Kamen Yotov 02/26/2003

C# Omega Interface

(overview)

1 INTRODUCTION

This document describes a C# object-oriented interface to the Omega library. The interface is not complete in the sense that it does not allow you to use the full potential of the library, but it is complete enough to be able to do the assignment. Please refer to the file *Original_Omega.pdf* included to study the original (C/C++) interface to the library. Please read and study *Original_Omega.pdf* before you continue with this document.

2 DESIGN

2.1 EXAMPLE

After you have solved problem 1 using the Omega Calculator and you have read the original Omega Library interface you will probably find out that it is much easier to use the calculator than to program omega directly. Indeed, a relatively simple relation looking like

{[i1,j1,k1] -> [i2,j2,k2]: i2 = i1 && j1 < j2 || i2 = i1 && j2 = j1 && k1 <= k2 || i1 < i2 }

in the calculator need to be constructed with the library interface in the following way:

```
void main ()
{
        char *_1[3] = {"i1", "j1", "k1"};
        char *_2[3] = {"i2", "j2", "k2"};
        Variable_ID id1[3];
        Variable_ID id2[3];
        Variable_ID &i1=id1[0], &j1=id1[1], &k1=id1[2];
        Variable_ID &i2=id2[0], &j2=id2[1], &k2=id2[2];

        Relation s(3,3);

        for (int i=0; i<3; i++)
        {
                s.name_input_var(i+1,_1[i]);
                s.name_output_var(i+1,_2[i]);
                id1[i]=s.input_var(i+1);
                id2[i]=s.output_var(i+1);
        }

        F_And *a1 = s.add_and();

        F_Or *o1 = a1->add_or();
        F_And *a2 = o1->add_and();
        GEQ_Handle h1 = a2->add_GEQ();
        h1.update_coef(i1,1);
        h1.update_coef(i2,-1);
        h1.negate();

        F_And *a3 = o1->add_and();
        EQ_Handle h2 = a3->add_EQ();
        h2.update_coef(i1,1);
        h2.update_coef(i2,-1);

        F_Or *o2 = a3->add_or();

        F_And *a4 = o2->add_and();
        GEQ_Handle h3 = a4->add_GEQ();
        h3.update_coef(j1,1);
        h3.update_coef(j2,-1);
        h3.negate();

        F_And *a5 = o2->add_and();
        EQ_Handle h4 = a5->add_EQ();
        h4.update_coef(j1,1);
        h4.update_coef(j2,-1);
        GEQ_Handle h5 = a5->add_GEQ();
        h5.update_coef(k1,-1);
        h5.update_coef(k2,1);

        s.print();
}
```

There is no obvious need for using such an elaborate interface for creating relations. Furthermore the Bernoulli Compiler Construction Kit is developed in C# and we cannot just use the available C interface for incompatibility reasons. Therefore I have developed a C# interface to the Omega Library which is almost as intuitive as the Omega Calculator interface. The "almost" here is because C# has limited operator overloading capabilities. Here is how this relation needs to be constructed using the C# interface:

```
        Variable i1 = new Variable("i1");
        Variable j1 = new Variable("j1");
        Variable k1 = new Variable("k1");
        Variable i2 = new Variable("i2");
        Variable j2 = new Variable("j2");
        Variable k2 = new Variable("k2");

        Relation r = new Relation(
                VariableTuple.Create(i1, j1, k1),
                VariableTuple.Create(i2, j2, k2),
                (i1 < i2) | (i1 == i2) & ((j1 < j2) | (j1 == j2) & (k1 <= k2)));

        r.Print();
        r.Finalize();
        r.Simplify();
        r.Print();
```

And the result from executing this snippet is:

{[i1,j1,k1] -> [i2,j2,k2] (not (i2 <= i1) or i2 = i1 and (not (j2 <= j1) or j2 = j1 and k1 <= k2)) }

{[i1,j1,k1] -> [i2,j2,k2] i1 = i2 && j1 < j2 OR i1 = i2 && j1 = j2 && k1 <= k2 OR i1 < i2 }

2.2 SUPPORTED OPERATIONS

2.2.1 BUILDING RELATIONS

You noticed how straight-forward it is to create a new relation using the C# interface (above). Here are the operations you can use in the body of the relation:

- **&** – creates an F_And node in the relation body as described in *Original_Omega.pdf*;
- | – creates an F_Or node
- ! – creates an F_Not node
- == – creates a EQ_Handle constraint
- >= – creates a GEQ_Handle constraint
- <, >, <=, != – uses the above two constraints and "!" to emulate these constraints
- +, -, * – builds affine expressions that translate to updating coefficients and constants in EQ and and GEQ constraints (these are the calls to <handle>.update_coef(<variable_id>, <value>))

There are several notes that need to be made. First, C# does not allow overloading of the && and || operators. Therefore & and | were used. Because && has lower precedence than ==, which in turn has lower precedence than &, we need to enclose == and other relational constraints in parenthesis, as in the C# example above. Study the C# operator precedence (which is the same as C and Java) and figure out for yourself what you need to do to ensure the proper order of operations. Second, there are limitations to the type of expressions you can build. These limitations (e.g. only affine expressions) are the same as in the Omega Calculator. If you try to build an expression that is not acceptable, an Exception will be thrown upon the construction of the relation. Finally, there are a number of constructs that are not yet supported, like Exists and Every and some more… You don't need them to do the assignment, but if for any reason you cannot live without them, drop me a line of what you want and I will try to add it as soon as possible.

2.2.2 OPERATING WITH RELATIONS

The C# interface supports the following operations on relations:

- * – intersection, e.g. r1 = r2 * r3;
- + – union, e.g. r1 = r2 + r3;
- [] – composition, e.g. r1 = r2[r3];

Read about these relation operations in *Original_Omega.pdf.*

2.2.3 QUERYING RELATIONS

This feature of the library lets you examine some features of relations, constructed using the above methods. Make sure you read the corresponding section in the original Omega interface document. One important feature is that it is able to convert a relation to DNF, examine the different conjuncts separately and within each conjunct query the lower and upper bounds for a free variable. Here are the operations that you need:

- r.Conjuncts returns an array of integers, each one being a handle to a conjunct;
- r.GetVariableBounds(c, v, out lb, out ub) accepts a conjunct handle (c), a variable handle (v) and returns the lower and upper bounds of the variable in lb and ub…

To get a variable handle from a Variable object v you need to cast it to int, i.e. (int)v. Any questions are welcome, but in the newsgroup. This way it will be made sure that I answer them only once!