Python `and` and `or` operators because the language regards them as control-flow operations (due to short-circuit evaulation) and Jython and Python don't allow overloading of control-flow operations. ☹
- I've considered making some improvements to `toString()` to make it print cubes in a more "sensible" order (in fact, I'm not sure why it doesn't), to print the varibles in a cube in a more sensible order (keeping the elements of each `SymUInt` together), printing assignments to `SymUInt` objects as integers or integer ranges, more aggressive simplification, etc.

3 Class `SymUInt`

This class implements symbolic, unsigned, binary integers.

3.1 Constructors

`SymUInt(name, width)` where `name` is a string and `width` is an integer creates a symbolic, unsigned integer with `width` bits. Each bit is a `SymBool` object, where bit `i` has name `name[i]` for $0 \leq$ `i` $<$ `width`.

`SymUInt(n, width)` where `n` is an integer and `width` is an integer creates a `SymUInt` object where each bit is the constant `True` or `False` corresponding to the bit of `n`.

`SymUInt(v, width)` where `v` is an array of `SymBool` or `boolean` values (or a mix) creates a `SymUInt` object with `len(v)` bits where each bit has the value of the corresponding element of `v`.

3.2 Overloading of Jython Operators

These operators are overloaded so a `SymUInt` can be combined with a `SymBool`, `boolean`, or `int`. If two `SymUInt` objects of different widths are in an expression, then the shorter one is extended with bits set to the constant `False` to make them compatible. Likewise, a `SymBool` object or a `boolean` is treated as a one-bit `SymUInt` and extended accordingly. An `int` is converted to a `SymUInt` with the same width as the `SymUInt` operand.

3.2.1 Bitwise logical operators:

`&` (and), `|` (or), $\sim$ (not), and $\wedge$ `exclusive-or`.

3.2.2 Arithmetic operators:

+ and, −.
I haven't implemented shifts or multiplication (yet). I might eventually implement division and remainder, but I need to think of a way to handle division by zero.

3.2.3 Comparisons:

==, <, <=, !=, >=, and >.
These work as expected. The comparison applies to the word (i.e. the unsigned binary interpretation of the `SymUInt` objects), and the result is a `SymBool` object.

3.2.4 Arrays:

- For the most part, a `SymUInt` can be treated as a Jython array or list. If `x` is a `SymUInt` then `x[j]` is the $\mathtt{j}^{th}$ element of `x`. Assignments and slicing are supported. Constructions such as `for v in x:` or `[ expression for v in x]` work as expected.
- For the most part, a `SymUInt` can be treated as a `SymUInt` objects implement + as binary addition instead of list concatenation. Someday, `SymUInt` objects will probably implement $\star$ as binary multiplication, but they do not and will not (in my current plans) implement $\star$ as list repetition. If you want these list operations, you can convert a `SymUInt` to an array using the `toArray()` method, and usual Jython semantics will apply to the array. You can convert the result back to a `SymUInt` by using the `SymUInt` constructor.

  Returning to the example for relational product from Section 2.3.3, we can set `x` to 5 and `z` to 6 with a single application of compose as shown below:

```
>>> r12.compose(x.toArray() + z.toArray(), SymUInt(5, len(x)) + SymUInt(6, len(z)))
True
```

- Assignments are a bit messy because we have no way to "forget" variables. So, if you give the commands:

  ```
  >>> import SymBool
  >>> import SymUInt
  >>> a = SymBool("a") >>> x = SymUInt("x", 4)
  >>> print [xx for xx in x]
  [x[0], x[1], x[2], x[3]]
  >>> x[2] = a
  >>> print [xx for xx in x]
  [x[0], x[1], a, x[3]]
  ```

  Then, it might seem hard to find a reference to `x[2]` ever again. Don't worry, the `restore` method will give you the original value again (without modifying the current value).

- Concatenation: if `x`, and `y` are `SymUInt` objects, then

3.3 Other operations

Of course, there are methods corresponding to the overloaded Jython operations. Note that the methods `and`, `or`, `xor` and the comparison operations can be applied to an object without any arguments to "reduce" the `SymUInt` object to a `SymBool`. The `compose`, `exist`, and `forAll` methods apply the operation to each bit of the `SymUInt`.

4 Other Classes

- `AlreadyDeclaredException` – the Java Exception class for declaring a `SymBool` or `SymUInt` when there is already a symbolic variable of that name.
- `JavaSlice` – a class that provides a Java representation of Jython slices.
- `SymBDD` – the super-class of `SymBool` and `SymUInt`. If other symbolic types are created later; they'll probably be subclasses (directly, or indirectly) of `SymBDD`. For example, someday, I may create a class for signed, twos-complement integer, `SymInt` which would be a subclass of `SymUInt`.

  The `SymUInt` class provides the methods for overloading Jython operators. Because these operators can be applied to many different types of operands, each these methods comes in *many* different versions.
- `SymFactory` – the BDD package works with factories. This is the interface to those factories. The map that matches names to `SymBDD` objects is here. If you clobber a symbolic variable, you can get it back with the method `SymFactory.find(String name)`.

  Variable ordering is handled in `SymFactory`. I'd like to provide a way for the user to specify variable orders. If I do, it's likely to involve adding some methods to this class.

  The JavaBDD package allows there to be multiple `SymFactory` objects in use at the same time. I haven't seen a need for this, but it would be easy to make this functionality accessible if there's a need for it.
- `SymOp` – a Java enumeration. This provides a mapping of Jython operations to method names in class `SymBDD`.