The SYMPHONY Callable Library for Mixed-Integer Linear Programming

A Tutorial

Ted Ralphs and Menal Guzelsoy
Industrial and Systems Engineering
Lehigh University

INFORMS Computing Society Conference, Annapolis, MD, Wednesday, January 5, 2005
Outline of Talk

• Introduction to SYMPHONY
• Using SYMPHONY as a black box solver
  – Downloading and compiling
  – Using from the command line
  – Using the interactive shell
• Using the SYMPHONY callable library
  – C API
  – C++/OSI API
• Developing custom solvers using the SYMPHONY framework
  – Callback API
  – Example
• Advanced Features
  – Sensitivity analysis
  – Warm starting
  – Bicriteria solve
  – Parallel Execution
Brief Overview of SYMPHONY

- SYMPHONY is an open-source software package for solving and analyzing mixed-integer linear programs (MILPs).

- SYMPHONY can be used in three distinct modes.
  - Black box solver: Solve generic MILPs (command line or shell).
  - Callable library: Call SYMPHONY from a C/C++ code.
  - Framework: Develop a customized black box solver or callable library.

- Fully integrated with the Computational Infrastructure for Operations Research (COIN-OR) libraries.

- Advanced features
  - Sensitivity analysis
  - Warm starting
  - Bicriteria solve
  - Parallel Execution

- This talk based on version 5.1 (unreleased, but in CVS)
**Algorithmic Features**

- Core solution methodology is a state of the art implementation of the *branch, cut, and price* algorithm.
- Default search strategy is a hybrid depth-first/best-first strategy.
- Built-in strong branching mechanism.
- Uses the primal heuristic of CBC.
- Cuts can be generated with COIN-ORs Cut Generation Library.
  - Clique
  - Flow Cover
  - Gomory
  - Knapsack Cover
  - Lift and Project
  - Mixed Integer Rounding
  - Odd Hole
  - Probing
  - Simple Rounding
  - Two-slope MIR
What’s Available

• Packaged releases from www.branchandcut.org
• Current source at CVS on www.coin-or.org.
• An extensive user’s manual on-line and in PDF.
• A tutorial illustrating the development of a custom solver step by step.
• Configuration and compilation files for supported architectures
  – Single-processor Linux, Unix, or Windows
  – Distributed memory parallel (PVM)
  – Shared memory parallel (OpenMP)
• Source code for SYMPHONY solvers.
  - Generic MILP
  - Multicriteria MILP
  - Multicriteria Knapsack
  - Traveling Salesman Problem
  - Vehicle Routing Problem
  - Mixed Postman Problem
  - Set Partitioning Problem
  - Matching Problem
  - Network Routing
Download the latest packaged release from www.branchandcut.org.

Download from the COIN CVS server at www.coin-or.org.

- **Windows**: Use WinCvs to check out the SYMPHONY module from :
  pserver:anonymous@www.coin-or.org:/home/coin/coincvs
- **Unix/Linux**: 
  cvs -d :pserver:anonymous@www.coin-or.org:/home/coin/coincvs
  checkout SYMPHONY

Download a tarball from www.coin-or.org

- **Other software you may need**

  - Other COIN libraries (not required, but highly recommended):
    - **Osi**: To interface with SYMPHONY or the underlying LP solver.
    - **Clp**: To use as the underlying LP solver.
    - **Cgl**: To generate cutting planes,
    - **Coin**: To support above libraries (utilities).
    - **Win**: To compile COIN under Windows.
  - **GLPK**: To read GMPL files or use as the underlying LP solver.
  - Some other third-party LP solver.
Building

- If using COIN, unpack source in the COIN root directory.
- Otherwise, unpack it anywhere.

- **Unix/Linux**
  - Modify `SYMPHONY/Makefile`.
    * Set architecture.
    * Choose LP solver.
    * Set paths to other software packages.
  - Type `make` in the `SYMPHONY` directory.

- **Windows**
  - Using `nmake`
    * Modify the `SYMPHONY\WIN32\sym.mak` file as in Linux.
    * Type `nmake /f sym.mak` in the `SYMPHONY` directory.
  - Using MSVC++ 6.0
    * Open the workspace file `SYMPHONY\WIN32\symphony.dsw`.
    * Choose the LP solver and set the paths to other libraries.
    * Build the `symphony` project.
Using the Black Box Solver

• Read and solve a model in MPS format:
  – Linux/Unix: `bin.$(ARCH)/$(LP_SOLVER)/symphony -F sample.mps`
  – Windows: `WIN32\Debug\symphony.exe -F sample.mps`

• Read and solve a model in GMPL format:
  – Linux/Unix: `bin.$(ARCH)/$(LP_SOLVER)/symphony -F sample.mod -D sample.dat`
  – Windows: `WIN32\Debug\symphony.exe -F sample.mod -D sample.dat`

• SYMPHONY also has an interactive shell.

• SYMPHONY can also be used with FLOPC++, an open-source, object-oriented modeling environment similar to ILOG’s Concert Technology.

• Setting parameters
  – Command-line parameters are set Unix style (to get a list, invoke SYMPHONY with `-h`.
  – To set other parameters specify a parameter file with `-f par.par`.
  – The lines of the parameter file are pairs of keys and values.
  – Parameters are listed in the user’s manual.
Using the SYMPHONY Callable Library

• The SYMPHONY library is automatically built along with the executable.
  – Unix/Linux: Located in the `lib.$(ARCH)/$(LP_SOLVER)/` directory.
  – Windows: Located in the `WIN32\Debug` directory.

• Primary subroutines
  – `sym_open_environment()`
  – `sym_parse_command_line()`
  – `sym_load_problem()`
  – `sym_find_initial_bounds()`
  – `sym_solve()`
  – `sym_mc_solve()`
  – `sym_warm_solve()`
  – `sym_close_environment()`

• Auxiliary subroutines
  – Accessing and modifying problem data
  – Accessing and modifying parameters
  – User callbacks
Implementing a Basic MILP Solver with the C API

- Using the callable library, we only need a few lines to implement a solver.
- The file name and other parameters are specified on the command line.
- The code is exactly the same for all architectures, even parallel.
- Command line would be

  symphony -F model.mps

    int main(int argc, char **argv)
    {
      sym_environment *env = sym_open_environment();
      sym_parse_command_line(env, argc, argv);
      sym_load_problem(env);
      sym_solve(env);
      sym_close_environment(env);
    }
Using the OSI Interface

• The COIN-OR Open Solver Interface is a standard C++ class for accessing solvers for mathematical programs.
• Each solver has its own derived class that translates OSI calls into those of the solver’s library.
• For each method in OSI, SYMPHONY has a corresponding method.
• The OSI interface is implemented as wrapped C calls.
• The constructor calls `sym_open_environment()` and the destructor calls `sym_close_environment()`.
• The OSI `initialSolve()` method calls `sym_solve()`.
• The OSI `resolve()` method calls `sym_warm_solve()`.
• To use the SYMPHONY OSI interface, simply make the SYMPHONY OSI library.
Implementing a Basic MILP Solver with the OSI Interface

- Below is the implementation of a simple solver using the SYMPHONY OSI interface.
- Again, the code is the same for any configuration or architecture, sequential or parallel.

```c
int main(int argc, char **argv)
{
    OsiSymSolverInterface si;
    si.parseCommandLine(argc, argv);
    si.loadProblem();
    si.branchAndBound();
}
```
Using the SYMPHONY Framework to Develop a Custom Solver

- Advanced customization is performed using the user callback subroutines.
- There are more than 50 callbacks that can be implemented.
- The user can override SYMPHONY's default behavior in a variety of ways.
- Commonly used callback routines
  - user_initialize_root_node()
  - user_display_solution()
  - user_create_subproblem()
  - user_find_cuts()
  - user_is_feasible()
  - user_select_candidates()
  - user_compare_candidates()
  - user_generate_column()
  - user_logical_fixing()
Using the SYMPHONY Callbacks

- Function stubs for the callbacks are in the USER subdirectory.
- They are in files divided by functional module: Master/user_master.c, LP/user_lp.c, CutGen/user_cg.c, and CutPool/user_cp.c.
- Each callback returns either
  - USER_DEFAULT: Perform the default action (user did nothing)
  - USER_SUCCESS: User was successful in performing the function.
  - USER_ERROR: User encountered an error and could not perform the function.
- To use the callbacks, a new library is made including the callbacks.
  - Unix/Linux:
    * Type `make` in the USER subdirectory.
    * Executable will be `bin.$(ARCH)/$(LP_SOLVER)/symphony`
  - Windows:
    * Using `nmake`: Modify the USER\WIN32\user.mak file as before and type `nmake /f user.mak`.
    * Using MSVC++: Open the USER\WIN32\user.dsw file, modify settings as before, and build the user project.
**Example Callback Routine**

This code shows a custom solution display callback for a matching solver.

```c
int user_display_solution(void *user, double lpetol, int varnum,
                          int *indices, double *values,
                          double objval)
{
    user_problem *prob = (user_problem *) user;
    int index;

    for (index = 0; index < varnum; index++){
        if (values[index] > lpetol) {
            printf("%2d matched with %2d at cost %6d\n",
                   prob->match1[indices[index]],
                   prob->match2[indices[index]],
                   prob->cost[prob->match1[indices[index]]][prob->match2[indices[index]]]);
        }
    }
    return(USER_SUCCESS);
}
```
Warm Starts for MILP

• To allow resolving from a warm start, we have defined a SYMPHONY warm start structure, based on the CoinWarmStart class.

• The class stores a snapshot of the search tree, with node descriptions including:
  – lists of active cuts and variables,
  – branching information,
  – warm start information, and
  – current status (candidate, fathomed, etc.).

• The tree is stored in a compact form by storing the node descriptions as differences from the parent.

• Other auxiliary information is also stored, such as the current incumbent.

• A warm start can be saved at any time and then reloaded later.

• The warm starts can also be written to and read from disk.

• Has the same look and feel as warm starting for LP.
Warm Starting Code (Parameter Modification)

- The following example shows a simple use of warm starting to create a dynamic algorithm.

- Here, the warm start is automatically saved and reloaded.

```c
int main(int argc, char **argv)
{
    OsiSymSolverInterface si;
    si.parseCommandLine(argc, argv);
    si.loadProblem();
    si.setSymParam(OsiSymFindFirstFeasible, true);
    si.setSymParam(OsiSymSearchStrategy, DEPTH_FIRST_SEARCH);
    si.initialSolve();
    si.setSymParam(OsiSymFindFirstFeasible, false);
    si.setSymParam(OsiSymSearchStrategy, BEST_FIRST_SEARCH);
    si.resolve();
}
```
Warm Starting Code (Problem Modification)

- The following example shows how to warm start after problem modification.

```c
int main(int argc, char **argv)
{
    OsiSymSolverInterface si;
    CoinWarmStart ws;
    si.parseCommandLine(argc, argv);
    si.loadProblem();
    si.setSymParam(OsiSymNodeLimit, 100);
    si.initialSolve();
    ws = si.getWarmStart();
    si.resolve();
    si.setObjCoeff(0, 1);
    si.setObjCoeff(200, 150);
    si.setWarmStart(ws);
    si.resolve();
}
```
Example: Warm Starting

• Applying the code from the previous slide to the MIPLIB 3 problem p0201, we obtain the results below.

• Note that the warm start doesn’t reduce the number of nodes generated, but does reduce the solve time significantly.

<table>
<thead>
<tr>
<th></th>
<th>CPU Time</th>
<th>Tree Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate warm start</td>
<td>28</td>
<td>100</td>
</tr>
<tr>
<td>Solve orig problem (from warm start)</td>
<td>3</td>
<td>118</td>
</tr>
<tr>
<td>Solve mod problem (from scratch)</td>
<td>24</td>
<td>122</td>
</tr>
<tr>
<td>Solve mod problem (from warm start)</td>
<td>6</td>
<td>198</td>
</tr>
</tbody>
</table>
Bicriteria MILPs

• The general form of a bicriteria (pure) ILP is

\[
\text{vmax} [cx, dx],
\]

\[
\text{s.t. } Ax \leq b, \\
x \in \mathbb{Z}^n.
\]

• Solutions don't have single objective function values, but pairs of values called \textit{outcomes}.

• A feasible \( \hat{x} \) is called \textit{efficient} if there is no feasible \( \bar{x} \) such that \( c\bar{x} \geq c\hat{x} \) and \( d\bar{x} \geq d\hat{x} \), with at least one inequality strict.

• The outcome corresponding to an efficient solution is called \textit{Pareto}.

• The goal of a bicriteria ILP is to enumerate Pareto outcomes.
**Example: Bicriteria ILP**

- Consider the following bicriteria ILP:

  \[
  \begin{align*}
  \text{vmax} & \quad [8x_1, x_2] \\
  \text{s.t.} & \quad 7x_1 + x_2 \leq 56 \\
  & \quad 28x_1 + 9x_2 \leq 252 \\
  & \quad 3x_1 + 7x_2 \leq 105 \\
  & \quad x_1, x_2 \geq 0
  \end{align*}
  \]

- The following code solves this model.

```c
int main(int argc, char **argv)
{
  OsiSymSolverInterface si;
  si.parseCommandLine(argc, argv);
  si.loadProblem();
  si.setObj2Coeff(1, 1);
  si.multiCriteriaBranchAndBound();
}
```
Example: Pareto Outcomes for Example

Non-dominated Solutions

Y2

Y1
Example: Bicriteria Solver

By examining the supported solutions and break points, we can easily determine $p(\theta)$, the optimal solution to the ILP with objective $8x_1 + \theta$.

<table>
<thead>
<tr>
<th>$\theta$ range</th>
<th>$p(\theta)$</th>
<th>$x_1^*$</th>
<th>$x_2^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(−\infty, 1.333)$</td>
<td>64</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>$(1.333, 2.667)$</td>
<td>$56 + 6\theta$</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>$(2.667, 8.000)$</td>
<td>$40 + 12\theta$</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>$(8.000, 16.000)$</td>
<td>$32 + 13\theta$</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>$(16.000, \infty)$</td>
<td>$15\theta$</td>
<td>0</td>
<td>15</td>
</tr>
</tbody>
</table>
Example: Graph of Price Function
Other Sensitivity Analysis

- SYMPHONY will calculate bounds after changing the objective or right-hand side vectors.

```c
int main(int argc, char **argv)
{
    OsiSymSolverInterface si;
    si.parseCommandLine(argc, argv);
    si.loadProblem();
    si.setSymParam(OsiSymSensitivityAnalysis, true);
    si.initialSolve();
    int ind[2];
    double val[2];
    ind[0] = 4; val[0] = 7000;
    ind[1] = 7; val[1] = 6000;
    lb = si.getLbForNewRhs(2, ind, val);
    lb = si.getUbForNewRhs(2, ind, val);
}
```
Conclusion

• This has been a brief introduction to the capabilities of SYMPHONY.

• SYMPHONY can also be used in parallel, but this functionality has not been tested recently.

• We are currently in the process of further developing SYMPHONY’s warm start and sensitivity analysis capabilities.

• We are also working on preprocessing and better primal heuristics.

• Please check www.branchandcut.org for future developments.

• Questions?